Project no. IST-033576

XtreemOS

Integrated Project
BUILDING AND PROMOTING A LINUX-BASED OPERATING SYSTEM TO SUPPORT VIRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS

Specification of Federation Resource Management Mechanisms D2.2.1

Due date of deliverable: November 30th, 2006
Actual submission date: December 20th, 2006

Start date of project: June 1st 2006

Type: Deliverable
WP number: WP2.2
Task number: T2.2.1

Responsible institution: INRIA
Editor & and editor's address: Christine Morin
IRISA/INRIA
Campus de Beaulieu
35042 RENNES Cedex
France

Version 1.0 / Last edited by Christine Morin / December 20th, 2006

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.1	16/10/06	David Margery	INRIA	Initial template
0.2	23/10/06	Christine Morin	INRIA	Detailed Outline
0.3	26/10/06	Christine Morin	INRIA	Early draft of the introduction
0.4	2/11/06	Erich Focht	NEC	Outsourced NEC contribs to separate files
0.5	2/11/06	Adrien Lebre	INRIA	KerFS audit - Early draft
0.6	3/11/06	Erich Focht	NEC	Restructured core file
0.7	3/11/06	Thomas Ropars	INRIA	basic reconfiguration mechanisms - First draft
0.8	5/11/06	Erich Focht	NEC	Specs on porting
0.9	6/11/06	Erich Focht	NEC	Specs on scalability
0.10	7/11/06	Erich Focht	NEC	Added specs for pushing to kernel
0.11	7/11/06	Adrien Lebre	INRIA	Change document structure (article -> report)
0.12	9/11/06	Christine Morin	INRIA	First draft of Kerrighed overview
0.13	14/11/06	Erich Focht	NEC	First draft of VO integration
0.14	15/11/06	Christine Morin	INRIA	Revised outline for Kerrighed audit section and initial uncomplete draft of conclusion
0.15	15/11/06	Oscar D. Sanchez	INRIA	Some minor changes
0.16	16/11/06	Michael Schoettner	UDUS	Updated table 5.1
0.17	16/11/06	Christine Morin	INRIA	Tuning of Kerrighed Audit section - more material on containers - audit section outsourced in a separate file - executive summary
0.18	17/11/06	Adrien Lebre	INRIA	Minor structural changes in executive summary and conlusion parts - formatting text
0.19	30/11/06	Adrien Lebre	INRIA	scheduler kerrighed audit and structural changes, formatting text
1.0	20/12/06	Christine Morin	INRIA	final version - figure Kerrighed updated

Abstract

The XtreemOS operating system is intended to be executed on all computers in a Grid, making their resources available for use as part of virtual organizations. There will be three XtreemOS flavours, one for each kind of Grid node: individual computers (typically for PCs), clusters, and mobile devices.

As described in the "Description of Work" document [13], the XtreemOS operating system is composed of two parts: XtreemOS foundation called XtreemOS-F and XtreemOS high level operating system services called XtreemOS-G. XtreemOS-F is the modified Linux system embedding VO support mechanisms and providing an appropriate interface to implement XtreemOS-G services. XtreemOS-G is implemented on top of XtreemOS-F at user level, or more accurately, at VO level, as users would then be supported at an even higher level.

On top of XtreemOS, a Grid application is executed on one or several Grid nodes. Such an application is composed of application units, an application unit being executed on a single Grid node. An application unit is composed of one or several processes or threads.

This document relates to LinuxSSI-XOS specifications, the SSI cluster version of XtreemOS-F. In a cluster all nodes work closely together so that in many respects they can be viewed as though they were a single computer. A Linux SSI operating system provides the illusion that a cluster is a virtual multiprocessor machine executing Linux. For XtreemOS-G services, a cluster executing LinuxSSI-XOS will be seen as a powerful PC executing Linux-XOS. Thus, LinuxSSI-XOS is a standard Linux kernel modified in two ways: firstly to incorporate the modifications related to the VO support (the same modifications as those used to build Linux-XOS and described in [15]) and secondly to integrate distributed resource management services to provide a single system image (that is to say modifications related to LinuxSSI).

This document results from the joint work carried out in the framework of the WP2.2 work package by INRIA, NEC, XLAB, Düsseldorf University, ICT and SAP and from discussions with other partners.

LinuxSSI will be based on the existing Kerrighed SSI technology developed by INRIA in cooperation with EDF [16, 19, 22, 25, 28]. Indeed, Kerrighed is a sound basis to build the LinuxSSI component of XtreemOS as many SSI features are already implemented. However, it is not stable enough at the time of writing neither to start the implementation of the basic LinuxSSI features nor to allow proper execution of XtreemOS use cases. Thus, the implementation work in WP2.2 will start with a debugging phase with no addition of new functionalities to significantly improve the stability of the current version of Kerrighed software. During this phase, we plan to directly contribute to the Kerrighed community that shares with XtreemOS consortium the objective of a better stability of the existing

functionalities. We will report bugs and submit bug fixes to the Kerrighed community. This will allow XtreemOS consortium to take advantage of the work done towards a better stability of Kerrighed by key developers of the Kerrighed system[1]. Hence, we hope to reach a satisfactory state by the end of the first quarter 2007. The debugging work will be carried out along with the design of LinuxSSI basic functionalities.

In contrast to the current Kerrighed version, LinuxSSI should support SMP cluster nodes and x86-64 bit processors. This point is considered as the highest priority work that we plan to perform at the end of the stabilization phase. The hardcoded parameters currently limiting the scalability of Kerrighed will be removed in LinuxSSI and we plan to evaluate potential additional algorithmic limitations to Kerrighed scalability.

Checkpoint/restart mechanisms implemented in Kerrighed are neither complete nor reliable. By M18, LinuxSSI should provide an appropriate support to checkpoint parallel application units executing on a cluster whatever their communication model (shared memory or message passing). We propose a three level architecture for the checkpointer service including a kernel checkpointer able to checkpoint the state of individual processes, a system checkpointer taking care of establishing checkpoints for application units and a Grid checkpointer interacting with the system checkpointer for checkpointing applications that may span multiple Grid nodes. A mechanism to checkpoint/recover the state of containers to be developed constitutes one of the core components of the kernel checkpointer.

Reconfiguration mechanisms allowing a clean node shutdown and incremental boot of a LinuxSSI cluster will be implemented on the basis of the recently released HotPlug module of Kerrighed that is not yet stable at all. We aim at tolerating single node failures in LinuxSSI by M18. However, the reconfigurability of LinuxSSI-FS will only be studied after M18.

Concerning the implementation of high performance disk I/O we will leverage KerFS distributed file system for the design and implementation of LinuxSSI-FS. We will focus on two main aspects. First, we will improve KerFS stability to be able to use LinuxSSI-FS as the root file system. Second, we will target efficiency implementing customizable striping mechanisms and I/O scheduling strategies.

About the customizable scheduler of processes, we will focus on two complementary components: the load balancing scheduler and the long-term scheduler, which is being an optional to an LinuxSSI. The load balancing scheduler will be capable of accepting probing (monitor), analyzer and optimization function plugins, which will make the whole architecture highly customizable, and capable of accepting various scheduling policies. Regarding the long-term scheduler we plan to support DRMAA standard, which will allow us to use any existing batch scheduler supporting this standard.

From a more general point of view, we do not expect to have all LinuxSSI patches quickly accepted in the Linux community as they are too numerous and

[1]We emphasize that the Kerrighed Key developers do not belong to the XtreemOS consortium

they are interleaved. We will work on revisiting Kerrighed patches to minimize them when possible in LinuxSSI and to isolate some subparts of LinuxSSI such as containers to better push them into the Linux community. Getting LinuxSSI patches accepted in the mainline Linux kernel is one of our key objectives but it requires long-term efforts and careful design of LinuxSSI basic functionalities.

Some desirable advanced functionalities have already been identified but will only be implemented in the second half of the XtreemOS project, after M18. In particular, it would be interesting in large clusters to support high speed networks such as Infiniband. Indeed, supporting specific drivers for these networks rather than relying on the generic NetDevice Linux driver will improve the LinuxSSI performances.

Concerning Checkpoint/restart, it may be interesting to study different checkpointing strategies for parallel application units. The work performed on application unit checkpointing on SSI clusters in WP2.2 will be coordinated with the work performed on checkpointing application units on individual PC in WP2.1 even if we do not expect to have fully compatible mechanisms for both kinds of Grid nodes (individual PC running Linux and clusters running LinuxSSI). We will also coordinate our work in WP 2.2 with that of WP3.3. The application management service developed in WP3.3 will indeed use the checkpoint/restart mechanisms provided by LinuxSSI in the framework of a Grid level checkpointers taking into account every unit of an application spanning multiple Grid nodes.

Another advanced functionality is the fault tolerance support in LinuxSSI-FS to allow reconfigurability of the file system and to offer an efficient support to the system checkpointer for saving the state of the open files of an application when it is checkpointed. We plan interact with WP3.4 which is in charge of the design of the XtreemFS Grid data management service in our work on LinuxSSI-FS as LinuxSSI clusters may be client or server of XtreemFS. We will also further investigate how containers could be used in the framework of the GOM service developed in WP3.4.

For the customizable scheduler most important advanced functionality implemented is the adaptive feedback loop, which will allow adaptation of the scheduling policies and algorithms based on the system state. Additionally innovative load balancing policies exploiting features specific to SSI clusters will be studied.

Before M18, we do not plan to integrate virtual organization and security mechanisms in LinuxSSI basic version as these mechanisms will be developed concurrently with the design and development of LinuxSSI. We plan to carry out the integration work once the basic version of the various components of WP2.1 and WP3.5 and of LinuxSSI are released.

Contents

Chapter 1

Introduction

The XtreemOS operating system is intended to be executed on all computers in a Grid, making their resources available for use as part of virtual organizations. There will be three XtreemOS flavours, one for each kind of Grid node: individual computers (typically for PCs), clusters and mobile devices.

As described in the "Description of Work" document [13], the XtreemOS operating system is composed of two parts: XtreemOS foundation called XtreemOS-F and XtreemOS high level operating system services called XtreemOS-G. XtreemOS-F is the modified Linux system embedding VO support mechanisms and providing an appropriate interface to implement XtreemOS-G services. XtreemOS-G is implemented on top of XtreemOS-F at user level, or more accurately, at VO level, as users would then be supported at an even higher level. XtreemOS-G comprises of services for security, data and application management, all based on a common infrastructure for highly available and scalable services.

On top of XtreemOS, a Grid application is executed on one or several Grid nodes. Such an application is composed of application units, an application unit being executed on a single Grid node. An application unit is composed of one or several processes or threads.

This document relates to LinuxSSI-XOS specifications, the SSI cluster version of XtreemOS-F. In a cluster all nodes work closely together so that in many respects they can be viewed as though they were a single computer. A Linux SSI operating system provides the illusion that a cluster is a virtual multiprocessor machine executing Linux. For XtreemOS-G services, a cluster executing LinuxSSI-XOS will be seen as a powerful PC executing Linux-XOS. Thus, LinuxSSI-XOS is a standard Linux kernel modified in two ways: firstly to incorporate the modifications related to the VO support (the same modifications as those used to build Linux-XOS and described in [15]) and secondly to integrate distributed resource management services to provide a single system image (that is to say modifications related to LinuxSSI).

This document results from the joint work carried out in the framework of the WP2.2 workpackage by INRIA, NEC, XLAB, Düsseldorf University, ICT and SAP and from discussion with other partners.

LinuxSSI will be based on the existing Kerrighed SSI technology developed by INRIA in cooperation with EDF [16, 19, 22, 25, 28, 16].

The deliverable is organized as follows. First, we present in Section 2 a summary of application requirements related to the cluster flavour of XtreemOS that derive from from the work performed in WP4.2 [12]. LinuxSSI leveraging Kerrighed technology, we have started our work in WP2.2 by an audit of Kerrighed system. In Section 3, we present an overview of Kerrighed system and the results of the audit regarding the functionnalities offered in the most recent version of Kerrighed, the level of stability of the current Kerrighed prototype and the integration of Kerrighed patches in Linux kernel. Considering the application requirements and the current state of Kerrighed system, we present in Section 4 the specification of LinuxSSI. In Section 5, we discuss LinuxSSI development strategy. Section 6 concludes summarizing the current state of Kerrighed and stating our priorities in LinuxSSI implementation plan.

Chapter 2

Application Requirements

In deliverable D4.2.1, a range of requirements from the applications' point of view are defined. In the following, we summarize the requirements specifically addressing federation management. However, as work packages are highly inter-related, it is essential to examine the relevance of all other requirements in D4.2.1.

Node properties constraints in federations: It should be possible to specify some required properties of federation nodes: 1) node architecture (homogeneous, heterogeneous), 2) installed libraries, 3) installed web services and other software. Most applications depend on some libraries and other software. Furthermore, some applications would require additional effort to be able to distribute parallel execution among heterogeneous nodes. Although XtreemOS could optionally offer facilities to overcome some of or all the above difficulties automatically (which is not a subject of this requirement), such facilities would incur a performance penalty. *Mechanisms/suggestions*: The XtreemOS API should provide means to specify the constraints when starting the application.

Number of federation nodes used: It must be possible to specify the number of federation nodes to use because the number of nodes that can be used effectively is application-specific and thus cannot be determined by the system. Almost all applications require a variable number of nodes. Typically, the maximum number of nodes required is in the range between 100 and 1000 nodes within a federation though it must be considered that various applications could use as many nodes as they can get.

Changing number of federation nodes: It must be possible to change the number of nodes that the application uses during runtime. If the number of available federation nodes changes, XtreemOS must notify the running applications. The application then decides whether it can adapt to the change. *Mechanisms/suggestions:* If the application can adapt to the change, it is its responsibility to rearrange any variables and computations going on. If an application cannot adapt on-the-fly to fewer nodes being available, perhaps it can be checkpointed and restarted on fewer nodes. The notification mechanism can be decided on later. It must also be possible that the running application requests a change of the number of federation nodes. A running application can request additional nodes to start processes. These additional resources have to be provided by XtreemOS (if nodes are available). Furthermore, a running application may release certain resources after terminating calculations on these nodes. These nodes are then available for the execution of other applications. XtreemOS must be able to dynamically consider these released nodes in resource management and to provide them to other applications.

Specification of service qualities in federations: It must be possible to specify service qualities (e.g. maximum network delay, availability of resources, throughput) for a certain application. Resources that do not fulfill the defined service qualities must not be used by the application. XtreemOS also allows applications to specify a topology of the resources which provide the best performance.

Shared file system within a federation: XtreemOS must offer a shared file system within federations.

Checkpointing and restart: Automatic failure detection, checkpointing and restart must also be supported on federation nodes. The respective requirements are equivalent to those for Linux-XOS (please refer to D4.2.1).

Virtual nodes in federations: It must be possible to replicate processes on multiple federation nodes to increase robustness in case of resource failures, similarly to the grid-level virtual nodes mechanism. (see also R4). *Mechanisms/suggestions:* The application specifies which processes are critical and therefore require replication. The number of replicas can be specified by the application as well. Alternatively, a mechanism could allow specifying the desired robustness, after which the system would choose a suitable number of replicas, taking into account the estimated robustness of each node.

These requirements are adressed by the following document, except for the last one. Indeed process replication is out of the scope of WP2.2 as it would require extensive work on high availability mechanisms at the process level.

WP2.1 and WP2.2 will provide checkpoint/restart mechanisms required in case of node failure but cannot provide active duplication between related processes to enable a process to survive node failure whithout using a checkpoint/restart approach.

Some mechanisms could be provided at a higher level in XtreemOS. WP3.2 addresses this question for processes that are grid services and WP3.3 could address them for processes that are duplication aware.

Chapter 3

Kerrighed Audit

In this section, we give an overview of Kerrighed Linux-based single system image operating system for clusters. LinuxSSI leverages Kerrighed targeting clusters that are part of a Grid exploited with XtreemOS Grid operating system. We describe the main components of Kerrighed and evaluate their current state and how far the current version of Kerrighed fulfills the application requirements.

Kerrighed is an open source software distributed under the GPL licence. The source code repository is available in the Kerrighed project in the INRIA Gforge site [20]. The audit has been conducted from August to October 2006 with the latest version of Kerrighed based on Linux 2.6.11, available in the trunk branch in Kerrighed Gforge project.

3.1 Overview of Kerrighed

Kerrighed is a single system image operating system for high performance computing on clusters. It gives the illusion that the cluster is a multiprocessor machine. Based on Linux and implemented at kernel level, Kerrighed offers a Posix compliant interface. Hence legacy applications can be executed without any modification or recompilation on top of Kerrighed. Kerrighed implements global and dynamic resource management by a set of distributed services. Figure 3.1 shows the software architecture of Kerrighed. In the remainder of this section, we briefly describe each service.

Three layers can be distinguished in Kerrighed software: the communication layer implementing a high performance communication system providing a kernel level interface, the basic blocks layer implementing Kerrighed main concepts, and the distributed service layer implementing cluster-wide traditional high level operating system services.

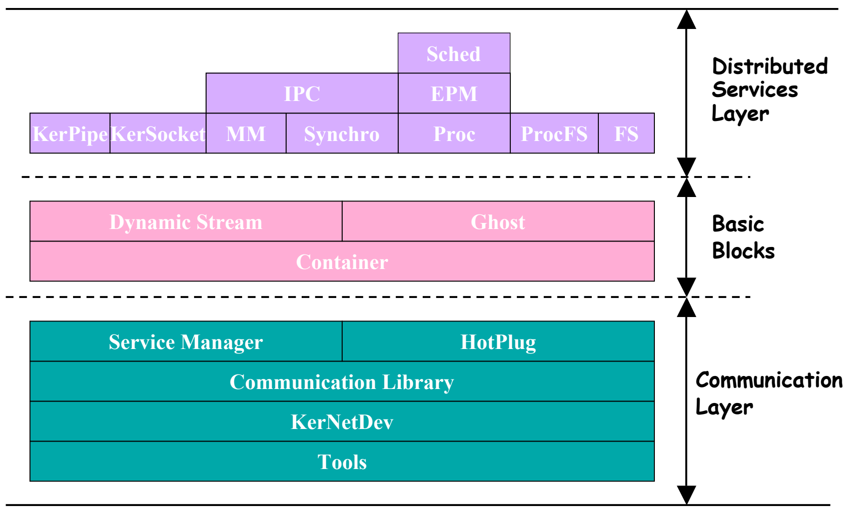

Figure 3.1: Kerrighed software architecture

3.1.1 Communication Layer

The communication layer comprises five modules. The Tools module contains basic tools used by other Kerrighed modules: debugging infrastructure, management of data structures, hash tables... The KerNetDev module is the lowest level in the communication stack in Kerrighed. It implements Kerrighed network driver interface providing access to the network interface cards through low level send/receive functions operating on packets. In particular, it supports the Netdevice generic network driver of the vanilla Linux kernel. Thus, Kerrighed does not depend on any specific networking technology as most networks provide a Netdevice interface. However, for the sake of performance in clusters based on SAN (Myrinet clusters for instance), it is recommended to use a KerNetDev module customized for the particular SAN technology that is used.

The Communication library module implements Kerrighed communication engine which targets high performance communication (low latency and high throughput). It provides at kernel level a high level interface used by other Kerrighed modules for their communications. The Communication library implements communication protocols such as for instance active messages. It is independent from the networking technology.

The Service Manager module implements a RPC interface on top of the communication library.

The HotPlug module is in charge of dealing with reconfigurations in the cluster. A reconfiguration event may be a node addition, removal or failure. This module manages node identifiers and maintains a map of correct nodes. It implements a heartbeat mechanism to detect failures. When a cluster reconfiguration has to be performed (after a failure detection or triggered from the user space by commands issued by the system administrator), the HotPlug module triggers a reconfiguration of containers. When containers have been dealt with, it notifies other Kerrighed services of the reconfiguration to allow them to undertake service specific actions.

3.1.2 Basic Blocks Layer

Kerrighed is built around three main concepts: containers, dynamic streams and ghost processes. These are supported by in the Basic Blocks Layer including three modules.

Containers [22, 23], implemented in the Container module, allow consistent data sharing cluster wide. Containers are used by many Kerrighed services to share meta-data between nodes. They are also used to implement page sharing. A container is associated with each system object to be shared: memory segment, IPC segment, file, meta-data, ...). A container is linked to high level services of the operating system (virtual memory, file system, ...) through interface linkers and to devices storing data (memory, disk) through I/O linkers. Linkers are not implemented in the container module.

The Dynamic Stream module [16, 17] implements dynamic data streams. A dynamic data stream is a data stream whose extremities can be transparently migrated in the cluster. In Kerrighed, traditional communication interfaces such as Unix and Inet sockets, pipes and FIFO are preserved while being implemented on top of the dynamic data stream mechanisms. Dynamic data streams rely on Kerrighed high performance communication system for data transfers within the cluster. Thanks to dynamic streams, communicating processes can be transparently migrated in the cluster keeping the highest possible performance. Direct communication between processes are indeed guaranteed despite process migration.

The Ghost module [28, 29] implements the importation and exportation of Linux kernel data through the ghost mechanism. The ghost mechanism offers a generic interface for kernel data storage whatever the device: memory, disk or network. The ghost mechanism is used as a basic building block for process duplication, migration, remote creation, checkpointing and restart.

3.1.3 Distributed Service Layer

The upper services of Kerrighed implementing the Kerrighed system call API constitute the distributed service layer that comprises ten modules.

Communication Interfaces for Applications

Two modules, KerPipe and KerSocket, implement the standard Posix communication interface on top of the Dynamic Stream module. KerPipe provides the standard pipe interface for processes whatever their location in the cluster. KerSocket implements the traditional socket interface.

IPC Management

Three modules are involved in IPC management. The IPC module provides the standard IPC interface cluster-wide: shared memory segments (system V segments), semaphores, message queues. It provides unique names for IPC objects and implements them in a distributed way to allow any process to use the IPC interface whatever its location in the cluster. The IPC module relies on the MM module for shared memory management and on the Synchro module for the distributed synchronization management.

The MM module implements on top of containers shared memory segments that can be accessed from any cluster node. Shared memory segments are for instance used to support threads (shared address space) and to migrate a process address space. Based on containers, the MM module implements the memory I/O linker to manage the memory device in a distributed way and a memory interface linker to link the virtual memory management service of Linux to containers.

The Synchro module implements in a distributed way synchronization mechanisms (such as locks and barriers) allowing to synchronize multiple threads or processes whatever their respective execution node.

Process Management

Three modules are involved in process management: Sched, EPM (Enhanced Process Management) and Proc.

The Proc module implements process management cluster-wide. It deals with global process identifiers, signaling, maintaining links between parent and child processes and process termination.

The EPM module implements advanced process management features such as process migration, duplication, checkpoint and restart. It relies on the Ghost module functionalities for dealing with the importation and exportation of the process context and on other modules such as container and dynamic stream modules for managing the process address space and IPC.

The Sched module is in charge of load balancing between cluster nodes.

File Management

The FS module implements the KerFS distributed file system. This file system manages the disks attached to cluster nodes as a large virtual disk. Files stored in KerFS can be accessed from any cluster node. KerFS relies on containers for file data sharing and consistency. Containers are also used for global management of i-nodes and open file descriptors in KerFS.

Global /proc Management

The ProcFS module implements a global /proc directory for the whole cluster, providing a global vision of the cluster resources.

3.2 Audit of Kerrighed Basic Features

During this section we report on the status of some basic features of Kerrighed. The subsequent sections deal with the audit for the services under the scope of WP2.2 tasks relating to the design and implementation of LinuxSSI:

- T2.2.2: building scalable SSI mechanims (see Section 3.3),
- T2.2.3: checkpoint/restart mechanisms (see Section 3.4),
- T2.2.4: reconfiguration mechanisms (see Section 3.5),
- T2.2.5: high performance disk input/output (see Section 3.6),
- T2.2.6: customizable scheduler (see Section 3.7).

The audit has been done on the Kerrighed version based on Linux 2.6.11 kernel. This version of Kerrighed is still under development in the Kerrighed community.

3.2.1 Communication System

Now, the KerNetDev only supports the NetDevice generic driver of Linux. The communication library offers the send/receive, pack/unpack and active message interfaces. These interfaces being intensively used by other modules, they are stable.

The communication engine currently implements four protocols providing a kernel level interface:

- The *acknowledgement* protocol dedicated to the packet acknowledgement mechanism,
- The *kernel* protocol dedicated to the kernel to kernel communications,
- The *user* protocol dedicated to the user-space communication support,
- The *hotplug* protocol dedicated to the node addition/removal/failure notifications.

The RPC interface provided by the Service Manager module is also stable and intensively used by other modules. However, the Kerrighed community plans to remove the Service Manager module and to integrate the RPC interface support in the Communication Library module (preserving the current interface).

3.2.2 Dynamic Streams

Concerning communications at user level, the *Dynamic Stream* module that has been rewritten on top of containers is not stable. The *KerSocket* module is also not stable and will only provide in the short term a support for TCP *inet* sockets. The KerPipe module that was implemented for the Kerrighed version based on Linux 2.4 kernel has not been maintained in the new version of Kerrighed based on Linux 2.6.11.

Today, Inet sockets (TCP, UDP), Unix socket, pipe and FIFO are not supported on top of the *Dynamic Stream* module. The standard Linux communication stack can be used by applications running on top of Kerrighed. However, the communicating processes can neither be migrated nor checkpointed.

3.2.3 Containers

Kerrighed implements the concept of container as a unique set of mechanisms to globally manage the cluster physical memory. All operating system services using memory pages access the physical memory through containers.

In a cluster, each node executes its own operating system kernel, which can be roughly divided into two parts: (1) system services and (2) device managers. Kerrighed implements a generic service inserted between the system services and the device managers layers called *container* [23]. 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 container gives the illusion to system services that the cluster physical memory is shared as in an SMP machine.

Container A container is a software object that allows the cluster-wide storing and sharing of data. 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, be used as a file cache entry, etc.

By integrating this generic sharing mechanism within the host 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 the cluster traditional services offered by a standard operating system (see figure 3.2). This allows to keep the OS interface, as 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. This model is the one offered by a physically shared memory.

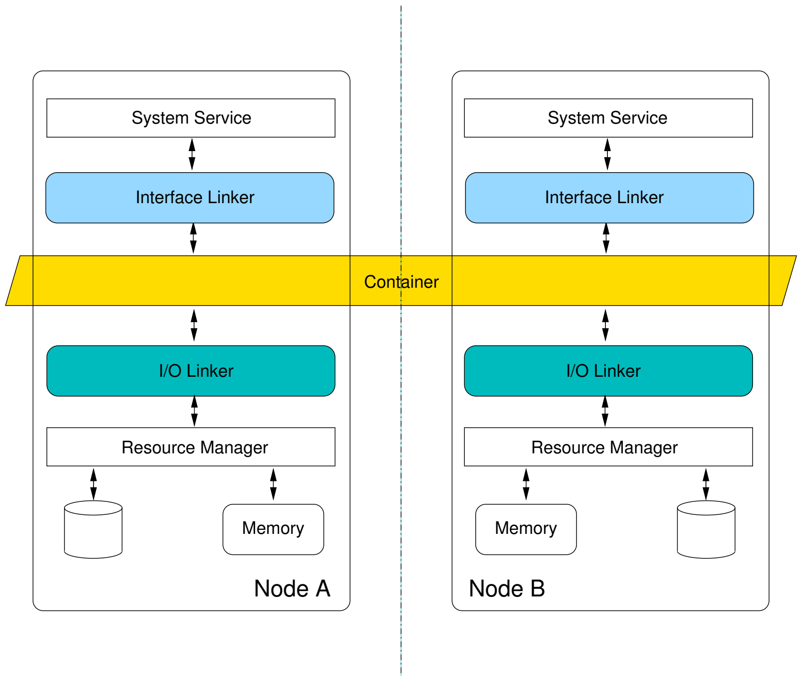

Figure 3.2: Container Architecture

Linkers Many mechanisms in a 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 an I/O linker allows a container to access a device manager.

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. For instance it is possible to map a container in the address space of a process P1 on a node A and to access it thanks to a read/write interface within a process P2 on a node B.

During the creation of a new container, an input/output linker is associated to it. The container then stops being a generic object to become an object sharing data coming from the device it is linked to. The container is said to have been instanciated. 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, it is completely empty, i.e. 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 it is destroyed or in order to release page frames when the physical memory of the cluster is saturated.

The *Container* module, which is heavily used during the functioning of Kerrighed, is certainly the most stable module in Kerrighed. It is possible to manage any kind of data in containers. The I/O linker interface is stable.

3.2.4 Process Management

The EPM module currently supports process migration, remote fork.

Kerrighed implements the following shell command to migrate a given process (providing its pid) to a given node (providing its nodeid). This command can be called from any cluster node.

```
migrate <pid> <nodeid>
```

Process migration relies on the following modules:

- *Ghost* to write the process state to a ghost,
- *Container* to get information on the parent and children processes, to migrate the process memory space on demand, to deal with open files,
- *Dynamic Stream* to migrate open streams extremities

Migrating a process with open KerFS files is not supported currently (KerFS is not used by default in the Linux 2.6 based Kerrighed version). When a process with open files is migrated, the open files are simply re-opened on the target node if they exist there.

Moreover, migration of dynamic stream extremities is not supported (see Section 3.2.2).

The remote fork functionality is implemented relying on the same mechanisms as the process migration. Only regular fork is supported currently. Three interfaces are provided:

- shell command (*krg_rsh* or *krg_capset -d +DISTANT_FORK*),
- User level C function (using a regular fork provided that the *DISTANT_FORK* capability has been enabled using *krg_capset*),
- Kernel level interface (*do_fork* with the *DISTANT_FORK* capability set).

The process checkpoint/restart functionalities have to be finalized. Some particular points such as restarting a process from a "scratch" node required some debug phases to be completely stable.

To execute an EPM operation, it must be ensured that the user-level registers are available and that there is no execution in a kernel code path. To fulfill these requirements, EPM operations are executed by sending an internal signal to the target process. The operation to be executed for this process is described in a dedicated field of the process task-struct. These mechanisms work well.

3.2.5 Memory Management

The *MM* module seems to be stable. However, it has not yet been heavily used in the current version of Kerrighed as the multithreading support relying on its functionalities is currently inactivated. System V segments are fully supported in the current version of Kerrighed with a satisfactory level of stability.

Moreover, in the current version, there is no advanced swapping mechanism allowing to swap out pages into remote memories rather than into the local disk when the physical memory of a node is full. Such a functionality would allow to take advantage of the cluster architecture to enhance the performance of applications dealing with very large data sets. Note that the lack of this advanced functionality is not blocking for experimentating XtreemOS use cases on top of Kerrighed.

3.2.6 Multithreading and Synchronization

The multithreading support is inactivated in the current version of Kerrighed based on Linux 2.6.

The *Sync* module is stable but not used as the multithreading support using it is currently inactivated. It is important to note that this module has not been rewritten from the Kerrighed version based on Linux 2.4.29. In particular, it does not rely on containers for sharing cluster-wide data structures used for managing synchronization objects. This module will have to be rewritten to take advantage of containers. Exploiting containers will simplify the code and even more importantly will considerably facilitate the management of synchronization objects in the event of cluster reconfigurations.

Currently synchronization objects that are natively supported in Kerrighed are the following: locks, barriers, semaphores, condition variables. The IPC semaphore interface has not yet been implemented on top of the *Sync* module.

3.3 Scalability/SMP Support

The scalability audit has been conducted during August - September 2006, the SVN revision of the kerrighed code being $\leq r600$.

The following potential scalability issues have been identified:

- SMP: No support for shared memory parallel nodes.
- 64 bit support: No support for 64bit CPUs.

- Hardcoded parameters: Certain parameters and dimensions are hardcoded and will lead to problems when increasing the number of nodes.
- Dynamic reconfigurability: Dynamic node addition and eviction not supported.
- Functional scalability: Potential problems due to non-scalable code and algorithms.
- High speed interconnects: Only TCP/IP stack is currently supported. Native support for more advanced high speed interconnects is required.

3.3.1 SMP

The motivation for SMP support is twofold: (1) the CPU development is currently aiming at multi-core processors, and (2) most server platforms have at least two CPU sockets. Therefore the single CPU machines will pretty soon become very rare.

Kerrighed was not coded with SMP support in mind. Currently the kernel and the kerrighed.ko module do not build when CONFIG_SMP is enabled. The reason for this is missing implementations of certain functions in the SMP path.

Because Kerrighed was coded with a uni-processor (UP) platform in mind, the protection and access to all shared data-structures as well as the logic of RPC calls must be re-evaluated and checked for SMP design problems like missing locks and potential race conditions.

3.3.2 64 bit Support

With 64 bit processors like AMD Opteron and Intel Nocona and Intel Woodcrest (Xeon 5100) becoming the main deployed server processors, the support for the x86_64 CPU architecture is mandatory. This is correlated with a growing demand for bigger memory spaces seen in today's applications, which can only be covered by the 64 bit virtual address space of 64 bit processors.

The 64 bit port is currently in work by Arkadiusz Danilecki from Poznan University. During compilation the code shows many warnings of integers treated like pointers, but the kernel has booted and very basic functionality like global process view was working.

3.3.3 Hardcoded Parameters

Kerrighed has several hardcoded parameters which will lead to scalability limitations for bigger clusters. The currently identified ones are:

- Number of Kerrighed nodes: this number is limited to seven bits, which leads to an upper limit of 128 nodes in a Kerrighed cluster. The limitation is visible for example when dealing with global process IDs.

- Node addressing map: the node addressing is currently done with a static MAC address map.
- Hashtables: The hashtables are used for accessing several variables like container pointers, RPC callback function pointers, etc. The size of a hashtable is fixed at compile time. This, for example, limits the global number of containers in the cluster currently to 1024.

3.3.4 Dynamic Reconfigurability

At the time of the audit Kerrighed could only run with a number of nodes which was specified at boot time. Each booting node received the number of nodes in the cluster and its own ID on the kernel boot command line. The failure of one node lead to a lockup in the entire cluster. A clean shutdown of the cluster was not possible, this leads to filesystem corruption.

Meanwhile the reconfigurability is included into Kerrighed through the Hot-Plug component. The system can tolerate the failure of one node at a time. A clean shutdown is not yet possible (svn r800).

3.3.5 Functional Scalability

Until now Kerrighed experiments have been carried out on relatively small clusters (less than 32 nodes). Scalability issues due to non-scalable algorithms were not significant. A quick look into the code showed that such problems are around and might become significant. Example: the broadcast of a message to all cluster nodes is coded serially, the handling of the */proc* global process view is ineffective and might lead to slowing down significantly a big cluster.

It will be useful to prepare a benchmarking methodology in order to localize and quantify such performance and scalability problems.

3.3.6 High Speed Interconnects

Currently Kerrighed uses exclusively the **netdevice** interface for communication. This is working well and reliably with Ethernet devices was also tested on Myrinet a while ago. No native support for Infiniband, Myrinet or Quadrics is available. For big HPC clusters of hundreds or thousands of nodes the support for some high speed interconnect is desirable.

3.4 Checkpoint/Restart Mechanisms

3.4.1 Principles

Checkpoint/Restart mechanisms in Kerrighed are based on the ghost mechanism used for process migration, thread creation and distant fork capabilities. This mechanism takes a snapshot of a process and sends it to disk or to distant memory as well as reads a snapshot from disk or memory to start a process. Depending on the cases, the started process will keep its process identifier (migration or restart) or change process identifier (distant fork).

Checkpoint/restart in Kerrighed is therefore deeply integrated in the overall design. Indeed, ghost only handle process context and contain references to memory areas which are stored in containers. The references to open network connections, implemented over dynamic streams and to filenames are handled by the different implementation managing file access such as the File Access Forwarding (FAF) layer or KerFS if applicable.

3.4.2 Level of implementation

In its current implementation, the system checkpoints Kerrighed applications.

In the context of XtreemOS, applications are composed of application units running on different Grid nodes. An application unit is then defined a collection of processes under the control of one operating system instance (*ie* a Grid node), either Linux-SSI or Linux-XOS. These processes could be multi-threaded.

In the context of Kerrighed, applications are a means to identify subtrees of the process tree in order to group processes that are part of the same computations. As a first approximation, Kerrighed applications are XtreemOS' application units. In the rest of this section, the term application refers to Kerrighed's definition.

By default, processes do not belong to any application and are therefore not checkpointable. If a process (*ie* a shell) sets the CAP_CHECKPOINTABLE capability in its inheritable effective capabilities, all processes started (`fork`ed) from it will become the head process of an application. By default, all processes started within the framework of an application belong to that application. Using this mechanism, Kerrighed provides a natural means of grouping processes into applications, and then checkpointing whole applications cluster-wide.

Snapshots of application are stored either in memory or in the */var/chkpt/* directory of the node the chekpoint is taken on. For each application, a directory whose name is the application id is created. Inside this directory, for each process of an application, the following files are created

- global_v*sequence number*.bin, to store application information, *ie* the nodes hosting processes of the application, as well user information about who is running the application.
- node_*node_id*_v*sequence number*.bin, to store information about the processes running on each node
- task_*pid*_v*sequence number*.bin, to store the context of process of identifier *pid*
- task_mm_*pid*_v*sequence number*.bin to store memory areas of a particular process

Memory area checkpointing is being rewritten at the time of writing. The Kerrighed community expects to have a working implementation of checkpointing for a sequential process soon, but complete debugging of checkpointing for applications composed of more than one process is not a high priority for them.

3.4.3 API

The current API for Kerrighed is based on pseudo-system calls implemented through `ioctl` calls. The checkpoint system call has the following parameters:

app_id the application that should be checkpointed. Kerrighed checks the credentials of the caller before checkpointing

chkpt_sn checkpoint sequence number: unused at the time being

type interpret app_id as a process identifier or as an application identifier

media the media the checkpoint should be stored on. Current supported media are disk and memory. No distant media are supported

3.5 Reconfiguration Mechanisms

The HotPlug module is the latest Kerrighed module. It has been released end of October 2006. This module is currently neither stable nor efficient.

It provides the following interface to the system administrators:

- *krgadm nodes add -n x,y,z* for the hot addition of nodes x, y and z.
- *krgadm nodes del -n x,y,z* for the hot removal of nodes x, y, and z.

The nodes in the list provided in the *krg adm* command are added or removed one after the other, leading to poor performance. A node failure occuring during the execution of the *krgadm nodes add* command is not supported.

The *krgadm nodes del* function should be used to properly shutdown the Kerrighed system. However, experiences we carried out show that the shutdown of a node may leave other nodes in an erroneous state.

Kerrighed implements an internal mechanism based on hooks allowing its own services to provide call-back functions to be called by the HotPlug module in the event of a reconfiguration event. This mechanism will evolve in the future.

Node failures are well supported by some Kerrighed services such as the container service. However, Kerrighed distributed file system, KerFS, in its current state does not tolerate any node failure.

3.6 High Performance Disk I/O

To federate disk resources, the Kerrighed infrastructure includes a specific module, entitled KerFS. This module in coordination with the container layer (cf. Section 3.2.3) provides a distributed file system tailored to the Kerrighed system.
Even if available solutions such as PVFS, parallel version of NFS or Lustre provide some ways to exploit distributed storage resources, they do not fulfill the expected requirements by the Kerrighed community. Keeping in mind that the Kerrighed file system should cooperate with or benefit from other Kerrighed services, the KerFS module has been designed.

The main objectives were:

- Federate all disk resources and provide a unique name-space,
- Enable customizable storage policies for files and directories (distributed, fully/partially redundant, . . .), such policies could improve both efficiency and fault tolerance,
- Be fully transparent to data storage location,
- Exploit a cooperative distributed buffer cache for efficiency,
- Support process checkpoint/migration at file system level,
- Support hot node addition/removal.

First, we give an overview of the KerFS architecture implemented in Kerrighed version 1.0.2 and evaluated by the Kerrighed staff under Linux 2.4.29. Even if the code has been ported to Linux 2.6.11 it has not been really tested. Second, we deal with the state of the implementation. The current position of the Kerrighed community on KerFS has changed. In the last part, we present their roadmap for the next months.

3.6.1 KerFS Architecture

The KerFS relies on two main concepts: using the native file system available on each node to facilitate the meta-data management and coordinate all accesses by the containers mechanism. The containers maintain consistency and provide a global distributed cache. Most of the complexity is located in that part. Figure 3.3 illustrates the architecture.

Three types of container are used :

- File containers, one per physically different file or directory. This container corresponds to the cache of a file.
- Inode container, one clusterwide. The stored information is some significant values of the `inode struct`.

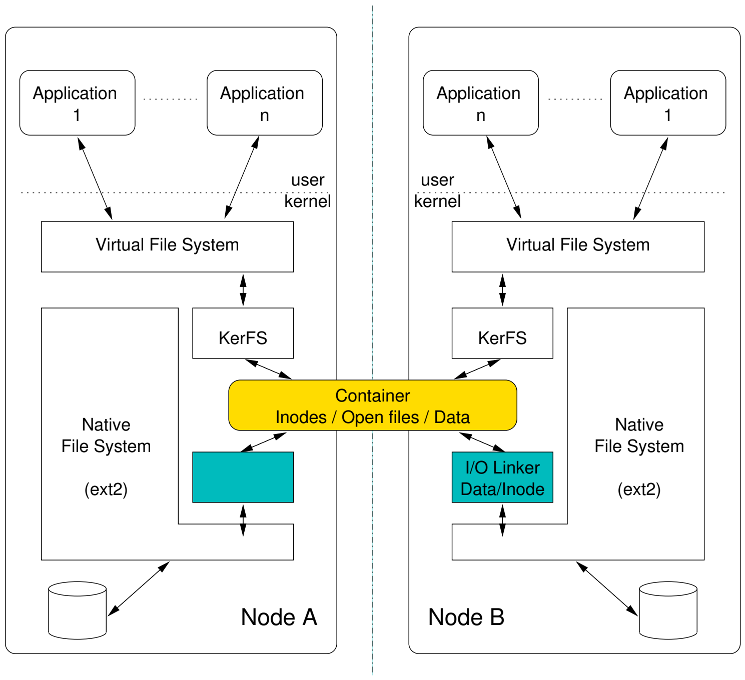

Figure 3.3: Overview of the KerFS system

- Open files information container, one cluster wide.
  It stores information about the interaction between an open file and a process. This information usually contained in kernel memory during the period when a process has the file open is maintained inside the file container. Now, only the file position is stored.

But for open file information container which is associated with a traditional memory linker (cf. Section 3.1.2), the inode and the file containers exploit specific linkers: the inode I/O linker and the file I/O linker. The former has been designed to insert/extract and maintain meta-data information from the local inode copy. The latter one enables to read/write data from/to the local file system to/from the local page cache and then put/remove the pages to/from the associated file container. It also maintains the associate local cache to invalidate copies when it is required.

3.6.2 Current State of the Implementation

The latest implementation of KerFS (Linux 2.4.29) provides:

- A unique name-space cluster wide,
- A fully redundant storage for significant data (all directories),
- A distributed storage based on a file granularity for ”temporary data” (each file created by applications is stored locally),
- Transparency to data storage location (POSIX compliant),

- A cooperative distributed file cache (similar to the Linux page cache but only for the KerFS file system),
- A cooperative distributed inode cache for KerFS files and directories.

Even if the code has been ported to Linux 2.6.11, this implementation is just a proof-of-concept and needs to be rewritten to a production version. For instance, the KerFS could and should not be used as the root FS (memory leaks, unsafe cache write-behind policy).
Currently, all nodes have a ext2/ext3 root partition where all main directories, binaries and configurations files are stored. The KerFS is mounted after the boot via a traditional `mount` call (all KerFS data are stored in /.KERFS_ROOT on each node). Then, all directories/files written inside a KerFS mount point will take advantages of the KerFS mechanisms.

Finally, the following features are still not implemented: customized storage policy, fault tolerant mechanisms (find the right I/O linker between replicas), file checkpointing and the hotplug features.

3.6.3 A Major Change, Distributed *VFS*

The use of KerFS is mandatory for fully exploiting all the Kerrighed SSI features. This seriously limits the use of common distributed/parallel file systems such as NFS, PVFS or Lustre. As a consequence, the Kerrighed community has temporarily stopped its implementation around the KerFS solution to focus on a new architecture for file management. Roughly, the idea consists in developing a distributed *Virtual File System.* This solution will enable to exploit all Kerrighed features (such as processes migration) whatever the mounted file system. Moreover to provide a completely global name-space, the distributed cache mechanisms provided only in the KerFS system will be available for all file systems.

From the Kerrighed community, this approach will contribute to the portability of the Kerrighed system. From the XtreemOS requirements, it does not directly feet our needs. For instance, it neither takes into account the federation of available hard drives in the cluster nor provides particular storage data policy. Those features should be implemented in another system. These points should be taken in consideration for the design and the implementation of the Linux SSI file systems.

Finally, the Distributed *VFS* is a kind of ”concept”. The Kerrighed community has not yet started to design its architecture. They plan to start working on it in February/March 2007.

3.7 Scheduler

The current version of the Sched module implements a single load balancing policy which is very close to the one implemented in Mosix system [10]. A load processor probe is activated at each clock tick. Its goal consists in computing the local processor load based on the average number of task ready to run on the cpu (size of the run queue). Moreover, in order to select the right task to migrate, the load induced by each local task is also computed every second. Each node stores this information in a common container.

On each node, the load balancing mechanism is activated if the load of a CPU is higher than a threshold and the last activation was at least two seconds sooner. Finally, a process will not be migrated if the expected load of the target node will be higher than the expected local load.

The Kerrighed version based on Linux 2.6 kernel does not provide any mean to customize the global scheduling policy. However, a modular framework for customizing the load balancing policy in Kerrighed is currently being implemented in the Kerrighed community. This framework is inspired from [28] and based on the use of sysfs mechanisms for hot changing the global scheduling policy without stopping the execution of the applications.

3.8 Conformance to Linux Kernel Programming

Coding style

The Linux kernel developer community is managing a huge amount of code and respects strictly the coding style conventions which define indentation rules, naming rules, reasonable function splitting rules. They are all based on common sense and aim at making kernel code easily readable, understandable and maintainable. Linux kernel code originates from thousands of developers, therefore a coherent coding style is absolutely necessary for maintaining a clear structure in the code.

The reasons for a clean coding style apply even stronger for Kerrighed and XtreemOS: we ultimately need to keep the code produced by a large IP project clearly structured and maintainable and want to get it accepted (at least partly) into the Linux kernel mainline.

Kerrighed followed its own coding style which was incompatible with the Linux kernel coding style until svn r644, when the official Linux coding style was adopted. Since then most of the indentation discrepancies were fixed.

The variable naming in Kerrighed often reflects the fact that it has been programmed by different generations of coders over a long period of time. Some variable and function names are very long and some contain unnecessary information. Example: `notify_migration_start_to_analyzer`.

Hooks and Patches

The Kerrighed components are currently very strongly depending on each other, i.e. meaningful functionality can only be obtained when a large number of interdepending components are enabled. This means that the minimum number of components which would need to be pushed towards mainline kernel is very large and covers almost the entire Kerrighed code. In addition the Kerrighed components are tightly bound into the Linux kernel code by a large number of hooks and patches to the mainline code. These hooks and patches spread over more than 100 main kernel files.

3.9 Stability

At the time of writing, the 1.0.2 stable version of Kerrighed based on Linux 2.4.29 is obsolete. A new release of Kerrighed based on Linux 2.6.11 is expected to be released by the Kerrighed community by end 2006. The current version of Kerrighed based on Linux 2.6.11 (extracted from the svn repository) is not stable enough to allow the execution of XtreemOS use cases. Significant efforts will have to be devoded to the enhancement of the stability of the current version before implementing new functionalities.

Chapter 4

LinuxSSI Specification

We present in this section the specification of LinuxSSI XtreemOS-F component for clusters. Basic functionalities are those that we plan to design and implement by M18 (November 2007). We also give initial thoughts about advanced functionalities that we plan to develop during the second half of the project (after M18).

4.1 Overview

Five main directions of work have been identified in XtreemOS "Description of Work" document [13] for the design and implementation of LinuxSSI:

- Building scalable SSI mechanisms,
- Design and implementation of checkpoint/restart mechanisms,
- Design and implementation of reconfiguration mechanisms,
- Design and implementation of high performance disk input/output operations in a cluster,
- Design and implementation of a customizable scheduler.

We detail the specifications of LinuxSSI along these five directions in the remainder of this section.

In addition to LinuxSSI core features, we have identified two other work directions in WP2.2. First, there is the need to port LinuxSSI patches and modules to new Linux kernels. This work is essential to get a chance to have some of LinuxSSI kernel changes accepted in the mainstream Linux development. Pushing LinuxSSI patches to Linux kernel is indeed another important activity that will be carried out in WP2.2. These aspects of our future work relates to LinuxSSI development strategy. They are further discussed in Chapter 5.

4.2 Scalable SSI Mechanisms

The targets set by XtreemOS-SSI scalability requirements are correlated to general HPC cluster technology development. During the project's lifetime top performance clusters will reach petaflops range, therefore XtreemOS-SSI needs to aim at supporting clusters with hundreds or even thousands of nodes. The component nodes should be able to use several multicore 64 bit processors. The nodes will be interconnected by specialized high speed interconnects. With this large number of components the mean time between failures will increase dramatically and XtreemOS-SSI should be able to handle such failures gracefully by dynamically reconfiguring the SSI cluster in case of failures or intentional node addition or eviction.

As discussed in section 3.3 scalability touches several aspects of Kerrighed: SMP support, 64 bit support, algorithm choice and implementation, dynamic reconfigurability and high speed interconnects. For the first 18 months of the project the targets in decreasing order of priority are:

- Dynamic reconfigurability: survive single node failure, dynamic node addition and eviction. Allow clean shutdown of the SSI cluster. Very high priority.
- SMP nodes support: Add support for shared memory parallel nodes and multicore CPUs. Very high priority.
- 64 bit processor support: add support for x86_64 CPUs. High priority.
- Removing hardwired parameters: Get rid of limitation to 128 nodes and increase number of containers in the system. High priority.
- Support for high speed interconnects: add support for the OpenIB infiniband stack. Medium priority.
- Functional scalability: prepare benchmarking methodology for detecting issues in this area. Medium priority.

4.3 Checkpoint/Restart Mechanisms

Based on the state-of-the-art, application checkpointing in XtreemOS involves three levels of checkpointer. After the description of the general checkpointer stages the specific extensions for LinuxSSI will be presented.

1. The kernel checkpointer, providing basic functionality to take a snapshot of a single application unit
2. The system checkpointer, providing checkpoint management at the application unit level, *ie* automatic checkpointing and snapshot management using the basic functionnality offered by the kernel checkpointer

3. The Grid checkpointer, providing checkpoint facilities at the application level

The Kernel Checkpointer

The kernel checkpointer offers a very basic checkpoint interface that enables:

- Checkpointing of a single application unit,
- Notification to the checkpointed application that it is about to be checkpointed,
- Registration of callbacks from an application to tailor checkpointing to the application's needs,
- Enabling and disabling of checkpoint from the application if it is written in a checkpoint aware way.

The callbacks are a means for the application unit to extend the boundaries of a checkpoint as made by the kernel level checkpointer.

The System Checkpointer

The system checkpointer is an OS service that manages checkpointing for an application unit. It is a configurable service, e.g. checkpointing time interval, garbage collection parameters, etc.

- It will use resources given to it to call the kernel checkpointer or request those resources on behalf of the calling process.
- It implements periodic checkpointing.
- It implements staged checkpoints.
- It implements checkpoint garbage collections.
- On LinuxSSI it manages checkpointing/restart for the cluster.

The Grid Checkpointer

The Grid checkpointer is the service responsible for supervision of checkpoints for an application: it applies the checkpointing strategy to all running application units.

- It registers the application units with the checkpointer service on the nodes running the application's application units.
- It provides resources to store the checkpoints.

- It detects node failure and takes appropriate mesures to restart the application. It must therefore manage the credentials of the user running the application to enable restart.
- It is able to launch applications in a checkpoint context.
- It coordinates the checkpoint of an application running on different nodes.

This Grid level checkpointer is the JobCheckpointing service described in WP3.3 The system level and kernel level checkpointers are described in more details in the next section.

4.3.1 Checkpointing in Linux-SSI

It will be based on the current Kerrighed implementation that provided rudimentary checkpointing for sequential application units.

During the first 18 months the first step will be to complete and stabilize the Kernel Checkpointer level checkpointing of applications. The major challenge will be to save and restore kernel/device states. To allow checkpointing of MPI programs open files and network connections (at least for connection between application units of an application) will be implemented. Furthermore, checkpointing of shared memory applications will be implemented, too (e.g. required by some XtreemOS applications). An elegant way to checkpoint different resources in Kerrighed is to checkpoint containers. The latter is a basic building block providing consistent data sharing in a cluster used to share I-Nodes, memory, process migration etc. Right now it is open if checkpointing a container is sufficient for all cases.

Specific extensions may be necessary for some resources. Application unit notification of checkpointing will be implemented in a similar way to the mechanisms used in WP2.1.

We expect that the System Checkpointer implemented as part of WP2.1 can be used as it is on Kerrighed, provided the Kernel Checkpointer of both WPs use the same interface as it is intended. Nevertheless, it will be extended to improve efficiency of checkpointing/restart mechanisms relying on specific Kerrighed features. Node failure detection is available in Kerrighed whithout the help of the Grid Checkpointer. As Kerrighed provides the illusion that a cluster appears as a single Grid node, it is natural that Kerrighed manages checkpointing and restarting for the total cluster. Therefore, the System Checkpointer of WP2.1 will be extended to manage distributed snapshots for a cluster including a cluster restart. The distributed snapshot management will include coordinated and independent checkpointing strategies. We expect synergies and code sharing with the Grid Checkpointer.

The System Checkpointer for LinuxSSI can of course not restart autonomously a cluster if one or several application units reside outside the cluster, e.g. on an outside PC or another cluster. In such a case the System Checkpointer will contact the Grid Checkpointer. But we expect that a lot of applications execute on one cluster, only, and benefit from an optimized System Checkpointer for LinuxSSI.

4.4 Reconfiguration Mechanisms

The aim of this section is to specify the support for reconfigurations in LinuxSSI. The module called *HotPlug* deals with reconfigurations in LinuxSSI. We present in a first part the reasons why LinuxSSI should allow reconfigurations in the cluster. Then, we expose what we consider as basic and advanced features and present them in two sections.

4.4.1 Introduction

Context

In a cluster, we can imagine three kinds of reconfigurations: (i) new nodes are added to the cluster; (ii) nodes are removed from the cluster; (iii) nodes fail. LinuxSSI has to adapt to these reconfigurations to ensure that they will not compromise the execution of the applications on the cluster. All clusterwide services must be able to continue to work despite the reconfiguration. That's why reconfiguration mechanisms need to be integrated in the LinuxSSI design.

For a better understanding of what we will have to do, it is interesting to see first why reconfigurations can happen in a cluster.

Node Addition: Nodes are added to the cluster by its system administrator. Nodes that are added can be new nodes added to improve cluster performance or nodes restarted after a failure.

Node Removal: Nodes are removed from a cluster by its system administrator. Nodes can be removed for upgrade (software or hardware). Other reasons can make the system administrator stop some nodes of the cluster, a too high temperature in the cluster for instance.

Node Failure: Some nodes of the cluster may fail but the cluster must continue to work.

Temporary Disconnections: Some nodes of the cluster may be temporary unavailable and thus considered as failed nodes. Conflicts can occur if those nodes have not really failed and come back in the LinuxSSI cluster.

We can assume that reconfigurations are not frequent in a cluster. So, the first goal that we have to achieve is not efficiency but correctness and robustness of the reconfiguration mechanisms. So, we can afford a degradated functioning mode of the whole cluster during the reconfiguration procedure. We make this assumption because LinuxSSI targets cluster of 1000 nodes at the most. So we can assume the mean time between failure will be about one per day rather than one per hour.

Basic Features and Advanced Features

Node addition and removal are treated as basic features. Handling node failures will be an advanced feature. It seems to be easier to handle node addition and removal because in this case we can inform the LinuxSSI services that a reconfiguration occurs and they can do the work they need before the reconfiguration occurs. For example, in case of the withdrawal of a node, processes running on this node can be migrated before removing the node. This cannot be done in case of node failure. That's why handling node failure is an advanced feature. Handling reconfigurations in LinuxSSI-FS is also an advanced feature. Finally, we will study the need of synchronization between services to perform adaptation to reconfigurations in the advanced feature as these synchronizations will probably concern LinuxSSI-FS.

HotPlug

HotPlug is the component of LinuxSSI that deals with cluster reconfigurations. HotPlug is the first component informed of the reconfigurations. It is also the one that implements failure detectors. It has to inform the other components of the reconfigurations and he coordinates the actions taken by the other components to adapt to the reconfiguration. It can also inform the applications of the cluster reconfigurations. If the cluster is part of a Grid, the Grid Resource Management Service has to be inform of the reconfigurations of the cluster. In this case, Hotplug has to notify the local part of the service in charge of publishing local resource information in the node directory service.

4.4.2 Basic Features

For the basic features, we present the specifications in a *chronological* order, i.e. the order in which things will happen in case of reconfiguration. First we explain how the administrator of the system can inform LinuxSSI of the reconfigurations. Then we study the work to be done in LinuxSSI to adapt to the reconfigurations. We also describe how we deal with temporary disconnections. Finally, we present how Hotplug communicates with the applications so that they can adapt to these reconfigurations. To have a global view of the problem, we finish by presenting the detailed procedure of nodes addition and nodes removal.

User Command

The administrator of the cluster is the one that can decide of node addition or removal in the cluster. He must be able to inform LinuxSSI, especially HotPlug, that he wants to remove nodes for instance. Thus LinuxSSI can perform an adaptation to ensure that the application will continue to run before allowing the administrator to remove the nodes. To do this, we need to provide commands to the administrator.

`ssi_add_nodes` `Node1,Nodes2...`
  add Node1,Node2... to the LinuxSSI cluster

`ssi_remove_nodes` `Node1,Nodes2...`
  remove Node1,Node2... from the LinuxSSI cluster

Both of these commands return when the action is performed or display an error message. When the `ssi_remove_nodes` command return, the removed nodes can be stopped.

Reconfigurations of LinuxSSI Components

We detail now the adaptation of LinuxSSI in case of node addition or node removal. We expose how components have to adapt to reconfigurations. We identify the role of HotPlug and its way to interact with other services.

Membership: HotPlug updates the membership on each node of LinuxSSI. For instance, the number of correct nodes and the list of these nodes must be updated on each node according to the cluster modification. We expect that the other services of LinuxSSI will automatically take into account modifications in the membership.

Containers: A container is a generic mechanism to share data cluster wide. Most of the distributed services are built on top of containers. Containers are the components that are mainly impacted by reconfigurations. In case of node removal, `Owner`, `probOwner` and `copyset` have to be updated for each object. If the nodes removed have the only copy of some objects, these objects have to be moved to other nodes. In case of node addition, the main work is to extend the containers to the new nodes. Adaptation of the containers will be coordinated by HotPlug. To ensure consistency, accesses to containers are forbidden during the adaptation procedure. We say that the cluster is in *adaptation mode*.

Distributed services: Distributed services are built on top of containers. As reconfigurations are handled by the containers, it will be transparent for distributed services. Services that need to have their own adaptation mechanisms, LinuxSSI-FS for instance, will be informed of the reconfigurations by HotPlug.

Processes: If processes are present on nodes that have to be removed. HotPlug inform the scheduler of the reconfiguration and the scheduler migrate the processes that can migrate. The processes that do not have the migration capability are killed.

Signaling Reconfigurations: HotPlug should signal the reconfigurations to the distributed services. The way to inform the services has to be further investigated, according to the need of synchronization between the services.

Temporary Disconnections

A node can be temporary disconnected from the cluster without having failed. If the duration of the disconnection is very short, the node is not suspected of failure and can come back transparently in the cluster without any problem. If the duration of the disconnection is longer, the node is detected as failed by the failure detector of Hotplug and a procedure to handle the failure start. To avoid conflicts, we decide that the node can not come back in the cluster unless an adding procedure is started.

Communication with User Space

Some applications can adapt to the number of nodes they have at their disposal. For example, they can create new processes if new nodes are available. We should provide them a way to be informed of reconfigurations in the cluster.

The solution we chose is to provide a way to register callback functions which are triggered when a reconfiguration event occur. Each application can register one callback function for node addition and one callback function for node removal. A parameter in the callback function gives the number of nodes implied in the reconfiguration.

Summary

To have a global view of how LinuxSSI will adapt to reconfiguration, we detail here the different steps in the procedure of nodes addition and nodes removal. Let's begin with node addition. In this example, we only want to add one node. It would be the same procedure with many nodes.

1. The administrator asks for node addition with `ssi_add_nodes`.
2. HotPlug receives the command and makes the cluster working in *adaptation mode*.
3. HotPlug starts the new node.
4. Containers are extended to the new node.
5. HotPlug actualizes the membership information on every node of the cluster.
6. The LinuxSSI cluster can restart working normally.
7. HotPlug activates the callbacks registered by the applications.

Now, we study the case of node removal. We want to remove one node.

1. The administrator asks for node removal with `ssi_remove_nodes`.
2. HotPlug receives the command and makes the cluster work in *adaptation mode*.
3. HotPlug asks the scheduler to migrate the processes executing on the node that will be removed. Processes that can not migrate are killed.
4. Objects on the removal candidate are flushed to other nodes.
5. HotPlug actualizes the membership information on every node of the cluster.
6. The LinuxSSI cluster can restart working normally.
7. HotPlug activates the callbacks registered by the applications.

4.4.3 Advanced Features

In this section, we briefly describe the advanced features we plan to add after M18 to improve the system.

Handling Node Failures: LinuxSSI should continue to work despite node failures. HotPlug will implement a failure detector and will have to bring back LinuxSSI to a safe state, checking lost objects for instance. We can imagine to develop duplication policies to avoid lost of data or to use checkpointing mechanisms to restore lost processes. An application management service could be in charge of applying fault tolerance policies.

Highly Available File System (c.f. 4.5.4): We have to investigate the mechanisms needed to allow nodes addition and removal. LinuxSSI-FS should be able to adapt the file system structure to the cluster reconfigurations, to exploit disks of nodes added to the cluster for instance.
To avoid lost of data in case of node failure, solutions based on replication can be used. Furthermore, to improve process checkpointing mechanisms, we are thinking of implementing a generic mechanism to be able to checkpoint opened files. Thus, when restarting a process from a checkpoint, we would also be able to restore the files. We will first study the ”checkpoint file” mechanism for LinuxSSI-FS. But our goal is to provide a generic mechanism that could work whatever the file-system type is.

Collaboration Between Distributed Services: Collaboration between services can be a solution to optimize adaptations of LinuxSSI to reconfigurations. We should study the possible interactions between distributed services. For example, a process is running on a node and is using one file present on the node. The node has to be removed. So, the scheduler has to migrate the process to another node. To choose this node, it can ask LinuxSSI-FS where the copies of the file are. Thus it can migrate the process to another node where the file is. If there is no copy of the file, it can be interesting for LinuxSSI-FS to know where the process will be migrated before choosing where to copy the file.

4.5 High Performance Disk I/O

There has been a lot of work done on distributed/network file servers [11] since the early eighties with various issues tackled: security, performances through caches, high availability, disconnected mode, consistency, ... In a cluster context, the most common approach consists in exploiting specific nodes to provide the distributed/parallel file system. If the existing systems offer many features, they strongly rely on the hypothesis that they are deployed on dedicated nodes. Thus, the cluster is divided into two groups: the compute nodes and the I/O nodes. The hard drives available on the compute nodes are only used for the system and temporary files, thus wasting both a lot of space (several TBytes on large clusters!) and throughput.

Such an approach does not meet our wishes defined in task D2.2.5 [13]:

- Aggregate storage resources,
- Provide an unique name-space,
- Provide efficient access for both small and large files.

In our view of an SSI cluster OS, all nodes could potentially provide both CPU and storage resources. Our proposal should be able to efficiently exploit most of the available storage and, in the meantime, take into account the resource usage of the applications (CPU, memory, network and hard drive). Such an approach in designing a file system for a Linux SSI should lead to several innovative works. Indeed, our proposal should obviously provide the common features of a distributed file system but moreover it should exploit, cooperate with and complete the SSI system itself by improving services and global performance.

This section introduces the specifications of our proposal. First, we list the requirements and constraints according to the other XtreemOS services. Second, we describe each functionality that we plan to offer.

4.5.1 Requirements and Constraints

This section lists the requirements defined in WP4.2 ([12], R4 of WP2.2) with regards to those from WP3.4. We also mention the constraints that we should keep in mind for the design and the development of our proposal.

R1: Provide a global name-space and federate all/several available hard drives (transparent data storage location, consistency).
The objective is to perform only one Linux installation per cluster with a common and global / root directory.

R2: Scalable to 256 nodes first and second to 1000.

R3: Efficient accesses for small and large files (customizable striping policies, I/O scheduler, cooperative cache, memory mapped files). In other words, it consists in providing high performance disk I/O in a cluster.

R4: Replication, fault tolerance, recovery and hotplug.
Two targets are identified: first provide required mechanisms to ensure node failures, and second, dynamically adapt the physical structure of the file system in case of node additions or removals.

R5: File checkpointing (or file snapshot).
The aim is to complete checkpoint services to solve open files issues which might appear during checkpointing phases.

R6: Integration with the D*VFS*
The purpose is to transparently ensure all nodes see the same mount tree.

In our context, we should consider the relations with, first, other Linux SSI services and second, with other XtreemOS work packages. For the moment, the points that should be investigated are:

- according to Linux SSI services:
  - Coordination with the SSI scheduler to improve efficiency and resources usage. For instance, the scheduler should consider the dependence/usage of the local hard drive before to migrate one process of a parallel application (this is a common case for an MPI I/O intensive application). It could be interesting to provide some details to the SSI scheduler or to exploit its knowledge to inform LinuxSSI-FS that some files need to be migrated.
  - Integration with the checkpointing mechanisms provided by Linux SSI and its evolutions. Roughly, it consists in keeping a history of the last changes in a file. Thus, when a process is restarted from the last checkpoint, all files which are associated with this process, reverse to their old version.
- according to XtreemFS (XtreemOS Grid File System, WP 3.4), we have to analyse the specifications which could have a direct impact on LinuxSSI and more precisely with data management. Indeed, we should take into account that the Linux SSI cluster could be both:
  - an XtreemFS client (so, we have to define how the Grid file system will be mounted),
  - an XtreemFS Object Storage Server (thus, keeping in mind that LinuxSSI could be exploited as a storage backend of XtreemFS).

If the relations with other LinuxSSI services are more advanced features, we have to consider as soon as possible the interconnection with XtreemFS as a client or as a server.

The results of our audit led us to position our work in the continuity of the KerFS solution. Indeed, the new "Distributed *VFS*" idea recently introduced by the Kerrighed community would need further works to fulfill our requirements. In contrast and even if the current implementation needs to be cleaned, several parts of the KerFS system could be directly reused in our solution (global name-space, location transparency,...). Moreover, the integration of our proposal with the D*VFS* would not require a great amount of work since our system will be seen like a traditional file system.

In the following part, we present our proposal: LinuxSSI-FS. To improve the readability, we choose to describe on a component basis. In each subsection, basic and advanced functionalities are addressed.
First, we deal with the fundamental mechanisms to federate all devices and provide a global and scalable name-space. Then, the mechanisms to provide high performance disk I/O are described. The third part deals with fault tolerance aspects and reconfiguration algorithms. Finally, we gives some details about additional features that might be design according to the achievement of the former ones.

4.5.2 LinuxSSI FS Foundations

This paragraph briefly explains how remote hard drives will be federated and how the global name-space will be achieved. These two points are the major ones since other file system services will rely on them. As it has been mentionned, we plan to directly reused the KerFS system since its current design is quite good and it would enable us to quickly have the bases of our proposal. The only limitation is that all nodes require to have the same directory tree, otherwise, some migration issues might appear (a process might loose files access after it has been migrated to a remote node).

Due to the KerFS design, the scalability of our proposal mainly depends on the container system one. We already know few limitations such as the maximum number of inodes (2^{32}, it corresponds to the highest number of objects that a container could record) or the maximum file size (16TBytes, 2^{32}*page size). During the first step of the XtreemOS project, we do not consider these issues as important. The only critical point lies in the maximal number of containers for the whole LinuxSSI, which is 1024. Since all Linux services share available containers, it means for the LinuxSSI-FS that less than 1000 files could be open at the same time. This point has been emphasized in Section 3.3 (scalability issue).

One of the general requirements for XtreemOS is to provide an environment where existing applications shall be able to run without any modifications (legacy code issue). Since our solution will be built as a traditional file system and thus located below the *VFS*, it should be compliant with common APIs. Thus, it will be possible to exploit all standard APIs such as the C library, *MPI I/O*, . . . directly. Concerning file mapping, it seems that the *VFS* is in charge of providing such a functionality (`mmap` call). So, it should not require any particular work to support it in the LinuxSSI-FS. However, it might be usefull to overload them if we need custom behaviour in our case. This point requires further investigations and will be addressed in the second half of the XtreemOS project. From a general point of view, all APIs will rely on the VFS capabilities in the first part of the project. According to the needs and to the state of our work, we will focus on particular optimizations.

The last but not the least point concerns the consistency issue. Thanks to the container mechanisms, our proposal will be conform to the UNIX one. No particular works is thus needed.

4.5.3 I/O Performance

This paragraph describes optimizations to provide an high performant I/O accesses. Indeed, even if the current implementation of KerFS already provides a cooperative cache (that needs to be stabilized), we should integrate in our proposal some approaches to efficiently exploit hard drives throughput. To achieve that, we plan to implement file striping mechanisms and I/O scheduling strategies. Both approaches are discussed in the following paragraphs.

Striping Policies

The objectives of the striping policies are twofold: first, to balance I/O requests over several nodes to reduce ”heavily loaded” points and second to benefit from aggregating throughputs provided by several hard drives. Indeed in a more conventional cluster, some dedicated nodes attached to RAID devices are exploited to deliver the expected bandwidth for each application. In an SSI cluster, each node should provide its storage support which corresponds in most of cases to one traditional hard drive with a throughput peak around 60MBytes. In such a case, striping policies[1] are mandatory.

We plan to provide two striping modes: the first is automatic and transparent whereas the second is based on users parameters.

- In the transparent mode, all data is stored locally. Two cases should be considered: sequential and parallel accesses.
  For the first one, only large files (out-of-core) could suffer from such an approach. A way to solve this issue is to strip a file on multiple hard drives when its size is bigger than a defined threshold.

[1]We distinguish striping policies to improve performances from the ones which deal with fault tolerance aspects addressed in Section 4.5.4.

For the second one, parallel accesses, the performance should be excellent. For instance, in MPI applications, each MPI instance will write its data locally according to the striping policy selected by the application (CYCLIC, BLOCK/BLOCK, . . .). From the file system point of view, there is no fixed striping size. This mode should improve the performance since it avoids all striping issues which may appear when the file striping does not correspond to the application one.
This mode is the default one and does not require to extend the POSIX API.

- In the second striping mode, each user will be able to define a specific striping policy on a per file or per directory basis. The striping policy is defined during the creation of the file/directory and cannot be changed (at least for the ”basic features”). Furthermore, the striping policy of a directory is by default propagated to its files. If the user wants to specify a distinct policy, he/she has to to define it at file creation.
  To associate a particular striping policy (number of chunk per stripe, striping size, . . .) with a file (or a directory), a specific API is needed. The specification of such routines is still under investigation. The common routines (POSIX calls and ubiquitous shell commands) require several extensions. For M18, we plan to provide RAID 0 and RAID 1. More sophisticated striping strategies might be considered during the second part of the XtreemOS project.

For both modes, all placement information for each file is stored within the associated LinuxSSI-FS meta-data. The striping geometry is based on an ”object” granularity (from 1 to n block). For instance, the meta-data lists all objects: a first object which is p blocks long is stored on the first drive of node x and a second one, q blocks long, is available on the drive of node y, . . . (with $p \neq q$).

I/O scheduling

The second important aspect concerns the throughput usage of each hard drive. To maximize it, we need to study the integration of our proposal with the available low level I/O schedulers and maybe complement them with newer ones.
Indeed, recent work showed the importance of such strategies in a multi-application cluster where several applications are executed concurrently, competing for accessing to storage subsystems. The I/O subsystem layer has to perform optimizations that take advantage of accesses regularity from each application while balancing storage access between them. Most of the available file systems do not use I/O scheduling strategies as they are just built on schedulers located in block device layer. At this low level, due to kernel and file system implementation applications information is not available and I/O access patterns cannot be exploited for throughput optimization.
Based on the aIOLi work [21], we plan to add I/O scheduling strategies to improve

as much as we can the deliverable throughput for each hard drive by maintaining a QoS for each application. We plan to add such mechanisms at the file I/O linker level (cf. Section 3.6). However, we need further investigations to confirm this choice. To be optimal, such an approach requires to have a global point of view of all I/O interactions incoming within the architecture.

4.5.4 Fault Tolerance and Reconfiguration

As the number of nodes composing a cluster increases, the probability of failure gets higher. Thus, LinuxSSI-FS should provide adequate mechanisms to ensure as much as possible files accesses in the event of resource failures. In the meantime and since our proposal is cluster wide, a particular service should handle node additions or removals to re-adapt the physical structure of LinuxSSI-FS. This paragraph deals with these two issues. The presented mechanisms are considered as advanced features and will be available in a future version, developed after M18.

Fault Tolerance

In an approach similar to striping strategies, LinuxSSI-FS should provide mechanisms to define replication policies. We plan to classify files and directories in two main categories: ”important files/directories” will be automatically replicated whereas ”user files/directories” will be replicated according to user settings. We think that such an approach will benefit to every application. First, important files (such as common executables/libraries or directories) will be still available whatever node failures. Second, each application will define its own fault tolerance strategy (such as no replica or n replicas based on a RAID1 approach). So, each user will be aware of the implied overhead. We want to emphasize that for one I/O intensive application, a RAID strategy could add an important overhead and, thus, we want to offer the possibility to the users to enable or disable such strategies.
In case of a failure, the deprecated I/O linker (cf. Section 3.6) will be dissociated from the containers and a new I/O linker will be created on one of the replica devices and associated with the different containers. In other words, when a failure occurs, the I/O linker associated with containers refers to a bad device (since the node or the device is unreachable). If there is at least one replica, we have to "reconnect" containers to a valid I/O linker otherwise data is simply lost. The location of replica devices is done thanks to the meta-data information. As for the striping approaches, the LinuxSSI-FS meta-data contains placement information and thus replica locations.

Finally and with regards to the implementation progress of the RAID 1 approach, a RAID 5 strategy might be considered. However, we stress the implementation of RAID 5 is much complex and requires a particular algorithm to rebuild the file before fixing the I/O linker issue. Due to the decreasing price of hard drives and the increasing number of nodes within a cluster, we think that RAID 1 should be sufficient.

File System Reconfiguration

This point consists in providing adequate algorithms to adapt the global file system structure to node additions and removals. This item will be developed in close co-operation with the reconfiguration mechanism task (cf. Section 4.4.3).
For a node addition, the hotplug service notifies LinuxSSI-FS that new resources are available (hard drive information such as the file system type, available partitions, ... is given). Thanks to these details, LinuxSSI-FS is able to exploit this new storage space and extend the physical structure of the file system. When all LinuxSSI services have been enabled on the new node and some applications are executed, LinuxSSI-FS stores data on the related hard drive (cf. Section 4.5.3).
For a node removal and in a similar way, the hotplug service informs LinuxSSI-FS. According to replication policies of the files stored on the node, LinuxSSI-FS replicates data and copies the meta-data repository (cf. Section 3.6) if needed. One issue remains: how handle files with ”no replicas” policy associated but currently exploited by some applications. In a naive approach, they are simply lost, in a more advanced, we can imagine specific interactions with the administrator. Indeed, It seems a bit tricky to discover if it is better to migrate a process and its attached files or to just restart the application from a distinct node. This last point needs further investigations in collaboration with T2.2.3, T2.2.4 and T2.2.6.

Even, if we plan to follow the roadmap of T2.2.4, we cannot assume that file system reconfiguration will be provided at M18. During this period, LinuxSSI-FS will only exploit non-resilent nodes.

4.5.5 Miscellaneous

In the first part of this last paragraph, we present two additional functionalities of our proposal: file checkpoint and file system sandbox. In the second part, we give some details about the potential integration of LinuxSSI-FS with the D*VFS*.

File Checkpoint

File checkpoint (or file Snapshot) is required to provide a complete checkpointing service. It consists in providing a way to reverse changes in a file between two process checkpoint. Such a feature is mandatory to restart a process in a consistent state from memory and file point of view.
Our idea is to extend the *VFS* API by adding some calls which enable to give specific states to files: ”normal” or ”checkpoint”. When the file is tagged as a checkpointable file, a copy-on-write policy is associated with it. Thus, all following writes accesses on it imply a copy of the old data in another shadow file. We choose to extend the POSIX API instead of just use the traditional `open` flags. So, it is possible to change the state of a file at any time. This could be usefull to checkpoint a whole application which does not plan to exploit checkpoint services (for instance, when an administrator make a manual checkpoint in case of a node removal).

When an application restarts from its last checkpoint, another VFS call enables to reverse the changes in the file and restore the right image. We could imagine several versions of such a checkpointing file: the most simple is based on one copy which is overwritten at each checkpoint and a more sophisticated might be an incremental checkpoint based on several shadow files. This functionality will be developed in coordination with the checkpointing task from work packages 2.1 and 2.2.

Integration with the D*VFS*

This last paragraph adresses the integration of our proposal with the D*(*VFS) approach suggested by the Kerrighed community (cf. Section 3.6.3). We point out that we do not plan to directly develop the D*VFS*. However, since the implementation of the D*VFS* is in the roadmap of the Kerrighed community, it might be available in a future version of Kerrighed. As a consequence, we have to prepare a potential integration between our proposal and the D*VFS* solution to take benefit from it. The major issues will consist in considering mechanisms directly provided by the D*VFS* (such as global caches, ...) and then disable them in our proposal.

To conclude on the high performance disk I/O usage, we emphasize that the advanced functionalities list is not fixed. It just corresponds to the different requirements that we already know and thus it might be revisited. Finally and from a general point of view, these last features require further investigation in the early stage of their design.

4.6 Customizable Scheduler

A need for an advanced adaptable customizable scheduler has been identified. Therefore a special task for building such scheduler has been introduced in the WP2.2. In this section we shall present the outline of the work on this task, where we shall first describe the motivation and design goals. This subsection is followed by an architectural view and definition of the interaction requirements with the Grid level services developed in WP3.3. The last subsection is dedicated to the implementation of the scheduler for a SSI Linux.

4.6.1 Scheduling Terms and Concepts

We define three object classes, which we shall take into account in the definition of schedule for a job. First is resource class (machines and tools in the scheduling theory). These are the objects that provide some feasibility to the system, whether this is a CPU time, dist storage, system software, or service. Second is task class. Task (operation in the scheduling theory), consumes resources in order to achieve a given goal. Third class is job class. A job consists of a set of least or more tasks, related resource requirements, and various constraints (i.e. finish time, cost ...). Between the tasks belonging to a same job there can exists a dependency relation, where a task can depend on other task in two ways. First is "a prerequisite" dependency, where a certain conditions have to be met in order for a task to start. A typical example of such scenarios is a use of output file as input data, or a use of common resource (i.e. same service or a global state of the application). Second "a concurrent" dependency, where two or more tasks have to start at the same time. A typical example is an MPI application. The task dependencies construct a directed acyclic graph, which can also be called a workflow \citeBrucker2004. Grid related terminology is nicely compiled in [27].

Summary of main object classes:

- Resource: *A resource is any physical or virtual (logical) component of limited availability within a computer system.*
- Task : *A task is an action which accomplishes a portion of a job.*
- Job: *A job is one or more tasks grouped together into a directed acyclic graph in order to achieve a certain goal or objective.*

4.6.2 XtreemOS Scheduler Overview

Design goals of the XtreemOS scheduler

There are several objectives of a Grid scheduling subsystem. Each owner of the resources strives for high **utilization** and **optimal resource usage**. By optimal resource usage we want to reduce the ratio between non productive resource usage (i.e. migration, checkpointing, replication…) and usage necessary to perform a job. A **predictable** system keeps a user satisfaction high. Even if the system loads are high, the user wants his job to be executed as it was declared on the job submission. One important component in order to have a predictable system is Service Level Agreement – SLA, which prescribes a contract based relation between the resource provider and the consumer. **Resistance to failures** is important feature in large distributed systems. In such systems we witness various failures, and high node churn. Services and protocols should take this fact into account, and still provide the users of the system the desired stable behavior.

Architecture and interaction with XtreemOS-G

The proposed architecture of the XtreemOS scheduling services consist of three levels. The highest level is Grid level – **jScheduler** service (definition is given in next paragraph), whose primary role is to find resources through the resource discovery process, and assign them to the tasks. The optimization goal at this level is **job optimization**, based on various optimization criteria. Once a schedule has been constructed, and the job optimization goals have been achieved, and SLA negotiated with **rAllocation** service (definition also is given in next paragraph), the tasks are put to the LTSchedulers – *long term schedulers* (local or Grid level).

The cluster level optimization goals are **optimal resource utilization**. On the level of the resources the goal is to reduce overhead as much as possible, which results in better **efficiency**.

The architecture is presented in figure 4.1. The jScheduler and rAllocation services are Grid level services. rAllocation service negotiates use of resources, and puts job in a LTScheduler queue. A job is submitted to execution to a cluster node, and after it started to run, a LBScheduler – *load balancing scheduler* is performing a load balancing in order to level the load on a cluster. Both LTScheduler and LB-Scheduler are executed on each site, and do not have any means of communication between different sites.

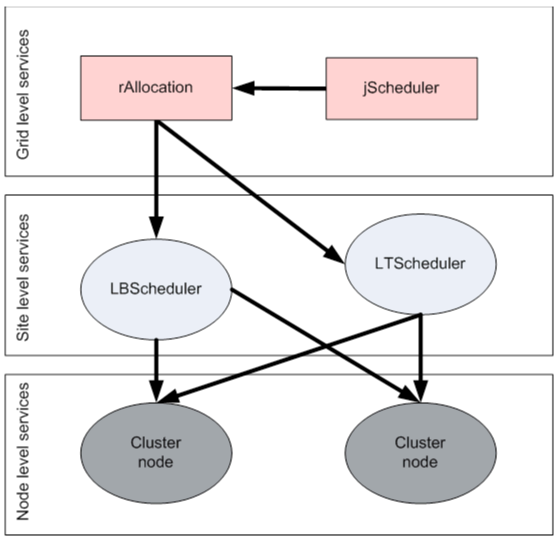

Figure 4.1: Scheduling architecture

Use of services defined outside WP2.2

The services defined in this workpackage depend on Grid level services, which will be devised in sub project – SP3. In the following text we shall give a short description of the services, on which the Scheduler relies, or requires communication with. **jControler** is a important service providing a fault tolerant access point to the job running on a Grid. This service relies on **jScheduler** service to provide schedule matching resources with this job tasks. The purpose of the **jResource-Matching** service is to provide resource availability information to the jScheduler. Next is **rAllocation** service, which handles the requests to allocate resources to a job and to execute a jobs' task under conditions that have been negotiated.

During the execution of a task there are two services, which are called from this level. The first one is the **rMonitor** service provide resource monitor functionality, and the second is **jEvent** service, which is responsible to distribute various events through the Grid [14].

4.6.3 SSI Scheduler

On an SSI system we can have two separate scheduling subsystems. First is a queuing scheduler – **long term scheduler – LTScheduler**, which is considered to be optional (longer explanation follows later on in the text), and the second is a **load balancing scheduler – LBScheduler**, which optimizes the overall system load. The detailed functionality of the LTScheduler module and relation to other Grid level services is described in [14], where in this text we shall concentrate on the parts, which are specific to SSI systems, and the interaction between the LTScheduler and the SSI-LTScheduler.

The XtreemOS Grid environment consist of various resources, where some of them already provide LTScheduler functionality (i.e. clusters with Maui installed), and some of them do not (i.e. ordinary Linux-XOS). Among the resources, which do not provide LTScheduler functionality are also SSI systems. The goal of this task is to provide services and interfaces, which will exploit the benefits of the SSI system to the full extend.

SSI-LTScheduler

The most important requirements regarding the long term scheduling of an SSI cluster is to be able to properly describe SSI cluster resource in order to achieve a high utilization. The LTScheduler will be aware of the specific advantages of such system and therefore be able to schedule the jobs/tasks, which are capable of taking advantage of the SSI capabilities. In the advanced version of the SSI-LTScheduler, the service will on the functionality provided by the SSI cluster (i.e. advanced migration, checkpoint and restart mechanisms). By using this functionality, the SSI-LTScheduler will be able to further optimize the resource usage.

API In order to support the described architecture, the SSI scheduler has to be able to act as a resource manager (job submission, monitoring and control). In order to provide an open and widely accepted interface, we have decided to adopt DRMAA [26, 1]. We assume that many of the available schedulers support DRMAA API or will do so in the future, which will allow XtreemOS to deploy an arbitrary scheduler such as MAUI [4], PBS Professional [9], or Grid Engine [2].

Basic Implementation The basic implementation of the SSI Scheduler will support DRMAA API for job submission, monitoring and control, which will enable using an SSI cluster without a resource manager (i.e. TORQUE).

Advanced Implementation In the advanced implementation we shall extend the resource description with SSI specific capabilities (i.e. Virtual Shared Memory / Virtual SMP), which will allow more efficient scheduling of the queuing scheduler. In order to fully exploit the benefits of the SSI cluster, we expect that we shall need to adapt / extend the algorithm used in the scheduling solution. SSI cluster provides functionality, which is used by scheduler (i.e. checkpointing). The advanced version of the LTScheduler will use the SSI specific implementation instead of usual one.

Innovations The most notable innovation in the LTScheduler is the capability to describe SSI specific capabilities (i.e. global shared memory). SSI cluster presents itself as a virtual SMP machine, which reflects in a **virtual resource boundaries.** SSI cluster is thus presented to the resource management services as a SMP machine, with several CPUs and a large virtually combined memory. If a process would require a great amount of memory, and no machine is available, that meet the requirements, the SSI system is still capable to host such process in conjunction with a process with requires only a small amount of memory. Based on this information, previously mentioned processes can be scheduled on adjacent machines with high speed interconnecting network, and still run near optimally.

Second innovation is called **resource sharing** and it will allow better resource utilization, and consequently open new possibilities for improving SSI scheduling. Today an LTScheduler is capable to schedule one task to a CPU. In a case this task requires only a fraction of the resource capabilities, this resource is significantly underutilized. We plan do investigate how a resource requirement description can be extended in order to allow simultaneous execution of more than one task on a resource, where the total execution time of the tasks is shorter than when executed in sequential manner.

SSI-LBScheduler

The function of the LBScheduler is to level the load between the nodes connected to an SSI cluster (a non-SSI cluster LBScheduler is not focus of this WP). The two most important parts of the scheduler are therefore the measurement of the resource state (i.e. CPU load, free memory, etc) and the decision function, which decides whether a process can be migrated to more suitable node. The work on the SSI-LBScheduler will therefore first enable easy development of its capabilities in the future, which will result in highly customizable scheduler. In second step – the advanced version will plan to provide capabilities to customize load balancing we shall also extend the architecture to allow self-adaptability based on the current state of the system.

In the next paragraph we will describe currently available solutions, and how the work on this workpackage goes beyond current capabilities of load balancing systems and algorithms.

Current SSI Scheduler implementations We examined four different SSI solutions regarding on what information is monitored in order to support the load balancing algorithm.

- MOSIX [5] is capable to reconcile different resources like bandwidth, memory and CPU cycles when performing load balancing decisions based on economic principles and competitive analysis [18].
- OpenMOSIX [6] computes the opportunity cost by summing a normalized CPU and memory load [24].
- OpenSSI [7] is using current CPU load only.
- Kerrighed [3] is using similar information as OpenMOSIX.

The so gathered information has to be dispatched to other nodes, and also here different approaches are taken. MOSIX for example select a set of random nodes, to which the load information is send, where on other hand on OpenSSI the information is gathered on one node - *clms master node* [8].

Basic Implementation In the basic implementation we shall focus on the interfaces to support customization. One of the most important services which will be implemented is **rMonitor** depicted in figure 4.2. The innovation of the service is to provide a hot-pluggable interface for a probing and analyzing capabilities of the Monitor. Each probe plugin is capable of measuring a specific resource property or state (i.e. CPU load, etc). The information retrieved from the probe shall be analyzed by the analyzer plugin, and the information forwarder to the jResource-Matching service. This capabilities will allow us to implement a highly customizable scheduler.

Advanced Implementation In the advanced version of the LBScheduler we will implement two innovations. First is the capability relies on the capabilities provided in the basic implementation, and allows optimization of the resource usage by extending the load balancing algorithm with a custom probing plugins providing the inter resource dependencies information (i.e. IPC, shared memory pages) and a custom evaluation function, which evaluates a gain or loss by a selection of specific resources. We extend the work described in [18] in the following way: the loadable modules provide at the run-time new measurement capabilities and evaluation functions to the decision making process, which reduces the need to foresee all the possible needs of the system in design time. In as a proof of concept we shall implement a set of plugins that will provide the possibility to evaluate the cost of placement or migration of more than one process based on a **shared memory measurement.** The functionality is best explained by example. On a SSI machine, two processes run on different nodes and use virtual shared memory capabilities of the system. The goal of the LBScheduler is to minimize the resource usage, and in order to retrieve a best possible process placement, it should retrieve information on shared memory requirements. This information is retrieved from the kernel through the probe plugin, forwarded to the evaluation function which output is send to the LBScheduler.

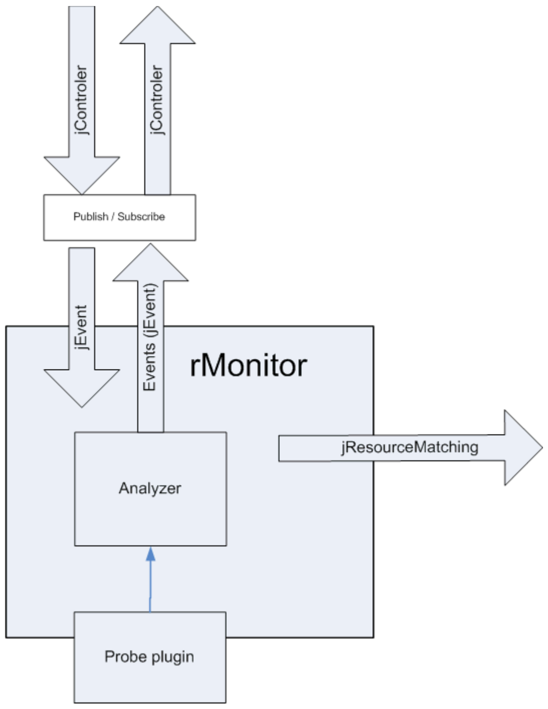

Figure 4.2: rMonitor Service.

Second innovation introduces **algorithm feedback loops** in the scheduling process, which results in self-adaptable capabilities of the system. With the adaptability of the system we shall achieve that the system will be capable of selecting the best possible algorithm to perform the load balancing based on the current situation. In example an algorithm which is reacts optimal under small system loads typically performs poorly under heavy loads, and with the information retrieved from the system we will be able to select best suitable algorithm in any given time. The architecture, which implements the innovation is presented in the figure 4.6.3.

In the figure we present two levels decision architecture. On a cluster level LBScheduler service provides a load balancing functionality, where on the Grid level jScheduler manages a load balancing requests. We have adaptable feedback loop on both cluster and Grid level, but for the purpose of understanding the concept, it is enough to explain one of them. The LBScheduler will be on base of the information received from the rMonitoring service able to determine, if the currently selected load balancing algorithm is optimal for performing load balancing, and in a case when a different algorithm would be more suitable, a LBScheduler selects different algorithm and a reconfiguration request is send to the rMonitoring service in order to adapt to the change. In this way the internal LBScheduler steers the rMonitoring service in order to optimize the number and the type of events received.

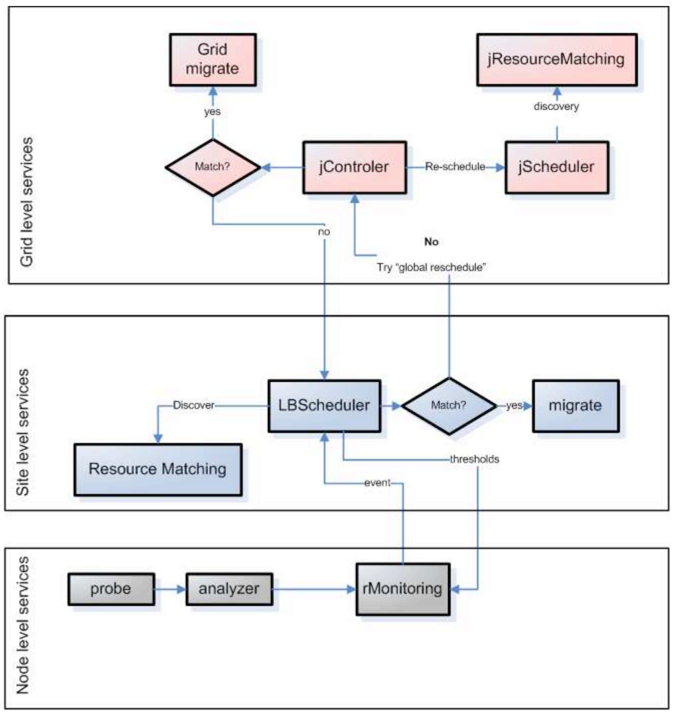

Figure 4.3: Self-adaptable load balancing scheduler with related services and information flow.

Interaction with other WPs

- **WP2.1:** definition and use of VO functionality is important for authorization and accounting of the resources.
- **WP3.3:** load balancing scheduler can trigger a Grid level load balancing, which is performed through the jScheduler. Additionally similar to the LTScheduler, the information about the state of the tasks has to be reported to the Grid level. A SLA re-negotiation is also a protocol envisioned when an unpredicted event happens and can not be handled with load balancing at any level. The SLA re-negotiation will be performed through the jController.
- **WP3.5:** The security infrastructure is an important and crosscutting concern, which we have to implement in order to achieve global goals defined by this WP.

4.7 Virtual Organization Support

The integration of LinuxSSI with the XtreemOS Grid services will need to begin with adding support for the Virtual Organization management services and infrastructure as designed in the workpackage WP2.1.

The first step will be the addition of support for the kernel key retention service (KRS) in Kerrighed. KRS is needed for transporting and attaching Grid certificates and proxies to processes. Currently Kerrighed is unable to migrate or checkpoint processes when the kernel is built with `CONFIG_KEYS` enabled. This feature will mainly require changes in the **ghost** component.

On single Grid nodes, Grid user processes will run under temporary and local user IDs (UID) and group IDs (GID) mapped to global Grid entities like global user ID and VO ID. The local UID/GID will also be used by the XtreemFS Grid filesystem access layer to present a meaningful POSIX view on ownership and access permissions of Grid files. Linux-SSI will let an SSI cluster appear to the Grid as one big powerfull SMP machine. Grid-related services which usually run on each Grid node will run in only one instance on a LinuxSSI cluster. This is important for the daemon taking care of the mapping of local to global user, group and VO identities because the entire LinuxSSI cluster uses common user and group IDs.

Chapter 5

Development Strategy

5.1 Enhancing Kerrighed Stability

The Kerrighed audit has shown that Kerrighed has a serious stability problem. Before adding code developed within the XtreemOS project the Kerrighed tree should undergo a phase of stabilization. Every XtreemOS-SSI developer is encouraged to participate in this step.

Proposed actions:

- Create wiki pages with easy installation instructions for Kerrighed clusters based on the most common distributions (RHEL, CentOS, Fedora, Ubuntu, ...).
- Write and collect a set of functionality tests and benchmarks for Kerrighed. Automated regression testing based on these would be desirable.
- Create metric for code functionality and quality based on the tests such that quality increase and progress can be tracked in some way.
- Test Kerrighed, localize bugs, log them in the Kerrighed bug database, help fixing them.

5.2 Strategy for porting LinuxSSI to new Linux kernel versions

The Kerrighed development is currently done on top of the 2.6.11 kernel. In order to be able to attempt to push any of the changes to the mainline kernel we will need to port the changes to the latest mainline kernel version. Patches are only considered if they are on top of the latest kernel. Currently this means version 2.6.18, but this version will certainly increase during the lifetime of the XtreemOS project.

The port to the latest kernels is also desirable when trying to integrate the Kerrighed changes into a newer distribution like SLES10 or Fedora Core 6, which are adapted to the APIs of the new kernels ($\geq$ 2.6.16).

Porting Kerrighed to newer kernels is a step by step procedure because each kernel version upgrade brings own API changes which need to be dealt with carefully. Therefore the porting should be done first from version 2.6.11 to 2.6.12, then from 2.6.12 to 2.6.13, and so on. A first attempt to forward port to 2.6.14 has been made and showed that each port brings its own difficulties. While 2.6.12 and 2.6.13 only had minor API changes and some variable renamings, 2.6.14 also changed some significant file related structures which required changes in Kerrighed ghost related protocols.

The M18 target kernel version for LinuxSSI should be that of the latest available mainline kernel in month 15 of the project. A forward port of Kerrighed to 2.6.18 should be the next step after the stabilization phase of Kerrighed, before starting the implementation of the LinuxSSI features on this basis.

5.3 Strategy for getting LinuxSSI patches integrated into the mainstream Linux kernels

Kerrighed started as an academic project more than six years ago and has been developed by several students and PhD students. The primary purpose of the project was research and not productization, therefore no serious attempt was made to merge Kerrighed changes into the mainline kernels.

Over time and with growing number of features the modifications became very big and the Kerrighed kernel has diverged quite significantly from mainline kernels. An attempt to push Kerrighed to mainline would encounter significant resistance from the kernel community: no one will spend time to try to understand the huge size of patches and big number of hooks (over 100 kernel files touched), not to mention analysing the proper Kerrighed code, which has around 76000 lines of code. An additional discouraging fact: three other SSI related projects have tried to push their code into mainline and failed, OpenMOSIX, BProc and OpenSSI. The benefit from incorporating SSI concepts into Linux is considered to be irrelevant to the vast majority of Linux users.

Identified problems

A set of problems have been identified during the audit of Kerrighed code. One of them was the coding style which was incompatible to the Linux kernel coding style guide. Such code is usually ignored by the kernel community. This has been mostly fixed in the meantime.

A second problem is the way how `#ifdef` statements are used to encapsulate Kerrighed code. In principle this is a good idea because the compiled code path with Kerrighed disabled keeps no trace of the modifications. On the other hand the large number of `#ifdef` blocks makes the code harder to read and to maintain and is rarely accepted in important source files like `sched.c`, `fork.c` or `exec.c`. Instead of using `#ifdef` statements everywhere one should try to hide them inside macros defined in include files. This way the core C code remains well readable.

An already mentioned problem is the proliferation of the number of kernel hooks. Hooks can be regarded as pluggable function calls which digress normal kernel code execution to Kerrighed specific code. Kernel hooks are very rarely accepted by the community due to their potential to allow unknown proprietary code to interact in unwanted way with the kernel. With more than 100 modified core kernel files, the hooks are a serious problem for pushing kernel code into mainline. For the future development their need should be re-evaluated and alternatives should be investigated.

A last problem on the way to mainline kernels is the strong interdependence of Kerrighed components. While they are modularly coded, the components like *Container*, *Ghost*, *EPM*, *RPC*, *IPC*, *Service Manager*, etc... depend strongly on each other. It is not possible to strip out a small and logically encapsulated component which would have a chance to be accepted into the mainline kernel because there is no component that is functional alone. Kernel patches need to be pushed in small portions which are logically separated and could be used by other (existing) kernel components, as well. Ideally Kerrighed components should be mergeable one by one.

Strategy

A realistic appraisal shows that pushing Kerrighed in near future to the mainline kernel development tree is impossible. Only a long term strategy with small steps and continuous interaction with the community can lead to the acceptance of parts of Kerrighed.

As a first stage we propose to focus on the *Container* component and try to push it into the mainstream. This component requires no kernel hooks at all, therefore the modifications of the core kernel code consist of added code. For Kerrighed the *Container* is a core component on which most of the other components are building. Getting *Container* accepted would open the path to merge other Kerrighed components.

Merging *Container* requires some work which appears only indirectly beneficial to XtreemOS-SSI. The Kerrighed network layer consisting currently of three components (KnetDev, Communication, Service and rpc) needs to be replaced by kernel equivalent functionality. The main candidate for this is TIPC, but its suitability needs to be investigated. In this step *Container* might need to be rewritten to deal with the new network layer. Additionally: as *Container* makes no sense as a stand-alone component, we will need to add a “user” component relying on *Container* functionality and useful for a large number of Linux users. This could be a user level API for using *Container* functionality, some simplified global filesystem cache component or a stripped down version of the *MM* component.

Task	**Functionality**	**Description**	**Expected date**	**Interaction with other WPs**
Scalable mechanisms (T2.2.2, NEC)	SMP support	Support for SMP kerrighed nodes	M18	–
	64 bit support	Add support for x86_64 processors	M18	–
	New kernel version	Port to newer kernel (2.6.18)	M18	–
	Large clusters	Allow more than 128 nodes in a cluster (preferably $>$ 1024)	M48	–
Checkpointing (T2.2.3, UDUS)	Checkpointing of open files and network communication	Needed to support MPI applications	MXX	WP2.1
	Checkpointing of shared memory	Needed to support OpenMP applications.	M18	WP2.1
	Checkpointing containers	Will simplify checkpointing of resources (e.g. open files, shared memory) in LinuxSSI/Kerrighed.	M24	–
	Sytem checkpointer extensions	Extensions for LinuxSSI (see section 4.3.1)	M36	_
	Distributed checkpointing	Improve performance and reliabilty of system checkpointer for LinuxSSI.	M36	_
	Reliable checkpoints	Improve reliability of checkpoints, e.g. by storing them in the LinuxSSI-FS file system.	M36	T2.2.5
	High-speed checkpointing & restart	Allow very short checkpointing time intervals, e.g. 30s.	M36	may be WP3.4 OSS
Reconfiguration mechanisms (T2.2.4, INRIA)	Nodes Addition support	New nodes can be added to the LinuxSSI cluster without stopping it	M18	_
	Nodes Removal Support	Nodes can be removed from the LinuxSSI cluster without stopping it	M18	_
	Notification at user level	We should provide a way for applications running on the cluster to be aware of reconfigurations	M18	WP3.1(API)
	Node failure support	The cluster must continue to run despite multiple node failures	M36	_
	Highly Available File-System	LinuxSSI-FS should adapt to reconfigurations	M36	T2.2.5
	Improvement of reconfiguration mechanisms	Studying cooperation between services to improve efficiency of reconfigurations mechanisms	M36	T2.2.5
LinuxSSI-FS file system (T .2.2.5, INRIA)	Global name-space and hard drives federation	Federate available hard drives and provide a global name-space for the file sytem structure	M9	T2.2.2 (scalability)
	Efficient access	Provide the two striping modes (transparent and customizable) in coordination with the I/O scheduling policies	M12	WP3.1 (API definitions)
	Fault tolerant mechanisms	Provide replica mechanisms (mainly RAID1) and recovery algorithms	M24 (beta version) M36	T2.2.4 (reconfiguration) WP3.1 (API defintions)
	File checkpoint	Extend the *VFS* to add file checkpoint routines	M24	T2.2.3, W2.1 (checkpoint) WP3.1 (API defintions)
	File system reconfiguration	Coordination with reconfiguration mechanisms to readapt file system structure according to node addition/removal.	M36	T2.2.4 (reconfiguration)
	Distributed *VFS*	Integration with the D*VFS* that should be implemented by the Kerrighed community	undefined	–
Scheduler (T2.2.6, XLAB)	Customizable scheduler	Scheduler will support definition of customable probing (monitoring), analyzing and re-scheduling cost functions.	M18	WP3.3
	Self-adaptable scheduler	The scheduler will support active algorithm selection based on a feedback loop connection monitoring and load balancing functionality.	M36	WP3.3, WP3.2

Table 5.1: Linux SSI tasks summarize

Chapter 6

Conclusion

LinuxSSI will be implemented based on the Kerrighed SSI technology. An audit of the current version of Kerrighed for Linux 2.6.11 kernel has been performed from August to October 2006. It showed that Kerrighed is a sound basis to build the LinuxSSI component of XtreemOS as many SSI features are already implemented. However, Kerrighed is not stable enough at the time of writing neither to start the implementation of the basic LinuxSSI features nor to allow proper execution of XtreemOS use cases. Thus, the implementation work in WP2.2 will start with a debugging phase without adding new functionalities to significantly improve the stability of the current version of Kerrighed software. During this phase, we plan to directly contribute to the Kerrighed community that shares with the XtreemOS consortium the objective of a better stability of the existing functionalities. We will report bugs and submit fixes to the Kerrighed community. This will allow XtreemOS consortium to take advantage of the work done towards a better stability of Kerrighed by key developers of the Kerrighed system[1]. Hence, we hope to reach a satisfactory state by the end of the first quarter 2007. The debugging work will be carried out along with the design of LinuxSSI basic functionalities.

LinuxSSI basic functionalities have been specified in this document. In contrast with the current Kerrighed version, LinuxSSI should support SMP cluster nodes and x86-64 bit processors. This point is considered as the highest priority work that we plan to perform at the end of the stabilization phase. The hard-coded parameters currently limiting the scalability of Kerrighed will be removed in LinuxSSI and we plan to evaluate potential additional algorithmic limitations to Kerrighed scalability.

Checkpoint/restart mechanisms implemented in Kerrighed are neither complete nor reliable. By M18, LinuxSSI should provide an appropriate support to checkpoint parallel application units executing on a cluster whatever their communication model (shared memory or message passing). We propose a three level architecture for the checkpointer service including a kernel checkpointer able to checkpoint the state of individual processes, a system checkpointer taking care of

[1]We emphasize that the Kerrighed Key developers do not belong to the XtreemOS consortium.

establishing checkpoints for application units and a Grid checkpointer interacting with the system checkpointer for checkpointing applications that may span multiple Grid nodes. A mechanism to checkpoint/recover the state of containers to be developed constitutes one of the core components of the kernel checkpointer.

Reconfiguration mechanisms allowing a clean node shutdown and incremental boot of a LinuxSSI cluster will be implemented on the basis of the recently released HotPlug module of Kerrighed that is not yet stable at all. We aim at tolerating single node failures in LinuxSSI by M18. However, the reconfigurability of LinuxSSI-FS will only be studied after M18.

Concerning the implementation of high performance disk I/O we will leverage KerFS distributed file system for the design and implementation of LinuxSSI-FS. We will focus on two main aspects. First, we will improve KerFS stability to be able to use LinuxSSI-FS as the root file system. We will also target efficiency implementing customizable striping mechanisms and I/O scheduling mechanisms.

About the customizable scheduler fo processes, we will focus on two complementary components: the load balancing scheduler and the long-term scheduler, which is being an optional to an LinuxSSI. The load balancing scheduler will be capable of accepting probing (monitor), analyzer and optimization function plug-ins, which will make the whole architecture highly customizable, and capable of accepting various scheduling policies. Regarding the long-term scheduler we plan to support DRMAA standard, which will allow us to use any existing batch scheduler supporting this standard. We shall concentrate on supporting Maui or GridWay scheduler. Basic functionality, with a simple exemplary plug-ins and DRMAA support will be implemented by M18.

Some desirable advanced functionalities have already been identified but will only be implemented in the second half of the XtreemOS project, after M18. In particular, it would be interesting in large clusters to support high speed networks such as Infiniband. Indeed, supporting specific drivers for these networks rather than relying on the generic NetDevice Linux driver will improve the LinuxSSI performances.

Concerning Checkpoint/restart, it may be interesting to study different checkpointing strategies for parallel application units. The work performed on application unit checkpointing on SSI clusters in WP2.2 will be coordinated with the work done on checkpointing application units on individual PC in WP2.1 even if we do not expect to have fully compatible mechanisms for both kinds of Grid nodes (individual PC running Linux and clusters running LinuxSSI). We will also coordinate our work in WP 2.2 with that of WP3.3. The application management service developed in WP3.3 will indeed use the checkpoint/restart mechanisms provided by LinuxSSI in the framework of a Grid level checkpointers taking into account every unit of an application spanning multiple Grid nodes.

Another advanced functionality is the fault tolerance support in LinuxSSI-FS to allow reconfigurability of the file system and to offer an efficient support to the system checkpointer for saving the state of the open files of an application when it is checkpointed. We plan to interact with WP3.4, in charge of the design of the XtreemFS Grid data management service in our work on LinuxSSI-FS as LinuxSSI clusters may be client or server of XtreemFS. We will also further investigate how containers could be used in the framework of the GOM service, developed in WP3.4.

For the customizable scheduler, the most important advanced functionality implemented is the adaptive feedback loop, which will allow adaptation of the scheduling policies and algorithms based on the system state. Additionally innovative load balancing policies exploiting features specific to SSI clusters will be studied. Moreover, the customizable scheduler of LinuxSSI interacts mostly with application execution management services - AEM. The AEM structure defines services [14], which are responsible for job scheduling on a Grid level, and depend on the information retrieved from local resource management and scheduling systems.

Table 5.1 shows a summary of LinuxSSI functionalities with the planned schedule The schedule after M18 is a tentative schedule subject to revision.

Before M18, we do not plan to integrate virtual organization and security mechanisms in LinuxSSI basic version as these mechanisms will be developed concurrently with the design and development of LinuxSSI. We plan to carry out the integration work once the basic version of the various components of WP2.1 and WP3.5 and of LinuxSSI are released. Some WP2.2 partners are also involved in WP2.1 and WP3.5. Thus, we anticipate that we should be able to take into account fundamental design decisions regarding security and VO management functionalities in LinuxSSI design.

We do not expect to have all LinuxSSI patches quickly accepted in the Linux community as they are too numerous and they are interleaved. We will work on revisiting Kerrighed patches to minimize them when possible in LinuxSSI and to isolate some subparts of LinuxSSI such as containers for instance to better push them in the Linux community. Getting LinuxSSI patches accepted in the mainline Linux kernel is one of our key objectives but it requires long term efforts and careful design of LinuxSSI basic functionalities.

Bibliography

[1] Distributed resource management application api working group. http://drmaa.org/wiki.

[2] Grid engine. http://gridengine.sunsource.net.

[3] Kerrighed. http://www.kerrighed.org.

[4] Maui cluster scheduler. http://www.clusterresources.com/pages/productts/maui-cluster-scheduler.php.

[5] Mosix. http://www.mosix.org.

[6] Openmosix. http://openmosix.sourceforge.net.

[7] Openssi. http://openssi.org.

[8] Openssi process load balancing. http://openssi.org/cgi-bin/view?page=docs2/1.9/README-mosixll.

[9] Pbs professional. http://www.altair.com/software/pbspro.htm.

[10] Amnon Barak and Oren La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4–5):361–372, 1998.

[11] Peter J. Braam. File systems for clusters from a protocol perspective. In *Extreme Linux Workshop #2, USENIX Technical Conference*. USENIX, June 1999.

[12] XtreemOS consortium. Deliverable D4.2.1, November 2006.

[13] XtreemOS consortium. Annex 1 - description of work. Integrated Project, April 2006.

[14] XtreemOS consortium. Requirements and specification of xtreemos services for application execution management, November 2006.

[15] XtreemOS consortium. Specification of federation resource management mechanisms, November 2006.

[16] Pascal Gallard. *Conception d'un service de communication pour systèmes d'exploitation distribué pour grappes de calculateurs: mise en oeuvre dans le système à image unique Kerrighed*. Thèse de doctorat, IRISA, Université de Rennes 1, IRISA, Rennes, France, December 2004.

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

[18] Arie Keren and Amnon Barak. Opportunity cost algorithms for reduction of i/o and interprocess communication overhead in a computing cluster. *IEEE Trans. Parallel Distrib. Syst.*, 14(1):39–50, 2003.

[19] Kerrighed website. http://www.kerrighed.org. http://www.kerrighed.org.

[20] Kerrighed project on INRIA gforge. http://gforge.inria.fr.

[21] Adrien Lebre, Guillaume Huard, Przemyslaw Sowa, and Yves Denneulin. I/O Scheduling service for Multi-Application Clusters. In *Proceeding of the IEEE International Conference on Cluster Computing, Barcelona, SP, to appear*, Sept 2006.

[22] Renaud Lottiaux. *Gestion globale de la mémoire physique d'une grappe pour un système à image unique : mise en œuvre dans le système Gobelins*. Thèse de doctorat, IRISA, Université de Rennes 1, December 2001.

[23] 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.

[24] J. Michael Meehan and Adam Wynne. Load balancing experiments in openmosix. In *Computers and Their Applications*, pages 314–319, 2006.

[25] Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, David Margery, Jean-Yves Berthou, and Isaac Scherson. Kerrighed and data parallelism: Cluster computing on single system image operating systems. In *Proc. of Cluster 2004*. IEEE, September 2004.

[26] H. Rajic, R. Brobst, W. Chan, F. Ferstl, J. Gardiner, J. Robarts, A. Haas, B. Nitzberg, and J. Tollefsrud. Distributed resource management application api specification 1.0. Technical report, Global Grid Forum, 2004. http://www.gridforum.org/documents/GFD.22.pdf.

[27] M. Roehrig, W. Ziegler, and P. Wieder. Grid scheduling dictionary of terms and keywords. Technical report, Global Grid Forum, 2002. http://www.ggf.org/documents/GFD.11.pdf.

[28] Geoffroy Vallée. *Conception d'un ordonnanceur de processus adaptable pour la gestion globale des ressources dans les grappes de calculateurs : mise en oeuvre dans le système d'exploitation Kerrighed.* Thèse de doctorat, IFSIC, Université de Rennes 1, France, March 2004.

[29] 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. Industrial Track.