Project no. IST-033576

XtreemOS

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

Design and Implementation of Basic Reconfiguration Mechanisms
D2.2.4

Due date of deliverable: November 31th, 2007
Actual submission date: January 10th, 2008

Start date of project: June 1st 2006

Type: Deliverable
WP number: WP2.2
Task number: T2.2.4

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

Version 1.0 / Last edited by Matthieu Fertré / January 10th, 2008

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	5/10/07	Matthieu Fertré	INRIA	Initial version of the document
0.2	15/10/07	Matthieu Fertré	INRIA	Use XtreemOS stylesheet
0.3	18/10/07	Matthieu Fertré	INRIA	Some typographic corrections, update LinuxSSI architecture figure
0.4	19/10/07	Matthieu Fertré	INRIA	Cosmetic changes
0.5	23/10/07	Matthieu Fertré	INRIA	Information about subsession
0.6	07/11/07	Matthieu Fertré	INRIA	Typo fixes, various English fixes
0.7	09/11/07	Matthieu Fertré	INRIA	Modification of the layout, various section updates
0.8	12/11/07	Matthieu Fertré	INRIA	Various section updates
0.9	13/11/07	Matthieu Fertré	INRIA	Update section about necessity of reconfiguration mechanisms
0.10	13/11/07	Matthieu Fertré	INRIA	First version of abstract
0.11	14/11/07	Matthieu Fertré	INRIA	Update section about RPC service
0.12	14/11/07	Matthieu Fertré	INRIA	Update section about Hotplug API
0.13	14/11/07	Matthieu Fertré	INRIA	First version of conclusion
0.14	18/11/07	Matthieu Fertré	INRIA	Update references (bibtex file)
0.15	26/11/07	Matthieu Fertré	INRIA	Typo fixes
0.16	27/11/07	Matthieu Fertré	INRIA	Update section about cluster membership
0.17	28/11/07	Matthieu Fertré	INRIA	Update section about LinuxSSI architecture
0.18	03/12/07	Matthieu Fertré	INRIA	Update section about Hotplug service
0.19	03/12/07	Matthieu Fertré	INRIA	Modify outline
0.20	03/12/07	Matthieu Fertré	INRIA	Update section about Services reconfiguration
0.21	03/12/07	Matthieu Fertré	INRIA	Update introduction, conclusion and abstract
0.22	12/12/07	Matthieu Fertré	INRIA	Typo, english fixes
0.23	18/12/07	Matthieu Fertré	INRIA	Reviewers remarks
1.0	10/01/08	Matthieu Fertré	INRIA	Latest reviewers remarks

Reviewers:
Guillaume Pierre (VUA), Samuel Kortas (EDF)

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T2.2.4	Design and implementation of advanced reconfiguration mechanisms	INRIA*

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Contents

1 Introduction 3

2 Management of reconfiguration events 4
2.1 Typical reconfiguration cases 4
2.2 Goal of reconfiguration service 4
2.3 Command-line Interface for cluster administration 4

3 LinuxSSI Software Architecture 5
3.1 Overview of LinuxSSI 5
3.2 The Low Level services: RPC and Hotplug 6
3.2.1 The RPC service 6
3.2.2 The Hotplug service 6
3.3 The KDDM service 7
3.4 The High Level services 7

4 Cluster membership 8
4.1 Identification of nodes 8
4.2 State of nodes 9

5 Hotplug 12
5.1 Updating node vectors 12
5.2 Coordinating reconfiguration of services 12
5.2.1 Registration of service to Hotplug notifications 13
5.2.2 Order of reconfiguration 14
5.2.3 Example of Hotplug registration and reaction to reconfiguration event 16

6 Reconfiguration in a LinuxSSI cluster 16
6.1 RPC service reconfiguration 16
6.2 KDDM service reconfiguration 16
6.3 Reconfiguration of High Level Services 18
6.3.1 Principle 19
6.3.2 Impact on High Level Services 19

7 Conclusion 21

Executive summary

The goal of LinuxSSI is to manage cluster of hundreds or thousands of nodes. In a LinuxSSI cluster all nodes work closely together so that in many respects they can be viewed as though they were a single SMP computer running Linux-XOS. For maintenance reasons, it is mandatory to provide reconfiguration mechanisms. These mechanisms should allow administrators to upgrade hardware on one or many nodes without stopping the whole cluster and especially without stopping applications execution.

Reconfiguration operations initiated by the administrator should be transparent to applications. Moreover, failure must not lead to cluster crash nor to application crash if no process of the application is running on the node(s) impacted by the failure.

During the work on the reconfiguration mechanisms, we have designed a simple interface for administration. We have also designed the Hotplug service, a framework that handles hot node(s) addition/removal and coordinates services reconfiguration. High Level services register to the Hotplug service that calls functions provided by services. These functions are called in a predefined order to take dependencies between services into account.

The administration interface is implemented but should be completed to give more feedback to the administrator. The Hotplug service is nearly finalized, thus the framework needed to coordinate reconfiguration operations is available. Reconfiguration of KDDM service is working in most case but it can lead to bug in corner cases. Reconfiguration of High Level services needs additinal work. Many services have not been studied yet.

Node addition is working pretty well and the global scheduler already takes new nodes into account. Node removal works in very simple case but needs more High Level services support.

In the short term, the roadmap will focus on stabilization of basic reconfiguration mechanisms and especially node removal. In the medium term, we plan to add support for reconfiguration mechanisms for all High Level services. In addition, we also plan to study and add mechanisms to handle node or network link failure.

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 [3], the XtreemOS cluster flavour relies on the SSI approach. 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 grid services, a cluster executing LinuxSSI-XOS will be seen as a powerful SMP computer executing Linux-XOS.

Former investigations from WP2.2 work package [4] led to leverage the Kerrighed SSI technology developed by INRIA in cooperation with EDF [7, 8, 9, 11, 12].

In this document, we present the design and the implementation of basic reconfiguration mechanisms in LinuxSSI single system image for clusters. Major functionalities implemented until M18 are emphasized.

The goal of reconfiguration mechanisms is to handle cluster start-up and stop, hot node(s) addition or removal and node or network link failure. For maintenance reasons, it is mandatory to provide reconfiguration mechanisms. These mechanisms should allow administrators to upgrade hardware on one or many nodes without stopping the whole cluster and especially without stopping applications execution.

The work on reconfiguration mechanisms has been done in collaboration with the Kerrighed community (mostly Pascal Gallard from Kerlabs company) to avoid code and effort duplication.

This document focuses on basic mechanisms to deal with reconfiguration events initiated by the administrator such as node addition or node removal. Node or network link failure is out of the scope of this document. We do not discuss about application reconfiguration in this document.

In Section 2, we introduce the reconfiguration problem. Then, Section 3 presents the software architecture of LinuxSSI that is implemented as a set of distributed services. Section 4 provides details about the naming and the different states of cluster nodes. Section 5 explains the role and the interface of the Hotplug service. Finally, Section 6 presents what has to be done in each service to support reconfiguration events.

2 Management of reconfiguration events

2.1 Typical reconfiguration cases

In a cluster, many events can lead to reconfigurations. The first one is the complete start-up of cluster for the first time, or after a complete shutdown of the cluster. Complete shutdown of the cluster may be due to an electric failure or to a maintenance operation. Some events are driven by the administrator, such as addition of new nodes to make the cluster more powerful. Removal of nodes is driven by the administrator too, the nodes can be removed definitely or for the time of a maintenance operation. Failures also lead to reconfiguration. We can identify two kinds of failure: node failure (the node stops execution because of hardware or software breakdown), and network link failure.

From these events, we can identify five kinds of reconfigurations: (i) the whole cluster starts with all available nodes; (ii) the whole cluster stops; (iii) some nodes are added to the cluster; (iiii) some nodes are removed from the cluster; (iiiii) a failure occurs that makes some cluster nodes unavailable.

2.2 Goal of reconfiguration service

At the beginning of the XtreemOS project, Kerrighed that LinuxSSI leverages did not provide any reconfiguration mechanisms but only cluster start-up and cluster shutdown (see case (i) and (ii) in Section 2.1). It means that adding a node to a running cluster or removing a node forced to restart the whole cluster and all running applications. Moreover, one node or network link failure led to a crash of the whole cluster.

LinuxSSI has to adapt to events that lead to reconfiguration. All clusterwide services must be able to continue to work despite reconfigurations. That is why reconfiguration mechanisms need to be integrated in the LinuxSSI design. Reconfiguration operations initiated by the administrator should be transparent to applications. Moreover, a failure must not lead to cluster crash nor to application crash if no process or object of the application is running or stored on the node(s) impacted by the failure. However, high availability of applications is out of the scope of reconfiguration mechanisms. If one or more processes of an application is running on a node that fails, the application will probably fails unless the application is fault tolerant by itself.

Reconfiguration events that are not initiated by the administrator are out of the scope of this document.

2.3 Command-line Interface for cluster administration

For the time being, node addition or removal can be issued manually by the administrator.

In the following section, we assume that each cluster node is identified by a number known by the cluster administrator.

The command line interface used in LinuxSSI is the same as the one used in Kerrighed. It is available only to the cluster administrator.

`krgadm cluster start` starts a cluster with all available nodes (see case (i) in Section 2.1).

`krgadm cluster stop` stops all nodes in the cluster. (see case (ii) in Section 2.1)

`krgadm nodes add -n16:18` tries to add nodes 16 and 18 to an already running cluster (see case (iii) in Section 2.1). If nodes are not available (*eg*: turned-off), an error message is returned to the administrator.

`krgadm nodes del -n21:34` tries to remove nodes 21 and 34 from the cluster (see case (iiii) in Section 2.1). If nodes are not participating in cluster (*eg*: turned-off, not yet added in the cluster), an error message is returned to the administrator.

In the current implementation of LinuxSSI, the addition or removal of a node n cannot be requested from node n. It must be requested from another cluster node.

3 LinuxSSI Software Architecture

As reconfiguration mechanisms need to be integrated in the LinuxSSI design, it is important to study first the initial design of LinuxSSI (based on Kerrighed operating system).

3.1 Overview of LinuxSSI

Kerrighed is a modified Linux operating system extended with distributed services running on top of Linux kernel. However, to implement these distributed services, modifying the Linux kernel is sometimes necessary.

The LinuxSSI architecture is divided in several services as shown in Figure 1. We can sort services in three sets of services, each set representing a level in Linux SSI software stack:

- Low Level services (RPC and Hotplug) are related to the LinuxSSI communication sub-system;
- KDDM service provides a generic mechanism to easily and efficiently access remote data;
- High Level services are all the other services.

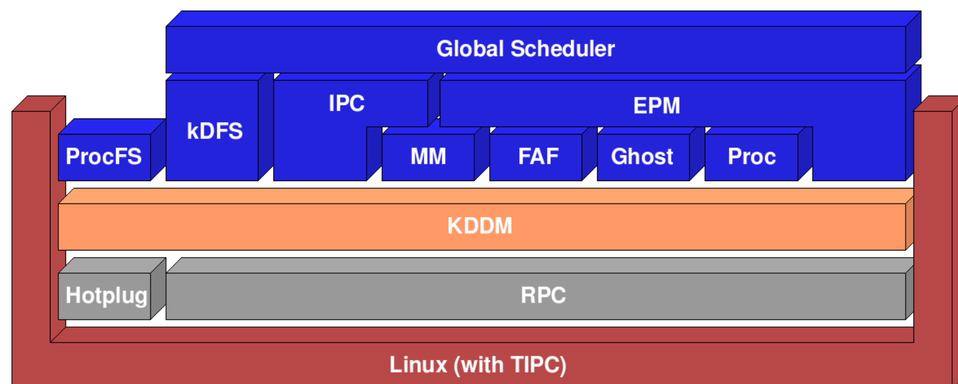

Figure 1: Overview of LinuxSSI Architecture

3.2 The Low Level services: RPC and Hotplug

3.2.1 The RPC service

The RPC service implements a high level distributed service manager. Explicit communications between nodes are implemented on top of this interface.

The RPC service provides an API to communicate from one node (the client) to one or many nodes (the server(s)). Nodes are called by their identifier (see Section 4.1) not by their IP/Ethernet address.

The client initiates the connexion by providing an RPC identifier and a set of server nodes. It gets a RPC descriptor. The RPC identifier is used by the server(s) to know which function to execute and which initial parameters have been sent to it. The RPC descriptor is used to communicate data from the client to the server or from the server to the client.

3.2.2 The Hotplug service

The Hotplug service aims to coordinate reconfiguration of services in the event of cluster reconfiguration. It provides an infrastructure for reconfiguration of all other services in LinuxSSI. The Hotplug service is described in section 5.

Currently, communication in the cluster relies on TIPC (Transparent Inter Process Communication) protocol [2]. TIPC has been designed by Ericsson and is available in Linux kernel since version 2.6.16. However development is made out of the kernel mainstream.

TIPC Open Source project defines the TIPC protocol as follows:

> *TIPC is designed for use in clustered computer environments, allowing designers to create applications that can communicate quickly and reliably with other applications regardless of their location within the cluster. The TIPC protocol originated at the telecommunications manufacturer, Ericsson, and has been deployed in their products for years.*

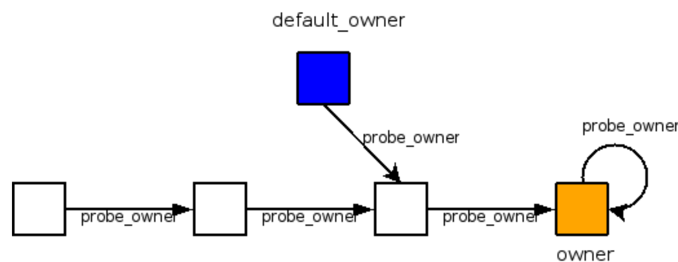

Figure 2: KDDM default and probe owner

TIPC automates the detection of nodes in a cluster. Thus, the TIPC layer can inform the Hotplug service of the arrival or departure (failure or shutdown) of individual nodes.

3.3 The KDDM service

The *Kerrighed Distributed Data Manager* (KDDM) service, formerly known as *Container* [10], offers high level mechanisms to easily and efficiently access remote data. Most Kerrighed services are implemented on top of the *KDDM* service.

Originally designed to store pages, the KDDM mechanism has evolved to store any object in sets. A set is identified by a `kddm_set_id_t`. Objects in the same set have the same data type.

From any node of the cluster, the programmer can request an object in read (`kddm_get_object`) or read-write (`kddm_grab_object`) mode. After using the object, the programmer is responsible for unlocking the object (`kddm_put_object`). An object is retrieved by the programmer giving its set identifier and object identifier within the set. The set identifiers and object identifiers are global to the cluster, and therefore share the same name space.

The current *owner* of an object is the only node which has the valid copy. If a node has already used an object and is not the owner anymore, it has a link to a *probe owner*. Following the chain of *probe owners*, one is guaranteed to find the *current owner*.

If a node needs to access an object it has never used before, it contacts the *default owner*, which must have a link to a *probe owner*. The default owner is defined by a hash function.

Figure 2 illustrates the probe owners chain and the default owner.

3.4 The High Level services

ProcFS implements the global `/proc` directory. It provides the same files as a regular `/proc` directory but with data representing the whole cluster (global memory usage, list of all running processes, etc).

kDFS implements a distributed file system for cluster [6].

IPC implements a distributed version of IPC mechanisms (shared memory segments, message queues and semaphores). It relies on the MM service for shared memory segments.

FAF (File Access Forwarding) implements support for migration of open files and the sharing of file pointers for processes running on different nodes.

MM is responsible for the migration of process address space and for cluster wide memory sharing.

Ghost implements a generic layer used to export / import kernel-level meta-data into files or network streams. It is used to migrate, checkpoint and duplicate processes.

Proc implements distributed processes management. It is responsible for global process naming, distant signalling, etc.

EPM (*Enhanced Process Management*) implements process migration, process checkpointing, distant fork and distributed thread management.

Global Scheduler implements different global process scheduling policies [5].

4 Cluster membership

In this section we describe how nodes are identified. Then we explains the various states in which nodes can be when reconfiguration events are taken into account. It explains the node view of the cluster membership.

4.1 Identification of nodes

Several Kerrighed clusters may share the same physical network and be independent. Each cluster therefore defines a `session_id` parameter that is determined for each node by the administrator either as kernel parameter with the boot loader or as an option in file `/etc/kerrighed_nodes`. Nodes from a session communicate only with other nodes from the same session; messages coming from other sessions are simply ignored. Physical node can change session only after a reboot.

Nodes are identified in a session by an integer called the *node_id*. Nodes that are not in the same session may have same *node_id* (see Figure 3). Before the integration of the hotplug subsystem, Kerrighed used an integer variable `kerrighed_nb_nodes` copied on all nodes. `kerrighed_nb_nodes` counted the number of nodes in the cluster and the *node_id* of each node was from `0` to `kerrighed_nb_nodes`. The assumption was that there was no hole in the node identifiers range.

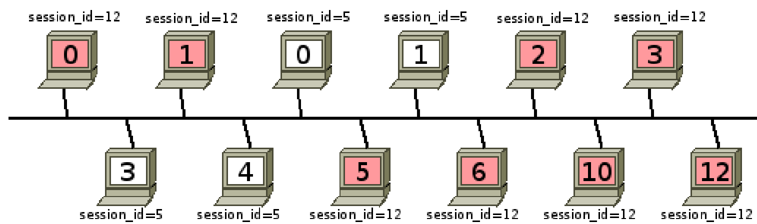

Figure 3: Two logical clusters sharing the same physical network

So, iterating on all nodes of the cluster was possible with pieces of code like the following one:

```
kerrighed_nodeid_t n;
for (n = 0; n < kerrighed_nb_nodes; n++)
    do_some_stuff(n);
```

In the presence of reconfigurations, the previous assumption becomes invalid. To be able to add / remove nodes while the cluster is running, 2 alternatives were identified: recomputing node identifiers to ensure continuation of node identifiers or authorizing discontinuation in node identifiers. We choose the second solution because the first one would have cause some communication interruption between two nodes when one of them is renamed.

To allow discontinuity of node identifiers, we use a bit vector with one bit per node. If the bit is set to '1', it means that the node exists or else the node does not exist. `kerrighed_nb_nodes` is replaced by a function counting the bit set to '1' in the vector.

4.2 State of nodes

In the context of reconfiguration, nodes can be in four different states from the view of the session.

OUT OF THE SESSION: the node is outside the cluster session. The physical node may not exist, may be turned-off but it may also participate in a different Kerrighed session.

PRESENT: the node is physically connected to the cluster through TIPC and shares the same *session_id*. As soon as a node is PRESENT, it *can* handle RPC requests. Nevertheless, requests may be ignored if no handler is registered to answer to them.

ONLINE: base foundations of Kerrighed are running (up to KDDM), but the node can be in "transition" phase: being inserted, removed, in reconstruction, etc. ONLINE implies PRESENT.

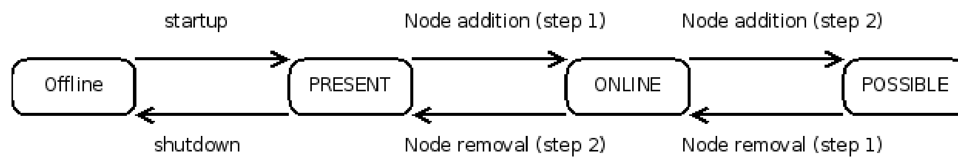

Figure 4: State machine of cluster node

POSSIBLE: the node is logically connected to the cluster. All high-level services of Kerrighed are running. The node is one of the *possible* nodes to ask for a service. POSSIBLE implies ONLINE.

The name of the states comes from the Linux kernel subsystem to add or remove CPU [1].

Figure 4 represents the state machine of node in the event of a node addition or removal. *Step 1* of reconfiguration operation are initiated by the administrator. After *step 1* is finished, *step 2* requires an agreement of all cluster nodes to be initiated.

In contrast to a peer-to-peer system, each Kerrighed node has a global view of the system. Kerrighed services rely on the consistency of this global view. Using a centralized server for storing this global view is not a suitable approach because it would create a single point of failure.

The states of all nodes of the cluster are stored in each node in three bit vectors, which define the membership:

MEMBERSHIP_PRESENT: Nodes marked with '1' in this bit vector are PRESENT. Nodes marked with '0' are OUT OF THE SESSION. This vector defines the physical membership and is used to know if we can add a node in the cluster.

MEMBERSHIP_ONLINE: Nodes marked with '1' in this bit vector are ONLINE. This vector defines what we can see as the KDDM logical membership. Services depending only on MEMBERSHIP_ONLINE can be used during an addition or a removal of a node.

MEMBERSHIP_POSSIBLE: Nodes marked with '1' in this bit vector are POSSIBLE. This vector defines the logical cluster membership and is used by high-level services.

Note that the following property is always true:

$$\begin{aligned} MEMBERSHIP_POSSIBLE \subseteq\ & MEMBERSHIP_ONLINE \\ & \subseteq MEMBERSHIP_PRESENT \end{aligned}$$

When no reconfiguration operation is in progress, all nodes agree on the value of each vector. Mechanisms to maintain this global view are described later in Section 5.1.

	node identifier							
	0	1	2	3	4	5	6	7
MEMBERSHIP_POSSIBLE	0	0	1	1	0	1	0	1
MEMBERSHIP_ONLINE	0	1	1	1	0	1	1	1
MEMBERSHIP_PRESENT	0	1	1	1	1	1	1	1

Figure 5: Bit vectors representing a running cluster of 8 nodes in various states

Example: Figure 5 presents the membership bit vectors of an 8-node cluster where:

- Node 0 is OUT OF THE SESSION. Node 0 may not exist, may be turned-off or may be participating in another Kerrighed session.
- Nodes 2, 3, 5 and 7 are up and running Kerrighed.
- Node 4 is physically connected to the cluster. Thus, node 4 is a Kerrighed node sharing same session than the cluster. However node 4 is not in the cluster and no adding operation for this node is in progress.
- Nodes 1 and 6 are on-line and there is an addition or removal operation in progress for them. Notice that the operations can be different for node 1 and node 6 (*eg*: node 1 is currently involved in a removal operation, node 6 is involved in an addition operation).

Nodes 2, 3, 5, and 7 have this complete view.
Node 1 must maintain a consistent view of the MEMBERSHIP_POSSIBLE vector as long as node 1 is not removed (still in the MEMBERSHIP_ONLINE vector). Thus, node 1 has also this complete view.
Node 6 may have an non consistent view of the MEMBERSHIP_POSSIBLE vector but has a consistent view of the MEMBERSHIP_ONLINE and MEMBERSHIP_PRESENT vectors. Node 6 does not care of MEMBERSHIP_POSSIBLE vector as long as node 6 is not fully participating in the cluster (not yet in the MEMBERSHIP_POSSIBLE vector).
Information about the MEMBERSHIP_POSSIBLE and the MEMBERSHIP_ONLINE vectors is uninteresting for node 4 and it may have invalid information.
As node 0 is OUT OF THE SESSION, it has no view about membership in the current session.

5 Hotplug

The Hotplug service is responsible for:

- responding to administrator requests for node(s) additions or node(s) removals (section 2.3);
- updating the node vectors: MEMBERSHIP_PRESENT, MEMBERSHIP_ONLINE, MEMBERSHIP_POSSIBLE;
- synchronizing the upper services and informing them about reconfiguration events.

5.1 Updating node vectors

As explained in Section 3.2, the MEMBERSHIP_PRESENT vector is updated on all cluster nodes thanks to the information provided by TIPC. This is consistent with the MEMBERSHIP_PRESENT definition (see Section 4), which is to represent the physical membership.

The Hotplug service is in charge of updating the MEMBERSHIP_ONLINE and MEMBERSHIP_POSSIBLE vectors by itself.

Nodes are appended to the MEMBERSHIP_POSSIBLE vector at the very end of a node(s) addition operation when all the Kerrighed services are ready on the added node(s). The update is done on the whole cluster by broadcasting a message packing a bit vector with added node(s) marked as '1' in the bit vector. Then each node can update its MEMBERSHIP_POSSIBLE vector.

On the contrary, when removing node(s), the evicted nodes are removed from the MEMBERSHIP_POSSIBLE at the very beginning of the removal operation. This guarantees that high-level services, which rely on the MEMBERSHIP_POSSIBLE vector, do not create nor migrate objects on the leaving node(s). Thus, no additional dependency is created from the cluster to the leaving node(s) during the removal operation.

The MEMBERSHIP_ONLINE vector refers to the state of the KDDM service. A node is inserted in the MEMBERSHIP_ONLINE vector as soon as the KDDM base mechanisms are ready and the KDDM namespaces are merged.

5.2 Coordinating reconfiguration of services

The Hotplug module coordinates the reconfiguration of services. The following section explains how services register to Hotplug notifications. Then, it explains which order is chosen to notify each service one by one in case of reconfiguration events.

```
1  typedef enum {
2          HOTPLUG_NOTIFY_ADD,
3          HOTPLUG_NOTIFY_REMOVE_LOCAL, /* the local node will
               leave the cluster */
4          HOTPLUG_NOTIFY_REMOVE_ADVERT, /* some nodes (but
              not the local one) will leave the cluster */
5          HOTPLUG_NOTIFY_REMOVE_ACK, /* some nodes have left
              the cluster */
6          HOTPLUG_NOTIFY_FAIL, /* some nodes have left the
              cluster by failure */
7  } hotplug_event_t;
8
9  struct hotplug_node_set {
10         int subclusterid;
11         krgnodemask_t v;
12 };
13
14 struct notifier_block;
15
16 int register_hotplug_notifier(int (*notifier_call)(struct
       notifier_block *, hotplug_event_t, struct
       hotplug_node_set *), int priority);
```

Figure 6: API to register to the Hotplug service

5.2.1 Registration of service to Hotplug notifications

Services can register to the Hotplug service (using the API showed in Figure 6) to be informed when a node addition, removal or failure occurs.

Each service that registers to Hotplug notifications defines one function f with the following prototype:

```
1  int f(struct notifier_block *, hotplug_event_t, struct
       hotplug_node_set *);
```

The function takes a `struct notifier_block*`, a `hotplug_event_t` and a `struct hotplug_node_set*` as arguments. The `struct notifier_block*` is given by the standard Linux notification API and is useless in our approach. The `hotplug_event_t` describes the kind of events happening, the `struct hotplug_node_set*` provides information about which nodes are in reconfiguration stage. This function is called each time the Hotplug service sends a reconfiguration events.

The function is provided (as a pointer to a function) to the Hotplug service by calling the following function:

```
1  register_hotplug_notifier(f, priority);
```

`priority` is an integer used to order the notification callback of the different

services.

5.2.2 Order of reconfiguration

As explained in Section 5.2.1, each service registers to Hotplug notifications with a priority. This priority helps the Hotplug service to know in which order services must be reconfigured.

Services are informed one by one by the Hotplug service from the lowest to the highest level service when an addition happens and from the highest to the lowest one when a removal occurs.

Node addition requires to notify Kerrighed services from the lowest to the highest level because highest level services need lowest level services to work. On the contrary, when removing a node, highest level services are notified first so that they can stop one by one. Moreover, highest level services have the semantic view of system objects and know what to do with them in the event of a removal operation (exporting object from/to the cluster, destroying the object, etc.).

The proposed chain order is the following:

1. HOTPLUG_PRIO_MEMBERSHIP_PRESENT
2. HOTPLUG_PRIO_RPC
3. HOTPLUG_PRIO_KDDM_BASE
4. HOTPLUG_PRIO_MEMBERSHIP_ONLINE
5. HOTPLUG_PRIO_KDDM_SERVICES
6. HOTPLUG_PRIO_PROCFS
7. HOTPLUG_PRIO_EPM
8. HOTPLUG_PRIO_MEMBERSHIP_POSSIBLE

Priorities HOTPLUG_PRIO_MEMBERSHIP_* refers to the update of the different node vectors. Other priorities are related to some services. In the current implementation, only the RPC, KDDM, ProcFS and EPM services are taken into account.

KDDM notification is splitted over HOTPLUG_PRIO_MEMBERSHIP_ONLINE: HOTPLUG_PRIO_KDDM_BASE, HOTPLUG_PRIO_KDDM_SERVICES. The KDDM *Base* matches KDDM point-to-point interactions. This is used at low level, for merging/splitting the KDDM namespace. The KDDM *Services* relies on default, probe owners and thus on MEMBERSHIP_ONLINE vector. KDDM *Services* works on consistent KDDM namespace while KDDM *Base* can work on evolving namespace(s).

```
1  static int membership_possible_notification(struct
       notifier_block *nb, hotplug_event_t event, struct
       hotplug_node_set *node_set)
2  {
3          switch(event){
4          case HOTPLUG_NOTIFY_ADD:
5                  membership_possible_add(&node_set->v);
6                  break;
7          case HOTPLUG_NOTIFY_REMOVE_LOCAL:{
8                  kerrighed_node_t node;
9                  for_each_possible_krgnode(node)
10                         if(node != kerrighed_node_id)
11                                 clear_krgnode_possible(node
                                       );
12                 break;
13         }
14         case HOTPLUG_NOTIFY_REMOVE_ADVERT:
15                 membership_possible_remove(&node_set->v);
16                 break;
17         default:
18                 break;
19         } /* switch */
20         return NOTIFY_OK;
21 }
22
23 int hotplug_membership_init(void)
24 {
25         register_hotplug_notifier(
26                 membership_present_notification,
27                 HOTPLUG_PRIO_MEMBERSHIP_PRESENT);
28         register_hotplug_notifier(
29                 membership_online_notification,
30                 HOTPLUG_PRIO_MEMBERSHIP_ONLINE);
31         register_hotplug_notifier(
32                 membership_possible_notification,
33                 HOTPLUG_PRIO_MEMBERSHIP_POSSIBLE);
34         return 0;
35 }
```

Figure 7: Example of a service reacting to reconfiguration operations

5.2.3 Example of Hotplug registration and reaction to reconfiguration event

Figure 7 presents an example of a service reacting to reconfiguration operation. This service is part of the Hotplug service and is in charge of the update of the MEMBERSHIP_POSSIBLE vector.

The service has registered to Hotplug notifications thanks to line 31 of code in Figure 7.

When a reconfiguration event happens, function `membership_possible_notification`, defined in line 1 of Figure 7, is called.

It tests (`swith (event){`, line 3) whether it is an addition (line 4), a local removal (line 7) or a remote removal (line 8) of node(s) and updates the membership vector consequently thanks to information provided by `node_set`.

In the event of a node(s) addition, as described in Section 5.2.2, following the proposed chain order, function `membership_possible_notification` registered by this service is the last called by the Hotplug service. All other services are notified before. On the contrary, in the event of a node (s) removal (local or remote), function `membership_possible_notification` registered by this service is the first called by the Hotplug service. All other services are notified after.

6 Reconfiguration in a LinuxSSI cluster

This Section describes, for each LinuxSSI service, how it needs to be modified to handle reconfiguration events.

6.1 RPC service reconfiguration

The RPC service is stateless and provides point-to-point(s) communication. Thus, it does take membership into account.

Reconfiguration of the RPC service only consists of flushing the sending queue at the very end of a local node removal operation.

6.2 KDDM service reconfiguration

In a dynamic cluster, nodes may join or leave at any time. During the addition or the removal of a node, some KDDM objects can be added/removed from the global namespace. The KDDM mechanism must therefore guarantee the consistency of the probe owner and the default owner for all objects at all time even if nodes are added or removed.

When a kernel object has to be accessible from several nodes, this object is stored using the KDDM service. As detailed in Section 3.3, objects are retrieved with a set identifier and an object identifier in the set. One must guarantee object identifier uniqueness. For performance reasons, identifiers have to be created without any network request. A node must be able to create a unique identifier by

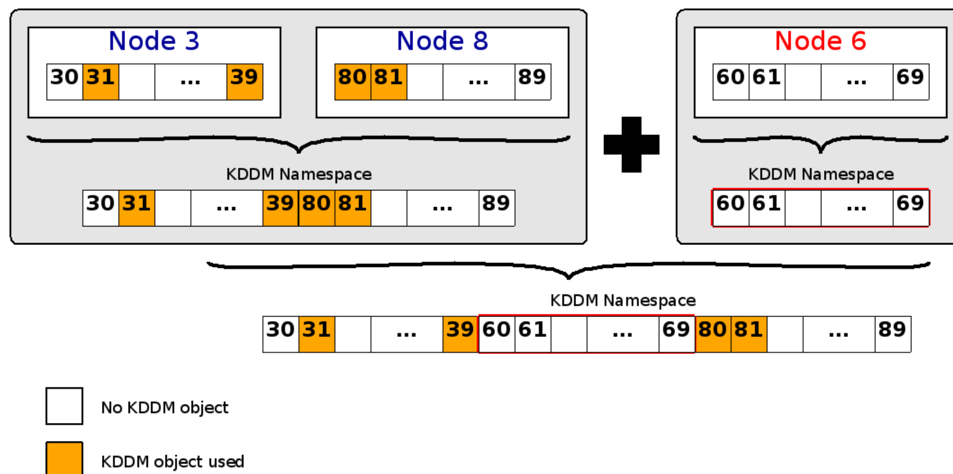

Figure 8: KDDM namespace extension on node addition

itself. Using the *node_id* in the formula used to generate identifiers is quite a good solution. (The formula used can be masking and can be different for each set). Of course, it means that the namespace available for each set on each node is reduced. The maximum numbers of node in Kerrighed is 256. Thus 1 byte (8 bits) is needed for the *node_id*. As identifiers are coded as `long` variables, the maximum number of unique identifiers per set that can be created on one node is $2^{8\times(sizeof(long)-1)}$, which equals to 2^{24} on x86 computer nodes and to 2^{56} on x86_64 computer nodes.

Node addition

When adding a fresh node n in a cluster, n does not host any KDDM object. Moreover, as namespaces of object identifiers are different for each node, KDDM objects can be created on the added node as soon as KDDM service is up without risking any conflicts with already allocated objects on other cluster nodes.

Figure 8 illustrates this idea with a simplified namespace limited to ten objects per node. Node 6 is added to a cluster of two nodes composed of node 3 and node 8. KDDM objects existing in the cluster are 31, 39, 80 and 81. Node 3 has created objects 31 and 39 and is their default owner, node 8 has created objects 80 and 81 and is their default owner. At the end of the addition operation, this is unchanged. Neither the default owner nor the probe owner chain have needed any update.

Node removal

When a node leaves the cluster, it may host some KDDM objects that must stay in the cluster. Those objects need to be moved to other nodes. The leaving node may also be the default owner of some objects. As the default owner computation takes into account the MEMBERSHIP_ONLINE vector, a default owner is guaranteed to exist after the node removal. Nevertheless, the probe owners chain may

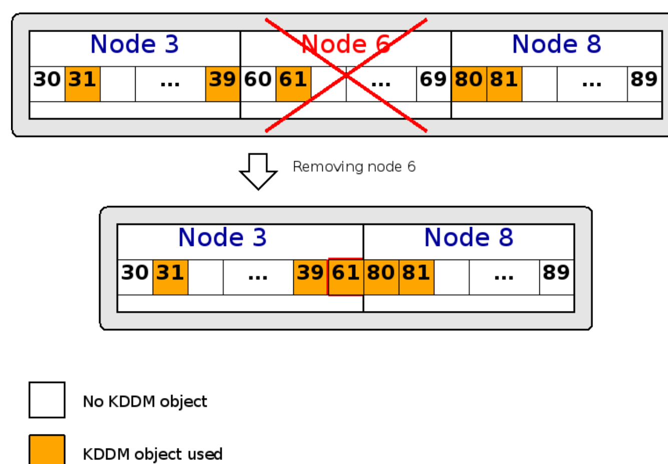

Figure 9: Recomposition of the KDDM namespace upon a node removal

be broken. That's why, during a node removal, on each node, the KDDM objects probe owner are checked and updated if necessary.

Figure 9 shows the KDDM namespace upon a node removal. After removal of node 6, no object will be created from its namespace but objects already created from its namespace are now linked to another node. For instance, the default owner of object 61 becomes node 3 after removal of node 6 but the object itself may be on node 8.

The hash function used to compute the default owner from the object identifier is not presented in this document because current implementation may changed to optimize default owner localization (trying to make default owner the real owner as often as possible). It is important to notice that all nodes agree on a same function per set of objects and that this function must only take into account nodes that are ONLINE (A node does not need to be in POSSIBLE state to access the KDDM service).

Reconfiguration events that are not initiated by the administrator are out of the scope of this document. In the event of reconfiguration triggered by failure, the probe owner chain of each object has probably to be checked. Further investigation is necessary and is planned when developing advanced reconfiguration mechanisms.

6.3 Reconfiguration of High Level Services

Nearly all services will require modification to handle failure, however, this is out of the scope of this document and will be studied later. We only deal here with the node(s) addition and eviction at the initiative of the system administrator.

6.3.1 Principle

Source code has been updated to use the membership information as described in Section 4.

Using membership functions and macros, iterating other possible nodes looks as follows:

```
1 kerrighed_nodeid_t n;
2 for_each_possible_krgnode(n)
3         do_some_stuff(n);
```

Node addition does not generally need reconfiguration on the cluster nodes, neither on the joining node. This is because there is no dependency from the joining node to the cluster nodes and conversely.

On the contrary, when removing a node, there are probably dependencies from this node to the other cluster nodes and from other cluster nodes to this node. During the reconfiguration operation, these dependencies must be removed.

There are exactly two kinds of services that are insensitive to reconfigurations:

1. some High Level services, such as the Ghost service, only provides an API used by other services. Those services are stateless and thus insensitive to reconfigurations.

2. services that do not use RPC interface but ONLY rely on the KDDM service are insensitive. Such services are rare because they do not use any local references (or these local references are managed by another service).

6.3.2 Impact on High Level Services

The current section sorts the services in 2 kinds: those that are sensitive to reconfigurations and those that are not.

Services insensitive to reconfiguration

The global scheduler, Ghost and MM services do not require any modification for basic reconfiguration events as explained bellow.

- **Global Scheduler** is implemented as a set of probes in the kernel and policies in userspace [5]. Local probes automatically discover the arrival or departure of resources and then the scheduling policy has to adapt to these changes in the cluster configuration. So policies must be designed taking the dynamic presence of resources into account.

- **Ghost** does not make use of membership information or KDDM object.

- **MM** manages process memory. It strongly relies on the KDDM service. The memory is either linked to one process or shared by multiple processes. In any case, the EPM service should decide what to do with the processes.

Thus, the MM service does not need to be notified of reconfiguration operations.

- **Proc** manages processes and strongly relies on the KDDM service. Different data structures related to process are stored in KDDM sets: `struct task_kddm_object`, `struct children_kddm_object`, `struct sighand_struct_kddm_object`, `struct signal_struct_kddm_object`, `struct pid_kddm_object`. During a removal operation, the EPM service decides if processes must be migrated, checkpointed or killed.

Services sensitive to reconfiguration
Other services have to be updated, mainly to deal with node removal operation:

- **FAF** forwards access to remote by file. By definition, it creates dependencies between nodes. Reconfiguration mechanisms are not yet implemented nor studied for the FAF service.

- **IPC** handles IPC message queues, System V semaphores and System V shared memory segment. Those object are global to the cluster and any process can use them. Reconfiguration mechanisms are not yet implemented nor studied for the IPC service.

- **KDFS** [6] provide a distributed file system. It means that files can be splitted on many nodes. Managing removal operation is quite complex as you may need to migrate pieces of files. The reconfiguration mechanisms for the distributed file-system are advanced features of the XtreemOS system requiring further investigation.

- **EPM** has to decide which processes could/should be migrated, checkpointed or killed, regarding effective capacities, file dependencies over FAF (File Access Forwarding), memory sharing, etc. Reconfiguration mechanisms are already implemented for the EPM service but need to be refined.

- **ProcFS** implements the global `/proc` directory. Some files (such as `/proc/cpuinfo` or `/proc/meminfo`) are dynamically filled when reading it. The ProcFS services maintains a list of files `/proc/nodes/*` that refer to cluster nodes. Those files are created/removed in the event of a node(s) addition/removal. Reconfiguration mechanisms already take these files into account.

  The ProcFS service maintains a list of `/proc/<pid>` files with one file per process. This is currently done by scanning the local pid bitmap of all cluster nodes. A process is visible by its creation node. Thus, if a process p is migrated from node A to node B and then node A is removed, `/proc/<pid of` p`>` does not exist. It must be fixed to fully support reconfiguration.

7 Conclusion

During the work on the reconfiguration mechanisms, in collaboration with Kerlabs INRIA spin-off that maintains Kerrighed, we have designed a simple interface for administration. We have also designed the Hotplug service, a framework that handles hot node(s) addition/removal and coordinates services reconfiguration. Reconfiguration of High Level services as well as reconfiguration of the KDDM service and low level services have been studied. Some services are not impacted, others need specific action in the event of a reconfiguration. They register to the Hotplug service that calls functions provided by services. These functions are called in a predefined order to take dependencies between services into account.

The administration interface is implemented but needs to be completed to give more feedback to the administrator. The Hotplug service is nearly finalized, thus the framework needed to coordinate reconfiguration operations is available. Reconfiguration of KDDM service is working in most cases but can lead to bug in corner cases. Reconfiguration of High Level services needs additional work. Services IPC, kDFS and FAF have not been studied yet. Service ProcFS needs extra work. Reconfiguration mechanisms are already implemented for the EPM service but need to be refined.

Node addition is working pretty well and the global scheduler already takes new nodes into account. Thus, the load is well-balanced. Node removal works in very simple case but needs more High Level services support.

In the short term, the roadmap will focus on stabilization of basic reconfiguration mechanisms and especially node removal. In the meanwhile, we plan to add support for reconfiguration mechanisms for FAF, ProcFS, kDFS and IPC. In the mean term, we plan to study and add mechanisms to handle node or network link failure.

References

[1] Cpu hotplug support in linux(tm) kernel. `http://www.kernel.org/doc/Documentation/cpu-hotplug.txt`.

[2] Tipc open source project website. `http://tipc.sourceforge.net/`.

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

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

[5] XtreemOS consortium. Design and implementation of a customizable scheduler. Deliverable D2.2.6, November 2007.

[6] XtreemOS consortium. Design and implementation of high performance disk input/output operations in a federation. Deliverable D2.2.5, November 2007.

[7] 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.

[8] Kerrighed website. `http://www.kerrighed.org`.

[9] 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.

[10] 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.

[11] 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.

[12] 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.