Project no. IST-033576

XtreemOS

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

First prototype of a standalone KDDM module
D2.2.9

Due date of deliverable: November 30th, 2008
Actual submission date: December 2nd, 2008

Start date of project: June 1st 2006

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

Responsible institution: NEC
Editor & and editor's address: Erich Focht
NEC Deutschland GmbH
Hessbruehlstr. 21b
70565 Stuttgart, Germany

Version 1.0 / Last edited by Erich Focht / December 2, 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	10/10/08	Erich Focht	NEC	Initial template + Outline
0.2	30/10/08	Erich Focht	NEC	Content complete
0.3	06/11/08	Erich Focht	NEC	Incorporated corrections from reviewers
0.4	28/11/08	Erich Focht	NEC	Switched instructions to 2.6.27 kernel
1.0	02/12/08	Erich Focht	NEC	Ready for submission

Reviewers:
Felix Hupfeld (ZIB), Marko Novak (XLAB)

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T2.2.7	First prototype of a standalone KDDM module	NEC*

°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 Structure **5**
2.1 KDDM in Kerrighed . . . 5
2.2 KDDM Standalone . . . 6
2.2.1 Tools . . . 6
2.2.2 RPC . . . 6
2.2.3 Hotplug . . . 6
2.2.4 KDDM . . . 7

3 Developments **9**
3.1 Build system changes . . . 9
3.2 Forward porting to latest kernels . . . 10
3.3 Module unloading . . . 10
3.4 Hotplug layer changes . . . 10
3.4.1 TIPC subscription reporting . . . 11
3.4.2 Node states . . . 11
3.4.3 State transitions . . . 12
3.4.4 Notification mechanism . . . 12
3.4.5 Cluster autostart . . . 13
3.5 KDDM benchmark . . . 13
3.6 Flexible I/O linker and RPC IDs . . . 13
3.7 Applications . . . 14

4 User Guide **17**
4.1 Building KDDM . . . 17
4.1.1 Building the Kernel . . . 17
4.1.2 Building KDDM and Utilities . . . 18
4.2 Starting KDDM . . . 18
4.3 Stopping KDDM . . . 19

5 Conclusion and Future Work **21**

Chapter 1

Introduction

KDDM (Kernel Distributed Data Management) is a concept providing a very natural method for creating distributed kernel services. It is a layer that allows cluster-wide sharing of kernel data objects. KDDM is a very flexible and configurable distributed shared memory system that uses a strict consistency model (invalidate on write). It was introduced under the name Containers and Gobelins in [1] and [2]. The Kerrighed SSI (single system image) operating system [3], [4] uses it extensively for providing various kernel services access to distributed shared objects in the cluster.

KDDM can be used outside Kerrighed SSI for programming distributed services in a new and easy way: By starting with the well-tested code of non-distributed services and putting the relevant variables into KDDM sets (i.e. making them accessible cluster-wide). This approach relieves the programmer of the need to explicitly take care of synchronization, distributed locking and managing the data transport between the cluster nodes.

The *KDDM standalone* project is an effort to split out the base infrastructure on which Kerrighed was built and make it usable for a larger user-base. It is implemented as a set of kernel modules which build against a (more or less) unmodified kernel. Currently the kernel needs to be rather new (2.6.25 and above) and include TIPC [5] version 1.7.5 or newer. KDDM standalone doesn't require any own patches to the kernel.

The target of this subproject is to make *KDDM standalone* a true stand-alone component which is clean enough to eventually be proposed to the kernel developer community for inclusion. An assessment of its state showed that several changes and developments are necessary:

- Restructuring the code and adapting the build system.
- Forward porting to newer kernels.
- Cleaning up the hotplug functionality and basing it entirely on TIPC functionality available in the Linux kernel.
- Allowing applications using KDDM functionality to allocate and free IDs for I/O linkers and RPCs in flexible manner.
- Enable modules to load/unload cleanly.

Not all developments were finished, this document presents the state of *KDDM standalone* after month 30 of the XtreemOS project.

The following chapter (2) describes the structure of the core components needed for KDDM, and the modular splitting of the functionality. Chapter 3 describes developments that have been done for KDDM standalone. Chapter 4 is a guide for users, explaining how to build, install and start KDDM standalone.

Chapter 2

Structure

2.1 KDDM in Kerrighed

Kerrighed SSI consists of a rather big kernel patch (currently for kernel version 2.6.20) and the modules subdirectory that builds to a monolithic kerrighed module. The source files in the modules directory are spread in well structured way over a set of directories reflecting their functionality, for example: hotplug, rpc, ctnr, proc, ghost, procfs, mm, scheduler, ipc, epm, etc... These are the distinct components of the layered architecture of Kerrighed sketched in fig. 2.1.

Unfortunately the modular structure of the code does not translate into a similar structure of kernel modules. Building Kerrighed leads to one single large monolithic module containing all functionality. Although the basic infrastructure of Kerrighed could be used separately, for non-SSI applications, they cannot be split out easily from the whole Kerrighed code.

Building monolithic modules is straight forward and spares the effort of cleaning up the interfaces between the modules and deciding which symbols need to be used from other modules and need to be exported with EXPORT_SYMBOL. A monolithic module also hides circular dependencies between functional parts, which again can lead to unclean design.

In a Kerrighed SSI cluster there is a very deep interaction between the kerrighed system and some Linux kernel components. Once the SSI is started some variables like process IDs become global and have cluster-wide meaning. There is no clean way

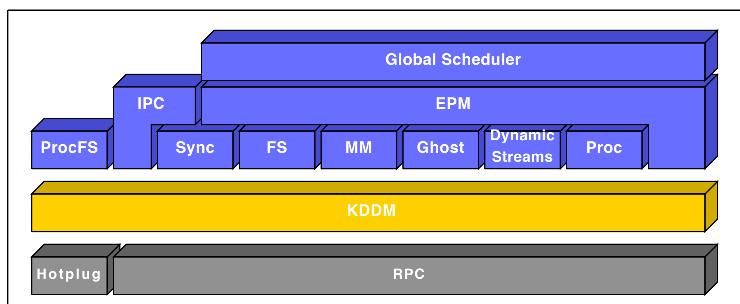

Figure 2.1: Kerrighed layered architecture.

implemented to reverse the starting of a SSI cluster and make the kernel return into the state of a stand-alone node. Therefore unloading a Kerrighed module makes no sense, currently. The only way to switch off the SSI mode of a node is to reboot it.

2.2 KDDM Standalone

The target for *KDDM standalone* is to have a separately buildable set of modules that can be easily forward ported to new kernel versions and require no patching of the base kernel, ultimately easing the porting of Kerrighed to newer kernels. At the same time the cleaned up and restructured code should enable us to propose the basic infrastructure components of Kerrighed for integration into the main line kernel.

The standalone KDDM infrastructure consists of several software layers. Each of the layers has its own source directory and builds an own loadable kernel module. The following sub-sections each describe one component module belonging to the KDDM standalone code.

2.2.1 Tools

The tools layer is the lowest one and contains helper functions and structures used by all higher layers, for example hashtables, unique_ids, ioctl interface services registration. It builds the module krg-tools.ko. This module is cleanly unloadable.

2.2.2 RPC

The communication infrastructure is built of:

- comlayer: the basic communication layer which uses TIPC and provides the basic mechanisms for exchanging data packets among the cluster nodes,
- rpc: an RPC-like infrastructure for registering and dispatching services
- rpclayer: a communication abstraction using pack/unpack transactions for data transport.

This part builds the module krg-rpc.ko. Before the RPC module is loaded the TIPC network **must be** configured! In order to keep the KDDM standalone modules free from the need of kernel patches, the kerrighed node addressing configuration is done purely by using TIPC node addresses. If TIPC is unconfigured when the krg-rpc module is loaded, the load will fail "uncleanly", requiring a reboot of the machine before trying to load that module again.

2.2.3 Hotplug

The hotplug layer is taking care of cluster node membership management, node addition and removal and invocation of notifiers and callback functions that react to cluster changes. This components lives in the module krg-hotplug.ko.

2.2.4 KDDM

The layer providing the distributed shared memory functionality builds into the module krg-kddm.ko. It provides the infrastructure for creating KDDM sets, the cluster-wide containers of shared objects of a certain size, components for registering specialized I/O linkers which can extend the behavior shared objects in various ways, the infrastructure for managing shared objects locking, copying and cluster-wide coherence, and more.

The KDDM layer's functionality is complex and a full description is not the scope of this paper. A good entry point to KDDM programming is the paper "Reducing Kernel Development Complexity in Distributed Environments" [6], which also describes the kDFS parallel filesystem design and it's implementation with the help of KDDM.

Chapter 3

Developments

3.1 Build system changes

The code was restructured to allow building in a separate directory, but also to allow easy merging into a kernel tree. All pieces of code that are compiled directly into the kernel with Kerrighed have been moved out and concentrated in the **tools** module:

- hashtable.c: infrastructure for hashtables,
- krgsyms.c: infrastructure for key-value pairs,
- krg_syscalls.c: main user-space interface through flexibly registrable ioctls for a virtual file: /proc/kerrighed/services,
- krg_sysfs.c: sysfs interface,
- procfs.c: tools for manipulating KDDM related virtual files in /proc,
- unique_id.c: mechanism for generating unique IDs in the cluster.

The ctnr subdirectory was renamed to kddm. The Makefiles have been adapted to build modules in the subdirectories tools, rpc, hotplug and kddm. In the subdirectory utils we collected tools for setting up, configuring and accessing the KDDM standalone cluster:

- kddmadm: basically a slightly adapted version of the krgadm tool from Kerrighed. It's main role is to show the online nodes and their status, and trigger state changes for testing. It is not needed any more for starting a cluster.
- tipc-config: the TIPC tool for configuring the network. It is available in two versions, the choice is done automatically depending on the TIPC version in the kernel.
- tipc-auto.sh: configures the TIPC network, is invoked automatically if modprobe.conf for the krg-rpc module is configured correctly.

Code was added to enable the creation of separate modules for tools, rpc, hotplug and kddm. Symbols declared in one module and used in others had to be exported with the EXPORT_SYMBOL macro. Some code needed to move from one module to the

other because it was logically not belonging to that module and creating circular dependencies. A potential circular dependency still exists with the current structure where **hotplug** uses functions from **rpc** and **rpc** actually would need to register functions into **hotplug** data structures. This will be solved by merging the two modules, currently the reaction of the rpc layer to hotplug events was disabled to avoid the circular dependency.

3.2 Forward porting to latest kernels

Some work went into decoupling the KDDM code from the Linux kernel. This was aimed at making the code easy to build outside the kernel tree, but also to make it more portable across kernel versions. The exercise of forward porting was made from the original kernel version 2.6.20 to the versions 2.6.24, 2.6.25.4 and 2.6.27.4. With the current setup the effort is minimal. It mainly requires three steps:

1. Find and apply patches for the kernel debugger KDB (if needed and wanted), they can be downloaded for the required architecture from the site ftp://oss.sgi.com/www/projects/kdb/download/v4.4/.
2. Download and apply patches for the latest TIPC version from http://tipc.sf.net. The latest version is 1.7.6. Instructions and patches for TIPC 1.7.5 and 1.7.6 are in the KDDM standalone svn repository.
3. Create an appropriate .config file for the kernel. It will need to have TIPC support and enable KDB, if that was added.

After building the kernel normally KDDM standalone should build fine if no major kernel APIs have changed.

3.3 Module unloading

The functions for starting and stopping the modules' functionality were available in the Kerrighed code, but most of the ones for stopping (functions with names ending with _cleanup) were just empty functions. Work has been started to write the code for stopping the single modules but the only ones implemented to date are for the "tools" module and a part of the rpc module.

3.4 Hotplug layer changes

The role of the hotplug component is to manage node states, be able to switch between node states and notify components which registered for such notifications and react to state changes. The hotplug layer also manages membership to the cluster in which KDDM sets can be shared.

The Kerrighed implementation contains the notion of cluster and subcluster. Obviously the subcluster was thought as a subset of nodes or a different administration domain. Subclusters were not fully implemented and not functional when work on KDDM standalone has started. The author's inquiries for information about the purpose and the targetted design for Kerrighed subclusters were not answered by the Kerrighed core developers, therefore the notion of subcluster was ignored in the devel-

opment of KDDM standalone. Over a long term the variables related to subclusters should be stripped out of the code.

3.4.1 TIPC subscription reporting

In Kerrighed the TIPC layer was patched to call a special handler when nodes appear on the TIPC network. These nodes don't even need to have their TIPC layer fully configured for handling Kerrighed RPC events, they simply are detected as present in the cluster.

KDDM standalone's requirement for unpatched kernel lead to a design change in the reactions to nodes appearing or vanishing from the cluster. These membership and state change events are now handled by native TIPC mechanisms. In TIPC a published port has attached to itself a scope: the port can be valid only locally (on the local node), or valid for the entire cluster or even zone. In KDDM standalone we use two cluster-wide ports on each node for announcing the state of a node to the entire cluster. Each node subscribes to TIPC's subscription reporting service and monitors ports particular ranges. A handler will be invoked when ports appear on the cluster or when ports are disappearing. This way state changes are tracked without the need to send any explicit state change messages to every node. This way of tracking state changes is more elegant and reliable than the one used in native Kerrighed.

Though the used mechanisms were available in TIPC since long time, their implementation was buggy. Starting with TIPC version 1.7.5 the subscription notification mechanisms are working reliably, therefore this is the minimal TIPC version required for KDDM standalone.

3.4.2 Node states

The meaning of node states is similar to that in Kerrighed, though slightly changed and simplified. The names of states are inspired by the CPU hotplug infrastructure in the Linux kernel:

- "offline": node is physically present, has a TIPC address, but no Kerrighed RPCs are active. In this state the node would reply to pings and might execute user space commands. It does not need to have any kerrighed modules loaded. In this state the node is of no use for the cluster.

- "online": node has loaded the krg-rpc module, has published its TIPC port related to Kerrighed services. This state can be recognized remotely by TIPC nodes which check for the published Kerrighed RPC ports. All "online" nodes see each other. The published port has the address: <TIPC_KRG_SERVER_TYPE, kerrighed_node_id>.

- "possible": the cluster services have been started. For KDDM standalone this means nothing more than KDDM is ready to go. This state can be recognized remotely by TIPC nodes which check for the published Kerrighed RPC ports. Each node in p̈ossibles̈tate opens a cluster-wide visible TIPC port with the address: <TIPC_KRG_BASE, kerrighed_node_id>.

The node states are stored and tracked locally on each node in the bitmaps krgnode_online_map and krgnode_possible_map.

3.4.3 State transitions

Node state transitions can be initiated in following ways:

offline -> online

By loading the krg-rpc module the node's TIPC communication port for RPCs is published cluster-wide. On every node with activated hotplug this triggers an ADD_ONLINE notification. No API for manual addition is needed.

online -> possible

By calling hotplug_announce_possible() from handle_start_cluster(). This publishes a dummy TIPC port on every server with the cluster-wide address <TIPC_KRG_BASE, kerrighed_node_id>. Cluster start can be initiated by user intervention: "kddmadm cluster start".

possible -> online

Triggered by hotplug_revoke_possible() called from handle_cluster_stop(). When the node fails (is rebooted or powered off) its TIPC port marking the "possible" state is implicitly withdrawn, so all nodes see the state change. The state switching can be initiated from userspace by "kddmadm cluster stop -n <node_number>".

online -> offline

Triggered by removal of krg-rpc module, which implies that the main TIPC port is disconnected. This is seen on all nodes as withdrawal of the publication of this node and triggers on every node the notification mechanisms.

possible -> offline

Special case when node fails suddenly. In that case both REMOVE_ONLINE and REMOVE_POSSIBLE notifications appear at almost the same time, in arbitrary order. If the REMOVE_ONLINE notification appears before the REMOVE_POSSIBLE, then it will first generate a REMOVE_POSSIBLE notification.

3.4.4 Notification mechanism

Any component of KDDM standalone (and applications of it) can react to state change messages of nodes by registering handlers with the hotplug subsystem. These mechanisms were designed and implemented as part of the XtreemOS deliverable D2.2.4 [7]. The API allows for:

- registering a handler

```
int register_hotplug_notifier(
            int (*handler)(struct notifier_block *,
                           hotplug_event_t, void *),
            int priority);
```

- unregistering a handler

```
void hotplug_notifier_unregister(
            int (*handler)(struct notifier_block *,
                           hotplug_event_t, void *));
```

Handlers are actually put into two kernel notifier chains: hotplug_chain_add and hotplug_chain_remove.

It is important that state changes trigger handlers in well-defined order, for example a transition from “possible” to “offline” state must first lead to a transition from “possible” to “online” state, then from “online” to “offline”. On SMP machines the handlers

should not be invoked in parallel. This is ensured by serializing notification handling and letting them happen in the context of one special worker thread. This change in the hotplug logic was done as part of the KDDM standalone development, the subject of this deliverable.

3.4.5 Cluster autostart

The use of TIPC for managing cluster membership and node states allowed easilly to implement a cluster "autostart" functionality and get rid of the need to start the KDDM functionality explicitely.

Once the krg-rpc.ko module is loaded, the node is recognized by the rest of the cluster as node in the "online" state which is ready to do handle basic RPC requests.

When the krg-kddm.ko module was loaded its initialization process ends with announcing to the other nodes the reaching of the "possible" state. All nodes in "possible" state are able to share data of KDDM sets.

Unfortunately KDDM sets are not yet able to handle cleanly the removal of nodes in "possible" state. KDDM objects owned by the removed node are lost.

3.5 KDDM benchmark

The KDDM benchmark serves as a built-in example for using KDDM sets, which can be used to easily start coding own applications but also should be used to test the functionality of the KDDM standalone modules. In Kerrighed this benchmark was a quick hack with hardcoded node IDs and required to have at least four nodes available, with node IDs 1, 2, 3 and 4. The test wasn't running long enough to accumulate decent statistics and was leaving behind a few KDDM sets. The available code was modified to allow any node IDs, autodetect them and run the benchmark if two or more nodes are found in the cluster.

The KDDM benchmark can be invoked by reading a virtual file in the /proc filesystem:

```
# cat /proc/kerrighed/kddm/bench
```

The execution takes time, it can be around 30s, depending on the used hardware. It will create temporarily three KDDM sets, which will be used to transfer objects between the test nodes. The test runs in kernel space and cannot be interrupted. At the end simple statistics about the runtime are printed as content of the virtual file.

3.6 Flexible I/O linker and RPC IDs

An application which wants to use KDDM sets and creates its own I/O linkers needs to allocate I/O linker IDs which are known cluster-wide and are not used by any other application. If the application needs to register some own RPCs the same problem applies: the IDs of the RPCs must be known cluster-wide and cannot be hardcoded in the application (because another application could use those IDs as well). In Kerrighed these IDs are hardcoded in include files, because the SSI system is closed and monolithic. No external application is expected to use the infrastructure. In *KDDM standalone* applications of KDDM are supposed to work on top of the infrastructure, without the need to change any include files and recompile it.

KDDM standalone implements flexible I/O linker and RPC IDs. The IDs are stored in KDDM sets, that means they are globally visible to the cluster, and have strings associated with them. The strings can be used by distributed applications to reserve an ID and find if it has already been reserved from the same application started on another node. The API is simple and uses global locking semantics implemented also through the KDDM sets.

Users (programmers of kernel modules which use kddm features) have a simple API:

```
int get_iolinker_id(char *name);
void put_iolinker_id(int id);
int get_krgrpc_id(char *name);
void put_krgrpc_id(int id);
```

The names can have up to 16 characters.

An usage example: When an application needs to register an I/O linker, it should now decide for a name for it. Then the module with the application should reserve the name and get an I/O linker ID by

```
id = get_iolinker_id(name);
```

This creates an object in a KDDM-set, such that the linker ID associated to the linker name is visible in the whole cluster. The first module that registers the name creates the object, all other modules will see the registered name and use its ID.

When unloading a module which has registered a global ID, this should be freed by calling

```
put_iolinker_id(id);
```

The global ID will stay valid until all users have freed it.

3.7 Applications

As mentioned before, KDDM standalone is an infrastructure to provide distributed shared object management for Linux kernel components. As for any infrastructure, it's value is in the applications that uses it. KDDM has demonstrated its usefullness as core component of Kerrighed, but more simpler applications that are less demanding with respect to kernel patches would contribute to KDDM's popularity. Inside KDDM standalone there are two examples on how KDDM sets should be used: the global IDs and the KDDM benchmark.

A great external application is the symmetric parallel filesystem kDFS [6]. It allows the aggregation of file space from cluster nodes into a parallel filesystem for the cluster by using KDDM sets as containers for sharing inodes, dentries and pagecaches.

kDFS was ported from Kerrighed to the standalone KDDM framework. This allows using kDFS without running the complete Kerrighed operating system but the much smaller KDDM kernel module. It also shows the ability of the standalone KDDM framework to support full-blown applications.

While porting kDFS to the standalone KDDM framework some issues were encountered: First, since the KDDM module is written for a much more recent kernel than Kerrighed's kernel (2.6.20), kDFS had to be adapted to the new VFS infrastructure (for example, the `iget` function was removed and replaced by `iget_locked`, `read_inode` isn't a superblock operation anymore, the `struct path` changed, etc.).

Secondly, kDFS was developed tightly with Kerrighed. As a result it used internal KDDM functions, which are not exported to the outside world. Since the standalone KDDM module and the kDFS are separated, these internal functions are not available to kDFS anymore. Also, kDFS uses a kernel function (`remove_from_page_cache`) exported by the Kerrighed kernel patch (because the Kerrighed module needs this function). This function in not available using the standalone KDDM module. The current fix is to ship kDFS with its own version of the KDDM framework with a few additional exported symbols and its own kernel with the function `remove_from_page_cache` exported. A better fix for these issues will hopefully be found soon.

kDFS makes use of the new flexible I/O linker ID framework offered by KDDM standalone, removing the need to hardcode IDs in include files.

The standalone kDFS version is developed in its own SVN branch. To download it, use:

```
# SITE=https://scm.gforge.inria.fr
# svn co $SITE/svn/kerrighed/branches/kdfs-standalone
```

The compilation and installation procedures are documented in the README file in the top directory.

More applications planned for the next future are a distributed lock manager based on shared lock objects, as well as a simple data sharing facility with virtual files representing memory from the nodes of the cluster. For both applications the in-kernel implementation is almost trivial with KDDM standalone, the bigger challenge being represented by the selection and implementation of the interface to user space.

Chapter 4

User Guide

4.1 Building KDDM

4.1.1 Building the Kernel

1. Download the kernel from ftp.kernel.org. Suppose we do all following actions in /home/src:

```
# cd /home/src
# wget ftp://ftp.kernel.org/pub/linux/kernel/v2.6/linux-2.6.27.4.tar.bz2
```

2. Unpack the kernel:

```
# tar xjvf linux-2.6.27.4.tar.bz2
```

3. Download the KDDM standalone source: let's suppose we continue using /home/src as base directory for the build. The kddm source tree can be checked out to the directory /home/src/kddm with:

```
# cd /home/src
# svn co https://scm.gforge.inria.fr/svn/kerrighed/branches/kddm/trunk kddm
```

4. Check out the patches svn directory:

```
# svn co https://scm.gforge.inria.fr/svn/kerrighed/branches/kddm/patches
```

5. Patch the kernel with the KDB kernel debugger. You can skip this step but if you don't need kdb.

```
# cd linux-2.6.27.4
# bzip2 -dc ../patches/2.6.27.4/kdb-v4.4-2.6.27-common-1.bz2 | patch -p1
# bzip2 -dc ../patches/2.6.27.4/kdb-v4.4-2.6.27-x86-1.bz2 | patch -p1
# cd ..
```

6. Patching the kernel with the latest TIPC version. (This step is very much recommended as 2.6.25.4 comes with the quite obsolete tipc version 1.6.3. The 2.6.27.4 kernel has also a pre-1.7.0 TIPC version. The required version is 1.7.5 and hotplug will only work with that or newer versions.)
Remove the old TIPC version from the vanilla kernel and install the new one:

```
# cd linux-2.6.27.4
# rm -rf net/tipc include/linux/tipc* include/net/tipc
# tar xzvf ../patches/2.6.27.4/tipc-1.7.6.tar.gz
```

7. Configure and build the kernel

```
# cp ../kddm/utils/configs/linux-2.6.27.4-i386.config .config
# make oldconfig
# make -j 3
```

8. Install the kernel and modules to the appropriate place. In the example setup we are using an exported NFS read-only root directory for the KDDM cluster nodes, which is overlayed and assembled to some usable root directory with the help of the tools from http://diskless-imager.googlecode.com. Therefore the target directory for the kernel is /tftpdir and the NFS root content is in /tftpboot/nfsro. The kernel installation is therefore:

```
# cp arch/x86/boot/bzImage /tftpboot/vmlinux-2.6.27.4-kddm
# make INSTALL_MOD_PATH=/tftpboot/nfsro modules_install
```

4.1.2 Building KDDM and Utilities

1. Build the KDDM modules and utilities:

```
# cd ../kddm
# make KSRC=`pwd`/../linux-2.6.27.4
# make KSRC=`pwd`/../linux-2.6.27.4 utils
```

2. Installing (as root):

   (a) ... to the "own" filesystem:

   ```
   # make KSRC=`pwd`/../linux-2.6.27.4 install
   ```

   (b) ... to another root filesystem (the NFS root, for example):

   ```
   # make KSRC=`pwd`/../linux-2.6.27.4 DESTDIR=/tftpboot/nfsro install
   ```

4.2 Starting KDDM

Loading the modules

You'll need to have the TIPC address configured for the node when loading the krg-rpc module! This can happen automatically by letting modprobe run the tipc-auto.sh script.

Create a file /etc/modprobe.d/krg_tipc with the content:

```
#--- file: krg_tipc
install krg-rpc /usr/bin/tipc-auto.sh && modprobe --ignore-install krg-rpc
#--- end of file
```

This will execute the tipc-auto.sh script before loading the krg-rpc module. tipc-auto.sh is loading the tipc module and setting the TIPC address according to the first found IP address or according to the command line options. The command:

```
utils/tipc-auto.sh [-s|--sessid <SESSION_ID>] [-n|--nodeid <NODE_ID>] \
                   [-c|--clusterid <CLUSTER_ID>] \
                   [-z|--zoneid <ZONE_ID>]
```

initializes the TIPC address with either the passed IDs or with the <1,1,N>, where N is autodetected from the first network interface which has an IP set.

The session ID is the TIPC network identifier, by default set to 4711. Only nodes with the same network identifier will be able to communicate, therefore different network identifiers can help running multiple clusters on the same network.

KDDM can be started by simply loading the krg-kddm.ko module with the command:

```
# modprobe krg-kddm
```

This will load all modules it depends of: krm-rpc, krp-hotplug, which will trigger the load of krg-tools. Before loading krg-rpc the TIPC network will be configured, the initialisation of krg-rpc will announce the node's availability in "online" state to the cluster. When krg-kddm is finally loaded, the node will switch to "possible" state and is fully usable for distributed shared objects.

4.3 Stopping KDDM

In theory stopping a KDDM node should be handled by unloading the applications using the KDDM layer and then unloading the KDDM modules. In the current state this doesn't lead to a clean shutdown of KDDM. The module unloading will not be entirely successful and will not do a complete clean up. Simply shutting down a node will reliably take it out of the KDDM cluster and all shared objects "owned" by the node will be removed from the cluster. A clean shutdown and improved behavior when stopping nodes are high priority development targets for KDDM standalone.

Chapter 5

Conclusion and Future Work

KDDM standalone has been successfully split out of Kerrighed and provides infrastructure for creating, managing and accessing distributed shared objects from inside the kernel, for building in-kernel cluster-wide services. This enables a programing paradigm that simplifies substantially coding distributed services at the operating system level [6]: entities that need to be shared are put into KDDM sets and can be accessed from any node of the cluster, with no need to explicitely code communication or locking protocols.

The standalone KDDM functionality can now be built as a set of four kernel modules for kernels newer than 2.6.25. The Linux kernel still needs to be patched to contain a version of TIPC equal or newer than 1.7.5, but no other kernel patches are required.

Several developments were needed for providing the isolated functionality in a way that can be used by external applications of the KDDM functionality. A change of the hotplug logic managing the membership of cluster nodes and the introduction of globally shared IDs for I/O linkers and RPCs were the biggest ones. Some developments are unfinished and will be done in near future.

One very interesting application was ported from Kerrighed clusters to run on top of KDDM standalone: kDFS, the symmetric parallel filesystem using shared objects. Two more distributed applications which should demonstrate the power of KDDM are planned: a distributed lock management and a simple distributed shared memory interface to user space.

Besides additional distributed kernel applications KDDM standalone still needs major improvements in several areas, thus leading to the following list of items for future work:

- Merge rpc and hotplug functionality into one module.
- Fixing module unload functionality is crucial for any attempt to push this to the linux kernel community.
- Improvement of behavior at node failure: when a node fails objects owned by it are deleted from the KDDM set. Even if the failure was due to a short network interruption, objects disappear and won't be recovered. This must be improved significantly and a method for recovering and re-parenting valid objects must be found. Current design ideas suggest versioning of KDDM objects might help.
- Add push functionality allowing to create replicated objects, such that objects would survive failures of nodes.

Bibliography

[1] 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.

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

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

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

[5] Tipc website. http://tipc.sourceforge.net. http://tipc.sourceforge.net.

[6] Adrien Lèbre, Renaud Lottiaux, Erich Focht, and Christine Morin. Reducing Kernel Development Complexity in Distributed Environments. In *14th International Euro-Par Conference (Euro-Par 2008)*, pages 576–586, 2008.

[7] XtreemOS consortium. Design and implementation of basic reconfiguration mechanisms, January 2008.