

Reducing Kernel Development Complexity in Distributed Environments ^{*}

Adrien Lèbre¹, Renaud Lottiaux², Erich Focht³, and Christine Morin¹

¹ IRISA, INRIA Rennes Bretagne Atlantique, France

² Kerlabs, Gevezé, France

³ NEC, Stuttgart, Germany

Abstract. Setting up generic and fully transparent distributed services for clusters implies complex and tedious kernel developments. More flexible approaches such as user-space libraries are usually preferred with the drawback of requiring application recompilation. A second approach consists in using specific kernel modules (such as FUSE in Gnu/Linux system) to transfer kernel complexity into user space.

In this paper, we present a new way to develop kernel distributed services for clusters by using a cluster wide consistent data management service. This system, entitled kDDM for “kernel Distributed Data Management”, offers flexible kernel mechanisms to transparently manage remote accesses, cache and coherency. We show how kDDM simplifies distributed kernel developments by presenting the design and the implementation of a service as complex as a fully symmetric distributed file system.

The innovative approach of kDDM has the potential to boost the development of distributed kernel services because it relieves the developers of the burden of dealing with distributed protocols and explicit data transfers.

1 Introduction

Clusters are today a standard computation platform for both research and production. A lot of work has already been done to simplify efficient use of such architectures: batch schedulers, distributed file systems, new programming models, . . . and it is likely to continue as cluster constraints are still changing: more cores per CPU socket, faster interconnects, larger scale.

Setting up generic and fully transparent distributed services for clusters implies complex and tedious kernel developments. More flexible approaches such as user-space libraries are usually preferred with the drawback of requiring application recompilations. However a lot of applications are mainly based on standards such as POSIX and recompilation is sometimes not possible (in particular for legacy codes). In such cases, distributed services have to be perfectly transparent, requiring kernel extensions. However, only few kernel mechanisms have been suggested. Current approaches consist in completing major kernel components by modules to bring back kernel complexity to user space. As an example, the FUSE module from the Gnu/Linux system makes distributed file system implementation easier. If such an approach solves the transparency issue, it impacts performance by the multiple copies from user to kernel and symmetrically. On

^{*} The authors from INRIA and NEC carry out this research work in the framework of the XtremOS project partially funded by the European Commission under contract #FP6-033576.

the other side, userland cluster services are designed by leveraging generic libraries such as MPI making their design and their implementation much easier.

In contrast with userland, only few work has focused on providing generic layers to facilitate distributed kernel services. From our best knowledge, developers are only aware about the remote procedure call protocol. The SUN RPC model [1] is based on a client server where a node (the client) asks for a service delivered by another node (the server). This model offers some flexibility but has several drawbacks. For instance, it only enables point to point communication and is not well designed to share data structures at a fine grain.

In this paper, we present a new way to design and implement kernel distributed services for Gnu/Linux clusters by using a cluster wide consistent data management service. From our point of view, providing such a service is really innovative. First, this system, entitled kDDM for kernel Distributed Data Manager, is built with the purpose to ease the design and the development of more complex distributed services. Second, it provides a real different way to exchange and share kernel data between distinct nodes within a cluster. By using the kDDM mechanism, programmers are able to focus on the real purpose of cluster kernel services instead of dealing with distributed protocols. The kDDM infrastructure helps reducing cluster kernel development complexity to a level comparable to the development on a SMP node.

We show how kDDM makes distributed kernel developments easier by presenting the design and the implementation of a service as complex as a fully symmetric distributed file system. This file system, named kDFS enables to:

- Aggregate storage resources available within a cluster,
- Provide a unique cluster wide name-space,
- Provide cooperative cache for both data and meta-data.

More generally, the use of kDDM could be exploited in almost all local kernel services to extend them to cluster scale. Cluster wide IPC, distributed namespaces (such as `/proc`) or process migration are only a few of the candidates.

The document is organized as follows. Section 2 outlines the kDDM mechanisms. Section 3 is focused on kDFS design and its implementation. Related work is addressed in Section 4. Finally, Section 5 concludes the document and gives some perspectives.

2 Kernel Distributed Data Manager

The Kernel Distributed Data Manager, kDDM [2], allows consistent and transparent data sharing cluster wide. This concept was formerly called *container* and has been renamed to kDDM to avoid confusion with current kernel container mechanisms.

The latest implementation of the kDDM service allows to share at kernel level collections of *objects* between the nodes of a cluster. In kDDM, an object is a set of bytes defined by the kDDM user (data structure, memory page content, etc). Objects of the same kind are stored in a *set*. An object is then identified using a pair (set identifier; object identifier).

The main goal of the kDDM mechanism is to implement distributed kernel services.

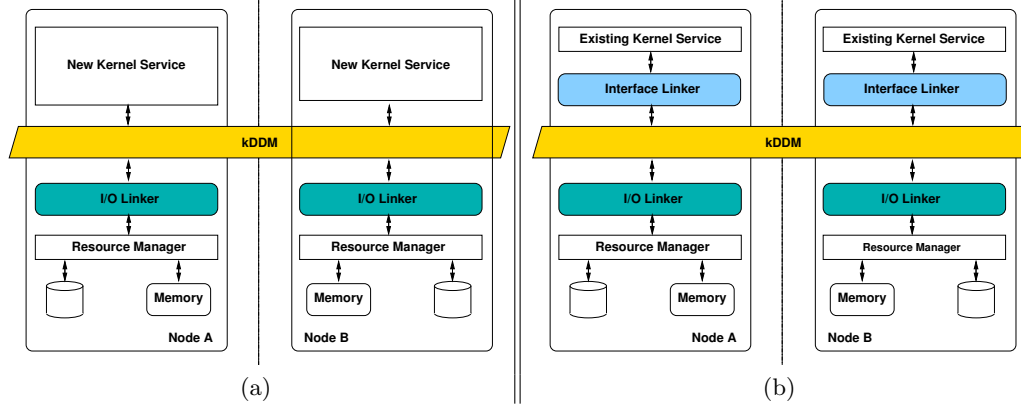


Fig. 1. kDDM overview

Assuming that an OS could be roughly divided into two parts: (1) system services and (2) device managers, developers could design and implement their own cluster wide services (cf. Figure 1 (a)) or extend the existing ones by using *interface linkers* (cf. Figure 1 (b)). In both cases, kDDM sets are plugged to device managers thanks to *IO linkers*.

2.1 kDDM Sets

A kDDM set is a collection of similar objects a kernel developer wants to share cluster wide. Each set can store up to 2^{32} objects. Objects are managed by the kDDM service without any assumptions on contents and semantics. In other words, developers have the opportunity to share any kind of objects.

For each kind of object to share, a new kDDM set family is created to host this kind of object. It is then possible to create several sets of the same family.

For instance, it is possible to define a kDDM set family to share clusterwide all pages of a system V memory segment. From this kDDM set family, we can create a new set for each system V segment.

Defining a new kDDM set family mainly consists in creating a new IO linker designed to manage the kind of object they are intended to host.

2.2 IO Linkers

Each kDDM set is associated to an *IO linker* depending on the family set it belongs to. For each family there is a different IO linker. During the creation of a new kDDM set, an IO linker is associated to it. Doing this instantiation, the set can be attached to a physical device or simply attached to a dedicated memory allocator (memory pages allocator, specific slab cache allocator, etc). Indeed, the IO linker defines how objects are allocated, freed, transferred from one node to another, etc.

Right after the creation of a kDDM set, the set is completely empty, i.e. it does not contain any data. Memory is allocated on demand during the first access to an object through the IO linker. Similarly, data can be removed from a kDDM set when it is destroyed, when an object is no more in use or in order to decrease the memory pressure when the physical memory of the cluster is saturated. Again, these operations are performed through the IO linker.

2.3 Interface Linkers

Existing kernel services can be extended to a cluster scale thanks to *interface linkers*. An interface linker is the glue between existing services and kDDM sets.

Moreover, it is possible to connect several system services to the same kDDM set by using different interface linkers. For instance, a kDDM set can be used to map pages of a file in the address space of a process P1 on a node A using an interface linker L1, while a process P2 on a node B can access the same kDDM set through a read/write interface using another interface linker L2.

2.4 Manipulation Functions

Objects are handled using manipulation functions which ensure data replication and coherence. These functions can be compared to read/write locking functions. They are mainly used to create *kDDM critical section* enabling to safely access kDDM objects regardless of data location in the cluster.

Objects stored in a kDDM set are handled using a dedicated interface. This interface is quite simple and mainly relies on three functions used to create *kDDM critical sections*: *get*, *grab* and *put*.

The *get* function is close to a *read-lock*: it places a copy of the requested object in local memory and locks it cluster wide. This locking ensures that the object can not be written on any other node in the cluster. However, concurrent read accesses are allowed.

The *grab* function is close to a *write-lock*: it places a copy of the requested object in local memory and locks it cluster wide. No other concurrent access (read or write) is allowed cluster wide (cf. Section 2.5).

The *put* function is used to unlock an object.

In addition to these main functions, a few other ones are used, such as *remove*, *flush* or *sync*. The *remove* function removes an object from the memory cluster wide since the *flush* function only removes an object from local memory and ensures that there is at least one node still hosting a copy.

Finally, the *sync* function synchronizes an object with its attached physical device, if any. This function is useful for instance to write back data to disk.

2.5 Replication and Coherence

During a kDDM critical section, object data is stored in the memory of the node doing a *get* or a *grab* and can be accessed using regular memory operations. Outside a kDDM critical section, there is no guarantee that the object is still present in node local memory. In most cases the data is still present, but the kDDM semantics does not guarantee coherency outside a kDDM critical section.

As suggested in the previous section, objects are moved from one node to another during a *grab* operation and can be replicated on different nodes for efficiency reasons during a *get*. Replication introduces a data coherence issues. Coherency is managed using an invalidation on write protocol, derived from the one presented by Kai Li [3]. The *grab* function, semantically equivalent to a write, is used to catch object modifications. In this case, all the existing remote copies are invalidated before the *grab* returns. During the *grab* critical section, the object is locked on the node (where the *grab* has been done) and cannot be moved or replicated.

3 Kernel Distributed File System

In order to show the interest of generic distributed kernel mechanisms such as the kDDMs, we chose to design and implement a service as complex as a fully symmetric file system. To our knowledge, only few distributed file systems have been designed with fully symmetric constraints [4,5]. The implementation complexity of such systems is generally dissuasive and the current trend consists in setting up distributed file systems composed by one or two meta-data servers and several I/O servers [6,7]. By dividing meta-data and data management, such solutions, entitled parallel file systems, make the implementation of the client stack easier. However such a model can lead to a non balanced architecture between distinct nodes. For example: the scalability limitation imposed by a single metadata server does not exist in a fully symmetric file system. Thanks to kDDM, we were able to quickly design and implement a symmetric file system, kDFS, providing a cooperative cache for both data and meta-data.

The evaluation of kDFS is behind the scope of this paper. Several parameters have been arbitrarily chosen and advanced mechanisms such as data-striping, redundancy or distributed I/O scheduling are addressed in a more technical document [8]. The innovative work, addressed in this article, consists in setting up such a fully distributed file system at kernel level without dealing with distributed protocols or consistency issues.

The design of kDFS relies on two main concepts:

- Using native file systems available on each node
- Using kDDM sets to provide a consistent cluster wide name space.

The use of kDDM sets facilitates the implementation of a global distributed cache with Posix semantics since kDDM mechanisms directly ensure meta-data and data consistency. Figure 2 illustrates the overall architecture of kDFS.

After describing how kDFS uses native file systems to store data, we present how kDDM mechanisms have helped kDFS be implemented with elegance and simplicity.

3.1 Disk Layout

To avoid block device dependencies and make storage management easier, we have chosen to build kDFS upon native file systems provided by the cluster nodes. As a consequence, each node has to specify if it takes part in the kDFS storage space or not. Storage space provided by a particular cluster node is considered as a kDFS partition only if it has been kDFS formatted. As kDFS

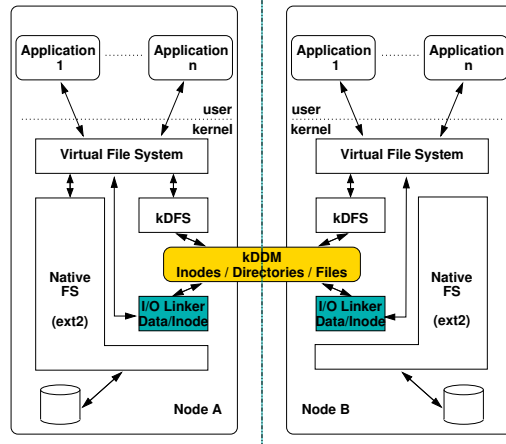


Fig. 2. Overview of the kDFS system

does not directly manipulate block devices, a kDFS partition actually refers to a local directory with a particular hierarchy. This section introduces different sub-directories and different files contained in a kDFS partition.

To format a directory which can be used afterwards in the kDFS storage space, administrators have to use the `mkfs.kdfs` command. This command takes two arguments: `DIR_PATHNAME` and `ROOT_NODEID`. The first one corresponds to the absolute path to store kDFS meta-data and data, the second one is the node identifier for the kDFS root entry (the node that stores the kDFS root inode).

`mkfs.kdfs` creates the kDFS "superblock" file (...) for the node. This file is stored on the local native file system in the given directory. If the current node identifier (last byte of the IP address) equals to the given *id*, `mkfs.kdfs` creates the root entry (the '/' of the file system).

Table 1 describes the creation of a kDFS structure over two nodes:

On node A: (nodeid = 1)	on node B: (nodeid = 2)
<code>mkfs.kdfs /PATH1 1</code>	<code>mkfs.kdfs /PATH2 1</code>
Create kDFS local '...'	Create kDFS local '...'
Create kDFS root entry	

Table 1. kDFS structure creation (two nodes)

For each entry (a directory or a file) of the kDFS hierarchy, a "native" directory is created on one kDFS partition within the cluster. it contains two files:

- *The .meta file* that stores meta-data associated with the entry (size, timestamp, rights, striping information, ...)
- *The .content* that stores real data (directory and file contents).

The name of the "native" directory is defined by the kDFS local inode identifier (each kDFS superblock file contains an identifier bitmap to define the next free inode *id*).

To avoid scalability issues with large directories, we have arbitrarily chosen to sort each kDFS partition in groups of one hundred entries. For instance, when `mkfs.kdfs` creates the kDFS root entry, the command first creates a sub-directory '`DIR_PATHNAME/0-99/`'. Then, it creates the corresponding "native" directory which is the '`DIR_PATHNAME/0-99/1/`' directory. Finally, the file '`.meta`' and the file '`.content`' are created inside this latest directory.

Every hundred entries, kDFS adds a new sub-directory corresponding to the appropriate range ('`DIR_PATHNAME/100-199/`', '`DIR_PATHNAME/200-299/`', ...).

Once the partition is formatted, users can access the kDFS file system thanks to the mount command: `mount -t kdfs ALLOCATED_DIR|NONE MOUNT_POINT` where `ALLOCATED_DIR` corresponds to the native file system directory formatted with `mkfs.kdfs` and `MOUNT_POINT` is the traditional mount point.

Table 2 describes kDFS mounting procedure from two nodes:

On node A: (nodeid = 1)	on node B: (nodeid = 2)
<code>mount /PATH1 /mnt/kdfs</code>	<code>mount /PATH2 /mnt/kdfs</code>

Table 2. Mount kDFS partitions

`/mnt/kdfs` is now a global kDFS namespace for both nodes

Since files and directories are stored in a distributed way on the whole cluster, we need mechanisms to find the kDFS entry point and thus be able to join the file system. kDFS provides two ways of retrieving the root inode. The first

one is based on the superblock file stored in the formatted kDFS partition. As mentioned, the kDFS superblock file provides several information including the kDFS root inode id. Thus, when a mount operation is done, the `'...'` file is read from `'ALLOCATED_DIR'` and the root inode *id* is used to retrieve the kDFS root entry. The second mechanism relates to diskless nodes or nodes which do not want to take part in the kDFS physical structure. In such a case, users do not specify a "device" but have to provide the kDFS root inode id as an additional mount parameter.

3.2 File System Architecture

The use of kDDM in our symmetric file system enables to propose a simple design based on two layers. The highest one, in charge of forwarding local requests to the kDDM service, is directly plugged under the VFS. The lowest one, composed by the different I/O linkers, applies kDDM requests on proper kDFS partitions previously presented.

- Currently, kDFS has been implemented using three families of kDDM sets:
- *Inode kDDM set*, one cluster wide. It provides a cache of inodes recently accessed by processes.
 - *Dir kDDM set*, one per directory. Each Dir kDDM set contains directory entries (roughly names of subdirectories and files).
 - *File kDDM set*, one per file. It stores data related to the file contents.

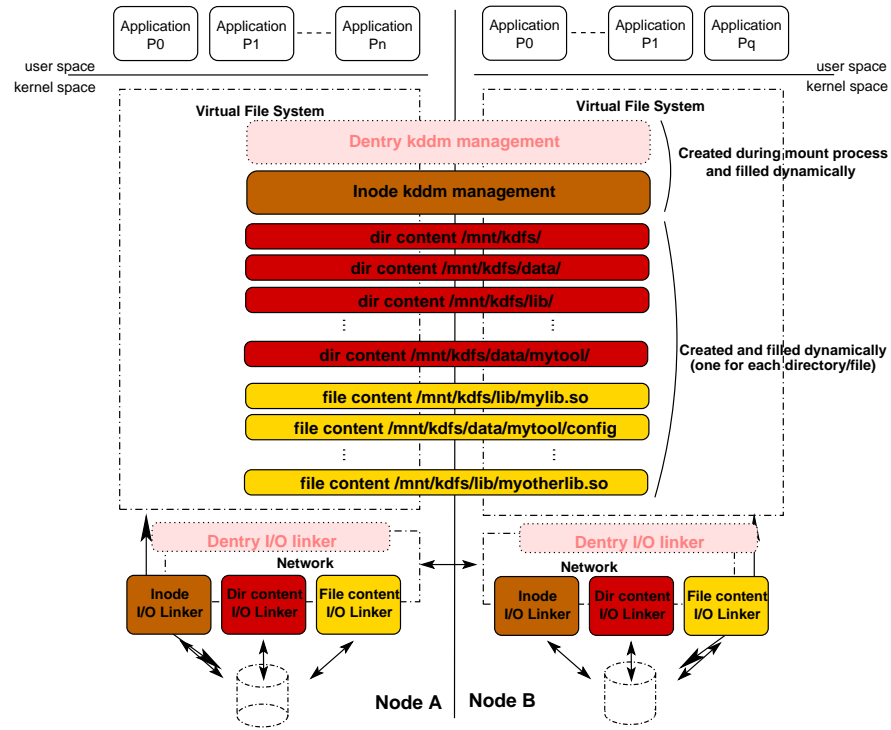
Figure 3 depicts the kDDM entities for several kDFS entries. To make reading and understanding easier, we also provide a potential representation of the regular files for each kDFS entry on the native file systems.

Next sections introduce each of these three families of kDDM sets. A fourth kDDM set is depicted in Figure 3: the dentry kDDM set. This unique kDDM set provides a distributed cache to manage all dentry objects. It is currently not implemented and requires deeper investigations.

3.3 kDFS Inode Management

kDFS inode management relies on one global kDDM set. This set exploits a dedicated I/O linker for inserting/removing and maintaining each kDFS entry on the corresponding local file system (sub-directory creation/deletion and updating of the `'meta'` file). The inode set is created during the first kDFS *mount* operation within the cluster. At the beginning, it only contains the kDFS *root* inode. Then, when a process wants to access one file/directory, its corresponding kDFS inode is added into the set (providing by this way a fully distributed inode cache). All file/directory creations are performed locally whenever possible. That means, when a process wants to create a new file or a directory, kDFS looks for a kDFS partition. If there is one directly available on the node, a new inode identifier is obtained from the superblock file. Otherwise, kDFS stores the new entry on the same node as the parent directory.

When a *mount* operation is done, the root inode identifier is used to retrieve the root inode structure directly inside the inode kDDM set (cf. Section 3.1). If the kDDM set already exists, the structure is already cached and the inode is simply returned to the node. Otherwise, the request is forwarded by the kDDM



On node A: (nodeid = 1) DIR_PATHNAME: /PATH1	on node B: (nodeid = 2) DIR_PATHNAME: /PATH2
mount /PATH1 /mnt/kdfs	mount /PATH2 /mnt/kdfs
"/mnt/kdfs/" ↪ /PATH1/0-99/1/.meta /PATH1/0-99/1/.content	"/mnt/kdfs/data/" ↪ /PATH2/0-99/1/.meta /PATH2/0-99/1/.content
"/mnt/kdfs/lib/" ↪ /PATH1/0-99/2/.meta /PATH1/0-99/2/.content	"/mnt/kdfs/data/mytool/" ↪ /PATH2/100-199/104/.meta /PATH2/100-199/104/.content
"/mnt/kdfs/lib/mylib.so" ↪ /PATH1/200-299/280/.meta /PATH1/200-299/280/.content	"/mnt/kdfs/data/mytool/config" ↪ /PATH2/100-199/105/.meta /PATH2/100-199/105/.content
...	"/mnt/kdfs/lib/myotherlib.so" ↪ /PATH2/100-199/180/.meta /PATH2/100-199/180/.content

Fig. 3. Global kDFS namespace distributed on two nodes

service to the proper I/O linker within the cluster. Finally, the I/O linker exploits the inode *id* to retrieve required information from the hard drive.

KDFS inodes are currently based on 32 bits, the 8 MSB bits provide the node *id* within the cluster and the 24 LSB ones correspond to the local *id*.

3.4 kDFS Content Management

Since kDFS file hierarchy is based on native file systems, both directories and files are stored as regular files. In contrast to traditional file systems, the management of a large kDFS directory containing a huge number of directory entries is similar to the management of a kDFS file content. Regardless of the kDFS entry, its content is stored in its respective '.content' file.

kDFS Directory Management When an application tries to list the contents of a directory stored in the kDFS storage space, kDFS creates a new directory kDDM set. This new set is linked to the inode object stored in the inode kDDM set and caches all directory entries on a page basis. In other words, all file and subdirectory names stored in the directory are loaded in objects of this new kDDM set (one object corresponding to one page). After that, all filename manipulations such as *create*, *rename* and *remove* apply modifications of these pages. The associated dir I/O linker is in charge of propagating changes to the proper hard drive into the `.content` file.

kDFS File Management In this first prototype, kDFS file management is similar to directory management: when an application tries to access a file *f*, if *f* is not already "cached", kDFS creates a new file kDDM set. As for directory management, this new set is linked to the corresponding inode object of the file *f*. The associated file I/O linker is in charge to read/write data from/to the `.content` file of *f* and remove/put pages in the set.

4 Related Work

While distributed applications programmers can choose between a multitude of parallelization concepts, libraries and languages, infrastructure extensions for supporting distributed operating systems are rather rare. Most distributed services such as the Linux ones (the global filesystems Redhat GFS and Oracle's OCFS2, the distributed lock manager or the clustered logical volume manager) are each using their own protocols for communicating across nodes and have their own notion and view of what group of nodes they regard as "the cluster".

All these services are implemented on top of the socket interface, with no cluster abstraction or communication layer in between.

A component worth mentioning as piece of infrastructure specially built to ease distributed computing is SunRPC [1]. It is used heavily in the network file-system NFS implementations and offers mechanisms to register, locate and invoke remote procedures, while transferring data in the exchangeable data format XDR, allowing nodes of different architectures to interoperate transparently. Programming with RPCs has been adopted widely in both user and kernel space and has contributed significantly to the development of distributed computing.

A further step towards distributed operating systems is the introduction of cluster aware network stacks like the Transparent Inter Process Communication protocol TIPC [9]. It uses a cluster oriented addressing scheme, implements features like reliable multicast, cluster membership with subscription to membership change notifications, cluster-wide services and name-spaces. TIPC provides a socket interface to ease transition of socket based codes, as well as a raw kernel-only interface which allows full control over all features. kDDM is leveraging TIPC by using it as low level cluster communication infrastructure.

All former solutions are based on the message passing model. Another approach close to the kDDM one has been proposed by the PLURIX project [10]. Built on top of distributed JAVA objects, this system shows that a shared object model can make the design of distributed services easier. The PLURIX object consistency is based on a transactional model whereas kDDM service exploits an invalidation-on-write protocol.

5 Conclusion and Future Work

The paper has introduced kDDM as infrastructure component suitable for building cluster-wide services in a simple and natural way. The symmetric filesystem kDFS was presented as one highly complex distributed service leveraging kDDM functionality. Its implementation shows how the complexity of managing distributed data transfers and protocols can be reduced dramatically by using the kDDM's distributed shared objects concept (no additional network protocol has been required).

The paper also aimed at pointing to a general method of designing and implementing distributed services by extending their well-known non-distributed variants with kDDM functionality. This methodology has been applied in the past years and lead to the Kerrighed single system image system, one of the most complete implementations of SSI. Kerrighed is a distributed operating system where all distributed components are implemented on top of the kDDM infrastructure.

Future work aims at providing kDDM as component loadable as module for normal unpatched Linux kernels, and increasing the number of stand-alone distributed services built on top of kDDM. Work on the symmetric file system kDFS will continue ⁴, next planned steps being: improvement of stability, evaluation and tuning of performance.

References

1. Sun Microsystems, I.N.W.G.: Rpc: Remote procedure call protocol specification version 2. RFC1057 Internet Request For Comments, June 1988
2. Lottiaux, R., Morin, C.: Containers: A sound basis for a true single system image. In: Proceeding of IEEE International Symposium on Cluster Computing and the Grid (CCGrid '01), Brisbane, Australia (May 2001) 66–73
3. Li, K.: Shared Virtual Memory on Loosely Coupled Multiprocessors. PhD thesis, Yale University (September 1986)
4. Anderson, T.E., Dahlin, M.D., Neefe, J.M., Patterson, D.A., Roselli, D.S., Wang, R.Y.: Serverless network file systems. Computer Science Division, University of California at Berkeley, CA 94720 (1995)
5. Li, Q., Jing, J., Xie, L.: Bfxm: a parallel file system model based on the mechanism of distributed shared memory. SIGOPS Operating Systems Review **31**(4) (1997)
6. Carns, P.H., Ligon III, W.B., Ross, R.B., Thakur, R.: PVFS: A parallel file system for linux clusters. In: Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, GA, USENIX Association (October 2000) 317–327
7. Schwan, P.: Lustre : Building a file system for 1,000-node clusters. In: Proceedings of the Linux Symposium, Ottawa. (July 2003)
8. XtremOS consortium: Design and implementation of high performance disk input/output operations in a federation. Deliverable D2.2.5 (November 2007)
9. Maloy, J.: Tipc: Providing communication for linux clusters. In: In Proceedings of the Linux Symposium, July 21st-24th, Ottawa, Ontario, Canada. (2004) 347–356
10. Goeckelmann, R., Schoettner, M., Frenz, S., Schulthess, P.: Plurix, a distributed operating system extending the single system image concept. In: Canadian Conference on Electrical and Computer Engineering, 2004. Vol.4. (2004) 1985–1988

⁴ The prototype is available at: <http://www.kerrighed.org/wiki/index.php/KernelDevelKdFS>