Kernel Level Distributed Inter-Process Communication System (KDIPC)

Thesis by

Cristian Ţăpuş

In Partial Fulfillment of the Requirements
for the Degree of
Master of Science

California Institute of Technology
Pasadena, California

2004
(Submitted August 31, 2004)

To my wife, Diana Mariela, and
to my parents, Mariana and Nicolae.

Acknowledgements

“La reconnaissance est la memoire du coeur.”
(Gratitude is the memory of the heart)
Jean Baptiste Massieu

I would like to thank all the people who, through their valuable advice and support, made this work possible. First and foremost, I am grateful to my adviser, Professor Jason Hickey, for his distinguished mentorship, for the many insightful conversations that led to the development of the ideas in this thesis, for his constant support and his positive attitude, and for giving me the opportunity to work in the great research environment of the Mojave group at Caltech. I would also like to thank my colleagues in the Mojave group for their constructive input and feedback, and, in particular, to Nathaniel Gray, Aleksey Nogin and Justin Smith, for their comments and contributions to the development of this work. Furthermore, I am thankful to Professor Peter Keleher for introducing me to the field of distributed shared memory and for the valuable resources he maintains on the topic.

Last, but not least, I would like to thank my wife, Diana, my parents, Mariana and Nicolae, and my late grandparents for their love and continuous support and encouragements.

Abstract

This thesis presents a kernel level distributed inter-process communication library (KDIPC) with support for distributed shared memory and distributed semaphores. KDIPC uses the System V inter-process communication programming interface and enhances it to provide functionality in distributed environments. The library uses a sequential consistency model for shared memory that provides ease of programming and preserves the semantics of parallel programs. A key feature of KDIPC is the use of the semaphore interface to support distributed synchronization. The implementation is done at the Linux kernel level to reduce the overhead induced by the strict consistency model.

The thesis also investigates several protocols for maintaining sequential consistency, and for providing location discovery of remote copies of shared memory segments and semaphores. A case-study illustrates the benefits of KDIPC as compared to other inter-process communication libraries, such as the ease of programming provided by the sequential consistency model. Various details related to the implementation, the challenges encountered, and the benefits and disadvantages of using the different protocols are also presented.

Future avenues of research include the use of speculative execution to improve the performance of the protocol used to maintain sequential consistency, the development of a distributed file system on top of the shared memory system, and the implementation of applications from applied sciences that would benefit from using the simple synchronization and sharing mechanisms of KDIPC.

Contents

List of Figures

Chapter 1

Introduction

This thesis introduces a new approach to building an interprocess communication (IPC) library for distributed applications consisting of a distributed shared memory (DSM) subsystem and a distributed semaphores subsystem.

This chapter describes the motivation for building such a library and discusses a class of applications that would benefit from using it. The remainder of the thesis presents a specification of the system and a detailed discussion of the implementation of this library.

1.1 Overview

In today's computing world, interprocess communication has become an important programming tool. It is no longer conceivable to build programs that do not interact with each other, or do not make use of information stored on a remote computer. Several techniques may be used for IPC: semaphores, shared memory, message passing, and remote procedure calls. In this work, the emphasis is on the first two mechanisms: shared memory and semaphores.

The main motivation for this research was to provide a tool for building distributed applications that has a simple programming model, makes programs easier to understand and reason about, and is efficient enough to be appealing to programmers.

Most of the existing systems that provide inter-process communication libraries either provide a programming model that is very complex, which makes programs hard to understand and reason about, or are inefficient while providing a simple model. Performance is delivered in the first class of systems by putting the burden of designing complex applications on the programmer. In the other approach, performance is sometimes sacrificed at the expense of providing a simple and intuitive *single system image* of the distributed system.

1.2 Applications

There are many applications that can make use of inter-process communication mechanisms like distributed shared memory or distributed semaphores. The most common class of applications that would benefit from using these mechanisms is the class of scientific computing applications, like simulation of physical processes. Applications falling into this category perform high-demanding computation on given sets of data, and can be easily parallelized. Consider a computation problem that can be efficiently solved by mapping the input data to a grid of computation nodes, as presented in Figure 1.1. Each node uses part of the results of the computation of its neighbors for a given stage as part of its own next computation stage. One approach is to allow nodes to output the results of each computation stage into a shared space. The shared space is further used as part of the next stage. These simulations usually perform the computations for a very large number of time steps, so designing a simple and efficient sharing scheme is necessary.

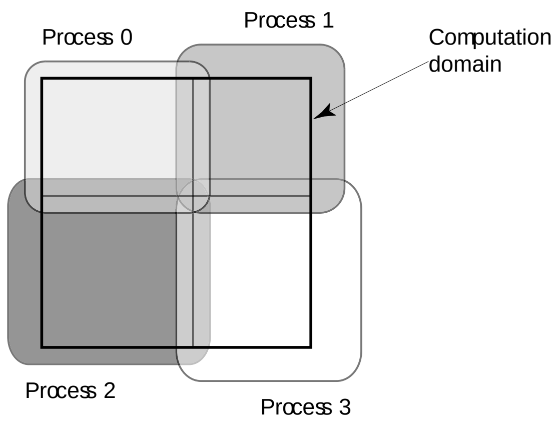

Figure 1.1: A simulation involving high demanding computation performed on a given two dimensional domain for a very large number of time steps. Distributing the computation speeds up the global simulation time

Many applications in this category can be significantly sped up by distributing the computation across a computation grid, where neighboring machines share only their border information at each time step. An example of a way to distribute such a computation is presented in Figure 1.2. The programmers usually implements a custom protocol to be used by each node in the computation grid to exchange the required information with its neighbors. The protocols are usually very specific to the problem they were designed for and are not easily adaptable to other problems. Another drawback is that the presence of the communication protocol as part of the computation code makes the application more error prone and also harder to maintain and understand.

The approach presented in this work provides this functionality without the need for home-grown protocols, through a very popular and simple interface for inter-process communication.

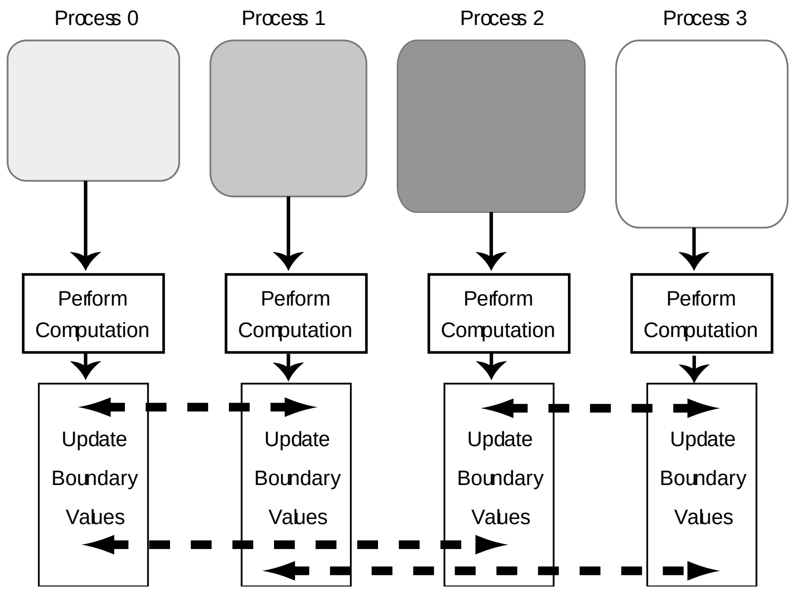

Figure 1.2: Parallel implementation of heat equation application using message passing

1.3 Problem statement

Communication libraries like MPI [1] have become very popular in the community of programmers for distributed numerical computing applications. However, they require a good understanding of principles of network programming. Furthermore, programmers have to become familiar with the problems of using explicit communication in their code, like handling faults, ensuring synchronicity of nodes in a distributed environment, and understanding how to maintain consistency of the shared data in a distributed environment. Faults can occur both at the network layer and at the application layer. Synchronization mechanisms using message passing are not very intuitive, as compared to those using classic synchronization structures, like semaphores. When using a message passing mechanism, there are usually multiple copies of the same data stored at several nodes in the system, because parts of the data are transfered from one node to the other. The programmer has to make sure that when performing any sort of computation involving this data, the node the computation is running on has the latest version of the data. The consistency of shared data is a very complex problem in itself and can have a great impact both on the correctness of the program and on its performance. More details about different consistency models will be discussed in Section 2.3.

While these issues related to home-grown protocols for sharing data across a computation grid are familiar and present interesting challenges to researchers in computer science, they are often of no interest to physicists, for example, who need to share data across a computation grid as part of their applications.

1.4 Contributions

The obvious solution to the problems presented in Section 1.3 is to isolate the communication protocol from the computation code. The natural way to do this is to provide a library of functions that would provide a higher-order interface for handling shared data, and that would take care of all the problems associated with communication and maintaining consistency of the shared data. This thesis addresses this issue by presenting a system that enhances an existing simple shared memory interface to work over distributed environments, in a way transparent to the programmers. We extend System V IPC application programming interface (API) to support distributed shared memory blocks and distributed semaphores. In this implementation the communication mechanisms and the used infrastructure are hidden from the programmer. The developers of distributed applications are presented with a higher level interface for handling distributed shared objects, that provides a clear semantics and a simple consistency model for shared objects (memory or semaphores).

1.4.1 Preserving semantics

System V IPC API is one of the most commonly used application programming interfaces for interprocess communication. While it only works for processes running on a single machine, its API provides a simple way to create and use IPC structures like semaphores and shared memory, and it also provides a very simple semantics for shared objects accesses. It is fairly accurate to say that learning the System V IPC is one of the first steps taken by programmers who want to build parallel applications.

Many other projects have addressed the issue of isolating the communication infrastructure from the programmers, by providing an interface for managing shared objects in distributed environments. However, most of the interfaces provided by these systems are significantly different from the high-order IPC APIs for single machines, like System V, and also from other higher order interfaces for distributed environments. Another drawback of these systems is that they usually provide a more complex consistency model for the shared objects, which involves more synchronization on the programmer' side. Therefore, programmers have to adapt their code to meet the requirements of the consistency model provided by the system they are using. Since most of these systems are research projects, their developers sometimes decide to stop maintaining them or they investigate new consistency models for updated versions of the systems. Furthermore, some applications that use these systems can be easily adapted to loose consistency models, while others require stricter models. In any of these cases, the programmers are faced with new challenges, like learning new interfaces or adapting their code to new consistency models, which sometimes can involve an almost complete rewrite of the application.

The Kernel Level Distributed Inter-process communication (KDIPC) system, presented in this thesis, enhances the System V IPC API to make it suitable for a distributed environment, while maintaining the simple interface and the consistency model provided.

The decision to extend System V IPC API was also motivated by the belief that programs designed for parallel machines with shared memory and those designed for distributed environments should have the same semantics with respect to shared objects. More specifically, programs written for symmetric multiprocessor machines should be ported without effort to a distributed environment.

1.5 Organization of this thesis

Chapter 2 begins with a presentation of related projects and their approach to the problem of distributed shared memory. It also contains a discussion of different consistency models encountered in these systems and it concludes with a comparison of three implementations of an example application using three different distributed shared memory systems. Chapter 3 discusses the benefits of distributed semaphores and provides an overview of a few existing systems that provide semaphores or other synchronization mechanisms for distributed applications. Chapter 5 presents details of the implementation of the Kernel Level Distributed Inter-Process Communication (KDIPC) system. Chapter 6 presents the conclusions of the work presented in this thesis and future directions of research.

The thesis also contains a set of Appendices, that provide further clarifications of some of the issues, details of related projects and descriptions of spin-off libraries developed along with the KDIPC system. Appendix A describes the distributed shared memory interface of TreadMarks. Appendix B presents a socket library kernel module developed in parallel with the KDIPC system. Appendix C contains a set of benchmarks that were used to shape the design of the KDIPC system. Finally, Appendix D focuses on details of the implementation by discussing the interaction between the KDIPC modules and the Linux kernel data structures.

Chapter 2

Semantics of Distributed Shared Memory

This chapter gives an overview of the semantics of different shared memory models, and focuses on the choice made for the semantics of KDIPC distributed shared memory model. First, the approaches to modeling distributed shared memory taken by different related projects are presented, followed by a discussion of different consistency models and semantics of shared memory. Finally, the model, the semantics of the distributed shared memory used in the KDIPC system, and the provided application programming interface will be discussed.

2.1 What is shared memory

Shared memory is a simple and efficient way of passing information between processes. On a more technical note, shared memory presents a single address space that is accessible by multiple processors. Shared memory systems cover a very wide range of systems, from multi-processor machines to loosely coupled systems. The multi-processor machines provide shared memory at the hardware level, while , at the other end, the loosely coupled distributed systems enforce it through software intervention.

2.2 Related work

The concept of distributed shared memory was introduced by Kai Li in 1986 [17]. Since then, many systems have emerged [6, 21]. This section focuses on those which are consider to have shaped the development of distributed shared memory systems.

The first generation of DSM systems used *strict consistency* memory models, like sequential consistency, and differed mostly in the mechanism used to maintain coherence. One of the first systems to provide distributed shared memory capabilities was Ivy [18]. Ivy was developed by Li and Hudak and implemented conventional multiprocessor cache hardware coherency protocol in software. The *unit* for coherence was a virtual memory *page*, and the used model was sequential consistency. Ivy implemented a single writer, multiple readers protocol and provided a distributed directory for page lookup. Each page had associated a “probable owner” and a “copy set”, used to maintain consistency. The main issues with the Ivy system were the following: sequential consistency required heavy communication, the system did not provide any synchronization mechanism, and it couldn’t handle false sharing within the same consistency unit. False sharing refers to simultaneous accesses to different data located on the same shared memory page, for example. When one node changes the data on its local copy of the shared memory page, the other copies of the page are invalidated. This forces the other nodes to retrieve the update for that page before performing their own operations, even if their local accesses do not interfere with the invalidating operations. From the same generation it’s worth mentioning Clouds [23] which introduced synchronization mechanisms to address some of the performance issues.

Another category of systems used *objects* as the coherence unit. Representative for this class of systems was Emerald [12], which implemented the same consistency model and suffered from the same performance issues as Ivy and Clouds.

The third category of first generation systems used *segments* as coherence units, and implemented a request freezing mechanism to improve performance. This work was done by Brett Fleisch who first implemented a distributed System V IPC API for Locus [10] and then used his experience to develop Mirage [11].

The biggest progress in terms of performance in the history of DSM systems was done in the early nineties with the introduction of the release consistency memory model. The concept was introduced in Munin [7, 8] and was further improved in TreadMarks [16]. Munin was the first system to introduce the release consistency model. It forced programmers to use synchronization operations before performing read or write operations to shared data. The synchronization mechanism provided by Munin were locks, barriers, and condition variables, allowing for a multiple writers, multiple readers protocol. The option of combining updates to multiple pages and variables into a single message reduced significantly the communication overhead of the system. However, release operations would be more expensive than any other operations, even compared to operations performed in earlier systems. Another important disadvantage was that semantics of programs had to change from the first generation of distributed memory systems due to the semantics of the new consistency model.

TreadMarks [15], developed by Peter Keleher at Rice University in early 90s, introduced the concept of lazy release consistency. The lazy release consistency model used by TreadMarks reduced significantly the communication bandwidth, as compared to Munin, but increased the complexity of the distributed shared memory system. The burden of guarding every shared memory access and adapt the programs to the new semantics of the DSM system was all put on the programmer.

TreadMarks is considered by many to be the state of the art DSM system despite its drawbacks.

In the late 90s, there was a new effort to create a DSM system to be included in the Linux distributions. Distributed inter-process communication (DIPC) [14, 13] for Linux was developed by Kamran Karimi from Iran University of Science and Technology. DIPC uses a strict consistency protocol that makes programming very intuitive. For DIPC, the major disadvantages were that it only supported a statically defined cluster of computers, and that it was mostly implemented as a user-level program, which had a strong impact on its performance. The merit of DIPC consists of being the first DSM system to be implemented as part of the Linux kernel (although it was back in the 2.1 release) supporting the System V IPC API.

2.3 Consistency models

A consistency model refers to the way memory accesses behave. It usually makes sense to talk about consistency models in the context of shared memory systems, where multiple processing units (CPUs, nodes in a distributed system, etc.) access the same memory area. Consistency models for shared memory are usually divided into *strict* and *relaxed*. Some of these have been mentioned in Section 2.2 and more details of each of the aforementioned consistency models will be provided in this section.

2.3.1 Sequential Consistency

A system implements a sequentially consistent memory model if the result of any execution is the same as if the operations of all the processors were executed in sequential order, and furthermore, the operations issued at each processor appear in the same sequence as specified in the program.

The definition of sequential consistency, as given above, imposes a total order on all memory accesses in the distributed system. It provides the same semantics as uniprocessors and multiprocessor machines. Systems that implement this consistency model usually support single writer multiple readers with write invalidation protocols.

An example of a simple distributed system with two processes and two executions, one which is sequential consistent and one which is not are presented in Figure 2.1. The first outcome can be obtained by a sequentially consistent run, which is an interleaving of code lines from P1 and P2 that keeps the per process order: P2.1, P2.2, P1.1, P1.2. The second outcome would not be possible through any sequentially consistent run because there is no run that would preserve the order of instructions in P1 and P2 and produce the output x=0 and y=0.

	P1	P2		Outcome
1:	x=1	y=1	SC Run	1 0
2:	print(y)	print(x)	Non-SC Run	0 0

Figure 2.1: Sequential Consistency. (Variables x and y both start with value 0).

2.3.2 Eager Release Consistency

Release consistency is considered to be a *weak* (or *relaxed*) consistency model because it requires the use of synchronization mechanisms to control the succession of events and it allows delayed propagation of changes made to shared data.

Release consistency, by default, refers to eager release consistency. There are other versions of release consistency memory models, like lazy release consistency.(see Section 2.3.3)

Weak consistency models, like release consistency, classify memory accesses in four operations: acquire, read, write, and release. The acquire and release operations are synchronization operations, while read and write operations usually only occur inside a synchronization block. Release consistency imposes the following restrictions on memory accesses:

- acquire and release operations are performed in a sequential consistent manner and in the order they occur in each process
- read and write operations to shared data can be performed only when all previous acquire operations have completed successfully
- release operation is performed only after all the previous read and write operations have been successfully performed

2.3.3 Lazy Release Consistency

Lazy release consistency [15] is a version of release consistency that tries to reduce the communication overhead by changing the consistency requirements. Read and write operations are not required to happen before the release operation, but can be postponed until the next acquire request happens. More formally, as presented by their author Peter Keleher, the conditions imposed by the lazy release consistency model are:

- Before an ordinary read or write access is allowed to perform with respect to another process, all previous acquire accesses must be performed with respect to that other process, and
- before a release access is allowed to perform with respect to any other process, all previous ordinary read and write accesses must be performed with respect to that other process, and

- synchronization operations (acquire and release) are sequentially consistent with respect to one another.

This requires less communication than in the eager release consistency model, because accesses do not need to be performed globally at the next release, but rather be performed at other processes, as release operations become visible to them.

2.4 The KDIPC approach

As seen in Section 2.2 there is a wide variety of approaches to building distributed shared memory systems. Each has its own advantages and disadvantages. One of the main issues considered during the designing phase of the KDIPC system was maintaining the semantics of parallel programs when porting them to a distributed environment. This led to the adoption of System V IPC API as the interface for the KDIPC system and further enforced the use of a sequential consistency model in preserving consistency of the objects stored by the system. Another important design decision was to implement the system as a kernel module and fully integrate it with the Linux kernel. The decision to use System V IPC API as the interface for KDIPC required the implementation of the system as part of the kernel. More details about this and other design decisions made for the KDIPC system are discussed in more detail in Chapter 5 in the context of the related work presented in Section 2.2.

2.4.1 Maintaining Single Processor Semantics

Programmers are mostly used to writing programs on single processor machines. These programs usually support sequential consistency. It is only natural to support the same kind of semantics when designing distributed systems, to take some of the burden off the programmer. However, maintaining sequential semantics was not driven by this reason alone. Designing programs in a sequential consistency model is easier, and the programs themselves are simpler and easier to reason about than those in which a relaxed consistency semantics is used.

A system using a looser consistency model is, in principle, faster than a system using a strict consistency model, like sequential consistency. In KDIPC, however, the cost of sequential consistency is minimized, and the overhead introduced by the sequential consistency requirements is offset by the ease of programming, making the adoption of the sequential consistency memory model worthwhile.

2.4.2 System V / KDIPC API

System V IPC API for shared memory is defined by the following system calls:

- int shmget(key_t key, int size, int shmflg);

"shmget() returns the identifier of the shared memory segment associated to the value of the argument key. A new shared memory segment with size equal to the round up of size to a multiple of PAGE_SIZE is created..." (shmget man page)

- void *shmat(int shmid, const void *shmaddr, int shmflg);

  "The function shmat attaches the shared memory segment identified by shmid to the address space of the calling process" (shmat man page).

- int shmdt(const void *shmaddr);

  "The function shmdt detaches the shared memory segment located at the address specified by shmaddr from the address space of the calling process" (smdt man page).

The only addition to the standard System V IPC API was the introduction of a new flag to mark the distributed shared memory blocks. The new flag, called *IPC_KDIPC*, is used by the shmget function to declare a new distributed shared memory segment.

As described in this Section, the interface is very small and simple. For a comparison with the API of other distributed shared memory systems, like TreadMarks, please see Appendix A.

Chapter 3

Semantics of Shared Semaphores

This chapter provides an overview of the semantics of different shared semaphores models, and details on the design decisions taken in the KDIPC implementation of semaphores. The description of different related work project is first addressed along with the discussion of the advantages and disadvantages of each approach. Next, the KDIPC API for distributed semaphores is presented.

3.1 What are semaphores?

Semaphores are a classic method for enforcing synchronization in parallel and distributed environments. They were first introduced by Dijkstra in 1960s and have been widely used ever since.

A semaphore s is a protected variable with non-negative value that can be accessed only through two operations: P(s) and V(s). P and V stand for Dutch "Proberen", to test, and "Verhogen", to increment. The functionality of the P and V operations is illustrated below.

```
P(Semaphore s) :
  while (s==0) ; // wait for s to become positive
  s--;


V(Semaphore s) :
  s++;
```

The operation P(s) decrements the value of the semaphore by 1 if the value is positive, or it waits for the value to become positive and then it decrements it. The V(s) operation increments the value of the semaphore s by 1. Both operations are atomic, so only one process is allowed to access the semaphore during a P or a V operation. If the P operation is blocked, then V operations can occur to allow the unblocking of the P operations.

3.2 Related work

This section focuses on presenting two semantics for locally shared semaphores provided by the POSIX interface and by System V interface, and one effort to provide distributed semaphores for clusters [22].

3.3 POSIX Semaphores

The POSIX (Portable Operating System Interface) [2] IPC API is part of an effort by IEEE (Institute for Electrical and Electronics Engineers, Inc.) to develop a family of standards for Operating Systems. POSIX has also been adopted by ISO (the International Organization for Standardization) and it will probably become the de facto standard. POSIX defines interfaces for semaphores, message queues and shared memory. The focus of this section is to describe only the semaphores interface

The main functions presented in the POSIX standard [2, 24] for shared semaphores are:

- Create a semaphore. This function requires a name for the semaphore, some flags, and an initial value of the semaphore. As described in [2, 24], the following interface for the create function can be used:

  ```
  sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
  ```

- Close a semaphore. This function closes a created semaphore. This operation is said to occur automatically on process termination for any named semaphore still open. [24]:

  ```
  int sem_close(sem_t *sem);
  ```

- Wait for a semaphore. This function tests the value of the specified semaphore and tries to decrement (lock) it if the value is positive.

  ```
  int sem_trywait(sem_t *sem);
  int sem_wait(sem_t *sem);
  ```

  There are two versions of this function. The sem_trywait function does not block the calling process if the semaphore is already locked (has a non-positive value). The sem_wait function blocks the calling thread until the decrement action can be performed (the value of the semaphore becomes positive).

- Post to a semaphore. This function increments the value of the semaphore. The interface for the semaphore function is:

```
int sem_post(sem_t *sem);
```

The POSIX standard describes only the interface programmers have for accessing the semaphores. There can be multiple implementations, depending on the underlying mechanisms used in deploying the POSIX IPC library. Richard Stevens [24] presents several possible implementations of POSIX semaphores interface, one of which is of interest to us since it uses the System V IPC API (discussed in Section 3.4). As described in [24], System V IPC API is a very powerful interface that can be naturally used to implement the POSIX standard.

One drawback of the POSIX standard is that the naming interface, as well as the semaphores interface are not very thoroughly defined in the specifications. This leaves room for incompatibilities between different implementations. Issues with these incompatibilities are also discussed in [24].

3.4 System V Semaphores

System V Semaphores have been developed during the late 1970s and were added to commercial Unix with System V in the early 1980s. The main feature of the System V Semaphores is the ability to define sets of semaphores, on which operations are atomic. The interface provided by System V semaphores API is the following:

- `int semget(key_t key, int nsems, int semflg);`

  This function returns the semaphore set identifier associated with the argument *key*. Depending on the flags passed in *semflg* as arguments to this function, a new semaphore set can be created if no existent semaphore set exists associated with the *key*. The semaphore set contains a number of semaphores equal to the number *nsems* passed as argument. The details of the initialization of the data structure associated with the semaphore set are described in detail in the *man* page of this function.

-
  ```
  int semop(int semid, struct sembuf *sops, unsigned nsops);
  int  semtimedop(int  semid, struct sembuf *sops, unsigned nsops, struct
         timespec *timeout);
  ```

  The *semop* operation performs operations on members of the semaphore set identified by the parameter *semid*. The members of the semaphore set that are affected by the operation are specified by the *sops* parameter. *sops* points to an array of elements, each containing the semaphore number that is affected by the operation, the actual operation to be performed and a set of flags. The operation to be performed can be either increment or decrement the value of a semaphore by a specified value. An operation on a given semaphore is successful if the value of the semaphore after the operation is performed is positive. If all operations performed on a semaphore set are successful the *semop* call is successful. Otherwise, the process invoking the *semop* function is put to sleep until all operations described by the function parameters become simultaneously successful.

The function *semtimedop* behaves identically to the function *semop*, with the exception that if the operation can not be performed, the process is not put to sleep indefinitely, but the sleep operation times out after a specified time.

- `int semctl(int semid, int semnum, int cmd, ...);`

  This function performs control operations on a semaphore set identified by the *semid* parameter. The type of control operations performed include getting the value of one or all the semaphores in a semaphore set, initializing the value of one or all semaphores in a semaphore set to a given value, and marking the semaphore set for deletion.

3.5 Distributed Semaphores

Semaphores are very well understood and their implementations for single-machine and parallel systems are very efficient. Ramachandran and Singhal describe an approach to implementing POSIX like semaphores in a distributed environment [22]. They present a theoretical hierarchical model to increase the availability of the semaphore, that makes it usable for structured, grid-like distributed systems. Even if the result is interesting from the theoretical point of view, at the time of publication there was currently no simulation or real-implementation of the algorithm.

The main idea presented in their paper is to divide the value of the semaphore and distribute parts of the total value to clusters of nodes. Operations are then performed on the "local" slice of the semaphore. If a certain operation can not be performed, another cluster is contacted and the value of the semaphore is redistributed to permit the operation.

Ramachandran and Singhal [22] state that distributed synchronization mechanisms using the semaphore interface have not been investigated independent of distributed shared memory systems. They argue that the need for a programming interface with support for the synchronization of distributed processes is incontestable and that the decision to use a semaphore interface for such a library is natural.

3.5.1 Synchronization in TreadMarks

This Section describes the approach to distributed synchronization mechanisms and distributed barriers taken by TreadMarks. As described in Section A, there are three main functions that address this functionality: Tmk.barrier, Tmk.lock.acquire, and Tmk.lock.release.

TreadMarks defines a constant, fixed numbers of locks and barriers, described by #define primitives. The locks are not general purpose semaphores, but represent binary semaphores (mutex). Barriers, on the other hand, prevent the progress of a process until all of the other processes reach the barrier. The values of semaphores and barriers are initialized during a required initialization section when the Tmk_startup function is called. This function takes care of initializing all the locks and barriers. The number of processes is specified as an argument of this function and the barriers are initialized accordingly.

While general purpose semaphores and barriers could be implemented using TreadMarks' locks, barriers and shared memory functionality, this is somewhat cumbersome and require a full library implementation. Furthermore, parallel accesses to semaphores implemented using these techniques, might be significantly affected by the requirement that TreadMarks barriers require all processes to arrive at the barrier before proceeding. TreadMarks has another disadvantage, related to the distributed code that is executed. TreadMarks requires that all processes using certain shared memory blocks, and/or semaphores execute the same code. Variation of the executed code is achieved by manually verifying the id of the process and taking different actions depending on the value of the *Tmk.proc.id* variable.

3.6 KDIPC approach

One of the main advantages of the System V IPC API, supported by the KDIPC system, are the operations on semaphore sets. This functionality allows programmers to associate semaphores in a semaphore set with various resources and try to reserve multiple resources simultaneously, in an atomic operation. Although this could be implemented by the user using single semaphores, the simplicity of the System V interface makes it more appealing than a home-grown implementation. Furthermore, the mechanisms provided by System V are portable and easily re-usable in multiple applications, where as the home-grown implementation is tightly integrated in the code. This can be very useful for certain distributed application.

As described in Sections 3.3 and 3.4, the System V IPC API is very powerful and can be used to implement even the POSIX IPC standard in a natural way. The KDIPC system enhances this API to make it available for distributed environments. The only addition to the interface is the introduction of a new flag (*IPC_KDIPC*) to mark the fact that semaphore sets are distributed. The interface and guarantees used in the KDIPC implementation are otherwise identical to those provided by local System V IPC API.

Implementing a distributed System V API is difficult. Challenges related to the implementation are presented in Chapter 5. The main issues are the availability of the semaphore set in a distributed environment and maintaining fairness and consistency of the semaphore sets. In KDIPC, the problem of semaphore location discovery is addressed similarly with the way Ramachandran and Singhal have done it in their approach to implement distributed semaphores. Details about the implementation are discussed in Chapter 5.

Chapter 4

Case study

This chapter presents an example illustrating the usage of the KDIPC API in comparison to normal parallel System V IPC API and to the distributed TreadMarks API. The example chosen is a heat distribution problem characterized by a 2D PDE.

4.1 Problem statement

The example is constructed based on the heat equation problem in two dimensions. Consider a rectangular metal plate that is uniformly heated at one of its corners. The problem reduces to computing the heat distribution in the plate for a given period of time. The two-dimensional plate is discretized uniformly in x and y directions with $\Delta x = \Delta y = s$, and heat is computed at discrete times, with time step Δt. If u defines the heat equation as a function of x,y, and t, the second derivation of u at any grid point with respect to x and y can be approximated by the average value of the neighbors in the north, south, east and west directions.

The single processor implementation creates a matrix representing the two-dimensional plate and computes the value of each grid point based on the values of its neighbors at the previous time step. Details of the solution of the heat equation are not presented in this work. Parrallelizing the application and making it run in a distributed environment is of particular interest to the problem we address in this work.

The decomposition of the computation grid and the distribution of the work for a simple example is presented in Figures 4.1 and 4.2. As it can observed, the computation domain of each node overlaps with the computation domain of its neighbors. This parallelization of the computation alows nodes to compute local information and only need limited border-line information from their neighbors. While this can be done using customized message passing interface, as presented in Figure 1.2, a simpler solution is using a shared memory system, where updates occur automatically, and are transparent to the user.

The data structures and the global variables used in all three implementation (KDIPC, System

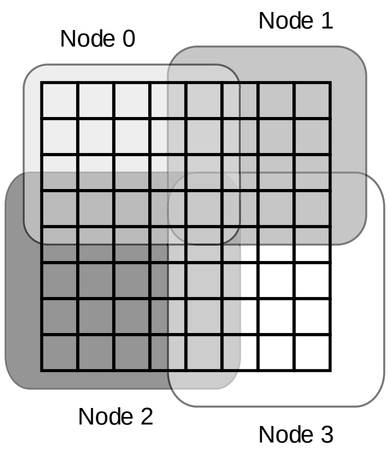

Figure 4.1: 2D domain decomposition

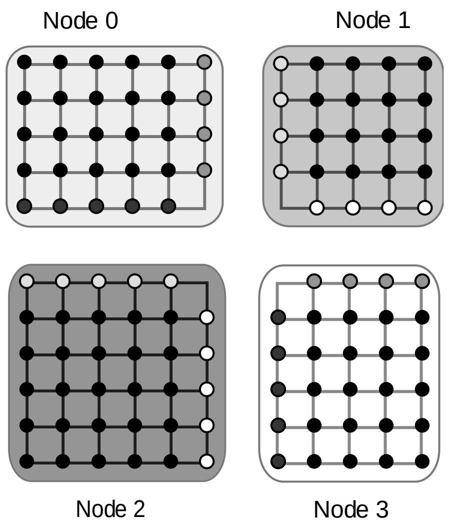

Figure 4.2: Work distribution per node

V and Treadmarks) are presented in Figure 4.3. The *matrix* data structure characterizes the grid points of the plate. The matrix is characterized by size, the number of rows and columns of the grid and by a pointer to the actual data. For each computation node, respectively neighbor of one computation node, the following information is stored: the current computation time step and the range of grid points computed by the node.

4.2 System V implementation

The System V implementation creates a globally shared matrix and maps it into the memory of each process. Each process only accesses the region of the matrix it has been initially assigned. The

```
/*
 * Main matrix information
 */
struct matrix {
  key_t key; // System V & KDIPC
  int memid; // System V & KDIPC
  long size;
  double *region;
  int rows, cols;
};

/*
 * Node information
 */
struct node {
  key_t id;
  int memid;
  int *time;
  struct grid start, stop;
};
```

```
/*
 * Computation grid size
 */
struct grid {
  int row, col;
};

/***********
 * Variables
 */

// Information about local node
static struct node self;
// The main computation matrix
static struct matrix mtx;
// Information on neighbors in the grid
static struct node other[DIRECTIONS];

// Size of computation grid
struct grid map;
```

Figure 4.3: Data structures and variables used in the example

```
  mtx.memid = shmget(mtx.key, mtx.size, flags);
  mtx.region = (double *)shmat(mtx.memid, NULL, 0);
```

Figure 4.4: Mapping main matrix to local memory

mapping process of the global matrix is shown in Figure 4.4. The flags variable is defines as per the man page for the shmget function (flags = IPC_CREAT | 0644). After the matrix is attached to the memory space of a process, the accesses to any part of the matrix are done through normal pointer access mechanisms.

After mapping the global matrix, each process needs to have access to some information about its neighbors. Each node information is associated with a given key, known by all nodes in the computation grid. For this example the key associated with each node's information has been chosen to be the key of the matrix incremented by one and by that node's id. Each node maps his local information in the *self* variable and maps its neighbors information in the *neighb* array. It is important to note that a node only needs to know the keys for its neighbors only, and it does not need to have global knowledge of the keys associated with the information of all the nodes in the system. Even though in the simple scheme used in this example the keys can be deduced, in a real application it might not be feasible or scalable to know certain information on all nodes in the system. The mapping scheme is presented in Figure 4.5.

The computation stage for each node is presented in Figure 4.6. As it can be seen in the code excerpt, the accesses to shared memory are done in a normal fashion, no glue code being necessary.

```
/*
 * Map local information
 */
self.memid = shmget(mtx.key + self.id + 1, sizeof(int), flags);
self.time = (int *)shmat(self.memid, NULL, 0);
self.time = (int *)memset((int *)self.time, 0, sizeof(int));

/*
 * Map neighbors
 */
for (i = 0; i < DIRECTIONS; i++) {
  if (neighb[i].id == NONE) continue;
  // Get shared memory id
  neighb[i].memid = shmget(mtx.key + neighb[i].id + 1, sizeof(int), flags);
  // Attach memory to process
  neighb[i].time = (int *)shmat(neighb[i].memid, NULL, 0);
}
```

Figure 4.5: System V and KDIPC. Mapping the neighbors information to local memory

```
for (*(self.time) = 1; *(self.time) <= timesteps; *(self.time)=*(self.time)+1) {
  oldptr = (*self.time - 1) % LAYERS;
  curptr = *self.time % LAYERS;

  check_neighbors_are_done(neighb);

  for (g.row = self.start.row; g.row < self.stop.row; g.row++) {
      for (g.col = self.start.col; g.col < self.stop.col; g.col++) {
        off = INDEX(curptr, mtx.rows, mtx.cols,g.row, g.col);
        mtx.region[off] = new_value(&mtx, oldptr, g);
      }
  }

 }
```

Figure 4.6: System V and KDIPC. Performing computations

```
Tmk_startup(argc, argv);

if (Tmk_proc_id == 0) {
   mtx.region = (int *) Tmk_malloc(mtx.size);
   Tmk_distribute(&mtx.region, mtx.size);
}

node[Tmk_proc_id].time = (int *) Tmk_malloc(sizeof(int));
Tmk_distribute(&node[Tmk_proc_id].time, sizeof(int));
Tmk.lock.acquire(Tmk_proc_id);
node[i].time = 0;
Tmk.lock.release(Tmk_proc_id);

Tmk_barrier(0);
```

Figure 4.7: TreadMarks: Mapping matrix and neighbors information

4.3 KDIPC implementation

The KDIPC implementation of this application is identical to the System V IPC implementation, with the exception of the flags passed to shmget. The value of the flag variable has to be or-ed with the IPC_KDIPC value defined in *linux/kdipc.h* (flags | IPC_KDIPC). The overhead of porting parallel applications implemented using System V IPC API to a distributed environment is therefore null. As we will further see, this is not the case if another distributed shared memory system, like Treadmarks, is used.

4.4 TreadMarks implementation

The TreadMarks implementation, which is presented in Figures 4.7 and 4.8 has a few disadvantages, as follows. While the code is similar to that of the System V implementation (see Figures 4.5 and 4.6), there are some critical conceptual differences.

First, TreadMarks forces all programs running a common distributed computation to execute the exact same source code. The only way to differentiate between different instances of the computation nodes is through the value of the *Tmk_proc_id* variable. As shown in Figure 4.7, the programmer has to hard-code the actions of different nodes through branch statements based on the value of the Tmk_proc_id variable. In this example, the process with id 0 would execute the mapping and initalization of the matrix shared memory region. The mapping needs to be done only once, after which all processes running the same computation see the shared memory regions. The *Tmk_distribute* function makes the memory region visible to all nodes in the system, making it rather difficult, if not impossible, to have shared memory regions distributed across only a subset of the cooperative nodes.

Another disadvantage of using TreadMarks is related to the memory consistency model used by the system. The TreadMarks documentation [3] describes the necessity of locks for shared memory accesses as follows:

> At an abstract level, a programmer associates a set of privileges with a lock. If a process wants to use one of those privileges, it must acquire the lock. Then it can use the privilege. After the process is done using the privilege, it should release the lock. A typical privilege for which we may use a lock is "Read the value in shared-memory location X" or "Write an updated value to shared-memory location X". The association of the lock with this privilege must be enforced by the programmer . The lock is not associated with the shared-memory location X in any formal way whatsoever.
>
> There are two typical uses for locks:
>
> 1. one process at a time needs to read an updated value written by other processes or
>
> 2. multiple processes may be writing to the same shared memory location and we do not want the writes to conflict. [3](page 8)

Due to the release consistency model used, the presence of explicit locks is necessary. As shown in Figure 4.8, some of the explicit locks have the sole purpose of assuring memory consistency and clutter the computation code. Furthermore, the barrier synchronization mechanism provided by Treadmarks can not be enforced for only a subset of the processes doing the cooperative computation. This affects the overall performance of the computation, by slowing down all the nodes of the system to the speed of the slowest node. This is not the case when using System V synchornization mechanisms, like semaphores.

4.5 Conclusion

As shown in the previous sections, the System V implementation of the application is straight forward and is very intuitive. While System V suffers from limited parallelism, restricted by the number of processors existent on a single machine, the overhead of making a System V implementation distributed by using the KDIPC system is virtually nonexistent. Furthermore, the overhead of KDIPC and System V specific code as compared to the TreadMarks implementation is significantly less. The observations made during this case-study support the design decision to extend System V IPC API to a distributed environment.

```
for (*(self.time) = 1; *(self.time) <= timesteps; *(self.time)=*(self.time)+1) {
  oldptr = (*self.time - 1) % LAYERS;
  curptr = *self.time % LAYERS;

  Tmk_barrier(self.time);

  Tmk.lock.acquire(neighb[NORTH].id);
  Tmk.lock.release(neighb[NORTH].id);
  Tmk.lock.acquire(neighb[SOUTH].id);
  Tmk.lock.release(neighb[SOUTH].id);
  Tmk.lock.acquire(neighb[EAST].id);
  Tmk.lock.release(neighb[EAST].id);
  Tmk.lock.acquire(neighb[WEST].id);
  Tmk.lock.release(neighb[WEST].id);


  Tmk.lock.acquire(Tmk_proc_id);
  for (g.row = self.start.row; g.row < self.stop.row; g.row++) {
      for (g.col = self.start.col; g.col < self.stop.col; g.col++) {
        off = INDEX(curptr, mtx.rows, mtx.cols,g.row, g.col);
        mtx.region[off] = new_value(&mtx, oldptr, g);
      }
  }
  Tmk.lock.release(Tmk_proc_id);

 }
```

Figure 4.8: TreadMarks: Performing computations

Chapter 5

Implementation

This chapter presents an overview of the implementation of the KDIPC system, with details on the key elements of protocols used in the implementation. Design and implementation decisions are discussed in the context of the related work presented in Chapters 2 and 3.

An overview of the system is presented in the first part of this chapter, followed by a detailed discussion of the interactions between different components of the system. The chapter concludes with a discussion of the challenges of the implementation. A thorough discussion of the changes made to the Linux kernel and the interaction between the components and the kernel data structures is provided in Appendix D.

5.1 Overview

A brief introduction of the design decisions for the KDIPC system has been presented in Sections 2.4.2 and 3.6.

Using System V IPC as the API of the system requires the use of the same consistency model provided by System V IPC for parallel programs. There are two obvious approaches to implementing sequential consistency, as follows:

- Keep replicated copies of the shared objects and use a total order communication protocol to enforce an order on accesses to replicated copies of the same shared object
- Maintain a single active copy of the shared object, which is passed around as access to it is requested.

The KDIPC prototype uses the second approach. The decision to use the single copy approach was made to keep the prototype simple, but, in retrospect, the first approach might have been equally simple. The modularity of the system allows the replacement of the protocol without major changes to the system. As it will be discussed in Chapter 6, the investigation of multiple protocols for maintaining sequential consistency makes the ground for future research avenues.

System V IPC API is currently present in the Linux operating system as a set of user-level system calls. The KDIPC system makes use of the same interface as the System V API and is implemented as a set of kernel modules that are hooked to the System V IPC system calls. KDIPC could have also been implemented as a user-level system, with minimal kernel modifications. While this second approach was more appealing from the implementation point of view, since it involved less interaction with the kernel, it also suffered from performance penalty due to the large number of context switches between kernel and user space at each system call. A series of benchmarks and tests (presented in Appendix C) proved the advantages of having a kernel level implementation. In this respect, KDIPC is different from most of the other IPC systems (especially distributed shared memory systems) that are implemented as user-level applications. As presented in Appendix C, implementing KDIPC as set of kernel modules can help compensate for the use of a strict consistency model.

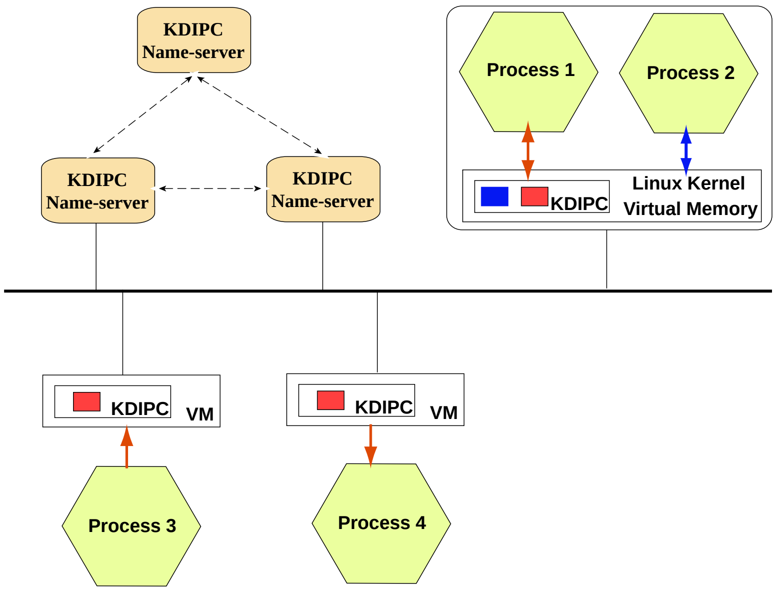

Figure 5.1: Overview of the KDIPC system

The overview of the KDIPC system is presented in Figure 5.1. KDIPC is composed of a set of KDIPC Name-Servers and a set of kernel modules local to each node. The name servers handle the distributed key resolution, and keep track of where different System V keys were requested. A set of kernel modules, depicted by KDIPC blocks in the figure, interact with the kernel and the data structures associated with the virtual memory system to maintain consistency of the shared memory segments. The modules also handle the inter-node interaction through a communication layer defined by a kernel level communication library (see Appendix B).

By design, the KDIPC system is modularized and its interference with the Linux kernel and the kernel data structure is limited.

Type	Remark
KSHM_REQ_MESG	request for information identified by an id, and associated with a shared memory segment
KSHM_UPDT_MESG	update message sent from the name-server to other name-servers and to certain nodes, containing updates of the information identified by a given key and associated with a distributed shared memory segment. Information usually refers to the set of nodes in the system interested in that distributed key
KSEM_REQ_MESG	request for information identified by a distributed key, and associated with a semaphore set
KSEM_UPDT_MESG	update message sent from the name-server to other name-servers and to certain nodes, containing updates of the information identified by a given key and associated with a distributed semaphore set. Information usually refers to the set of nodes in the system interested in that distributed key
KDIPCNS_REPLY	reply to a request for information associated with either shared memory segments or shared semaphore sets

Table 5.1: Messages processed by name-servers

5.1.1 KDIPC name servers

KDIPC name servers have a functionality very similar to that of network name server: naming resolution. Whenever a distributed key is used on a node in the system, the node interacts with the name servers to find out if the key has been previously used and which are the nodes that have previously used it. Table 5.1 explains the meaning of messages exchanged between the name server and remote Client(see 5.1.2.4) threads and provides a view of the possible interactions between these entities.

5.1.2 KDIPC kernel modules

The architecture of the KDIPC modules and their interaction is presented in Figure 5.2. The KDIPC daemon provides the interface between the kernel and the other KDIPC modules located on remote machines. When inserted in the kernel, it starts a listening thread, a supervising thread, a pool of server threads and a pool of client threads. The functionality of each is described below. The KDIPC module is hooked into the kernel System V API system calls. See Appendix D for details on how this is done at the kernel level.

5.1.2.1 The Listener

The listening thread waits for incoming connections from remote KDIPC client threads. Whenever a new connection is observed the information about the connection is saved in a *connections* list, and the supervisor thread is notified.

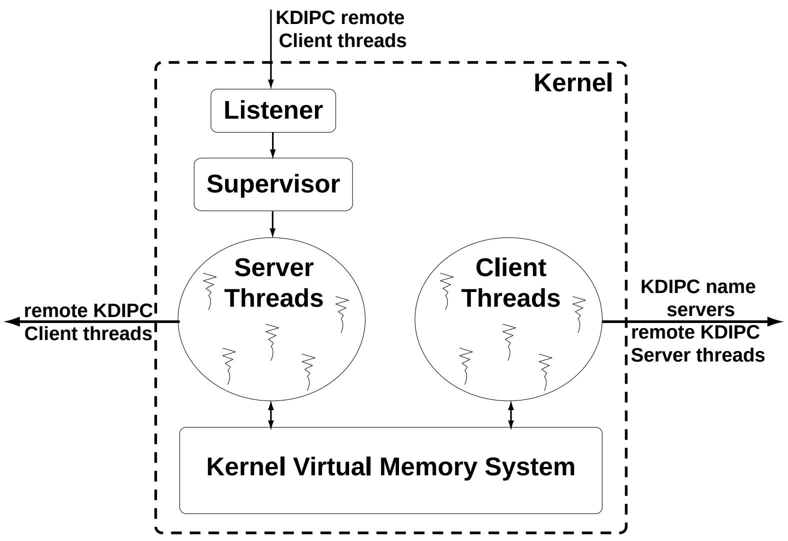

Figure 5.2: Architecture of KDIPC modules

The listening thread acts as a gateway between remote machines and the local processing threads.

5.1.2.2 The Supervisor

The supervisor thread manages the connections list created by the listener. When data becomes available on any of the connections it moves the connection to an *active_connections* list and notifies one of the server threads that information needs to be processed. Although not very intuitive, the separation between the functionality of the Listener thread and the Supervisor thread is very clear. The Listener thread reacts to incoming connections from remote client threads, while the Supervisor thread reacts to incoming messages (requests) on the already established connections.

5.1.2.3 The Server

The server threads, along with the client threads, are the core of the interface between the local KDIPC modules and remote KDIPC kernel modules. The server threads process information sent by remote client threads. The messages processed by the server threads are described in Table 5.2. Server threads also interact with the kernel data structures in order to provide the information requested by remote clients.

Since the current implementation of the KDIPC keeps only one active copy of each shared object (shared memory segment or semaphore set), nodes that need to access these objects have to retrieve them from remote location. The server thread has to respond to such requests initiated by remote clients and, when sending the shared object, it has to make sure that the local data structures are in a consistent state so that subsequent local accesses to the object are handled properly and do not generate unexpected faults at the process or the kernel level.

KDIPC_REQUEST_SHM	remote KDIPC Client thread requests a certain part of a shared memory segment
KDIPC_REQUEST_SEM	remote KDIPC Client thread requests a semaphore set
KDIPC_REPLY_SHM	reply sent by KDIPC Server thread containing requested shared memory data
KDIPC_REPLY_SEM	reply sent by KDIPC Server thread containing requested semaphore set
KDIPC_EMPTY	reply sent by KDIPC Server thread if requested shared memory object is not currently present
KDIPC_NONE	reply sent by KDIPC Server thread if requested shared memory object has not been defined locally
KDIPC_NOTIFY	notification sent to remote node when a previously requested shared memory object becomes available locally

Table 5.2: Messages processed by Server Threads

5.1.2.4 The Client

Client threads are the entities that request shared objects from remote locations, to allow local processes to access them. Clients become active when a memory page or a semaphore set is accessed by a user space process and is not currently present on the local machine. The kernel identifies the shared object and the key associated with it and wakes up one of the client threads. The client thread requests the needed information either from a KDIPC name-server (in case of a new segment allocation) or from a remote KDIPC server thread (in case a previously defined shared memory page or semaphore set are accessed).

The interaction between client threads and remote nodes (name servers or server threads) is done through the messages presented in Tables 5.1 and 5.2.

5.2 Protocol description

This section presents details about the interaction between different components of the system, as specified by the protocol implemented to maintain sequential consistency of shared objects used by the KDIPC system. Throughout this section the term shared object will be used to refer to both shared memory segments and shared semaphore set, when there is no distinction between the ways the system treats the two IPC structures.

5.2.1 Shared Object Creation/Mapping

The process of creating or mapping a shared object into the local space of a process is presented in Figure 5.3. The user process calls the shm/semget system calls, which transfers control to the kernel. One of the parameters of the system call is the distributed KDIPC key. In the figure this is represented by the *red* and the *blue* parameters to the functions. If the key given to the shm/semget function has been previously seen by the kernel then the information associated with that particular key is stored locally and a local *id* is immediately returned to the user. The information associated with the key refers to permissions, size and list of other nodes in the system that have requested the key through a shm/semget system call. In the case of shared memory, if the process later requests the mapping of the shared memory segment to its local memory space, through a shmat system call the returned *id* is used to identify the mapping.

If the key has not been previously seen by the KDIPC kernel module, the request is forwarded to the nearest KDIPC name server. The name server runs a discovery protocol to find the information associated with the key. It contacts other name-servers in the system and it retrieves the information it needs. When the discovery is complete it returns the information to the node that initiated the request. Once the information is received by the requesting node a local *id* is generated and the system call returns control to the user.

The discovery mechanism used by the name servers in the current version of the KDIPC system is mostly static. However, a scheme similar to that of IP name servers can be employed for the KDIPC name servers without significant changes to the protocol

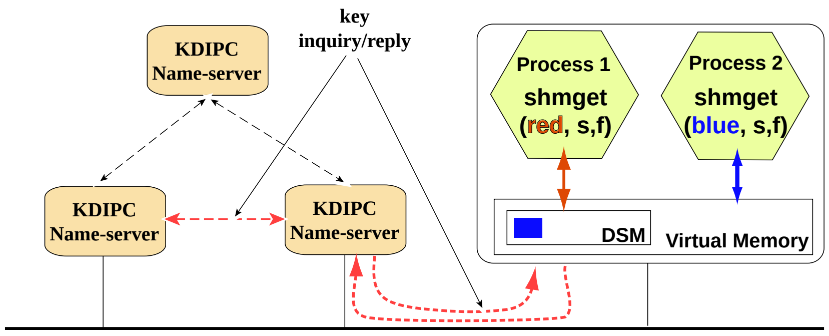

Figure 5.3: Create a new shared object. The shared memory segment identified by key *red* is seen for the first time at this node, so a discovery mechanism is initiated, to find out the information associated with this key.

In the current implementation of the KDIPC system the delay for retrieving the information associated with a given distributed IPC key incurs during the call to the shm/semget system call. The synchronicity of this approach provides a simpler processing mechanism for accesses to the shared objects. Another approach considered during the design of KDIPC involves an asynchronous model for retrieving information associated with a distributed key. In this second approach the system call would return control to the user process right away and the retrieving of information happens asynchronously. Since there usually is a delay between the mapping of a shared objects to the local space of a process and the first access to the shared object, the asynchronous approach could improve the overall performance of the system calls. However, this approach requires slightly more complex synchronization mechanisms between user processes and the kernel to guarantee that the information associated with the distributed key has arrived and is up to date before access to the object is granted. Future versions of the KDIPC system will include the asynchronous protocol for retrieving information associated with a distributed IPC key.

5.2.2 Shared Objects Accesses

The process of accessing shared objects, is different based on the kind of object being accessed. The KDIPC system deliberately implements different retrieval mechanism for remote shared memory pages and remote shared semaphore sets to allow for comparison between the different approaches. This section presents the two mechanism and protocols used when processes access shared memory segments and shared semaphore sets.

5.2.2.1 Shared Memory Accesses

Accesses to shared memory pages are processed by the Linux kernel Virtual Memory system. The access is successful if the page is present locally, or a page fault is generated by the Virtual Memory system if the page is marked as not being locally present.

If the page is present locally then the node is currently the *owner* of the page. This means that no other node can perform read or write operations on that particular page, and the operation issued by the process can be performed without any delay.

If a page fault has been generated, the page is not present. We distinguish two cases :

- The page has been present at some moment in the past but has been requested by another node
- This is the first time the page has been accessed at this node.

In both cases the latest copy of the page has to be retrieved from a remote location. Different actions are taken once the page becomes present locally, since in the second case the page has to be created in the process's virtual memory space.

The process of retrieving a page from a remote location is presented in Figure 5.4. Process 3 tries to write to a page that is not locally present. The page is associated with distributed key *red*. The node on which Process 1 is running is the current owner of the page. The node on which Process 4 is running has seen the page in the past, or it has shown interest in it, but it does not currently own it. As described in Section 5.2.1 each node keeps a list of nodes which have shown interest in the distributed IPC key associated with the shared memory segments. This list is updated by the name servers every time a new node shows interest in a given key or when a node deletes the information associated with the key, due to the fact that all processes running on that node have deleted the shared memory segment using shmdt.

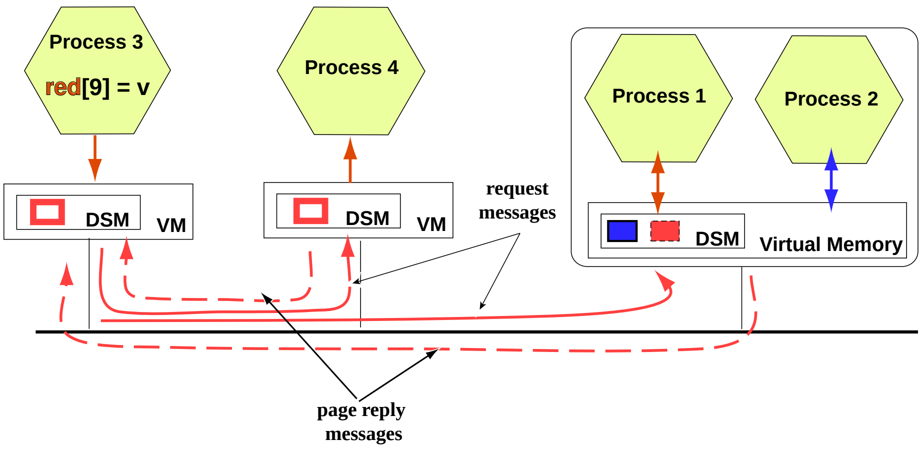

Figure 5.4: Retrieve a page from a remote location.

The KDIPC kernel module identifies which segment the required page belongs to and identifies the remote nodes that could possibly currently own the page. The obvious approach is to contact each of the remote nodes, find the one that currently owns the page and request the page from that node. While this approach is straightforward, this naive implementation has one major flaw, presented in Figure 5.5.

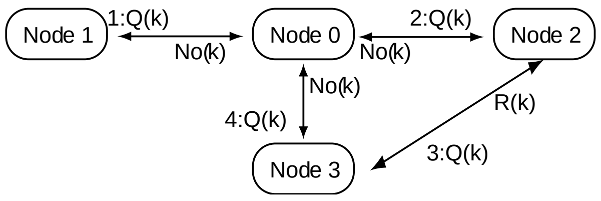

Figure 5.5: Find a remote page. Node 0 tries to retrieve page associated with key k. Q(k) is a query message for key k. No(k) is a not present message in reply to the query. R(k) is a return page message. The number preceding the message represents global time, when relevant.

The problem with the naive implementation is that the discovery of the location of the page is not atomic and multiple nodes can look up the same page at the same time. In the example presented in Figure 5.5, both Node 0 and Node 2 are looking for page associated with distributed key k. Node 3 is the owner of the key at global time 0. While Node 0 tries to locate page k, by contacting Nodes 1,2 and 3, in this order, the page could migrate from its current location (Node 3) to some other location. Assume this happens right after Node 0 asks Node 2 if it owns the page, but before it asks Node 3. The event leading to this situation could be generated by the fact the page k is requested by Node 2 from Node 3. Node 3 grants ownership to Node 2 and the page is not located at Node 3 when the request form Node 0 arrives. In this case Node 0 has exhausted all the possible locations of the page, and has not been able to locate it. Re-iterating through the nodes does not solve the problem but it rather clutters the protocol and increases the communication overhead of the protocol, since, in the worst case scenario, the page can move to a remote location every time Node 0 is about to ask for ownership the current owner of the page.

In the general case, consider the set of nodes in the system which are interested in a given key k to be

$N_k = \{\text{nodes } n_i \mid n_i \text{ executed a process that issued a shmget operation with parameter } k\}$.

Let $owner_k$ be the node which is the current owner of shared memory page associated with key k ($owner_k \in N_k$). For simplicity, we consider that each key represents only one shared memory page. This does not affect the generality of the analysis but it makes the reasoning clearer. Assume node $n_a \in N_k$ is interested in becoming the owner of the page, and $n_a \neq owner_k$. According to the protocol presented, at discrete times t_i, node n_a asks node n_i if it owns the page and if it can transfer ownership to it. There is an obvious starvation issue if node n_i asked at time t_i becomes the owner of the page at time t, where $t_i < t < t_{i+1}$.

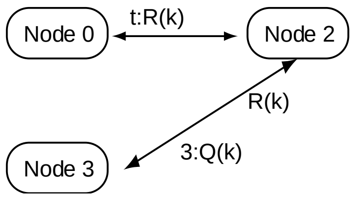

Figure 5.6: The notify mechanism for page location. Node 2 keeps track of Node 0's request. After finishing its operations on page k, Node 2 asks Node 0 if it still needs the page and, if so, it sends it over at global time t ($t > 3$)

To address this problem, the following protocol is used. When a node can not serve a request for page ownership received from a remote location, it keeps track of the request. If the requested page becomes available locally at some moment in the future it notifies the requester that it currently has ownership of the page. If the requester still needs the page, the page is sent to the remote node after the current owner has served some of the local request to access the page. This protocol is illustrated in Figure 5.6 in the context of the situation presented in the example above. When Node 2 receives the request for page k from Node 0, and is not able to satisfy it, it records the request in a queue. When Node 2 is granted ownership of page k from Node 3, it takes the following actions:

- first, it processes the local accesses to the page
- next, it notifies Node 0 that it currently owns page k and that it has an outstanding request for this page from Node 0

. If Node 0 still needs the page it replies positively to Node 2's inquiry and the page ownership is then transfered to Node 0.

This protocol guarantees that for each requested page the number of messages is proportional to the number of nodes interested in that page, and that a page will eventually arrive at the node requesting it. This defines a key liveness property of the system.

5.2.2.2 Shared Semaphore Set Access

Accesses to shared semaphore sets are different from accesses to shared memory pages because accesses to shared semaphore sets may be postponed until the semaphore set is in a state to satisfy the access. The situation when an access can be blocked is described next. As described in Section 3.4, operations on semaphore sets are performed through calls to semop system call. The operation on a semaphore set consists on changes to the values of the semaphores in the set, as defined by the parameter passed to the semop system call. Figure 5.7 presents two different attempts to access a shared semaphore set. The values of the semaphores in the semaphore set S are presented in the figure, along with the opreations defining the accesses. We assume that both operations would be applied to the initial value of the semaphore set. The first operation would not block since the changes in the values of the semaphores keep the final values above 0. The second operation would block the intiating process because semaphore 4 has only two resources, while the operation requests three such resources. The operation is then blocked until another operation performed on the semaphore set makes this one non-blocking. We will further call an operation that can not be performed right away due to insufficient resources a blocking operation.

Semaphore set S has the following initial values

semaphore	0	1	2	3	4	5
value	1	0	5	3	2	4

Operation	Resulting semaphore set	Comment
semop(S,[-1,0,0,-1,-1,0])	[0,0,5,2,2,4]	Successful (non-blocking), since the resulting semaphore set has only positive values
semop(S,[-1,0,0,-1,-3,0])	[0,0,5,3,-1,4]	Unsuccessful (blocking), since the resulting semaphore set would have negative values for individual semaphores. Process is put to sleep until the values of the semaphores allow the operation to be performed.

Figure 5.7: Accesses to a shared semaphore set

When an access to a distributed semaphore set is issued by a process two cases are distinguised:

- The semaphore set is present and currently owned by the node where the process is running.
- The semaphore set is not present, in which case it has to be retrieved from a remote location.

If the semaphore set is present on the local host, the operation is attempted. If the operation is blocking due to lack of resources, the process that initiated it is put to sleep and the operation is stored in a queue associated with the semaphore set. Each semaphore set will be associated with a queue of sleeping processes. The queue stores both the process id and the node id on which the process was running, along with the operation that forced the process to sleep. When ownership is transfered from one node to another the queue associated with the semaphore set migrates along with the semaphore set. This queue is used to wake up sleeping processes when the values of the semaphores in the semaphore set allow a certain operation to be performed.

If the semaphore set is not present on the local host, the host has to locate the semaphore set and retrieve it from a remote location. The discovery protocol used for semaphores is different from the one used for shared memory. In this case each node keeps track of where it has last sent the semaphore set to. When trying to locate a semaphore set, a node sends a request to the node it last sent the semaphore set to. If the receiver of such a request does not currently own the semaphore set it forwards the request to the node it itsef has sent the semaphore set to. At the same time, it keeps track of the fact that it has not been able to satisfy the request, in a manner similar to that used for shared memory pages. If the semaphore set becomes locally owned at some moment in the future the pending requests are processed in a way similarly to the one described in the protocol for shared memory pages. If a node receives a request while it owns the requested semaphore set, it transfers ownership directly to the original requester.

An alternate discovery algorithm is the following. When a request for access to a semaphore set reaches the owner of the semaphore set, the operation is processed locally and only the outcome of the operation is sent to the orginiator of the access. The decision to either process the operation locally or send the semaphore set to the node it requested the access can be taken based on the ration of accesses at the current host vs. the remote location. If the operation is processed locally the two possible outcomes: operation is successful (non-blocking), or operation is blocking. In the second case the process issuing the operation will be put to sleep on the originating node and the current owner of the semaphore set adds the operation, the process id and the originator node to the queue associated with the semaphore set. This alternate protocol is considered for future versions of the system. The main advantage of this approach is that it uses both the existing protocol and the presented extension. The parameter used in deciding how to satisfy a request can be dynamically adjusted and the analysis of its effect on the performance of the protocol is interesting for future research.

5.3 Challenges of the implementation

The main challenge of the current implementation was raised by the decision to implement the functionality of the system at the kernel level. In general, the existent documentation on Linux kernel internals is very scarce. The dynamism of the kernel development has also proved to be a major issue. The KDIPC system was initially designed for the Linux 2.2 kernel distribution. The major design change suffered by the Linux Virtual Memory system that came along with the Linux 2.4 kernel series required the KDIPC system to be completely re-designed. Since the Linux 2.4 kernel involved a new Virtual Memory system, the documentation explaining the new design was practically nonexistent. This has significantly slowed down the development of the KDIPC system. Most of the initial development was made by trial and error. The data structures of the kernel interacted in unpredicted and undocumented ways to the changes necessary by the KDIPC system.

Another issue related to the Linux kernel is the complete lack of network communication libraries at the kernel level. Since the Linux kernel is not designed to be part of a distributed environment and communicate with other kernels, there are no primitives for communication implemented inside the kernel. The KDIPC system required this functionality, so a socket library for the kernel had to be implemented. Details of this implementation are found in Appendix B.

The kernel threads library in the early Linux 2.4 kernels suffered from problems as well. There were a few existent work-arounds to get real parallelism of the kernel threads, but they turned out to be insufficient for the functionality of the KDIPC threads. Further changes were required in the kernel, and a thread library based on the previous work-arounds has been used in the implementation of the KDIPC system.

As a general comment, kernel development can be very slow, especially when dealing with critical parts of the kernel, like the Virtual Memory system. Almost every bug existent in the KDIPC code generated unexpected, and some times unpredictable, behavior of the kernel, which led almost without exception to the hang of the system. Debugging the kernel is not usually an option, since the corruption of the kernel space renders any debugging tool useless. Speeding up the development/recovery process was possible through the use of virtual machines, like VMWare [5]. A similar abstraction of a Linux machine is provided by the UserMode Linux [4] project.

Chapter 6

Conclusions and Future Work

This thesis presents a new Inter-Process Communication library, developed for distributed environments. The Kernel-level Distributed Inter-Process Communication (KDIPC) system extends the System V IPC Application Programming Interface to distributed environments. It provides functionality for dsitributed shared memory and distributed semaphore sets through normal System V IPC system calls. Since it extends the System V IPC it also preserves its semantics. The sequential consistency model currently supported by the KDIPC system involves maintaining a single valid copy of the shared objects. To improve the performance of the system and to make the overhead acceptable for distributed systems, the functionality is implemented as part of the kernel. The kernel module approach of the implementation reduces the number of context switches for each system call and allows for more time to be invested in the consistency protocol.

6.1 Initial Benchmarks

In order to evaluate the design and performance of the KDIPC system a series of experiments was conducted. The results presented in Figure 6.1 were obtained by running different versions of a matrix multiplication program: a multi-threaded parallel matrix multiplication, a shared memory version using System V IPC, a distributed version using message passing, and the distributed shared memory version compiled with our system.

The implementation of the matrix multiplication programs uses the naive approach, where for each element in the result matrix we need one row and one column from the multiplying matrices, respectively. The time spent for computation (including memory accesses and communication) for each of the four implementation is discussed in the next Section.

6.1.1 Analysis

For each version of the matrix multiplication program there is a main thread that creates the matrices to be multiplied and separate threads that do the actual computation of the result matrix. Square

Matrix	Version of matrix multiplication			
Size	Par	ShMem	MsgPass	DSM
100	0.006	0.01	0.183	0.08
200	0.046	0.06	0.583	0.34
1000	12.616	11.35	14.106	12.983

Figure 6.1: Effective computation (in sec)

matrices were used for these tests and the split of the computation for the result matrix was in four equal submatrices, having one thread computing each quarter. The times shown in Figure 6.1 measure the best time obtained from running all the four threads computing the result matrix. For the parallel version of the matrix multiplication program the results shown include the time to start the cloned thread from the main thread, but it does not include the actual time of starting the entire process. All the other times include the time spent in starting the processes.

As it can be seen from Figure 6.1, the KDIPC version of the DSM implementation always performs better than the message passing version and it is not much worse than the parallel version or the local shared memory version. The main point is that that none of the two versions that perform better on a per thread basis can outperform the total running time of the DSM matrix multiplication program because they are bound to run on a single computer. Even when we split the matrix in four on our dual-processor machines, the running time of both parallel and shared memory version is at least twice the running time of one of the threads.

6.2 Future Work

The current version of the KDIPC system has a few mechanisms and protocols that were used for the simplicity in the initial prototype. The plan is to upgrade them to the alternate protocols presented throughout this thesis.

We plan to investigate different protocols for maintaining sequential consistency in the KDIPC system. This research would require a thorough comparison of the single copy protocol with protocols that provide total order of operations or serializable sequential consistent orders of the operations.

Future research avenues also include the use of speculations and process migration to improve the performance of the KDIPC system. Part of this effort is presented in Cristian Ţăpuş et al. [9].

We also want to use the KDIPC system as an infrastructure for different kinds of applications, like distributed file systems and distributed objects repositories. This work is related to the two previously presented directions of research.

Appendix A

TreadMarks API

The TreadMarks system requires application to use the following routines and variables in their code [3]:

```
/*
 * the maximum number of parallel processes supported by TreadMarks
 */
#define TMK.NPROCS

/*
 * the actual number of parallel processes after Tmk.startup
 * (Initialized by Tmk.startup.)
 */
extern unsigned Tmk.nprocs;

/*
 * the process id, an integer in the range 0 ... Tmk.nprocs - 1
 * (Initialized by Tmk.startup.)
 */
extern unsigned Tmk.proc.id;

/*
 * the maximum number of shared memory pages
 */
#define TMK.NPAGES

/*
```

```
 * the shared memory page size * (Initialized by Tmk.startup.)
 */
extern unsigned Tmk.page.size;

/*
 * the number of lock synchronization objects provided by TreadMarks
 */
#define TMK.NLOCKS

/*
 * the number of barrier synchronization objects provided by TreadMarks
 */
#define TMK.NBARRIERS

/*
 * Initialize TreadMarks and start the remote processes.
 */
void Tmk.startup(argc, argv)
int argc; char **argv;

/*
 * Terminate the calling process. Other processes are unaffected.
 */
void Tmk.exit(status)
int status;

/*
 * Block the calling process until every other process arrives
 * at the specified barrier.
 */
void Tmk.barrier(id)
unsigned id;

/*
 * Block the calling process until it acquires ownership
 * of the specified lock.
 */
void Tmk.lock.acquire(id)
unsigned id;

/*
 * Release the specified lock.
 */
void Tmk.lock.release(id)
unsigned id;

/*
 * Allocate the specified number of bytes of shared memory, aligning
 * the block on a page boundary if the block's size is at least
 * Tmk.page.size. Return NULL if the shared memory heap is exhausted.
 */
char *Tmk.malloc(size)
unsigned size;

/*
 * Free shared memory allocated by Tmk.malloc.
 */
void Tmk.free(ptr)
char *ptr;

/*
 * Allocate the specified number of bytes of shared memory. Return
 * NULL if the shared memory heap is exhausted. N.B. It is
 * impossible to free a block of memory allocated by Tmk.sbrk.
 */
char *Tmk.sbrk(incr)
int incr;

/*
 * Distribute (copy) the contents of a block of PRIVATE memory on the
 * calling process to every other process. After the call completes
 * every process has identical values in the specified memory
 * locations. Ordinarily used by a program’s initialization code to
 * distribute ‘‘root’’ pointers to shared data structures.
 */
void Tmk.distribute(ptr, size)
char *ptr; unsigned size;
```

Appendix B

The kernel socket library module

This component of the system was developed because socket communication at kernel level was needed. In order to make development of future kernel level applications easier set of standard socket functions has been developed:

- void ksock_getaddr(char *hostname, int port, struct sockaddr_in* addr);
- struct socket *sock_alloc(void)
- int ksock_create(struct socket** skp);
- int ksock_bind(struct socket *sock, char *hostname, int port);
- int ksock_listen(struct socket *sock, int qlen);
- int ksock_accept(struct socket* lsock, struct socket** asock, int flags);
- int ksock_connect(struct socket *sock, char *hostname, int port);
- int ksock_send(struct socket *sock, void *mesg, int len);
- int ksock_recv(struct socket *sock, void *mesg, int len, int flags);
- void sock_release(struct socket *sock);

Appendix C

Initial Benchmarks

To determine on which side of the kernel-user boundary the KDIPC implementation should be, a set of benchmarks were used to measure the context switch time between two processes and the time required to send a large block of data across the network in a Local Area Network (LAN).

The results obtained from running the LMBench [19] benchmark on the Mojave cluster are presented in Figure C.1. The cluster consists of 16 machines, each with 700MHz dual processors, running Red Hat 8.0 and the 2.4.18 Linux kernel, and connected through a 100Mbit network. The results indicate that when there are more than 5 actively running processes on one node with size of at least 128k each, the context switch time is at least as long as sending 400 bytes of data on a TCP connection from one machine to the other.

Context switch time (in μs)

	Number of processes			
Process Size	2	5	10	20
size=64k	7.77	46.01	99.90	123.00
size=128k	69.64	196.19	233.09	232.89
size=200k	277.29	281.24	289.84	287.34

Network latencies for remote host (in μs);
includes context switch time for two processes

UDP latency using mojave3 (350 bytes):	187.1849
UDP latency using mojave3 (600 bytes):	272.3542
TCP latency using mojave3 (400 bytes):	167.2341
TCP latency using mojave3 (4000 bytes):	342.1603

Figure C.1: Benchmarks for Mojave Cluster

Based on some previously collected benchmarks [20], we observed that the relative times spent in context switching and in network latency have changed over the last seven years toward supporting the assumption that in todays computing environments context-switching time and network latency are on the same order of magnitude. We concluded that moving the page allocation mechanism inside the kernel could compensate for a consistency algorithm that uses more communication in order to provide a stricter consistency model.

Appendix D

Details of the implementation

This Appendix describes details of the implementation of the KDIPC system. While this is not a complete coverage of the entire implementation, it shows the main steps taken to implement the KDIPC system at the kernel level. Each modules composing the KDIPC system will be described through its interaction with the kernel and the kernel data structures.

The KDIPC system is composed of the following modules: *kdipcns*,*kdipcd*, and *kdipc*. The *kdipcns* module runs the KDIPC name server thread. The *kdipcd* module runs all the threads presented in Figure 5.2. The *kdipc* module contains all the glue code that makes possible the interaction between the kernel and the KDIPC modules. Each will be described in more detail in Section D.2.

The KDIPC modules can be inserted and removed at will from the kernel. In order to seamlessly support this feature certain changes had to be made to the actual Linux kernel. These will be discussed in Section D.1

D.1 KDIPC kernel changes

This section discusses the changes incurred by the Linux kernel during the implementation of the KDIPC system. The changes will be presented in the context of the kernel files they belong to. Only the changes relevant to the protocols implemented by the KDIPC system are presented. A patch containing all the changes can be found as part of the distribution of the KDIPC system.

D.1.1 include/linux/ipc.h

The *ipc.h* header file contains the definitions of the flags used by the System V IPC system calls and the data structures associated with IPC keys used by the kernel. The changes to the ipc header file were minor, but rather important. First, the *IPC_KDIPC* flag that characterizes the distributed objects was defined in this file. Next, a new field has been added to the *kern_ipc_perm* data structure. This data structure, shown in Figure D.1, contains the IPC key, user and group ids of the process who created it, the permissions associated with the key and a sequencer used for generating local

```
/* used by in-kernel data structures */
struct kern_ipc_perm
{
        key_t           key;
        uid_t           uid;
        gid_t           gid;
        uid_t           cuid;
        gid_t           cgid;
        mode_t          mode;
        unsigned long   seq;
        kdipc_info_list_t info;
};
```

Figure D.1: The kern_ipc_perm data structure in *ipc.h*

```
typedef struct kern_kdipc_info_list {
  struct semaphore mutex; // a mutex protecting the info list
  unsigned long policy; /* the request policy used */
  unsigned long *pflags; /* flags - PSTATE_* SSTATE_* */
  unsigned long *ppte; // used with shared memory only
  struct vm_area_struct *vmap; // used with shared memory only
  wait_queue_head_t *task_q;
  int nr_ips; /* the nr of ip addresses - for debugging */
  struct list_head ip_list;
} kdipc_info_list_t;
```

Figure D.2: The kdipc_info_list_t data structure defined in *kdipc.h*

ids associated with the key. The new field, called *info*, contains information specific to the KDIPC system, and is needed in the data structure associated with the System V IPC. The type of this new field is *kdipc_info_list_t* and is described in Section D.1.2, since it belongs to file *include/linux/kdipc.h*. The decision to enchance kernel data structures with KDIPC specific fields was made to be able to re-use the helping functions used by the kernel and to better integrate the KDIPC system with the Linux kernel.

D.1.2 include/linux/kdipc.h

The *kdipc.h* header is specific to the KDIPC system and did not exist in the Linux kernel distribution. This file contains the definitions of the flags used for marking the state of shared memory pages (PSTATE_*) and semaphore sets (SSTATE_*) and to define data structures used by the kernel to manipulate the information specific to the KDIPC system.

The possible states of the shared objects are: present, requested, allocated, fresh, and notify for the shared memory pages, and present or requested for semaphore sets.

This header file contains the definition of the *kdipc_info_list_t* type, which is presented in Figure D.2. Some of the fields are specific to the shared memory subsystem and some are used by both

```
/* these are hooks to the functions defined in kdipc.c */
extern int kdipc_shm_get(key_t key, int shmflg, int size);

asmlinkage long sys_shmget (key_t key, size_t size, int shmflg)
{
   ...
   if (shmflg & IPC_KDIPC) {
       /* we have a distributed shared memory request */
       err = kdipc_shm_get(key, size, shmflg);
   }
   ...
}
```

Figure D.3: Changes to the sys_shmget function in *shm.c*

```
/* this function pointer is changed when the module loads/unloads */
int (* kdipc_shm_get_mod)(key_t key, size_t size, int shmflg)=NULL;

int kdipc_shm_get(key_t key, size_t size, int shmflg) {
  if (kdipc_shm_get_mod) {
    // KDIPC module is loaded
    return kdipc_shm_get_mod(key, size, shmflg);
  }
  else
    // KDIPC module is not loaded
    return -EINVAL;
}
```

Figure D.4: The kdipc_shm_get function defined in *kdipc.c*

the shared memory subsystem and the semaphores subsystem, as presented in the brief descriptions shown in the figure. The mutex field is used to prevent simultaneous accesses to the fields of the data structure. The *task_q* field is used to maintain the list of processes who are blocked on the semaphore set. The *ppte* and *vmap* fields are used to keep track of the page table entries associated with the shared memory segment and the mapping of these pages in the process' virtual memory.

Most of the other data structures defined in this file are for administrative purposes only and are not significant to the description of the KDIPC system.

D.1.3 ipc/sem.c, ipc/shm.c, and ipc/kdipc.c

This section presents the kernel hooks needed to integrate KDIPC with the System V system calls. We are only going to focus on one particular example, the changes needed for the *shmget* system call. The changes required by the other system calls are similar, and they can be seen in detail in the patch to kernel available with the distribution of the KDIPC system.

Each System V IPC system call was added "hook" code, which identifies the presence of the KDIPC kernel module and calls the module functions if the functionality of the KDIPC is needed.

```
extern int (* kdipc_shm_get_mod)(key_t key, size_t size, int shmflg);
int kdipc_shm_get_module(key_t key, size_t size, int shmflg);

/* Initialize the module */
int init_module(void) {
  /* overload the kernel functions (which are supposed to be NULL) */
  kdipc_shm_get_mod = kdipc_shm_get_module;
  ...
}

void cleanup_module(void) {
  kdipc_shm_get_mod = NULL;
  ...
}
```

Figure D.5: Initialization and clean-up functions for the *kdipc* module

If the kernel module is not present and distributed functionality is requested by a user process the system returns an error to the user, without doing anything to affect the correct functionality of the kernel.

The additions to the sys_shmget functions are presented in Figure D.3. When a new key is declared with the IPC_KDIPC flag, the KDIPC functionality is required, and the *kdipc_shm_get* function is called. This function is not normally present in the kernel, and it is called a "hook" function, since it provides the KDIPC system a way to hook itself to the Linux kernel. The code for the "hook" function is shown in Figure D.4. When the kdipc modules are inserted into the kernel, the *kdipc_shm_get_mod* function pointer is updated with the value of a function inside the module. When the module is removed, this value is returned to NULL.

Similar changes have been added to all System V system calls: *sys_shmat, sys_shmdt, sys_semget, sys_semop*, and so on.

D.2 KDIPC modules

The three KDIPC modules will be discussed in this section. First, the functionality of the *kdipc* module is presented, which is the module that provides the glue code for the kernel "hooks". Next, the key elements of the KDIPC system: the Client and the Server threads will be presented.

D.2.1 KDIPC glue code - the *kdipc* module

Section D.1 described the changes made to the Linux kernel. One of the elements presented there was the existence of external functions that are executed in case the KDIPC functionality is required. This section continues the illustration of the KDIPC extensions to the *shmget* system call.

When a module is inserted in the kernel, the *init_module* function is called to do the initialization required by the module. On removal, the *cleanup_module* function of the module is invoked. Figure D.5 shows the initialization and the cleanup function of the *kdipc* module. The init function changes the function pointers declared in the *ipc/kdipc.c* to point to the functions defined by the module. The cleanup function reverts the values of the function pointers back to NULL.

The pseudo-code for the function that is executed when the KDIPC functionality is required by the *sys_shmget* system call is present in Figure D.6. The *kdipc_shm_get_module* function is the one that is assigned to the *kdipc_shm_get_mod* function pointer in the init function of the *kdipc* module.

First, the function checks if the *key* provided as parameter has been previously defined on the local node. The *kdipc_findkey* function checks the local kernel data structures by invoking the *ipc_findkey* function. If the key has not been previously seen locally, one of the name-servers is contacted, and the name resolution is done. This is accompanied by a retrieval of the information associated with the *key*. The information is stored in the *mesg* variable. Depending on whether the key has been seen before or not, different actions are taken. At the end of this call the data structures associated with the key are properly initialized and all the information associated with the key is locally stored. The actual contents of the shared memory segment is not retrieved at this step. This happens later, when the first access occurs. Any inconsistencies between the flags provided as parameters to the *kdipc_shm_get_module* and the permissions of the shared memory segment are reported at this time in a manner similar to that of the regular System V *shmget* system call. This is done to preserve the semantics associated with the System V IPC API.

The *kdipc* module also defines the *no_page* function which is invoked when one of the following two situations occurs:

- A page is accessed for the first time, so it has not yet been allocated to the process' memory space
- A page is accessed after it has been sent to a remote location. The page table entry is invalid at this time.

This is one of the functions that provide the interaction with the Client threads. Figure D.7 shows the pseudo-code for this function.

When the *no_page* function is invoked, it first checks the status of the page. Since we are working in an environment with parallel threads, by the time this invocation of the *no_page* function is executed another thread might have retrieved the page from its remote location. If the page is not present and it has not yet been requested by another thread, one of the client threads is given the task to retrieve the page from its remote location. Until the page is retrieved the process who generated the *no_page* fault is put to sleep. When the page becomes locally *owned* the process is woken up and a pointer to the page is returned to the process to continue its execution.

```
int kdipc_shm_get_module(key_t key, size_t size, int shmflg) {
    // locate the kdipc key. Find and retrieve the information
    // associated with the kdipc key.
    // This function contacts the name server to retrieve information
    // associated with the kdipc key if the key has not been
    // previously requested locally.
    id = kdipc_findkey(&shm_ids, key, &mesg, &mlen, IS_SHM);

    if (id<-1) return -EINVAL; // there was an error

    if (id==-1) { // the key has not been seen locally before
       if (mesg!=NULL) { // the key has been defined remotely
          // create a new VM segment locally
          err = newseg(key, shmflg, size);
          // store the information associated with the key
          process_and_store_information(mesg);
       } else { // the key has not been defined anywhere else
          if  (!(shmflg & IPC_CREAT))
             /* no create flag - error; the user expected it to exist */
             err = -ENOENT;
          else {
             // create a new VM segment locally
             err = newseg(key, shmflg, size);
          }
       }
     } else  if ((shmflg & IPC_CREAT) && (shmflg & IPC_EXCL)) {
       /* the key exists but the user wanted to create it himself */
       err = -EEXIST;
     } else {
          // the key has been previously defined locally
          // we have the information stored
          // we need to return the information to the user
          // check permissions associated with the key and build a new id
          shp = shm_lock(id);
          if (shp->shm_segsz < size)
             err = -EINVAL;
          else if (ipcperms(&shp->shm_perm, shmflg))
             err = -EACCES;
          else
             err = shm_buildid(id, shp->shm_perm.seq);
          shm_unlock(id);
     }
   return err;
}
```

Figure D.6: Pseudo-code for kdipc_shm_get_module function

```
struct  page *  kdipc_shmem_nopage_module(struct vm_area_struct * vma,
unsigned long address, int no_share) {
  /* Get the number of the page that was referenced */
  index = (address - vma->vm_start) >> PAGE_SHIFT;

  /* Get info */
  shp=vma->vm_private_data;

  if (shp->shm_perm.info.pflags[index] & PSTATE_PRESENT) {
    /* this page is already in the memory */
      pte = __pte(shp->shm_perm.info.ppte[index]);
    if(!pte_present(pte))
        return NOPAGE_SIGBUS;
    pagep = pte_page(pte);
    return pagep;
  } else if (shp->shm_perm.info.pflags[index] & PSTATE_REQUESTED)
    // It looks like the page was already requested so do nothing
  else {
    // page is not present locally, so ask a client to retrieve it
    shm_data.perm = &shp->shm_perm;    shm_data.idx = index;
    __add_kdipc_shm(&lookup_dshm, &shm_data);
    up(&kdipcd_shm_client_waiting_sem);
  }

  // wait for the client thread to retrieve the page from its remote location
  wait_for_page_to_become_local();

  // the page was returned
  pte = __pte(shp->shm_perm.info.ppte[index]);
  if(!pte_present(pte))
     return NOPAGE_SIGBUS;

  pagep = pte_page(pte);
  return pagep;
}
```

Figure D.7: Pseudo-code for kdipc_shmmem_nopage_module function

D.2.2 KDIPC daemon - kdipcd

D.2.2.1 The client thread

The client thread provides the interaction of the KDIPC system with the remote nodes in the distributed environment. When the local kernel needs to retrieve remote information the client thread is woken up and proceeds to solving the task. The tasks performed by the client threads are:

- request the information associated with a distributed key
- request a page which is owned by a remote node
- request a semaphore set which is owned by a remote node

The pseudo-code for the function performing the different tasks is illustrated in Figure D.8

When a page is retrieved from a remote location, it has to be placed in the right position in the shared memory space of the process. This is done by the *place_page* function, presented in Figure D.9.

D.2.2.2 The server thread

The server thread serves the requests initated by remote client threads. The main functionality of the server thread includes:

- reply to remote requests for shared objects
- maintain consistency of local kernel data structures when serving remote requests for shared objects

The function handling remote requests for shared memory pages is presented in Figure D.10, while the code presenting the interaction between the kernel data structures and the server thread is presented in Figure D.11.

```
void kdipcd_shm_perform_task (  kdipc_shm_list_t *shm_data  ) {
  // lock down the information so that nobody modifies it while we process it
  for_each_info(ip_info, shm_data->tokdipc, k) {
     ksock_connect_directly(tmp_sock, ip_info->addr, htons(KDIPCD_PORT));
     kdipc_send_shm_request(tmp_sock, req_mesg);
     /* wait for reply */
     mesgtype = kdipc_get_message(tmp_sock, mesg,sizeof_kdipc_shm_reply_msg_t);
     if (mesgtype==KDIPC_REPLY_SHM) {
        /* put the page where it belongs in memory */
        success=1;
        place_page(shm_data, reply_mesg);
     } else if (mesgtype==KDIPC_EMPTY) {
        /* could not find it there, try the next one */
        do_sleep=1;
     } else if (mesgtype==KDIPC_NONE) {
        // could not find it there because it was not defined; remove ip from list
     }
  }
  if (!success && do_sleep) {
    // let process sleep. We missed the page, so we wait for notifications
      q->key=shm_data->perm->key;
      q->idx=shm_data->idx;
      q->tag=TAG_DATA;
      q->shm_data=shm_data;
      append_to_queue(&notify_locally_shm, q);
      return;
  }
  if (!success) {
    /* check if the page was allocated or not */
    if ((shm_data->tokdipc.pflags[shm_data->idx] & PSTATE_ALLOCATED)==0) {
      /* the page was never allocated so we need to do that */
      pagep = alloc_pages(GFP_HIGHUSER, 0);
      SetPageReserved(pagep);

      /* Build the page-table-entry */
      pte = mk_pte(pagep, PAGE_SHARED);
      shm_data->tokdipc.ppte[shm_data->idx] = pte_val(pte);
      shm_data->tokdipc.pflags[shm_data->idx] |= PSTATE_ALLOCATED | PSTATE_PRESENT;
      shm_data->tokdipc.pflags[shm_data->idx] &= ~(PSTATE_REQUESTED);
    } else BUG();
  }
  // here we have to check if the page we received has been requested by
  // other remotes nodes while it was away. If so, we need to contact them,
  // and see if they still need the page. If they do, we have to send it
  // after certain local access requests are satisfied
  if ((shm_data->tokdipc.pflags[shm_data->idx] & PSTATE_NOTIFY)!=0) {
    //need to notify
    need_to_notify= 1;
    notify_key = shm_data->perm->key;
    notify_idx = shm_data->idx;
    notify_id = shm_data->perm->info.id;
  }
  // wake up local process, because page is here.
  wake_up_interruptible(&(shm_data->tokdipc.task_q[shm_data->idx]));
  if (need_to_notify) {
    notify_remote_processes(notify_key, notify_idx, notify_id);
  }
}
```

Figure D.8: Pseudo-code for client thread (kdipcd_shm_perform_task)

```
void place_page (kdipc_shm_list_t *shm_data, kdipc_shm_reply_msg_t *reply_mesg) {
  // set the time information for the shared data structure
  // for now we want to keep the page around for about a fifth of the time
  // it took us to retrieve it from its previous location.
  shm_data->tokdipc.duration_jiffies = (jiffies - shm_data->tokdipc.start_jiffies)/3;
  shm_data->tokdipc.start_jiffies = jiffies;

  /* check if the page was allocated or not */
  if ((shm_data->tokdipc.pflags[shm_data->idx] & PSTATE_ALLOCATED)==0) {
    /* the page was never allocated so we need to do that */
    pagep = alloc_pages(GFP_HIGHUSER, 0);

    SetPageReserved(pagep);
    /* Build the page-table-entry */
    pte = mk_pte(pagep, PAGE_SHARED);
    shm_data->tokdipc.ppte[shm_data->idx] = pte_val(pte);
    shm_data->tokdipc.pflags[shm_data->idx] |= PSTATE_ALLOCATED;
    shm_data->tokdipc.pflags[shm_data->idx] &= ~(PSTATE_REQUESTED);
  }
  else {
    // Page allocated
    pagep = pte_page(__pte(shm_data->tokdipc.ppte[shm_data->idx]));
  }
  /* now copy the data onto the page */
  addr = kmap_atomic(pagep, KM_USER0);
  memcpy(addr, reply_mesg->page, PAGE_SIZE);
  kunmap_atomic(pagep, KM_USER0);

  shm_data->tokdipc.pflags[shm_data->idx] |= PSTATE_PRESENT;
}
```

Figure D.9: Pseudo-code for client thread (place_page)

```
/*
 * Handle a request for a shared memory block from another computer
 * We need to first make sure that we have the block in memory.
 * If we don't have it, then keep in mind that block was requested
 *   so we can notify the requester when we get the block.
 * If we do have it, then we need to take it from memory and send it
 *    to the interested party. I think this part of the code generates
 *    a race condition and eventually leads to deadlock.
 */
void handle_kdipc_req_shm (char *mesg, struct socket *tmp_sock) {
  /* lookup key in the local structure */
  id = ipc_findkey(&shm_ids, key);

  /* if success then invalidate the page and return it to the requester */
  if (id<0) { // page non-existent in my local database
     emptym->msgtype = KDIPC_NONE;
     kdipc_send_empty(tmp_sock, emptym);
  } else {// id >= 0
    shp=shm_lock(id);

    if ( (shm_ids.entries[id].p->info.pflags[idx] & PSTATE_PRESENT) &&
         (shm_ids.entries[id].p->info.pflags[idx] & ~PSTATE_NOTIFY) && shp) {
      send_page_reply(id, shp, key, idx, tmp_sock);
      shm_unlock(id);  // id is locked.
    } else { // page not currently present locally
      // we need to add the information about this request in a notify queue
      // so we can later send it to the person requesting it.
      q = (shm_queue_t *) kmalloc(sizeof(shm_queue_t), GFP_USER);
      q->addr.s_addr = tmp_sock->sk->daddr;
      q->key=key;
      q->idx=idx;
      q->tag=TAG_ADDR;
      append_to_queue(&notify_shm,q);

      // we also need to mark the page so that we NOTIFY the sleeping processes
      shm_ids.entries[id].p->info.pflags[idx] |= PSTATE_NOTIFY;

      shm_unlock(id); // the id is locked so we should release it now.

      /*return a not-found message with an empty page */
      emptym->msgtype = KDIPC_EMPTY;
      kdipc_send_empty(tmp_sock, emptym);
    }
  }
}
```

Figure D.10: Pseudo-code for server thread (handle_kdipc_req_shm)

```
void send_page_reply (int id, struct shmid_kernel *shp, int key,
int idx, struct socket *tmp_sock) {
   // make sure that page was local enough time to serve a few local requests
   // keep track of how long it has been local vs. how long it took to
   // retrieve the page in the first place. A certain ratio needs to be
   // met before the page can be sent
   /*    check if I can send the page right away or if I need to wait */
   sleep_jiffies = shp->shm_perm.info.start_jiffies + shp->shm_perm.info.duration_jiffies - j
   if (sleep_jiffies > 0) {
     // we need to wait for a certain period of time
     // setup the timer here and go to sleep
     set_current_state(TASK_INTERRUPTIBLE);
     schedule_timeout(sleep_jiffies);
   }
   // prepare reply message
   replym->key = key;
   replym->msgtype = KDIPC_REPLY_SHM;
   pte = __pte(shm_ids.entries[id].p->info.ppte[idx]);
   pagep = pte_page(pte);

   /* Release the page */
   shm_ids.entries[id].p->info.ppte[idx] = pte_val(pte);
   shm_ids.entries[id].p->info.pflags[idx] &= (~PSTATE_PRESENT);
   // zap the page table entries in all atached segments
   // first for shared segments
   for (vmap = shp->shm_file->f_dentry->d_inode->i_mapping->i_mmap_shared;vmap; vmap = vmap->
      unsigned long addr = vmap->vm_start + idx * PAGE_SIZE;
      if ((vmap==NULL) || (addr > vmap->vm_end)) continue;
      pgd_t *pgd = pgd_offset(vmap->vm_mm, addr);
      pmd_t *pmd = pmd_offset(pgd, addr);
      pte_t *ppte = pte_offset(pmd, addr);
      pte_clear(ppte);
   }
   // next for non-shared segments
   for (vmap = shp->shm_file->f_dentry->d_inode->i_mapping->i_mmap;vmap; vmap = vmap->vm_next
      unsigned long addr = vmap->vm_start + idx * PAGE_SIZE;
      if ((vmap==NULL) || (addr > vmap->vm_end)) continue;
      pgd_t *pgd = pgd_offset(vmap->vm_mm, addr);
      pmd_t *pmd = pmd_offset(pgd, addr);
      pte_t *ppte = pte_offset(pmd, addr);
      pte_clear(ppte);
   }
   flush_tlb_all();
   __free_page(pagep);
}
```

Figure D.11: Pseudo-code for server thread (send_page_reply)

Bibliography

[1] Message passing interface standard. http://www-unix.mcs.anl.gov/mpi/.

[2] Posix standard. http://www.opengroup.org/onlinepubs/007904975/toc.htm.

[3] TreadMarks API Documentation. http://www.cs.rice.edu/CS/Systems/papers/doc.ps.gz.

[4] User-mode linux kernel. http://user-mode-linux.sourceforge.net/.

[5] Vmware virtual machine technology. http://www.vmware.com/.

[6] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[7] J.K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory using multi-protocol release consistency. In *Dagstuhl Seminar on Operating Systems of the 90s and Beyond*, volume 563 of *Lecture Notes in Computer Science*, pages 56–60. Springer Verlag, 1991.

[8] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Implementation and performance of munin. In *Proceedings of the 13th Symposium on Operating System Priniciples*, pages 152–164. ACM Press, October 1991.

[9] Cristian Ţăpuş, Justin D. Smith, and Jason Hickey. Kernel level speculative dsm. In *IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2003), Tokyo, Japan.*

[10] B D Fleisch. Distributed system v ipc in locus: a design and implementation retrospective. In *Proceedings of the ACM SIGCOMM conference on Communications architecture and protocols*, pages 386–396. ACM Press, 1986.

[11] Brett D. Fleisch, Randall L. Hyde, and Niels Christian Juul. Mirage+: A kernel implementation of distributed shared memory on a network of personal computers. *Software - Practice and Experience*, 24(10):887–909, 1994.

[12] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[13] Kamran Karimi and Tim Bynum. dipc sources. http://wallybox.cei.net/dipc/.

[14] Kamran Karimi and Mohsen Sharifi. Dipc: The linux way of distributed programming. In *The 4th International Linux Conference, Würzburg, Germany*, 1997.

[15] Peter Keleher. *Lazy Release Consistency for Distributed Shared Memory.* PhD thesis, Rice University, 1995.

[16] Peter Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter 94 Usenix Conference*, pages 115–131, January 1994.

[17] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, 1986.

[18] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, november 1989.

[19] Larry McVoy and Carl Staelin. lmbench sources. http://www.bitmover.com/lmbench/.

[20] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. *Usenix*, 1996.

[21] A. Mohindra and U. Ramachandran. A comparative study of distributed shared memory system design issues. Technical Report GIT-CC-94/35, 1994.

[22] Mahendra Ramachandran and Mukesh Singhal. Distributed semaphores.

[23] U. Ramachandran, M. Yousef, and A. Khalidi. An implementation of distributed shared memory. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 21–38, Berkeley, CA, 1989. USENIX Association.

[24] W. Richard Stevens. *UNIX network programming, volume 2 (2nd ed.): interprocess communications.* Prentice Hall PTR, 1999.