Interprocess Communication in Java

George C. Wells
Department of Computer Science, Rhodes University
Grahamstown, 6140, South Africa
G.Wells@ru.ac.za

Abstract— *This paper describes a library of classes providing support for interprocess communication in Java programs, using the mechanisms present in the native operating system. This approach is particularly well-suited for use with independent Java processes running on a single multicore (or multiprocessor) computer. At this stage, a comprehensive class library has been implemented for the Linux operating system, allowing access to the rich set of interprocess communication mechanisms provided by it. Initial testing shows significant performance improvements over the standard Java mechanisms available for such systems.*

Keywords: Java, interprocess communication, JNI, Linux

1. Introduction

For several years we have been investigating aspects of parallel and distributed computing in Java[1], [2], [3]. Recently this led to the observation that there seems to be a gap in the communication mechanisms provided by the Java language and the associated class libraries. At the level of *threads* there is very good support for communication and synchronisation through the use of the built-in monitor mechanism, and, more recently, through the Concurrency Utilities package. At the level of *distributed programming* there is strong support for network communication, both directly by using sockets, and through higher level abstractions, such as Remote Method Invocation (RMI). Our recent work has focused on multicore processors, and has led to the development of systems constructed of multiple *processes*, rather than threads. The only interprocess communication (IPC) mechanisms provided by Java in this scenario are the distributed programming mechanisms, using the "loopback" network connection.

While the use of the loopback network is adequate, and allows the easy use of any of the conventional Java distributed programming mechanisms, we were concerned with the efficiency of this approach. In particular, we questioned whether it is appropriate or efficient to force such localised communication to go through all the layers of a typical network protocol stack.

Of course, communication in this scenario (multiple processes executing on a single computer) *is* addressed by a wide range of IPC mechanisms present in almost all modern operating systems. This insight led us to develop an experimental library allowing systems composed of separate Java processes (executing in independent virtual machines) to make use of the IPC facilities of the underlying operating system. For the current prototype implementation, this support has been developed for the Linux operating system.

The next section of this paper expands on the problem scenario and discusses the mechanisms available in Java and in Linux for interprocess communication. This is followed by a description of our new library of IPC mechanisms and the presentation of the results of initial performance testing. The final sections of the paper discuss the limitations of our approach, present some conclusions and suggest areas for further investigation.

2. The Communication Problem

Our previous work on tuple space systems for Java (the eLinda project[4]) has either used a centralised approach, in common with most of the commercial systems and other research projects in this field, or else has used a fully-distributed approach with the tuple space distributed over all processing nodes. During the course of development of a new version of the eLinda system we have been exploring intermediate levels of data distribution, and have started to consider its application to multicore (or, more generally, multiprocessor) parallel architectures, rather than distributed "network of workstations" (NOW) systems. This investigation is partly in response to the growing trend towards multicore processors and the consequent importance of providing simple, but efficient, parallel programming environments for these systems[5].

In order to separate the logic for a parallel application from the new tuple space service it was desirable to run these as separate processes, each using a separate Java Virtual Machine (JVM)[6]. While developing the system along these lines, it became apparent that Java had little support for communication (or synchronisation) between independent processes/JVMs. What is provided is support for communication through the loopback network (this is expanded on briefly in section 2.1 below). An initial prototype was developed using this mechanism but, subjectively, the performance was felt to be poor. This led to the investigation of the use of the IPC mechanisms provided by Linux. These mechanisms are discussed in more detail in section 2.2.

2.1 IPC in Java

As stated already, Java has excellent support for *multi-threaded* applications consisting of several threads of execution running in the context of a single JVM. This support has been present from the inception of Java. Most current JVM implementations make use of the thread support provided by the underlying operating system and/or processor architecture to provide good levels of efficiency for such applications. In turn, this strong support for threads is carried through to the support for communication and synchronization provided by the language. This takes the form of a "monitor" mechanism, implemented through the provision of "synchronized" methods or code blocks, together with the `wait` and `notify` methods to control access to the monitors. In Java 5.0, a high-level concurrency library (the Concurrency Utilities package, `java.util.concurrent`) was introduced, based largely on the work of Doug Lea[7]. This provides high-level synchronisation mechanisms, improved support for creating and managing threads and a number of concurrent data structures.

At the other end of the concurrent programming spectrum, Java provides an extensive set of distributed programming mechanisms. These range from simple, direct network communication using TCP or UDP sockets, through to sophisticated distributed object-oriented mechanisms such as Remote Method Invocation (RMI) and the Common Object Request Broker Architecture (CORBA). As stated previously, these mechanisms are all available for use by multiple processes executing on a single computer system, through the Internet Protocol's loopback network (i.e. using network addresses in the range 127.x.x.x[1]).

2.2 IPC in Linux

When we encountered this problem, we recalled work we did some time ago on real-time programming in Unix[8]. This included a number of sophisticated mechanisms, relatively new at the time having been introduced in Unix System V, including message queues, access to shared memory blocks and semaphores. Prior to this there was only support for the simple, but effective "pipe" mechanism familiar to most users of the Unix command line.

2.2.1 Pipes

Pipes come in two slightly different forms. The first form is the standard pipe. These are created by a program prior to the creation of a child process (or processes), which is then connected to the pipe (most commonly through the standard input and output streams). The other form is the *named pipe* (also called a "FIFO", for the First-In First-Out nature of the pipe), which is created using the `mkfifo` system call and appears as a file within the file system (however, it should be noted that communication using the pipe does not involve any disk access).

[1] 127.0.0.1 is almost invariably used in practice.

2.2.2 System V IPC

The System V IPC facilities (message queues, shared memory and semaphore sets) have a rather complex API[8]. The first step in using any of these facilities is to generate a numeric "key". This should be unique to the application, but common to all processes wishing to communicate through the IPC facility being used. To aid in the generation of a key, a system call (`ftok`) is provided which generates a key from a file name and a single-character "project identifier". However the key is generated, it is then used to "get" one of the IPC facilities. The get operations create a new IPC facility (if necessary, and assuming there is no conflict with existing facilities and that access permissions are correct), and return a unique "identifier" that is then used to access and manage the specific IPC mechanism.

a) Message Queues: As outlined above, the `msgget` system call returns an integer identifier that can then be used to send and receive messages. This is done using the `msgsnd` and `msgrcv` system calls. Messages consist of a four-byte "type" field and an unformatted, variable-length block of bytes. Parameters to the `msgrcv` call allow messages to be retrieved in the order that they were sent, or else to use the type field to selectively retrieve messages from the queue. In addition to these fundamental operations, there is a `msgctl` system call used to manage the message queue. This allows the current status of the queue to be retrieved, the access-control permissions to be updated or the queue to be removed from the system.

b) Semaphore Sets: The `semget` system call allows for the creation of a *set* consisting of one or more semaphores. Synchronisation operations specify a subset of the semaphores. If more than one semaphore is manipulated in a single operation, the operation is performed atomically, and the operation suspends until *all* the suboperations specified can be performed. Synchronisation operations are performed using the `semop` method, which takes an array of individual operations to support this behaviour. Again, there is a `semctl` method used to manage the semaphore set, including the ability to query or set the value of a semaphore or several semaphores, to query the number of processes waiting on various semaphore conditions, etc.

c) Shared Memory: Again, the `shmget` system call is used to create a shared memory block and to get an identifier that can be used to access it. This identifier is used with the `shmat` system call to "attach" the shared memory block to the address space of a process. The `shmat` system call returns an address/pointer for this purpose. Once a process

no longer requires access to the shared memory block it can be "detached" using the shmdt system call. Two processes can obviously communicate through reading and writing data in a shared memory segment, but will usually need to synchronise their access (for example, by using a semaphore set).

3. The Java Linux IPC Library

In order to provide access to the Linux IPC facilities for Java programs we have developed an API using the Java Native Interface (JNI)[9]. JNI is a specification and a set of tools that allow Java programs to interact with code written in other programming languages (primarily C or C++ — we used C). There is a considerable amount of complexity required in order to do this, especially in terms of mapping data types and structures between these very different programming systems. JNI provides a number of useful utilities to aid in this translation. The "native" language components are created as a "shared library" which can then be dynamically loaded by the JVM at runtime.

For the moment, our interprocess communication API is simply being called *LinuxIPC*. It provides access to named pipes and to the System V IPC facilities described in the previous section, and also some useful I/O stream-based classes using the System V IPC mechanisms. Each of these will be discussed in the following subsections.

3.1 Direct IPC Access

3.1.1 Named Pipes

These were very simple to implement, as all that is required is access to the mkfifo system call. This simply takes a "file name", locating the pipe within the file system, and a bit mask containing the access-control permissions to be used. Once the named pipe has been created it can be accessed using the usual Java I/O mechanisms used for files (however, as noted in section 2.2.1, no access is made to the underlying hard disk when using the named pipe). This allows all the usual stream-based communication mechanisms, including serialization, to be used for communication and coordination between processes using the pipe.

3.1.2 System V IPC

a) Message Queues: Access has been provided to the msgget, msgsnd and msgrcv system calls. For the most part, the Java methods provided follow the format of the Linux system calls directly. The one exception to this is in regard to the message format for the msgsnd call. When using C, this makes use of a struct with a four-byte field for the message type and a second, variable-length field for the message content. This is difficult to emulate given the data structures available in Java, so the type field is specified as a separate parameter to the LinuxIPC msgsnd method.

For similar reasons, the msgctl system call presented a number of difficulties. This multipurpose system call is supported in C by the use of a complex data structure (struct msqid_ds). An initial attempt was made to provide an equivalent Java class and to map the C structure to and from this class type, but this rapidly became very cumbersome. As a simpler, albeit less complete, solution, a method (called msgRmid) was provided to allow access to the msgctl system call for the purpose of removing a message queue from the system. This method simply calls msgctl with the necessary parameters (the message queue identifier and the IPC_RMID operation code). Similar issues arose with the "control" system calls for the other IPC mechanisms.

b) Semaphore Sets: Support has been provided to use the semget and semop systems calls. The latter posed some interesting problems as it expects a variable-length array of sembuf structures as one of its parameters. The sembuf structure is relatively simple, consisting of three integer fields (the semaphore number, the operation to be performed, and bit flags), so this was solved by providing an equivalent Java class and copying the data fields between the Java and C data structures. Two versions of the method are provided: one that explicitly specifies the number of operations to be performed and one that makes use of the self-describing nature of Java arrays, assuming all elements of the array are to be used.

The Linux operating system also includes a variant of the semop system call, called semtimedop. This provides the ability to have semaphore operations "time out" if necessary. Support for this has been provided in LinuxIPC. The semctl system call presented the same problems as the msgctl system call. This was solved in the same way, by providing a number of more specialised Java methods that allow the various functions provided by semctl to be accessed directly. These include semRmid to remove a semaphore set, semGetVal to query the current value of a semaphore in a set, semSetVal to set the value of a semaphore and semGetNCnt to retrieve the number of processes waiting on the semaphore.

c) Shared Memory: The LinuxIPC library provides methods for the shmget, shmat and shmdt system calls. However, another interesting problem arises here in that the shared memory mechanism works with memory addresses as pointers. This does not map easily to the Java memory model. The solution that has been provided is to return the pointer returned by the shmat system call as a Java int value. This allows the Java application to generate "pointers" to arbitrary addresses within the shared memory block. To support access to the shared memory itself, two additional methods have been provided (shmWrite and shmRead)

that copy data (specified as arrays of bytes) between the Java memory space and the shared memory block.

Detaching a shared memory block from the address space of a process does not remove it from the system. As usual this is done using the "control" system call, which has the same complexities as for message queues and semaphore sets. The same solution has been adopted, providing a `shmRmid` method in the LinuxIPC library.

3.1.3 Other Utilities and Error Handling

Several other related Unix system calls have also been provided as part of the LinuxIPC library. As mentioned in section 2.2.2, the `ftok` system call is useful when generating keys for use with the System V IPC facilities. Accordingly, it has been included in the LinuxIPC library.

Many of the system calls included in the LinuxIPC library return the value -1 to indicate that an error has occurred. In this case, a system-level variable called `errno` is set by the operating system to indicate the cause of the error. Where this may occur, the LinuxIPC native functions retrieve the error number and store it in a private field of the `LinuxIPC` class. Java programs may then retrieve this using a method of the class (`getErrnum`). Furthermore, access is also provided to the standard C `strerror` function which maps the error numbers used by the operating system to text strings to aid in the interpretation of errors and the display of suitable error messages.

3.2 Specialised I/O Streams

Given Java's current dependence on network mechanisms for all parallel and distributed communication and coordination purposes (except between threads) it was perhaps obvious to consider how specialised I/O streams might be provided based on the Linux IPC facilities described in the previous section. This has been done using both message queues and shared memory.

a) Message Queue Streams: Two additional Java classes have been provided as part of the LinuxIPC library, `MessageQueueInputStream` and `MessageQueue-OutputStream`, which are subclasses of the standard Java `InputStream` and `OutputStream` classes respectively. The constructors for the new classes take a Linux IPC key as a parameter, but otherwise these classes abstract away the details of working with the message queue. Calls to the `read` and `write` methods of the new classes result in the data being communicated as messages on the message queue. From the perspective of the application using them, the new classes may be used like any other stream class in Java. In particular they may be composed with other stream-handling classes such as `BufferedOutputStream` for buffering, or `ObjectInputStream` for object serialization.

b) Shared Memory Streams: As with the message queue streams described above, two classes `Shared-MemoryInputStream` and `SharedMemoryOutput-Stream` have been provided in the LinuxIPC library. These abstract away the details of using shared memory, with semaphores for synchronisation, from the application using them (except for the need to specify the IPC key to be used, and optionally to specify the size of the block of shared memory to be used). The initial implementation of these classes simply used the individual methods provided by the LinuxIPC library to create and access the shared memory block and the semaphores needed to synchronise access to this memory. However, the performance of this solution was found to be very poor. Investigation of the reasons for this revealed that the JNI system introduces a considerable overhead into the calling of native C functions (due to the need to map between different calling conventions and data formats). These overheads were effectively being multiplied by the use of separate calls to the semaphore and shared memory access methods. In order to optimise these classes, by minimising the JNI calls, a new JNI class was developed to provide higher-level support for the shared memory streams. Essentially, this simply migrated some of the logic required for buffering and data transfer from the Java classes to C code in a shared library. This approach required fewer cross-language calls and improved the performance of the shared memory stream classes.

4. Testing and Results

The computer used for testing was a Dell Inspiron 1525 laptop with a 2.0GHz Intel Core 2 Duo twin-core processor, 2MB of L2 cache and 4GB of memory. The version of Java used was 1.6.0_07, and the operating system was Ubuntu version 8.04.1 (Linux kernel 2.6.24-22).

4.1 Performance Testing

The LinuxIPC library has been tested using a simple bidirectional communication benchmark. This utilises two processes. The first gets the current system time in nanoseconds (using the `System.nanoTime()` method) then sends this as a serialized `Long` object to the second process. The second process immediately returns the time object back to the first process, which then uses the elapsed time to measure the round-trip time for the communication. This was done using conventional socket communication with the loopback network as a reference, then with named pipes, message queues and shared memory. Each test was repeated 20 times, and the average of 19 runs is reported in section 4.2 (excluding the first run to discount the start-up costs of class-loading, etc.). Further testing was also performed using variable-size messages in order to characterise the communication methods with differing levels of data transmission.

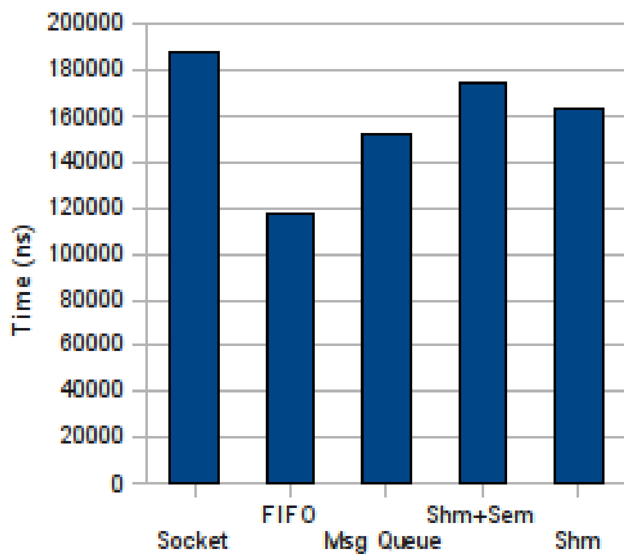

Fig. 1: Simple Results (minimal data)

4.2 Results

Figure 1 shows the results for the first communication test (i.e. with minimal data transfer). As can be seen quite clearly, all the LinuxIPC mechanisms perform better than the use of the loopback network (labelled "Socket" in the graph). The best result is obtained using named pipes (about 63% of the time taken by the loopback socket version), which is not surprising as the use of JNI is minimised in this case (all that is required is the `mkfifo` system call to create the pipe). Message queues are reasonably efficient (81% of the socket time). As expected, the implementation of shared memory streams using semaphores explicitly (labelled "Shm+Sem") does not perform very well, but is still more efficient than the socket version. When using the specialised shared memory support library, performance improved by about 6%.

The second communication test showed a similar pattern, with the transmission time increasing as the amount of data contained in the messages increased, as shown in Table 1 and Figure 2. What is very noticeable about these results is that the loopback socket increases in efficiency as the message size increases, until a certain threshold when it dramatically decreases in performance (however, at no point is it more efficient than using named pipes). This behaviour requires further investigation to better characterise the performance and to isolate the inflection point in the efficiency of the loopback socket mechanism.

5. Discussion

The results indicate the clear performance benefits to be gained by utilising native IPC mechanisms for communication between Java processes. While there are some situations in which sockets outperform shared memory and even message queues, the use of named pipes is substantially more efficient under all conditions tested. This is a useful result and provides an additional advantage, in that this mechanism can be used on a Linux system *without* the need for the JNI library, by using the `mkfifo` command in a shell script or directly from the command line. The Java program may then open a connection to the pipe using the standard I/O library (i.e. the IPC mechanism is *completely* transparent to the application).

Of course, a major drawback in using the Linux IPC mechanisms is the loss of the "Write Once, Run Anywhere" (WORA) cross-platform portability property of Java programs. While this is a significant problem, it is mitigated by the very real performance benefits to be obtained by using the IPC mechanisms provided by the host operating system. In this regard, we would again stress the likely importance of efficient IPC for applications composed of multiple processes running on a multicore architecture as these systems become increasingly common (and as the number of cores increases, requiring higher levels of parallelism in applications). One possible solution to this problem would be to provide an IPC library at a higher level abstraction, providing mechanisms that are essentially common to all widely-used operating systems. This would require distinct native shared libraries to be developed for all the desired operating systems, and may result in some loss of performance due to the increased level of abstraction required. However, it would be worth investigating further. An initial step would be to survey the IPC facilities provided by the major operating systems targeted by Java applications in order to establish the common features and to design a suitable set of abstractions.

6. Conclusion

The investigation described in this paper has been a very interesting and potentially beneficial digression from our main research programme in the area of tuple space systems for parallel and distributed programming. The relative lack of support for communication and synchronisation operations in Java applications composed of multiple processes executing on a multicore (or multiprocessor) system is likely to become increasingly important as the trend to increasing numbers of processor cores continues. The LinuxIPC library provides an initial basis for improving the support for this scenario in Java applications, and our experimental results demonstrate some of the potential performance benefits to be gained from providing improved IPC mechanisms.

6.1 Future Work

As already mentioned in section 5, an important next step would be to address the issue of cross-platform portability by surveying the IPC features of widely-used operating systems. This would form the basis for the design of a higher-level,

Table 1: Detailed Results (with varying data sizes)

Data Size (bytes)	**40**		**400**		**4 000**		**40 000**	
	Time (ns.)	**%**	**Time (ns.)**	**%**	**Time (ns.)**	**%**	**Time (ns.)**	**%**
Socket	281296.4	100	273087.9	100	297739.4	100	5457363.1	100
FIFO	169501.4	60	177989.3	65	221903.3	75	567776.4	10
Msg Queue	164101.2	58	268437.8	98	671227.0	225	877882.1	16
Shm+Sem	193932.4	69	302841.2	111	694057.1	233	1455767.2	27
Shm	166692.9	59	273473.4	100	700971.7	235	1270286.3	23

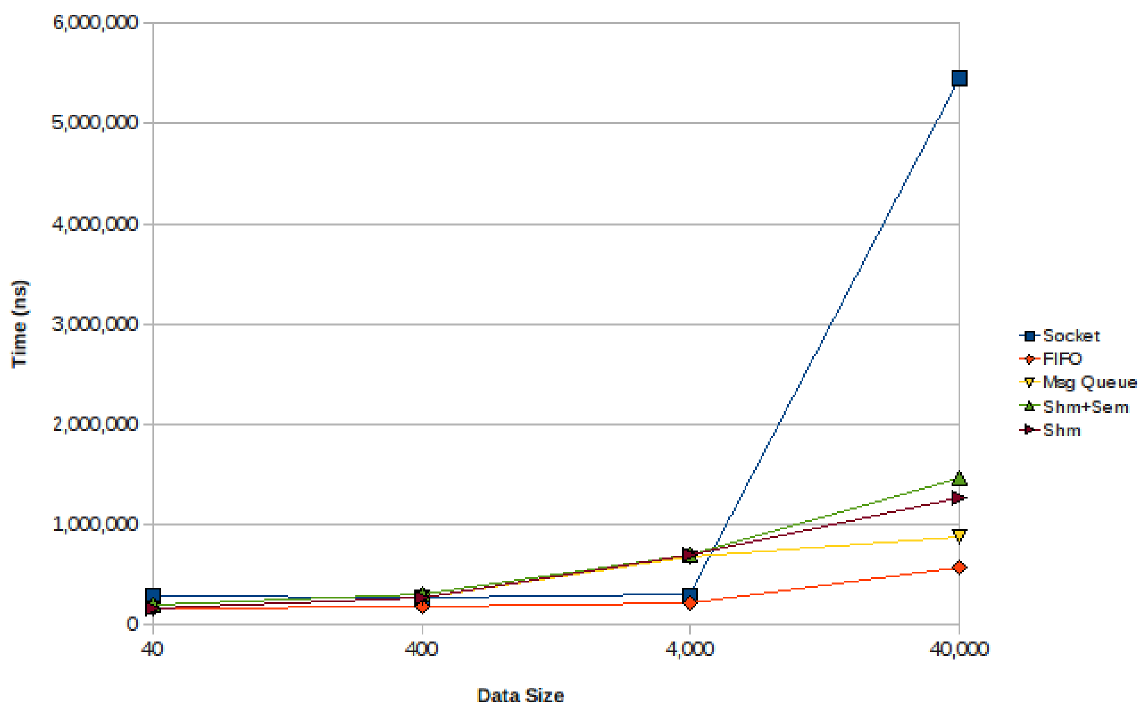

Fig. 2: Detailed Results (with varying data sizes)

more abstract IPC library that could then be implemented across these operating systems.

The performance testing performed to date has utilised simple, artificial communication benchmark programs, as described in section 4.1. It would be useful to extend the performance testing by using the LinuxIPC library in a larger application. This would add useful information with regard to the potential performance benefits for real applications. We plan to use the LinuxIPC library in our current work on tuple space systems, and use the results of this to assess the performance in larger-scale applications.

Acknowledgments: This research was performed while visiting the Department of Computer Science at the University of California, Davis at the kind invitation of Dr. Raju Pandey. Financial support was received from the Distributed Multimedia Centre of Excellence (funded by Telkom, Comverse, StorTech, Tellabs, OpenVoice, Amatole Telecommunication Services, Mars Technologies, Bright Ideas Projects 39 and THRIP), from Rhodes University and the South African National Research Foundation (NRF).

References

[1] G. Wells, A. Chalmers, and P. Clayton, “Linda implementations in Java for concurrent systems,” *Concurrency and Computation: Practice and Experience*, vol. 16, pp. 1005–1022, Aug. 2004.

[2] G. Wells, “A tuple space web service for distributed programming,” in *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2006)*. CSREA Press, June 2006, pp. 444–450.

[3] G. Wells and G. Atkinson, “Grid computing in an academic environment,” in *Proc. 2007 International Conference on Grid Computing and Applications (GCA'07)*, Las Vegas, June 2007, pp. 49–55.

[4] G. Wells, “New and improved: Linda in Java,” *Science of Computer Programming*, vol. 59, no. 1–2, pp. 82–96, Jan. 2006.

[5] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick, “The landscape of parallel computing research: A view from Berkeley,” EECS Department, University of California, Berkeley,

Tech. Rep. UCB/EECS-2006-183, Dec. 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf

[6] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd ed. Prentice Hall PTR, 1999.

[7] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, 2nd ed. Prentice Hall, 1999.

[8] G. Wells, "An evaluation of XENIX System V as a real-time operating system," *Microprocessing and Microprogramming*, vol. 33, no. 1, pp. 57–66, 1991.

[9] S. Liang, *Java Native Interface: Programmer's Guide and Specification*. Prentice Hall, June 1999.