Message Passing, Remote Procedure Calls and Distributed Shared Memory as Communication Paradigms for Distributed...

Andrzej Goscinski

Message Passing, Remote Procedure Calls and Distributed Shared Memory as Communication Paradigms for Distributed Systems

J. Silcock and A. Goscinski

{jackie, ang@deakin.edu.au}

School of Computing and Mathematics
Deakin University
Geelong, Australia.

Abstract

Message passing and remote procedure calls are the most commonly used communication paradigms for interprocess communication in distributed systems. Distributed shared memory is an equally valuable but less often used paradigm. The advantage offered by distributed shared memory is that it abstracts away from the fact that the memory is distributed and allows the programmer to use the familiar shared memory model. However, this ease of use comes at a price, the overheads of DSM, at the operating system level, are significant compared with those of the other two paradigms. In order to establish whether DSM is worth implementing these overheads need to weighed up against the advantages offered by the paradigm. To this end we have evaluated and compared the three paradigms. In particular they are evaluated at the programming and operating system level and then compared based on their performance when used to solve the producer-consumer problem.

Table of Contents

1 Introduction

Processes often need to communicate with each other. This is complicated in distributed systems by the fact that the communicating processes may be on different workstations. Interprocess communication provides a means for processes to cooperate and compete. This includes synchronized access to shared variables. Message passing and remote procedure calls are the most common methods of interprocess communication in distributed systems. A less frequently used but no less valuable method is distributed shared memory. Message passing involves the passing of messages between processes using simple primitives to send and receive messages. It requires the programmer to know the message and the names of the source and destination processes. Remote procedure calls represent higher abstraction than message passing; remote procedures are called in a similar way to local procedures with the operating system taking care of the details of locating the remote procedure and preparing the arguments for inclusion in the message which calls the remote procedure. Distributed shared memory shares variables between computers on a network transparently. The user is unaware that the underlying mechanism for communication is message passing and is able to use the well known shared memory paradigm. However it is important to evaluate distributed shared memory and compare it with the other two communication paradigms in order to assess its usefulness. It is also important to assess whether the ease of programming that comes with distributed shared memory outweighs the complexity of its addition to the operating system.

The goal of this report is to evaluate and compare message passing, remote procedure calls and distributed shared memory as communication paradigms. The three paradigms will be evaluated first at operating system level using the following criteria:

— their basic concepts, syntax and semantics;

— issues at the logical level; and

— implementation issues.

Secondly they will be evaluated at the programming level based on the implications for the programmer of using these paradigms, i.e. ease of use, particularly with regard to synchronization.

2 Message Passing

Message passing is the basis of most interprocess communication in distributed systems. It is at the lowest level of abstraction and requires the application programmer to be able to identify the destination process, the message, the source process and the data types expected from these processes.

2.1 Syntax

Communication in the message passing paradigm, in its simplest form, is performed using the **send()** and **receive()** primitives. The syntax is generally of the form:

send(*receiver, message*)

receive(*sender, message*)

The **send()** primitive requires the name of the destination process and the message data as parameters. The addition of the name of the sender as a parameter for the **send()** primitive would enable the receiver to acknowledge the message. The **receive()** primitive requires the

name of the anticipated sender and should provide a storage buffer for the message.

2.2 Semantics

Decisions have to be made, at the operating system level, regarding the semantics of the **send()** and **receive()** primitives. The most fundamental of these are the choices between blocking and non-blocking primitives and reliable and unreliable primitives [Goscinski 91], [Tanenbaum 85].

Blocking/non-blocking — A blocking **send()** blocks the process and does not execute the following instruction until the message has been sent and the message buffer has been cleared. In the same way a blocking receive blocks at the **receive()** until the message arrives. A non-blocking send returns control to the caller immediately. The message transmission is then executed concurrently with the sending process. This has the advantage of not leaving the CPU idle while the send is being completed. However, the disadvantage of this approach is that the sender does not know and will not be informed when the message buffer has been cleared. To overcome this the kernel can either make a copy of the message buffer or send an interrupt to the sender when the message buffer has been cleared. At the implementation level, although non-blocking primitives are flexible they make programming and debugging very difficult, hence, for the sake of easier programming, blocking primitives are often chosen.

Buffered/unbuffered messages — An unbuffered **receive()** means that the sending process sends the message directly to the receiving process rather than a message buffer. The address, *receiver,* in the **send()** is the address of the process, but in the case of an buffered **send()** the address is that of the buffer. There is a problem, in the unbuffered case, if the **send()** is called before the **receive()** because the address in the send does not refer to any existing process on the server machine. Buffered messages are saved in a buffer until the server process is ready to receive them. They can best be implemented through a mechanism in the operating system which can keep a backlog of **sends**, a port in which messages are queued waiting until requested by the receiver. The buffer capacity can either be bounded, where a predetermined number of messages can be stored, or unbounded. An unbounded buffer could be implemented using dynamic memory allocation, thus its capacity would be fixed only by the size of available memory.

Reliable/unreliable send — Unreliable **send()** sends a message to the receiver and does not expect acknowledgement of receipt, nor does it automatically retransmit the message to ensure receipt. A reliable **send()** guarantees that, by the time the **send()** is complete, the message has been received. The primitive itself handles acknowledgements and retransmission in response to lost messages. At the implementation level the operating system must wait only for a specified length of time, so that a process does not remain blocked indefinitely waiting for a response from a receiver that has terminated. Lost messages are handled either by the operating system retransmitting the message or informing the sender of the message's loss or by the sender detecting the loss itself [Silberschatz 85].

Direct/indirect communication — Ports allow indirect communication. Messages are sent to the port by the sender and received from the port by the receiver. Direct communication involves the message being sent direct to the process itself, which is named explicitly in the send, rather than to the intermediate port.

Fixed/variable size messages — Fixed size messages have their size restricted by the system. The implementation of variable size messages is more difficult but makes programming easier; the reverse is true for fixed size messages.

Passing data by reference/value/address mapping — Data, in message passing, is often

passed by value, since the processes execute in separate address spaces [Goscinski 91]. However, another parameter passing mechanism is available which would be suitable for a message passing system. This is referred to as call-by-copy/restore. The variable is essentially passed by value but the returned value overwrites the original value so that the final result is the same as if it were passed by reference [Tanenbaum 95].

In message passing systems the onus is on the application programmer to control data movement between processes and to control the synchronization of these processes, where they have access to shared data.

3 Remote Procedure Calls

Message passing leaves the programmer with the burden of the explicit control of the movement of data. Remote procedure calls (RPC) relieves this burden by increasing the level of abstraction and providing semantics similar to a local procedure call.

3.1 Syntax

The syntax of a remote procedure call is generally of the form:

call procedure_name(*value_arguments; result_arguments*)

The client process blocks at the **call()** until the reply is received. The remote procedure is the server processes which has already begun executing on a remote machine. It blocks at the **receive()** until it receives a message and parameters from the sender. The server then sends a **reply**() when it has finished its task. The syntax is as follows:

receive procedure_name(*in value_parameters; out result_parameters*)

reply(*caller, result_parameters*)

3.2 Semantics

The semantics of RPC are the same as those of a local procedure call — the calling process calls and passes arguments to the procedure and it blocks while the procedure executes. When the procedure completes it can return results to the calling process. In the simplest case, the execution of the **call()** generates a client stub which marshals the arguments into a message and sends the message to the server machine. On the server machine the server is blocked awaiting the message. On receipt of the message the server stub is generated and extracts the parameters from the message and passes the parameters and control to the procedure. The results are returned to the client with the same procedure in reverse [Mullender 89].

The following issues regarding the properties of remote procedure calls need to be considered in the design of an RPC system if the distributed system is to achieve transparency [Birrell et al. 84], [Mullender 89], [Goscinski 91]:

Binding — Binding provides a connection between the name used by the calling process and the location of the remote procedure. Binding can be implemented, at the operating system level, using a static or dynamic linker extension which binds the procedure name with its location on another machine. Another method is to use procedure variables which contain a value which is linked to the procedure location.

Communication transparency — The users should be unaware that the procedure they are calling is remote. The three difficulties when attempting to achieve transparency are: the

detection and correction of errors due to communication and site failures, the passing of parameters, and exception handling. Communication and site failures can result in inconsistent data because of partially completed processes. The solution to this problem is often left to the application programmer. Parameter passing in most systems is restricted to the use of value parameters. Exception handling is a problem also associated with heterogeneity. The exceptions available in different languages vary and have to be limited to the lowest common denominator.

Concurrency — Concurrency mechanisms should not interfere with communication mechanisms. Single threaded clients and servers, when blocked while waiting for the results from a RPC, can cause significant delays. These delays can be exacerbated by further remote procedure calls made in the server. Lightweight processes allow the server to execute calls from more than one client concurrently [Mullender 89].

Heterogeneity — Different machines may have different data representations, the machines may be running different operating system or the remote procedure may have been written using a different language. Static interface declarations of remote procedures serve to establish agreement between the communicating processes on argument types, exception types (if included), type checking and automatic conversion from one data representation to another, where required.

Generally RPC proves a simpler means for an application programmer to construct distributed programs than simple message passing because it abstracts away from the details of communication and transmission. However, the achievement of true transparency is a problem which has not been completely resolved for RPC, still leaving much of the work and responsibility for the application programmer.

4 Distributed Shared Memory

The next step towards higher abstraction is Distributed Shared Memory (DSM). DSM increases the complexity of the operating system but makes the job of application programmers far easier by allowing them to use the concept of shared memory when writing programs. Distributed shared memory is memory which, although distributed over a network of autonomous computers, gives the appearance of being centralized. The memory is accessed through virtual addresses, thus processes are able to communicate by reading and modifying data which are directly addressable. DSM allows programmers to use shared memory style programming, which makes application programming considerably easier. Programmers are able to access complex data structures and are relieved of the concerns of message passing. However, message passing cannot be avoided altogether. The operating system has to send messages between machines with requests for memory not available locally and to make replicated memory consistent.

4.1 Syntax

The syntax used for DSM is the same as that of normal centralized memory multiprocessor systems.

read(*shared_variable*)

write(*data*, *shared_variable*)

The **read()** primitive requires the name of the variable to be read as its argument and the

write() primitive requires the data and the name of the variable to which the data is to be written. The operating system locates the variable through its virtual address and, if necessary, moves the portion of memory containing the variable to the machine requiring it.

4.2 Semantics

There are several issues related to the semantics of DSM [Coulouris et al. 89], [Nitzberg et al. 94], [Tanenbaum 95].

Structure and granularity of the shared memory — These two issues are closely related. The memory can take the form of an unstructured linear array of words or the structured forms of objects, language types or an associative memory [Nitzberg et al. 94]. The granularity relates to the size of the chunks of the shared data. A decision has to be made whether it should be fine or coarse grained and whether data should be shared at the bit, word, complex data structure or page level. A coarse grained solution, page-based distributed memory management, is an attempt to implement a virtual memory model where paging takes place over the network instead of to disk. It offers a model which is similar to the shared memory model and is familiar to programmers, with sequential consistency at the cost of performance. Finer grained models can lead to higher network traffic.

Consistency — In the simplest implementation of shared memory a request for a non-local piece of data results in a trap, which causes the single copy of the data to be fetched. If a piece of data was required by more than one machine the data could be moved backwards and forwards between the machines. This is very similar to thrashing in virtual memory and has the effect of considerably lowering performance. The problem of thrashing is overcome by allowing multiple copies of data on the distributed machines. The problem then becomes one of maintaining the consistency of the replicated data. The cache coherence protocols of tightly coupled multiprocessors are a well researched topic [Mosberger 94], however many of these protocols are thought to be unsuitable for distributed systems because the strict consistency models used cause too much network traffic [Nitzberg et al. 94]. Consistency models determine the conditions under which memory updates will be propagated through the system. These models can be divided into those with and those without synchronization operations. The former include strict, sequential, causal, processor and PRAM consistency models, while models with synchronization operations include weak, release and entry consistency models. There is a weakening of the consistency models from strict to entry consistency. Weaker models reduce the amount of network traffic hence the performance of the system improves. Thus, weaker consistency models have been used in an attempt to achieve better performance in distributed systems. However, this makes the programming model more complicated and makes weaker consistency the concern of operating systems and language designers [Mosberger 94].

Synchronization — Shared data must be protected by synchronization primitives, semaphores, eventcounts, monitors or locks. There are three methods of managing synchronization. Firstly, it can be managed by a synchronization manager, as in the case of page-based systems, or secondly, it can be made the responsibility of the application programmer, using explicit synchronization primitives, as in the shared variable implementation. Finally, it can be made the responsibility of the system developer, as in object based implementations, with synchronization being implicit at application level.

Heterogeneity — Sharing data between heterogeneous machines is an important problem for distributed shared memory designers. Data shared at the page level is not typed, hence

accommodating different data representations of different machines, languages or operating systems is a very difficult problem. The Mermaid approach mentioned in [Li et al. 88] is to only allow one type of data on an appropriately tagged page. The overhead of converting the data might be too high to make DSM on a heterogeneous system worth implementing.

Scalability — One of the benefits of DSM systems mentioned in much of the literature [Nitzberg et al.94], [Tanenbaum 95] is that they scale better than many tightly-coupled shared-memory multiprocessors. However, scalability is limited by physical bottlenecks, e.g., buses in tightly-coupled multiprocessor systems and operations which require global information or distribute information globally, e.g. broadcast messages [Nitzberg et al. 94].

5 Producer-Consumer Example

The best method of measuring the value of the DSM paradigm is to measure its performance against the other communication paradigms. This can be done by using all three paradigms to solve the same problem and measuring their relative performance using well defined performance criteria. We have chosen the producer-consumer problem because it is a well known problem which involves shared variables and requires synchronized access to these variables. The performance criteria we will use in our analysis are:

— ease of implementation for system designer;

— ease of use at application programming level; and

— relative performance.

The following sections describe the underlying actions taken by the operating system in response to the commands of the producer-consumer code for a centralized system, and a distributed system. The latter includes code for implementations using message passing, RPC and for DSM implemented at system and user level.

5.1 Producer-Consumer on a Centralised System

The producer and consumer physically exist in the same centralized memory and the address of the variable *next* is the same for both processes. Access to *next* is controlled by a semaphore (*sem*) variable. The only allowable operations on a semaphore variable are **wait**(*sem*), **signal**(*sem*) and **initialise**(*sem*) [Ben-Ari 82]. As depicted in Figure 1, the producer executes the primitive operation **wait**(*sem*), which is, in effect, requesting a lock on the semaphore *sem*. In this case, since there are only two processes trying to access *next* the semaphore, *sem,* is a binary semaphore which can only have two values zero or one, i.e., locked or unlocked. This lock is achieved at the operating system level. If the value of *sem* is one it is decremented to zero and the lock is granted, if it is zero the lock will be refused. In the latter case the requesting process is suspended. A **wait**(*sem*) should precede and a **signal**(*sem*) should follow all accesses to *next*, defining a critical region. When the lock is obtained the producer can safely change *next* as it will be the only process with access to the critical region. The producer produces an item and places it in the shared variable *next,* and then executes the **signal**(*sem*) call which causes the operating system to check the variable *process_waiting*, if true it indicates a process is waiting which is made executable, if false the semaphore is simply unlocked.

The consumer also requests a lock on *sem* by executing a **wait**(*sem*). When it gains access to the critical region it consumes the item in *next* and releases the lock by executing a **signal**(*sem*).

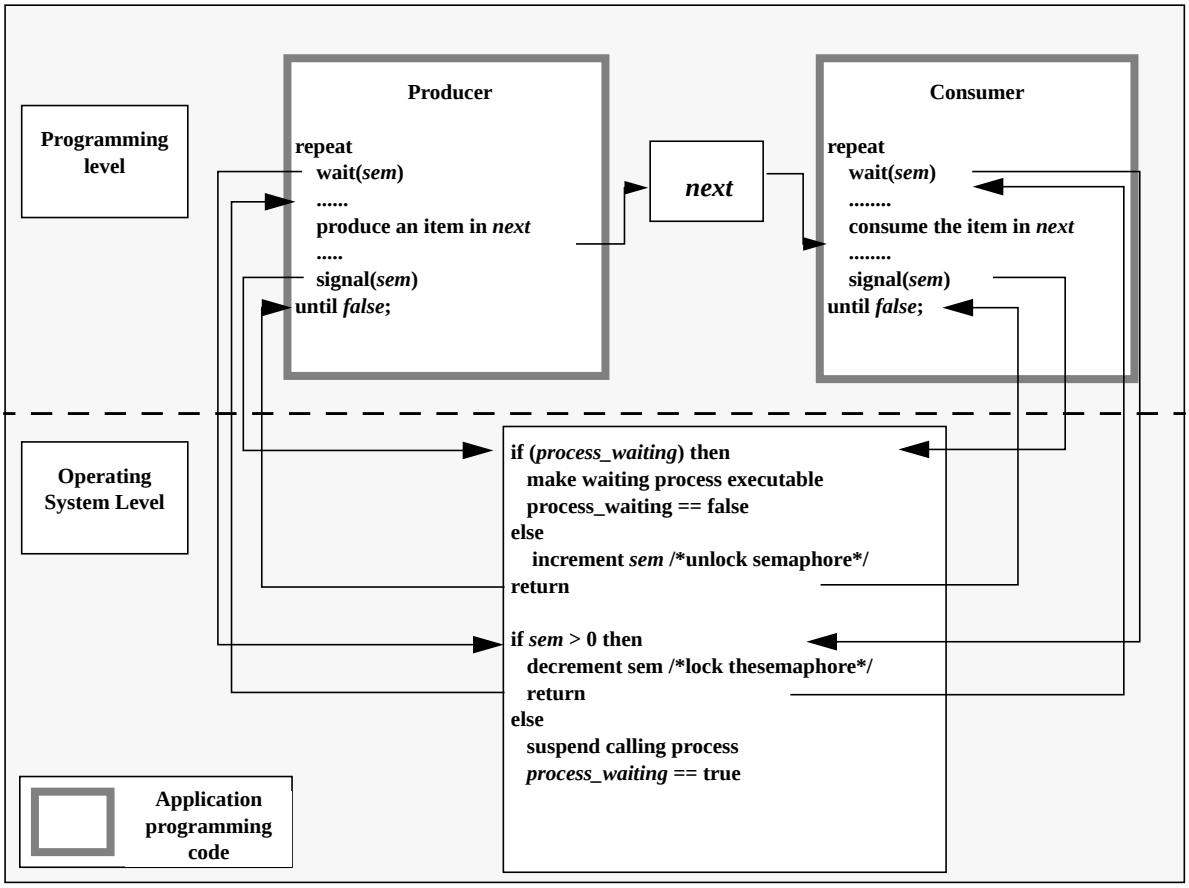

Figure 1. Producer-Consumer on a Centralized System

5.2 Producer-Consumer supported by Message Passing on a Distributed System

In a distributed system the producer and the consumer have disjoint address spaces. Thus to share a variable using message passing the value of that variable must be sent from one process to another explicitly. The producer produces an item and places it in the variable *next*. It then executes a **send**(*consumer, next*) which results in the operating system constructing a message and sending it to the consumer (Figure 2). It then blocks pending the clearing of the message buffer, in which the message is held until it is sent.

The consumer executes a **receive**(*producer, next*) which causes the consumer to be blocked, at the operating system level, awaiting a message from the producer. When the message is received the operating system removes the data from the message and places it in *next*.

5.3 Producer-Consumer supported by RPC on a Distributed System

In the RPC version of the producer-consumer (Figure 3) the producer produces an item and places it in *next*. It then issues a remote procedure call to the consumer. The operating system generates a client stub which marshals *next* for transmission. The call is then sent to the consumer. The producer is suspended while awaiting a reply from the consumer. When the return message is received from the consumer, the producer is made executable and control is passed back to the programmer level.

The consumer executes a **receive**(*producer, next*) which at operating system level blocks the consumer awaiting the procedure call. When the call arrives the operating system generates

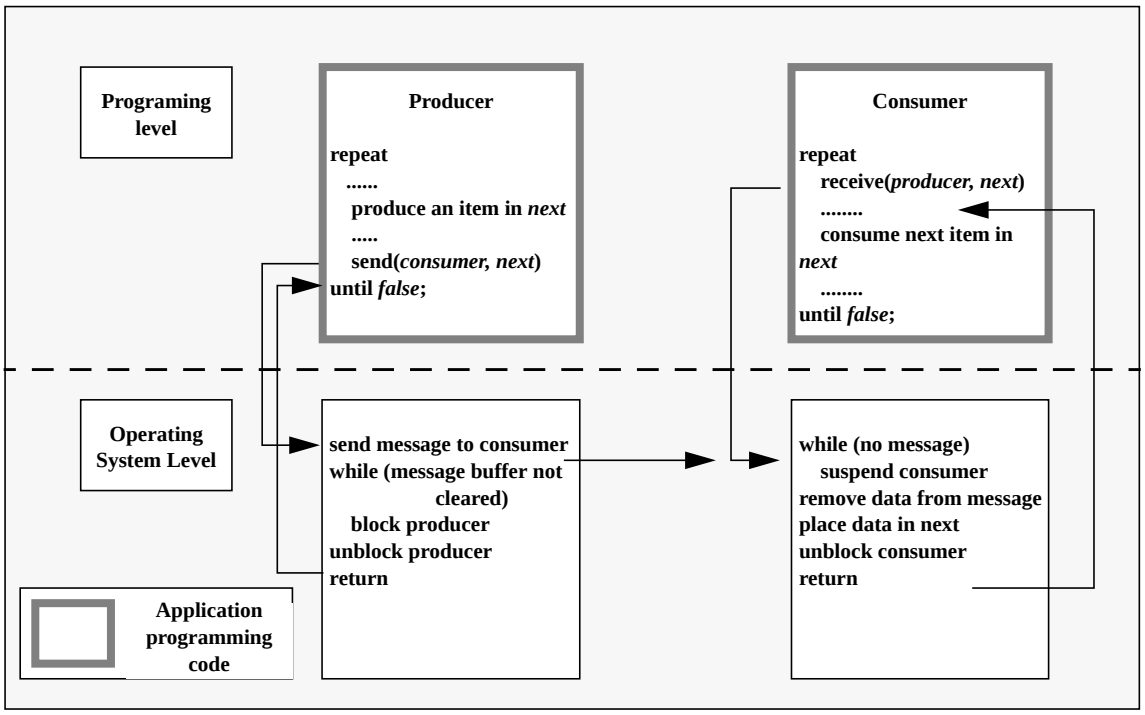

Figure 2. Producer-Consumer supported by Message Passing System.

the server stub which unmarshals *next* and passes it to the consumer. Control is then passed back to the programmer level where *next* is consumed and a **reply**(*producer*) is executed acknowledging receipt of the call.

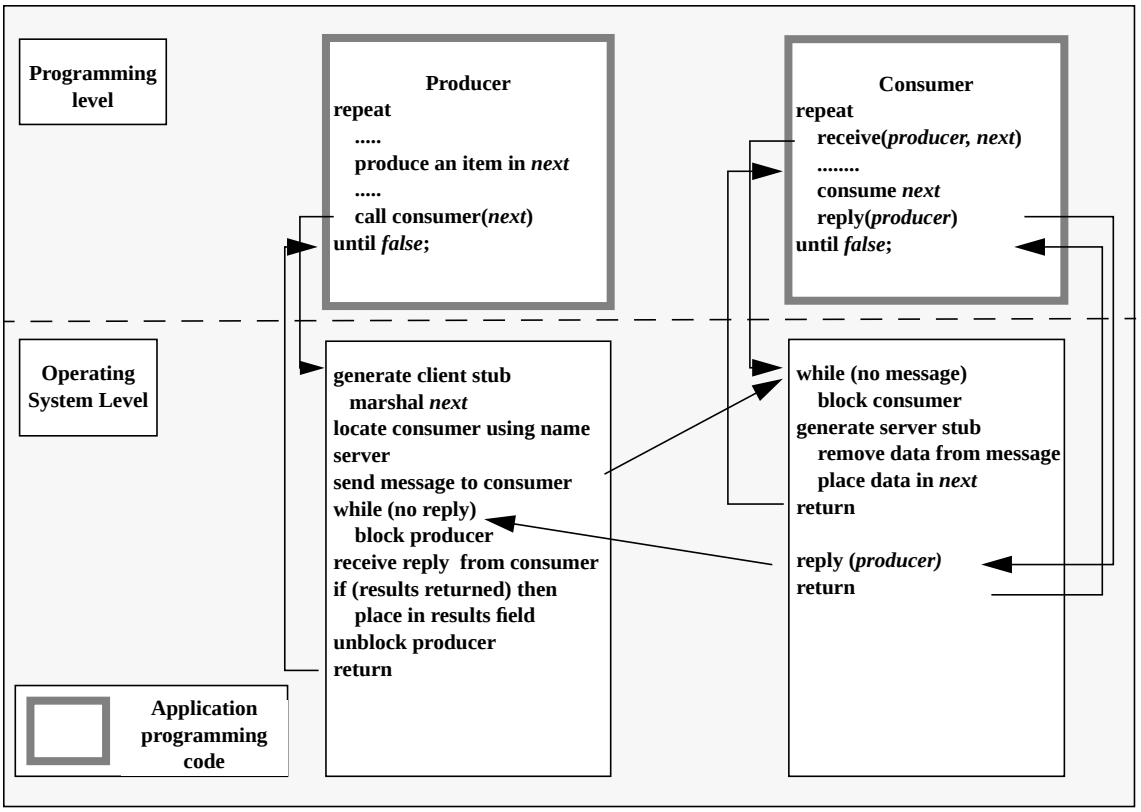

Figure 3. Producer-Consumer supported by Remote Procedure Call

5.4 Producer-Consumer supported by DSM at system level on a Distributed System

The syntax of the code using DSM is the same as that for centralized memory but the implications for the operating system are quite different (Figure 4).

The simplest method of implementing synchronization on a distributed system with distributed shared memory is to use a centralized synchronization manager. This is the method depicted in Figure 4. It is probably not the most efficient method since the use of any centralized system or manager such as this can cause a bottleneck. However, it gives an indication of the large number of messages required to implement synchronization in DSM.

The process requiring a lock on the semaphore executes a **wait**(*sem*). In response to this the system executes a remote procedure call to the synchronization manager which grants access if the semaphore variable, *sem*, is one or if not able to grant access sets the variable *process_waiting* and blocks the process. In Figure 4 the consistency model being used is entry consistency, a weak model in which replicated memory is made consistent on entry into a critical region. The messages required to make memory consistent are not depicted in this diagram for simplicity. If the synchronization manager is granting access to the critical region it decrements *sem* to zero and sends a message back to the requesting process granting access. In the calling process, when the access message is received, control is passed back to the programming level. The process then attempts to access the shared variable, *next*. If the memory containing *next* is not local to that machine a trap will occur which will be caught by the operating system. The operating system will use some mechanism to locate and fetch a copy of the required piece of memory on the network, this mechanism will not be discussed here. When the process has completed the critical section it will execute a **signal**(*sem*) which will cause the operating system to send an RPC to the synchronization manager releasing the lock on the semaphore. In the synchronization manager, if the *process_waiting* variable is set to true this indicates that a process is blocked waiting for access to the critical region. The synchronization manager will send a message granting access to this waiting process.

5.5 Producer-Consumer supported by DSM at user level on a Distributed System

The producer-consumer code with user level DSM depicted in Figure 5 is based on an implementation described in [Libes 85]. In this implementation the shared memory exists at user level in the memory space of a server, the central memory manager (cmm), the clients maintain a local copy of the shared variable, called *local_next*. Because this implementation has only a single producer and consumer consistency maintenance between these local copies and the actual variable is trivial. The producer and consumer both use remote procedure calls to call the cmm. The variable *consumed* determines whether the producer can produce another item (if *consumed* is true) or the consumer can consume an item (*consumed* is false). The two processes are suspended until they receive this acknowledgement. If a call cannot be acknowledged the variables *producer_waiting* and *consumer_waiting* are set to true to indicate which process is waiting. This mechanism synchronizes the processes.

6 Analysis

This analysis of the implementations of the producer-consumer problem (Table 1) using message passing, RPC and DSM at user and operating system level will be based on the

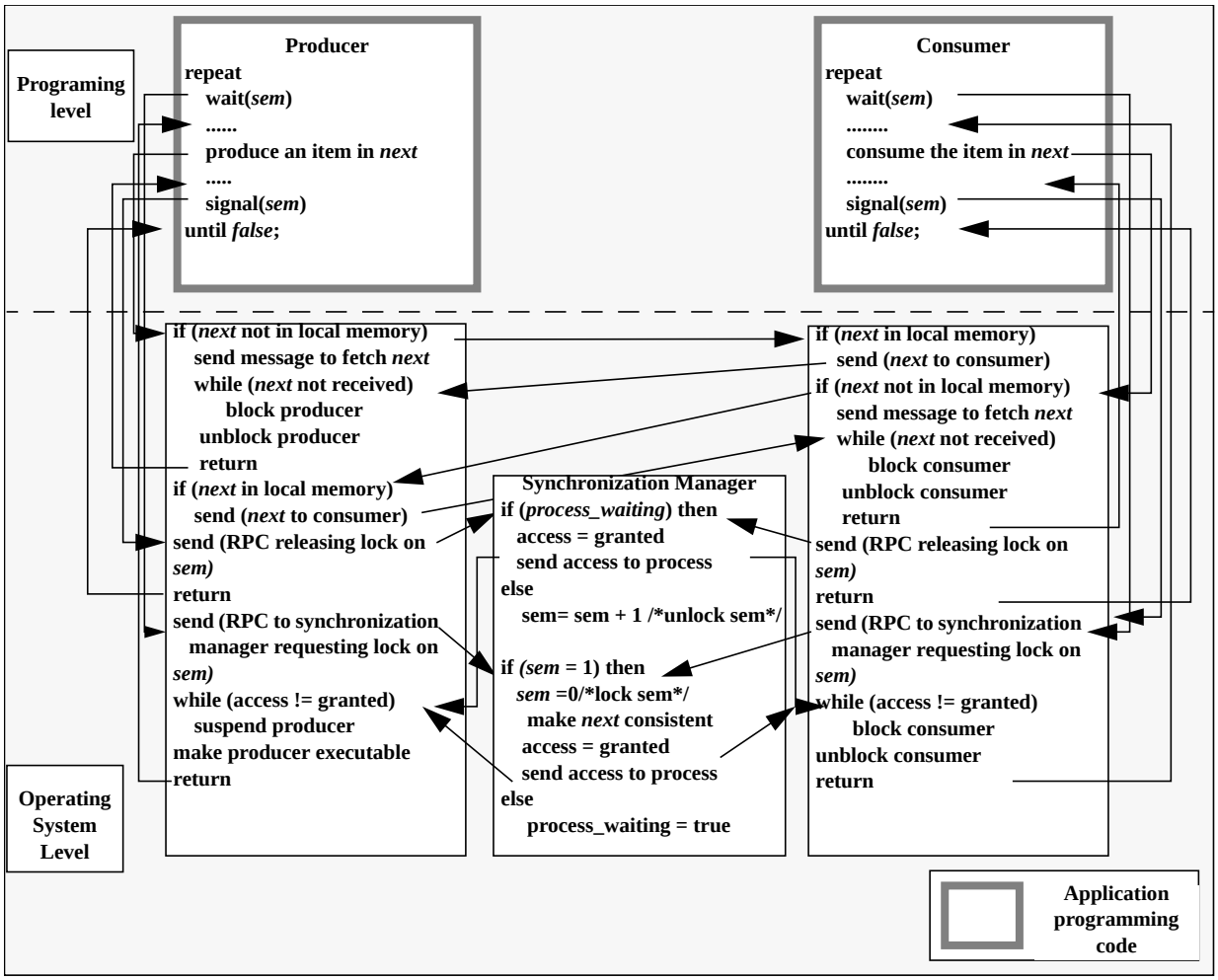

Figure 4. Producer-Consumer supported by DSM at Operating System Level on a Distributed System.

following three criteria:

— Ease of implementation for system designer. This will be a discussion of the implications for the system designer of the implementation of the syntax.

— Ease of use at application programming level. In this section we will discuss how easy the paradigm is to use at programming level. This criterion is difficult to evaluate as it involves attempting to measure programmer satisfaction. To an extent a quantitative measure of programmer satisfaction can be made by measuring the performance of the code written to solve the same problem using the three paradigms. The final criterion is a measure of this.

— Performance. The number of messages sent between processes is used as a measure of performance, since programmer satisfaction is often related to the execution speed of code.

6.1 Ease of Implementation for System Designer

When using the message passing paradigm for communication between processes the system must provide a mechanism for passing messages between processes. The simplest form of message passing, unstructured message passing requires the system to simply send the message to the location given as a parameter in the **send** primitive and to pass control back to the user level once the message buffer has been cleared. At the receive end the system expects a message

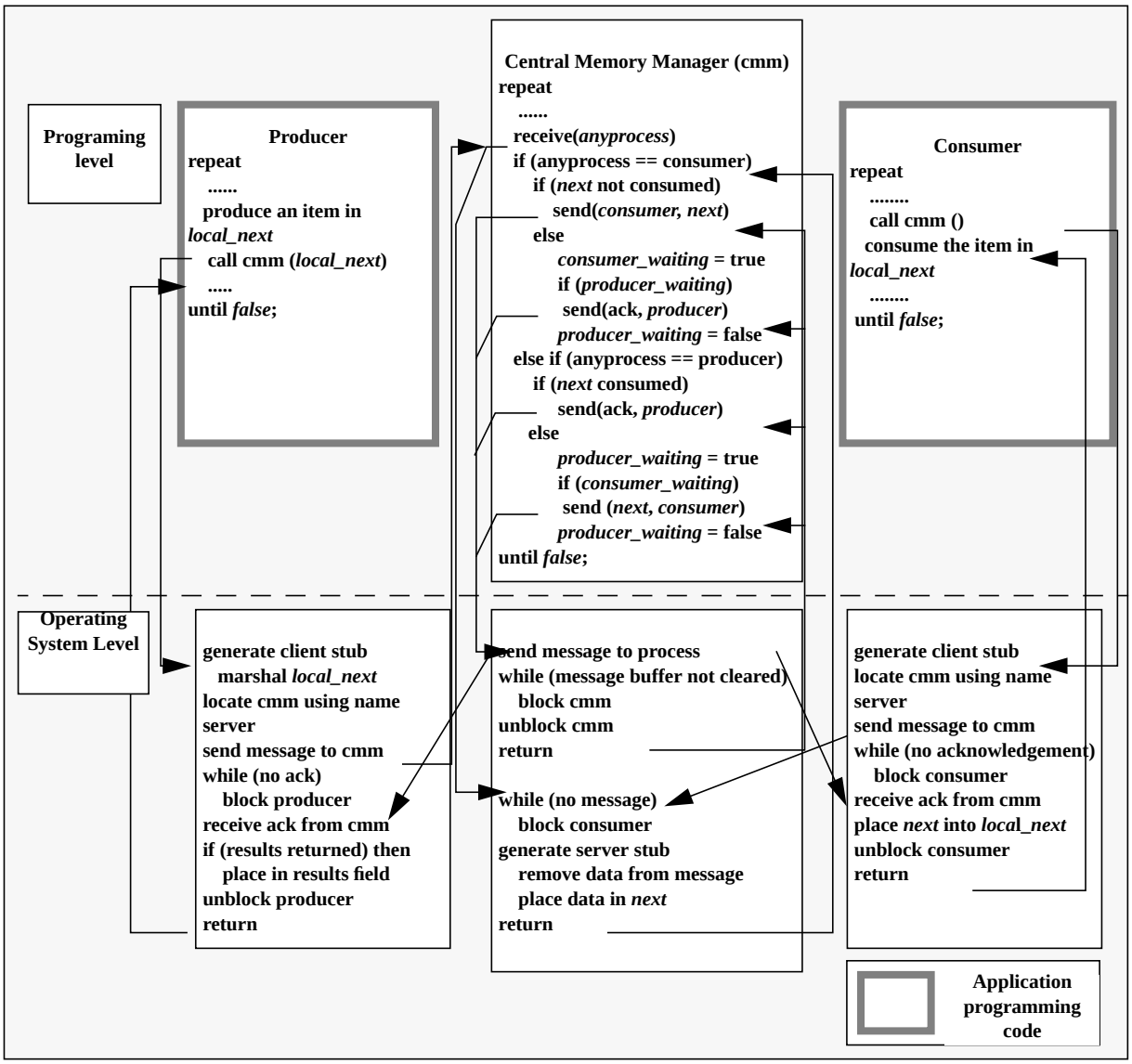

Figure 5. Producer-consumer supported by DSM at user level on a Distributed System

containing data of a known type from a known location.

The RPC paradigm requires the system designer to implement mechanisms to generate the client and server stub at the user level. These stubs have to be able to marshal and unmarshal parameters. The mechanism to send the message is, at this point, the same as that used in message passing, except the calling process must be blocked pending a reply. This represents an increase in abstraction which, in turn, means that the system programmer has extra work to do to add a layer over message passing making it easier for the application programmer to use.

The producer-consumer example for DSM, implemented at operating system level, shows that the system must now provide primitives, semaphores in this case, to allow application programmers synchronize the processes. In message passing and RPC synchronization is implicit in the organization of the code. The system must provide the mechanism to locate and fetch non-local data. If the data fetched is replicated, a mechanism must exist to make the multiple copies of the data consistent based on some protocol decided before implementation. This represents a further increase in abstraction over RPC. The system designer is still using message passing and RPC but their use is completely hidden at

application level. The code written at system level is complex compared with that for message passing and RPC.

In the final example, DSM implemented at user level the system designer must provide message passing and RPC and then use these two mechanisms to implement a central memory manager. The latter is a user level server which is accessed by the producer and consumer using RPC. The additional messages that are required to locate a copy of the required memory and to make multiple copies coherent in the system level DSM are not required in this implementation. The central memory manager maintains the single valid copy of the variable, however, this represents a bottleneck and for any more processes than a single producer and consumer would affect performance.

6.2 Ease of Use at Application Programming Level

In the case of message passing application programmers must be aware of the semantics of the implementation of the **send** and **receive** primitives and must take care of acknowledgements and synchronization. Application programmers must organize their code to ensure the correct synchronization of the processes.

When using RPC application programmers can call a procedure without knowing that it is remote and can program as they would when using local procedures.

DSM at system level, is the easiest paradigm to use at programming level, since it provides a familiar programming model for application programmers. The code is the same as that for a centralized system. The only indication that the system is not a centralized system might be the performance. Similarly DSM at user level is simple for the application programmer to use as the producer produces the item and calls the central memory manager and the consumer calls the central memory manager when it is ready to consume an item. The central memory manager takes care of synchronization and mutual exclusion.

6.3 Performance

Since interprocess communication consumes a large amount of time, a relative measure of performance between the three paradigms can be realized by comparing the number of messages required to be sent between processes. The message passing implementation requires only one message to be passed between the communicating processes while RPC requires two messages, ignoring the messages which would be required to locate the server using the name server in both cases. DSM implemented at operating system level, on the other hand, requires twelve messages, ignoring the messages required to make the replicated data consistent. The DSM implemented at user level requires four messages, the consumer and producer each call the cmm and receive a reply. Thus DSM has a large overhead which must be minimised as much as possible if it is to be tolerable.

7 Conclusion

In this report we have discussed and compared three high level communication paradigms message passing, remote procedure calls and distributed shared memory. We have based this discussion on their syntax and semantics and then discussed the implications of the implementation and use of these paradigms for the application programmer and the operating system designer.

Message passing is the least abstract of the paradigms and requires the least work from the operating system. It is, however, the most difficult for the application programmer to use

Table 1: Comparison of Message Passing, RPC and DSM

	Message Passing	RPC	DSM (OS Level)	DSM (User Level)
Ease of Implementation for System Designer	Mechanisms required: - send given data to given location; - receive given data type from given location.	Mechanisms required: - generate client and server stub; - marshal and unmarshal parameters; - send message.	Mechanisms required: - synchronize processes; - fetch non-local data; - maintaining consistency of replicated memory.	Mechanisms required: - Remote Procedure Calls. - Central Memory Manager
Ease of Use at Application Programming Level	Programmer must be aware of semantics of message passing implementation, and must organize code to include acknowledgement and synchronization.	Programmer calls procedure without knowing whether it is local or remote, call will block until reply received.	Programmer can use memory as though it were local.	Programmer uses RPC to access Central memory Manager.
Relative Performance	1 message	2 messages	12 messages	4 messages

since it requires the programmer to have knowledge of the semantics of the implementation of the primitives it uses. Remote procedures are called in the same way as local procedures. This demonstrates an increase in the level of abstraction compared to message passing, where programmers need not be aware of that procedure they are calling is remote. RPC requires mechanisms at the operating system level for marshalling the parameters, transporting them and unmarshalling them and likewise for the returned results. Finally in distributed shared memory the level of abstraction is increased even further and the application programmer accesses distributed memory in the same way as they would access centralized memory. This makes DSM easy for the application programmer to use, however, the price is the addition of mechanisms to the operating system which can add overheads to the system. These overheads have to measured against the gains to assess their value.

The producer-consumer example serves to give a direct comparison between the different paradigms using the same problem. The number of messages required show that, if message passing is assumed to be the major overhead, that DSM implementations could take in the order of six times as long as RPC. However, the obvious ease of use makes it worth further investigation, especially into techniques that may serve to reduce the number of messages required to implement DSM. A suite of communication paradigms made up of message passing, remote procedure calls and distributed shared memory would give application programmers complete flexibility to write programs specifically for their applications.

8 Bibliography

[Ben-Ari 82] M. Ben-Ari, *Principles of Concurrent Programming*, Prentice-Hall

International., 1982.

[Birrell et al. 84] A. Birrell, B. Nelson, *Implementing Remote Procedure Calls*, ACM Transactions on Computer Systems, Vol. 2, No. 1, February 1984, Pages 39-59.

[Coulouris et al. 89] G. F. Coulouris, J. Dollimore, *Distributed Systems. Concepts and Design.* Addison-Wesley Publishing Company.

[Goscinski 91] A. Goscinski, *Distributed Operating Systems. The Logical Design*., Addison-Wesley Publishing Company.

[Li et al. 88] K. Li, M. Stumm, D. Wortman., *Shared Virtual Memory Accomodating Heterogeneity*, Technical Report CS-TR-210-89, February 1989.

[Libes 85] Don Libes, *User-Level Shared Variables*, Proceedings, Tenth USENIX Conference, Summer 1985.

[Mosberger 93] D. Mosberger, *Memory Consistency Models*, Tech. Report TR 93/11, Dept. of Computer Science, Univ. of Arizona, 1993.

[Mullender 89] S. Mullender, *Interprocess Communication. Distributed Systems*, ACM Press, Addison-Wesley Publishing Company, 1989.

[Nitzberg et al. 94] B. Nitzberg, V. Lo, *Distributed Shared Memory: A Survey of Issues and Algorithms*, In Casavant T.L. and Singal M. (eds), Readings in Distributed Computing Systems, IEEE Press, 1994, pp 375-386.

[Silberschatz 85] A. Silberschatz, *Operating System Concepts.*, Addison-Wesley Publishing Company, 1985.

[Tam et al. 90] M. Tam, J. Smith, D. Farber, *A Survey of Distributed Shared Memory Systems*, ACM SIGOPS, June 1990.

[Tanenbaum 85] A. Tanenbaum, R. Van Renesse, *Distributed Operating Systems*, Computing Surveys, Vol. 17, No.4, December 1985.

[Tanenbaum 95] A. Tanenbaum, *Distributed Operating Systems*., Prentice Hall, 1995.