IOSR Journal of Engineering (IOSRJEN) **www.iosrjen.org**
ISSN (e): 2250-3021, ISSN (p): 2278-8719
Vol. 08, Issue 4 (April. 2018), ||VIII|| PP 09-15

A Study on Inter process Communications in Distributed Computer systems

Dr. Shamsudeen.E
Asst. Professor of Computer Applications, E.M.E.A. College of Arts and Science, Kondotty, Kerala, India
Corresponding Authors: Dr. Shamsudeen.E

Abstract: This paper deals with a study of interprocess communication in distributed systems. The interprocess communication means how the processes communicate with each other while fulfilling the users' needs. It says what the matters concerned while communicating between processes in distributed systems. The paper describes different interprocess communication mechanism like shared memory, message passing and remote procedure calls used in distributed systems. Finally the paper describes the how different distributed systems like Mach, Charlotte, Sprite, amoeba, V, and MOSIX accomplishing interprocess communication. All the system uses different methods to deals with interprocess communication.

Date of Submission: 12-04-2018 Date of acceptance: 30-04-2018

I. INTRODUCTION

A distributed system is a collection of independent computers that appears to its users as a single coherent system [1]. One important characteristic is that differences between the various computers and the ways in which they communicate are mostly hidden from users. Thus it provides a single system image to the user. The operating system hides all details of communication among processes from the user. The user doesn't know the existence of multiple systems. The inter process communication called IPC[1] in distributed systems is accomplished by different mechanisms and these mechanisms are different for different systems. Another important characteristic is that users and applications can interact with a distributed system in a consistent and uniform way, regardless of where and when interaction takes place.

This paper is organized in 4 sections. Section 1 provides a brief introduction about the interprocess communication and distributed operating system. Section 2 describes the interprocess communication in detail including shared memory, message passing and remote procedure call. Section 3 explains the interprocess communication in different distributed systems like Mach, Charlotte, Sprite and Amoeba, and MOSIX and then finally the conclusion is given in section 4

II. INTERPROCESS COMMUNICATION

Interprocess communication [1] is at the heart of all distributed systems and it is important to understand how the processes on different machines can exchange information. Inter Process Communication or IPC as name suggests, is used to share data between two applications or processes. The processes can be on the same computer or somewhere else in the network. Communication in distributed systems is always based on low-level message passing as offered by the underlying network. Expressing communication through message passing is harder than using primitives based on shared memory, as available for nondistributed platforms. Modem distributed systems often consist of thousands or even millions of processes scattered across a network with unreliable communication such as the Internet.

Inter process communication (IPC) is a mechanism which allows processes to communicate each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other using these two ways:

1. Shared Memory or distributed memory[2]
2. Message passing[2]

The Fig.1 below shows a basic structure of communication between processes via shared memory method and distributed memory. An operating system can implement both method of communication.

Shared Memory: Communication between processes using shared memory requires processes to share some variable and it completely depends on how programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process A and process B are executing simultaneously and they share some resources or use some information from other process, Process A generate information about certain computations or resources being used and keeps it as a record in shared memory. When Process B need

to use the shared information, it will check in the record stored in shared memory and take note of the information generated by Process A and act accordingly. Processes can use shared memory for extracting information as a record from other process as well as for delivering any specific information to other process.
Processors do not explicitly communicate with each other so communication protocols are hidden within the system. This means communication can be close to the hardware (or built into the processor), with the shared bus system deciding on how most efficiently to manage communication.
Since communication occurs as part of the memory system, smart shared memory architecture can make communication faster by taking advantage of the memory hierarchy.

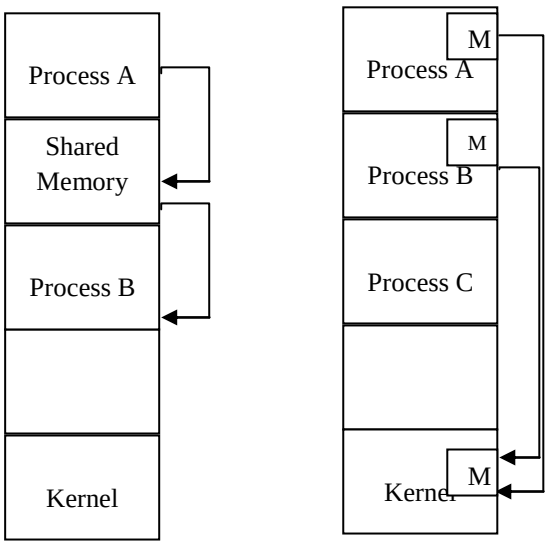

Fig. 1. Shared Memory Vs
Distributed Memory

Message Passing Systems: The message passing (Fig.2) communication protocols are fully under user control. These protocols are complex to the programmer causing communication to be treated as an I/O call for portability reasons. This can be expensive and slow. In this method, processes communicate with each other without using any kind of shared memory.

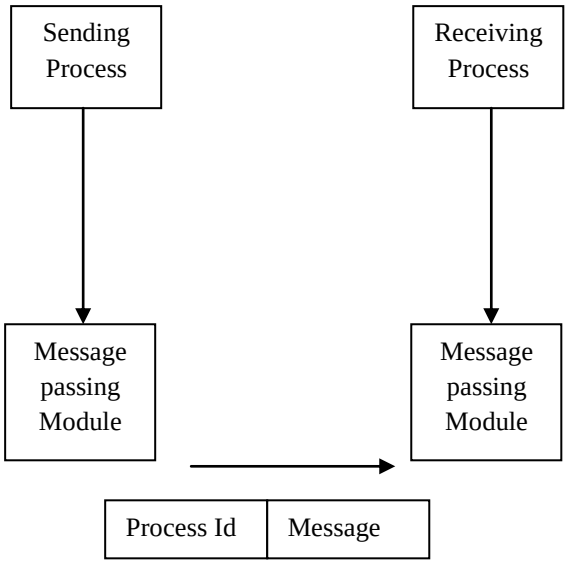

Fig.2. Message Passing Mechanism

The Characteristics of Message Passing: Message passing between a pair of processes supported by two communication operations: send and receive

- Defined in terms of destinations and messages.
- In order for one process A to communicate with another process B:
- A sends a message (sequence of bytes) to a destination
- Another process at the destination (B) receives the message.
- This activity involves the communication of data from the sending process to the receiving process and may involve the synchronization of the two processes

The message size can be of fixed size or of variable size. If it is of fixed size, it is easy for OS designer but complicated for programmer and if it is of variable size then it is easy for programmer but complicated for the OS designer. A standard message can have two parts: header and body.
The header part is used for storing Message type, destination id, source id, and message length and control information. The control information contains information like what to do if runs out of buffer space, sequence number, priority. Generally, message is sent using FIFO style.
Message passing leaves the programmer with the burden of the explicit control of the movement of data. Remote procedure calls (RPC) [3] [4] [5] relieves this burden by increasing the level of abstraction and providing semantics similar to a local procedure call.

Remote procedure call: A remote procedure call (RPC) is a network programming model or interprocess communication technique that is used for point-to-point communications between software applications. Client and server applications communicate during this process. A remote procedure call is sometimes called a function call or a subroutine call.
The way RPC (as in Fig.3) works is that a sender or client creates a request in the form of a procedure, function or method calls to a remote server, which RPC translates and sends. When the remote server receives the request, it sends a response back to the client and the application continues its process.
When the server processes the call or request, the client waits for the server to finish processing before resuming its process. In general, RPC applications use software modules called proxies and stubs, which make them look like local procedure calls (LPC).

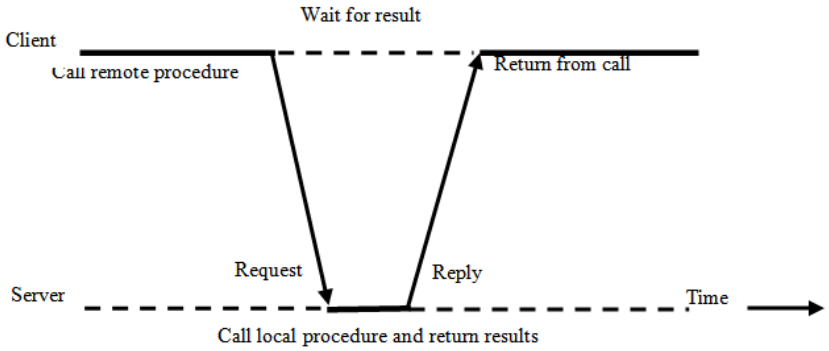

Fig.3. Principle of RPC between a client and server process

III. CASE STUDIES

Now, how different distributed system implement interprocess communication is discussed in detail, the discussion includes the distributed systems like, Mach, The V System, Amoeba and Sprite, Charlotte and then finally MOSIX is mentioned.

1. Mach

Mach [6] system ensures location independency during IPC by sending messages between location-independent ports. The messages contain typed data for ease of interpretation. The two components of Mach IPC are ports and messages. Almost everything in Mach is an object, and all objects are addressed via their communication ports. Messages are sent to these ports to initiate operations on the objects by the routines that implement the objects. By depending on only ports and messages for all communication, Mach delivers location independence of objects and security of communication. Data independence is provided by the NetMsgServer task. Mach ensures security by requiring that message senders and receivers have rights. A right consists of a port name and a capability—send or receive— on that port, and is much like a capability in object-oriented systems. Only one task may have received rights to any given port, but many tasks may have send rights. When an object is created, its creator also allocates a port to represent the object and obtains the access rights to that

port. Rights can be given out by the creator of the object, including the kernel [7] [8], and are passed in messages. If the holder of a receive right sends that right in a message, the receiver of the message gains the right, and the sender loses it. A task may allocate ports to allow access to any objects it owns or for communication. The destruction of either a port or the holder of the receive right causes the revocation of all rights to that port, and the tasks holding send rights can be notified if desired.

Ports: A port is implemented as a protected, bounded queue within the kernel of the system on which the object resides. If a queue is full, a sender may abort the send, wait for a slot to become available in the queue, or have the kernel deliver the message.

Messages: A message consists of a fixed-length header and a variable number of typed data objects. The header contains the destination's port name, the name of the reply port to which return messages should be sent, and the length of the message. Each data section may be a simple type (numbers or characters), port rights, or pointers to out-of-line data. Each section has an associated type, so that the receiver can unpack the data correctly even if it uses a byte ordering different from that used by the sender. The kernel also inspects the message for certain types of data.

The NetMsgServer: For a message to be sent (fig. 4) between computers, the message's destination must be located, and the message must be transmitted to the destination. One of Mach's tenets is that all objects within the system are location independent and that the location is transparent to the user. This tenet requires Mach to provide location transparent naming and transport to extend IPC across multiple computers. This naming and transport are performed by the Network Message Server (NetMsgServer), a user-level, capability-based networking daemon that forwards messages between hosts.

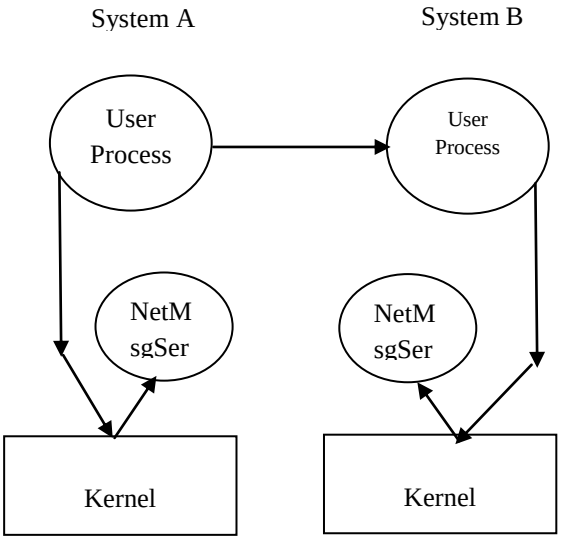

Fig. 4. IPC forwarding NetMsgServer

Synchronization through IPC: The IPC mechanism is extremely flexible and is used throughout Mach. For example, it may be used for thread synchronization. A port may be used as a synchronization variable and may have n messages sent to it for n resources. Any thread wishing to use a resource executes a receive call on that port. The thread will receive a message if the resource is available. Otherwise, it will wait on the port until a message is available there. To return a resource after use, the thread can send a message to the port.

2. The V System

In the V[9] system, the kernel interprocess communication facility was designed to provide a fast transport-level service for remote procedure calls, as characterized by file read and write operations. A client Send operation sends a request to a server and waits for, and returns, the response. The request corresponds to a (remote) call frame and the response corresponds to the return results. The server may execute as a separate dedicated server process, receiving and acting on the request following the message model of communication, That is, the receiver executes a Receive kernel operation to receive the next request message, invokes a

procedure to handle the request and then sends a response. Alternatively, the server may effectively execute as a procedure invocation executing on behalf of the requesting process, following the remote procedure call model. In the message model, the request is queued for processing should the addressed process be busy when the request is received. The client process is oblivious to which model is used by the server because the client process blocks waiting for the response in both cases. The message model appears preferable when the serialization of request handling is required. The procedure invocation model is preferred when there are significant performance benefits to concurrent request handling. It is depicted in Fig. 5.

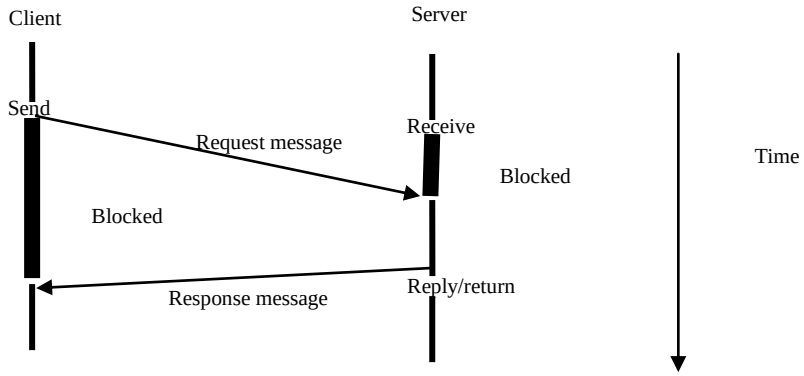

Fig. 5. Basic interaction using V IPC

The kernel of V provides time, process, memory, communication and device management in addition to the basic communication facilities. Each of these functions is implemented by a separate kernel module that is replicated in each host, handling the local processes, address spaces and devices, respectively. Each module is registered with the interprocess communication facility and invoked from the process level using the standard IPC facilities, the same as if the module executed outside the kernel as a process. Replicating these modules in each instantiation of the kernel and interfacing to these modules through the standard IPC mechanism has several significant advantages. First, operations on local objects, the common case, are performed fast because the operation is handled entirely by the local server. Also, the implementation of each module is simplified because each instance of the server module only manages local objects, not remote objects. Second, a client can access the kernel servers the same as the other servers (using the same IPC-based network-transparent access), allowing the use of remote procedure call mechanisms and run-time routines that support the high-level protocols.

3. Amoeba and Sprite

Both Amoeba [10] and Sprite [11] [12] implement communication mechanisms to enable processes to communicate with each other and to hide machine boundaries. Their mechanisms for doing so, however, are different. Amoeba presents the whole system as a collection of objects, on each of which a set of operations can be performed using RPC. Like Amoeba, Sprite uses RPC for kernel-to-kernel communication. Sprite has not really addressed the problems of building distributed applications, but it does provide a mechanism that can be used to support some kinds of client-server communication. Considering kernel communication in isolation, Amoeba and Sprite have more in common than not. Both use RPC to communicate between kernels on different machines. The implementations vary in minor ways. Sprite uses the implicit acknowledgements to avoid extra network messages when the same parties communicate repeatedly. On the other hand, Amoeba sends an explicit acknowledgement for the server's reply to make it possible for the server to free its state associated with the RPC. This simplifies the implementation of the RPC protocol but requires an additional packet to be built and delivered to the network. This difference is largely due to the necessity to perform a context-switch in Sprite when an RPC is received. For large RPCs, Sprite uses a blast protocol to send many packets without individual acknowledgments.

Communication in Sprite is integrated into the file system name space using "pseudo-devices," which permit synchronous and asynchronous communication between user processes using the file system read, write and I/O control kernel calls. User-level communication in Sprite is more expensive than in Amoeba for four reasons: first, Sprite's user-level communication is layered on a kernel-to-kernel RPC that is significantly slower than Amoeba's for small transfers and about the same performance for large transfers; second, as a result of this layering, the Sprite calls involve additional locking and copying that Amoeba avoids; third, all buffers in

Amoeba are contiguous and resident in physical memory, so no per-page checks need be performed; and fourth, Amoeba performs context switching much faster than Sprite. Thus, these differences in performance arise from both low level implementation differences, such as contiguous buffers and context-switching speeds, and the higher-level philosophical differences that led to Sprite's layered approach.

4. Charlotte

The distinctive features of Charlotte[13] IPC are duplex connections (links), dynamic link transfer, lack of buffering, non blocking send and receive, synchronous wait, ability to cancel pending operations, and selectivity of receipt.

Charlotte processes communicate by messages sent on links. A link is a software abstraction that represents a communications channel between two processes. Each of the two processes has a capability to its end of the link. This capability confers the right to send and receive messages on that link. These rights cannot be duplicated, restricted or amplified and can be transferred, but there is at most one capability in existence to any link end.

Synchronization: Basic communication activities never block. The Send and Receive service calls initiate communication but do not wait for completion. In this way, a process may post Send or Receive requests on many links without waiting for any to finish. This design allows processes to perform useful work while communication is in progress. In particular, servers can respond to clients without fear that a slow client will block the server. Posting a Send or Receive is synchronous (a process knows at what time the request was posted), but completion is inherently asynchronous (the data transfer may occur at any time in the future.) Charlotte provides three facilities for dealing with this asynchrony. First, versions of Send and Receive are available that block until they complete. Second, a process may explicitly wait for a Send or Receive to finish. Third, a process may poll the completion status of a Send or Receive. We also considered a fourth notification facility, software interrupts, but such a facility would have clashed with the blocking primitives we had, and it would have been difficult to use, because Charlotte provides no shared-memory synchronization primitives.

Request cancelling: A Receive or Send operation may be cancelled before it has completed. The Cancel request will fail if the operation has already been paired with a matching operation on the other end of the link. Cancellation is useful in several situations. A server may grow impatient if its response to a client has not been accepted after a reasonable amount of time. A sender may discover a more up-to-date version of data it is trying to Send. A receiver may decide it is willing to accept a message that requires a buffer larger than the one provided by its current Receive. A server that keeps Receives posted as a matter of course may decide it no longer wants messages on some particular link.

Message filtering: The Receive and Wait requests can specify a single link end or all link ends owned by the caller. Wait can also specify whether a Send or Receive event (or either) is to be awaited

5. MOSIX

MOSIX [14] [15] uses an optimized TCP/IP Protocol for Inter Process Communication (IPC).

IV. CONCLUSION

In distributed systems how the interprocess communication is made is discussed above in detail. The interprocess communication is accomplished either by shared memory which is less used in distributed systems or by message passing which is less reliable one. One more method which is widely used is by remote procedure call. More part of the discussion is focussed on the interprocess communication in different distributed systems. As the part of the case study the interprocess communications in distributed systems like Amoeba, Sprite, Mach, V, and Charlotte are discussed in detail. Finally communication in Mosix is mentioned as it is implemented with TCP/IP.

REFERENCES

[1]. Andrew S. Tanenbaum, Distributed Systems, Prentice-Hall, 2003.
[2]. G. Coulouris, J. Dollimore, and T. Kind berg, Distributed Systems Concepts and Designs, Third Edition, Addison Wesley , 2001.
[3]. K. M. Chandy and L. Lamport, Distributed Snapshots: Determining Global States of Distributed Systems, ACM Transactions on Computer Systems, Volume 3, pp 63-75, Number 1, 1985.
[4]. Pradeep K. Sinha, Distributed Operating Systems-Concepts and Design, Prentice Hall, 2008
[5]. Andrew S. Tanenbaum, Maarten Van Steen, Distributed Systems: Principles and Paradigm, Prentice-Hall, 2n Edn. 2002.

[6]. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young, "Mach: A new kernel foundation for UNIX development", Proc. Summer J986 USENIX Tech Conf Exhibition, 1986-June.
[7]. P. Bovet and M. Cesati, Understanding the Linux Kernel, Second Edition, O Reilly & Associates ,2002.
[8]. D. Bovet and M. Cesati, Understanding the Linux Kernel, Third Edition, O Reilly & Associates ,2006.
[9]. David R Cheriton. The V Distributed System
[10]. Andrew S. Tanenbaum & Gregory J. The Amoeba Distributed Operating System Sharp Vrije Universiteit De Boelelaan 1081a Amsterdam, The Netherlands
[11]. Welch, B. The Sprite Remote Procedure Call System. Technical report UCB/CSD 86/302, June 1986.
[12]. John K. Ousterhou4 Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite Network Operating System. Computer, vol. 21, no. 2, February 1988. pp. 23-36.
[13]. Y. Artsy, H.-Y. Chang, R. Finkel, "Interprocess communication in Charlotte", IEEE Software, vol. 4, no. 1, pp. 22-28, Jan. 1987
[14]. Barak, S. Guday, R.G. Wheeler The MOSIX Distributed Operating System Load Balancing for UNIX Series: Lecture Notes in Computer Science, Vol. 672, Springer
[15]. Amnon Barak and Richard Wheeler. MOSIX: An Integrated Multiprocessor UNIX

Dr. Shamsudeen.E "A Study on Inter process Communications in Distributed Computer systems”.” IOSR Journal of Engineering (IOSRJEN), vol. 08, no. 4, 2018, pp. 09-15.