LNMIIT

TITLE

Mutex in a Computer Cluster

Team Members

1. Mayank Periwal (Y10UC172)
2. Hardik Sankhla (y10uc122)
3. Harsh Khedar (y10uc125)

Instructor Name

Prof. Gaurav Somani

Course Name

Operating Systems

ABSTRACT

In this project we have covered the various aspects and advantages of a Computer Cluster. We have managed to bridge different systems as a single coherent system. In this Cluster we have implemented Mutex to integrate different systems into one working unit. For Mutex we have used different algorithms and for implementing these algorithms in our system, we have used MPI (Message Passing Interface) for the communication between individual systems.

INTRODUCTION ➔

A Distributed System is an application that executes a collection of protocols to coordinate the actions of multiple processes on a network, such that all components cooperate together to perform a single or small set of related tasks.
A mutual exclusion (Mutex) object allows multiple threads to synchronize access to a shared resource. Mutex is a crucial and integral part to ensure that different systems work as a single coherent system.

This assignment is to study and implement different algorithms of Mutex in a Distributed System and analyze their working. Here we will study some of the popular algorithms like Ricart's and Agarwala's Algorithm, Ring Algorithm, Central Server Algorithm, Election Algorithm etc. and make a detailed report about their analysis. Our major problem was to make a real time system as a Computer Cluster and implementing the algorithms in this real time system. Our working OS was fedora 16 on three different machines.

Making a Beowulf Cluster

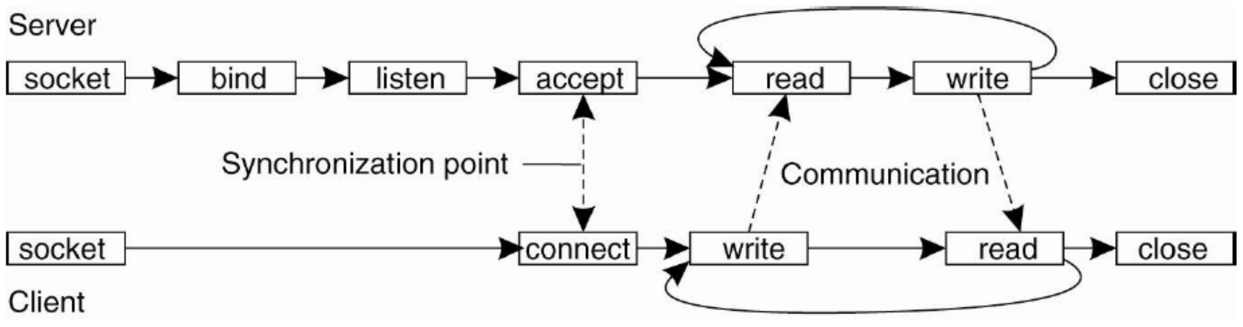
A **computer cluster** consists of a set of loosely connected computers that work together so that in many respects they can be viewed as a single system. The components of a cluster are usually connected to each other through fast local area networks, each node running its own instance on an operating system. Computer clusters emerged as a result of convergence of a number of computing trends including the availability of low cost microprocessors, high speed networks, and software for high performance distributed computing. Clusters are usually deployed to improve performance and availability over that of a single computer, while typically being much more cost-effective than single computers of comparable speed or availability.

A **Beowulf cluster** is a computer cluster of what are normally identical, commodity-grade computers networked into a small local area network with libraries and programs installed which allow processing to be shared among them. The result is a high-performance parallel computing cluster from inexpensive personal computer hardware.

Beowulf is a multi-computer architecture which can be used for parallel computations. It is a system which usually consists of one server node, and one or more client nodes connected together via Ethernet or some other network. It is a system built using commodity hardware components, like any PC capable of running a Unix-like operating system, with standard Ethernet adapters, and switches. It does not contain any custom hardware components and is trivially reproducible. Beowulf also uses commodity software like the FreeBSD, Linux or Solaris operating system, Parallel Virtual Machine (PVM) and Message Passing Interface (MPI).

Message Passing Interface (MPI)
It is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in Fortran 77 or the C programming language. Both point-to-point and collective communication are supported. MPI's goals are high performance, scalability, and portability.

The Message Passing Interface :



Point-to-point basics
A number of important MPI functions involve communication between two specific processes. A popular example is MPI_Send, which allows one specified process to send a message to a second specified process. Point-to-point operations, as these are called, are particularly useful in patterned or irregular communication, for example, a data-parallel architecture in which each processor routinely swaps regions of data with specific other

processors between calculation steps, or a master-slave architecture in which the master sends new task data to a slave whenever the prior task is completed.

Collective basics

Collective functions involve communication among all processes in a process group (which can mean the entire process pool or a program-defined subset). A typical function is the MPI_Bcast call (short for "broadcast"). This function takes data from one node and sends it to all processes in the process group. A reverse operation is the MPI_Reduce call, which takes data from all processes in a group, performs an operation (such as summing), and stores the results on one node. Reduce is often useful at the start or end of a large distributed calculation, where each processor operates on a part of the data and then combines it into a result.

Algorithms for Mutex

♦ Requirements for Mutual Exclusion Algorithms in Message-Passing Based Distributed Systems

ME1: at most one process may execute in the critical secion at any given point in time (safety)

ME2: requests to enter or exit the critical section will eventually succeed (liveness)

– impossible for one process to enter critical section more than once while other processes are awaiting entry

ME3: if one request to enter the critical section is issued before another request (as per the → relation), then the requests will be served in the same order

♦ Performance criteria to be used in the assessment of mutual exclusion algorithms

- Bandwidth consumed (corresponds to number of messages sent)
- Client delay at each entry and exit
- Throughput: number of critical region accesses that the system allows here: measured in terms of the synchronization delay between one process exiting the critical section and the next process entering

♦ Central Server-based Algorithm central server receives access requests

– if no process in critical section, request will be granted

– if process in critical section, request will be queued process leaving critical section

– grant access to next process in queue, or wait for new requests if queue is empty

♦ Properties

- Satisfies ME1 and ME2, but not ME3 (network delays may reorder requests)
- Two messages per request, one per exit, exit does not delay process
- Performance and availability of server are the bottlenecks

♦ Ring-based Algorithm

- Logical, not necessarily physical link: every process p(i) has connection to process p(i+1) mod N
- Token passes in one direction through the ring

- Token arrival

  – Only process in posession of token may access critical region

  – If no request upon arrival of token, or when exiting critical region, pass token on to neighbor

- Satisfies ME1 and ME2, but not ME3 performance

  – constant bandwidth consumption

  – entry delay between 0 and N message transmission times

  – synchronization delay between 1 and N message transmission times

♦ Algorithm by Ricart and Agrawala

- Based on multicast

  – process requesting access multicasts request to all other processes

  – process may only enter critical section if all other processes return

- Positive acknowledgement messages assumptions

  – all processes have communication channels to all other processes

  – all processes have distinct numeric ID and maintain logical clocks

- If request is broadcast and state of all other processes is RELEASED, then

- All processes will reply immediately and requester will obtain entry

- If at least one process is in state HELD, that process will not reply until it has left critical section, hence mutual exclusion

- If two or more processes request at the same time, whichever process' request bears lower timestamp will be the first to get N-1 replies

- In case of equal timestamps, process with lower ID wins

- performance

  – getting access requires 2(N-1) messages per request

  – Synchronization delay: just one round-trip time (previous algorithms: up to N)

Shortcomings

- In our Cluster, the was only one way password less ssh even after appending the public keys of all nodes in every system's Authorized_keys file. Due to this our code cannot be checked properly.
- There might be some errors in the MPI code that are still unresolved as we are not able to check our code properly.

References

- http://www.cs.rutgers.edu/~pxk/rutgers/notes/content/11-mutex.pdf
- Modern Distributed Systems by Dr. Gaurav Somani
- http://www.eecis.udel.edu/~pollock/367/manual/node18.html
- http://tldp.org/HOWTO/html_single/Beowulf-HOWTO/
- http://www.mcsr.olemiss.edu/bookshelf/articles/how_to_build_a_cluster.html

- http://www.wikipedia.org/
- https://computing.llnl.gov/tutorials/mpi/
- http://www.techotopia.com/index.php/Configuring_Fedora_Linux_Remote_Access_using_SSH
- http://tldp.org/HOWTO/NFS-HOWTO/server.html