IJESRT

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

Performance Comparison of Physical Clock Synchronization Algorithms

Z. Mahesh Kumar[*1], R. Manjula[2]

[*1,2] VIT University, Vellore (T.N.), India

mahesh.cse349@gmail.com

Abstract

This project mainly focuses on comparing and analyzing clock synchronization algorithms in distributed system. Clock synchronization is required for transaction processing applications, process control applications etc. This generates transmission delays and synchronization errors for processes and the clock synchronization algorithms try to synchronize the clocks in the system under the effect of these barriers. Two centralized clock synchronization algorithms are used for testing Cristian's and Berkeley clock synchronization algorithms, and the third, the distributed clock synchronization algorithm, Network time protocol for synchronization of clocks in the internet.

Keywords: Clock Synchronization, Coordinator, Distributed System, Global Time, Transmission Delay, Time Server.

Introduction

The three clock synchronization algorithms used for experiment in this report are Cristian's and Berkeley clock synchronization algorithms and Network Time Protocol. A distributed system consists of set of processes and these processes communicate by exchanging messages. In distributed system synchronization between processes is required for various purposes, for example in transaction processing and process control operations.

For processes to be synchronized and have a common view of global time, clock synchronization algorithms are applied for ensuring that physically dispersed processes have a common knowledge of time. The clock synchronization algorithms are of following types:

1. Distributed Algorithm: NTP (Network Time Service Protocol)
2. Centralized Algorithm:
   - Cristian's clock synchronization algorithm.
   - Berkeley clock synchronization algorithm.

Problem & Related Algorithms

Clock Synchronization Problem:

Cristian's algorithm and Berkeley algorithm are for the relative clock synchronization. Cristian's algorithm suffers from implementations using single server. In Cristian's algorithm we use centralized time server where as in Berkeley's, we can't establish it, and synchronizes all clocks to average and machines run time daemon. In Berkeley's master send offset by which each clock needs adjustment to each slave.

When skew is too great, we ignore readings from those clocks. The third one to synchronize the physical clocks, NTP, having goals of enable clients across Internet to be accurately synchronized to UTC despite message delays, enabling clients to synchronize frequently. NTP has synchronization models, multicast mode, procedure call mode, symmetric mode. NTP calculates offset for each pair of messages, delay and filter dispersion.

In this project we will analyze and compare performances of Cristian's and Berkeley clock synchronization algorithms on a set of processes having same set of variables and instructions and are asynchronous (each process execute actions with arbitrary speeds). The synchronization algorithms try to minimize effect of these delays and errors. The experiment is done on the basis of these parameters on varying number of processes in the system. Algorithm runs for a finite number of iterations in order to minimize the effect of delays and errors.

Cristian's algorithm:

The simplest algorithm for setting the time would be to simply issue a remote procedure call to a time server and obtain the time. That does not account for the network and processing delay. We can attempt to compensate for this by measuring the time (in local system time) at which the request is sent (*T0*) and the time at which the response is received (*T1*). Our best guess at the network delay in each direction is to assume that the delays to and

from are symmetric (we have no reason to believe otherwise). The estimated overhead due to the network delay is then (*T1- T0*)/2. The new time can be set to the time returned by the server plus the time that elapsed since the server generated the timestamp:

$$T_{new} = T_{server} + \frac{T_1 - T_0}{2}$$

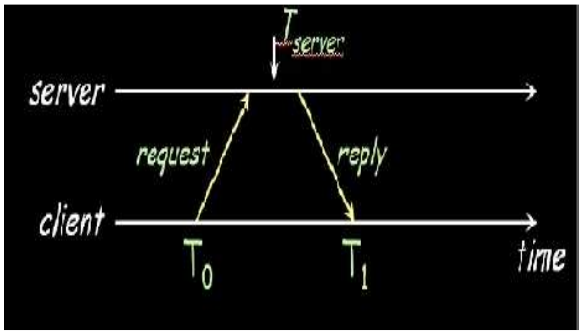


Figure 1: Cristian algorithm client-server model

Cristian's algorithm suffers from the problem that afflicts all single-server algorithms: the server might fail and clock synchronization will be unavailable. It is also subject to malicious interference.

Berkeley algorithm:

The Berkeley algorithm, developed by Gusella and Zatti in 1989, does not assume that any machine has an accurate time source with which to synchronize. Instead, it opts for obtaining an average time from the participating computers and synchronizing all machines to that average.

The machines involved in the synchronization each run a time daemon process that is responsible for implementing the protocol. One of these machines is elected (or designated) to be the master. The others are slaves. The server polls each machine periodically, asking it for the time. The time at each machine may be estimated by using Cristian's method to account for network delays. When all the results are in, the master computes the *average* time (including its own time in the calculation).

The hope is that the average cancels out the individual clock's tendencies to run fast or slow. Instead of sending the updated time back to the slaves, which would introduce further uncertainty due to network delays, it sends each machine the offset by which its clock needs adjustment. The operation of this algorithm is illustrated in Figure 7. Three machines have times of 3:00, 3:25, and 2:50. The machine with the time of 3:00 is the server (master). It sends out a synchronization query to the other machines in the group. Each of these machines sends a timestamp as a response to the query. The server now averages the three timestamps: the two it received and its own, computing (3:00+3:25+2:50)/3 = 3:05. Now it sends an offset to each machine so that the machine's time will be synchronized to the average once the offset is applied. The machine with a time of 3:25 gets sent an offset of -0:20 and the machine with a time of 2:50 gets an offset of +0:15. The server has to adjust its own time by +0:05.

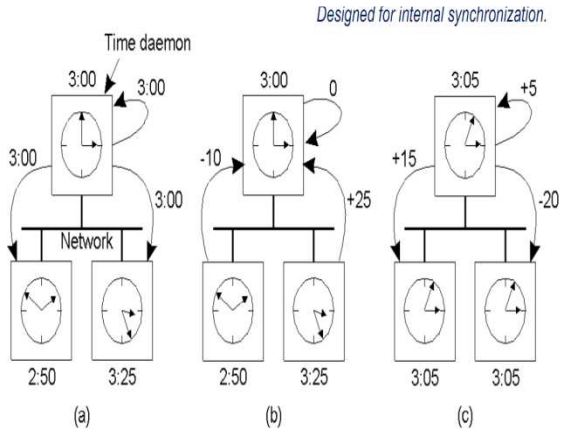


Figure 2: Berkley algorithm design

The algorithm also has provisions to ignore readings from clocks whose skew is too great. The master may compute a *fault-tolerant average* – averaging values from machines whose clocks have not drifted by more than a certain amount. If the master machine fails, any other slave could be elected to take over.

Network Time Protocol (NTP):

The Network Time Protocol [1991, 1992] is an Internet standard (version 3, RFC 1305) whose goals are to:

1. Enable clients across the Internet to be accurately synchronized to UTC (universal coordinated time) despite message delays. Statistical techniques are used for filtering data and gauging the quality of the results.
2. Provide a reliable service that can survive lengthy losses of connectivity. This means having redundant paths and redundant servers.
3. Enable clients to synchronize frequently and offset the effects of clock drift.
4. Provide protection against interference; authenticate that the data is from a trusted source.

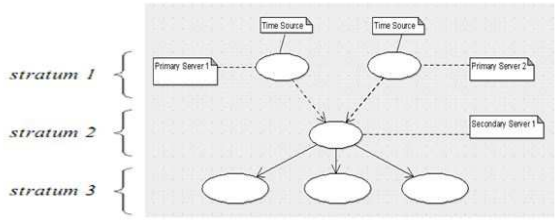


Figure 3: NTP Protocol server strata

The NTP servers are arranged into strata. The first stratum contains the primary servers, which are machines that are connected directly to an accurate time source. The second stratum contains the secondary servers. These machines are synchronized from the primary stratum machines. The third stratum contains tertiary servers that are synchronized from the secondary servers, and so on. Together, all these servers form the *synchronization subnet.* A machine will often try to synchronize with several servers, using the best of all the results to set its time. The *best* result is a function of a number of qualities, including: round-trip delay, consistency of the delay, round-trip error, and server's stratum, the accuracy of the server's clock, the last time the server's clock was synchronized, and the estimated drift on the server. Because a system may synchronize with multiple servers, its stratum is dynamic: it is based on the server used for the latest synchronization. If you synchronized from a secondary NTP server then you are in the third stratum. If, next time, you used a primary NTP server to synchronize, you are now in the second stratum.

Machines synchronize in one of the following modes:

1. *Symmetric active mode*:

In this mode, a host sends periodic messages regardless of the reachability state or stratum of its peer

2. *Symmetric passive*:

This mode is created when a system receives a message from a peer operating in symmetric active mode and persists as long as the peer is reachable and operating at a stratum less than or equal to the host. This is a mode where the host announces its willingness to synchronize and be synchronized by the peer. This mode offers the highest accuracy and is intended for use by master servers. A pair of servers exchanges messages with each other containing timing information. Timing data are retained to improve accuracy in synchronization over time.

3. *Procedure call mode*:

This call mode is similar to Cristian's algorithm; a client announces its willingness to by synchronize by the server, but not to synchronize the server.

4. *Multicast mode*:

This mode is intended for high speed LANs; relatively low accuracy but fine for many applications.

All messages are delivered unreliably via UDP. In both the procedure call mode and symmetric mode, messages are exchanged in pairs. Each message has the following timestamps:

Ti-3: local time when previous NTP message was sent.

Ti-2: local time when previous NTP message was received.

Ti-1: local time when current NTP message was sent.

The server notes its local time, *Ti*. For each pair, NTP calculates the offset (estimate of the actual offset between two clocks) and delay (total transit time for two messages). In the end, a process determines **three products:**

1. *Clock offset*: this is the amount that the local clock needs to be adjusted to have it correspond to a reference clock.

2. *Roundtrip delay*: this provides the client with the capability to launch a message to arrive at the reference clock at a particular time; it gives us a measure of the transit time of the mesge to a particular time server.

3. *Dispersion*: this is the "quality of estimate" (also known as *filter dispersion*) based on the accuracy of the server's clock and the consistency of the network transit times. It represents the maximum error of the local clock relative to the reference clock. By performing several NTP exchanges with several servers, a process can determine which server to favor. The preferred ones are those with a lower stratum and the lowest total filter dispersion. A higher stratum (less accurate) time source may be chosen if the communication to the more accurate servers is less predictable.

The *Simple Network Time Protocol*, SNTP (RFC 2030), is an adaptation of the Network Time Protocol that allows operation in a stateless remote procedure call mode or multicast mode. It is intended for environments when the full NTP implementation is not needed or is not justified. The intention is that SNTP be used at the ends of the synchronization subnet (high strata) rather than for synchronizing time servers.

SNTP can operate in either a unicast, multicast, or any cast modes:

- In unicast mode, a client sends a request to a designated server
- In multicast mode, a server periodically sends a broadcast or multicast message and expects no requests from clients

- In any-cast mode, a client sends a request to a local broadcast or multicast address and takes the first response received by responding servers.

From then on, the protocol proceeds as in unicast mode. NTP and SNTP messages are both sent via UDP (there is no point in having time reports delayed by possible TCP retransmissions).

Related Work & Methodology

JAVA RMI

Java RMI is actually the extension of Java Object Model to support distributed objects. In particular it allows objects to invoke methods on remote objects using the same syntax for local invocations. The object making remote invocation must handle Remote Exception and must implement Remote interface. Java implements the RMI functionality in the java.rmi package. RMI uses object serialization to marshal and un-marshal parameters and does not truncate types, supporting true object-oriented polymorphism.

Java RMI Architecture

The design goal for the RMI architecture was to create a Java distributed object model that integrates naturally into the Java programming language and the local object model. RMI architects have succeeded in creating a system that extends the safety and robustness of the Java architecture to the distributed computing world.

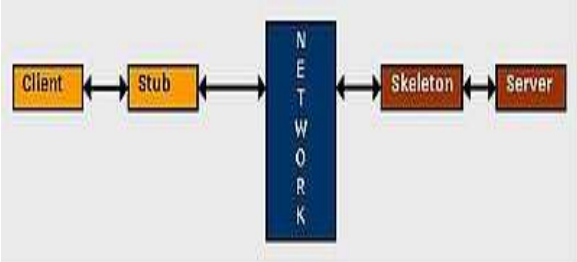


Figure 4. Java RMI Architecture Overview

Interfaces

The heart of RMI is the definition of behavior (method) which is called as the Interface. RMI considers interface and implementation as separate concepts. RMI allows the code that defines the method and the code that implements the method to remain separate and to run on separate JVMs.

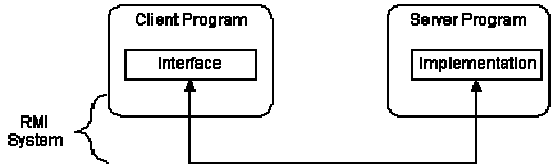


Figure 5: Java RMI Interfaces

In RMI, the definition of a remote service is coded using a Java interface. The implementation of the remote service is coded in a class. Therefore, the key to understanding RMI is to remember that *interfaces define behavior* and *classes define implementation.*

A Java interface does not contain executable code. RMI supports two classes that implement the same interface. The first class is the implementation of the behavior, and it runs on the server. The second class acts as a proxy for the remote service and it runs on the client. A client program makes method calls on the proxy object, RMI sends the request to the remote JVM, and forwards it to the implementation. Any return values provided by the implementation are sent back to the proxy and then to the client's program.

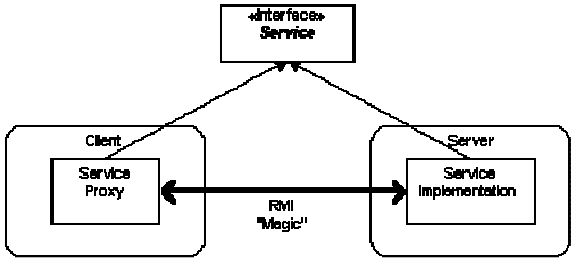


Figure 6: Java RMI Overview

Java RMI Detailed Architecture

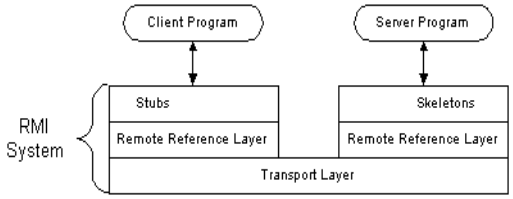


Figure 7: Java RMI Detailed Architecture

RMI implementation is built from three layers. First Stub or Skeleton layer which lies beneath the user layer.

Stub

A stub for a remote object acts as a client's local representative or proxy for the remote object. The caller invokes a method on the local stub which is responsible for carrying out the method call on the remote object. In RMI, a stub for a remote object implements the same set of remote interfaces that a remote object implements.

When a stub's method is invoked, it does the following:

1. Initiates a connection with the remote JVM containing the remote object.

2. Marshals (writes and transmits) the parameters to the remote JVM,
3. Waits for the result of the method invocation,
4. Un-marshal (reads) the return value or exception returned, and
5. Returns the value to the caller.

The stub hides the serialization of parameters and the network-level communication in order to present a simple invocation mechanism to the caller.

Skeleton

In the remote JVM, each remote object may have a corresponding skeleton. The skeleton is responsible for dispatching the call to the actual remote object implementation. When a skeleton receives an incoming method invocation it does the following:

- Un-marshal (reads) the parameters for the remote method,
- Invokes the method on the actual remote object implementation, and
- Marshals (writes and transmits) the result (return value or exception) to the caller.

Stubs and skeletons are generated by the rmic compiler.

Project Setup

Cristian Algorithm

In Cristian's algorithm each process sends a request and a delay is generated at each process, after which the request will be delivered to the message queue. The simulation engine removes the message from queue head, calculates a random transmission delay and sends a reply message to the destination message by adding destination identifier to the message and message delay. The calculated delay is represented as ***delay_at_rqst_queue*** in the equation.

Each process makes 30 requests to the Time Server and then averages the delay values which it gets in each "reply message" from the time server. A difference between the current process and the global time (Time Server) is calculated. These differences are displayed in the results.

The run () method for Cristian's Algorithm:

1. The messages are delivered in the increasing order of delay, to Time Server.
2. Time Server computes ***message_queue_delay*** (states for which the message was in queue) sends a reply message to requesting process.
3. Process sends 30 requests to the Time Server and gets a value for delay at request queue.
4. Calculates the average on 30 delay values and calculates its local time.

In Cristian's Algorithm, all the process send request for synchronizing its time. All processes suffer transmission delay. Each process makes 30 requests to Time Server. There are no faulty processes in the system (Time Server never crashes). Processes are asynchronous and delays are generated randomly. A process sends request after waiting random amount of time. In Berkeley Algorithm, all the processes get message in each iteration, they are asynchronous. The Coordinator never crashes. The error calculated by Coordinator is between 1-2 milliseconds and generated randomly.

Berkley's Algorithm

In Berkeley algorithm the Simulation Engine (Coordinator) polls processes and measures the clock difference between its time and time of other process in the system. It selects a largest set of processes that do not differ from its value by more than a fixed value (in the experiment fixed value is selected as 20 milliseconds). It then averages the differences of these processes. It also calculates a synchronization error for each process clock. The Coordinator asks each process to correct its clock by a quantity equal to the difference between the average value and the previously measured difference between the clock of the Coordinator and that of a process. The run () method for Berkeley Algorithm:

1. Coordinator calculates time difference between itself and other processes in the system.
2. Coordinator polls processes with a bound on difference (in experiment this value is 20 milliseconds)
3. Calculates the average
4. Calculates an error which represents error in approximation of other clocks in the system.
5. Inform all the processes about the correction & Does 10 iterations of above step.

Project Evaluation and Results

Christian Algorithm

We can implement using RMI

3 files should be there. Chat.java, Client.java, Server.java in a directory

STEP 1: START--> cmd --> (OPEN 2 COMMAND PROMPTS , ONE FOR CLIENT AND ONE FOR SERVER)

In the command Prompt, first go to the directory of files present

STEP 2:

javac *.java ENTER

OR

javac Chat.java Client.java Server.java (all three executing once)

STEP 3: start rmiregistryENTER

STEP 4: In one command prompt,

java Server ENTER(Now server is Up and Running)

STEP 5: In another command prompt

java Client ENTER

Request for time (to server) : i.e. give any string and ENTER, Server response time we can get, give another string for milliseconds and OVERHEAD etc, we get another response from the server that is number of mille seconds and overhead etc.

Berkley Algorithm

We can implement using RMI

3 files should be there. Chat.java, Client.java, Server.java in a directory

STEP 1: START--> cmd --> (OPEN 2 COMMAND PROMPTS, ONE FOR CLIENT AND ONE FOR SERVER)

In the command Prompt, first go to the directory of files present

STEP 2 : :> javac *.java ENTER OR

:> javac Chat.java Client.java Server.java (all three executing once)

STEP 3: :> start rmiregistry ENTER

STEP 4: In one command prompt,

:>java Server ENTER (Now server is Up and Running)

STEP 5: In another command prompt

:>java Client ENTER

Request for time (to server): i.e. give any string

:>Time please and ENTER

Server response time we can get

Give another string for milliseconds and OVERHEAD etc

:> milli seconds ENTER

Another Server response time we can get.

NTP (Network Time Protocol):

STEP1:

OPEN 2 COMMAND PROMPTS

:> javac *.java

STEP2:

java Primary1.java

(RETURNS ITS TIME) PRIMARY SERVER'S TIME

STEP3:

In One Window,

java Primary2

(waiting for secondary server)

In another Window

java Secondary1

ENTER YOUR MESSAGE: TIME PLEASE

First window (primary server) receives the request and gives its response back to secondary server.

(ITS COMPLETE TIME)

After getting that time, Secondary server synchronizes to it.

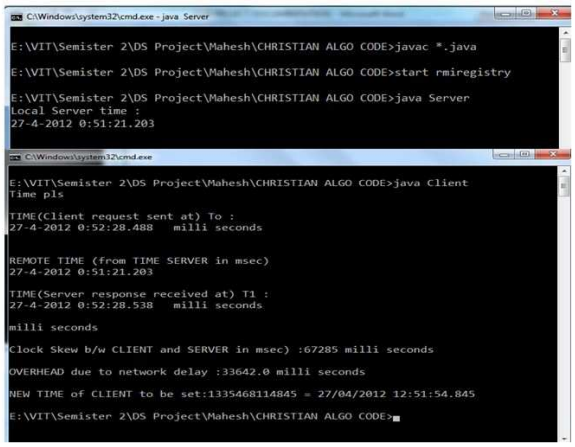
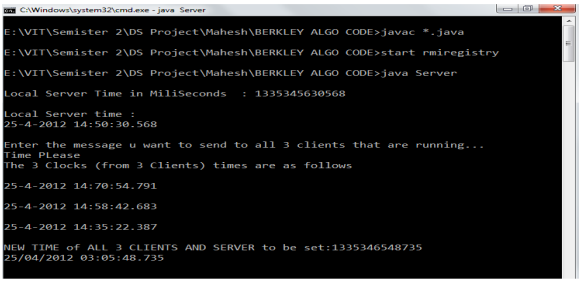


Figure 8: Christian Algorithm using RMI – server and client



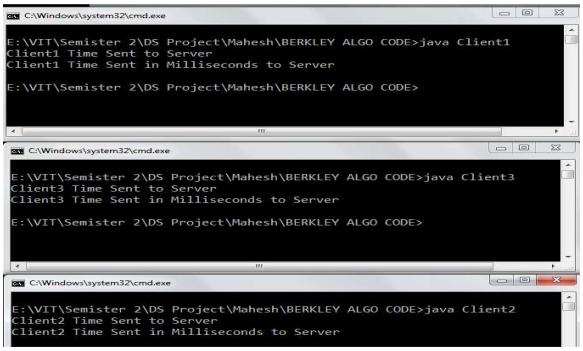


Figure 9: Berkley Protocol using RMI – Server and 3 Clients

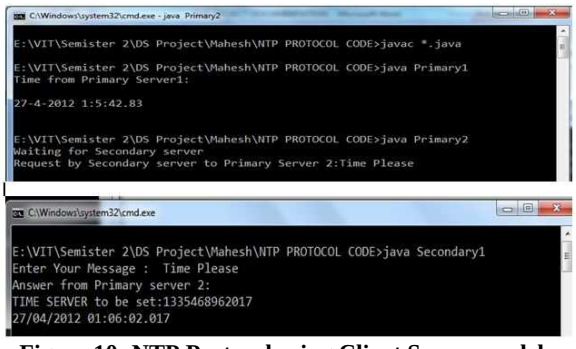


Figure 10: NTP Protocol using Client Server model

The experiment is conducted on 5, 10, 20 and 30 processes in the system. In Cristian algorithm

each process makes 30 requests to the Time Server and then calculates an average on these delay values. In results all the processes of the system are shown and the difference between them and global time are also shown. In Berkeley algorithm an average is calculated by the coordinator on the basis of equation (2). The graph is generated for one process by showing its difference at each iteration.

Note: For all diagrams, the Berkeley curve shows the difference curve for one process on which 10 iterations of Berkeley algorithm was performed. The value of the 4 processes on which Berkeley Algorithm were performed is shown in Table 1. Cristian curve shows the difference of all the processes in the system from the global time. Also in a system of five processes one process is Time Server, therefore only four processes are shown in the curve.

Comparison of Christian and Berkley in real-time

- **Observations**

5 processes:

Observation: In Cristian curve shows tendency to converge as the global time increases. Berkeley algorithm converge a process at a very fast rate with every iteration.

10 processes:

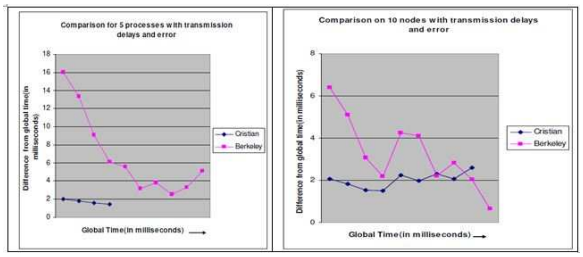
Observation: Cristian curve shows that as the number of processes increase the difference of each process from global time varies in a very small interval. In Berkeley Algorithm the process gets closer to global time value.

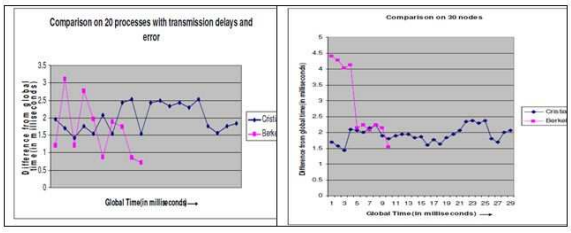
20 processes:

Observation: There is randomness in processes of Cristian Curve due to variable transmission delay. But this randomness is in a finite range. Berkeley curve shows that clock of the process converges closer to the global time after 10 iterations.

30 processes:

Observation: Cristian curve shows that as the number of processes increase the difference of each process from global time varies in a very small interval. Berkeley curve shows that number of iterations brings clock value closer to global time.





Conclusion

Considering the results by having from RMI and simulation engine, we can conclude that for internal clock synchronization, cristian's algorithm as well as Berkley algorithm used, in those if we are using less number of processes (clients) then Berkley algorithm shows much difference in global time to synchronize where as cristian's algorithm not shows that much. So when less number of clients are there in out subnet cristian's algorithm is best whereas for more number of clients are present Berkley algorithm much efficient. And finally NTP protocol, which is used for internet clocks synchronization, is best in its own conditions.

Clock synchronization is required for internal and external synchronization of clocks for various transaction processes and process controls. A more efficient algorithm will lead to a better convergence.

References

[1] F. Cristian Probabilistic clock synchronization. In Distributed Computing, volume 3, pages 146-158. Springer Verlag, 1989

[2] R. Gusella and S. Zatti, "TEMPO-A network time controller for a distributed Berkeley UNIX system."IEEE Distributed Processing Tech.Comm. Newslett., vol. 6, no. S1-2, pp. 7-15, June 1984.

[3] J.Y. Halpern et al., "Fault-Tolerant Clock Synchronization,"Proc. Third Ann. ACM Symp. Principles of Distributed Computing, ACM, New York, 1984, pp. 89-102.

[4] T. Clouser, R. Thomas, M. Nesterenko "Emuli: Emulated Stimuli for Wireless Sensor Network Experimentation", technical report TR-KSU-CS-2007-04, Kent State Univesity.

[5] Distributed Systems ©2000-2009 by Paul Krzyzanowski, Rutgers University.