ASSIGNMENT NO. 04

TITLE: BERKELEY ALGORITHM

AIM: To Implement Berkeley algorithm for clock synchronization.

OBJECTIVE:
Students should be able to understand:-
- Basic concept of Clock.
- Clock Synchronization.
- Berkeley algorithm for clock synchronization.

PROBLEM STATEMENT:-

Write a program to Implement Berkeley algorithm for clock synchronization.

TOOLS / ENVIRONMENT:
- S/W:
  - Can be developed on any platform Windows /Linux.
  - C/C++/Java Language
- H/W:
  - Any basic configuration loaded machine (e.g. P IV)

THEORY:

Introduction

Time is an important and interesting issue in distributed systems, for several reasons. First, time is a quantity we often want to measure accurately. In order to know at what time of day a particular event occurred at a particular computer it is necessary to synchronize its clock with an authoritative, external source of time. For example, an eCommerce transaction involves events at a merchant's computer and at a bank's computer. It is important, for auditing purposes, that those events are time-stamped accurately.

Synchronization and coordination are two closely related phenomena. In process synchronization we make sure that one process waits for another to complete its operation. When dealing with data synchronization, the problem is to ensure that two sets of data are the same. When it comes to coordination, the goal is to manage the interactions and dependencies between activities in a distributed system. From this perspective, one could state that coordination encapsulates synchronization.

Clock synchronization

In a centralized system, time is unambiguous. When a process wants to know the time, it simply makes a call to the operating system. If process A asks for the time, and then a little later process B asks for the time, the value that B gets will be higher than (or possibly equal to) the value A got. It will certainly not be lower. In a distributed system, achieving agreement on time is not trivial. Just think, for a moment, about the implications of the lack of global time on the Unix make program, as a simple example. Normally, in Unix large

programs are split up into multiple source files, so that a change to one source file requires only one file to be recompiled, not all the files. If a program consists of 100 files, not having to recompile everything because one file has been changed greatly increases the speed at which programmers can work. The way make normally works is simple. When the programmer has finished changing all the source files, he runs make, which examines the times at which all the source and object files were last modified. If the source file input.c has time 2151 and the corresponding object file input.o has time 2150, make knows that input.c has been changed since input.o was created, and thus input.c must be recompiled. On the other hand, if output.c has time 2144 and output.o has time 2145, no compilation is needed. Thus make goes through all the source files to find out which ones need to be recompiled and calls the compiler to recompile them. Now imagine what could happen in a distributed system in which there was no global agreement on time. Suppose that output.o has time 2144 as above, and shortly thereafter output.c is modified but is assigned time 2143 because the clock on its machine is slightly behind, as shown in Figure. Make will not call the compiler. The resulting executable binary program will then contain a mixture of object files from the old sources and the new sources. It will probably crash and the programmer will go crazy trying to understand what is wrong with the code.

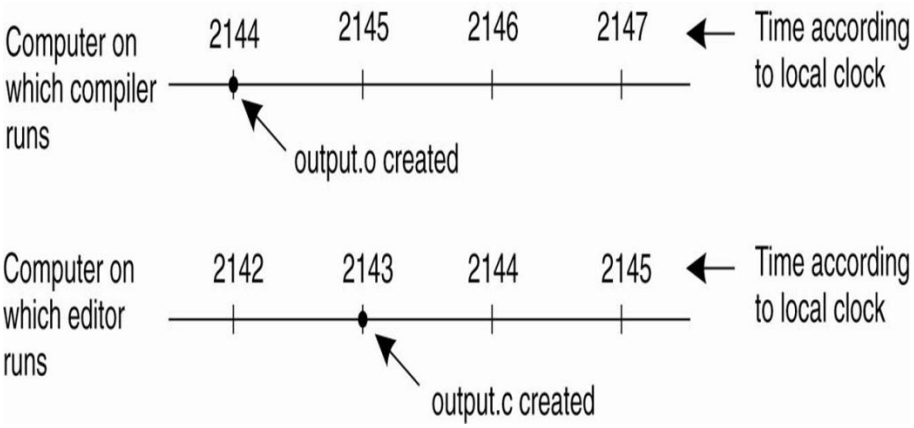


Figure: When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

Physical clocks

Nearly all computers have a circuit for keeping track of time. Despite the widespread use of the word “clock” to refer to these devices, they are not actually clocks in the usual sense. Timer is perhaps a better word. A computer timer is usually a precisely machined quartz crystal. When kept under tension, quartz crystals oscillate at a well-defined frequency that depends on the kind of crystal, how it is cut, and the amount of tension. Associated with each crystal are two registers, a counter and a holding register. Each oscillation of the crystal decrements the counter by one. When the counter gets to zero, an interrupt is generated and the counter is reloaded from the holding register. In this way, it is possible to program a timer to generate an interrupt 60 times a second, or at any other desired frequency. Each interrupt is called one clock tick. When the system is booted, it usually asks the user to enter the date and time, which is then converted to the number of ticks after some known starting date and stored in memory. Most computers have a special battery-backed up CMOS RAM so that the date and time need not be entered on subsequent boots. At every clock tick, the interrupt service procedure adds one to the time stored in memory. In this

way, the (software) clock is kept up to date. With a single computer and a single clock, it does not matter much if this clock is off by a small amount. Since all processes on the machine use the same clock, they will still be internally consistent. For example, if the file input.c has time 2151 and file input.o has time 2150, make will recompile the source file, even if the clock is off by 2 and the true times are 2153 and 2152, respectively. All that really matters are the relative times.

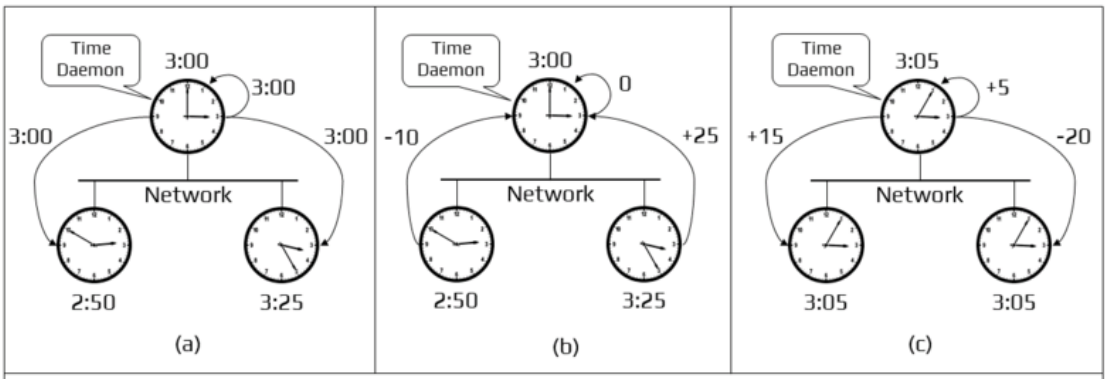
Clock synchronization algorithms

If one machine has a UTC receiver, the goal becomes keeping all the other machines synchronized to it. If no machines have UTC receivers, each machine keeps track of its own time, and the goal is to keep all the machines together as well as possible. Many algorithms have been proposed for doing this synchronization.

The Berkeley algorithm

In many clock synchronization algorithms the time server is passive. Other machines periodically ask it for the time. All it does is respond to their queries. In Berkeley Unix exactly the opposite approach is taken [Gusella and Zatti, 1989]. Here the time server (actually, a time daemon) is active, polling every machine from time to time to ask what time it is there. Based on the answers, it computes an average time and tells all the other machines to advance their clocks to the new time or slow their clocks down until some specified reduction has been achieved. This method is suitable for a system in which no machine has a UTC receiver. The time daemon's time must be set manually by the operator periodically. The method is illustrated in Figure. In Figure at 3:00, the time daemon tells the other machines its time and asks for theirs. In Figure 6.6(b) they respond with how far ahead or behind the time daemon they are. Armed with these numbers, the time daemon computes the average and tells each machine how to adjust its clock.

Note that for many purposes, it is sufficient that all machines agree on the same time. It is not essential that this time also agrees with the real time as announced on the radio every hour. If in our example of Figure 6.6 the time daemon's clock would never be manually calibrated, no harm is done provided none of the other nodes communicates with external computers. Everyone will just happily agree on a current time, without that value having any relation with reality. The Berkeley algorithm is thus typically an internal clock synchronization algorithm.



(a) The time daemon asks all the other machines for their clock values.
(b) The machines answer.
(c) The time daemon tells everyone how to adjust their clock.

Example:-
Let's sum-up steps followed to synchronize the clock using the Berkeley algorithm,

Nodes in the distributed system with their clock timings −

N1 -> 14:00 (master node)
N2 -> 13: 46
N3 -> 14: 15

Step 1 – The Leader is elected, node N1 is the master in the system.

Step 2 – leader requests for time from all nodes.

N1 -> time : 14:00
N2 -> time : 13:46
N3 -> time : 14:20

Step 3 – The leader averages the times and sends the correction time back to the nodes.

N1 -> Corrected Time 14:02 (+2)
N2 -> Corrected Time 14:02 (+16)
N3 -> Corrected Time 14:02 (-18)

IMPLEMENTATION:
(Students should write here the implementation for their program. Students should attach printout of their programs with commands used to run the programs. Also attach the proper outputs of programs.)

CONLCUSION:
This shows how the synchronization of nodes of a distributed system is done using Berkeley's algorithm

REFERENCES: "**Distributed Systems: Concepts and Design**" by George Coulouris, J Dollimore and Tim Kindberg, Pearson Education, ISBN: 9789332575226, 5th Edition, 2017

FAQ:
1. What is meant by synchronization?
2. Why we need synchronization in Distributed System?
3. Is it possible to synchronize all the clocks in a Distributed System? If yes, how?
4. How do we measure time?
5. Explain any other clock synchronization algorithm.