4

Scheduling Real-Time Tasks in Multiprocessor and Distributed Systems

The use of multiprocessor and distributed systems in real-time applications is becoming popular. One reason for this popularity of multiprocessor and distributed systems is the recent drop in their prices. Now dual processor machines are available at 50 to 60,000 rupees and the prices are set to drop even further. Besides, distributed platforms such as networked PCs are common place. Another reason that attracts real-time system developers to deploy multiprocessor and distributed systems is the faster response times and fault-tolerance features of such systems. Further, distributed processing is often suitable for applications that are naturally distributed and the events of interest are generated at geographically distributed locations. An example of such an application is an automated petroleum refinery, where the plant is spread over a considerable geographic area.

As compared to scheduling real-time tasks on a uniprocessor, scheduling tasks on multiprocessor and distributed systems is much more difficult. We had seen that optimal schedulers for a set of independent real-time tasks have polynomial complexity (in the number of tasks to be scheduled). However, determining an optimal schedule for a set of real-time tasks on a multiprocessor or a distributed system is an NP-hard problem [12].

Multiprocessor systems are known as tightly coupled systems. This characteristic of a multiprocessor system denotes the existence of shared physical memory in the system. In contrast, a distributed system is called a loosely coupled system and is devoid of any shared physical memory. In a tightly coupled system, the interprocess communication (IPC) is inexpensive and can be ignored compared to task execution times. This is so because inter-task communication is achieved through reads and writes to the shared memory. However, the same is not true in the case of distributed computing systems where inter-task communication times are comparable to task execution times. Due to this, a multiprocessor system may use a centralized dispatcher/scheduler whereas a distributed system can not. A centralized scheduler would require maintaining the state of the various tasks of the system in a centralized data structure. This would require various processors in the system to update it whenever the state of a task changes and consequently, would result in high communicational overheads.

Scheduling real-time tasks on distributed and multiprocessor systems consists of two subproblems: task allocation to processors and scheduling tasks on the individual processors. The task assignment problem is concerned with how to partition a set of tasks and then how to assign these to the processors. Task assignment can either be static or dynamic. In the static allocation scheme, the allocation of tasks to nodes is permanent and does not change with time whereas in the dynamic task assignment, tasks are assigned to nodes as they arise. Thus, in the dynamic case, different instances of a task may be allocated to different nodes. After successful task assignment to the processors, we can consider the tasks on each processor individually and, therefore, the second phase of the multiprocessor and distributed systems reduces to the scheduling problem in uniprocessors.

The task allocation to processors in multiprocessor and distributed environments is an NP-hard problem and determining an optimal solution has exponential complexity. So, most of the algorithms that are deployed in practice are heuristic algorithms. Task allocation algorithms can be classified into either static or dynamic algorithms. In static algorithms all the tasks are partitioned into subsystems and each subsystem is assigned to a separate processor. In contrast, in a dynamic system tasks ready for execution are placed in one common priority queue and dispatched to processors for execution as the processors become available. It is, therefore, possible that different instances of periodic tasks execute on different processors. Most hard real-time systems built to date are all static in nature. However, intuitively a dynamic real-time system can make more efficient utilization of the available resources.

This chapter is organized as follows. We first discuss task allocation schemes for multiprocessors. We next discuss dynamic allocation of tasks to processors. Finally, we address the clock synchronization problem in distributed systems.

4.1 MULTIPROCESSOR TASK ALLOCATION

In this section we discuss a few algorithms for statically allocating real-time tasks to the processors of a multiprocessor system. We already know that in a static task allocation algorithm, allocation is made before run time, and the allocation remains valid throughout a full run of the system. As already mentioned, the task allocation algorithms for multiprocessors do not try to minimize communication costs as interprocess communication time is low. This is because in multiprocessors, communication time is the same as memory access time, due to the availability of the shared memory. This is the reason why the task allocation algorithms that we discuss in this section may not work satisfactorily in distributed environments. The allocation algorithms that we discuss are centralized algorithms. In the following, we discuss a few important multiprocessor task allocation algorithms.

Utilization Balancing Algorithm. This algorithm maintains the tasks in a queue in increasing order of their utilizations. It removes tasks one by one from the head of the queue and allocates them to the least utilized processor each time. The objective of selecting the least utilized processor each time is to balance utilization of the different processors. In a perfectly balanced system the utilization u_i at each processor equals the overall utilization of the processors $\bar{u}$ of the system. Here the utilization of a processor P_i is the summation of the utilization of all tasks assigned to it. If ST_i is the set of all tasks assigned to a processor P_i, then the utilization of the processor P_i is $u_i = \Sigma_{j \epsilon ST_i} u_{tj}$, where $\bar{u}_{tj}$ is the utilization due to the task T_j. If PR is the set of all processors in the systems, then $\Sigma_{j \epsilon PR}\, u_i$. However, using this algorithm it is very difficult to achieve perfect balancing of utilizations across different processors for an arbitrary task set. That is, it is very difficult to make $u_i = \bar{u}$ for each P_i. The simple heuristic used in this algorithm gives suboptimal results. The objective of any good utilization balancing algorithm is to minimize $\Sigma_{i=1}^{n} |(\bar{u} - u_i)|$, where n is number of processors in the system, $\bar{u}$ is average utilization of processors, and u_i is utilization of processor i.

U

P_i

$\bar{u}_i =$

$\bar{U} =$

$\Sigma \bar{U} \to 4.$

This algorithm is suitable when the number of processors in a multiprocessor is fixed. The utilization balancing algorithm can be used when the tasks at the individual processors are scheduled using EDF.

Next-Fit Algorithm for RMA. In this algorithm, a task set is partitioned so that each partition is scheduled on a uniprocessor using RMA scheduling. This algorithm attempts to use as few processors as possible. Unlike the utilization balancing algorithm this algorithm does not require the number of processors of the system to be predetermined and given before hand. It classifies the different tasks into a few classes based on the utilization of the task. One or more processors are assigned exclusively to each class of tasks. The essence of this algorithm is that tasks with similar utilization values are scheduled on the same processor.

In this algorithm, tasks are classified based on their utilization according to the following policy. If the tasks are to be divided into m classes, a task T_i belongs to a class j, $0 \le j < m$, if

$$(2^{\frac{1}{j+1}} - 1) < e_i/p_i \le (2^{\frac{1}{j}} - 1) \qquad \textbf{(4.1)}$$

Suppose, we wish to partition the tasks of a system into four classes. Then, by using Expr. 4.1, the different classes can be formulated depending on the utilization of the tasks as follows:

Class 1: $(2^{\frac{1}{2}} - 1) < C_1 \le (2^{\frac{1}{1}} - 1)$

Class 2: $(2^{\frac{1}{3}} - 1) < C_2 \le (2^{\frac{1}{2}} - 1)$

Class 3: $(2^{\frac{1}{4}} - 1) < C_3 \le (2^{\frac{1}{3}} - 1)$

Class 4: $0 < C_4 \le (2^{\frac{1}{4}} - 1)$

From the above, the utilization grid for the different classes can be found to be: class 1: (0.41, 1), class 2: (0, 26, 0.41), class 3: (0.19, 0.26), and class 4: (0, 0.19).

We can view Expr. 4.1 as defining grids on the utilization plot of the tasks. A task is assigned to a grid depending on its utilization. It is not difficult to observe that the size of the grids at higher task utilization values are coarser compared to that at low task utilization values. The grid size of class 1 tasks is 1 − 0.41 = 0.59, whereas the grid size of the class 3 tasks is 0.07. Simulation studies indicate that using the next fit algorithm at most 2.34 times the optimum number of processors are required. We now illustrate the working of this task allocation method using an example.

Example 4.1

The following table shows the execution times (in milliseconds) and periods (in milliseconds) of a set of 10 periodic real-time tasks. Assume that the tasks need to run on a multiprocessor with four processors. Allocate the tasks to processors using the next fit algorithm. Assume that the individual processors are to be scheduled using RMA algorithm.

TABLE 4.1 Task Set for Example 4.1

Task	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
e_i	5	7	3	1	10	16	1	3	9	17
p_i	10	21	22	24	30	40	50	55	70	100

Now, we can determine the utilization of the different tasks and based on that we can assign the different tasks to different classes. The following table shows the computation of the utilization of each task and based on that the assignment of the tasks to different classes.

TABLE 4.2 Solution for Example 4.1

Task	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
e_i	5	7	3	1	10	16	1	3	9	17
p_i	10	21	22	24	30	40	50	55	70	100
u_i	0.5	0.33	0.14	0.04	0.33	0.4	0.02	0.05	0.13	0.17
Class	1	2	4	4	2	2	4	4	4	3

$\frac{5}{10}$

Bin Packing Algorithm for EDF. This algorithm attempts to allocate tasks to the processors such that the tasks on the individual processors can be successfully scheduled using EDF. This means that tasks are to be assigned to processors such that the utilization at any processor does not exceed 1. Bin packing is a standard algorithmic problem. We formulate the task allocation problem as a bin packing problem in the following.

We are given n periodic real-time tasks. When the individual processors are to be scheduled using the EDF algorithm, the number bins necessary can be expressed as: $[\Sigma_{i=1}^{n} u_i]$. The bin packing problem is known to be NP-complete.

Several bin packing algorithms exist. The two that we discuss are the first-fit random algorithm and the first-fit decreasing algorithm. In the first fit random algorithm, tasks are selected randomly and assigned to processors in an arbitrary manner as long as the utilization of a processor does not exceed 1. In this scheme, at most 1.7 times the optimum number of processors are required.

In the first-fit decreasing algorithm, the tasks are sorted in non-increasing order of their CPU utilization in an ordered list (that is, the task with the highest utilization is assigned the first position, and so on). The tasks are selected one by one from the ordered list and assigned to the bin (processor) to which it can fit in (that is, does not cause the utilization to exceed 1). A task is assigned to the processor to which it fits first. Simulation studies involving large number of tasks and processors show that the number of processors required by this approach is 1.22 times the optimal number of processors.

4.2 DYNAMIC ALLOCATION OF TASKS

So far we had discussed static allocation of tasks to processors. However, in many applications tasks arrive sporadically at different nodes. In such a scenario, a dynamic algorithm is needed to handle the arriving tasks. Dynamic algorithms assume that any task can be executed on any processor. Many of the dynamic solutions are naturally distributed and do not assume that there is a central allocation policy running on some processor.

In the dynamic algorithms, rather than preallocating tasks to processors, the tasks are assigned to processors as and when they arise. Since the task allocation to nodes can be made on the instantaneous load position of the nodes, the achievable schedulable utilization in this case should be better than the static approaches. However, the dynamic approach incurs high run time overhead since the allocator component running at every node needs to keep track of the instantaneous load position at every other node. In contrast, in a static task allocation algorithm, tasks are permanently assigned to processors at the system initialization time and no overhead is incurred during run time. Also, if tasks are bound to a single processor or a subset of processors, dynamic allocation would be ineffective.

In this section we discuss two popular dynamic real-time task allocation algorithms.

Focussed Addressing and Bidding. In this method, every processor maintains two tables called status table and system load table. The status table of a processor contains information about the tasks which it has committed to run, including information about the execution time and periods of the tasks. The system load table contains the latest load information of all other processors of the system. From the latest load information at the other processors, the surplus computing capacity available at the different processors can be determined.

The time axis is divided into windows, which are intervals of fixed duration. At the end of each window, each processor broadcasts to all other processors the fraction of computing power in the next window that is currently free for it—that is, the fraction of the next window for which it has no committed tasks. Every processor on receiving a broadcast from a node about the load position updates the system load table. When tasks arise at a node, the node first checks whether the task can be processed locally at the node. If it can be processed, then it updates its status table. If not, it looks out for a processor to which it can offload the task.

While looking out for a suitable processor, the processor consults its system load table to determine the least loaded processors in the system which can accommodate this task. It then sends out Request for bids (RFBs) to these processors. While looking out for a processor, an overloaded processor checks its surplus information and selects a processor (called the focussed processor). However, remember that the information (i.e., the system load table) might be out of date. Therefore, there is a likelihood that by the time the processor with excess load sends a task to a focussed processor, the focussed processor might have already changed its status and become overloaded. For this reason, a processor can not simply off-load a task to another node based on the information it has in the system load table.

The problem of obsolete information at the nodes is overcome using the following strategy. A processor sends out RFBs only if it determines that the task would complete in time, even when the time needed to get the bids from the other processors and then sending out the task to the focussed processor. The criteria for selecting a processor may be based on factors such as proximity to the processor, its exact load information, etc.

The focussed addressing and bidding strategy, however, incurs high communication overhead in maintaining the system load table at the individual processors. Window size is an important parameter determining the communication overhead incurred. If the window size is increased, then the communication overhead decreases; however, the information at various processors would be obsolete. This may lead to a scenario where none of the focussed processors bids due to status change in the window duration. If the window duration is too small then the information would be reasonably uptodate at the individual processors, but the communication overhead in maintaining the status tables would be unacceptably high.

Buddy Algorithm. The buddy algorithm tries to overcome the high communication overhead of the focussed addressing and bidding algorithm. The buddy algorithm is very similar to focussed addressing and bidding algorithm, but differs in the manner in which the target processors are found.

In this algorithm, a processor can be in any of the following two states: underloaded and overloaded. The status of a node is underloaded if its utilization is less than some threshold value.

That is, a processor P_i is said to be underloaded if $u_i < Th$, The processor is said to be overloaded if its utilization is greater than the threshold value (i.e., $u_i \geq Th$).

Unlike focussed addressing and bidding, in the buddy algorithm broadcast does not occur periodically at the end of every window. A processor broadcasts only when the status of a processor changes either from overloaded to underloaded or vice versa. Further, whenever the status of a processor changes, it does not broadcast this information to all processors and limits it only to a subset of processors called its *buddy set*. There are several criteria on which the buddy set of a processor design is based. First, it should be neither too large nor too small. In multi-hop networks, the buddy set of a processor is typically the processors that are its immediate neighbours.

4.3 FAULT-TOLERANT SCHEDULING OF TASKS

Task scheduling techniques can be used to achieve effective fault-tolerance in real-time systems. This is an efficient technique as it requires very little redundant hardware resources. Fault-tolerance can be achieved by scheduling additional ghost copies in addition to the primary copy of a task. The ghost copies may not be identical to the primary copy but may be stripped down versions that can be executed in shorter durations than the primary. The ghost copies of different tasks can be overloaded on the same slot and in case of a success execution of a primary, the corresponding backup may be deallocated.

4.4 CLOCKS IN DISTRIBUTED REAL-TIME SYSTEMS

Besides the traditional use of clocks in a computer system, clocks in a system are useful for two main purposes: determining timeouts and time stamping. Timeouts are useful to a real-time programmer in a variety of situations, and its use includes determining the failure of a task due to the missing of a deadline. Timeouts at both the sender and receiver ends is especially convenient for communication in distributed environments. They can be used as indicators for possible transmission faults or delays, or for non-existent receivers. Time stamping is useful in several applications. But a prominent use of time stamping is in message communication among tasks. The idea is that the message sender would also include the current time along a message. Time stamps not only give the receiver some idea about the age of a message, but can also be used for ordering purposes. Time stamping relies on good real-time clock services.

A distributed system typically has one clock at each node. Different clocks in a system tend to diverge since it is almost impossible to have two clocks that run exactly at the same speed. This lack of synchrony among clocks is expressed as the *clock slew* and determines the attendant drift of the clocks with time. Lack of synchrony and drift among clocks makes the time stamping and timeout operations in a distributed real-time system meaningless. Therefore, to have meaningful timeouts and time-stamping spanning more than one node of a distributed system, the clocks need to be synchronized. This makes clock synchronization a very important issue in distributed real-time systems. The following discussions are intended to provide some basic ideas regarding clock synchronization.

4.4.1 Clock Synchronization

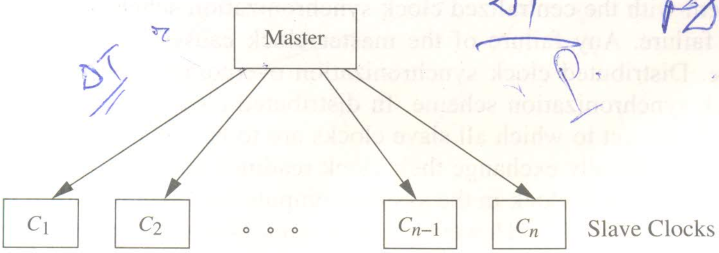
The goal of clock synchronization is to make all clocks in the network agree on their time values. For most distributed real-time applications, it is often sufficient to get the different clocks of a system agree on some time value which may be different from the world time standard. Many of you might know that the world time standard is called universal coordinated time (UTC). UTC is based on the international atomic time (TAI) maintained at Paris by averaging a number of atomic clocks from laboratories around the world. UTC signals can be made use of through GPS (Global Positioning System) receivers and specialized radio stations.

When the clocks of a system are synchronized with respect to one of the clocks of the system, it is called internal clock synchronization. When synchronization of a set of clocks with some external clock is performed, it is called external synchronization. There are two main approaches for internal synchronization: centralized clock synchronization and distributed clock synchronization.

4.5 CENTRALIZED CLOCK SYNCHRONIZATION

In centralized clock synchronization, one of the clocks is designated as the master clock. The other clocks of the system are called slaves and are kept in synchronization with the master clock. The master clock is also sometimes called the *time server*. The arrangement of the clocks in this scheme of clock synchronization has schematically been shown in Fig. 4.1. In Fig. 4.1, the clocks $C_1, \ldots C_n$ are the slave clocks that are to be synchronized with the master clock.

The server broadcasts its time to all other clocks for synchronization after every ΔT time interval. Once the slave clocks receive a time broadcast from the master, they set their clock as per the time at the master clock. The parameter ΔT should be carefully chosen. If ΔT is chosen to be too small, then the broadcast from the master is frequent and the slaves remain in good synchronization with the master at all times, but unnecessarily high communication overhead is incurred. If ΔT is chosen to be too large, then the clocks may drift too much apart. Let us assume that the maximum rate of drift between two individual clocks is restricted to ρ. It should be possible to determine the maximum drift rate between any two clocks, clock manufacturers usually specify this as one of the specification parameters of a clock. The parameter ρ is unit



ΔT

$\frac{\Delta T}{\rho}$

$\rho \Delta T < \varepsilon$

$\rho \Delta T$

$n 2 \rho \Delta T$

▲ **FIGURE 4.1**

Centralized Synchronization System

Example 4.2
Assume that the drift rate between any two clocks is restricted to $\rho = 5 \times 10^{-6}$. Suppose we want to implement a synchronized set of six distributed clocks using the central synchronization scheme so that the maximum drift between any two clocks is restricted to $\epsilon = 1mSec$ at any time, determine the period with which the clocks need to be resynchronized.

Solution. The maximum drift rate between any two arbitrary clocks when the clocks are synchronized using a central time server with a resynchronization interval of ΔT is given by $2\rho\Delta T < \epsilon$. Therefore, the required resynchronization interval ΔT can be expressed as:

$$\Delta T < \frac{1 \times 10^{-3}}{5 \times 10^{-6} \times \boxed{2}} \text{ sec} = \frac{10^{-3}}{10^{-5}} \text{ sec} = \frac{1}{10^{-2}} \text{ sec} = \underline{100 \text{ sec}}$$

$2\rho\Delta T < \epsilon$

Therefore, resynchronization period must be less than 100 sec.

$2\rho\Delta T < e$

$\Delta T < \epsilon/2\rho$

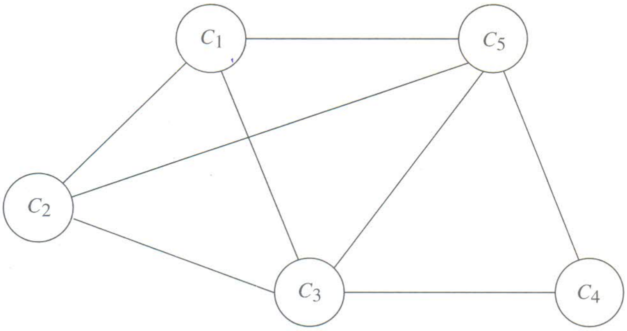
less since it measures drift (time) per unit time. Suppose clocks are resynchronized after every ΔT interval. Then, the drift of any clock from the master clock will be bounded by $\rho\Delta T$. From this, it can be concluded that the maximum drift between any two clocks will be limited to $2\rho\Delta T$.

In the above calculations, we have ignored the communication time. That is, the time it takes for a clock time broadcast to be received at the other clocks. Similarly, we have assumed that once the clock broadcasts are received, the clocks are set to the received time instantly. However, in reality it takes a finite amount of time to set a clock. Therefore, unless the communication time and the time to set the clock are suitably taken care of, the synchronized time would become slower and slower with respect to an external clock. Though they would still remain synchronized among themselves within the specified bound. However, it is very difficult to compensate these two terms in practical systems. We leave this as an exercise to the reader to determine the rate at which a centrally synchronized clock would drift with respect to an external clock.

$\Delta T < e/2\rho$

4.6 DISTRIBUTED CLOCK SYNCHRONIZATION

The main problem with the centralized clock synchronization scheme is that it is susceptible to single point failure. Any failure of the master clock causes breakdown of the synchronization scheme. Distributed clock synchronization overcomes this severe handicap of the centralized clock synchronization scheme. In distributed clock synchronization, there is no master clock with respect to which all slave clocks are to be set. But, all the clocks of a system are made to periodically exchange their clock readings among themselves. Based on the received time readings each clock in the system computes the synchronized time, and sets its clock accordingly (see Fig. 4.2). However, it is possible that some clocks are bad or become bad during the system operation. Bad clocks exhibit large drifts—drifts larger than the manufactured specified tolerance. Bad clocks may even stop keeping time all together. Fortunately, the bad clocks can be easily identified and taken care of during synchronization by

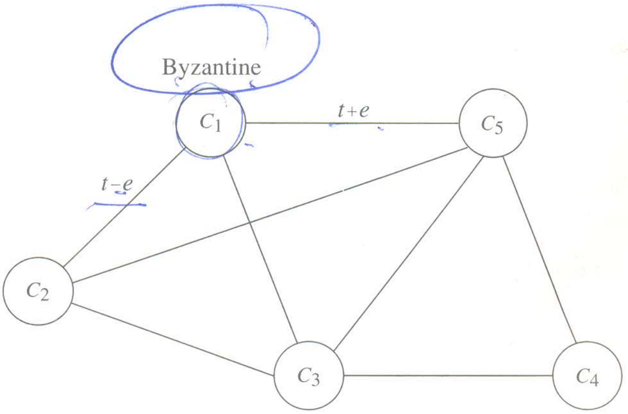


▲ **FIGURE 4.2**

Distributed Clock Synchronization

rejecting the time values of any clock which differs by any amount larger than the specified bound. A more insidious problem is posed by Byzantine clocks. A Byzantine clock is a two-faced clock. It can transmit different values to different clocks at the same time. In Fig. 4.3, C_1 is a Byzantine clock that is sending time value $t + e$ to clock C_5 and $t - e$ to clock C_2 at the same time instant.

It has been proved that if less than one-third of the clocks are bad or Byzantine (i.e., no more than one out of four are bad or Byzantine), then we can have the good clocks approximately synchronized. The following is the scheme for synchronization of the clocks. Let there be n clocks in a system. Each clock periodically broadcasts its time value at the end of certain interval. Assume that the clocks in the system are required to be synchronized within ϵ time units of each other. Therefore, if a clock receives a time broadcast that differs from its own time value by more than ϵ time units, then it can determine that the sending clock must be a bad one and safely ignore the received time values. Each clock averages out all good time values received after a broadcast step and sets its time value with this average value. This scheme has



▲ **FIGURE 4.3**

Byzantine Clock is a Two Faced Clock

been presented in pseudo code form in the following. Each clock C_i carries out the following operations:

```
Procedure distributed clock synchronization:
good-clocks = n;
for(j = 1; j < n;j++){
if (|(ci − cj)| > ε) good-clocks--;    //Bad clock
else total-time = total-time + cj;
ci = total-time/good-clocks;           //set own time equal to the computed
                                         time
}
```

Note that each clock of the system independently carries out the same set of steps. If all n clocks of a distributed system carry out the above steps, and at most m clocks out of n clocks are bad, and $n > 3 * m$, then we show in the following that the good clocks will be synchronized within $\frac{3\epsilon m}{n}$ bound.

We first show in the following theorem that a Byzantine clock can make two good clocks differ in their computed average time by at most $\frac{3\epsilon}{n}$.

THEOREM 4.1 *In a distributed system with n clocks, a single Byzantine clock can make two arbitrary clocks in a system to differ by $\frac{3\epsilon}{n}$ in time value, where ϵ represents the maximum permissible drift between two clocks.*

PROOF: Let us consider three clocks C_1, C_2, C_3 of a distributed system as shown in Fig. 4.3. In Fig. 4.3, C_2 and C_3 are two good clocks and C_1 is a Byzantine clock. The clocks C_2 and C_3 are required not to differ by more than ϵ. C_3 being a Byzantine clock shows two different values to C_1 and C_2. Now, the effect of the Byzantine clock in the total time calculation is to make the two good clocks differ by at most $3 * \epsilon$ as shown in Fig. 4.4. Therefore, the effect of a single Byzantine clock can make two arbitrary clocks in a system to differ by $\frac{3\epsilon}{n}$ in time value.

So, for m Byzantine clocks can make two good clocks differ by at most $3\epsilon m$ in average computation. From this it follows that from the time computation by the individual clocks, the individual clocks will be synchronized within $\frac{3\epsilon m}{n}$.

Let the time required for two clocks to drift from $\frac{3\epsilon m}{n}$ to ϵ be ΔT.

or, $$2\Delta T\rho \leq \frac{n\epsilon - 3\epsilon m}{n}$$

or, $$\Delta T \leq \frac{n\epsilon - 3\epsilon m}{n \times 2\rho} \tag{4.2}$$

where ΔT is the time required for two good clocks to drift from $\frac{3\epsilon m}{n}$ to ϵ.

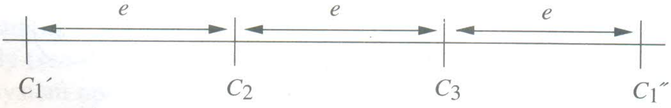


FIGURE 4.4
Drift Between Two Clocks in Presence of Byzantine Clocks

We know that

$$\Delta T \leq \frac{(3m+1)\epsilon - 3\epsilon m}{n \times 2\rho}$$

$$\Delta T \leq \frac{\epsilon}{2n\rho} \qquad \textbf{(4.3)}$$

We now illustrate the computation of the synchronization period for the distributed clock synchronization algorithm using an example.

Example 4.3

Let a distributed real-time system have 10 clocks, and it is required to restrict their maximum drift to ϵ = 1mSec. Let the maximum drift of the clocks per unit time (ρ) be 5×10^{-6}. Determine the required synchronization interval.

Solution. From the derivation for distributed clock synchronization, we have

$$\Delta T = \frac{10^{-3}}{2 \times 10 \times 5 \times 10^{-6}}$$

$$\Delta T = 10 \text{ sec}$$

Thus, the required synchronization interval is 10 sec.

SUMMARY

- In this chapter we first discussed how real-time tasks can be scheduled on multiprocessor and distributed computers. Task scheduling in multiprocessor and distributed systems is a much more complex problem than the uniprocessor scheduling problem.
- We saw that the task scheduling problem in multiprocessor and distributed systems consists of two sub-problems: task allocation to individual processors and task scheduling at the individual processors. The task allocation problem is an NP-hard problem.
- In a distributed real-time system, it is vital to have all the clocks in the systems synchronized within acceptable tolerance. We examined a centralized and a distributed clock synchronization scheme.
- Centralized clock synchronization is susceptible to single point failure. On the other hand, a distributed clock synchronization scheme can keep the good clocks in a distributed system synchronized only if no more than 25% of the clocks are bad or Byzantine.
- In the centralized synchronization scheme, unless the communication time and the time to set the clock are suitably compensated, the synchronized time may progressively become slower and slower with respect to the world time (UTC).

EXERCISES

1. State whether you consider the following statements TRUE or FALSE. Justify your answer in each case.

   (a) Optimal schemes for scheduling hard real-time tasks in multiprocessor computing environments have been devised by suitably extending the EDF algorithm.

(b) Using the distributed clock synchronization scheme, it is possible to keep the good clocks of a distributed system having 12 clocks synchronized, when two of the clocks are known to be Byzantine.

(c) In a distributed hard real-time computing environment, task allocation to individual nodes using a bin packing algorithm in conjunction with task scheduling at the individual nodes using the EDF algorithm can be shown to be the most proficient.

(d) The focussed addressing and bidding algorithm used for task allocation in distributed real-time systems statically allocates tasks to nodes.

(e) The focussed addressing and bidding algorithm for task allocation can handle dynamic task arrivals and is suited for use in multiprocessor-based real-time systems.

(f) Buddy algorithms require less communication overhead compared to focussed addressing and bidding algorithms in multiprocessor real-time task scheduling.

(g) The bin-packing scheme is the optimal algorithm for allocating a set of periodic real-time tasks to the nodes of distributed system.

(h) In a distributed system when the message communication time is non-zero and significant, the simple internal synchronization scheme using a time server makes the synchronized time incrementally delayed by the average message transmission time after every synchronization interval.

2. Explain why algorithms that can be satisfactorily used to schedule real-time tasks on multiprocessors often are not satisfactory to schedule real-time tasks on distributed systems, and vice versa?

3. In a distributed system, six clocks need to be synchronized to a maximum difference of 10 mSec between any two clocks. Assume that the individual clocks have a maximum rate of drift of 2×10^{-6}. Ignore clock set-up times and communication latencies.

   (a) What is the rate at which the clocks need to be synchronized using (i) a simple central time server method? (ii) simple internal synchronization (averaging) method?

   (b) What is the communication overhead in each of the two schemes?

   (c) Assuming the average communication latency to be 0.1 mSec, what would be the drift of the synchronized time with respect to the UTC for each of the two synchronization schemes?

4. (a) Why is the clock resolution provided to real-time programs by different commercial real-time operating systems rarely finer than few hundreds of milliseconds though giga hertz clocks are used by these systems?

   (b) Can clock resolution finer than milliseconds be provided to real-time programs at all? If yes, briefly explain how.

5. Why is it necessary to synchronize the clocks in a distributed real-time system? Discuss the relative advantages and disadvantages of the centralized and distributed clock synchronization schemes.

6. Describe the *focussed addressing and bidding* and the *buddy* schemes for running a set of real-time tasks in a distributed environment. Compare these two schemes with respect to communication overhead and scheduling proficiency.

7. Suppose a distributed system has 12 clocks. Assuming that no clocks in the system are Byzantine, determine the total number of message exchanges required per hour to keep the clocks synchronized within 1 mSec of each other in the centralized and distributed schemes. Assume that the maximum drift rate of the clocks is given to be $6 * 10^{-6}$.