However, what happens if the primary has not crashed, but is merely slow (i.e., we have an asynchronous system)? There is no way to distinguish between a slow primary and one that has gone down. Yet there is a need to make sure that when the backup takes over, the primary really stops trying to act like the primary. Ideally the backup and primary should have a protocol to discuss this, but it is hard to negotiate with the dead. The best solution is a hardware mechanism in which the backup can forcibly stop or reboot the primary. Note that all primary-backup schemes require agreement, which is tricky to achieve, whereas active replication does not always require an agreement protocol (e.g., TMR).

A variant of the approach of Fig. 4-22 uses a dual-ported disk shared between the primary and secondary. In this configuration, when the primary gets a request, it writes the request to disk before doing any work and also writes the results to disk. No messages to or from the backup are needed. If the primary crashes, the backup can see the state of the world by reading the disk. The disadvantage of this scheme is that there is only one disk, so if that fails, everything is lost. Of course, at the cost of extra equipment and performance, the disk could also be replicated and all writes could be done to both disks.

4.5.7. Agreement in Faulty Systems

In many distributed systems there is a need to have processes agree on something. Examples are electing a coordinator, deciding whether to commit a transaction or not, dividing up tasks among workers, synchronization, and so on. When the communication and processors are all perfect, reaching such agreement is often straightforward, but when they are not, problems arise. In this section we will look at some of the problems and their solutions (or lack thereof).

The general goal of distributed agreement algorithms is to have all the non-faulty processors reach consensus on some issue, and do that within a finite number of steps. Different cases are possible depending on system parameters, including:

1. Are messages delivered reliably all the time?
2. Can processes crash, and if so, fail-silent or Byzantine?
3. Is the system synchronous or asynchronous?

Before considering the case of faulty processors, let us look at the "easy" case of perfect processors but communication lines that can lose messages. There is a famous problem, known as the **two-army problem**, which illustrates the difficulty of getting even two perfect processors to reach agreement about 1 bit of information. The red army, with 5000 troops, is encamped in a valley. Two blue armies, each 3000 strong, are encamped on the surrounding hillsides overlooking the valley. If the two blue armies can coordinate their attacks on the red army, they will be victorious. However, if either one attacks by itself it will be slaughtered. The goal of the blue armies is to reach agreement about attacking. The catch is that they can only communicate using an unreliable channel: sending a messenger who is subject to capture by the red army.

Suppose that the commander of blue army 1, General Alexander, sends a message to the commander of blue army 2, General Bonaparte, reading: "I have a plan—let's attack at dawn tomorrow." The messenger gets through and Bonaparte sends him back with a note saying: "Splendid idea, Alex. See you at dawn tomorrow." The messenger gets back to his base safely, delivers his messages, and Alexander tells his troops to prepare for battle at dawn.

However, later that day, Alexander realizes that Bonaparte does not know if the messenger got back safely and not knowing this, may not dare to attack. Consequently, Alexander tells the messenger to go tell Bonaparte that his (Bonaparte's) message arrived and that the battle is set.

Once again the messenger gets through and delivers the acknowledgement. But now Bonaparte worries that Alexander does not know if the acknowledgement got through. He reasons that if Bonaparte thinks that the messenger was captured, he will not be sure about his (Alexander's) plans, and may not risk the attack, so he sends the messenger back again.

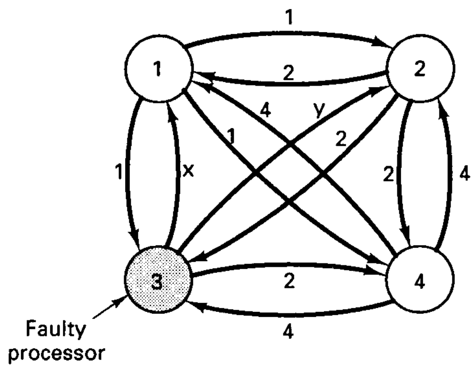
Even if the messenger makes it through every time, it is easy to show that Alexander and Bonaparte will never reach agreement, no matter how many acknowledgements they send. Assume that there is some protocol that terminates in a finite number of steps. Remove any extra steps at the end to get the minimum protocol that works. Some message is now the last one and it is essential to the agreement (because this is the minimum protocol). If this message fails to arrive, the war is off.

However, the sender of the last message does not know if the last message arrived. If it did not, the protocol did not complete and the other general will not attack. Thus the sender of the last message cannot know if the war is scheduled or not, and hence cannot safely commit his troops. Since the receiver of the last message knows the sender cannot be sure, he will not risk certain death either, and there is no agreement. Even with nonfaulty processors (generals), agreement between even two processes is not possible in the face of unreliable communication.

Now let us assume that the communication is perfect but the processors are not. The classical problem here also occurs in a military setting and is called the **Byzantine generals problem**. In this problem the red army is still encamped in the valley, but *n* blue generals all head armies on the nearby hills. Communication is done pairwise by telephone and is perfect, but *m* of the generals are traitors (faulty) and are actively trying to prevent the loyal generals from reaching agreement by feeding them incorrect and contradictory information (to model malfunctioning processors). The question is now whether the loyal generals can still reach agreement.

For the sake of generality, we will define agreement in a slightly different way here. Each general is assumed to know how many troops he has. The goal of the problem is for the generals to exchange troop strengths, so that at the end of the algorithm, each general has a vector of length n corresponding to all the armies. If general i is loyal, then element i is his troop strength; otherwise, it is undefined.

A recursive algorithm was devised by Lamport et al. (1982) that solves this problem under certain conditions. In Fig. 4-23 we illustrate the working of the algorithm for the case of $n = 4$ and $m = 1$. For these parameters, the algorithm operates in four steps. In step one, every general sends a (reliable) message to every other general announcing his truth strength. Loyal generals tell the truth; traitors may tell every other general a different lie. In Fig. 4-23(a) we see that general 1 reports 1K troops, general 2 reports 2K troops, general 3 lies to everyone, giving x, y, and z, respectively, and general 4 reports 4K troops. In step 2, the results of the announcements of step 1 are collected together in the form of the vectors of Fig. 4-23(b).



(a)

1 Got (1, 2, x, 4)
2 Got (1, 2, y, 4)
3 Got (1, 2, 3, 4)
4 Got (1, 2, z, 4)

(b)

1 Got	2 Got	3 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

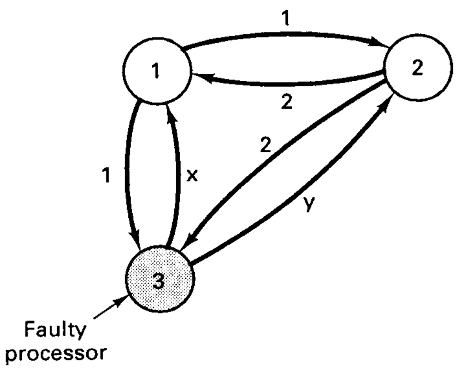
Fig. 4-23. The Byzantine generals problem for 3 loyal generals and 1 traitor. (a) The generals announce their troop strengths (in units of 1K). (b) The vectors that each general assembles based on (a). (c) The vectors that each general receives in step 2.

Step 3 consists of every general passing his vector from Fig. 4-23(b) to every other general. Here, too, general 3 lies through his teeth, inventing 12 new values, a through l. The results of step 3 are shown in Fig. 4-23(c). Finally, in step 4, each general examines the ith element of each of the newly received vectors. If any value has a majority, that value is put into the result vector. If no value has a majority, the corresponding element of the result vector is marked UNKNOWN. From Fig. 4-23(c) we see that generals 1, 2, and 4 all come to agreement on

(1, 2, UNKNOWN, 4)

which is the correct result. The traitor was not able to gum up the works.

Now let us revisit this problem for $m = 3$ and $n = 1$, that is, only two loyal generals and one traitor, as illustrated in Fig. 4-24. Here we see that in Fig. 4-24(c) neither of the loyal generals sees a majority for element 1, element 2, or element 3, so all of them are marked UNKNOWN. The algorithm has failed to produce agreement.



(a)

1 Got (1, 2, x)
2 Got (1, 2, y)
3 Got (1, 2, 3)

(b)

1 Got	2 Got
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

(c)

Fig. 4-24. The same as Fig. 4-23, except now with 2 loyal generals and one traitor.

In their paper, Lamport et al. (1982) proved that in a system with m faulty processors, agreement can be achieved only if $2m + 1$ correctly functioning processors are present, for a total of $3m + 1$. Put in slightly different terms, agreement is possible only if *more* than two-thirds of the processors are working properly.

Worse yet, Fischer et al. (1985) proved that in a distributed system with asynchronous processors and unbounded transmission delays, no agreement is possible if even one processor is faulty (even if that one processor fails silently). The problem with asynchronous systems is that arbitrarily slow processors are indistinguishable from dead ones. Many other theoretical results are known about when agreement is possible and when it is not. Surveys of these results are given by Barborak et al. (1993) and Turek and Shasha (1992).

4.6. REAL-TIME DISTRIBUTED SYSTEMS

Fault-tolerant systems are not the only kind of specialized distributed systems. The real-time systems form another category. Sometimes these two are combined to give fault-tolerant real-time systems. In this section we will examine various aspects of real-time distributed systems. For additional material, see for example, (Burns and Wellings, 1990; Klein et al., 1994; and Shin, 1991).

4.6.1. What Is a Real-Time System?

For most programs, correctness depends only on the logical sequence of instructions executed, not when they are executed. If a C program correctly computes the double-precision floating-point square root function on a 200-MHz engineering workstation, it will also compute the function correctly on a 4.77-MHz 8088-based personal computer, only slower.

In contrast, **real-time programs** (and systems) interact with the external world in a way that involves time. When a stimulus appears, the system must respond to it in a certain way and before a certain deadline. If it delivers the correct answer, but after the deadline, the system is regarded as having failed. *When* the answer is produced is as important as *which* answer is produced.

Consider a simple example. An audio compact disk player consists of a CPU that takes the bits arriving from the disk and processes them to generate music. Suppose that the CPU is just barely fast enough to do the job. Now imagine that a competitor decides to build a cheaper player using a CPU running at one-third the speed. If it buffers all the incoming bits and plays them back at one-third the expected speed, people will wince at the sound, and if it only plays every third note, the audience will not be wildly ecstatic either. Unlike the earlier square root example, time is inherently part of the specification of correctness here.

Many other applications involving the external world are also inherently real time. Examples include computers embedded in television sets and video recorders, computers controlling aircraft ailerons and other parts (so called fly-by-wire), automobile subsystems controlled by computers (drive-by-wire?), military computers controlling guided antitank missiles (shoot-by-wire?), computerized air traffic control systems, scientific experiments ranging from particle accelerators to psychology lab mice with electrodes in their brains, automated factories, telephone switches, robots, medical intensive care units, CAT scanners, automatic stock trading systems, and numerous others.

Many real-time applications and systems are highly structured, much more so than general-purpose distributed systems. Typically, an external device (possibly a clock) generates a stimulus for the computer, which must then perform certain actions before a deadline. When the required work has been completed, the system becomes idle until the next stimulus arrives.

Frequently, the stimulii are **periodic**, with a stimulus occurring regularly every ΔT seconds, such as a computer in a TV set or VCR getting a new frame every 1/60 of a second. Sometimes stimulii are **aperiodic**, meaning that they are recurrent, but not regular, as in the arrival of an aircraft in a air traffic controller's air space. Finally, some stimulii are **sporadic** (unexpected), such as a device overheating.

Even in a largely periodic system, a complication is that there may be many types of events, such as video input, audio input, and motor drive management, each with its own period and required actions. Figure 4-25 depicts a situation with three periodic event streams, *A*, *B*, and *C*, plus one sporadic event, *X*.

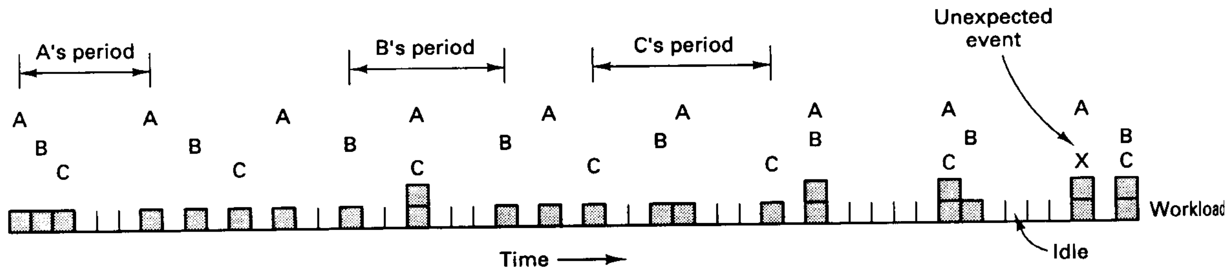


Fig. 4-25. Superposition of three event streams plus one sporadic event.

Despite the fact that the CPU may have to deal with multiple event streams, it is not acceptable for it to say: It is true that I missed event *B*, but it is not my fault—I was still working on *A* when *B* happened. While it is not hard to manage two or three input streams with priority interrupts, as applications get larger and more complex (e.g., automated factory assembly lines with thousands of robots), it will become more and more difficult for one machine to meet all the deadlines and other real-time constraints.

Consequently, some designers are experimenting with the idea of putting a dedicated microprocessor in front of each real-time device to accept output from it whenever it has something to say, and give it input at whatever speed it requires. Of course, this does not make the real-time character go away, but instead gives rise to a distributed real-time system, with its own unique characteristics and challenges (e.g., real-time communication).

Distributed real-time systems can often be structured as illustrated in Fig. 4-26. Here we see a collection of computers connected by a network. Some of these are connected to external devices that produce or accept data or expect to be controlled in real time. The computers may be tiny microcontrollers built into the devices, or stand-alone machines. In both cases they usually have sensors for receiving signals from the devices and/or actuators for sending signals to them. The sensors and actuators may be digital or analog.

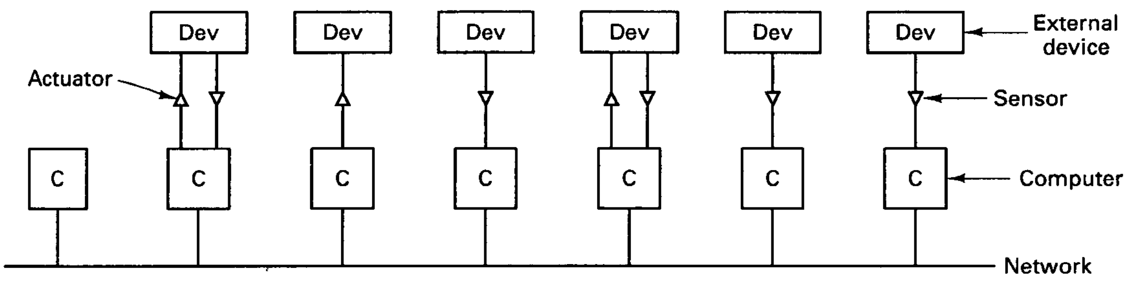


Fig. 4-26. A distributed real-time computer system.

Real-time systems are generally split into two types depending on how serious their deadlines are and the consequences of missing one. These are:

1. Soft real-time systems.
2. Hard real-time systems.

Soft real-time means that missing an occasional deadline is all right. For example, a telephone switch might be permitted to lose or misroute one call in 10^5 under overload conditions and still be within specification. In contrast, even a single missed deadline in a **hard real-time** system is unacceptable, as this might lead to loss of life or an environmental catastrophe. In practice, there are also intermediate systems where missing a deadline means you have to kill off the current activity, but the consequence is not fatal. For example, if a soda bottle on a conveyor belt has passed by the nozzle, there is no point in continuing to squirt soda at it, but the results are not fatal. Also, in some real-time systems, some subsystems are hard real time whereas others are soft real time.

Real-time systems have been around for decades, so there is a considerable amount of folk wisdom accumulated about them, most of it wrong. Stankovic (1988) has pointed out some of these myths, the worst of which are summarized here.

Myth 1: Real-time systems are about writing device drivers in assembly code.

This was perhaps true in the 1970s for real-time systems consisting of a few instruments attached to a minicomputer, but current real-time systems are too complicated to trust to assembly language and writing the device drivers is the least of a real-time system designer's worries.

Myth 2: Real-time computing is fast computing.

Not necessarily. A computer-controlled telescope may have to track stars or galaxies in real time, but the apparent rotation of the heavens is only 15 degrees of arc per hour of time, not especially fast. Here accuracy is what counts.

Myth 3: Fast computers will make real-time system obsolete.

No. They just encourage people to build real-time systems that were previously beyond the state-of-the-art. Cardiologists would love to have an MRI scanner that shows a beating heart inside an exercising patient in real time. When they get that, they will ask for it in three dimensions, in full color, and with the possibility of zooming in and out. Furthermore, making systems faster by using multiple processors introduces new communication, synchronization, and scheduling problems that have to be solved.

4.6.2. Design Issues

Real-time distributed systems have some unique design issues. In this section we will examine some of the most important ones.

Clock Synchronization

The first issue is the maintenance of time itself. With multiple computers, each having its own local clock, keeping the clocks in synchrony is a key issue. We examined this point in Chap. 3, so we will not repeat that discussion here.

Event-Triggered versus Time-Triggered Systems

In an **event-triggered real-time system**, when a significant event in the outside world happens, it is detected by some sensor, which then causes the attached CPU to get an interrupt. Event-triggered systems are thus interrupt driven. Most real-time systems work this way. For soft real-time systems with lots of computing power to spare, this approach is simple, works well, and is still widely used. Even for more complex systems, it works well if the compiler can analyze the program and know all there is to know about the system behavior once an event happens, even if it cannot tell when the event will happen.

The main problem with event-triggered systems is that they can fail under conditions of heavy load, that is, when many events are happening at once. Consider, for example, what happens when a pipe ruptures in a computer-controlled nuclear reactor. Temperature alarms, pressure alarms, radioactivity alarms, and other alarms will all go off at once, causing massive interrupts. This **event shower** may overwhelm the computing system and bring it down, potentially causing problems far more serious than the rupture of a single pipe.

An alternative design that does not suffer from this problem is the **time-triggered real-time system**. In this kind of system, a clock interrupt occurs every ΔT milliseconds. At each clock tick (selected) sensors are sampled and (certain) actuators are driven. No interrupts occur other than clock ticks.

In the ruptured pipe example given above, the system would become aware of the problem at the first clock tick after the event, but the interrupt load would not change on account of the problem, so the system would not become overloaded. Being able to operate normally in times of crisis increases the chances of dealing successfully with the crisis.

It goes without saying that ΔT must be chosen with extreme care. If it is too small, the system will get many clock interrupts and waste too much time fielding them. If it is too large, serious events may not be noticed until it is too late. Also, the decision about which sensors to check on every clock tick, and which to check on every other clock tick, and so on, is critical. Finally, some events may be shorter than a clock tick, so they must be saved to avoid losing them. They can be preserved electrically by latch circuits or by microprocessors embedded in the external devices.

As an example of the difference between these two approaches, consider the design of an elevator controller in a 100-story building. Suppose that the elevator is sitting peacefully on the 60th floor waiting for customers. Then someone pushes the call button on the first floor. Just 100 msec later, someone else pushes the call button on the 100th floor. In an event-triggered system, the first call generates an interrupt, which causes the elevator to take off downward. The second call comes in after the decision to go down has already been made, so it is noted for future reference, but the elevator continues on down.

Now consider a time-triggered elevator controller that samples every 500 msec. If both calls fall within one sampling period, the controller will have to make a decision, for example, using the nearest-customer-first rule, in which case it will go up.

In summary, event-triggered designs give faster response at low load but more overhead and chance of failure at high load. Time-trigger designs have the opposite properties and are furthermore only suitable in a relatively static environment in which a great deal is known about system behavior in advance. Which one is better depends on the application. In any event, we note that there is much lively controversy over this subject in real-time circles.

Predictability

One of the most important properties of any real-time system is that its behavior be predictable. Ideally, it should be clear at design time that the system can meet all of its deadlines, even at peak load. Statistical analyses of behavior assuming independent events are often misleading because there may be unsuspected correlations between events, as between the temperature, pressure, and radioactivity alarms in the ruptured pipe example above.

Most distributed system designers are used to thinking in terms of independent users accessing shared files at random or numerous travel agents accessing a shared airline data base at unpredictable times. Fortunately, this kind of chance behavior rarely holds in a real-time system. More often, it is known that when event *E* is detected, process *X* should be run, followed by processes *Y* and *Z*, in either order or in parallel. Furthermore, it is often known (or should be known) what the worst-case behavior of these processes is. For example, if it is known that *X* needs 50 msec, *Y* and *Z* need 60 msec each, and process startup takes 5 msec, then it can be guaranteed in advance that the system can flawlessly handle five periodic type *E* events per second in the absence of any other work. This kind of reasoning and modeling leads to a deterministic rather than a stochastic system.

Fault Tolerance

Many real-time systems control safety-critical devices in vehicles, hospitals, and power plants, so fault tolerance is frequently an issue. Active replication is sometimes used, but only if it can be done without extensive (and thus time-consuming) protocols to get everyone to agree on everything all the time. Primary-backup schemes are less popular because deadlines may be missed during cutover after the primary fails. A hybrid approach is to follow the leader, in which one machine makes all the decisions, but the others just do what it says to do without discussion, ready to take over at a moment's notice.

In a safety-critical system, it is especially important that the system be able to handle the worst-case scenario. It is not enough to say that the probability of three components failing at once is so low that it can be ignored. Failures are not always independent. For example, during a sudden electric power failure, everyone grabs the telephone, possibly causing the phone system to overload, even though it has its own independent power generation system. Furthermore, the peak load on the system often occurs precisely at the moment when the maximum number of components have failed because much of the traffic is related to reporting the failures. Consequently, fault-tolerant real-time systems must be able to cope with the maximum number of faults and the maximum load at the same time.

Some real-time systems have the property that they can be stopped cold when a serious failure occurs. For instance, when a railroad signaling system unexpectedly blacks out, it may be possible for the control system to tell every train to stop immediately. If the system design always spaces trains far enough apart and all trains start braking more-or-less simultaneously, it will be possible to avert disaster and the system can recover gradually when the power comes back on. A system that can halt operation like this without danger is said to be **fail-safe**.

Language Support

While many real-time systems and applications are programmed in general-purpose languages such as C, specialized real-time languages can potentially be of great assistance. For example, in such a language, it should be easy to express the work as a collection of short tasks (e.g., lightweight processes or threads) that can be scheduled independently, subject to user-defined precedence and mutual exclusion constraints.

The language should be designed so that the maximum execution time of every task can be computed at compile time. This requirement means that the language cannot support general **while** loops. Iteration must be done using **for** loops with constant parameters. Recursion cannot be tolerated either (it is beginning to look like FORTRAN has a use after all). Even these restrictions may not be enough to make it possible to calculate the execution time of each task in advance since cache misses, page faults, and cycle stealing by DMA channels all affect performance, but they are a start.

Real-time languages need a way to deal with time itself. To start with, a special variable, *clock*, should be available, containing the current time in ticks. However, one has to be careful about the unit that time is expressed in. The finer the resolution, the faster *clock* will overflow. If it is a 32-bit integer, for example, the range for various resolutions is shown in Fig. 4-27. Ideally, the clock should be 64 bits wide and have a 1 nsec resolution.

Clock resolution	Range
1 nsec	4 seconds
1 μsec	72 minutes
1 msec	50 days
1 sec	136 years

Fig. 4-27. Range of a 32-bit clock before overflowing for various resolutions.

The language should have a way to express minimum and maximum delays. In Ada®, for example, there is a delay statement that specifies a minimum value that a process must be suspended. However, the actual delay may be more by an unbounded amount. There is no way to give an upper bound or a time interval in which the delay is required to fall.

There should also be a way to express what to do if an expected event does not occur within a certain interval. For example, if a process blocks on a semaphore for more than a certain time, it should be possible to time out and be released. Similarly, if a message is sent, but no reply is forthcoming fast enough, the sender should be able to specify that it is to be deblocked after *k* msec.

Finally, since periodic events play such a big role in real-time systems, it would be useful to have a statement of the form

```
every (25 msec) { ... }
```

that causes the statements within the curly brackets to be executed every 25 msec. Better yet, if a task contains several such statements, the compiler should be able to compute what percentage of the CPU time is required by each one, and from these data compute the minimum number of machines needed to run the entire program and how to assign processes to machines.

4.6.3. Real-Time Communication

Communication in real-time distributed systems is different from communication in other distributed systems. While high performance is always welcome, predictability and determinism are the real keys to success. In this section we will look at some real-time communication issues, for both LANs and WANs. Finally, we will examine one example system in some detail to show how it differs from conventional (i.e., non-real-time) distributed systems. Alternative approaches are described in (Malcolm and Zhao, 1994; and Ramanathan and Shin, 1992)

Achieving predictability in a distributed system means that communication between processors must also be predictable. LAN protocols that are inherently stochastic, such as Ethernet, are unacceptable because they do not provide a known upper bound on transmission time. A machine wanting to send a packet on an Ethernet may collide with one or more other machines. All machines then wait a random time and then try again, but these transmissions may also collide, and so on. Consequently, it is not possible to give a worst-case bound on packet transmission in advance.

As a contrast to Ethernet, consider a token ring LAN. Whenever a processor has a packet to send, it waits for the circulating token to pass by, then it captures the token, sends its packet, and puts the token back on the ring so that the next machine downstream gets the opportunity to seize it. Assuming that each of the *k* machines on the ring is allowed to send at most one *n*-byte packet per token capture, it can be guaranteed that an urgent packet arriving anywhere in the system can always be transmitted within *kn* byte times. This is the kind of upper bound that a real-time distributed system needs.

Token rings can also handle traffic consisting of multiple priority classes. The goal here is to ensure that if a high-priority packet is waiting for transmission, it will be sent before any low-priority packets that its neighbors may have. For example, it is possible to add a reservation field to each packet, which can be increased by any processor as the packet goes by. When the packet has gone all the way around, the reservation field indicates the priority class of the next packet. When the current sender is finished transmitting, it regenerates a token bearing this priority class. Only processors with a pending packet of this class may capture it, and then only to send one packet. Of course, this scheme means that the upper bound of *kn* byte times now applies only to packets of the highest priority class.

An alternative to a token ring is the **TDMA (Time Division Multiple Access)** protocol shown in Fig. 4-28. Here traffic is organized in fixed-size frames, each of which contains *n* slots. Each slot is assigned to one processor, which may use it to transmit a packet when its time comes. In this way collisions are avoided, the delay is bounded, and each processor gets a guaranteed fraction of the bandwidth, depending on how many slots per frame it has been assigned.

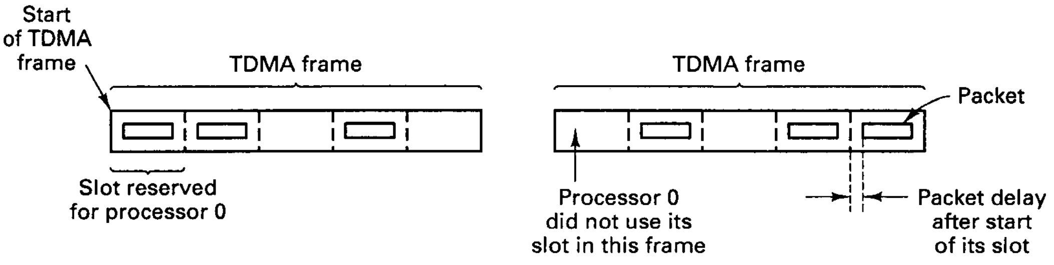

Start of TDMA frame
TDMA frame
TDMA frame
Packet
Slot reserved for processor 0
Processor 0 did not use its slot in this frame
Packet delay after start of its slot

Fig. 4-28. TDMA (Time Division Multiple Access) frames.

Real-time distributed systems operating over wide-area networks have the same need for predictability as those confined to a room or building. The communication in these systems is invariably connection oriented. Often, there is the ability to establish **real-time connections** between distant machines. When such a connection is established, the quality of service is negotiated in advance between the network users and the network provider. This quality may involve a guaranteed maximum delay, maximum jitter (variance of packet delivery times), minimum bandwidth, and other parameters. To make good on its guarantees, the network may have to reserve memory buffers, table entries, CPU cycles, link capacity, and other resources for this connection throughout its lifetime. The user is likely to be charged for these resources, whether or not they are used, since they are not available to other connections.

A potential problem with wide-area real-time distributed systems is their relatively high packet loss rates. Standard protocols deal with packet loss by setting a timer when each packet is transmitted. If the timer goes off before the acknowledgement is received, the packet is sent again. In real-time systems, this kind of unbounded transmission delay is rarely acceptable.

One easy solution is for the sender *always* to transmit each packet two (or more) times, preferably over independent connections if that option is available. Although this scheme wastes at least half the bandwidth, if one packet in, say, 10^5 is lost, only one time in 10^{10} will both copies be lost. If a packet takes a millisecond, this works out to one lost packet every four months. With three transmissions, one packet is lost every 30,000 years. The net effect of multiple transmissions of every packet right from the start is a low and bounded delay virtually all the time.

The Time-Triggered Protocol

On account of the constraints on real-time distributed systems, their protocols are often quite unusual. In this section we will examine one such protocol, **TTP (Time-Triggered Protocol)** (Kopetz and Grunsteidl, 1994), which is as different from the Ethernet protocol as a Victorian drawing room is from a Wild West saloon. TTP is used in the MARS real-time system (Kopetz et al., 1989) and is intertwined with it in many ways, so we will refer to properties of MARS where necessary.

A node in MARS consists of at least one CPU, but often two or three work together to present the image of a single fault-tolerant, fail-silent node to the outside world. The nodes in MARS are connected by two reliable and independent TDMA broadcast networks. All packets are sent on both networks in parallel. The expected loss rate is one packet every 30 million years.

MARS is a time-triggered system, so clock synchronization is critical. Time is discrete, with clock ticks generally occurring every microsecond. TTP assumes that all the clocks are synchronized with a precision on the order of tens of microseconds. This precision is possible because the protocol itself provides continuous clock synchronization and has been designed to allow it to be done in hardware to extremely high precision.

All nodes in MARS are aware of the programs being run on all the other nodes. In particular, all nodes know when a packet is to be sent by another node and can detect its presence or absence easily. Since packets are assumed not to be lost (see above), the absence of a packet at a moment when one is expected means that the sending node has crashed.

For example, suppose that some exceptional event is detected and a packet is broadcast to tell everyone else about it. Node 6 is expected to make some computation and then broadcast a reply after 2 msec in slot 15 of the TDMA frame. If the message is not forthcoming in the expected slot, the other nodes assume that node 6 has gone down, and take whatever steps are necessary to recover from its failure. This tight bound and instant consensus eliminate the need for time-consuming agreement protocols and allow the system to be both fault tolerant and operate in real time.

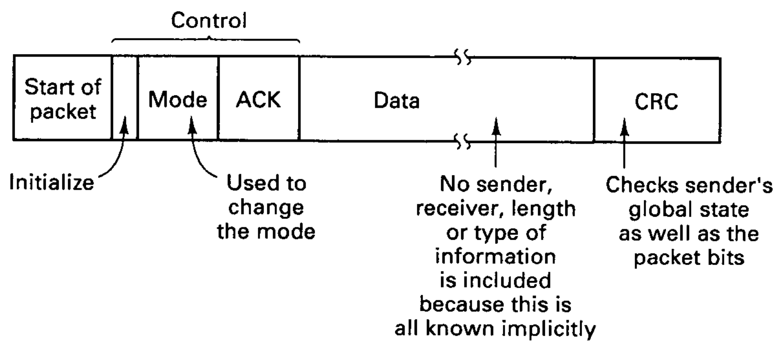
Every node maintains the global state of the system. These states are required to be identical everywhere. It is a serious (and detectable) error if someone is out of step with the rest. The global state consists of three components:

1. The current mode.
2. The global time.
3. A bit map giving the current system membership.

The mode is defined by the application and has to do with which phase the system is in. For example, in a space application, the countdown, launch, flight, and landing might all be separate modes. Each mode has its own set of processes and the order in which they run, list of participating nodes, TDMA slot assignments, message names and formats, and legal successor modes.

The second field in the global state is the global time. Its granularity is application defined, but in any event must be coarse enough that all nodes agree on it. The third field keeps track of which nodes are up and which are down.

Unlike the OSI and Internet protocol suites, the TTP protocol consists of a single layer that handles end-to-end data transport, clock synchronization, and membership management. A typical packet format is illustrated in Fig. 4-29. It consists of a start-of-packet field, a control field, a data field, and a CRC field.



Control
Start of packet
Mode
ACK
Data
CRC
Initialize
Used to change the mode
No sender, receiver, length or type of information is included because this is all known implicitly
Checks sender's global state as well as the packet bits

Fig. 4-29. A typical TTP packet.

The control field contains a bit used to initialize the system (more about which later), a subfield for changing the current mode, and a subfield for acknowledging the packets sent by the preceding node (according to the current membership list). The purpose of this field is to let the previous node know that it is functioning correctly and its packets are getting onto the network as they should be. If an expected acknowledgement is lacking, all nodes mark the expected sender as down and expunge it from the membership bit maps in their current state. The rejected node is expected to go along with being excommunicated without protest.

The data field contains whatever data are required. The CRC field is quite unusual, as it provides a checksum over not only the packet contents, but over the sender's global state as well. This means that if a sender has an incorrect global state, the CRC of any packets it sends will not agree with the values the receivers compute using their states. The next sender will not acknowledge the packet, and all nodes, including the one with the bad state, mark it as down in their membership bit maps.

Periodically, a packet with the initialization bit is broadcast. This packet also contains the current global state. Any node that is marked as not being a member, but which is supposed to be a member in this mode, can now join as a passive member. If a node is supposed to be a member, it has a TDMA slot assigned, so there is no problem of when to respond (in its own TDMA slot). Once its packet has been acknowledged, all the other nodes mark it as being active (operational) again.

A final interesting aspect of the protocol is the way it handles clock synchronization. Because each node knows the time when TDMA frames start and the position of its slot within the frame, it knows exactly when to begin its packet. This scheme avoids collisions. However, it also contains valuable timing information. If a packet begins *n* microseconds before or after it is supposed to, each other node can detect this tardiness and use it as an estimate of the skew between its clock and the sender's clock. By monitoring the starting position of every packet, a node might learn, for example, that every other node appears to be starting its transmissions 10 microseconds too late. In this case it can reasonably conclude that its own clock is actually 10 microseconds fast and make the necessary correction. By keeping a running average of the earliness or lateness of all other packets, each node can adjust its clock continuously to keep it in sync with the others without running any special clock management protocol.

In summary, the unusual properties of TTP are the detection of lost packets by the receivers, not the senders, the automatic membership protocol, the CRC on the packet plus global state, and the way that clock synchronization is done.

4.6.4. Real-Time Scheduling

Real-time systems are frequently programmed as a collection of short tasks (processes or threads), each with a well-defined function and a well-bounded execution time. The response to a given stimulus may require multiple tasks to be run, generally with constraints on their execution order. In addition, a decision has to be made about which tasks to run on which processors. In this section we will deal with some of the issues concerning task scheduling in real-time systems.

Real-time scheduling algorithms can be characterized by the following parameters:

1. Hard real time versus soft real time.
2. Preemptive versus nonpreemptive scheduling.
3. Dynamic versus static.
4. Centralized versus decentralized.

Hard real-time algorithms must guarantee that all deadlines are met. Soft real-time algorithms can live with a best efforts approach. The most important case is hard real time.

Preemptive scheduling allows a task to be suspended temporarily when a higher-priority task arrives, resuming it later when no higher-priority tasks are available to run. Nonpreemptive scheduling runs each task to completion. Once a task is started, it continues to hold its processor until it is done. Both kinds of scheduling strategies are used.

Dynamic algorithms make their scheduling decisions during execution. When an event is detected, a dynamic preemptive algorithm decides on the spot whether to run the (first) task associated with the event or to continue running the current task. A dynamic nonpreemptive algorithm just notes that another task is runnable. When the current task finishes, a choice among the now-ready tasks is made.

With static algorithms, in contrast, the scheduling decisions, whether preemptive or not, are made in advance, before execution. When an event occurs, the runtime scheduler just looks in a table to see what to do.

Finally, scheduling can be centralized, with one machine collecting all the information and making all the decisions, or it can be decentralized, with each processor making its own decisions. In the centralized case, the assignment of tasks to processors can be made at the same time. In the decentralized case, assigning tasks to processors is distinct from deciding which of the tasks assigned to a given processor to run next.

A key question that all real-time system designers face is whether or not it is even possible to meet all the constraints. If a system has one processor and it gets 60 interrupts/sec, each requiring 50 msec of work, the designers have a Big Problem on their hands.

Suppose that a periodic real-time distributed system has m tasks and N processors to run them on. Let C_i be the CPU time needed by task i, and let P_i be its period, that is, the time between consecutive interrupts. To be feasible, the utilization of the system, μ, must be related to N by the equation

$$\mu = \sum_{i=1}^{m} \frac{C_i}{P_i} \leq N$$

For example, if a task is started every 20 msec and runs for 10 msec each time, it uses up 0.5 CPUs. Five such tasks would need three CPUs to do the job. A set of tasks that meets the foregoing requirement is said to be **schedulable**. Note that the equation above really gives a lower bound on the number of CPUs needed, since it ignores task switching time, message transport, and other sources of overhead, and assumes that optimal scheduling is possible.

In the following two sections we will look at dynamic and static scheduling, respectively, of sets of periodic tasks. For additional information, see (Ramamritham et al., 1990; and Schwan and Zhou, 1992).

Dynamic Scheduling

Let us look first at a few of the better-known dynamic scheduling algorithms—algorithms that decide during program execution which task to run next. The classic algorithm is the **rate monotonic algorithm** (Liu and Layland, 1973). It was designed for preemptively scheduling periodic tasks with no ordering or mutual exclusion constraints on a single processor. It works like this. In advance, each task is assigned a priority equal to its execution frequency. For example, a task run every 20 msec is assigned priority 50 and a task run every 100 msec is assigned priority 10. At run time, the scheduler always selects the highest priority task to run, preempting the current task if need be. Liu and Layland proved that this algorithm is optimal. They also proved that any set of tasks meeting the utilization condition

$$\mu = \sum_{i=1}^{m} \frac{C_i}{P_i} \leq m(2^{1/m} - 1)$$

is schedulable using the rate monotonic algorithm. The right-hand side converges to ln 2 (about 0.693) as $m \to \infty$. In practice, this limit is overly pessimistic; a set of tasks with μ as high as 0.88 can usually be scheduled.

A second popular preemptive dynamic algorithm is **earliest deadline first**. Whenever an event is detected, the scheduler adds it to the list of waiting tasks. This list is always keep sorted by deadline, closest deadline first. (For a periodic task, the deadline is the next occurrence.) The scheduler then just chooses the first task on the list, the one closest to its deadline. Like the rate monotonic algorithm, it produces optimal results, even for task sets with $\mu = 1$.

A third preemptive dynamic algorithm first computes for each task the amount of time it has to spare, called the **laxity** (slack). For a task that must finish in 200 msec but has another 150 msec to run, the laxity is 50 msec. This algorithm, called **least laxity**, chooses the task with the least laxity, that is, the one with the least breathing room.

None of the algorithms above are provably optimal in a distributed system, but they can be used as heuristics. Also, none of them takes into account order or mutex constraints, even on a uniprocessor, which makes them less useful in practice than they are in theory. Consequently, many practical systems use static scheduling when enough information is available. Not only can static algorithms take side constraints into account, but they have very low overhead at run time.

Static Scheduling

Static scheduling is done before the system starts operating. The input consists of a list of all the tasks and the times that each must run. The goal is to find an assignment of tasks to processors and for each processor, a static schedule giving the order in which the tasks are to be run. In theory, the scheduling algorithm can run an exhaustive search to find the optimal solution, but the search time is exponential in the number of tasks (Ullman, 1976), so heuristics of the type described above are generally used. Rather than simply give some additional heuristics here, we will go into one example in detail, to show the interplay of scheduling and communication in a real-time distributed system with nonpreemptive static scheduling (Kopetz et al., 1989).

Let us assume that every time a certain event is detected, task 1 is started on processor *A*, as shown in Fig. 4-30. This task, in turn, starts up additional tasks, both locally and on a second processor, *B*. For simplicity, we will assume that the assignment of tasks to processors is dictated by external considerations (which task needs access to which I/O device) and is not a parameter here. All tasks take 1 unit of CPU time.

Task 1 starts up tasks 2 and 3 on its machine, as well as task 7 on processor *B*. Each of these three tasks starts up another task, and so on, as illustrated. The arrows indicate messages being sent between tasks. In this simple example, it is perhaps easiest to think of $X \longrightarrow Y$ as meaning that Y cannot start until a message from X has arrived. Some tasks, such as 8, require two messages before they may start. The cycle is completed when task 10 has run and generated the expected response to the initial stimulus.

After task 1 has completed, tasks 2 and 3 are both runnable. The scheduler has a choice of which one to schedule next. Suppose that it decides to schedule task 2 next. It then has a choice between tasks 3 and 4 as the successor to task 2. If it chooses task 3, it then has a choice between tasks 4 and 5 next. However, if

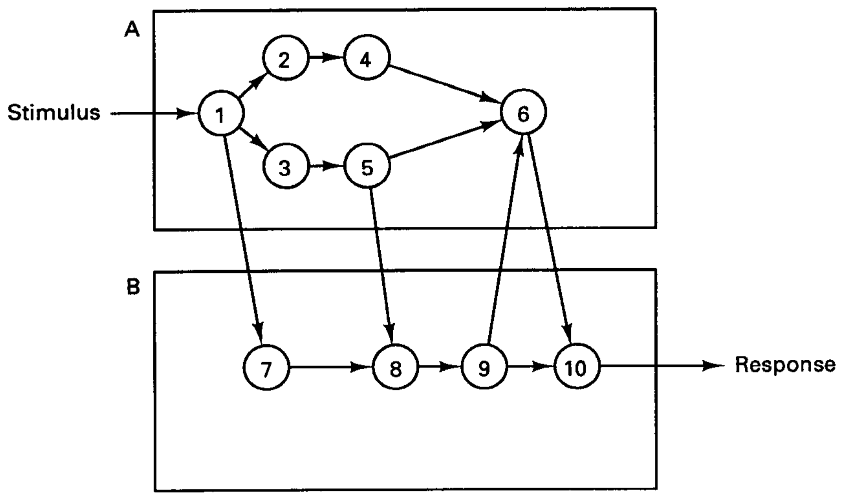


Fig. 4-30. Ten real-time tasks to be executed on two processors.

it chooses 4 instead of 3, it must run 3 following 4, because 6 is not enabled yet, and will not be until both 5 and 9 have run.

Meanwhile, activity is also occurring in parallel on processor *B*. As soon as task 1 has initiated it, task 7 can start on *B*, at the same time as either 2 or 3. When both 5 and 7 have finished, task 8 can be started, and so on. Note that task 6 requires input from 4, 5, and 9 to start, and produces output for 10.

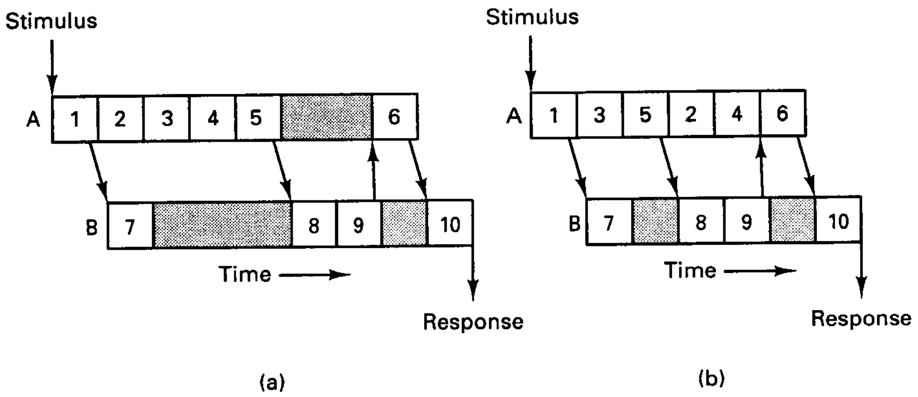


Fig. 4-31. Two possible schedules for the tasks of Fig. 4-30.

Two potential schedules are given in Fig. 4-31(a) and (b). Messages between tasks on different processors are depicted as arrows here; messages between tasks on the same machine are handled internally and are not shown. Of the two schedules illustrated, the one in Fig. 4-31(b) is a better choice because it allows task 5 to run early, thus making it possible for task 8 to start earlier. If task 5 is delayed significantly, as in Fig. 4-31(a), then tasks 8 and 9 are delayed, which also means that 6 and eventually 10 are delayed, too.

It is important to realize that with static scheduling, the decision of whether to use one of these schedules, or one of several alternatives is made by the scheduler *in advance*, before the system starts running. It analyzes the graph of Fig. 4-30, also using as input information about the running times of all the tasks, and then applies some heuristic to find a good schedule. Once a schedule has been selected, the choices made are incorporated into tables so that at run time a simple dispatcher can carry out the schedule with little overhead.

Now let us consider the problem of scheduling the same tasks again, but this time taking communication into account. We will use TDMA communication, with eight slots per TDMA frame. In this example, a TDMA slot is equal to one-fourth of a task execution time. We will arbitrarily assign slot 1 to processor *A* and slot 5 to processor *B*. The assignment of TDMA slots to processors is up to the static scheduler and may differ between program phases.

In Fig. 4-32 we show both schedules of Fig. 4-31, but now taking the use of TDMA slots into account. A task may not send a message until a slot owned by its processor comes up. Thus, task 5 may not send to task 8 until the first slot of the next TDMA frame occurs in rotation, requiring a delay in starting task 8 in Fig. 4-32(a) that was not present before.

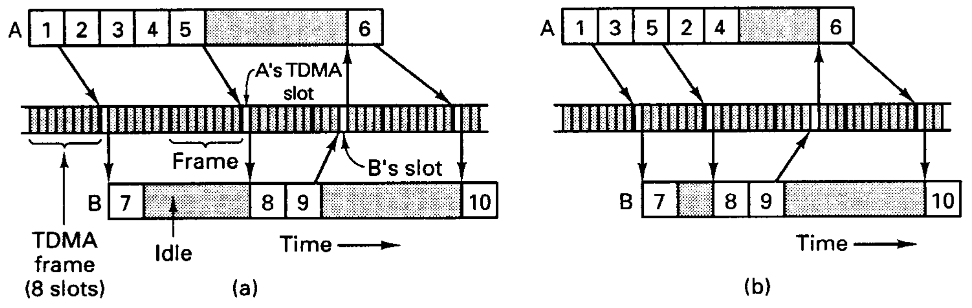


Fig. 4-32. Two schedules, including processing and communication.

The important thing to notice about this example is that the runtime behavior is completely deterministic, and known even before the program starts executing. As long as communication and processor errors do not occur, the system will always meet its real-time deadlines. Processor failures can be masked by having each node consist of two or more CPUs actively tracking each other. Some extra time may have to be statically allocated to each task interval to allow for recovery, if need be. Lost or garbled packets can be handled by having every one sent twice initially, either on disjoint networks or on one network by making the TDMA slots two packets wide.

It should be clear by now that real-time systems do not try to squeeze the last drop of performance out of the hardware, but rather use extra resources to make sure that the real-time constraints are met under all conditions. However, the relatively low use of the communication bandwidth in our example is not typical. It is a consequence of this example using only two processors with modest communication requirements. Practical real-time systems have many processors and extensive communication.

A Comparison of Dynamic versus Static Scheduling

The choice of dynamic or static scheduling is an important one and has far-reaching consequences for the system. Static scheduling is a good fit with a time-triggered design, and dynamic scheduling is a good fit for an event-triggered design. Static scheduling must be carefully planned in advance, with considerable effort going into choosing the various parameters. Dynamic scheduling does not require as much advance work, since scheduling decisions are made on-the-fly, during execution.

Dynamic scheduling can potentially make better use of resources than static scheduling. In the latter, the system must frequently be overdimensioned to have so much capacity that it can handle even the most unlikely cases. However, in a hard real-time system, wasting resources is often the price that must be paid to guarantee that all deadlines will be met.

On the other hand, given enough computing power, an optimal or nearly optimal schedule can be derived in advance for a static system. For an application such as reactor control, it may well be worth investing months of CPU time to find the best schedule. A dynamic system cannot afford the luxury of a complex scheduling calculation during execution, so to be safe, may have to be heavily overdimensioned as well, and even then, there is no guarantee that it will meet its specifications. Instead, extensive testing is required.

As a final thought, it should be pointed out that our discussion has simplified matters considerably. For example, tasks may need access to shared variables, so these have to be reserved in advance. Often there are scheduling constraints, which we have ignored. Finally, some systems do advance planning during run-time, making them hybrids between static and dynamic.

4.7. SUMMARY

Although threads are not an inherent feature of distributed operating systems, most of them have a threads package, so we have studied them in this chapter. A thread is a kind of lightweight process that shares an address space with one or more other threads. Each thread has its own program counter and