12.6 FLOW CONTROL

Flow control is a fundamental paradigm of distributed real-time. In fact, given that bandwidth and computational power are finite, it is impossible to guarantee timeliness if the load flow on the system is not controlled. The role of *real-time flow control* is to regulate the global load flow of the system and to throttle the instantaneous flow of individual source classes in terms of their arrival pattern (rate and amount of data), so as to preserve the capacity of the system to exhibit bounded response times.

The control of a periodic flow is a simple task. A sporadic flow, such as the one produced by sensors of discrete real-time entities, is harder to treat. In Section 12.1 we have discussed the possibility of a bursty sporadic arrival pattern being smoothened over a longer interval (*see* Figure 12.5b), as long as this is allowed by the service latency requirements of the arriving requests, and the interval is not greater than the burst period, if one exists. The mechanism that makes this possible is called rate-based flow control, or *rate control*, and helps balance system load without glitches. *Credit control* is a load control scheme based on allocating a certain amount of credit units (for example octets) to sinks (e.g. recipients), per flow of information coming from a source (e.g. sender or a group of senders). When the credit is over, the recipient refuses to accept more information. Credit complements rate control in case of sporadic real-time operation, whenever it is necessary to perform resource reservation for significant amounts of information of bursty nature.

Flow control is of little use if the average load is misadjusted in the first place. As such, there are measures which can be considered of implicit flow control, such as compacting information at the sensor representatives before sending it to the core of the system, or eliminating redundant messages corresponding to a same event (e.g. fire alarm), that otherwise generate event-message showers.

12.7 SCHEDULING

Real-time is not about having a lot of bandwidth and computational power, and even if it were, we would always find ways of exhausting it. Alternatively, we may think we solve the problem by letting the critical task always get the processor when needed. But the processor power may have to be shared by several critical tasks and the problem surfaces again. Even if we think we have enough power to guarantee the timely execution of our critical task(s), chances are we are using it up at the wrong moment. Remember La Fontaine's Fable of the Hare and the Turtle? Figure 12.8 illustrates this famous tale adapted to real-time scheduling[1].

System TURTLE is a slow system, with relative speed s=1, context switch delay c=1, scheduling first the task that must finish earlier (this is called earliest deadline first, and is a sensible real-time scheduling policy that we are going to study). System HARE is a very fast system, with relative speed s=10, almost

[1]We owe this example to Gerard LeLann.

negligible context switch delay c=0.01, scheduling first the task that comes first (this is called first-come-first-served, and is a not so sensible scheduling policy for real-time). The work presented to either system at time t is: two task execution requests arrive at approximately the same time t, task A before task B; task A, execution time Xa=270 relative time units, deadline Da=t+290; task B, execution time Xb=15, deadline Db=t+28.

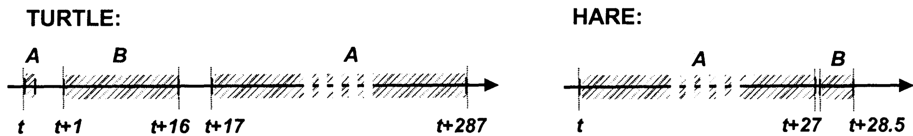

Figure 12.8. The Hare and the Turtle Schedulers

Let us analyze how system TURTLE serves the job. Observing the figure, we see that it starts by releasing A, then switches right after to B, because it has the earliest deadline, B executes during 15 time units, terminating before the deadline. Then, it switches to A again, finishing after 270 time units, before the deadline.

What happens when system HARE serves the job? We see that it starts serving A immediately it arrives, and A will run to completion, according to the first-come-first-served policy. The rationale behind the HARE approach is that the system is so fast that it serves any request "quickly" enough to get ready for the next. However, A executes in 27 time units (speed of 10), the context switches in almost negligible time to B, B executes in 1.5 time units, and ... it misses the deadline by half a time unit!

The lesson to be learned is that real-time is about determinism and guarantees, rather than speed, and in this context the **scheduling** paradigm is concerned with using the available resources in the right way in order to help the system (its programs, its algorithms) achieve timeliness guarantees.

12.7.1 Types of Scheduling

Scheduling assumes several facets, according to the objectives of the system, and to the problems posed, such as the load assumptions. Most non real-time scheduling policies aim at *fairness* and reasonable *performance*, in the access to resources by the several users. Real-time systems, on the other hand, aim at fulfilling timeliness requirements, if need be in detriment of the performance of less important tasks. Let us define some terminology before advancing further. Table 12.1 introduces some generic parameters specifying instants and intervals (that is, events and durations) related with task execution timing. Figure 12.9 depicts a task execution.

The main classes of real-time tasks are: aperiodic, periodic, and sporadic, named according to their main pattern of execution request arrivals (*see Arrival Distributions* in Section 12.1). Aperiodic tasks are impossible to treat deterministically, the best that can be done is service on a best-effort man-

Table 12.1. Task Execution Timing Parameters

Not.	Designation	Description
T_{trg}	trigger instant	arrival instant of event causing the execution
T_{off}	deferral time	delay introduced before execution request **(offset)**
T_{req}	request instant	instant of execution request **(release)**
T_{Rmin}	min. inter-req. time	minimum interval between any two consecutive requests (equals request period T_R, for periodic tasks)
T_{Xmin}	min. termin. time	minimum elapsed time from request to termin. event
T_{Xmax}	max. termin. time	worst-case elapsed time from request to termin. event
T_{WCET}	worst-case exec. time	maximum task duration in continuous execution
T_{lax}	laxity	slack time available for execution ($T_{Xmax} - T_{WCET}$)
T_{live}	earliest term. instant	earliest that task may complete, also called **liveline**
T_{trgt}	typ. termin. instant	*desired* instant of completion **(targetline)**
T_{dead}	latest term. instant	latest that task may complete, also called **deadline**
T_{int}	max. interfer. time	max. time task can be suspended by higher pri. tasks
T_{blk}	max. blocking time	max. time task can be blocked by lower pri. tasks
P	priority	importance of task w.r.t timing (highest is often 0)
U	max. utilization factor	max. percent. of CPU utilization (T_{WCET}/T_{Xmax})

ner. Periodic tasks are the workhorse of static schedule design. Sporadic tasks serve applications that do not have a regular behavior (request arrival is not periodic).

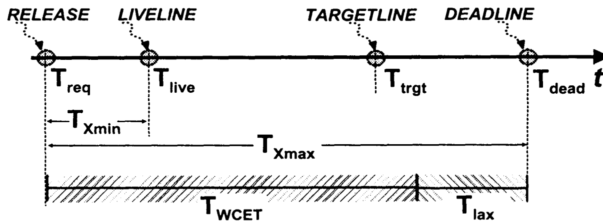

Figure 12.9. Task Execution Timings

We say scheduling is *static*, if all the scheduling plans or scheduling conditions are elaborated beforehand. Static scheduling is also called *off-line* scheduling, as it assumes predicting timing variables such as execution times, request times, resource conflicts, and so forth, and/or assigning static levels of importance to tasks (e.g., fixed priorities). On the other hand, *dynamic* scheduling, also called *on-line*, computes the schedule at run-time, based on the analysis of a list of tasks ready for execution.

Fixed-priority scheduling was very common in earlier real-time operating systems, and is still widely used. With *dynamic-priority* scheduling, priority may change during execution, to reflect the varying importance of tasks, e.g. as deadlines approach.

When the scheduler can interrupt the execution of a task in order to schedule a new one, normally of higher priority, we say scheduling is *preemptive*.

Otherwise, if tasks once scheduled run to completion, we say the scheduler is *non-preemptive*.

Centralized scheduling is performed at a central point, which is normal in single-processor systems. In *distributed* systems scheduling , this point would also become a single point of failure. In consequence, *decentralized* scheduling is preferred for distributed systems.

12.7.2 Schedulability

The usual scheduling scenario is that there is a set of N tasks with several deadlines, to be executed in the processor, over a maximum interval T_{Xmax}, corresponding to the latest deadline. In a periodic task set, the interval corresponds to the least common multiple of the periods. This situation is presented to the designer or design tool at design time for off-line schedulers, or to the scheduler itself for on-line or dynamic schedulers. One has to determine if the schedule is *feasible*. This action is termed **schedulability testing** and is an important step of scheduling. There are three classes of schedulability tests:

- **sufficient**– passing it indicates that the task set is schedulable
- **necessary**– failing it indicates that the task set is not schedulable
- **exact**– passing indicates schedulability; failing indicates non-schedulability

We say a scheduler is *optimal*, when it always finds a feasible schedule if one exists. Sufficient or necessary schedulability tests have an error margin but are simpler than exact ones, and sometimes the only reasonable solution. Obvious such tests are checking that $T_{Xmax} - T_{WCET} \geq 0$, or $T_{Rmin} - T_{Xmax} \geq 0$.

The simplest tests are *utilization-based* schedulability tests, which fail if the schedule will be using the CPU more than a certain percentage. Consider a set of N periodic tasks, each with an individual utilization factor of T_{WCET}/T_R: the schedulability test expression for this set is $\sum_{i=1}^{N}(T_{WCET}/T_R) \leq U_{max}$, where U_{max} depends on the algorithm being used (ultimately, the CPU cannot be used more than 100%!).

Utilization-based schedulability tests are go-no-go sufficient tests. An alternative approach is the *response-time-based* test, explained as follows. The individual WCET of each task is computed. Then, it is used to derive the actual worst-case termination (or response) time, by adding to the WCET the total time the task must yield to all other tasks (e.g. higher priority ones) on account of the scheduling policy, i.e., the *interference* time (T_{int}). The process is repeated for each task. The schedulability test finalizes by simply comparing the computed with the desired maximum termination times. The response-time-based analysis has the advantage of being an exact test, and of giving a quantitative output.

12.7.3 Static Scheduling

Static or off-line scheduling has to take into account worst-case execution times, and urgency, resources, causal precedence, synchronization and deadline requirements of all tasks involved. The schedule is computed by determining

the exact points in time where each task or code module should be launched to meet its deadline. Schedulability testing here means: can the schedule be constructed? Once constructed, the schedule is executed repeatedly, i.e., in a periodic way.

With static schedules, a way of improving schedulability is by using *mode changes*. A mode is a well-contained phase of the system operation (e.g., take-off, cruise, landing in a plane) such that only the necessary tasks, resources and respective requirements are considered for scheduling.

A widely known algorithm for static scheduling of independent periodic tasks in a single processor is **rate-monotonic** (Liu and Layland, 1973). It is a preemptive algorithm based on fixed or static priorities, with the following additional characteristics:

- *optimal when maximum termination time equals period* ($T_{X_{max}} = T_R$)
- *all worst-case execution times* (T_{WCET}) *are known*
- *priorities are assigned in inverse order of the period*

The scheduler, exemplified in Figure 12.10, wakes-up at every start of period, and schedules the highest priority ready task, preempting the running task if necessary. In the example, we have: task T_1, period $T_R = 1$, duration $T_{WCET} = 0.5$; task T_2, $T_R = 5$, $T_{WCET} = 1$; and task T_3, $T_R = 10$, $T_{WCET} = 3$. When all periods are multiples of the smallest, $U_{max} = 100\%$, which is the case of the example, so the schedulability test goes: $\sum_{i=1}^{3}(T_{WCET}/T_R) = 0.5/1 + 1/5 + 3/10 = 1$, which means it is OK.

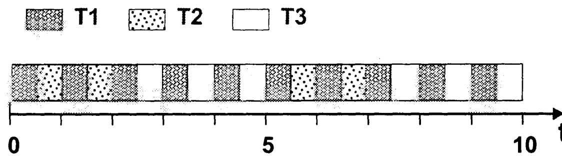

Figure 12.10. Rate Monotonic Scheduler in Action

12.7.4 Scheduling of Sporadic Tasks

Testing schedulability is easy for periodic tasks, since the arrival pattern of the future (period) is known. Attaining and analyzing the schedulability of task sets where periodic tasks coexist with sporadic tasks is a difficult "task" per se, because: (a) we may no longer make decisions completely off-line; (b) it may be hard problem, even when done on-line. Of course, we may consider that a sporadic task is a *pseudo-periodic* task whose period is the minimum inter-arrival time ($T_B \gg T_I$). This has the consequence that most of the service periods are empty, leading to a very low processor utilization.

Another approach is the *sporadic server*, for patterns where $T_{X_{max}} \ll T_I$, that is, single, rare, but very urgent sporadics (i.e., burst length is $N_B = 1$, and $T_B \simeq T_I$). The server task is a periodic task with high priority, scheduled

in competition with all the other tasks of the system. When it runs, it serves any pending sporadic request until exhausting its allocated execution time.

The *dynamic scheduling* approach applies well to task sets containing sporadic tasks. There is an algorithm based on dynamic priorities which is adequate for sporadics, and optimal for periodic tasks. It is called **earliest-deadline-first** (EDF). When the scheduler wakes-up it evaluates the time-to-deadline of every task, and orders their priorities by the inverse of that value: the task that has the earliest deadline receives the highest priority. This algorithm is very elegant and intuitive, and can achieve a maximum utilization of $U_{max} = 100\%$.

Dynamic deadline-oriented algorithms, such as EDF, are very adequate for scheduling of sporadics, since they adapt the priorities of the task set to newly requested sporadic tasks.

12.7.5 Resource Conflicts and Priority Inversion

If two or more tasks compete for resources other than the processor itself, such as a mutual exclusion semaphore or critical section, they become interdependent. This happens frequently in distributed systems, so we give a bit of attention to this problem. In most of these cases the exact schedulability test becomes an NP-complete problem, that is, it exhibits a computationally infeasible complexity. However, the computational power×time concerned with finding a schedule should be much lower than the power×time required to run the tasks themselves. In consequence, on-line scheduling resorts to simpler but inexact tests that sometimes have consequences.

For example, the scheduling of periodic tasks with or without sporadics may suffer what is known as *priority inversion*: a task loses the processor during a non-negligible and non-desirable amount of time, called *blocking time* (T_{blk}), blocked on a resource held by lower priority tasks. Consider the following example of a communications and telemetry system:

- A scheduler provides preemptive scheduling based on fixed priorities. The system has three tasks. Two of them compete for an information channel, a critical resource accessed in mutual exclusion (mutex semaphor).
- The highest priority task is an information dispatcher task, which takes care of dispatching all data to and from the channel, so that it does not overrun.
- The lowest priority task is a meteorological data gathering task, running infrequently with low priority, to publish data on the channel.
- The middle priority task is a communications task, handling system's communications. It does not compete for the information channel.

Now the "scene of the crime", depicted in Figure 12.11a:

- *1. The meteo task (L) acquires the mutex and publishes its information.*
- *2. The dispatcher task (H) runs: H preempts L, tries to acquire the mutex and blocks on it, awaiting for the meteo task to release it. L runs again.*

- *3. However, in the meantime, the communications task (M) requests service: M preempts L, and runs to completion.*
- *Conclusion: H was blocked, first by L then by M through L.*

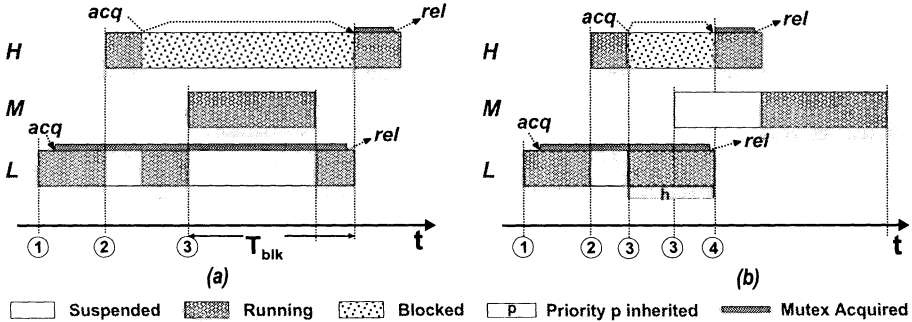

Figure 12.11. Resource Conflicts: (a) Priority Inversion; (b) Priority Inheritance

Is this example realistic? You should have replied yes, because this is what happened with the on-board computer of the NASA Mars Pathfinder probe that landed on that planet in 1997. Shortly after the Sojourner Rover— the small autonomous guided vehicle (AGV) released from the probe— started collecting data, the spacecraft began experiencing total system resets. These were caused by a watchdog mechanism that reacted to absence of activity from the (blocked) dispatcher task, and the system was re-initialized. Fortunately, this recovery strategy was acceptable at that point of the mission, otherwise, it could have been a catastrophe.

What happened was a classical case of priority inversion, a syndrome identified long ago (Lauer and Satterwaite, 1979), which also occurs with networking (Peden and Weaver, 1988). Solutions for it were first addressed in (Cornhill et al., 1987; Sha et al., 1990). In order to remedy the problem, the authors proposed a mechanism which introduces dynamic priority setting to a set of tasks with initial fixed priorities, called **priority inheritance**:

- *the dynamic priority of a process is the maximum of its initial priority and the priorities of any process blocked on account of it*

With priority inheritance, the Mars Pathfinder scene would be scheduled as in Figure 12.11b:

- *1. The meteo task (L) runs with priority l, acquires the mutex and publishes*
- *2. The dispatcher task (H) runs with priority h: H preempts L, tries to acquire the mutex and blocks on it, awaiting for the meteo task L, which runs again inheriting H's priority, h.*
- *3. The communications task (M), with priority m, becomes ready for execution: M waits for L, since h (current pri. of L) is higher than m.*
- *4. L finishes and releases the mutex unblocking H, which grabs the processor ($h > m$), acquires the mutex, and runs to completion. Then, M runs.*
- *Conclusion: H had the minimum blocking possible: waiting for L to finish.*

12.7.6 Scheduling in Distributed Systems

Some single processor algorithms behave well in distributed systems, but others are inadequate or have to be enhanced for distributed operation. Processors are separated by communication links, which themselves are shared and thus have to be scheduled as well. It is difficult to perform scheduling of these distributed resources. Some approaches have relied on heuristics for the cooperation of the distributed system nodes in finding a schedule (Ramamritham et al., 1989). Holistic approaches to the problem have also been considered, where a global scheduling complying with system-wide constraints is constructed from the timing analysis of each module. This has been attempted namely in the embedded systems area, relying on simplifying assumptions on the communications infrastructure, and assuming a periodic behavior of the system (Tindell et al., 1995; Kopetz et al., 1989a). On the other hand, the best known examples of distributed scheduling are the local area network medium access control algorithms, e.g., FDDI, Token Bus, Token Ring, CAN. Some of them exhibit real-time operation, such as the timed-token protocol used in the Token Bus and FDDI networks, or the priority scheduling based on the frame identifier of CAN.

Scheduling in distributed systems is still a subject of research. However, systems are built every day, and apparently, two main approaches can be taken to scheduling in distributed systems with the current state-of-the-art:

- constructing distributed systems whose hardware works in a lock-step fashion, synchronized with the network subsystem also in lock-step (e.g., TDMA) or bit-synchronized (e.g., CAN), and scheduled in a globally periodic manner (e.g., time-triggered cyclic schedules running over TDMA or CAN-like channels). These systems are normally small-scale hard real-time.
- constructing distributed systems whose hardware choices are dictated by the availability of existing COTS (commercial off-the-shelf components), and whose scale and dynamics are dictated by the problem being solved. These systems have better be designed such that tighter (hard real-time) schedules are ensured inside each microscopic component (e.g., nodes, network), and that the cooperative scheduling of the macroscopic system ensures the prosecution of the system's timeliness requirements with the best possible coverage. These systems are normally medium-scale mission-critical real-time.

12.8 CLOCK SYNCHRONIZATION

Observe Figure 12.12a: in the center (dashed line), it depicts a perfect clock, one that always represents real time. However, it also depicts real hardware clocks that are not perfect, i.e., they deviate from real time by a certain amount each second, called rate of drift. We studied the properties of global clocks made of local hardware clocks (*see Time and Clocks* in Chapter 2) and saw that if nothing is done, individual clocks will drift apart with the passing of time. The reader is referred to that section for all basic definitions concerning clocks.

Observe how clocks deviate from the perfect clock in Figure 12.12a: the outside thick dashed lines represent the bound on the rate of drift, a fundamental assumption for deterministic clock synchronization, since it allows us to predict the maximum deviation after a given interval. The amount of deviation between any clock and the perfect clock (which follows real time) at a given time, e.g. at tick t_k, is given by drawing a horizontal straight line passing through t_k in the clock time axis: the length of the line segment connecting the two clock timelines, measured in the real time axis, is the deviation. The current *accuracy* of the clock set is given by the maximum such deviation. As we see, this difference increases as time passes. If the desired accuracy of the clock set is α, no clock may drift to the outside of the grey band, of width α to either side of the perfect clock timeline, as depicted in Figure 12.12a. However, we see that the slowest clock (C_s) will leave the band from tick t_k on (point A), so it should have been re-synchronized before that.

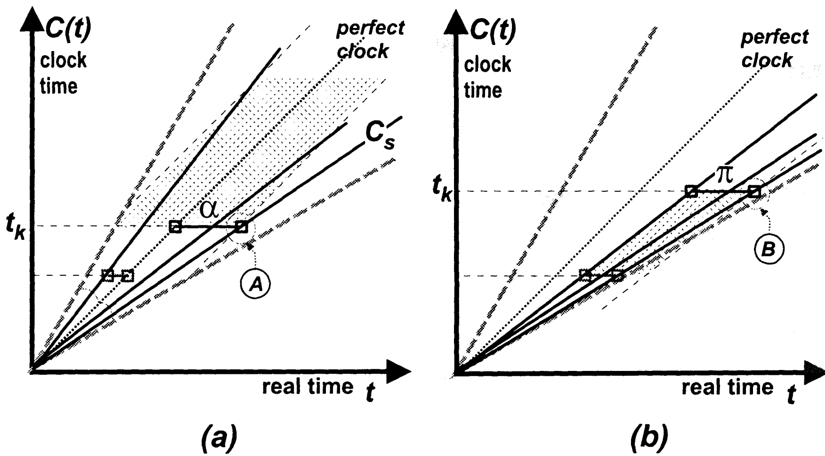

Figure 12.12. Behavior of a Clock with Time: (a) Accuracy Drift; (b) Precision Drift

Figure 12.12b represents the relative deviation among a set of clocks. The amount of relative deviation between the same tick t_k at any two clocks is found by drawing a horizontal straight line passing through t_k in the clock time axis: the length of the line segment connecting the clock timelines, measured in the real time axis, is the deviation. The current *precision* of the clock set is given by the deviation between the two outmost clocks. This difference increases as time passes as well. However, note that their absolute deviation from real time is irrelevant for determining precision: in the example, the set of clocks deviated considerably from the perfect clock, nevertheless they kept within the desired precision π until tick t_k, where the slowest clock got out of the π envelope (point B), as depicted in Figure 12.12b. Obviously, they should have been re-synchronized before that.

The process of maintaining the properties of Precision, Rate, Envelope Rate and Accuracy of a clock set is called *clock (re)synchronization*. Precision is se-

cured by **internal** synchronization, whereas **external** synchronization secures both accuracy and precision, since by securing accuracy, it guarantees that precision remains within $\pi = 2\alpha$. Internal clock synchronization is normally based on convergence functions, whereby the several clocks attempt to converge to a same value at the end of the re-synchronization run. External synchronization is normally based on having all local clocks periodically read from and adjust to one or more clocks containing an absolute time reference. Figure 12.13 exemplifies the latter: clocks are brought together and never deviate more than α from the perfect clock. With either internal or external clock synchronization, it often important to maintain the clock set within the envelope of drift from real time (the thick dashed lines in Figure 12.13). This has to do with securing the Envelope Rate property. The Rate property will be discussed ahead.

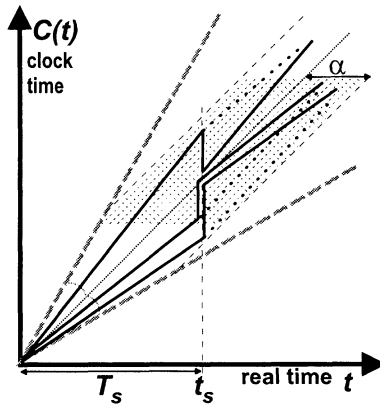

Figure 12.13. Clock Synchronization

Clock-Reading Error The precision achieved immediately after the synchronization, which we have called *Convergence*, δ_v, is a very important property of clock synchronization algorithms, since the quality of an algorithm is measured, amongst other things, by how close it brings the clocks together. Unfortunately, δ_v cannot be made arbitrarily small, and this was defined by a fundamental result in clock synchronization (Lundelius and Lynch, 1984b):

Basic Clock Imprecision - Given n clock processors on a network with maximum and minimum message delivery delays T_{Dmax} and T_{Dmin}, the convergence of any synchronization function is $\delta_v \geq (T_{Dmax} - T_{Dmin})(1 - 1/n)$

The number of processors n is normally large enough that a good working bound for δ_v is $\Delta\Gamma = T_{Dmax} - T_{Dmin}$. The intuition behind this result is that in order to synchronize their clocks, processes need to exchange messages. Unfortunately, the variance in message-passing delays introduces a remote **clock-reading error**, that is, we never know whether the clock value we have just received concerns $t_{now} - T_{Dmax}$ or $t_{now} - T_{Dmin}$.

A second order effect on the quality of synchronization that we neglected in the expressions above is that after reading and while the synchronization is in course, clocks continue to drift apart at a rate ρ_p. If the synchronization algorithm duration is Γ_s, then the additional error in precision caused by this phenomenon is $2\rho_p\Gamma_s$ (observed between by the fastest and slowest hardware clocks). This can normally be neglected. However, we advise the architect to always double check this term before deciding to do it.

Reducing the Clock Reading Error We cannot contradict the Basic Clock Imprecision result, but we can create conditions for achieving a reduced variance in communication delays *during* the critical steps of the execution of the algorithm. In essence, trying to get to a $\Delta\Gamma' = T'_{Dmax} - T'_{Dmin} \ll \Delta\Gamma$, such that $\delta_v \geq \Delta\Gamma'$ (instead of $\delta_v \geq \Delta\Gamma$). To ways of achieving this rely on: trying synchronization enough times until obtaining a run where the variance of the messages involved is small (Cristian, 1989; Mills, 1991); canceling the message delay terms that exhibit greater variance, either algorithmically (Halpern and Suzuki, 1991; Drummond and Babaoğlu, 1993; Veríssimo and Rodrigues, 1992), or through special hardware support (Kopetz and Ochsenreiter, 1987).

12.8.1 Clock Synchronization in Action

A clock synchronization algorithm has the following tasks:

- generating a periodic resynchronization event
- providing each correct process with a value to adjust the virtual clocks

The time interval between successive synchronizations is called the **resynchronization interval**, denoted T_s. At the end of synchronization, clocks are adjusted so that they become separated by at most δ_v. For the sake of convenience, the clock adjustment is usually modeled by the start of a new virtual clock upon each resynchronization event. It can be applied instantaneously, see the instant t_s in Figure 12.13, where the clocks are brought suddenly nearer, and then start drifting again in the next interval. However, this neatly violates the Rate property. One clock is even brought backwards, which is unacceptable. So the alternative is to spread the adjustment over a time interval, by simulating a clock with a slightly different rate, as exemplified by the dotted timelines of the clocks after t_s: the fast clocks become slower, the slow clocks become faster, so that they converge. This is called **amortization** and it is the way synchronization is usually applied: instead of changing the clock, a rate correction factor is applied in software when it is read. For a desired precision π, the value of the resynchronization interval T_s can be extracted from the expression $\pi = \delta_v + 2\rho_p T_s$. Adjusting clocks by changing their values is called state synchronization. It can be combined with another method, *rate synchronization*, which consists of adjusting the rate at which the hardware clock ticks, and even adjusting the instant (phase) of the ticks. These methods are employed when ultra accurate synchronization is desired.

Is clock synchronization a difficult task? The answer is yes, but. Designing clock synchronization algorithms in the presence of communication delay variance and faults is a complex task. However, once deployed, the mission of the architect is selecting the adequate protocol, and simply using time services supported by those algorithms. However, she should understand the limitations of the use of time and timestamps that have been discussed earlier.

12.8.2 Internal Synchronization

Internal clock synchronization algorithms are normally cooperative, where each process reads the values of every other process, and applies a *convergence function* to the set of remote readings. At the end, there is agreement on the adjustment. In what follows, we will use 'clock' and 'processor' interchangeably. Known internal clock synchronization algorithms are of the **agreement** class, and fall essentially into one of three types:

- averaging (AVG)
- non-averaging (NAV)
- hybrid averaging-non-averaging (ANA)

Averaging Clock Synchronization The principle of averaging algorithms, of which there are many examples (Lamport and Melliar-Smith, 1985; Lundelius and Lynch, 1984a) is shown in Figure 12.14a. Clocks start a synchronization run when a number of them reach the end of the period, or resynchronization interval, before the current precision π^{i-1} exceeds the allowed worst-case precision. Each clock disseminates its value to all others. The values received form a *clock-readings vector*, used as input to a convergence function, which computes the value to be applied to the new virtual clock launched in the next period. Assuming that clocks receive the same vector, the same value is applied at the end of this process to each clock, and the initial convergence or precision enhancement of the next period, measured by δ^i in the figure, is dictated by the jitter with which this action is performed.

Both forming the clock-readings vector and reaching agreement on the new clock value can be done in a local manner, or may involve a minimum number of interactions among a minimum number of clock processors. This depends on the fault assumptions of clocks, processors and network, with regard to type (e.g., crash, omissions, value) and number (e.g., how many clocks may fail). The convergence function itself also depends on these fault assumptions, and may be based on known functions such as the *fault-tolerant average*, or the *fault-tolerant midpoint* as exemplified in the figure, which consists of selecting the middle value in the ordered clock-readings vector (*see* also *Resilience* in Chapter 7). As a final note, the precision enhancement of this class of algorithms is directly affected by: the variance of communication delays; the variance in the execution duration of the algorithm.

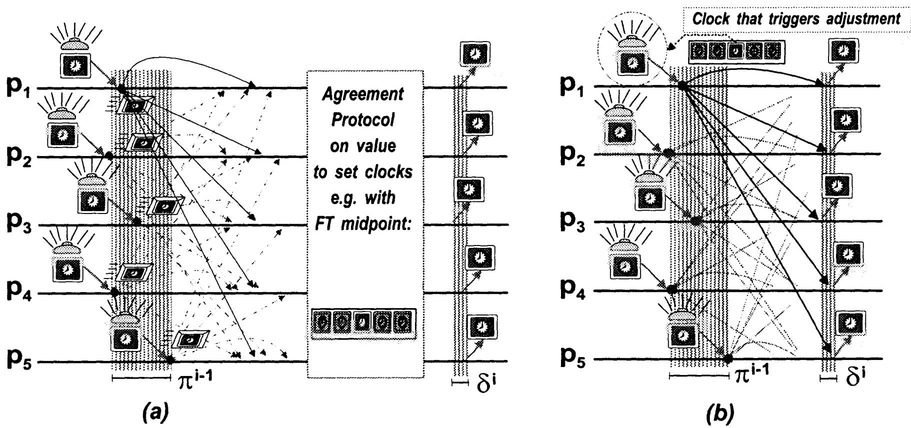

Figure 12.14. Clock Synchronization Algorithms: (a) Averaging; (b) Non-Averaging

Non-Averaging Clock Synchronization Non-averaging algorithms rely on a different principle: instead of disseminating the value of the clock, each clock processor disseminates a control message to signal the end of a period. Figure 12.14b illustrates the principle. The difference is that in the averaging class, the message just sends the clock value as input to a function that performs agreement on the value to adjust the clock with, at some later time. In the non-averaging class, the message does not carry a value, but itself positions an event in the timeline meaning that 'it is the start of period i now', e.g. 5:00, on the sender's clock. All the other processors will do the same. Since some clocks may fail and for example start too early, processors wait for the kth clock to signal the event, in order to get sufficient evidence that it is 5:00. The number k is dictated by the failure assumptions (we leave it to the reader to understand why the example in the figure is resilient to $f = 1$ failures, and thence $k = 2f + 1 = 3$, p_1's clock). After seeing this k^{th} time marker message, processors simply adjust their clocks to 5:00, instead of agreeing on a value, as in the averaging case. Precision enhancement, measured by δ^i in the figure, depends on the difference between the instants when the several processors adjust their clocks, dictated by the magnitude and variance of the delivery delay of the time marker messages and on the assumed faults. Examples of protocols relying on this principle are (Halpern et al., 1984; Srikanth, 1987; Drummond and Babaoğlu, 1993).

Averaging-Non-Averaging Clock Synchronization Hybrid averaging-non-averaging (ANA) clock synchronization algorithms combine the advantages of averaging and non-averaging classes. The principle, depicted in Figure 12.15, consists of reaching agreement on two issues: (i) the clock that triggers the adjustment (the NAV type of agreement); (ii) the adjustment value to load the clocks with (the AVG type of agreement). The algorithm starts by having each clock processor disseminate a message both to stand as time marker of the end

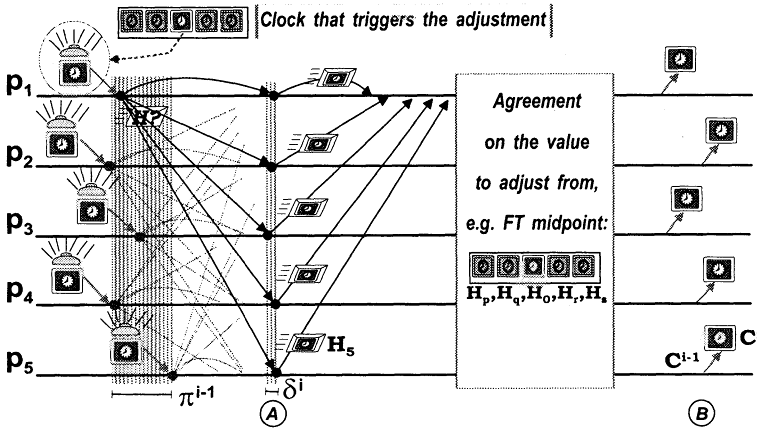

Figure 12.15. Hybrid Clock Synchronization Algorithms (clock p adjustment – $\Delta_p = H_0 - H_p$; new period clock value – $C^i = C^{i-1} + \Delta_p$)

of period $i - 1$, and to act as a remote clock-reading request message to all clocks ($H?$). Each message marks a point in all processors' timelines, as in non-averaging algorithms, and at the kth message, all agree that this is a valid synchronization point. The figure exemplifies p_1 as triggering the adjustment. Clock readings may be sent to the requester, as in the figure, or broadcast to all. This depends on how the agreement on the adjustment value is performed. Suppose an FT midpoint function is applied to the clock-readings vector, and thus H_0 is selected as the new clock value. Instead of setting the clocks with this value, adjustments to each individual clock p are determined by computing $\Delta_p = H_0 - H_p$. Note that the clock-readings vector contains the value of all clocks when the time marker message $\langle H? \rangle$ was received. In consequence, when adjusting the value of each clock for the new period, $C^i = C^{i-1} + \Delta_p$, we are referring the adjustment to the time marker reception instants. The precision enhancement δ^i, is indeed determined at that moment, as signalled in the figure (A), and thus independent of the moment when the adjustment is applied at each clock (B), and largely insensitive to the rate of drift of the hardware clocks during the agreement interval, which may be neglected for most situations. Examples of protocols relying on this principle are (Veríssimo and Rodrigues, 1992; Clegg and Marzullo, 1996).

12.8.3 External Synchronization

External clock synchronization aims at injecting the time of an external reference, the *master* clock, into all *slave* clocks of the system. In that sense, clocks synchronize themselves individually from that reference, rather than agreeing among each other. They either must trust the master, or find fault-tolerant configurations where several masters can be consulted. Known external clock synchronization algorithms are of the **master-slave** class, and the simplest

type relies on *dissemination* of time by the master. This can be achieved through radio broadcasting (spread-spectrum like in GPS, or long-wave beacons), and is for example the method used to synchronize GPS receiver units. However, it is seldom convenient, and sometimes not even possible, that all nodes of a distributed system have a radio receiver.

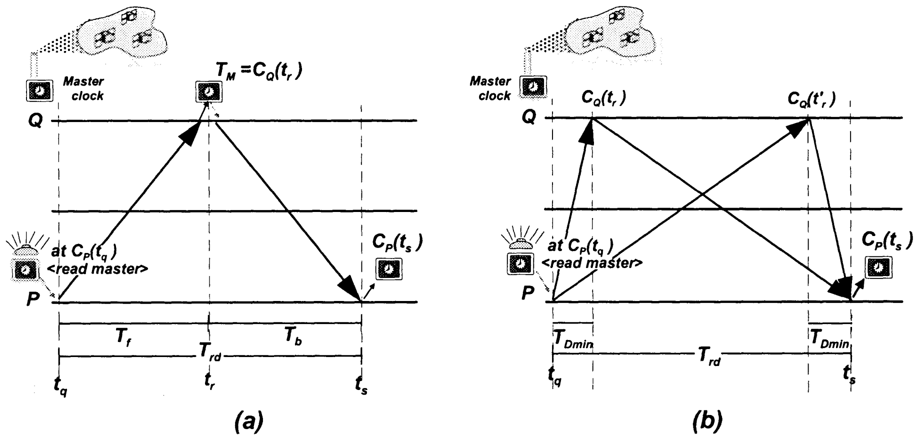

Figure 12.16. Round-Trip External Clock Sync.: (a) Zero-Error; (b) Measuring the Error

Round-Trip Clock Synchronization Most known external clock synchronization algorithms are of the round-trip type, whereby they take the initiative of performing a remote read of the master, receiving its time back and adjusting their own clocks at that time. Figure 12.16 illustrates the technique for estimating the master clock time value at t_s. This involves knowing t_r. Short of knowing T_f or T_b, we estimate t_r to be at the midpoint, or half the round-trip time T_{rd}. Then, $C_P(t_s) = C_Q(t_r) + T_{rd}/2$. Here we can see that only a symmetric round-trip yields zero error. However, the only run that P can detect as symmetric is a minimum delay run, where the request and reply messages have T_{Dmin} duration. P can find this out if it measures $T_{rd} = 2T_{Dmin}$. As a matter of fact, P can do better, it can determine a bound on the error of any remote clock read, given by: $\epsilon = \pm(T_{rd}/2 - T_{Dmin})$. The scenario is depicted in Figure 12.16b. Cristian proposed a well-known *probabilistic* clock synchronization protocol, in which, given a target accuracy, the slave makes several request-reply attempts trying to get a fast enough round-trip that yields the desired reading error (Cristian, 1989). The protocol is probabilistic, because the chances of success depend on the distribution of the network delay. This protocol inspired the Network Time Protocol (NTP) that we discuss in Chapter 14 (Mills, 1991).

12.9 INPUT/OUTPUT

Input/Output is concerned with all that crosses the computational border of a system. In real-time systems, this means talking about the perception or **observation** of, and the **actuation** on, the environment. These are performed, respectively, by **sensors** and **actuators**, devices through which the control system (the real-time computer system) captures and modifies the state of the controlled system (the physical process system). Besides the functional aspects of I/O, there is a general concern in industrial systems with making I/O reliable. A typical example of that concern is a 'trusted' valve used in critical fluid control, a quad compound represented in Figure 12.17. Distribution is as desirable for I/O as it is for computation, either for fault-tolerance or because of the often distributed nature of processes. In this section, we discuss the input/output paradigm under the functionality, distribution and fault-tolerance viewpoints.

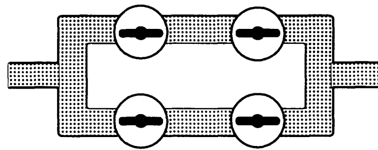

Figure 12.17. A Classical Mechanical Quad Valve

12.9.1 Observation

Observation is: *the act of acquiring and eventually pre-processing the state of a real-time entity, through one or more sensors.* Observations can be made through several techniques, generally falling into: sampling; polling; latching; interrupt.

We use *sampling* when it is possible to decide when to make an observation. Sampling is normally used to observe continuous entities, which exhibit more or less predictable variation curves. The frequency of observation is determined by known control rules. A basic rule-of-thumb is that it should never be less than twice the frequency of the highest harmonic of the entity's waveform. When entities exhibit sudden changes, information may be lost. With *polling* we can observe the entity in infinite loop. This should only be done during a short period where some significant changes are expected (this is analogous to the spinlock in programming), since it ties the processor up. It can be done normally if the sensor has a dedicated micro-controller (what are called intelligent sensors). Sensing can also be performed by *latching* significant changes. This is very appropriate for discontinuous entities, such as, but not only, digital values. Latches are memory elements that register one or more state changes. They can be combined with other techniques, such as sampling or interrupts. It is convenient to use *interrupts* when entities are sporadic: their state changes sel-

dom and at undetermined times. The sensor interrupts the computer system, and thus triggers the observation at the appropriate moment.

There are a set of *pre-processing* functions that improve the quality of observations. Amongst them: *low-pass filters* cancel glitches (e.g., switch debouncing); *likelihood* or *plausibility* tests filter out implausible deviations (e.g., abnormal air temperature readings); *correction curves* linearize sensor outputs (e.g., temperature transducers); *composite variables* are computed from several samples or sources (e.g., rates; averages; approximate convergence functions using sensor replicas or sensors of different physical magnitudes).

12.9.2 Sensor Types

Sensors measure a number of physical magnitudes, through several principles. A few examples of magnitudes: motion; speed; acceleration; pressure; temperature; humidity. Digital sensors are generally on-off: object passing; door open-shut; end-of-range. Some are sophisticated: pattern; color; shape. Several principles are used: magnetic field; photoelectric; hall effect; piezoelectric; image; light (visible, UV, infrared).

When there are not special reliability concerns, the sensor implementation is *simplex single-access*, that is, a single chain of elements, from the real-time entity through the sensor to the computing element. However, if the computing element fails, the sensor is lost. When the process is critical, use of some redundancy may be considered, in order to achieve reliability and availability. Figure 12.18a depicts an actively replicated sensor set, feeding a set of computing elements. This *fully-redundant* sensor configuration is characterized by having each sensor connected to a different representative, desirably in a different node. The computing elements receive the same set of values from the replicated representatives and perform consensus on the final value to be used.

12.9.3 Actuation

Actuation is: *the act of issuing and eventually post-processing a command to change the state of a real-time entity, through one or more actuators.* An actuation can be triggered in several ways: immediate; deferred; periodic.

Immediate actuation is the normal actuation as soon as issued. *Deferred* actuation is invoked to be issued after a specified delay, or at a specified time. *Periodic* actuation is invoked to be repeated at every T. The last two assume some processing capability by the actuator.

There are a set of **post-processing** functions that improve the quality of actuations. The output of actuators can be corrected, through *correction curves*. A command may be issued in its final form to a set of replicated actuators. In this case, *replica control* functions make sure that every replica receives the adequate stimulus. For example, making sure that the individual actuators of the quad valve of Figure 12.17 receive the adequate commands.

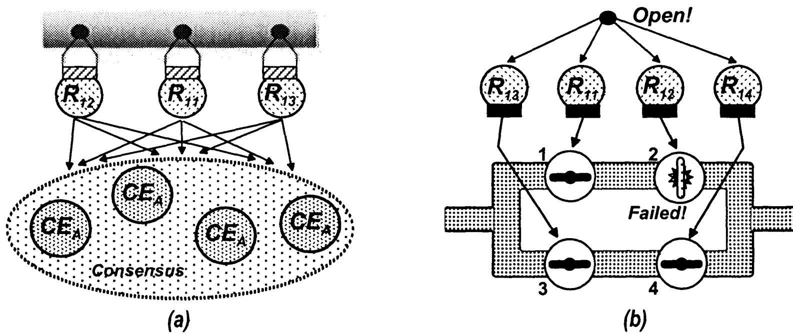

Figure 12.18. (a) A Reliable Sensor; (b) A Reliable Actuator

12.9.4 Actuator Types

Actuators are of different types, depending on the target of actuation. We may be talking of such different things as closing a switch, raising a digital boolean value, opening a valve, or driving a stepping motor. Actuators act on physical magnitudes of the environment, by means of an electric command. Examples of actuator driving principles are: electro-mechanical; electro-pneumatic; electro-hydraulic; electronic.

Similar to sensors, the simplest form is the *simplex single-drive* actuator. It is used when actuators have outstanding mechanical reliability, or when the loss of the actuator is not crucial for system operation. Otherwise, this is an unreliable combination. The *fully-redundant* actuator is the most dependable configuration. In the example shown in Figure 12.18b, there are 4 actuators, one computing element per actuator, residing in different nodes. In this particular example, the configuration controlling the quad valve ensures that an open or a close command always work, in the presence of arbitrary behavior of at most one controller-valve pair. We leave it to the reader to confirm this assertion.

12.9.5 Distributed and Fault-Tolerant I/O

Distributed and fault-tolerant input-output uses some of the techniques for communication and replication management presented in previous parts of this book, and the requirements and properties discussed in those contexts apply to I/O as well. Synchronization of inputs and outputs is most important. That was left clear in Section 12.4. Firstly, we have the problem of positioning the time of the observations and actuations at different nodes in the timeline. Secondly, we have the problem of replication. In multi-access or replicated inputs, the meaning of two samples of the same real-time entity separated by an even small interval may be very different. This is also valid for multi-drive or replicated outputs, for analogous reasons.

The *synchrony* of distributed observation and actuation can influence the quality and even the correctness of I/O operations on real-time (RTe) enti-

ties. *Steadiness* measures the jitter of distributed observations and actuations. *Tightness* measures the degree of simultaneity of sampling, and the allowed skew of replicated actuation commands.

12.10 SUMMARY AND FURTHER READING

This chapter addressed the main paradigms concerning real-time. We began by introducing ways of specifying timing constraints and of detecting their violation. Then, the relation between the real-time entity and its computational representative was defined, namely the duality between time and value. Real-time communication, flow control, scheduling, clock synchronization, and input-output, complete the set of paradigms studied in this chapter.

For further study on failure detection in distributed systems, a formal treatment is introduced in (Chandra and Toueg, 1996) for time-free systems. Timing failure detection is addressed in (Veríssimo and Raynal, 2000). On the formal embodiments of temporal specifications and synchronization, communication by time, temporal order, real-time causal message delivery, and time lattices, please read (Lamport, 1984; Kopetz, 1992; Suri et al., 1994; Veríssimo, 1996).

Response-time-based schedulability analysis is detailed in (Sha et al., 1990), or (Burns and Wellings, 1996; Audsley, 1993). In (Leung and J., 1982) the deadline-monotonic algorithm is described, optimal for maximum termination times smaller than the period. Scheduling of sporadics is further treated in (Mok, 1983; Sprunt et al., 1989; Jeffay et al., 1991). Priority inheritance blocking is a syndrome avoided by priority ceiling and immediate ceiling protocols, thoroughly described in (Cornhill et al., 1987; Sha et al., 1990; Burns and Wellings, 1996). Further material on scheduling can be found in (Ramamritham, 1996a; Burns and Welling, 1996; Audsley, 1993; Buttazzo, 1997) and (Ramamritham et al., 1989; Fohler, 1995; Tindell, 1994; Mossé et al., 1994) namely for distributed settings.

Formal aspects of clock synchronization are surveyed in (Schneider, 1987). The special issue published in (RTS Journal, 1997) gives a good account of current approaches to clock synchronization in open systems. Examples of a hybrid class of clock synchronization algorithms combining internal and external clock synchronization are featured in (Veríssimo et al., 1997; Fetzer and Cristian, 1997b). Further notes about the use of GPS in clock synchronization can be found in (Dana, 1996). Clock synchronization on embedded systems and field buses presents unusual problems, addressed in (Ramanathan et al., 1990; Kopetz and Ochsenreiter, 1987; Rodrigues et al., 1998b; Schossmaier et al., 1997). Interval clock synchronization, based on the notion of interval clocks, is discussed in (Marzullo, 1983; Schmid and Schossmaier, 1997).

A generic treatment of sensors and actuators in reactive systems is presented in (Wood, 1991). Paradigms for fault-tolerant sensors are discussed in (Marzullo, 1990; Kopetz and Veríssimo, 1993). The I/O paradigm may be generalized and extrapolated to generic devices (e.g., gateways, management information bases), a viewpoint taken in (Powell, 1991; Wood, 1991).

13 MODELS OF DISTRIBUTED REAL-TIME COMPUTING

In this chapter, we aim at providing a global view of what is timely behavior of a distributed system architecture. The chapter starts by introducing classes of real-time systems with different timeliness guarantees, setting the stage for introducing the several frameworks for structuring real-time systems. Then, it discusses strategies for the several approaches to building an architecture, where the paradigms presented in the last chapter show their usefulness. The main models of real-time systems are then presented. From the viewpoint of timing: partial synchronism; time-triggered; and event-triggered models. From a functional viewpoint: real-time communication; real-time control; real-time and active databases; and quality-of-service.

13.1 CLASSES OF TIMELINESS GUARANTEES

It is difficult, if not impossible, to build a real-time system satisfying all sorts of requirements concerning timeliness. In consequence, we normally devise our models and systems for specific classes of requirements: hard, soft, and mission-critical real-time.

Hard Real-time System - system where any failure to meet *timeliness* requirements may have a high cost associated

The hard real-time class specifies that the system *must always* be timely, in order to avoid costly timing failures (e.g., a slowed down module of a tomato processing unit causes all tomatoes to be smashed against each other; a slow

Exhibit 2026 Page 338

P. Verissimo et al. *Distributed Systems for System Architects*
© Kluwer Academic 2001

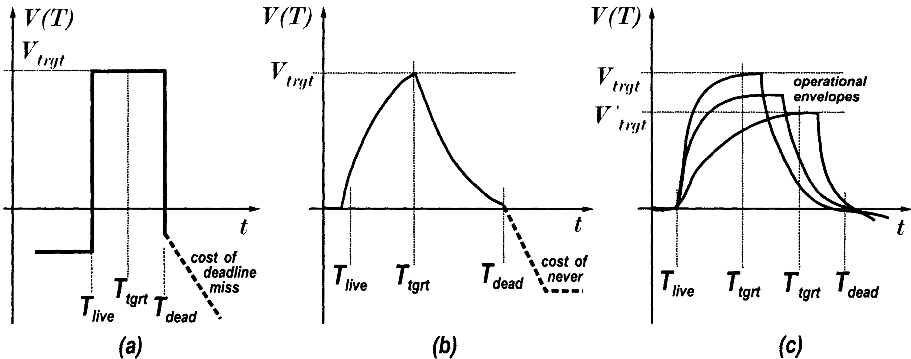

Figure 13.1. Time-Utility Curves for Several Real-time Classes: (a) Hard; (b) Soft; (c) Mission-Critical

robot arm may get stuck under a one-ton press). These systems are also called **time-critical**. Figure 13.1a shows a graphical way of defining criticality, through *time-utility* or *time-value* functions (Jensen and Northcutt, 1990; Burns and Wellings, 1996). These functions portray the value of the service as a function of the time at which it is provided, w.r.t. the time at which it should be provided. We see that not providing the service within the [liveline;deadline] interval carries a cost. The criticality of the system is given by the cost of failure versus the benefit of normal operation, measured by how deep the hatched line goes down (e.g., a stalled engine controller may destroy valves and cylinder pistons; or a late change in rail crossing points can cause a train to de-rail).

Soft Real-time System - system where occasional failure to meet *timeliness* requirements is acceptable

The soft real-time class specifies that the system *should often enough* be timely, that is, it can fail to meet timeliness specifications provided that failures do not occur with too high a probability, or too great a deviation or lateness degree (*see* Section 6.2). This is depicted in Figure 13.1b, where we see the value curve evolving smoothly to and from the targetline as time passes. For example, the specification of a real-time ticket reservation system might look like this: any transaction should terminate within 10 seconds for at least 90% of the times, and within 1 minute in 100% of the times (the latter defines the allowed lateness degree). Whenever there is a cost associated to an extremely long delay or omission of the service, this can be represented, as shown in the hatched line of the figure. Many *interactive systems* are non real-time systems, but the truth is that a fair number of them should indeed have been designed as soft real-time systems, if user requirements were to be respected. How many times did we have to wait in a long bank counter line, only to hear that "It's the computer system's fault"?.

Mission-Critical Real-time System - system where any failure to meet *timeliness* requirements may have a cost associated, and the occasional failure to meet those requirements is considered an exception

The class of mission-critical (also called *best-effort*) systems is designed in order to guarantee that timeliness requirements are systematically met. However, these systems are normally complex and/or the environment behavior is not totally specified, such that timeliness cannot be fully guaranteed. The occasional occurrence of timing faults is tolerated, but should be considered exceptional. If the service risks being provided near or after the original T_{dead} recurrently, because of timing failures, considerable degradation may take place. The system should provide some means for reconfiguration to a less demanding operational envelope where the service, despite having less value (V'_{trgt}), may be provided *predictably later*, thus avoiding timing failures. Figure 13.1c introduces this notion of dynamics in the time-value function of the service. Examples of systems of this class are air traffic control systems, weapons control systems, telecommunications intelligent network architectures.

13.2 REAL-TIME FRAMEWORKS

We spend this section analyzing the main frameworks at the disposal of the architect to build, or build-in, real-time capabilities in distributed systems. This discussion will refer to material presented in the previous chapters, and will introduce some of the models we will discuss later in this chapter. Some are concerned with the synchronism and timing, such as partial synchronism, time-triggered, event-triggered. Others are concerned with the functional models, such as real-time communication, real-time control, and real-time and active databases.

13.2.1 Budgeting

All starts by equating the budget of the time-related variables of the system to be designed. The average load budget derives from the system requirements (number of real-time entities, individual debit of information to and from the system, computation and communication costs). Then, it is essential to make assumptions on the arrival patterns, that is, on how the several load flows are distributed in the time domain: periodic, sporadic, aperiodic. In complement, the timeliness requirements are introduced: when, at what pace, how fast, and with what guarantees, is this load to be processed, and output. We have discussed a few of these issues in Section 12.1. These requirements are introduced by the necessary match between the environment where the system will run, and the type of service desired from it. This budget dictates a first approximation on the power of the computational and networking resources (MIPS, throughput, latency), that can be fine-tuned through subsequent design phases.

13.2.2 Synchronism and Timing

Satisfying the timeliness requirements is concerned with choosing the adequate system model. Whilst the hard-real time class is normally equated with a full-synchrony model (*see Synchronous Models* in Chapter 3), more adequate partial synchrony models have emerged that represent well the soft and mission-critical classes, as we are going to discuss in Section 13.4. On the other hand, choosing time-triggered or event-triggered frameworks implies many subsequent decisions about the system support, with regard to implementing timeliness, as we will see in Sections 13.5, 13.6, and 13.7 (e.g., information flow control, scheduling). As discussed in Section 12.7, periodic scheduling is more easily analyzed and tested, and is the workhorse of time-triggered system models, that we address in Section 13.7. However, when the environment does not comply with periodical operation, the combined scheduling of periodic and sporadic processes is necessary, as accommodated by the event-triggered model, studied in Section 13.6.

13.2.3 Networking

Distributed scheduling is still a subject of research, because of its complexity. For that reason, it is common to separate concerns between network scheduling and local scheduling, at least in larger scale systems. This justifies the importance of real-time networking, as the framework of building and configuring networks and achieving real-time communication. Most networks have their own distributed scheduling policies, on top of which communication protocols can be built. Section 12.5 has introduced the real-time communication paradigm, whose notions will be applied to a real-time communication model discussed in Section 13.8. For certain applications, it may be necessary to extend the bounded delay requirement on reliable multicast message delivery to richer paradigms, such as causal or total order.

13.2.4 Input-Output

Input-output, in complement to processing, is the framework dealing with the boundaries of the system. It is specially concerned with interfacing the environment, whatever architecture it applies to: control, producer-consumer, client-server. Section 12.9 described the main functional principles and techniques of input-output that should guide any implementation. For a matter of separation of concerns, I/O should be seen as a framework separated from the processing activities in the core of the system. In the generic real-time system model that we are going to discuss in Section 13.5, we show that this separation, desirably equated under the entity-representative paradigm presented in Section 12.3, is extremely convenient, for correctness, functional, and implementation reasons. The time-value paradigm (*see* Section 12.4) establishes the correctness conditions for dealing with representations of real-time entities, such as the time of their values, or their value over time.

13.2.5 Programming

Real-time systems programming is a rich and still evolving framework, since it currently combines a wealth of notions, such as: concurrency; robustness; timeliness. Real-time programming is by nature concurrent, and lies on essentially two approaches: language or system support. Language support is granted through language annotations or specialized languages, like Ada (ADA 83, 1983), whose current version is Ada 95 (Bodilsen, 1994), which offer primitives that support concurrency under timeliness constraints. These are enforced by run-time support environments built on top of a real-time kernel. The system support approach uses programming languages not necessarily designed with real-time in mind, which may lack built-in concurrency or timeliness enforcing constructs. These are granted through system libraries supplied as subsystems of a core real-time kernel. This is the track of the so-called *real-time multitasking executives*, so frequently used as a COTS basis to build real-time systems. The virtues of objects for structuring programs are also useful in a real-time context (Forestier et al., 1989). The *real-time object* paradigm serves as an encapsulation element. It can be used to represent real-time entities and computing elements, and provide functional as well as temporal containment, making it easier to reason in terms of the time-domain correctness of complex tasks.

13.2.6 System Support

Last but not least is the question of system support. Besides quantitative issues (e.g. of computational power), the architect must be concerned with qualitative issues. To begin with, there is the balance between specially developed and COTS components. Many critical embedded systems are purposely built. However, the growing trend for using COTS components applies to real-time systems as well, specially when the number of non-critical, complex application-specific systems in the soft real-time and mission-critical classes increases. Under this perspective, the real-time kernel should accommodate the scheduling needs equated during the early design stages, not only the scheduling policy, but also the thread dispatching issues. This reasoning applies to the choice of a standard LAN or field bus as networking support, which must comply not only with the raw data throughput and reliability requirements, but also with the goals for individual frame transmission delay in several urgency classes. This may require additional software-based mechanisms complementing the native priority schemes of the chosen network.

13.3 STRATEGIES FOR REAL-TIME OPERATION

The possible strategies to be followed by the architect in real-time system design are conditioned by several factors, such as: class of operation, price, performance, available technology. The latter present the architect with a number of tradeoffs. The main strategies line-up according to the main target of the system being considered: money-critical, safety-critical, complex large-scale,

quality-of-service. Once agreed on a strategy, the system is conceived along the guidelines suggested by the frameworks for real-time design just presented.

13.3.1 Money-Critical

We designate a system *money-critical*, when the stakes involved in the case of timing failures are high. For example, the control of a process factory, where failures or errors or delays may cause the malfunction or destruction of the process line, entailing considerable loss of product or estate. Having in mind the need for fault tolerance (we have made this statement repeatedly throughout this part), the real-time specific strategies imposed by this target class of systems foresee highly regular designs, and belonging to the hard real-time class of system. Systems should be as simple as possible, even at the cost of versatility, in order to be verifiable. The operation models should be as regular and predictable as possible, which is achievable when the operation of the controlled system can be put within pre-specified boundaries (e.g., a batch process line). This will be further debated in Section 13.9, on real-time control models. Of course, this strategy suggests choices such as: static scheduling; periodic processing and communication; sampling I/O. These are materialized for example in models of the time-triggered type, that we study in Section 13.7.

13.3.2 Safety-Critical

A money-critical system becomes *safety-critical* when the cost involved in case of catastrophic failure is incommensurate to the value of the service in normal conditions. This is related with but not limited to the potential to jeopardize human lives. These systems belong to the area of control (e.g., nuclear power, fly-by-wire, drive-by-wire, etc.). The strategy for building safety-critical real-time systems concerns the use of the most demanding combinations of real-time and fault tolerance techniques, and of structured and formal design and verification methods (Burns and Wellings, 1995; Sinha and Suri, 1999), bringing the discussion of the last section to an even higher level. The main difference to money-critical systems is that safety-critical systems have necessarily to pass a certification process against standards describing several types of requirements. The latter imposes such stringent criteria for all phases of system commissioning, from design to implementation, that it may overshadow decisions based on technical features. In the sense that catastrophic failure of money-critical systems also has a lack of safety aspect, even if in moderate terms, the former are also called *safety-related*.

13.3.3 Embedded

Today's embedded real-time systems have assumed a distributed nature. In the measure that embedded distributed real-time applications start being built on top of virtual-circuit like structures, such as LANs or field buses, the gap between the desired 'distributed systems' model of the applications and the

underlying 'networking' model becomes apparent. One such problem is the effect of the network *jitter* in the application timings. If analyzed superficially, it would seem the problem would disappear with more rigid interconnecting structures, such as physical-circuit or digital-bus ones, such as time-triggered TDMA networks. However, there is more to an architecture than the network. The strategy for modern embedded system design calls for looking at the complete architecture, what we might call a **field-bus distributed system**, where networking, middleware support and applications are seamlessly integrated and thus no gap exists. If an architectural approach to the problem is followed, this can be done in most of the existing field buses. Applications must see an adequate distributed systems support environment in the underlying infrastructure. The latter must be built with building blocks exhibiting adequate properties. Finally, the adequate algorithms must be used to interconnect these blocks, in order to achieve the desirable reliable real-time behavior of the whole.

13.3.4 Complex Large-Scale

Whenever a system is complex and/or has a considerable scale, the enforcement of timeliness requirements may conflict of the lack of assumptions restricting the behavior of the environment— because of the complexity of the process— and with weaknesses of the infrastructure— because of its scale. In this case, one cannot expect to fully predict the evolution of the environment, and as such, the design strategy of the system cannot be turned to static policies, and regular behavior. These systems fall to a great extent into the mission-critical of soft real-time classes. Some will be money-critical in nature, but cannot be designed by the strategies we proposed above. As a compromise, the strategy to design these systems suggests choices such as: timing error detection and recovery; dynamic scheduling; combined scheduling of sporadic and periodic processing and communication; interrupt-driven I/O. These choices are materialized for example in models of the event-triggered type, that we study in Section 13.6. Many architectures of this kind are client-server, which introduces a further ingredient of non-determinism. The real-time database model, addressed in Section 13.10, is a representative design strategy for this kind of systems.

13.3.5 Quality-of-Service

An emerging class of real-time systems is quality-of-service oriented. For example, video-on-demand, voice-over-IP, visualization and other multimedia services, of the soft real-time class. These systems are normally structured around the producer-consumer or client-server architectures. Less important than avoiding timing failures at all cost, is the limitation of their effect and production, through adaptation to uncertain timeliness of the environment (varying operation conditions) and uncertain utilization patterns (varying number of users and of flows). As such, a strategy for this kind of systems should be

based on: dynamic scheduling; combined scheduling of sporadic and aperiodic events; QoS failure detection and adaptation.

13.4 SYNCHRONISM MODELS REVISITED

We have addressed synchronism in Section 2.6. We have learned that it is expressed in terms of timeliness properties, and it underlies any mechanism for securing real-time behavior, since synchronism stipulates that it is possible to bound action delays (processing and network). There is an obvious relation between the *time* and *synchronism* paradigms: if time is expressed in durations and positions in the timeline, synchronism bounds the variance and imprecision with which those durations and positions are established, i.e., the *jitter* as defined in Section 12.1. There is also a relationship between the *synchronism* and *order* paradigms, established by the definition of temporal order made in Section 2.7: if synchronism expresses the steadiness of a protocol, the latter dictates how fine a temporal order the protocol can implement, that is, how small the δ_t*-precedence* potential causality threshold (Veríssimo, 1996).

13.4.1 Timeliness versus Liveness

We saw in Chapter 1 that it is convenient to formally specify what we wish of a system in terms of high-level safety and liveness properties. Safety properties specify that wrong events never take place, whereas liveness properties specify that good events eventually take place. Take the example specification we used then:

P1- any delivered message is delivered to all correct participants

P2- any message sent is delivered to at least one participant

P2 is a liveness property. We know it will happen, but we do not know when. This is not compatible with our expectations about real-time systems, where we decided to use time as artifact to guarantee synchronization with the environment, and all players involved: the user that produces inputs and receives outputs, the sensor that is read, the actuator that produces an output, the processor that has to finish a task, the network that has to deliver a message, the failure detector that detects that something was not done as and when it should. We learned that this is done by defining *timeliness* properties: whatever happens is only correct if done by T, at T, every T, etc. That is, we are introducing a safety condition. Timeliness properties have in fact a safety facet, specified by means of time operators or time-bounded versions of temporal logic operators (Koymans, 1990; Lamport, 1994). In order to turn our example specification into a real-time specification, we would augment it with a timeliness property:

P3- any delivered message is delivered within T_{Dmax} from the time of send request

13.4.2 Partial Synchrony Models

Real-time systems behavior is materialized by *timeliness* specifications, which in essence call for a synchronous system model. However, large-scale, unpredictable and unreliable infrastructures are not adequate environments for synchronous models, since it is difficult to enforce timeliness assumptions. Violation of assumptions causes incorrect system behavior. In alternative, the asynchronous model is a well-studied framework, appropriate for these environments. This status quo leaves us with a problem: fully asynchronous models do not satisfy our needs, because they do not allow timeliness specifications; on the other hand, correct operation under fully synchronous models is very difficult to achieve (if at all possible) in large-scale infrastructures, typical, for example, of mission-critical systems, since they have poor baseline timeliness properties.

What system model to use for applications with timing requirements running on environments with uncertain timeliness? The question probably has more than one answer. Recently, approaches have emerged that tolerate *partial synchrony* of the system while securing timeliness properties, or else detecting their absence, for example, the *timed asynchronous* (Cristian and Fetzer, 1998), or the *quasi-synchronous* (Veríssimo and Almeida, 1995) models. These models rely on the observation of two aspects of real-life environments:

- synchronism is not a homogeneous system property;
- worst-case termination times or delays are much larger than normal ones.

That is, parts of the system or epochs of its operational life can be reliably considered synchronous. As such, bounds on response time and other variables can be defined that hold on a subset of the system, or during limited periods of its operation.

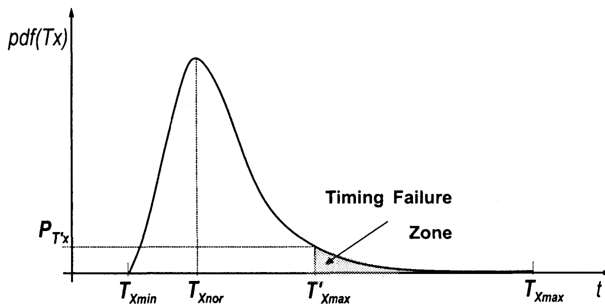

Figure 13.2. Distribution of termination times

Observe Figure 13.2: in real environments, the probability density function of the time it takes for an activity to complete (e.g., message delivery), rather than being a step, has a shape similar to the one represented in the figure. In settings with uncertain timeliness, such as large-scale systems, the worst-case termination time, $T_{X_{max}}$, if it exists, is much higher than the average case ($T_{X_{nor}}$), such that it becomes useless. The assumption of a shorter, artificial

bound T'_{Xmax}, increases the expected responsiveness. In contrast, it increases the probability of timing failures, since T'_{Xmax} has a non-zero probability of not holding $(1 - P_{T'_X})$, yielding a timing failure. However, if timing failures are detected when they occur (*see* Section 12.2), a reliable system can be built which works synchronously when the environment allows, and reacts in order to preserve correctness in the presence of timing failures.

Table 13.1. Partially Synchronous Model Properties

• Some of the system properties— processing or message delivery delays, rate of drift of local clocks, difference between local clocks— have a known bound
• For the others, a known bound may not exist or may be too large

The generic properties of this model are listed in Table 13.1 (*see* also Table 3.2 in Chapter 3). Partially synchronous systems offer support for building mission-critical or soft real-time applications that are dependable, in the sense that they offer resilience to failure of timing assumptions.

13.5 A GENERIC REAL-TIME SYSTEM MODEL

We present a generic real-time system model that will help us understand other, specialized models introduced later in this chapter (Kopetz and Veríssimo, 1993). The model is depicted in Figure 13.3, and it relies on the separation between input-output, communication, and computing.

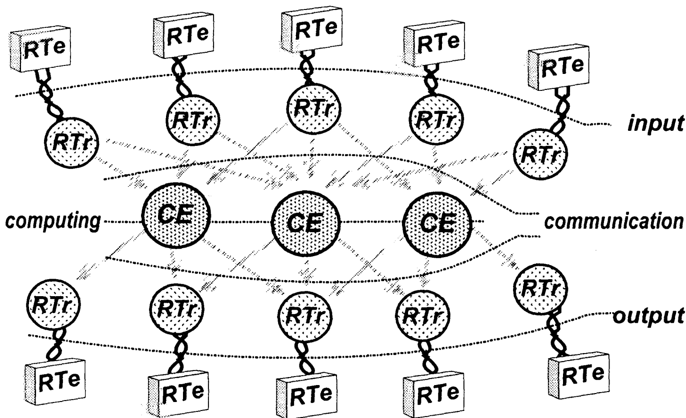

Figure 13.3. Generic Real-Time System Model: RTe- real-time entity; RTr- representative; CE- computing element

Computing is performed by *computing elements* (CE), computational entities which process observations of RT entities (RTe), modify the internal state

of the system, and eventually trigger actuations on other RTe's, to modify the state of the environment. *Input-output* is performed between the RTe's and their *representatives* (RTr) (*see Entities and Representatives* in Chapter 12). Representatives assume a crucial role of containment between the physical reality and the computer world. RTr's are to RTe's what device drivers are to normal computers: they hide the complexity and idiosyncrasy of sensors and actuators, and translate that physical reality into variables and structures, which computers can understand. Once defined the rules and boundaries for the relation between an RTe and its representing RTr, the latter can be assumed to "be the RTe", inside the computing system. *Communication* ensures the timely flow between the computing elements and the input and output representatives, since we are reasoning in terms of distributed systems.

The value of an RTe E at any instant t, $S = E(t)$, is represented *inside the computer* by $S^r = r(E)(t)$. The implications of using S^r instead of S have been studied in Sections 12.3 and 12.4. This separation of concerns simplifies system design. One step is to ensure that the RTr faithfully represents its RTe in all situations including assumed faults in the I/O area. Once this secured, the architect can devote his effort to ensuring that the computing mechanisms, given correct representations of all input and output RTe's in terms of computing abstractions, produce correct results. Last but not least, and given the needs dictated by the above steps, he must see to it that the communication mechanisms ensure that information flows in a timely manner between CE's and RTr's.

Particular models may implement specializations of this generic model. The producer-consumer may be concerned with the observation of real-time data and its timely processing and storage at one time, and its recovery and timely rendering at another, in an open-loop fashion. On the other hand, in real-time control the loop is closed, from observation of the environment, to computation and actuation, and feedback to the observed input, through the environment. This model can be activated in a time-triggered fashion, or in an event-triggered fashion. It is very useful to analyze these apparently contradictory schools under the same generic model, as we do next.

13.6 THE EVENT-TRIGGERED APPROACH

Figure 13.4 illustrates the architecture and information flow of an event-triggered (ET) system. The *information flow* retains an event-like nature up to the center of the system. This approach adapts well to overload and unexpected events, either from the same or different representatives.

Event-triggered system - one that reacts to significant events directly and immediately

13.6.1 Event-Triggered Architecture

In any real-time architecture, it is important to regulate the flow of information from the periphery (representatives) to the core of the system (computing

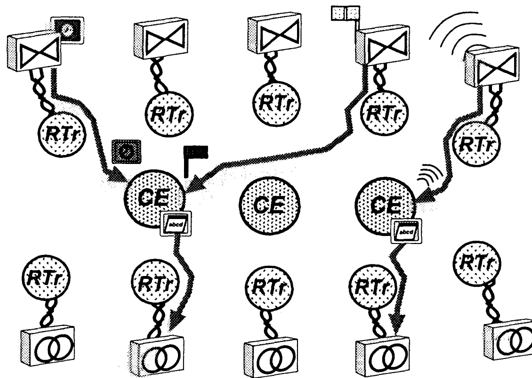

Figure 13.4. Event-triggered Architecture

elements). This is preferably done using policies adapted to real-time, such as *rate-based flow control*, which act at the input representatives, rather than at the computing elements (*see Flow Control* in Chapter 12).

The definition of the operational envelope of ET systems (the set of assumptions about the behavior of the environment to control) is by nature not rigid. A beneficial consequence is that they can admit situations where they are stressed beyond the design-time worst-case workload without falling apart. Average *responsiveness* of ET systems is the best possible in general, since an event gets through and is processed just depending on preemption speed and communications medium access time. Given the bursty nature of most ET traffic, priorities play an important role in letting urgent events get through in a message or event queue.

The dynamics of ET systems is at the same time their weakness and their strength. The irregular event distributions (sporadics) make *predictability* hard to ensure. It is an advantage however, when system complexity and lack of environment knowledge cannot ensure predictability but require versatility, as in mission-critical operation. Given their dynamic characteristics, the scheduling of ET systems is not decided *a priori*: it must be done *on-line*, will be *preemptive* in most cases, and will be prepared to schedule a computation serving a sporadic event of extreme importance, immediately it arrives (*see Scheduling Sporadic Tasks* in Chapter 12).

The ET approach is also valid under the perspective of critical applications. Imagine that such a system is processing critical hard real-time tasks, but sometimes enters overload periods: an ET system can be designed in order to withstand this extra load, exhibiting what is called *graceful degradation.* This prefigures the *mission-critical* class of operation, which ET systems are very apt for. In the impossibility of meeting all timeliness requirements, they will attempt at reducing the cost involved in failing to meet some, for example by selecting those that would, if not met, lead to catastrophic failure.

The extra complexity put into the design principles of ET systems, a disadvantage already pointed out, becomes an advantage once the system is designed and it is necessary to extend it. Because of the provisions to support dynamic operation, this turns out to be easier. The event-based approach allows selective dissemination of information and allocation of resources.

13.6.2 Event-Triggered Protocols

ET systems are specially devoted to treating *sporadic* events, which accurately represent the environment encountered in most real-time problems (*see Temporal Specifications* in Chapter 12). Event-triggered systems, as shown in Figure 13.5, are so to speak 'idle', waiting for something to happen. When an external event or burst of events occurs (e.g., objects passing under a detector beam), it is transformed into an *event message* or message interrupt, which is sent to the interior of the system, where one or several computing elements process it, modify the system state, and eventually produce outputs. Note that messages are processed as they arrive, and so are outputs. This is strict ET behavior.

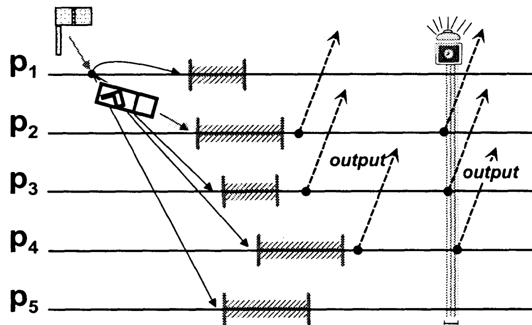

Figure 13.5. Timing Issues in an Event-triggered System

ET systems are said to be *susceptible to event showers*. However, this would be a simplistic way of looking at the problem. Note that our model establishes a barrier at the representatives, which can establish measures for filtering the redundant and spurious information out of the flow to the computing elements, neutralizing the 'event shower' effect. In other words, if e_1 is the leading event carrying information about an alarm situation, the information carried by the subsequent event shower tail contributes little to the information already brought by e_1. The system architect can thus create rules to: *compact* successive instantiations of the same alarm at the representatives; *discard* redundant events, either at the representative or upon arrival at the computing element (by a pre-processor); *reserve* communication and computing resources for the forthcoming shower; *smoothen* the transmission of event bursts to the computing elements, by using rate-based flow control to space them along the interval between bursts, when there is enough laxity.

Another common idea is that ET systems are *unsteady and untight*, or in other words exhibit a high jitter, making them inadequate for more demanding real-time control applications. This derives from the fact that processing and outputs are basically event-driven. However, with the assistance of synchronized clocks it is very easy to superimpose time-triggered behavior on top of an event-triggered communications or processing system. This is exemplified on the far right of Figure 13.5: actuations can be triggered by the clocks, at a steady (periodic) pace, and tightly at all nodes. In fact, timing of an event triggered system may be purely timer-driven or clock-less (*see* Figure 2.10 in Chapter 2), but it may also be clock-driven, without necessarily being periodic (*see* Figure 2.11 in Chapter 2).

13.7 THE TIME-TRIGGERED APPROACH

The architecture and information flow of a time-triggered (TT) system is shown in Figure 13.6. In TT systems, the *information flow* is throttled in the periphery of the system, as shown in the figure, because there is a preliminary transformation from event to state (state of the RTe).

Time-triggered system - one that reacts to significant events at pre-specified instants in time

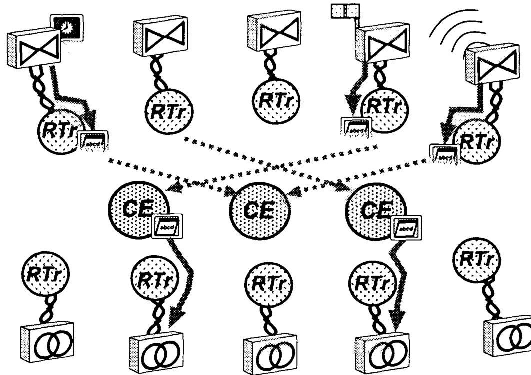

Figure 13.6. Time-triggered Architecture

13.7.1 Time-Triggered Architecture

By assumption, there is no such thing as *overload* in TT architectures. However, one cannot exert 'flow-control' on the environment. This is universally true, that is, the situation is no different from ET systems in this respect. So, *flow control* in TT systems is performed between the environment and the representative, which only accommodates the information it is prepared to recognize and transform in one period. In order that no overflow occurs at the representative, the event arrival distribution must be well known. Otherwise, unexpected but important events may be filtered out. As such, the bounded

demand assumption can be a dangerous one if the environment is not thoroughly described: a requirement of TT system design. Take the (unfortunate but realistic) example of a weapons-control system: a TT system designed for a maximum of 50 incoming enemies will be blind or puzzled when a 51st enemy arrives.

The crux of the TT approach is to create the conditions to make the system appear well-behaved enough that events occur in synchrony with the system clock, and simple enough that only the assumed event distributions occur. This works for a number of applications, namely in continuous process control. Once the I/O problem is solved, from then on the system is quite predictable. If we look at Figure 13.6, we see that events are transformed into state at the representatives. As a matter of fact, it is as if every representative holds a piece of the global state, all of which are disseminated to all computing elements when the start-of-period ticks, in the form of *state messages.* These messages contain structured data, the whole of which forms the global state or system context. In consequence, they are not consumed; instead, each overwrites its previous instantiation in the global state, like a piece in a puzzle. Going back to Figure 13.6, the flow between representatives and computing elements pictured there is periodic and static, and it is always the same amount of information: the objective is the cooperative refreshment of the global state. In consequence, *resource reservation* rather than flow-control is necessary.

TT systems are built as periodic automata, and as such are the perfect match for *periodic* events, mostly generated by artificial processes, but that is the case in a lot of process control settings (discrete or continuous). TT *response* is thus cyclical, occurring at pre-specified instants in time. Responsiveness depends on the system period. Given that the environment is asynchronous with regard to the system, an event may have a waiting time of one cycle in the worst-case, half-cycle on average. So, when a very urgent alarm arrives it may have to wait that long to be served. This is only relevant when the expected service delay for these urgent sporadic events is of the order of magnitude of the system period or shorter.

TT systems have other advantages. Given their cyclical and lock-step (in pulses) evolution they are simpler to test and show correct in the case of, for example, critical applications. Design for *predictability* is easily achieved in TT systems. In consequence, they are excellent for small closed systems controlling static environments and repetitive processes, like some manufacturing cells. Predictability and testability are very important factors of choice when reliability and safety figures have to be very high. In consequence, it is not surprising to see critical problems (nuclear, fly-by-wire or drive-by-wire control, train control, etc.) addressed by TT systems. On the other hand, they may be harder to commission for large-scale or often-varying settings.

Scheduling is calculated *off-line* and it is *static*. The system still has to treat several tasks, so the automaton we mentioned before is a multi-tasking one. However, since we know the evolution of the system *a priori*, we can also determine how long the processing steps last, and combine their interleaving

in order that all tasks perform their work by the end of a period. Once this schedulability exercise is done, the schedule may be cast into the system executive (*see Static Scheduling* in Chapter 12). Obviously, increasing the number of nodes, or adding new tasks will require a total redesign of the node slots, communications, schedule, etc. Mission-critical systems are thus not a field of preference for TT systems, which cannot handle unexpected events, and can only handle sporadic bursts if considered at design time, by means of several predefined *operating modes.* This means as many pre-tested schedules, and still leaves open the problem of falling outside the 'outer' operating mode.

13.7.2 Time-Triggered Protocols

Activity in TT is triggered at environment-independent instants, a shown in Figure 13.7. The period is short enough to match the rate of evolution of the environment and long enough for the duration of processing.

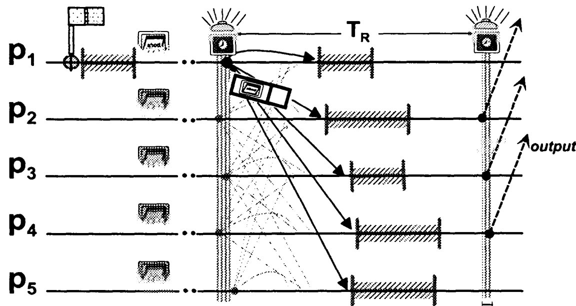

Figure 13.7. Timing Issues in a Time-triggered System

There is no event shower effect in TT systems. To comply with the cyclical operation style, events are collected and pre-processed in the periphery of the system, between periods, as shown in the figure. Note that observations are made by sampling, and pre-processed so that whatever relevant happened since the last period at each representative is included in the partial state information to be sent in *state messages* to the computational part. At given moments, these messages are disseminated to the computing elements, which assemble them in a complete state image of the system. The necessary tasks are then deterministically scheduled at each node, in order to perform the necessary state transformations and eventually schedule results to be output to representatives. The are also output at a pre-determined instant, which coincides with the end of the period. Actuation can be made as steady and tight as allowed by the precision of the synchronized clocks (mandatory in TT systems).

Flow control would also seem to be a non-problem. Note two things however: (1) the system is always working as in full load— be the environment 'quiet', or going through worst-case alarm situations, the information flowing is the same; and (2) the representative must handle any showers of alarm situations, and the maximum rate of event arrivals. The messages transactioned between the latter

and the computing elements must always carry enough octets to represent the maximum I/O information that can be generated in a period in any situation. The efficiency of use of resources is thus compromised in favor of determinacy of operation.

Time-triggered communication is mandatory to support the kind of operation depicted in Figure 13.7 for a TT system. In fact, the whole architecture, including the network, works in pulses. For example, the MARS system, a pure TT system (Kopetz et al., 1989a), accesses the network through a TDMA protocol called TTP, using a slotted access method where each node knows exactly when and for how long to transmit (*see* Figure 2.12 in Chapter 2).

13.8 REAL-TIME COMMUNICATION MODELS

When speaking of real-time networks, there are several fields of interest: the virtual circuit type of network, the realm of medium to large-scale settings; the physical circuit type, the realm of control networks; and the digital bus type, the realm of lock-step systems. These types were presented in Chapter 11, in increasing order of tightness of expectations about real-time behavior. The timescales involved also span an increasingly finer range, from the second to the microsecond. The use of specialized protocols and hardware is more related with the digital-bus end of the spectrum, in contrast with the standard components available for virtual circuit types over open networks.

We are going to develop a real-time communication model based on the definition of the paradigm made in Section 12.5. The steps are generic enough to be useful for design of any of the network types considered. The architect will have to specialize the implementation of the model according to the expectations of a given type. We discuss structure and configuration (load budget, urgency classes, network properties), connectivity enforcement (medium reliability, redundancy measures), and algorithmics (preventing timing faults, tolerating omission faults).

13.8.1 Configuration and Structure

In order to *configure* the network, a few points must be taken into account:

- traffic patterns;
- latency classes;
- network sizing and parameterizing.

The designer must be able to model the *traffic patterns* offered to the network by each individual node, falling in the types defined in Section 12.1: aperiodic, periodic or sporadic traffic. The next step is to perform some traffic separation in *latency classes*, corresponding to the classes of urgency in the system, which translate into a range of successively higher transmission time bounds. As a general rule, urgent data (critical) and protocol control frames should be mapped onto the highest network priority. The other traffic should

be distributed by the remaining priorities, according to their relative urgency and/or importance.

Regardless of the network technology used as the basis for a reliable real-time network, one important point is not to get tied to a particular implementation, i.e., achieve portability. *Abstracting* from physical particularities of a network is thus a good architectural principle. Take a LAN for example. LANs have a set of common properties, some never stated, which can be extremely helpful for the construction and proof of correctness of some distributed systems paradigms. For the sake of example, we describe a set of useful abstract properties of LAN-type networks, including field buses, in Table 13.2.

Table 13.2. Abstract LAN Properties

Broadcast	correct nodes receiving an uncorrupted frame transmission, receive the same frame
Error Detection	correct nodes detect any corruption done by the network in a locally received frame
Network Order	any two frames received at any two correct nodes, are received in the same order at both nodes
Full Duplex	frames transmitted are also received at the sending node
Bounded Omission Degree	in a known time interval T_{rd}, omission failures may occur in at most k transmissions
Tightness	correct nodes receiving an uncorrupted frame transmission, receive it at real time instants that differ, at most, by a known interval T_{tight}
Bounded Transmis. Delay	any frame is transmitted by the network within a bounded delay T_{TXmax} from the send request

The *Broadcast* and *Error Detection* properties impose detection of errors in the value domain, in a broadcast (e.g., the CRC protection mechanism). Frames not passing the test are simply discarded, usually by the MAC VLSI. *Network Order* is the physical order imposed by the mutual exclusion on the communication medium. The *Full Duplex* property is only directly supported by some LAN chipsets, such as the Token-Ring, and the switched Ethernet. It is also supported by the CAN chipset. The *Tightness* property measures the maximum interval between reception instants in different nodes, a function of the variances of the end-to-end propagation delay and the interrupt processing time for a frame. Omission errors normally have a physical origin: mechanical defects in the cable, EMI corruption of a passing frame, modem synchrony loss, receiver overrun, transmitter underrun, etc. As such, it is possible to make probabilistic assumptions about the occurrence of omission errors dur-

ing an arbitrary interval of concern T_{rd}, which boils down to a number k of successive transmissions hit by omission errors, as per the *Bounded Omission Degree* property. The *Bounded Transmission Delay* property specifies a maximum transmission delay, which is T_{TXmax} in the absence of faults. It is not a self-contained property of networks, depending on the particular network, its sizing, parameterizing and loading conditions.

13.8.2 Maintaining Connectivity

Medium reliability is the crucial issue to secure connectivity, and it involves fault tolerance measures in the networking infrastructure (*see Fault-Tolerant Networks* in Chapter 6). Of course, if the whole network is replicated, we have n paths to each destination, and can mask $n-1$ medium failures (*see* the examples of Figure 6.4 in Chapter 6). However, many real-time networks are simplex infrastructures that can also be provided of medium connectivity measures. In essence, we can count on two strategies to maintain connectivity (*see* the examples of Figure 6.5 in the same Chapter): space-redundant medium, where several media are available in parallel; reconfiguring medium, where the medium breaks-up and reconfigures to a new correct state.

13.8.3 Achieving Bounded Transmission Delay

Enforcing a bounded and known transmission time bound T_{TXmax} (*Bounded Transmission Delay*), is not guaranteed per se in a LAN. The load budget and the separation in latency classes allocated to LAN priorities must be equated with network throughput, to achieve figures for the transmission delay numbers. This process is analogous to the discussion of the schedulability tests and maximum termination time done in Section 12.7.

Each class is scheduled according to the available policy of the network, either a dedicated scheduling mechanism, such as in special purpose TDMA networks, or the available fixed priority mechanisms of standard LANs and field buses, which work by guaranteeing some precedence order or a certain amount of the channel bandwidth to fulfill latency requirements. Hybrid mechanisms have been the most successful: building improved schedulers on top of standard LAN or field bus priority mechanisms.

The network should be *parameterized* in order to reply to these requirements. Some LANs, such as the Token Bus, require a setting of several timers in order to correspond to the load budget and latency class definition just discussed (Janetzky and Watson, 1986; Gorur and Weaver, 1988). It may sometimes be discovered at this point that the latency aimed for is not achievable with the amount of load offered. Then, the load has to be adjusted, in an iterative procedure. The process is LAN dependent and should not be neglected for a successful design.

13.8.4 Achieving Bounded Delivery Delay

The *bounded omission degree* assumption introduced by the *Omission Degree* property is paramount for achieving a bounded delivery delay, since it dictates the level of redundancy needed, either time redundancy (number of repetitions), or space redundancy (number of network channels or media). Note that without these reliability assumptions, it is impossible to put a bound on the duration of the action of getting a message through.

In order to pursue the divide-and-conquer strategy, the architect must now consider the implementation of an omission-free network infrastructure, for an omission degree of k, through a choice of fault-tolerance mechanisms. The algorithms must also be aware of the measures taken at hardware configuration time about medium reliability, discussed in the previous section. In Chapter 7 we studied the basic error processing protocols, which point to several alternatives: (1) space redundancy with error masking; (2) time redundancy with error masking; (3) time redundancy with error detection/recovery.

Let us assume that up to k errors may occur. The time budget for alternative 1 is transparent to omissions, since at least one frame arrives on one of the media, in each redundant transmission on all of the $(k + 1)$-plicated networks. The time budget for alternative 2 has a constant overhead of transmitting not one but $(k + 1)$ copies of each frame. The time budget of alternative 3 has to take into account the accumulated durations of the error detection timeouts in the worst-case runs involving $k+1$ retries. Note that if a bounded transmission delay T_{TXmax} is ensured, then by the *Bounded Omission Degree* property either of mechanisms 1-3 achieves message delivery in bounded time despite omission errors, and in absence of partitioning.

13.8.5 Controlling Partitioning

Let a network be partitioned when there are subsets of the nodes such that nodes from different subsets cannot communicate with each other[1]. *Physical partitioning* may occur in a real-time network on account of physical defects: bus medium failure (cable or tap defect), transmitter or receiver defects, etc. On the other hand, there can be *virtual partitioning*, where networks exhibit glitches in their operation, e.g. *congestion* in wide-area networks, or *inaccessibility* in LANs or field buses (Veríssimo, 1993).

We can prevent physical partitioning (alternative (1) in the last section), or we can have a glitch of variable duration with medium-only space redundancy (the solution for alternatives (2) and (3)), or we can reduce but not avoid virtual partitioning. We had rather address all forms of partitioning in the same way, and talk about 'controlling', rather than 'preventing'. In consequence, we say partitioning is acceptable in a real-time network if it is *controlled*: duration is bounded and suitably short for the service requirements.

[1] The subsets may have a single element. When the network is completely down, *all* partitions have a single element, since each node can communicate with no one else.

13.8.6 Implementing Flow Control

Sometimes it is necessary to exert flow control on the communication load. Traditional flow control mechanisms normally used in non real-time communication, such as the sliding-window scheme used in TCP (Comer, 1991), can delay transmissions for long periods or even arbitrarily. Besides, they are not appropriate for multicast communication.

Rate control is a real-time flow control policy implementing a rhythmic operation that is equally suited for periodic and sporadic traffic. It aims at matching the sender's average debit with the recipient's capabilities, without discontinuities in traffic flow. The Xpress Transport Protocol (XTP) is a typical example of protocol using rate control, capable of real-time behavior (XTP, 1998). Credit and rate control may be combined in XTP.

13.9 REAL-TIME CONTROL

Control is historically the main application of real-time systems. As shown in Figure 11.3 in Chapter 11, it is concerned with observing selected real-time entities from a *controlled process*, and computing whatever corrections and actions necessary to maintain the process within the pre-specified operational envelope. The meaning of operational envelope may vary depending on the type of control:

continuous control	applies for example to batch processing, and its correct operational envelope is defined by ensuring that each of a certain number of variables stays within a maximum error from their pre-defined values, the *set-points*
discrete control	applies for example to manufacturing, and its correct operational envelope is defined by ensuring that a pre-defined sequence of actions takes place at the appropriate times, dictated by either a pre-defined static schedule, or in reaction to external events, or both

13.9.1 Architecture of Distributed Control Systems

What is called control loop in computer control is: observing the value of a read-only real-time entity RTe_1 at a given time; computing the necessary action; acting on a write-only RTe_2 in response to the input; physical feedback from RTe_2 through the environment, eventually changing the state of RTe_1, which will be read again, and so forth. A distributed real-time control architecture has the following building blocks:

- input and output representatives
- computing elements
- reliable multicast or broadcast communication subsystem

Representatives are connected to the real sensors and actuators. A representative is a driver or task that handles the respective sensor or actuator. In centralized control, sensors and actuators are normally connected to a single

computing element, or *controller* in this case, and in consequence representatives co-reside with the computing elements. In distributed control, the representatives of input and output real-time entities may reside at different nodes, where they may or not coexist with controllers.

Anyway, it is essential to disseminate the information from sensors to the several controllers, so that they acquire a common knowledge about the image of the system and of any events occurring. This is easily achieved through the synchronous reliable multicast or broadcast protocols that we have studied in Chapter 2, either under the initiative of representatives, or of controllers. Supposing that each controller schedules its tasks correctly in reaction to that information, some controllers will produce outputs to representatives connected to actuators. Here again, reliable multicast protocols may be used in case several actions must take place at different actuators in a synchronized way, for example, in the case of the quad valve studied in Section 12.9.

In *algorithmic* terms, real-time control implies the solution of essentially three problems:

- timely and correct observation of real-time entities of the environment
- timely and correct computation of the action to be executed next
- timely and correct actuation on real-time entities of the environment

The first problem has to do with the way observations are performed. Observations are essentially related with determining a value at a given time or determining the time at which a given value occurs. Considering that the information gathered from the environment is as accurate as needed in the value and time domains, the second problem has three facets: the control algorithms must be logically adequate to the problem; observations must be used correctly over time, since the state of the RTe may vary after being read; scheduling of the necessary tasks must ensure the production of results within the required response times.

The third problem concerns the correct implementation of those results when they imply actuations, and has to do with the ability to position an action at a given point in the timeline (*see* Sections 12.4 and 12.9).

Distribution and *replication* in control introduce additional problems:

- synchronizing and disseminating observations made at different nodes
- splitting and allocating control tasks to different nodes
- synchronizing actuations made at different nodes

Models that allow splitting tasks of a global computation through different nodes have been studied in Chapter 3. One can use global time for triggering a synchronized acquisition or actuation. Clock-driven observations allow synchronizing the acquisition of data from different or replicated sensors. This way one can determine the order of external events, even if acquired in different nodes. Clock-driven actuation serves the purpose of accurately positioning actions on the environment in the timeline, with cooperative or replicated actuators. The steadiness and tightness of the operations influence the results (*see Distributed and Fault-Tolerant I/O* in Section 12.9). In general, the time-

liness properties of control actions influence the quality of control, and even its correctness.

13.9.2 Quality of Control

Logical aspects of the algorithmics notwithstanding, the quality of control is affected by two time-domain factors: response time and jitter. The *response time* affects quality for two reasons: (a) if an RTe varies significantly during a single control cycle (observing/computing/actuating), when the actuation is issued the RTe no longer has the value read, producing an error; (b) if the response time exceeds an absolute bound, for example for discrete actions, a serious failure may occur. Fixed delays can sometimes be compensated for by prediction or extrapolation. The remaining error caused by the accumulated *jitter* in the several phases of the control cycle cannot: jitter of the observation; communication and execution time jitter; positioning jitter of the actuation. We are going to base our analysis of these errors using the time-value paradigm (*see* Section 12.4).

Time of a Value Observe Figure 13.8a, depicting a curve made of sensor observations ($\mathcal{H}_r(E)$) of the value of a real-time entity ($\mathcal{H}(E)$) along time. On the left half we depict the problem of observing of a value at a predetermined time T_{obs}. In the example, we assume a sensor amplitude error bounded by ν_s, and a jitter bounded by ζ_o. Recalling Section 12.4, a *consistent* observation of a value at a given time has a value domain error bounded by a known $\mathcal{V}_o$. In this case, $\mathcal{V}_o = \nu_s + \nu_o$, ν_o being the maximum equivalent value error caused by ζ_o. This situation is depicted in the figure: the real value is E_a at T_{obs}; the reading suffers the error ν_s and yields $E_a^{'}$; finally, the jitter ζ_o positions the reading action too early, yielding $E_a^{''}$ read at $T_{obs} - \zeta_o$, as if it were read at T_{obs} with an additional value error of ν_o. The alternative perspective of determining the time at which a predetermined value V_{obs} occurs may also suffer a maximum error that is shown on the right of Figure 13.8a: the real event is E_b at T_{val}; however, its observation is delayed by ζ_s, to when $\mathcal{H}_r(E)$ reaches the V_{obs} threshold (E_b'), given the error of ν_s; the observation receives a later timestamp, $T_{val} + \zeta_s + \zeta_o$, given the positioning jitter ζ_o. The error in determining the time at which a given value occurs is thus bounded by $\mathcal{T}_o = \zeta_s + \zeta_o$. For discrete entities like booleans, the effect of the sensor threshold error (the equivalent timing error ζ_s) can be neglected in the expression of $\mathcal{T}_o$.

Value over Time We take two control-related examples to illustrate this issue: the bearing of a ship; the angle of the crankshaft of an engine piston. There is a vast body of research on computer control algorithmics that this book does not intend to replace. The derivations discussed here are very simple, and aim at illustrating the generic problem of use of time-value entities in control, and its implications in a computerized and distributed context.

Consider an observation $\langle v_i, t_i \rangle$ of a time-value entity E (for now consider it perfect, that is, $v_i = E(t_i)$, neglecting any observation errors). This observation

will be used for some computation that depends on the value of E, e.g., a slight correction in the helm by the autopilot, or the computation of the ignition point by the engine controller.

When the result is produced, at some later time t_j, a non-negligible amount of time may have elapsed: the time spent in effectively delivering the observation (T_D), plus the execution time of the action (T_X) (see the figure). This *delay* error affects the accuracy of the control action, since it is supposed to concern the value of E at t_j, but was based on its value at t_i. Additionally, the error itself is uncertain, because of *jitter*: both the delivery time and the termination time have a variance, and in consequence, t_j cannot be determined accurately. Let us denote the maximum total error due to the execution of the computation (observation delivery and adjustment computation) as $\mathcal{T}_x$.

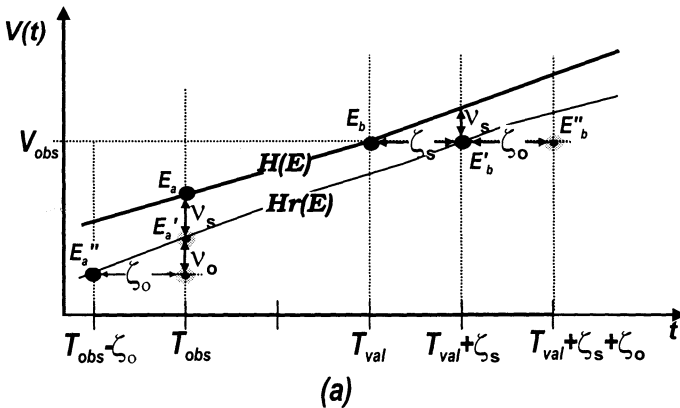

Figure 13.8. Time-Value Entities: (a) Time of a Value

If $E(t)$ is not predictable, either because it cannot be represented analytically (e.g., discrete, or too complex), or because the cost of computing the prediction would be too high, the original observation value must be used in the computation of the control adjustment. Then, the total timing error is bounded by the maximum response time of the system, $\mathcal{T}_x = T_{Dmax} + T_{Xmax}$. However, $E(t)$ is often a predictable function with a small *extrapolation* error for some near future time. This happens with most continuous variables, and opens the way to several optimizations that we do not discuss in detail. In general terms, it consists of computing a prediction $\langle v_k, t_k \rangle$ of the value of E for $t_k > t_i$ and using it instead of $\langle v_i, t_i \rangle$ as input for the computation. This adjustment cancels part of the $\mathcal{T}_x$ error. In order to capture the intuition, note the simplest of the approximations: to cancel the error due to the fixed part of the delays. As a first shot, consider making $t_k = t_i + T_{Dmin} + T_{Xmin}$. The result will be produced at an instant t_j anywhere between t_k and $t_k + \zeta_d + \zeta_t$, respectively the variances of the delivery time and the termination time. In consequence, the effect of the error is reduced to the jitter terms. Consider further adding the average jitter, e.g., $t_k = t_i + T_{Dmin} + \zeta_d/2 + T_{Xmin} + \zeta_t/2$:

this reduces the effect of the jitter to half, i.e. the total timing error becomes $\mathcal{T}_x = \pm(\zeta_d + \zeta_t)/2$. Since $t_k - \mathcal{T}_x \leq t_j \leq t_k + \mathcal{T}_x$, if we compute the adjustment for t_k, then neglecting the extrapolation error, the value error $\mathcal{V}_x$ will be bounded by the maximum deviation during $\mathcal{T}_x$.

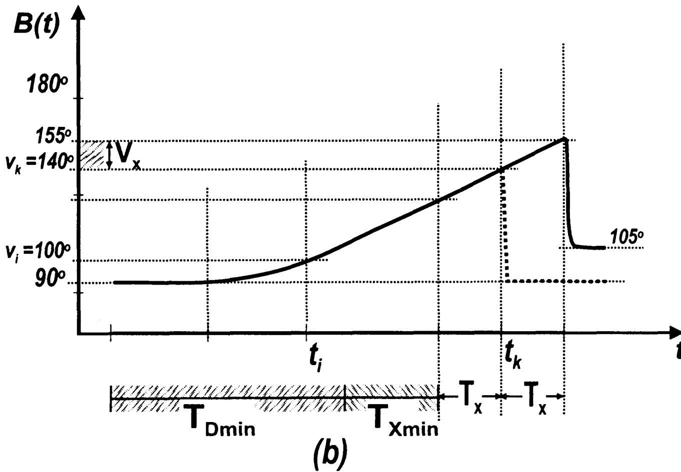

Figure 13.8 *(continued).* Time-Value Entities: (b) Value over Time

Coming back to our examples, we illustrate the first in Figure 13.8b (bearing of a ship), and leave it to the reader to elaborate on the second. Given the continuous nature of the real-time entity `bearing` ($B(t)$), timing errors translate into value errors in the helm correction. Observe the figure: the ship had a steady bearing of 90° (East) but starts to drift. The current bearing is read at t_i to be 100°, but the adjustment is computed using an extrapolation to time t_k using average delays. If the execution finishes at t_k, then the bearing correction is accurate. However, the termination event may take place at t_j anywhere between $t_k \pm \mathcal{T}_x$. In the extremes, the maximum error in bearing correction is attained ($\pm\mathcal{V}_x$): the autopilot applies a correction to make the ship come back from a bearing of 140° to 90°; however, for the situation shown in the figure, the ship will under-correct to 105°, because it is already aiming at 155° when the correction is applied ($t_k + \mathcal{T}_x$), and vice-versa for the other extreme of the interval.

If the response time gets to be of such magnitude that some boundary condition may be crossed before termination, more serious problems than mere control inaccuracy may occur. An excessive response time compared with the rate of drift will make the ship successively under-correct and over-correct, following an unstable route. Likewise, if the computation of the ignition point finishes after the crankshaft reaches the earliest possible ignition point, the ignition control may miss the ignition point, and even damage the engine.

To conclude, we now introduce the observation-related errors discussed in the previous section. The maximum timing error of the observation itself is

$\mathcal{T}_o = \zeta_s + \zeta_o$, with a corresponding value domain error bound of $\mathcal{V}_o$. If the observation was read from a representative, or a real-time database, it concerns a past state of the RTe. Recalling Section 12.4, the correctness condition is *temporal consistency*: an observation stored at a representative must not be older than a stipulated bound $\mathcal{T}_a$, called the absolute validity interval. This interval introduces a value domain error bounded by a known $\mathcal{V}_a$. The total timing error of the control loop in our example, from the instant of observation to the instant of actuation, is thus bounded by:

$$\mathcal{T}_{ctl} = \mathcal{T}_o + \mathcal{T}_a + \mathcal{T}_x$$

This bound on accumulated timing errors translates, for continuous variables, into a predictable equivalent value error bound:

$$\mathcal{V}_{ctl} = \mathcal{V}_o + \mathcal{V}_a + \mathcal{V}_x$$

Wrapping-up, quality of control implies securing the following, for all RTe's:

- the value stored at the representative must be consistent – this implies bounding the observation error by a known $\mathcal{T}_o$, and must be secured by the I/O instrumentation, which must be good enough;
- the value stored at the representative must be temporally consistent at all times, since it gets obsolete with time – this implies bounding the absolute validity interval of the representative value to a known $\mathcal{T}_a$, and must be secured by the I/O interface, which must refresh often enough;
- a reading from the representative must remain temporally consistent until used – this implies bounding the response time of the action where the reading is used, such that the associated timing error $\mathcal{T}_x$ is bounded, and must be handled inside the communications and computational parts. The error term $\mathcal{T}_x$ may be further reduced by using prediction functions;
- when computations concern several RTe's, the concerned observations must also be mutually consistent, i.e., the timestamps of all observations should fall within a known interval $\mathcal{T}_m$ (*see* Section 12.4).

13.9.3 Distributed Control

The ability to order related input events and to position related output events in the timeline is crucial for distributed real-time control, since now sensors and actuators are connected to several controllers with a network in the middle.

Consider the problem of analyzing a stream of events of a failure situation in an electrical power distribution network. Measurements are made by distributed representatives, which send them to the controllers. The causality relations between events will dictate the correct reconfiguration procedure. However, if two events that are causally related are inversely ordered, this will disturb the recovery procedure, and in some cases may worsen the problem.

Likewise, consider the problem of distributed control of the semaphores of a road crossing. Actuation of the several lights must be synchronized: light must go green on one side just after the light went red on the other side; pedestrian lights must change accordingly. The lack of tightness in positioning each related set of actuation events can cause incorrect behavior, for example, causing both lights to be green during some time. As for replicated actuators, they should

be exercised at approximately the same time, otherwise either a voting fails, or it looks like more than one actuation was made.

Distributed Observation Distributed observations receive timestamps from the local clocks. The meaning of the timestamps versus the effective separation of the observations depends, as we know, on the granularity (g) and the precision (π) of the clocks (*see Time and Clocks* in Chapter 2). The two relevant problems introduced by distribution are (Veríssimo, 1994): (a) what is the minimum separation of observation events that can be ordered by a system of global timestamps; (b) what is the minimum difference between timestamps of two distributed observations so that their mutual order can be determined.

Observe that these questions are extremely important, since up to now, we have been able to derive correctness conditions for control considering a conceptual timestamping facility common to all entities. Only if we know how to relate timestamps produced by different clocks at different sites, can we apply these derivations to distributed control. In consequence, the system *must* have a clock subsystem with good enough precision and granularity to address the application requirements in determining: δ_t-precedence— the minimum interval to generate causal relations (*see Temporal Order* in Chapter 2); jitter— the error in positioning distributed events in the timeline or in measuring distributed duration variance (*see Timing of Events* in Chapter 12).

Consider that the virtual granularity of the clocks is made $g \geq \pi$, that is, coarser than their physical granularity. This *granularity condition* (Kopetz, 1992) ensures that distributed timestamps of the same event are at most one tick apart, which looks like a sensible measure. With this approximation, one can draw interesting conclusions about distributed observations (Veríssimo, 1994), listed in Table 13.3.

Table 13.3. Separation, Timestamping and Ordering of Events

• any two events separated by at least $2g$ are correctly ordered • any two events separated by at least g but less than $2g$ are never inversely ordered, but may receive the same timestamp • any two events separated by less than g may receive the same timestamp or be arbitrarily ordered with consecutive timestamps • the same event observed at two sites may receive the same timestamp or be arbitrarily ordered with consecutive timestamps
• events with the same timestamp are always concurrent (not $2g$-precedent) • only events with timestamps separated by at least 2 are guaranteed to be in their physical order • only events with timestamps separated by at least 4 are guaranteed to be $2g$-precedent

Distributed Actuation Distributed actuation is normally concerned with synchronizing actions in two situations: actuators in different locations; replicated actuators. For example, the lack of tightness of a replicated actuation can transform an exactly-once actuation into several actuations with at-least-once semantics, and that may be undesirable. Consider *actuator granularity*, g_a, the interval from command input to completion of the action or until the actuator is ready for a new command, whichever is longer. During that interval, the actuator does not respond to further commands (e.g., a discharge laser is unable to fire again until it recharges). If the actuation points of the replicas are separated by g_a or more, they will be perceived as more than one actuation. In consequence: *the tightness of actuation must be better than actuator granularity,* $\tau < g_a$. However, it need not be much tighter than g_a, so, on the other hand, this condition removes the general belief that replicated output must be tightly synchronized. Take the example of the valve quad of Figure 12.17 in Section 12.9: many electric valves have actuation times in the order of the many hundreds of milliseconds, so replicated valves can support untightness of this order of magnitude.

13.10 REAL-TIME DATABASES

Real-time databases (RTDB) were born form the need to access, manipulate and update data with temporal constraints in a structured manner. That is, essentially what regular databases are about, except that the items of an RTDB have a *time-value* nature, and as such, their correctness "degrades" with time. For an introductory study on time-value entities that helps understanding this discussion, *see Time-Value Duality* in Chapter 12.

It is as if we were comparing the storage of bricks (non real-time database) with the storage of food (RTDB), for sale. If the bricks are intact they maintain their value practically forever, and we can sell and use them at any time. Food items decay with time, and as time goes by, they are worth less, until eventually becoming useless, if not sold in the meantime.

In complement to RTDBs, *active databases* were born from the need to detect and react to significant changes in the internal state of the database, triggering an action in consequence (Berndtsson and Hansson, 1995). This sequence is called *event-condition-action* or ECA.

13.10.1 Architecture of RTDB Systems

RTDBs are used: for recovery of values needed for a computation or an operation, in bounded time; for computing on sets of values with temporal validity (time-value entities), such as sensor observations; for combining these values with previously stored values in bounded time; for updating values cyclically, with pre-determined periods. A real-time database architecture has mostly the same building blocks as a non real-time one:

- transaction manager
- scheduler

- data manager
- data

The differences lie in the way the blocks work. The semantics of transactions, the rules for scheduling transactions, and the mechanisms for data management are oriented to fulfill time constraints, as well as logical. RTDBs offer transactions that aim at retaining the ACID properties (*see Transactions* in Chapter 8): atomicity; consistency; isolation; durability. Obviously, in a real-time context, this requires re-qualifying some of these definitions with a few constraining assumptions (Ramamritham, 1996b), in the context of what have been called **real-time transactions**. For example, consistency is different in a temporal context: when we abort a transaction so that the database remains logically consistent, it may no longer be temporally consistent, if some items lost their validity because of delay. In essence, an RTDB item is a *representative* ($r(E_i)$) of a *real-time entity* (E_i) (*see Entities and Representatives* in Chapter 12). The logical correctness of a database item depends on the observation originating it being *consistent* (*see Time-Value Duality* in Chapter 12). In temporal correctness terms, real-time databases additionally require the solution of two problems:

- keeping the value of each individual item consistent with its RTe;
- keeping the values of a collection of items mutually consistent.

The first is achieved by maintaining items *temporally consistent*. The second is secured by ensuring that the relevant collection of observations is *mutually consistent*. *How is this checked?* (a) By writing the observation timestamp and the absolute validity interval together with the item when an update is done $(\langle r(E_i), T_i, \mathcal{T}_a \rangle$. Then, when the item is used, T_{now} must be *within* $\mathcal{T}_a$ *from* T_i, for temporal consistency. (b) By checking that all timestamps of the collection of items about to be used fall within the relevant relative validity interval $\mathcal{T}_m$, for mutual consistency. *How is this enforced?* By ensuring that two conditions are simultaneously met: refreshing each item before it loses validity, that is, at most by the end of the absolute validity interval; ensuring that the update instants of all items of a collection always fall within an interval not greater than the relative validity interval.

In conclusion, an RTDB is consistent at time t, if and only if its items are consistent. The RTDB must be updated in accordance to the validity requirements of the several items. Note that enforcing mutual consistency (often called relative consistency in an RTDB context) may imply tightening update intervals for individual items as would be defined for enforcing absolute consistency alone.

13.10.2 Real-Time Transactions

In a client/server style of operation such as provided by RTDBs, there is a problem with guaranteeing all temporal specifications, if no assumptions are made about when and how often requests can be issued by clients to the database. In that sense, RTDBs can be designed having in mind the same classes of real-

time discussed earlier: hard, mission-critical, soft, and even non-real-time, as far as the coverage of those guarantees is concerned. This has implications on the internal organization of the transaction manager and scheduler modules. In functional terms, real-time transactions invoked on RTDBs can also be classified in several types, depending on the action performed:

write-only	an observation is written on the RTDB, possibly overwriting an existing item with a fresh value (e.g., a temperature)
update	an item is read, updated, written again (e.g. event counter)
read-only	an item is read, possibly for being used in computations, and/or to be output to an actuator

Real-time transactions have logical as well as temporal correctness criteria. Let us exemplify the latter. Consider the following RTe characterizations: E_1, E_2 : $\langle$*temperature*,$\mathcal{T}_T = 50\rangle$; E_3, E_4 : $\langle$*pressure*,$\mathcal{T}_P = 10\rangle$ (the absolute validity intervals $\mathcal{T}_T$ and $\mathcal{T}_P$ were computed based on the shortest time needed to attain a variation of ν_T and ν_P respectively, in the value of a temperature and a pressure). Consider further the set $S = \langle\{E_1, E_2, E_3, E_4\}, \mathcal{T}_m = 5\rangle$, for which a relative validity interval $\mathcal{T}_m$ is specified. Database items are specified as $R_i = \langle V_i, T_i, \mathcal{T}_i\rangle$. Now, consider the following situation at instant $T = 1800$ time units, w.r.t. stored items: $R_1 = \langle 147, 1752, 50\rangle$; $R_2 = \langle 162, 1755, 50\rangle$; $R_3 = \langle 1088, 1751, 110\rangle$; $R_4 = \langle 1114, 1750, 110\rangle$. The RTDB is absolutely consistent, because none of the items has lost its validity, and it is mutually (or relatively) consistent, because current timestamps are not separated by more than 5 units of time. Nevertheless, R_1 must be updated until $T = 1802$, in order to remain absolutely consistent. However, at the moment this is done, the set loses its mutual consistency. This shows the interdependence of both kinds of consistency. The remedy lies in bringing the update times of all variables in set S to within $\mathcal{T}_m$, while ensuring that the update period is not greater than the shortest absolute validity interval. When items belong to different sets bound by relative validity specifications, the smallest of the relative validity intervals should be used for the mechanism above.

13.11 QUALITY-OF-SERVICE MODELS

Quality-of-service models are mainly used to support soft or mission-critical real-time applications, when there is a need for ensuring an *end-to-end data flow* with real-time and reliability guarantees during a mission (radar data capture) or a session (remote video rendering), but unlike hard real-time systems:

- these guarantees must be *negotiated* on a need basis, in competition with other flows, in what forms a *contract*;
- what is granted by the infrastructure may be less than what was asked for;
- the contracted guarantees may vary during the mission/session, by initiative of the infrastructure or of the contractor.

The QoS model finds applicability in producer-consumer applications, such as those found in real-time instrumentation or telemetry, and in multimedia

capture and rendering. In particular, the concept of QoS adaptation, which means reviewing the QoS contract as a means to maintain some stability in the coverage of the received versus expected guarantees, has a generic application in the field of mission critical real-time systems. It may form the basis for the formal reasoning about mechanisms such as cost-value functions and operational envelopes (*see* Sections 13.1 and 14.5).

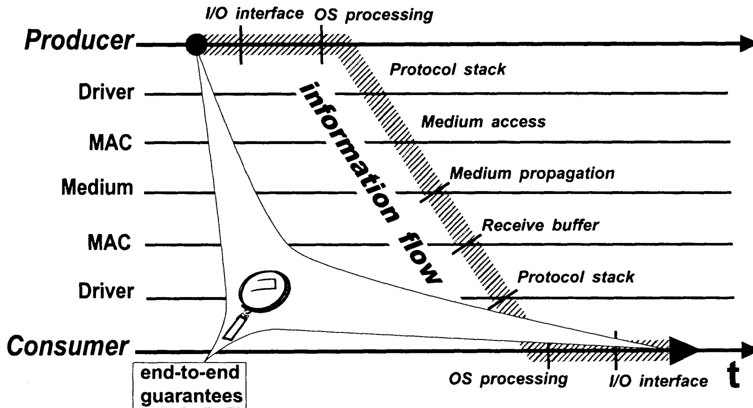

Figure 13.9. The Quality-of-Service Model

The main characteristics of the QoS model in distributed settings (Aurrecoechea et al., 1998) are suggested in Figure 13.9:

- end-to-end establishment of QoS guarantees for a given flow– timeliness, performance, reliability;
- piecewise enforcement of guarantees throughout the path– device, OS and communications driver scheduling, latency of access and medium throughput, receive buffer handling, flow control and error processing;
- maintenance of guarantees– surveillance of the flow QoS.

The steps towards implementing a QoS model concern QoS specification and management, and have been typified in (Hutchison et al., 1994):

- specification and mapping
- negotiation and resource allocation
- admission control
- maintenance, monitoring and policing
- renegotiation and adaptation

QoS specifications are made in terms of parameters falling into different classes or categories, such as timeliness, or volume, or reliability. Within each class, QoS is specified with metrics adequate to the magnitudes concerned. For example: `latency`, `tightness`, or `jitter` specify *timeliness*; whereas `throughput` or `BurstLength` specify *volume*; and `BER` (bit error rate) or `MTBF` (mean time between failures), or `MTTR` (mean time to repair) specify *reliability*.

A **QoS specification** consists of a list of parameters and values and intervals of validity for them, indicating the conditions, the goals and the importance of each parameter P. A possible specification may be like:

Sampling Period (TS)- the interval over which the value of the parameter is acquired. For example, if P is a round-trip time, a distribution function (pdf) is computed with the samples.

Threshold (TH)- the upper or lower acceptable bound on P. A typical upper bound is for a round-trip duration. A typical lower bound is for a bandwidth.

Weight (WT)- a measure of the relative importance of parameter P. A value of 0 would mean that the parameter would not be taken into account.

A specification is made in abstract terms, which must be mapped on the infrastructure. When the *mapping* takes place, the specification may unfold onto several more detailed sub-specifications. For example, a `BER` (bit error rate) requires error counters and local interval clocks (timers). A `latency` in message delivery is divided in the partial delays across the message path.

Through *negotiation* a *contract* is established with the infrastructure, and thus the several components involved, in order that there is a guarantee that the specification will be respected. After a service QoS specification is presented, components declare the best QoS they can supply for the desired service. Depending on that response, the management 'contracts' a given level of QoS or instead reports the end-user (e.g., an application) that the desired QoS cannot be obtained.

After negotiation, the components try to perform *resource allocation* in order to secure the promised support. For example, to reserve a certain amount of buffers, or to allocate a given level of priority to a given flow, or to redefine the O.S. schedule of processes. A successful negotiation should be seen as a promise rather than a firm commitment, which only takes place when *admission control* is passed, that is, when the system remains stable after the allocation of resources to the pending request. Namely, the QoS of other services should not be affected.

Parameters may have an associated assumed *coverage*, that establishes the desired level of guarantee for the service. This is for most applications a *statistical* rather than an *instantaneous* problem. In other words, it is necessary that the QoS holds over an integral of time, regardless of small glitches. In real-time language, this could be described in terms of mission-critical or soft real-time behavior. This means that maintaining QoS is subject to typical statistical control problems such as inertia, histeresis and instability.

A special kind of timing failure detectors should be used that understand this problem. We call them **QoS failure detectors**, QoS-FD, and they are designed to evaluate a number of relevant operational parameters, over an interval of time (Veríssimo and Raynal, 2000). The detector is configured through the QoS specification issued by the application, which instructs it how to perform *QoS tests*. The detector tests the end-to-end QoS seen in the flow from the

producer to all consumers involved in a distributed application. The value V of each parameter P of the QoS specification is evaluated by the QoS-FD over the sampling period TS. The sampled values are tested against the threshold TH. A variable *threshold exceeded*, TE, accounts for the percentage of the sampling periods of P where it fell beyond the bound TH during TS. Besides providing analog information about individual parameters, the Qos-FD may provide a boolean notion of *QoS failure* at a given node, akin to what is provided by crash failure detectors. This indication may be constructed by first computing a consolidated analog indicator, for each node, which is an average of the TE's of all parameters P, weighted by the respective WT variables. Then, define a global threshold for this analog indicator, which when passed generates a boolean failure indication. These individual failure indications can be consolidated in vectors, which form the opinion of one node about the others, or in a matrix, if nodes exchange their vector information, as crash failure detectors do (Veríssimo and Raynal, 2000). The specification may also contain the indication of the exception processing to be taken when the specified level of service is not attained. When the QoS-FD reports a failure, it can forward useful information to the application or middleware support, such as, for example, the lateness degree of timing failures, or the current point of the desired timeliness bound in the variable's pdf (probability density function). This information is precious for the higher layers to decide what to do. Depending on the system's ability to handle the situation, reaction to a QoS failure may have several outcomes:

termination- the activity cannot withstand a lower QoS; we say that the activity is not adaptable;

adaptation- the activity may reduce its requirements on the system, expecting that the offered QoS recovers (e.g., by reducing the refreshing rate of the scene in a teleconferencing application, or by reducing the refreshing rate of a radar trace); alternatively, the activity may live with the degraded QoS, by resorting to adaptation mechanisms using the own application heuristics (e.g., by passing from color to B&W rendering in the same teleconferencing application, or by using less sophisticated target trajectory prediction algorithms);

renegotiation- the activity triggers a renegotiation procedure, trying to obtain a new but satisfactory balance on QoS, normally by reducing its demand on the affected dimension but trying to recover in an alternative one (e.g., if bandwidth is stressed but there is spare computing power, augmenting the compression factor of the information transmitted from the remote sites relieves communication at the cost of more MIPS, but information significance— such as image quality, or sensor data accuracy— is maintained)

13.12 SUMMARY AND FURTHER READING

In this chapter, we discussed the main models of real-time distributed computing. The first part of the chapter was concerned with providing the system architect with a comprehensive view about the available architectural

options, frameworks and strategies. The rest of the chapter was devoted to models of distributed real-time computing addressing concrete functional and non-functional aspects: event-triggered and time-triggered operation; communication; control; databases; and quality-of-service. We advise the following works as further reading. A detailed study of real-time programming issues is done in (Burns and Wellings, 1996). Specifically on real-time object computing, we suggest (Kaiser and Livani, 1998; Yau and Karim, 2000; Becker et al., 2000; Siqueira and Cahill, 2000), or (Jensen, 2000; Kalogeraki et al., 2000).

On the formal specification and verification of the timeliness aspects of real-time designs, we suggest (Koymans, 1990; Lamport, 1994; Sinha and Suri, 1999; Graw et al., 2000; Bozga et al., 1998). The relation between order and synchronism specifications is explained in (Veríssimo, 1996). Some researchers have studied the partial synchrony of systems (Dolev et al., 1983; Dwork et al., 1988) under a time-free perspective. For further study on timed partially synchronous models see (Cristian and Fetzer, 1998; Veríssimo and Raynal, 2000). On programming with partial synchrony see (Almeida and Veríssimo, 1996; Fetzer and Cristian, 1997a; Veríssimo et al., 2000).

For further study of real-time networks see: (Tanenbaum, 1996; Halsall, 1994) about most common standard LANs; or (Pimentel, 1990; Pleinevaux and Decotignie, 1988) for field buses. Practical details on the definition and use of abstract network properties in the construction of reliable real-time communication protocols are given in (Veríssimo and Marques, 1990; Rufino et al., 1998). Studies about network scheduling can further be found in (Tindell and Burns, 1994; Tindell et al., 1994; Zuberi and Shin, 1995), or (Tindell et al., 1995; Lehoczky, 1998; Tovar et al., 1999). On medium reliability of real-time networks, see (Veríssimo, 1988; Rufino et al., 1999) for buses, or (FDDI, 1986; Rom, 1988) for rings. Likewise, a few space- and time-redundant approaches for fault tolerance in real-time communication are described in (Babaoğlu et al., 1986; Zheng and Shin, 1992; Cristian, 1990; Veríssimo and Marques, 1990; Cristian et al., 1985). Studies and measurements on the performability and inaccessibility of networks are described in (Meyer et al., 1989; Rufino and Veríssimo, 1992; Prodromides and Sanders, 1993; Veríssimo et al., 1997).

Concepts and design of distributed computer control systems are discussed in (Steusloff, 1981; Fisher, 1990; Bauer et al., 1991; Wikander, 1998).

A discussion on the issue of fulfilling temporal constraints on RTDBs can be found in (Ramamritham, 1995; Song and Liu, 1992). For further study on real-time and active databases, the reader is referred to (Ramamritham, 1996b; Berndtsson and Hansson, 1995; Purimetla et al., 1995; Korth et al., 1996; Ozden et al., 1996).

In (Jeffay, 1993; Rajkumar et al., 1995; Kaiser and Mock, 1999) the real-time producer-consumer model is discussed. For further study on QoS architectures, the reader is pointed to (Abdelzaher and Shin, 1998; Leslie et al., 1996; Volg et al., 1996). Discussions on the construction of adaptive applications are made in (Cosquer et al., 1996; Friday et al., 1999).

Exhibit 2026 Page 371

14 DISTRIBUTED REAL-TIME SYSTEMS AND PLATFORMS

This chapter gives examples of systems and platforms for real-time computing, consolidating the matters discussed in the previous chapters. Namely, it discusses: operating systems; real-time LANs and field buses; time services; embedded systems; dynamic mission-critical systems; real-time over the Internet. In each section, we will mention several examples in a summarized form, and then will describe one or two the most relevant in detail. Table 14.1 at the end of the chapter gives a few URL pointers to where information about most of these systems can be found. The table also points to the IETF Request for Comments site, where any RFCs cited can also be found.

14.1 OPERATING SYSTEMS

Several real-time operating systems have been deployed in the recent years. Many assume the form of a *real-time multitasking executive*, a simplified and highly-modular kernel, normally working with preemptive scheduling based on priorities. Most of these systems use fixed priorities. Concurrency is normally based on synchronization primitives offered by the system support. Typical functions offered through systems calls are: process management (creation, deletion, blocking, suspension, scheduling); memory management; synchronization and inter-process communication (queues, semaphores, mailboxes). Interrupt management is often a sophisticated two-level structure: front-end interrupt handler for immediate processing of an interrupt event; and interrupt

Exhibit 2026 Page 372
P. Verissimo et al. *Distributed Systems for System Architects*
© Kluwer Academic 2001

task, for more complete processing, scheduled in competition with all others, with a priority matched with the interrupt level. Amongst the examples of such kernels, one can enumerate: RTEMS, VxWorks, VRTX32; OS9. Some of these solutions have fully-fledged user-oriented subsystems, sometimes offering a UNIX-like interface along with the real-time functionality, such as QNX or LynxOS.

There are a number of experimental research real-time operating systems, such as RT-Mach and Spring. The Real-Time Mach (Tokuda et al., 1990) incorporated ideas from a previous project, the ARTS distributed real-time operating system. ARTS is an object-oriented system. A real-time object in the ARTS context is associated a maximum termination time (time fence), and an exception handler, for when the fence is jumped over. RT-Mach is called a resource kernel, i.e. one providing resource-centric services that can be used to satisfy end-to-end QoS requirements. These are normally handled by a QoS manager sitting on top of RT-Mach, which can make adaptive adjustments to resources allocated to applications.

The Spring kernel (Stankovic and Ramamritham, 1991) is the operating system of an experimental distributed real-time system addressing mission-critical and soft real-time applications. The architecture comprises a network of multiprocessor nodes (SpringNet). All system calls have a bounded WCET, and tasks perform resource reservation before execution, so that a task, when it executes, has a predictable WCET. Spring schedules according to a combined notion of timeliness and importance: critical, essential and unessential tasks. Spring gives a-priori guarantees to the critical tasks, while dynamically guaranteeing deadlines of the other arriving tasks.

Other operating systems are not made from scratch, but are rather evolutions of standard ones, such as the several variations of real-time UNIX (e.g., Solaris RT), or the real-time Linux (RTLinux). These variants modify UNIX in areas such as kernel scheduling (turned preemptive and re-entrant), IPC (improved synchronization), memory and disk management (swap prevention), and I/O subsystem, in order to achieve timeliness guarantees from an otherwise time-sharing system. Next, we analyze RTLinux with more detail.

14.1.1 Real-Time Linux

RTLinux was developed in the New Mexico Institute of Mining and Technology, USA, and presented in (Barabanov and Yodaiken, 1997). It is designed as a real-time kernel on the bare machine, on top of which several real-time and non real-time tasks may run. Linux itself runs as one of the latter kind. Linux runs at a non real-time level, with the lowest priority, and it can be preempted at any time by higher-priority tasks. The basic RTLinux scheduler is preemptive amongst tasks with fixed priorities. Alternative EDF and rate-monotonic schedulers are also available. It handles sporadic as well as periodic tasks. RTLinux provides low-level task creation, interrupt handlers, and IPC through shared memory and queues, for communication between interrupt han-

dlers and tasks, and Linux processes. RTLinux recursively uses Linux services when appropriate.

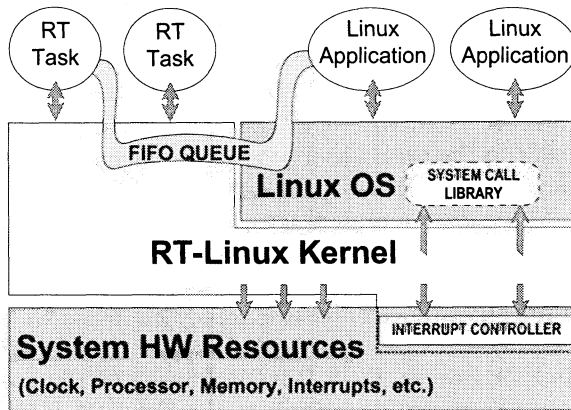

Figure 14.1. RTLinux Architecture

The RTLinux offers simple primitives. The `rt_task_init` primitive creates and launches a task, with a given priority. A task may be rendered periodic, with a pre-defined period, through `rt_task_make_periodic`. The `rt_task_wait` puts a task to sleep. IPC between RTLinux tasks and Linux processes takes place through FIFO queues, created by `rtf_create`. Primitives `rtf_put` and `rtf_get` enqueue and dequeue data respectively. However, since real-time tasks can communicate with Linux processes, it enjoys all the functionality of the latter for the (non-real-time) activities concerned with person-machine interfacing, storage, Internet access, etc. However, all communication is non-blocking, so that RTLinux is never delayed by synchronization or resource contention with a non real-time (low priority) process. RTLinux traps all external system interrupts and enable/disable interrupt instructions, so that Linux cannot disturb the real-time guarantees, for example by disabling interrupts (which it does very often).

14.2 REAL-TIME LANS AND FIELD BUSES

We have discussed the role of LANs and field buses in distributed real-time architectures, in Section 11, and addressed some LAN and field bus scheduling mechanisms in Section 12.7. The Token Bus LAN has been the most promising *real-time LAN*. Its timed-token scheduling alows the definition of one hard real-time and several soft real-time latency classes. However, the tuning of timers and other parameters is a complex operation, requiring considerable expertise. If not done properly, it can significantly degrade the operation of the network, and this was probably one of the reasons for the decline in the use of Token Bus. Several attempts have been made to achieve real-time operation on Ethernet. Earlier on, a modified Ethernet has been proposed, DCR Eth-

ernet, featuring deterministic collision resolution (Le Lann and Rivière, 1993). Recent full-duplex switched Ethernet networks open new perspectives for real-time operation, since collisions are avoided, and thus real-time communication protocols can be designed for this new physical reality of Ethernet.

Field buses are assuming increasing importance in distributed real-time systems. Early field buses such as MIL-STD (MIL-STD-1553B, 1988) or FIP (FIP, 1990) had a centralized structure. FIP (Factory Instrumentation Protocol) is a field bus oriented to the centralized produced-consumer paradigm, currently a european standard (EN-50170-3,1996) called WorldFIP. WorldFIP performs transactions of data buffers and messages, addressed point-to-point, in multicast or in broadcast. Data is limited to 128 bytes per frame. In fact, the initial purpose was to extend I/O cabling from a controller unit (the master node). It was not until the appearance of decentralized field buses such as Profibus or CAN that they started emerging as potential support for embedded distributed systems. Profibus, or Process Field Bus (Profibus, 1991) is a field bus with a MAC (medium access control) inspired by the Token Bus LAN, currently a european standard (EN-50170-2,1996). Although Profibus has a decentralized nature, it recognizes master and slave stations. Master stations are normally controller nodes, the ones that compete for bus access. Slave stations are passive, normally corresponding to I/O nodes.

14.2.1 CAN – Controller Area Network

CAN is inspired by the Ethernet, and is in fact a *carrier sense multi-access with deterministic collision resolution* network. Arbitration of medium access is done by direct bit-by-bit comparison of the 11-bit frame identifiers of the transmitting stations (*see* Figure 14.2). Bits can be *recessive* (zero) or *dominant* (one), and if a recessive and a dominant bit are transmitted simultaneously on the bus, the latter imposes itself to the former on the bus: while transmitting a frame identifier, each station monitors the bus; if it transmits a recessive bit, and a dominant bit is monitored, the station gives up transmitting and starts to receive incoming data; the station transmitting the lowest identifier goes through and gets the bus. Note that each identifier defines a *priority* (in inverse order of the identifier value), and frame transmission scheduling in CAN is thus by highest priority, on a per station basis.

Bit-by-bit arbitration works because the network works in quasi-stationary mode, that is, the signal phase is the same throughout the bus length. However, this requires a small length/rate product. The bus length and data rate are related, and typically, we have 40 meter @ 1Mbps, or 100 meter @ 50Kbps. Data is limited to 8 bytes per frame. CAN performs detection and recovers from a number of errors, such as bit errors, CRC errors, and missing acknowledgments. When an error in transmission is detected, an error flag is broadcast by recipients, the frame is re-transmitted immediately and the previous transmission discarded. This achieves an almost atomic broadcast behavior, except if very rare failure scenarios occur.

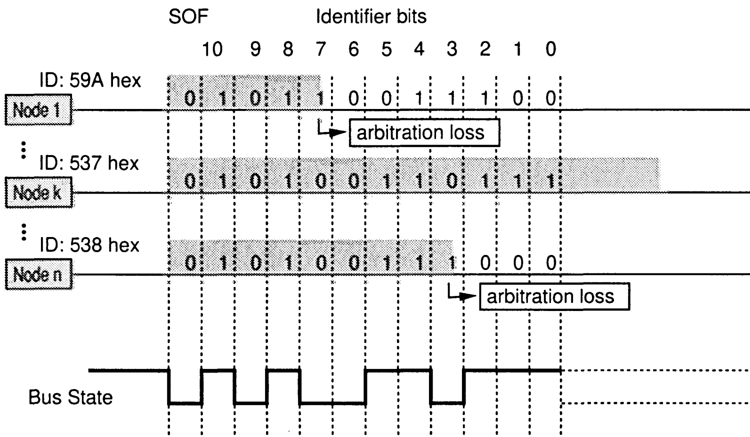

Figure 14.2. CAN Arbitration

14.3 TIME SERVICES

Systems have a basic local clock service, offered through a `getTime` primitive. This monotonic up-counter allows measuring interval durations. Most systems offer a timer or alarm service, which counts down from an initial value, normally provided through a `startTimer(timeout)` primitive. The timer can be canceled before it expires (`killTimer`), otherwise it raises an alarm. In some systems, the timer launch event can be linked to the automatic execution of a function in response to the timeout (`startTimer(timeout,function)`). This is the most precise way of taking an action upon a timeout, since there are no spurious delays in between.

Increasingly more often, local clocks in a distributed system are synchronized (*see Clock Synchronization* in Chapter 12), which gives the `getTime` primitive a global meaning: it returns the time at any site in the system, with a difference bounded by precision π. When clocks are externally synchronized to a standard source, such as UTC, `getTime` returns a time that is comparable with time in other external sources (e.g., our wrist watch), with a difference bounded by accuracy α. This also guarantees precision, since $\pi = 2\alpha$. A facility common in real-time operating systems is the ability to schedule the execution of functions at a given future time, such as `execAt(hour,function)`.

For large-scale systems, several global time services have been deployed through the recent years. The Berkeley TEMPO system (Gusella and Zatti, 1989) was developed in the context of the UNIX Berkeley effort, for internal synchronization. TEMPO is based on a hybrid agreement master-slave algorithm. An external hybrid algorithm combining master-slave round-trip and averaging agreement is presented (Fetzer and Cristian, 1997b). CESIUM-SPRAY is another system that provides external synchronization, based on a hybrid agreement master-slave synchronization approach (Veríssimo et al., 1997). The master is the GPS NavStar (Parkinson and Gilbert, 1983) system of satel-

lites, which disseminates its time reference through the slave nodes with GPS receivers on ground, at least one per LAN. All nodes in each LAN participate in an averaging-non-averaging clock synchronization algorithm that injects the GPS time in all other clocks.

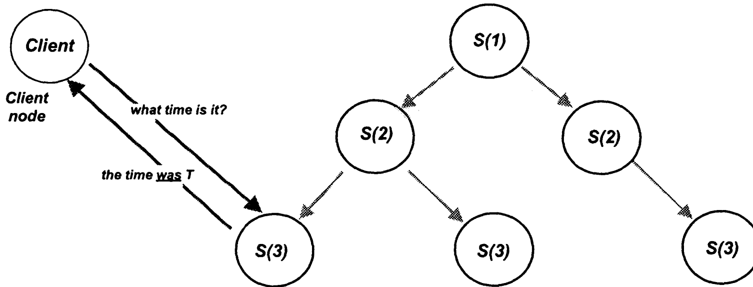

Figure 14.3. The NTP System with Server Hierarchy ($S(i)$ - Stratum i), and detail of Client Synchronization from a Remote Server

14.3.1 Network Time Protocol

The Network Time Protocol (RFC 1119) is the most widely deployed time service and protocol (Mills, 1991). It has been serving as the time reference for Internet nodes. The NTP Time Synchronization Service is a complex hybrid structure, inspired by Cristian's master-slave round-trip synchronization protocol (*see* Figure 12.16 in Chapter 12), which combines master-slave dissemination and round-trip with AVG agreement for synchronization of the server tree, with master-slave round-trip for client synchronization. The system is implemented as a hierarchical structure of masters spread across the Internet, as depicted in Figure 14.3. The masters supply UTC time, and synchronize the slaves (clients) with an accuracy that is probabilistic and depends on the stability of transmission times on the Internet, to and from the master. The NTP protocol currently offers some protection against spoofing of time datagrams, through authentication. It also offers interfaces for correcting the rate of drift of hardware clocks, if such mechanisms exist on the client or server computers.

Master clock servers are organized in descending order of intrinsic accuracy in the hierarchy. The **stratum** 1 servers are always directly connected to a UTC source of known accuracy, such as a GPS receiver. The *strata* below, 2 to n, synchronize themselves to servers of the immediately higher stratum. The inter-server synchronization can be round-trip, but when good quality accuracy is desired, servers can perform *symmetric* message exchanges, whereby servers of the same stratum or adjoining strata improve their synchronization through agreement-based adjustments. A third and simple scheme is available, obtaining good accuracies when servers are connected through a high-speed, low-delay link, by simply *multicasting* the time and doing a fixed correction on the (small) delay. At the time of this writing, the NTP system is reported to

have around 175,000 active hosts, of which over two hundred servers belong to the public infrastructure of NTP on the Internet (Strata 1 and 2), and has been reported to grant clients average accuracies in the order of the several tens of milliseconds (Minar, 1999).

The Distributed Time Service (DTS) of the DCE platform (*see DCE* in Chapter 4) is inspired by the same philosophy of the NTP system. The DTS is both available for users and for internal use of other services. For example, the protocols of the Kerberos authentication service of DCE rely on the existence of global time. The DTS format is : `1999-12-31-23:59:30.456+01:00I000.125` The first field indicates date, followed by the time, down to the millisecond (`23:59:30.456`). The *time differential factor* field (`+01:00`) indicates the deviation of the relevant time zone from the Greenwich meridian zone (longitude zero). DTS (such as NTP) is capable of estimating how accurate it is running, this is given by the *inaccuracy term*, `I000.125`, which would indicate in this case a worst-case accuracy of ±125 milliseconds.

14.4 EMBEDDED SYSTEMS

Embedded systems normally have a static character, are small-scale, and are used for hard real-time applications. They are often time-triggered.

A number of experimental systems have been developed in the past few years. Maruti-II (Saksena et al., 1995) is a distributed time-triggered real-time system. The computational model of Maruti is organized around elementary programming units that are connected to one another in acyclical unidirectional graphs, through a configuration language, forming execution threads. Schedulability of the threads is analyzed off-line. MAFT (Kieckhafer et al., 1988) and FTPP (Harper et al., 1988) are fault-tolerant real-time systems specially designed for safety critical embedded applications. They are based on communication systems resilient to arbitrary failures while securing timeliness requirements, and thus take a fully synchronous byzantine agreement approach and cyclic scheduling. GUARDS (Wellings et al., 1998; Powell et al., 1999) is a generic fault-tolerant computer architecture based on commercial off-the-shelf (COTS) hardware and software components. The architecture is configurable in two axes: in terms of different fault-tolerance strategies based on modular physical redundancy; and in terms of different integrity levels, allowing the co-existence of sub-systems of different criticality.

14.4.1 Mechatronic Architectures

Mechatronics is the integration of electronic and computational technologies on devices that were originally mechanical. Since they contribute to component integration, they are often used in small embedded systems. Examples are: the control devices of an automatic photographic or video camera; active car breaking or suspension devices; "intelligent actuators", such as robot manipulators. The advantages of mechatronic are manifold:

- reliability– integration reduces the weak points of interconnections

- simplicity– less moving/mechanical parts
- economy– lower price
- miniaturization– computer/electronics replacing cumbersome apparatuses;
- quality– better parametric quality, such as precision, responsiveness, etc.

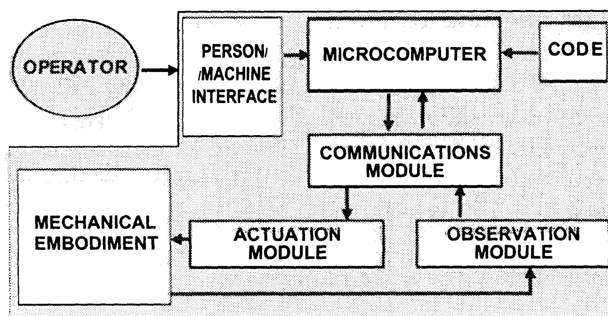

Figure 14.4. Mechatronic Architecture

The architecture of a typical mechatronic system is depicted in Figure 14.4. Note the integration between the computational, the I/O and the physical parts. The mechanical embodiment comprises all the mechanical parts, functional and encapsulation, and is connected to the control part through sensors and actuators, organized in two modules (observation and actuation). The communications module implements the abstraction of a miniature field bus interconnecting the sensors and actuators to the microcomputer assembly. Finally, the microcomputer can be hooked to a person-machine interface, when one exists.

14.4.2 The MARS System

MARS (MAintainable Real-time System) (Kopetz et al., 1989a) is an experimental distributed fault-tolerant hard real-time system for critical applications. Recently, an industrial prototype version called TTA (Time Triggered Architecture) has been implemented, with functionality incorporated in silicon with a view of applications in mass production areas such as automotive electronics.

The MARS architecture is oriented to testability of the proper operation of the system in the design phase. In order to help achieve this objective, MARS is based on the principle of resource adequacy, and follows a time-triggered (TT) activation discipline. Communication is performed by a specialized TDMA protocol called TTP (Time Triggered Protocol) (Kopetz and Grunsteidl, 1993). I/O pre-processors ensure that input events are transformed into state messages that are forwarded to the computing elements and instantiated as fresh copies of system state, overwriting previous ones. Periodic tasks then act at pre-defined moments on this newly updated state, proceeding without any external communication or synchronization until the production of output state

messages to tasks in other components, or command messages to the actuators. MARS follows a systematic software design methodology based on static (off-line) scheduling. Since there are no synchronization points, the maximum execution time of the tasks is easily determined by off-line code analysis.

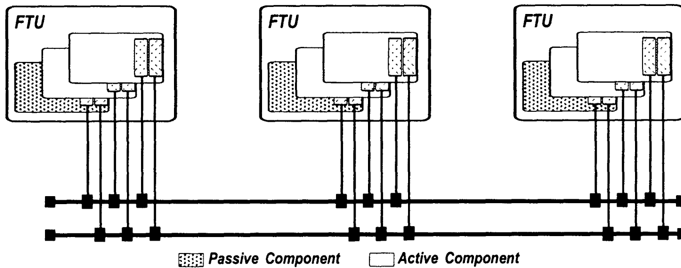

Figure 14.5. The MARS Architecture

The MARS architecture, shown in Figure 14.5, is organized around the cluster concept. A cluster can be decomposed into a set of Fault Tolerant Units (FTUs) interconnected by a replicated real-time communication channel that runs the TTP protocol. A Fault Tolerant Unit is normally composed of two replicated computer elements operating in active redundancy under the fail-silent model, backed-up by a shadow component: as long as any one of the components of a FTU is operational, the FTU is considered operational; the shadow acts as a hot-swap spare. A fault-tolerant clock synchronization service and a fault-tolerant membership service are integrated with the TTP protocol.

14.5 DYNAMIC SYSTEMS

Dynamic systems normally reflect the need for versatility and adaptability of large and complex real-time systems. Scheduling is dynamic, and the systems normally address mission-critical or soft real-time applications. They are often event-triggered.

HARTS (Shin, 1991) is an experimental real-time distributed system, based on a multicomputer cluster, interconnected by point-to-point links. Its operating system HARTOS allows hard and soft operation to coexist. HARTOS schedules groups of periodic tasks with the same priority, and schedules preemptively groups of different priorities. Processes can change priorities dynamically. HARTOS relies on reliable real-time communication and global time services for interacting with the other nodes. The ALPHA system (Jensen and Northcutt, 1990) also features dynamic priorities based on the *time-utility* principle. ALPHA is specially devoted to mission-critical systems, where scheduling decisions must comply with the uncertainty about the environment, yet provide the best possible timeliness assurances.

14.5.1 The Advanced Automation System

The Advanced Automation System (AAS) was developed by IBM for air traffic control (Cristian et al., 1996). The system architecture, comprises several modules or clusters, whose components are networked by replicated Token-Ring LANs. Figure 14.6 depicts the structure of an Area Control Computer Complex (ACCC), which supports the air traffic control functions. Clusters are interconnected to each other through a replicated backbone Token Ring, to each they connect by a Communication Gateway Subsystem, composed of replicated bridges. Radar sensor data is time-sensitive and must get to the central processors with real-time requirements. These must process the information, display it and help the operator make decisions. All these operations are time-critical.

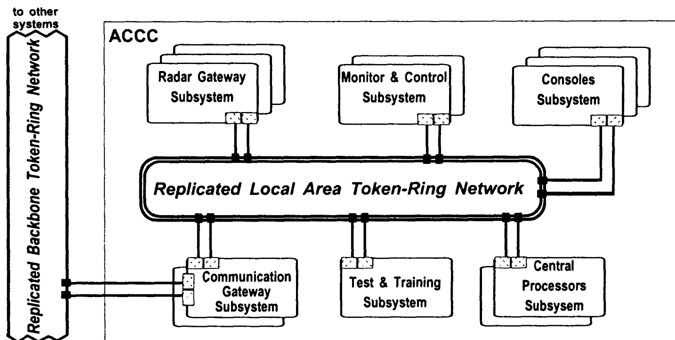

Figure 14.6. Advanced Automation System – Snapshot of a Cluster Network

Besides the functional modules, several services help achieve non-functional properties of AAS: topology and route managers, maintaining an accurate image of the current state of the network and its links; service availability manager, controlling the availability of the crucial AAS services, by means of failure detection, promoting passive replicas, re-inserting recovered units, etc.; global availability manager, controlling the state of all hardware and software components of the system. These services depend on a suite of lower-level fault-tolerant services: atomic broadcast; group membership; and synchronized clocks. The AAS protocols are clock-driven, and triggered either by time or events. The atomic broadcast and group membership protocols are of the Δ-protocol class, where messages are delivered after a Δ time on the local clock. The communication system is diffusion based, with space channel redundancy. Different levels of channel redundancy (up to four) are used in the different parts of AAS.

14.5.2 The Delta4-XPA System

Delta-4 is a system based on distributed fault tolerance (Powell, 1991). It has a modular architecture relying on software based fault tolerance, where components interact via reliable group communication and replication management

protocols (*see Distributed Fault-Tolerant Systems* in Chapter 9). Delta4-XPA, the Extra Performance Architecture of Delta-4, also described in (Powell, 1991), addresses the realm of mission-critical applications. XPA aims at implementing hard and soft real-time designs under an unbounded-demand perspective, that is, recognizing that a complete definition of the operational envelope is not possible, or that the definition imposed by design has a limited coverage. This requires the architecture to have some form of graceful degradation ability. In other words, while the system would normally act as a hard real-time one in the presence of the "foreseen" environmental and computational constraints, it would adapt to "unforeseen" situations — working in modes that are progressively less effective, precise, reliable, etc. — without abruptly falling apart. XPA can thus be described as a *hard real-time system with graceful degradation.*

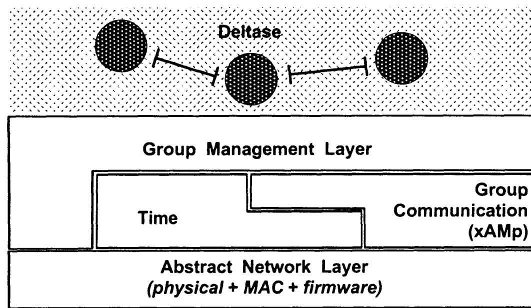

Figure 14.7. The Delta4-XPA Architecture

The architecture of XPA, depicted in Figure 14.7, comprises nodes interconnected through a LAN-based real-time communication subsystem, providing a suite of reliable group communication primitives called xAMp (Rodrigues and Veríssimo, 1992), and a fault-tolerant global timebase of synchronized clocks (Veríssimo and Rodrigues, 1992). The main policy for handling replication at the group management layer is *leader-follower*, a semi-active replication management mechanism that allows consistent replica preemption. XPA schedules under a combined notion of *importance*, the measure of the consequences of a computation not meeting its deadline, and *urgency*, the notion of approaching deadline. Scheduling is thus *priority-based* among classes of tasks ranked by importance, and then *earliest-deadline-first* inside each class.

14.6 REAL-TIME OVER THE INTERNET

The emergence of the multimedia era in Internet has encouraged a lot of research with the objective of achieving real-time communication guarantees on that uncertain infrastructure. Multimedia is intimately related to dissemination and conferencing, and as such the IP multicast subset of the Internet, the Mbone, was the field for early experiments on real-time over the Internet. Real-Time over the Internet is anyway associated with the notion of QoS specification, and of resource reservation. Ferrari does an extensive analysis of ways

of specifying client requirements for real-time communication quality of service, on open networks (RFC1193).

Efficient stream protocols appeared, such as ST2 (RFC1819), aiming at achieving end-to-end real-time communication through resource reservation. ST2 works at the same level of, and in complement to IP, but is connection-oriented. ST2 application areas are real-time transport of multimedia data, such as: digital audio and video packet streams, and distributed simulation and gaming. The ST2 communication model includes: a flow specification to express user real-time requirements; a setup protocol to establish real-time streams based on the flow specification; a data transfer protocol for the transmission of real-time data over those streams. ST2 is composed of two protocols: ST for the data transport; and SCMP, the Stream Control Message Protocol, for all control functions. Though ST2 supports multicast, it does not support the notion of group communication and management, which helps the construction of multiparty multimedia applications, such as in (Panzieri and Roccetti, 2000).

The Resource ReSerVation Protocol, RSVP, is a resource reservation setup protocol for Internet (RFC2205). RSVP is an Internet control protocol that provides receiver-initiated setup of resource reservations for multicast or unicast data flows, to be later used by transport protocols such as RTP. The RSVP protocol is used to ensure a specified QoS over a unidirectional data flow: it lets the sending host request the necessary resources, and it lets routers propagate the QoS requests along the route to the destination. QoS reservation is made by the recipients, and is propagated in the reverse data path back to the sender, which consolidates the requests of the several recipients.

RTP, the Real-Time Transport Protocol (RFC1889), provides end-to-end network transport of real-time data. It is accompanied by a control protocol (RTCP) to allow monitoring of the data delivery. RTP and RTCP are designed to be independent of the underlying transport and network layers. Applications typically run RTP on top of UDP. RTP can rely on low-level protocols such as ST2 to provide timeliness guarantees. RTP is complemented with companion standards defining how to specify the profiles and encoding rules for the several types of payload real-time data. With regard to security of the data flows, namely its confidentiality, RTP will use underlying services such as IPsec (*see Extranets and Virtual Private Networks* in Chapter 19).

14.7 SUMMARY AND FURTHER READING

This chapter gave examples of systems and platforms for distributed real-time computing. The objective of the chapter was to relate the notions learned throughout this Part with existing products and systems. We reviewed operating systems, real-time LANs and field buses, time services, embedded systems, and dynamic mission-critical systems. We finalized with an overview of protocols related with real-time on the Internet.

For further reading, Table 14.1 gives a few pointers to information about some of the systems described in this chapter. Detailed insight on the practical

design and SW/HW integration of real-time systems is given in (Laplante, 1997). Thorough discussions on real-time UNIX and Linux design issues are done in (Furht et al., 1991) and in the RTLinux links of Table 14.1. Real-time CORBA is an emerging OMG technology at the time of this writing, comprising: fixed priority scheduling, control over ORB resources for end-to-end predictability, and flexible communications. Real-time Java is receiving a great deal of interest, because of its potential for embedded applications. POSIX defines an important standard for a UNIX-based real-time programming API (POSIX, 1995).

On CAN scheduling, see further (Tindell and Burns, 1994; Tindell et al., 1994). On subtle CAN failure modes and CAN atomic broadcast, see (Rufino et al., 1998). CANopen is a set of technologies and recommendations for interoperability of CAN designs.

A recent survey on the NTP Time Service gives useful details on its current set-up and performance (Minar, 1999). Details on DCE-DTS can be found in (Lockhart Jr., 1994). Further material on AAS, Delta4-XPA, and MARS/TTA can be found in (Cristian, 1994; Speirs and Barrett, 1989; Kopetz, 1997). Amongst other interesting practical real-time systems we suggest looking at (Hachiga, 1992; Hedenetz, 1998; Heiner and Thurner, 1998).

Mechatronics is treated comprehensively in (Bradley et al., 1991; Wikander, 1998). Material on distributed industrial systems, such as Computer Integrated Manufacturing (CIM) and Supervisory Control and Acquisition (SCADA), can be found in (Beekmann, 1989; Veríssimo et al., 1996), and some of the pointers in Table 14.1. In (MAP, 1985; CNMA, 1993; MMS, 1990; ISODE, 1993) the main relevant standards are described.

Table 14.1. Pointers to Information about Real-Time Systems and Platforms

Sys. Class	*System*	*Pointers*
RFCs	(IETF)	www.rfc-editor.org/
OMG	(RT-CORBA)	www.omg.org
RTJ	(Sun RT-JAVA)	www.rtj.org
JC-RTJ	(J-Consort.)	www.j-consortium.org
RT-INFO	(RT Encyclopaedia)	www.realtime-info.be/encyc
IEEE-CS	(RT Systems)	cs-www.bu.edu/pub/ieee-rts
		www.britannica.com/clockworks
On Time		www.ubr.com/clocks
		www.newscientist.com/nsplus/insight/time
and		www.real-time.org
		tycho.usno.navy.mil/ctime.html
Real-Time		www.auto.tuwien.ac.at/Projects/SynUTC/time.html
	RTEMS	www.rtems.com
Real-Time	**RTLinux**	www.realtimelinux.org
Executives		www.rtlinux.org
and O.S.'s		www.realtimelinux.org/CRAN
		www.aero.polimi.it/projects/rtai
(experim.)		www.ittc.ukans.edu/kurt
	Alpha	www.realtime-os.com/alpha.html
	RT-Mach	www.cs.cmu.edu/afs/cs/project/art-6/www
	Spring	www-ccs.cs.umass.edu/rts/spring.html
Real-Time	**VxWorks**	www.vxworks.com
Executives	**QNX**	www.qnx.com
and O.S.'s	**LynxOS**	www.lynx.com
(commerc.)	**VRTX32**	www.mentorg.com/embedded/vrtxos
	OS-9	www.microware.com
	GPIB	www.ni.com/gpib
Real-Time	**PROFIBUS**	www.profibus.com
Networks	**WorldFIP**	www.worldfip.org
and	**CAN**	www.canopen.com
F. Buses		www.ems-wuensche.com
	XTP	www.ca.sandia.gov/xtp/forum.html
		dancer.ca.sandia.gov/pub/xtp4.0
	Time Services	tycho.usno.navy.mil/ctime.html
		www.boulder.nist.gov/timefreq
	Time Sync SW	www.eecis.udel.edu/~ntp/software.html
Time		www.ubr.com/clocks/timesw/timesw.html
and	**Time Sync HW**	www.eecis.udel.edu/~ntp/hardware.html
Clock	**NTP**	www.eecis.udel.edu/~ntp
Services	NTP CookBook	www.umich.edu/~rsug/services/ntp.html
	NTP Public Servers	www.eecis.udel.edu/~mills/ntp/servers.htm
	GPS	www.gpsworld.com
		www.gpsy.com/gpsinfo
RT DB's	**RTDB**	www.eng.uci.edu/ece/rtdb/rtdb.html
	Maruti-II	www.cs.umd.edu/projects/maruti
Real-Time	**HARTS**	www.eecs.umich.edu/RTCL/harts
Platforms	**GUARDS**	www.cs.york.ac.uk/rts/guards
	ACQUA	www.crhc.uiuc.edu/PERFORM
	MARS/TTA	www.vmars.tuwien.ac.at
	MAP/MMS	icawww.epfl.ch/MMS
Industrial	**FactoryLink**	www.USDATA.com/solution/factorylink.html
Platforms	**FactorySuite**	www.Wonderware.com/products/factorysuite71.htm
	LabView	www.ni.com

15 CASE STUDY: MAKING VP'63 TIMELY

This chapter continues the case study that we have been carrying throughout the book: The VP'63 (VintagePort'63) Large-Scale Information System. The wine company has planned to automate some of its industrial facilities and needs to guarantee two objectives: to implement distributed real-time control and automation of some units such as wine processing and bottling/corking; to incorporate the real-time supervisory, control and acquisition (SCADA) into the global enterprise resource planning and information system, under a CIM perspective.

15.1 FIRST STEPS TOWARDS CONTROL AND AUTOMATION

The reader should recall that this is the next step of a project implementing a strategic plan for the modernization of VP'63, started in Chapter 5, and continued in the Case-Study *chapters of each part of this book. The reader may wish to review the previous parts, in order to get in context with the project.*

Generalized networking does not exist, and the situation of the company's industrial facilities can be described as being in the stage of *islands of automation.* That is, even when there is some local networking inside a cell, there is not a seamless interconnection and information flow in the whole industrial plant. In the factory floor, and specially inside the islands of automation, the company has introduced ad-hoc automation during the course of the years, to solve localized problems or improve certain processes. Devices are wired di-

P. Verissimo et al. *Distributed Systems for System*
© Kluwer Academic 2001

rectly to the controllers, except for a few more recent units, where devices are wired to the PLC through a proprietary field bus. The degree of computerized control still lies on PLC (programmable logic controller) technology and relay logic programming, as exemplified in a bottling/corking cell in Figure 15.1a.

15.2 DISTRIBUTED SHOP-FLOOR CONTROL

There are essentially two areas in the factory or shop floor: the continuous control part of the problem, which concerns wine processing; the discrete control part, which concerns activities like bottling and corking and labeling. The team identified the following problems in the current setting:

- imprecise continuous control leads to lesser quality batches, and also makes it impossible to regularize crops with different attributes in order to achieve predictable characteristics of the company's blends;
- imprecise discrete control leads to a high percentage of rejects in quality control (e.g. imperfect filling, labeling and corking);
- generally, but more so in the discrete processes, the rigid hard-wired and hard-coded systems make it difficult and slow to change anything in the process, although the requirement for flexibility is ever growing.

Q.3. 1 How to evolve towards a flexible support architecture for distributed real-time shop-floor control?

The team devised a strategic development plan consisting of designing a pilot distributed control system, to test and assess the new technologies. After a sucessful pilot phase, the results will be extended to the whole of the industrial facilities. A bottling and corking cell was elected for this sub-project.

Essentially, the cell consists of three modules: `conveyor`, `filler` and `corker`. The labeling module was omitted for simplicity. The conveyor belt brings empty bottles into the cell, passes them in succession under the filler, under the corker and then out of the cell to the packaging cell. The filler fills the bottle with wine from a tap connected through a pipe system to the wine tanks. The corker consists of a cork insertion press, with its own cork feeding subsystem, which presses the compressed cork into the bottle. The system has a few sensors: `bottle-arrived-at-filler`, `bottle-arrived-at-corker`, `filler-flow-meter`. Additionally, it has the following actuators: `belt` (start, stop), `tap` (open, close), `press` (down,up).

The prototype cell architecture is depicted in Figure 15.1, where the team applied known notions on distributed real-time architectures. One computing element per module (conveyor, filler, corker) processes the relevant state evolution. The sensor and actuator representatives are materialized by drivers that handle the relevant controllers. Given the cell simplicity, computing elements and representatives were distributed by three physical nodes. Each node hosts a computing element. Sensors and actuators relevant to a module were placed with the node hosting the relevant computing element. This is mainly for the sake of wiring organization, since the information from sensors is disseminated

to all computing elements, so that they all have a common knowledge of the system state. Outputs to actuator representatives are also disseminated. This way further changes in topology do not require a reconfiguration of the IPC. Protocol choice and configuration should obey the hard real-time needs of this type of discrete control operation.

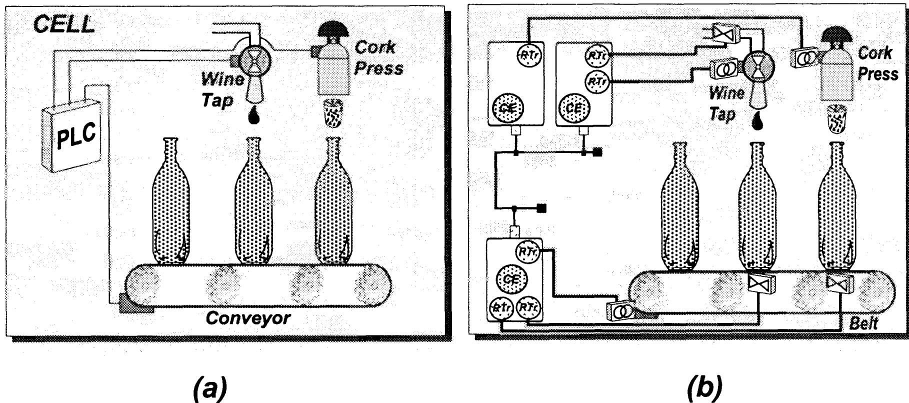

Figure 15.1. (a) Ad-hoc Computer Control; (b) Distributed Control of a Flexible Manufacturing Cell

15.3 INTEGRATION OF THE INDUSTRIAL SYSTEM

There are essentially two points to address in response to the corporate strategic plan:

- achieving a seamless industrial information flow between production management and shop-floor devices, in both directions (commands down, state image up)— this is now impossible because there is no global networking nor distributed information support;
- swifter interconnection of the industrial production information and the business information— achieving the first objective is a pre-condition to this one.

Q.3. 2 How to evolve towards an integrated industrial information system architecture?

Islands are not interconnected. Recipes and manufacturing orders have often to be inserted by hand in the shop-floor controllers.

Supervisory (SCADA) systems for different processes are proprietary, and having their own closed databases, making it impossible to dialogue with other subsystems and re-use their information. On the other hand, high-level applications (production management and control, maintenance management, quality control) run isolated, without an integrated connection to the business management system. In consequence, the business image of the complete enterprise has normally outdated information concerning the industrial facilities.

The team devised a plan for re-organizing the industrial facilities according to the CIM paradigm:

- design of an integrating networking infrastructure based on open protocols, TCP/IP for the matter, interconnecting all islands, with communication gateways to the office-level networking infrastructure;
- selection and installation of an open distributed SCADA platform providing a global image of the distributed processes to be shared by the high-level applications, and provision of a standard means of conveying commands and instructions from the latter to the shop-floor units;
- adaptation of existing applications or selection/installation of new applications capable of hooking to the open platform;
- design of a mechanism supporting the information flow between the industrial information system and the business information system.

The system architecture of a production unit is based on integrated networking between all cells, and a connection to the enterprise information system, which depends on the technology of the latter. Each cell is provided with a *cell controller*, where an instance of the distributed SCADA platform resides. Communication between devices and cell controllers, and between the latter and high level applications is ensured by the MMS factory communication standard, so that the VMD (virtual manufacturing device) paradigm can be used.

Cell controllers hold up-to-date copies of the cell state, and can replicate these images in other instances of the platform, namely those accessed by industrial applications. Conceptually, there is a seamless hierarchy of state freshness, from what may be called a *real-time image*, to a *historical image*. Real-time images are captured through the VMD event or polling interfaces, and held in volatile state in the cell controller. Their correct capture involves maintaining a real-time QoS stream between devices and the VMD. Historical images serve archival purposes, and hold the statistical memory of the process in persistent memory. They do not have real-time constraints, although they are often stored together with the timestamp of the sampling instant.

The other facet, that is, how orders, commands and instructions come from high-level applications to the shop-floor, is implemented through the VMD client-server interfaces. Devices, considered as VMDs, essentially dialogue with upper layer applications by acting as servers receiving commands in the form of client-server RPCs invoked by the industrial applications (e.g., modification of set-points in controllers, loading of new discrete control programs, loading of new continuous control recipes, loading of batch manufacturing orders etc.).

Further Issues

These issues need some refinement now, and the reader was assigned the study of a few questions that were still left unsolved:

Q.3. 3 *Would model would be more adequate for the continuous and for the discrete control cells respectively, time-triggered or event-triggered?*

Q.3. 4 *What measures should be taken to ensure the availability of the bottling/corking process implemented by the cell architecture just studied?*

Q.3. 5 *Are there safety issues in any of the production cells? If yes, money-critical or safety-critical?*

Q.3. 6 *Consider the remote applications visualizing shop-floor state images: in what way are they QoS-sensitive?*

Q.3. 7 *Could the distributed CIM platform take advantage from the generic distributed services put in place during the first phases (distribution and fault tolerance) of the project? How and which services?*

IV Security

If you think cryptography is the solution to your problem, then you do not understand cryptography, and... you do not understand your problem.
— Roger Needham

Contents

16. FUNDAMENTAL SECURITY CONCEPTS
17. SECURITY PARADIGMS
18 MODELS OF DISTRIBUTED SECURE COMPUTING
19 SECURE SYSTEMS AND PLATFORMS
20 CASE STUDY: MAKING VP'63 SECURE

Overview

Part IV – Security, addresses security of distributed systems, that is, how to ensure that they resist intruders. Security is paramount to the recognition of open distributed systems as the key technology in today's global communication and processing scenario. This part contains the fundamental notions concerning security, and provides a comprehensive treatment of the problem of security in distributed systems. Chapter 16, Fundamental Security Concepts, discusses the fundamental principles, such as the notions of risk, threat and vulnerability, and the properties of confidentiality, authenticity, integrity and availability. Chapter 17, Security Paradigms, treats the most important paradigms, such as: cryptography, digital signature and payment, secure networks and communication, protection and access control, firewalls, auditing. Chapters 18 and 19, Models of Distributed Secure Computing, and Secure Systems and Platforms, consolidate the notions of the previous chapters, in the form of models and systems for building and achieving: information security, authentication, electronic transactions, secure channels, remote operations and messaging, intranets and firewall systems, extranets and virtual private networks. Chapter 20 continues the case study, this time: making the VP'63 System secure.

16 FUNDAMENTAL SECURITY CONCEPTS

This chapter addresses the fundamental concepts concerning security. It starts by defining what security is: the reasons leading to insecurity, the types of computer misuse, and the evaluation of the risk associated with both the vulnerabilities of computers and the threats to which they are exposed. Then, it explains the foundations of secure computing, and traces the relationship between distribution and security, on the one hand, and fault tolerance and security, on the other hand. Next, it analyzes the behavior of the intruder, in an attempt to illustrate to the reader both the motivations for attack, and the techniques and procedures normally used to perform that attack. Finally, the most relevant architectural and technological approaches to security in networks and distributed systems are introduced, to be detailed in the subsequent chapters of this part.

16.1 A DEFINITION OF SECURITY

In order to understand security, we must first understand what are the impairments to security. As with fault tolerance, we can only assure correct behavior of systems, if we understand why and how they fail in the first place. Two of the things that make this field a challenging one are that: the direct causes of failure are faults (attacks) maliciously made by humans; and those are very often made possible by unintentional faults (vulnerabilities) made by designers, who are also humans... and humans are unpredictable.

P. Verissimo et al. *Distributed Systems for System*
© Kluwer Academic 2001

16.1.1 Insecurity, People and Computers

People and computers have a curious relationship: from an initial militant mistrust, at work and at home, people tend to evolve to an almost blind trust on the machines' abilities to solve problems, and to an overwhelming dependency on computers to hold important information and perform important tasks. Alas, this trust and dependency are frequently not supported on any technical evidence. We have seen that this may have serious consequences when accidental faults occur (*see Fault-Tolerant Computing* in Chapter 6). It can have even more serious ones in result of malicious faults caused by other people— the intruders— on the computer systems people— the legitimate users— so haphazardly use. We will spend sometime analyzing the situation from the human viewpoint, and will extract a few morale quotes, that we emphasize in italics.

Perhaps most of the problems with computer insecurity can be attributed to social factors, more precisely, to the fact that informatics evolved so fast that social rules have not accompanied it. To give an example, people tend to minimize the importance of the information that computers hold: they rarely think of preserving the privacy of their own information, and tend to do the same with information entrusted to them, e.g., professionally. However, are these people irresponsible? Not necessarily: many of them methodically lock everything inside their desk's drawers and office lockers, while leaving the whole disk of their PC within the reach of a few thumbscrews. This happens because their behavior codes with regard to computer-based information are still imperfect, if compared to other media.

Insecurity is as concerned with technical deficiencies as with people's attitudes.

On the other hand, most of the hackers are juveniles who consider that invading another person's computer, stealing a password, or eavesdropping on computer networks is part of an innocent game, and in fact a proof of their intellectual superiority over other youths or grown-ups, who for instance have such naive passwords as `snoopy`, or do not have their servers protected with the latest patch against the latest discovered vulnerability. In fact, most of these hackers have never thought that there is little difference, in ethical terms, between what they do and: invading another person's house, even if the door is not locked; stealing a key to a personal drawer; listening to private phone conversations, and so forth. The fact is that the ethics of informatics is not part of the basic education. That is, the rights to integrity and privacy of property and personal information are recognized in terms of the traditional icons (real estate, cars, lockers, telephones, etc.), but not so much with computers, disk files, and network bits.

However, the irresponsibility of the young and sometimes inept hackers does not make their actions less serious. The amount of information available today on the Net, for example, whole Web pages full of recipes for exploiting vulnerabilities in computer systems of all makes and brands, spreads the base of potential hackers. It is not too paranoid to speculate that behind an everyday thicker curtain of amateur hackers, people with a real interest in severing computer security— computer criminals, radical groups, terrorists— can act with an increased degree of impunity.

Since this book is mainly devoted to university students, we cannot refrain from addressing two words at the informaticians-to-be whose passion by computers is addressed at other people's computers. The first is that most employers will not be very eager to hire a computer science or engineering graduate with a record for computer-related crimes. The second is that there is a fantastic difference between a mechanical engineer, who is able to design and build a car, and a car thief, who is only able to break into the car. Of course, the latter is an expert on alarms, car door and steering wheel locks, which make, say, one hundredth of the important parts of a car. Guess who has the most comprehensive and creative activity?

Insecurity is quantitatively caused to a great extent by the actions of people who must be educated about the seriousness of their deeds.

Security costs money. If a security case is well made, it should normally cost less than the losses arising from rare but devastating incidents. However, such as with fault tolerance, it is very difficult to persuade people to invest on a system attribute whose effect they will rarely or never see. It is much easier to authorize an investment on making the system "perform faster", because that is palpable, than on making the system "be secure". (This message is dedicated to chief information or technology officers and other people with similar decision power). If you pay a security guard (or have an insurance), and there is never an attempt to rob your house, after a few years you start wondering why you are throwing all that money away. After you fire the guard (or cancel the insurance), you are finally robbed, and you lose ten times more than what you had paid until now, but that was a human reaction, do not be demoralized.

It is better to invest than to spend.

The general insecurity reaction to the first serious incident is no less human: close everything, resort to very restrictive emergency policies, like for example disconnecting from the network, and invade the place with all sorts of new bureaucratic rules. Again, an old saying in action: "locking the stable door after the horse has bolted". This is very frequently complemented with a bit of burying-head-in-the-sand, such as pretending nothing happened, not making a complaint, not requiring the services of computer security specialists. Moreover, since the pressure to be on-line is enormous these days, it can be decided, many often in the worst possible way, to buy "something to take care of the pirates", preferably "something that has cryptography inside". In 90% of the cases, the innocent target of this frenzy is a firewall, that mysterious panacea for all security incidents. Shortly after, and with great surprise, the system is attacked again. Sometimes because unprotected direct modem dial-up connections from behind the firewall remained active, sometimes because the intruders were already behind the firewall: they worked there. No technology works per se. Organization security requires a systemic approach, as many other architectural problems in computer systems do: analyze the problem; establish the requirements and/or the policy; specify the functionality; select the enabling technologies.

Cryptocracy (a form of technocracy) is enemy of good systems practice.

To end with, if there was a special advice we would give to beginners in this domain, it would be: in any security case, always use your knowledge on how

humans behave (the good and the bad). There is a well-known saying about the fundamental problems being those that remain no matter how much hardware or software you put or take: in security, what remains are humans and their values.

16.1.2 Vulnerability, Attack and Intrusion

Hackers exploit weaknesses in operating systems, applications, network software, and so forth. Reckless users or administrators may introduce other weaknesses, because of the way they configure or run the systems. Illustrative examples of weaknesses are: world-accessible files; accounts with default password or without one; easily guessed passwords; stack overflow and other low-level bugs; windows of vulnerability in the execution of certain system calls; cleartext remote logins; unprotected programs with root privileges; forgotten protocol ports; lack of authentication of most communication protocols.

Hackers also do social engineering, which is a way of exploiting human weaknesses. Social engineering is the art of extracting secrets from people with their unwitting consent, and it can do more for a hacker in five minutes than a fortnight of methodic probing. An example of social engineering that became folkloric is the following: a hacker phones a company's system operator pretending he works for the Telecom company, alleging that there is a problem with one of the company's lines that connects a given server to the outside; he says that in order to perform tests he will need a login and password into the server; he even asks the operator to remain on-line and report if the operation was successful. It is surprising to find out how fashionable variants of this story are.

Vulnerability - non-malicious fault or weakness in a computing or communication system that can be exploited with malicious intention

It is interesting to define vulnerability in terms of dependability: vulnerability is a non-malicious design or configuration fault that can be activated to introduce malicious faults and/or errors in that system. It is constructive to establish a parallel between fault tolerance and security, since after all both are facets of the same objective of dependability: achieving justified reliance in the operation of systems vis a vis the occurrence of faults, be they malicious (security) or not (fault tolerance). The resulting cross-fertilization may result in new ways of looking at either field.

Threat- potential of attack on computing or communication systems

The closest analogy to threat in dependability is the definition of hostility of the environment, that is, the potential of the environment to introduce faults (e.g., electromagnetic radiation).

Attack - malicious intentional fault introduced in a computing or communication system, with the intent of exploiting a vulnerability in that system

An attack may be successful or unsuccessful. The latter can happen because the system was not vulnerable to that attack (or, what is equivalent, the attack was perpetrated by an inept hacker). When successful, an attack may generate errors in that system. Attacks on computer systems (hacking) assume several forms, more or less disruptive or destructive, such as attempting to: penetrate a firewall or a computer, introduce viruses, damage resources, eavesdrop for information theft or privacy violation.

Intrusion - erroneous state resulting from a successful attack on a computing or communication system

If nothing is done, the intrusion will result in the failure of the security mechanisms. Alternatively, the erroneous state characterizing the intrusion may be detected and fought back with countermeasures, or it may be masked if the system has enough defenses to resist the intrusion. In fact, these are underlying techniques of what we may call intrusion tolerance. The process of security failure, depicted in Figure 16.1, has an obvious analogy with the fault-error-failure sequence studied in the Fault Tolerance Part of the book.

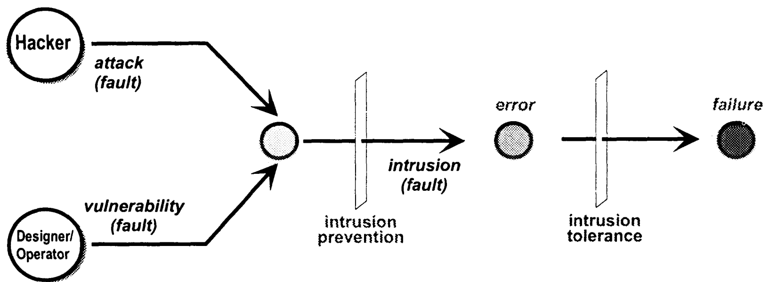

Figure 16.1. Vulnerability, Attack and Intrusion

The effects of intrusions, depending on the intention (and the skill) of the attackers, can take several forms, such as: interception and fraud in an electronic payment or home banking system; forging of electronic banking transfers; penetration of a company's information system for industrial espionage; or of a hospital's database, to disclose sensitive information about patients or use it for blackmail; destruction of a government's database, by a group of radicals; "bombarding" of a company's commercial server, in order to bring it down, for sabotage; fraud in the GSM mobile phone system, for free calling; breaking of an electronic wallet system, for counterfeiting digital money; and so forth.

Either the level of threat or the degree of vulnerability alone do not measure how secure a system is. In fact, consider the following two example systems, system Vault and system Sieve, whose degree of vulnerability and level of threat we have quantified for the sake of example. System Vault has a degree of vulnerability $v_{vault} = 0.1$, and since it has such high resilience, its designers have put it to serve anonymous requests in the Internet, with no control whatsoever to whoever tries to access it, that is, subject to a high level of threat, $t_{vault} = 100$.

System Sieve, on the other hand, is a vulnerable system, $v_{sieve} = 10$, and in consequence, its designers have safeguarded it, installing it behind a firewall, and controlling the accesses made to it, in what can be translated to a level of threat of $t_{sieve} = 1$. Now consider the product *threat* × *vulnerability*: with the imaginary values we attributed to each system, it equates to the same value in both systems (10), although system Vault is a hundred times less vulnerable than system Sieve. The correct measure of how secure a system is depends as much on the number and severity of the flaws of the system (vulnerability) as on the potential of the attacks it may be subjected to (threat). This measure is called risk, and it is very important since it explains that one cannot ascertain the level of security of system or product from the catalogue alone.

Risk - combined measure of the level of threat to which a computing or communication system is exposed, and of the degree of vulnerability it possesses

16.1.3 Security Properties

We have discussed until now what makes systems insecure, how they can be attacked, and what is the effect of insecurity. Now, we are ready to understand what we would like of a system, in order to consider it secure. We would like it to preserve our information from the eyes of intruders. We would like to be sure that whoever is dialoguing with us at the other end of the line is really who she says she is. We would like to avoid that someone changes or destroys our information. Finally, we would like to avoid that someone brings our system down on purpose. We may not always need all these attributes simultaneously, and some of them are not even achievable with the same techniques. These are good reasons for understanding as precisely as possible what we mean by security:

Confidentiality	the measure in which a service or piece of information is protected from unauthorized disclosure
Integrity	the measure in which a service or piece of information is protected from illegitimate and/or undetected modification
Authenticity	the measure in which a service or piece of information is genuine, and thus protected from personification or forgery
Availability	the measure in which a service or piece of information is protected from denial of authorized provision or access

Confidentiality is concerned with protecting information and computing and communication services from the eyes of intruders. Privacy is the default form of confidentiality for private information and communication. The simplest approach to ensure confidentiality is physical isolation, for example, locking a computer, or using networks which make tapping difficult, such as fiber optic media. However, this is not always possible or convenient, so we are going to study methods based on *encryption*, which ensures that although the intruder has access to the information, it is unintelligible to him.

Authenticity is concerned with guaranteeing the origin of a service request, a piece of data or a message, or the identity of a service provider or the creator of a piece of information. Intruders may wish to pretend they are someone else, as part of the attack on a system, for example for getting access to other people's information, or to forge the identity of the creator of a document. This is avoided by *authentication*. Malicious users may also pretend they did not do something they did do, for example, denying they signed a check with which they paid some goods. Avoiding it is called *non-repudiation*. For a piece of data, authenticity is recognizing the creator's signature, even if he denies. For a recipient, authenticity is being certain that a message signed by a sender was really sent by him, and that the sender cannot deny to have sent it. Key to ensuring authenticity is a technique that we will study called *digital signature*. Authenticity is not considered as a first order property by several authors, and is defined by some as being the 'integrity' of some meta-information concerning the identity of the object.

Integrity is concerned with avoiding or detecting the modification of information or messages, including service interactions, with malicious intent. The easiest way to secure integrity is to use some form of checksum, although with cryptographic resilience, to detect modification. We will study techniques to perform these checks based on *secure hashes* and *message digests*. However, sometimes we also require prevention of modification, if we fear a radical attack trying to erase our whole information system. Short of using redundancy techniques such as the ones we studied in the Fault Tolerance part of this book, the best solution, and one widely used in security, is not letting the intruders get anywhere near the information. We will also study *protection* techniques based on *access control*.

Availability is concerned with ensuring that information and computing or communication services remain accessible to authorized users, despite what are normally called denial-of-service attacks. These attacks may be performed for reasons such as sabotage, vandalism, terrorism and politics. Availability, such as integrity, can be procured with techniques based on protection. However, static access control has very little effect on denial-of-service attacks, as recognized in (Lampson, 1993). The reason is that an attack may come by the same channels as authorized users, such as bombing a Web server with legitimate requests in order to bring it down. Reactive (dynamic) access control based on intrusion detection, such as yielded by some firewall systems, may help mitigate the problem, for example by selectively blocking requests originating from a suspicious machine.

Availability may also be achieved by means of techniques based on redundancy management, that is, fault tolerance. The idea is to replicate the service, possibly in several places, making the intruder's life more difficult, since he has to attack all replicas in order to bring the service down. However, these techniques are not such a definite solution as they are for accidental faults. The reason is that for most of the latter it is possible to define a *fault model*, bounded both in type of behavior and maximum number of faults supposed to occur, and

build systems that can provably be available while that fault model holds. In the case of malicious intentional faults, we do not know yet how to put a bound on the type and number of attacks, since they are human-driven and depend on non-technological factors such as the persistence, time, and intelligence of a hacker[1].

16.1.4 Evolution of Secure Computing

Security is a very old activity. Before computers, people would already use ciphers to send secret messages. Among the first, we can count the **Caesar cipher**, used by the Romans. It was a *substitution* cipher, which worked by replacing each letter of the message with the letter 3 positions ahead in the alphabet, wrapping around from Z to A. Very naive, it worked well since there were not many cryptanalysts around at that time. A *transposition* or *permutation* cipher is yet another technique that works by shuffling the order of letters. A common transposition method would be to write the plaintext line by line, in a sheet of fixed character length per line, and then get the ciphertext by reading it column by column. Doing this twice would significantly increase robustness. It was reportedly used in World War II by the resistance, who would encode and decode their messages by hand.

Shannon launched the basis for cryptography, enunciating two fundamental principles: *confusion* and *diffusion*. Confusion is the property whereby it becomes extremely difficult for an intruder to find out the relationship between the plaintext and the ciphertext. The rationale behind it is to eliminate redundancies and statistical patterns. Substitution generally contributes well to create confusion. Diffusion is the property that dissipates the information patterns throughout the text, so that a single bit change should reflect itself in many places of the ciphertext. Current systems are a mix of the application of these two principles.

World War II brought the intensive use automated encryption/decryption, by means of mechanical devices, or **rotor machines**, such as the Enigma machine of the Germans. The principle of operation was based on substitutions, and permutations implemented by a mechanical wheel. One machine might have several rotors, the output of one wired to the input of the next. The Enigma was broken by the Allies, using computing resources. This is a historical example of a principle that is still true and haunts every cryptographic system: no matter how good a system is, it takes every year less time to break it by brute-force. Very important milestones for secure computing, still valid today, were erected after the war. We enumerate a few in Table 16.1.

The public key principle was one of the most important discoveries in cryptography, generally attributed to Diffie and Hellman. Merkle did contemporary work that also ranks him among the asymmetric cryptography pioneers. A recent announcement credits John Ellis, a cryptographer working for the British

[1] In fact, this may also prove difficult for software design faults.

Table 16.1. Major Milestones in Secure Computing

1972	Reference monitor protection model (Lampson, 1974; Anderson, 1972)
1973	BeLa Formal security model (Bell and LaPadula, 1973)
1975	DES- First widely used symmetric crypto algorithm (DES, 1977)
1976	Public key cryptography principle (Diffie and Hellman, 1976)
1978	RSA- First widely used asymmetric crypto algorithm (Rivest et al., 1978)
1978	Mediated authentication/key distribution (Needham and Schroeder, 1978)
1982	Blind signature for non-traceable digital cash (Chaum, 1983)
1988	Kerberos- Widely used KDC authentication service (Steiner et al., 1988)
1990	PGP- Public domain strong cryptography system (Zimmermann, 1995)
1994	ECash- First commercial non-traceable digital cash sys. (DigiCash, 1994)

government, for the invention of the public key cryptography principle a few years before Diffie and Hellman. This work was kept secret and never published until now.

Security in informatics[2] has long left the exclusive realm of the military and governmental institutions. Nowadays, networking and distribution on open systems have become the major paradigms for supporting information processing. Confronted with the need to work under the level of threat of environments such as the Internet, designers, vendors and users have become aware that the way to go can no longer be by reducing the threats to systems. Instead, the design of systems must be improved and adequate techniques incorporated, in order to reduce their vulnerabilities and achieve an acceptable risk of operation. This is why expressions such as 'public key cryptography', 'DES', 'digital cash', have become vox populi.

Generalized use of cryptography has made some governments nervous about the possibility of people having completely confidential communications and files, unlike what exists today with telephone communications and hard paper files, that can be disclosed under a warrant. In trying to preserve this metaphor, **key escrow** encryption systems were proposed. The system is a normal cryptographic system, except that the key is also deposited with a government agency. In special cases, namely a suspicion of crime, the key can be obtained from the agency by the competent authorities, which are then able to tap communications or decrypt files.

The use of cryptography has few *restrictions* in most countries, with the notable exceptions of France in Europe, and the prohibition of exporting strong cryptography products in the U.S.A. (Koops, 1999), partially lifted in the end of 1999. The near future will witness the increment of awareness by private users about the security risks of open distributed computing. This will give a great push for the proliferation of commercial systems incorporating cryptography.

[2] "Informatics" is a word of european origin getting increased acceptance in the community of computer users and developers. It is used to denote in general terms all that is related with use of computers and networks in information processing, access and manipulation.

Most probably, it will contribute to an evolution of the attitude of authorities with regard to security.

16.1.5 Distribution and Security

One of the oldest rules of security is to distribute the power and the knowledge about crucial issues, so that no single person can control and misuse that power or knowledge. We all remember the tales about the doors with three locks, and the secret map that was cut into several pieces. With the advent of computers, and since they were essentially centralized machines, the computer, as a unit of intrusion, became a single point of failure. Since it was not convenient to tear a mainframe in pieces, that ability of splitting the control of a secret was lost for a while, until distributed systems came into play.

Distributed systems, on the other hand, have presented some shortcomings in the transition from centralized ones. First, instead of being held in a central physical point, critical data was spread by several sites. Secondly, these sites were forced to exchange information through networks whose physical access cannot be controlled completely. These are detrimental factors to the baseline security of distributed systems.

However, distributed systems bring back the possibility of distributing control and knowledge. The fact that data reside in several sites may be used in one's favor, if spread in a way that the intruder must break into several sites to retrieve useful information. Cryptography has taught us how to securely send information over insecure networks. In conclusion, with the adequate techniques, distribution presents the systems architect with a powerful framework to achieve very high levels of security, and materialize back the metaphors with which we started this section.

16.1.6 Fault Tolerance and Security

Dependability is the justifiable reliance on the operation of a system. From that viewpoint, achieving reliance on the presence of accidental or of malicious faults are two faces of a same coin. Vulnerabilities of the system are non-malicious design or configuration faults that the intruder exploits to induce other faults of malicious nature, or attacks. This combination aims at achieving an intrusion on the system. The resulting erroneous state, if not handled, may lead to a security failure of the service that the system is supposed to provide (e.g., communication confidentiality, database integrity, etc.). Besides the conceptual analogy, there is a chance for using techniques of either field in a complementary way. In essence, intrusion detection and recovery, or intrusion masking, can be seen as techniques for achieving *intrusion tolerance*. Some fault-tolerant protocols use cryptographic signatures as a means for tolerating value faults. Byzantine agreement techniques, used in ultra-dependable systems to mask arbitrary errors, also have applicability in security.

16.2 WHAT MOTIVATES THE INTRUDER

As we have seen, behind the several possible security-related failures in a computer system, there is a human brain. It is difficult to establish a regular model for malicious faults, as we have done for example with accidental faults when studying dependability (*see Fault-Tolerant Computing* in Chapter 6). It is also not obvious how we award probability distributions to human decisions, for example, it would be surprising to discover that hackers' successful attempts at installing exploits in an operating system follow a Poisson curve. Short of such a formal framework, we may nevertheless try to understand what motivates the intruder: the *hacker* of computer systems, or the *phreaker* of telecommunication systems, in security lingo.

The first interesting thing to learn about intruders is the "why". The motivations of hackers vary: curiosity; collecting trophies; free access to computational and communication resources; bridging to other machines in a distributed system; damaging or sabotaging systems, for criminal, mercenary or political reasons; stealing information for own use or for sale, such as software, commercial or industrial secrets, etc. The "how" depends on the hacker's ability and on the system to be penetrated. Generally, he takes the following steps:

Exploration of vulnerabilities	Finding weak points in the computer system (e.g., accounts without password; vulnerable configurations or drivers present; accessible password file)
Access to the system	Making a plan of attack (e.g., matching dictionary words to password file entries; attack system driver with malicious program to get an account; eavesdrop on login/password pairs on the network)
Control of the system	Controlling all system resources, by becoming *root* user after attacking the system's security mechanisms with malicious programs (e.g., malicious script securing root access; racing a penetration program against a legitimate system call)
Deletion of traces	Concealing activity during intrusion campaign (e.g., disguising himself while logged in, erasing system logs after logout)
Continued stealth access	Perpetuating access to the machine in a stealthy manner (e.g., Trojan horses or backdoor programs that can be activated by special codes or sequences, abandoned accounts)
Exploration of new targets	Looking for new trophies (e.g., channels leading to other machines; access information in personal files; eavesdropping from the intruded machine)

Talking about the tools used, it is typical to find a hacker sitting on a good PC, behind a fast modem. Many hackers use blue-boxes and other "colored" boxes, which are devices to confuse the billing electronics of telephones and allow the hackers to call for free. "Software tools", in the form of attack programs, called "exploits" in hacker lingo, abound in the Internet, exchanged in Internet Relay Chat systems (IRC) or published in Web pages.

16.3 SECURE NETWORKS

Security in networks starts with physical protection. One of the major threats today is eavesdropping on networks. Technology can give a hand here, since for example, fiber optic cables are very difficult to tap without notice and in practical terms they do not radiate.

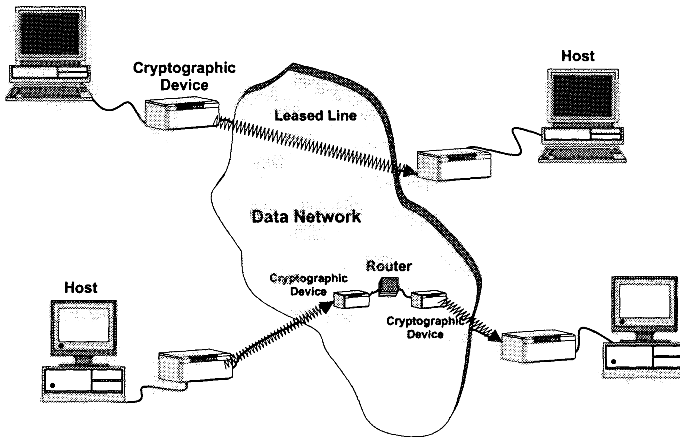

Figure 16.2. Physical Circuit Encryption

The latter attribute is important, since today it is possible to detect energy radiated by the passing bits on network cables, at considerable distances. However, while measures can be taken inside premises to make physical tapping difficult, wiretapping in large-scale networks is almost impossible to avoid. In consequence, we had better look elsewhere for network security: communication encryption. There are different levels at which encryption can be performed, in terms of the OSI model. Encryption can be done at the lowest layers, Physical and Data Link, or the higher layers, such as Network (e.g., Internet IP), Transport, Presentation, Application, or even by the user.

Physical circuit encryption, also called link encryption, is adopted when the idea is to encrypt all traffic passing through a link between two points. As shown in Figure 16.2, a physical link is connected by a pair of cryptographic devices, and all cleartext (non-encrypted data) going through that route is encrypted, that is, converted to ciphertext (denoted by zigzag lines), which is decrypted by the corresponding device at the other end. Encryption is done at the Physical or Data Link layers, either in software or in hardware, but this approach is most convenient for hardware-based encryption. As shown in the top of Figure 16.2, this approach is specially suited for linking hosts via long-haul leased lines. Physical circuit encryption applies strictly to a link between two points. In consequence, if there are routers in the middle, then the data has to be decrypted, so that the router can analyze the routing information, and then routed via another link, where it is encrypted again, as shown in the bottom of the figure. Typically, each pair of devices over a link shares one key.

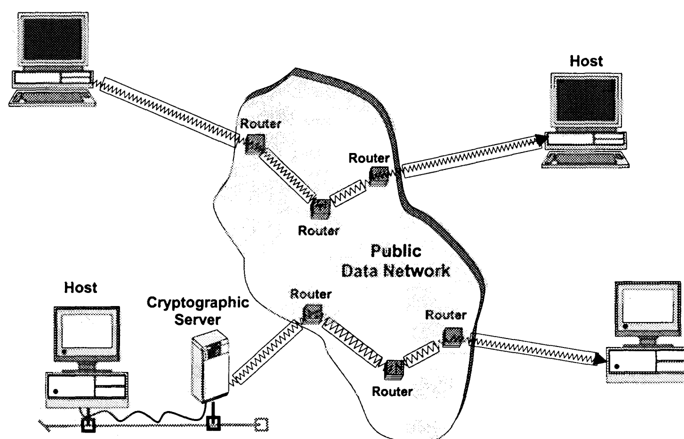

Figure 16.3. Virtual Circuit Encryption

Physical circuit encryption requires a large number of encryption devices, and a large number of encryption/decryption operations in routes with many hops. Alternatively, one can resort to *virtual circuit encryption*, also called end-to-end encryption, where encryption is performed on a per traffic flow basis, at the higher layers of the architecture. Not only can encryption be selective, because of the multiplexing existing at these layers, but it can also be preserved until the final destination. This is because the headers of the lower communication layers are appended *to* the encrypted data, instead of being embedded *in* it, like it is done in physical circuit encryption. This is exemplified in the top of Figure 16.3, where the encapsulation of encrypted data is shown by a rectangle enveloping the zigzag line. Encapsulation is removed and inserted again in the course of normal processing of communication functions at each hop, but the zigzag line remains intact. For example, encryption can be done at the Network or Transport layers, and only for certain destination domains, addresses or ports. The Network protocol header (e.g., IP) is inserted after encryption, and in consequence, the data can remain encrypted throughout its way. Virtual circuit encryption can also be performed higher up, such as the Presentation or Application layers, or even by the user process itself. Email communication, for example, has provisions for encryption. Virtual circuit encryption is the most used approach, and is normally performed in software. However, nothing prevents it from being performed in hardware between the user and the host, as is the case with applications relying on intelligent smart cards. Physical and virtual circuit encryption can be combined in order to achieve higher levels of security. The principle exemplified in Figure 16.3 can be used to build an encrypted link between two facilities of a same organization. The respective endpoints are connected by a cryptographic stream encapsulated in a network protocol, with the sole purpose of carrying (tunneling) the data through the network between these two endpoints. This principle can be extended to sev-

eral endpoints in a pair-wise manner, to create what are called **virtual private networks**, or VPNs, in a secure way.

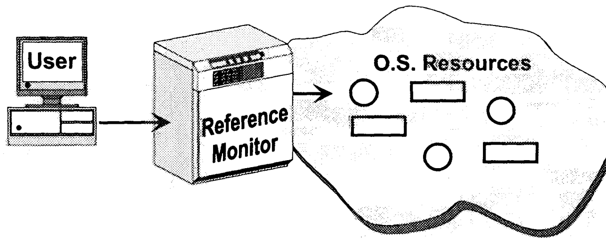

Figure 16.4. Reference Monitor

16.4 SECURE DISTRIBUTED ARCHITECTURES

This section gives an overview of the main architectures for distributed systems security. Conventional operating systems have protection mechanisms based on facilities implemented in hardware by the underlying microprocessors. Normally these provide for a number of layers of privilege for executing instructions and accessing resources, and also virtual and separate address spaces, or even full virtual machines executing in complete isolation over the same hardware (Tanenbaum, 1992). Operating systems then offer a higher-level of protection centered in their resources, such as files and network devices. Users have different rights of access to different resources, and may even have private resources.

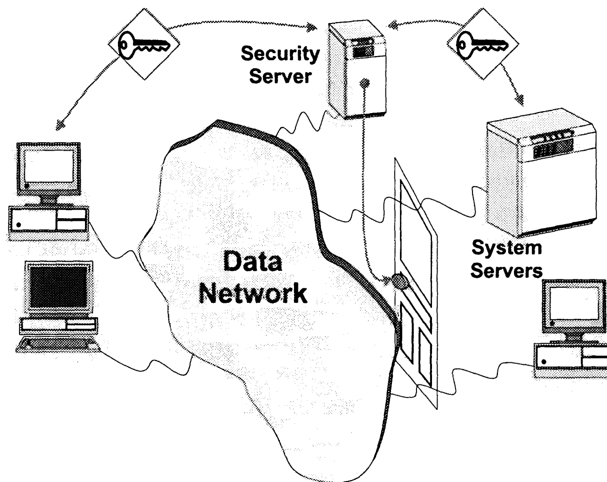

Figure 16.5. Security Server

In secure systems, the concern about protection is taken one step further: it is delegated on an entity, called *reference monitor*, such that all requests to

protected resources are addressed for authorization to that entity (Figure 16.4). The reference monitor is normally a secure part of the operating system kernel. In distributed systems, an *authentication and authorization server*, or *security server*, must perform that task for interactions between machines accessing resources that are distributed in a network. As a consequence, the gatekeeper function of the reference monitor is done in a virtual and distributed way, as illustrated in Figure 16.5. Although all machines we see may be physically interconnected by the same network, logically the clients on the left side can only access the system servers on the right side after granted permission by the security server. They have to authenticate themselves and get an authorization to access a given resource. The reason why this "virtual gate" works is that this dialog is cryptographic, and the authorizations take the form of *cryptographic credentials* that the system servers analyze.

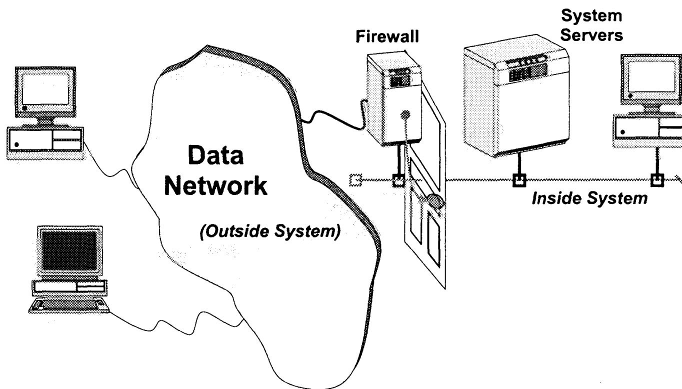

Figure 16.6. Firewall

A simpler form of protection that in a way extends the principle of the reference monitor for centralized systems is the *firewall.* It acts by physically interposing a barrier between an outside system and an inside system. Although the protected inside system may be a distributed system, for the firewall it is a hard-core in terms of security, that is, a *security perimeter* (all the bad guys are out and the good guys are in). In consequence, all requests coming from the outside system to resources in the inside system must pass through the firewall, as depicted in Figure 16.6. This model is weaker than the reference monitor model, since its access rules are less precise and normally directed to communication protocols (block or allow traffic to given ports, addresses, protocols, etc.). A special form of firewall function called *application gateway* can implement more sophisticated protection, but in an application-dependent form. On the other hand, in terms of network traffic, the firewall is more versatile, since it also allows protection to traffic from the inside to the outside.

Implementing secure remote operations is fundamental in open systems. The problem, as exemplified in the top of Figure 16.7, is achieving security of interactions between a client terminal and a server. A **sniffer**, a machine that

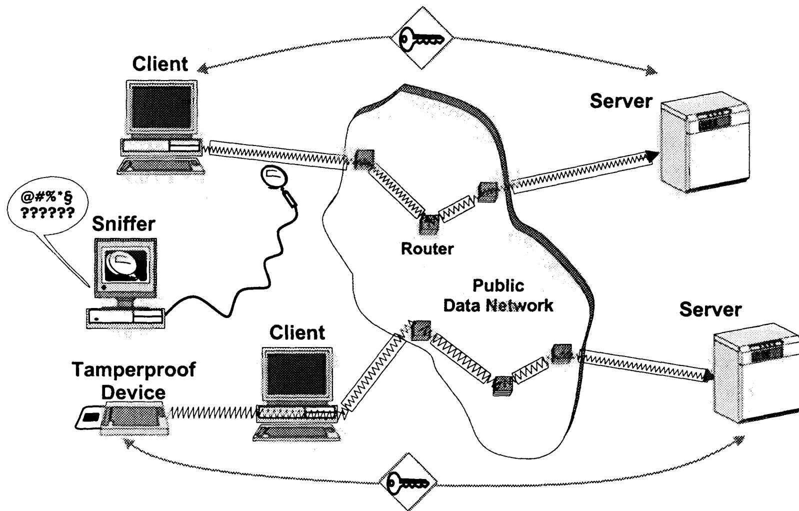

Figure 16.7. Secure Remote Operations: Normal, and with Tamperproof Devices

listens to everything that passes on a network, may read all the information, including passwords and sensitive information, for later misuse. As we see in the figure, secure remote operation architectures include some form of virtual circuit encryption (*see* Figure 16.3), which encrypts either all the communication, or just sensitive parts, in an end-to-end manner, so that eavesdropping does not succeed. These architectures are used in a variety of situations over insecure environments like the Internet, such as plain remote logins, client-server computations, Web HTTP server accesses, or electronic commerce. In this latter application, the level of security may be increased if, as exemplified in the bottom of Figure 16.7, the cryptographic channel is established between the server and a **tamperproof device** representing the user. The latter represents a good solution when the client machine is not trusted. These devices are for example: boxes with secure micro-controllers that read a card and execute cryptographic protocols, or intelligent smart cards themselves able to execute cryptographic protocols.

There are several variants of electronic payment architectures. The most promising are the ones centered around digital cash. As Figure 16.8 exemplifies, the relevant parts of the architecture are: a banking network; and tamperproof devices, also called wallets or purses. Digital cash is generated by the client's bank and loaded into the client's smart card (a tamperproof device of its own right) by means of a cryptographic protocol ran between the smart card and the bank server. This can be done in an ATM machine, for example. What is interesting in this architecture, in contrast to the ones presented before, is that not everything happens inside a connected network. The client takes her card to a merchant and pays completely off-line, again by running a cryptographic protocol that unloads digital cash to the merchant's terminal,

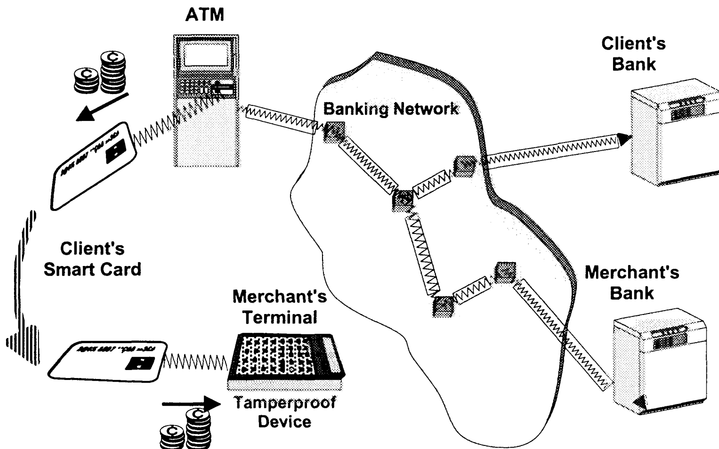

Figure 16.8. Electronic Payment

a tamperproof device that is not connected permanently as well. Later, the merchant deposits the money in his bank. The role of the banking network is to consolidate these electronic transactions, making the money flow between the several participants— in this example, from the client's account to the merchant's account.

16.5 SUMMARY AND FURTHER READING

This introductory chapter discussed the fundamental concepts concerning security, and introduced terms such as: vulnerability, threat, intrusion, security hazard, confidentiality, integrity, authenticity, availability, sniffer, physical and virtual circuit encryption, secure tunnel, reference monitor, security server, firewall, tamperproof device. During the following chapters, we will discuss them in greater depth. For more introductory level material, the reader should see the introductions of (Pfleeger, 1996; Kaufman et al., 1995). Pfleeger also does an extensive discussion on legal and ethic issues (Pfleeger, 1996). Kaufman et al. (Kaufman et al., 1995) and Abrams et al. (Abrams et al., 1995) give fairly complete glossaries of security terms. Stallings provides an interesting reading on security in networked and distributed systems (Stallings, 1999).

17 SECURITY PARADIGMS

This chapter discusses the main paradigms concerning security. An exhaustive description of all the major algorithms concerned would require a whole book's length in order not to be shallow. Instead, we will motivate each paradigm in a problem-solving manner, exemplified when applicable with one or two chosen algorithms that are analyzed with the necessary detail. Throughout the chapter, we are going to discuss: trusted computing bases, cryptography, digital signature, digital cash, authentication, protection, and secure communication.

17.1 TRUSTED COMPUTING BASE

Most of the paradigms to achieve security that we describe below require some form of computation. One might ask *where* is this computation supposed to take place, in order for it to be immune to hackers in the first place. The **Trusted Computing Base (TCB)** is the paradigm whereby it is possible to build a computing nucleus that is immune to intrusion, even if submersed in hackers attacks, or even if incorporated in a system built by hackers. A few other building blocks are based on its existence: reference monitors, firewalls, and authentication servers.

17.1.1 Specification of a TCB

A Trusted Computing Base (TCB) is that part of the system, comprising hardware, firmware and software, which is responsible for supporting the security

P. Verissimo et al. *Distributed Systems for System*
© Kluwer Academic 2001

mechanisms used to protect the system, such as authentication, access control, auditing. The specification of a TCB must secure a few desirable properties:

Interposition	the TCB position is such that no direct access to protected resources can be made bypassing the TCB
Shielding	the TCB construction is such that it itself is protected from unauthorized access
Validation	the TCB functionality is such that it allows the implementation of verifiable security policies

The TCB is a subset of the operating system, it has total control of the hardware it runs on, and no other software runs in a more privileged level. In consequence, *Interposition* means that nothing can detour the TCB to access system resources. Note that we are not saying that everything *has to go* through the TCB, but that the access to any resource *can* be controlled by the TCB, so that it can implement whatever functions necessary to control access. These functions depend on what sort of *security policy* is being enforced by mechanisms built on top of the TCB, what sort of attacks it is trying to resist, etc.

Shielding means that nothing should intrude the TCB. By construction, the TCB must be impervious to accesses to its own structures, other than by the authorized users, normally the *security administrators*. Again, all depends on the proper design of the TCB.

Finally, *Validation* means that the TCB functionality should be simple enough in order for the security mechanisms materializing the security policy in the computer system to be formally specifiable and verifiable.

The above-mentioned properties do not imply that a TCB cannot be fooled into doing wrong things: you should note the *separation of concerns* between the enabling architecture for protection (the TCB) and the protection mechanisms themselves (authentication, authorization, auditing). If the latter are improperly designed or else follow an improper security policy (or the absence of one), then security hazards can happen in spite of a correct TCB.

17.2 BASIC CRYPTOGRAPHY

Cryptography is a mandatory paradigm in security. Modern computer cryptography relies on a *cryptographic algorithm* or **cipher**, and a **key** or pair of related keys. The original data, *cleartext* or *plaintext*, passes through the algorithm and generates *ciphertext*. This is called **encryption**, and it is a function of an encryption key. The role of the key is to parameterize the algorithm, such that for the same cleartext, encrypting it with two even slightly different keys yields drastically different ciphertexts. The reverse operation, called **decryption**, recovers the original cleartext from the ciphertext, by running the algorithm again, with a decryption key. In some systems, the encryption and decryption keys are equal.

We will use the following notation: $E_{K1_a}(M)$ to mean "encrypt M with encryption key $K1$ related to a" (a may denote the key owner, or a session, or a message). Similarly for decryption: $D_{K2_a}(C)$, where $K2$ is the decryption key. If there is no ambiguity, we will omit indices, for example $E_{K_a}(M)$. As in all security textbooks, we will use the classic players for our metaphors: Alice and Bob, the good guys, helped when needed by Carol and Dave. Mallory and Eve, the bad guys. Trent, a mediator, or trusted third party. It is usual to use **principal** to designate any *participant* in a security protocol. We will use both words interchangeably.

Table 17.1. Basic Attributes of a Cryptosystem

- given a pair of encryption/decryption keys $K1$ and $K2$, if $E_{K1}(M) = C$, then $D_{K2}(C) = M$ and thus $D_{K2}(E_{K1}(M)) = M$
- given $E_{K1}(M)$, without $K2$ it is infeasible to recover M
- given M and $E_{K1}(M)$, it is infeasible to recover $K1$
- given $K1$, it is infeasible to recover $K2$ and vice-versa
- if keys are equal, $K1 = K2 = K$, first three statements still apply

K1=K2=K, the cryptosystem is symmetric and the above statements still apply . if K1=/=K2, the cryptosystem is assymmetric, and given K1 it is infeasible to obtain K2, and vice-versa.

The strength of the **cryptosystem** defined as above lies not on the algorithm's secrecy, but on the secrecy and quality of the keys. The algorithm can and should be public domain. A proprietary secret algorithm does not give users any guarantees about its robustness or seriousness, since the principles on which it relies are not known. Furthermore, if put in public domain, it will be exhaustively tested by the academic community, so that as years go by, either it ends up being broken (we say it is cryptanalyzed) or the confidence in it increases. However, the algorithm must be such that if an intruder does not have the key, he cannot obtain useful information from the cyphertext. In fact, this can be put in the form of more concrete attributes, listed in Table 17.1.

Infeasible does not mean impossible. Actually, by testing all possible combinations of keys, plaintexts, ciphertexts, etc., one may impair some of the premises above. This is a **brute-force attack**, and keys should be made long enough that the computational cost and/or time involved make this attack impractical. A word of caution about the cryptographic protocols based on an algorithm: the way the algorithm is used should not introduce vulnerabilities. Unfortunately, this may sometimes happen.

There are two major classes of algorithms in modern cryptography: symmetric and asymmetric. The names derive from the fact that the former relies on a secret key used in both operations of encryption and decryption, whereas the latter relies on two different keys, one used for encryption and the other for decryption. Another relevant building block is secure hashing, which has the

property that it cannot be tampered with, and interesting uses, such as creating unique "fingerprints" of texts or messages. In what follows, we discuss these three building blocks with more detail, emphasizing the aspects that matter to a systems architect. For a deeper study of cryptography, the reader should refer to the remarkably clear and complete book of Schneier (Schneier, 1996).

17.3 SYMMETRIC CRYPTOGRAPHY

Symmetric cryptography is characterized by the fact that there is only one key, which is kept secret, and instances of it have to be shared between both ends of a channel. In general: $K1 = K2 = K$, and thus if $E_K(M) = C$, then $D_K(C) = M$ and thus $D_K(E_K(M)) = M$. If a key is shared by two participants, like A and B depicted in Figure 17.1, it is also usual to denote it by K_{ab}.

The security of this approach relies on K being kept secret by both participants. This presents several problems. One is of self-containment: a new key must go through some secure channel to both participants (or at least from one to the other). Practical solutions involve alternative channels such as person-to-person or paper mail. However, the latter are not interesting if keys need to be exchanged frequently, in which case fast and secure means to distribute and exchange keys must be used. For example, by combining cryptosystems, we can use one to deliver the keys of the other. The other problem is of robustness: a key compromise at either end compromises the whole channel. A third problem is related with scale: key management becomes worrying for large systems. For example, since one key is needed for each pair of participants, in order to support arbitrary communication among 10 participants 45 keys are required, and this number goes up to almost 5000, for 100 participants. On the bright side, symmetric encryption algorithms are relatively fast.

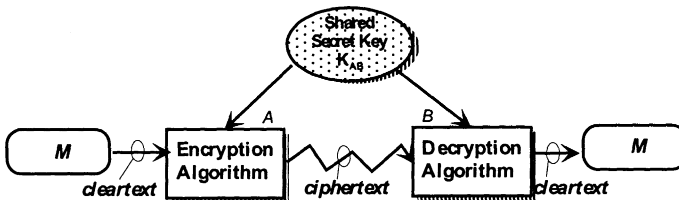

Figure 17.1. Symmetric Cryptography

There are two types of symmetric ciphers: *block ciphers* and *stream ciphers*. The block algorithms encrypt one block of data at a time (e.g., 64 bits). The stream algorithms process the cleartext one bit (or byte) at a time. A widely used block cipher is *DES*, the *Data Encryption Standard* (DES, 1977).

17.3.1 Data Encryption Standard

The DES was developed for the U.S. government and standardized in 1977. The DES algorithm is based on an iterative application of substitution and permutation functions, for 16 cycles. Substitutions achieve Shannon's paradigm of *confusion*, whereas the permutations, or transpositions, provide for another Shannon paradigm, *diffusion*.

The algorithm is quite simple and elegant. It is a block cipher, of 64-bit length. The key is 56 bits long. It works as outlined in Figure 17.2. The input is permuted initially. Then 16 equal iterations follow. Each round receives the result of the previous as input. The block is divided in two halves, one of the halves (R_{i-1}) is combined with a 48-bit sub-key generated from the 56-bit DES key (one different sub-key is generated per round), and then XORed with the other half (L_{i-1}), yielding R_i. R_i is concatenated with L_i (equal to R_{i-1}, giving the 64-bit result of the round. After the 16 rounds, a final permutation is performed, which is the inverse of the initial permutation. An interesting property of DES yields a very simple decryption procedure: if the ciphertext is run through DES in the same way as encryption is done, with the same key, the cleartext is obtained. The only condition is that the sub-keys are used in the reverse order.

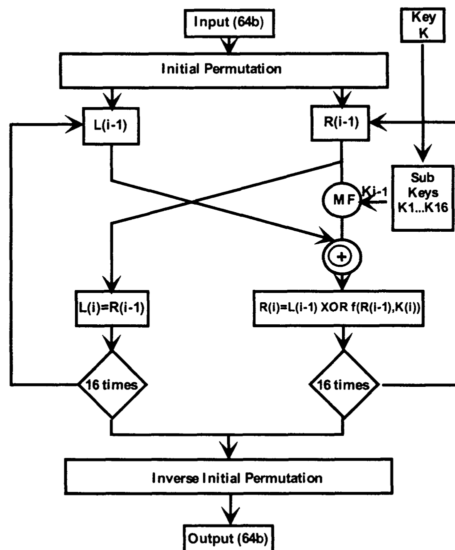

Figure 17.2. DES - Data Encryption Standard

DES is quite fast, and specially amenable to hardware implementations. It is one of the most used algorithms today, and no fundamental weakness was discovered so far. However, the 2^{56} search space of the 56-bit key length is a current source of worry. It is believed to have been chosen because it corresponded, around 1977, to a computational power within the reach of the

U.S. government security agency (NSA), but no one else, to break the cipher by brute force. However, advances in computational power have currently placed that power in the hands of too many organizations and people. Despite this proviso, DES is a very robust algorithm, if used in the adequate mode.

17.3.2 One-Time Pads

A *one-time pad* is the best representative of the stream cipher type of symmetric encryption. It is the only really unbreakable cipher. It is based on having a truly random, never-ending sequence of characters, bytes or bits, depending on what we want to encrypt, which we combine, one-by-one, with our plaintext stream. The original one-time pad idea applied to characters (Kahn, 1967): the pad was a stripe of truly random characters that were added modulo 26 to the plaintext stream of characters. In computers, the pad is binary, and it is XORed with the plaintext in transmission. In reception, it is XORed again with the ciphertext, yielding the original text. Since the pad is random and used only once, there is no information for the cryptanalist to withdraw. One-time pads can be a very useful building block, so they deserve a few comments:

- security relies on the secrecy of the pad, which must, as with any symmetric cipher, be distributed to both ends of the channel;
- security relies on the randomness and uniqueness of the pad: non-random sequences and reuse introduce weaknesses;

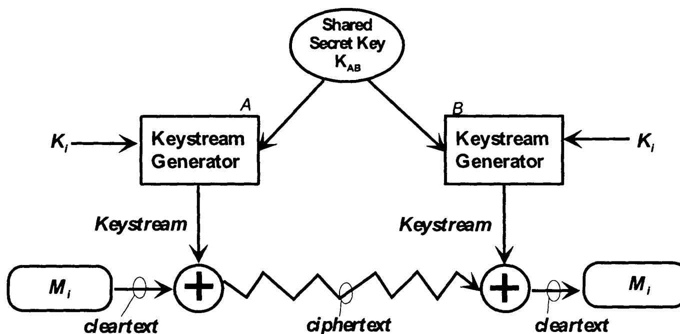

Figure 17.3. Stream Cipher

Real-life one-time pads are only possible if generated and distributed in advance. This has its uses, but for encryption of computer communication links, stream ciphers must be produced in real time, in the way shown in Figure 17.3. Hardware box A produces a stream (called *keystream*) that *looks* like random, and XORs it with the incoming plaintext stream. However, it cannot be random, since box B must produce *exactly* the same sequence, in the same phase, so that, when XORed again with the ciphertext, it recovers the plaintext. Of course, such a device will always produce the same sequence when turned on, becoming susceptible to attacks. In consequence, keystream

generators have keys, and the output is a function of the key, as shown in the figure. Stream ciphers are susceptible to bit errors that desynchronize the ciphertext stream. The quality of such a system is dictated by how much it resembles a random-number generator.

17.4 ASYMMETRIC CRYPTOGRAPHY

In asymmetric cryptography there is a pair of keys, the *public key* and the *private key*. Because of this fact, it is also called **public key cryptography**. Each participant owns such a pair of keys. Only the private key need be secret, the public key is handed away to anybody wishing to send the participant an encrypted message. As depicted in Figure 17.1, participant B gave participant A her key, or published it in a name server (or in a newspaper ad, why not?). A uses B's public key Ku_b to encrypt M and send it. Only the pair of the public key, private key Kr_b, can decrypt the message. In consequence, in our notation: if $E_{Ku_b}(M) = C$, then $D_{Kr_b}(C) = M$ and thus $D_{Kr_b}(E_{Ku_b}(M)) = M$.

The security of this approach relies on Kr being kept secret. This presents two advantages with regard to symmetric cryptography. The first is that there is no need for key exchange with secrecy: every participant generates keys and publishes the public one. The second is that a channel can only be compromised in one end, that is, the end of the key owner. The scale of key management is also better than for symmetric cryptography, since one public key is needed per participant. In consequence, for 100 participants we need only 100 keys. The down side is that asymmetric cryptography is 1000 (HW implementations) to 100 (SW implementations) times slower than symmetric. Key distribution deserves a word of caution: if a public key is not received in first hand from its owner, then it must be ensured that it is authentic. A key repository may be tampered with, and when fetching Alice's key, we might unwittingly be getting Mallory's key, who had in the meantime penetrated the key/name server as part of an attack against our interaction with Alice (*see* more on this in Section 18.6).

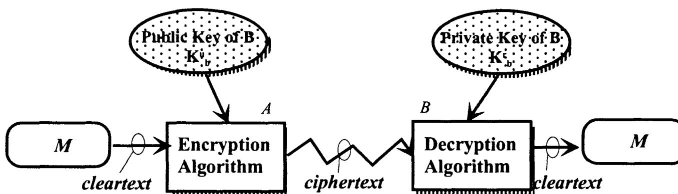

Figure 17.4. Asymmetric Cryptography

17.4.1 Diffie-Hellman

The first asymmetric algorithm published was the secret number computationDiffie-Hellman algorithm (Diffie and Hellman, 1976). It owes its security to the difficulty of calculating discrete logarithms to invert exponentiations in a finite field, or the difficulty of factoring numbers that result from the product

of large primes. The objective is to arrive at a shared secret number without ever passing it over a communication medium. As depicted in Figure 17.5, Alice (A) and Bob (B) only do public communication. To arrive at the shared secret number K, they execute the same procedure. They both agree on public numbers n, a large prime (e.g., 512 bits), and m, which can be small, and whose properties with regard to n are omitted here (*see* (Diffie and Hellman, 1976) or (Schneier, 1996) for details). Then, Alice generates a *secret* large random number x_a, performs $y_a = m^{x_a} \bmod n$, and sends y_a to Bob. Bob does the analogous computation, using x_b and sending y_b to Alice. Finally, they both compute the same number K, since

$$K = y_b^{x_a} \mod n = y_a^{x_b} \mod n = m^{x_b \cdot x_a} \mod n$$

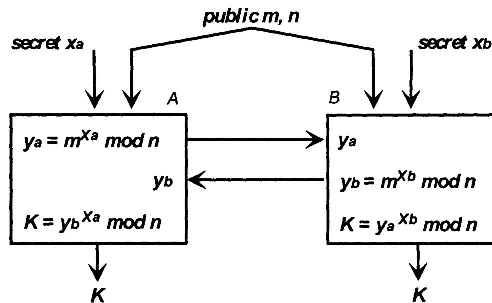

Figure 17.5. Diffie-Hellman

Diffie-Hellman does neither encryption nor authentication. D-H's obvious utility is to create shared secret keys to be used in symmetric cryptography (*see* more on this in Sections 17 and 17.11), and it is very effective at doing it.

17.4.2 RSA

RSA, published in 1978, owes the name to its inventors, Rivest, Shamir and Adleman (Rivest et al., 1978). It was the first widely used asymmetric encryption algorithm. Its security relies on the *trapdoor one-way function* concept, that is, a function that is not reversible (one-way) unless a secret is known (trapdoor). In the approach followed, this boils down to the difficulty of factoring numbers that result from the product of large primes, such as in the Diffie-Hellman algorithm.

A prior step consists in generating the key pair. The key length is variable (a typical size is 1024 bits). Alice selects two *secret* random large prime numbers p and q, of equal length, and computes $n = p.q$. Then she selects a random encryption key e, such that e and $(p-1)(q-1)$ are relatively prime. Finally, she computes the decryption key $d = 1/e \mod ((p-1)(q-1))$. For the user, $Kr_b = \langle d, n \rangle$ is the *private key*, which must be kept secret, and $Ku_b = \langle e, n \rangle$ is the *public key*, which Alice publishes or sends to people.

RSA is not a block cipher, but cleartext M is divided in blocks of size smaller than n that are encrypted one at a time. The ciphertext is the concatenation of the encrypted blocks. The algorithm itself is quite easy to understand. The encryption and decryption of one block is outlined in Figure 17.6. To encrypt a message for Bob (B), Alice (A) fetches Bob's public key $\langle e, n \rangle$, and performs $c_i = m_i^e \mod n$. Decryption is similar: Bob, using his private key , $\langle d, n \rangle$, computes $m_i = c_i^d \mod n$.

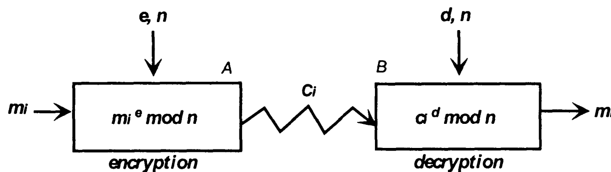

Figure 17.6. RSA– Rivest-Shamir-Adleman

RSA is slow compared to DES, but it is very secure. The variable key length allows it to accompany technology evolution. A 1024-bit key is believed to be extremely secure, given the enormous search space, but nothing prevents anyone from encrypting with a much longer key (and much slower...), say for extremely sensitive documents.

There are no known serious or feasible vulnerabilities to RSA itself. However, protocol flaws such as poorly chosen encoding of messages may open the door to some kinds of attacks. RSA Data Security proposed a standard to address this issue. It is called PKCS (Kaliski, 1993), and it is a set of documents with guidelines for encoding and setting-up structures for using RSA correctly and without vulnerabilities.

17.5 SECURE HASHES AND MESSAGE DIGESTS

Hash functions are compression functions. One-way functions are non-reversible functions. A very useful building block in cryptography is a **one-way hash function**, a function that is easy to compute, compressing a text to a block of fixed length (typically 128 bits), but very difficult to reverse. In that sense, it is also called a *secure hash* or **Message Digest (MD)** algorithm. We use the following terminology: the *digest* or *hash value* of M is $h_m = H(M)$. The security of a message digest lies on the fact that it is not reversible. We are more specific about the necessary properties in Table 17.2.

The first property is helpful for using hashes as representatives of documents without revealing those documents. The second and third are useful in signing texts: they state that a hash uniquely represents a given text, and no one can produce another text that hashes to the same value and say it was the previous text instead. They are also useful to checksum messages, in order to check if they were tampered with: if no one can produce a message that hashes to the same value as the original one, then any changes in a message after a digest was performed are detectable.

Table 17.2. Basic Attributes of a Secure Hash

- given h_m, it is infeasible to recover M such that $h_m = H(M)$
- given M, it is infeasible to find M' such that $H(M) = H(M')$
- it is infeasible to find a pair (M, M') such that $H(M) = H(M')$

Most secure hash algorithms work in the way exemplified in Figure 17.7: they digest a message recursively block by block, using the digest of the previous block as input to the next round. Secure hashes or message digests have a number of uses, for example, to fingerprint a text maintaining its privacy, to compress a text that is going to be signed, or to check the integrity of a block of data.

17.5.1 MD5

MD5, for Message Digest, was invented by Rivest (Rivest, 1992), who produced the whole series of MDi's. It is the direct successor of MD4, and is more secure and a bit slower than its predecessor. MD5 processes the text in 512-bit blocks (64 bytes), and produces a fixed-length output of 128 bits (16 bytes). MD5 operation is exemplified in Figure 17.7. The text is padded to a multiple of 512 bits, with a length field inserted in the pad. The algorithm makes four passes at each block, taking as inputs the 128-bit digest of the previous block and the current 512-bit block, and mangling them in different ways.

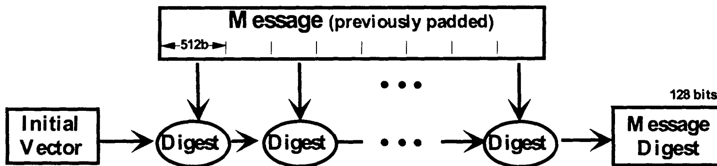

Figure 17.7. Message Digest Algorithm

17.6 DIGITAL SIGNATURE

In this section, we discuss several forms of signatures in latus sensus: message authentication codes, message integrity checks and digital signatures. Let us introduce some terminology: given a pair of keys $K1$ and $K2$ belonging to principal A, **signature** of A is $S_{K1}(M)$ and **verification** of A's signature is $V_{K2}(M)$. When there is no ambiguity, we may use $S_a(M)$ and $V_a(M)$. If a symmetric approach is used, then $K1 = K2 = K$ but the signature process becomes more complex. The obvious utility of signatures is more or less the same as in real life, but let us be a bit more formal and characterize what is a correct digital signature, that is, a well-formed and not-revoked signature:

Authenticity	a correct signature uniquely identifies a principal, and only that principal
Unforgeability	a correct signature was made by its owner deliberately
Integrity	a correct signature on a document ensures that it cannot be changed without that being noticed
Non-reutilization	the whole or part of the signed document cannot be reused in another document
Non-repudiation	a correct signature cannot be denied by the owner of the signature (key)

These properties emulate what we would desire of paper signatures. Actually, some of these properties can be violated in hand-made signatures. In computers, we have to be even more careful, since a computer file is vaporware compared with the hardness of a paper signature. *Authenticity* stipulates that the signature unmistakably identifies a given principal. That is, people can recognize s_a as being Alice's signature. However, Mallory might imitate Alice's signature, so it must also have the property of *unforgeability*, which ensures that if we are facing Alice's signature, it was really made by her deliberately, because no one else could forge it. Once a document is signed, it cannot be changed, at least without that being noticed. That property is called *integrity.* Suppose Mallory did cut-and-paste of Alice's signature from a legitimate document file to a document forged by him? This would be quite easy with ASCII files. In consequence, we know that it must be avoided by stipulating the *non-reutilization* property. In addition, now we know that digital signatures are not made in ASCII. Sometimes, our problem is not with Mallory forging a signature, but with Mona, who disguises herself, buys expensive jewelry, but later denies saying that someone stole her check wallet on that day. *Non-repudiation* is the property that ensures that the signer cannot deny having put her signature on a particular document. The fact that the signer can accept this property follows from the previous properties. As an exercise, deduce that a signed document obeying the first four properties *must* be a legitimate, unaltered document deliberately signed by the signature owner.

17.6.1 Cryptographic Checksums

Checksums are small fixed-length strings that are used to check the integrity of messages or files. Hashes give good checksums, like network packet CRC (cyclic redundancy check), but unlike checking against accidental bit errors, a mere hash is not enough against a deliberate attack: Mallory changes the message and then recomputes the hash (which is public). A *cryptographic checksum (CC)* is a non-forgeable hash, in a sense, a form of signature of a message, that depends on a key. We use the following terminology to differentiate from plain hashes: the *CC* of M is $H_K(M)$. CCs make sense when a message does not require encryption but integrity must be safeguarded. They may also authenticate messages exchanged between two users. They separate protection

from encryption: when a message is encrypted, it is naturally protected, but after decryption at the recipient, the protection is lost. Having CCs of sensitive files stored in disk would for example foil virus or Trojan horse invasion.

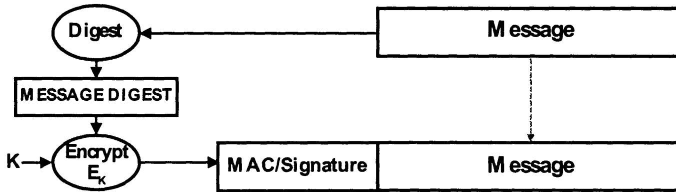

Figure 17.8. Generating and Appending a Signature or MAC to a Message

Cryptographic checksums may be implemented in several ways, and take several names: *Message Integrity Code* or message integrity check (Linn, 1993), or *Message Authentication Code (MAC)*, depending on whether they are securing integrity or authenticity, respectively. A simple way of generating a MAC is the following: Bob hashes the message, with say MD5, and then encrypts the hash with any symmetric algorithm, say DES, obtaining a MAC. The block diagram of this scheme is explained in Figure 17.8, considering that E_K is a symmetric block cipher with secret key K.

An alternative and very simple method is based on hashing only, dispensing with encryption. It only depends on Alice and Bob sharing a secret key K (not an encryption key, just a secret key): Bob computes the length L of message M, and concatenates L, M, and K; then he computes the message digest, obtaining $H_K(L, M, K)$. The approach is very fast and it is secure with a resilient message digest algorithm.

Cryptographic checksums secure the unforgeability and integrity properties. The above methods have the key distribution problem typical of symmetric approaches. By using public key cryptography the key distribution problem is minimized, while also providing an elegant method for true digital signature.

17.6.2 Signing and Verifying

A very interesting additional result of asymmetric cryptography is that *encrypting with a private key* or *decrypting first* (then encrypting) is equivalent to *signing*. Although there are public key signature algorithms, if an encryption algorithm is used for signing, then: *signature* by A is $S_a(M) \equiv D_{Kr_a}(M)$ and *verification* of A's signature is $V_a(M) \equiv E_{Ku_a}(M)$. The principle is depicted in Figure 17.9.

There is no problem in doing the operations in reverse order. Note that when Alice encrypts M with her private key getting S, she produces something unique and unforgeable, since her key is secret. On the other hand, anyone can verify Alice's signature: when Bob receives S supposedly signed by Alice, he fetches Alice's public key, and decrypts S, which could only have come from Alice. It is not as simple as that though: if Bob does not know what he is

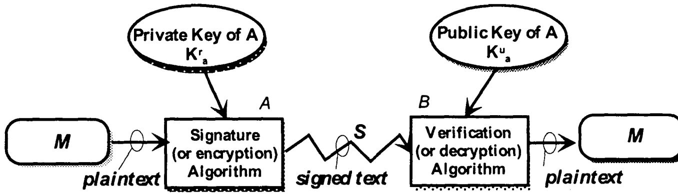

Figure 17.9. Asymmetric Digital Signature

expecting to verify, he can be fooled. Consider the following: Mallory can send a piece of data to Bob, pretending it is a signed document coming from Alice; under certain circumstances, Mallory may even construct a ciphertext that makes some sense when Bob verifies it; this is forging, and it is undesirable. Any encryption algorithm can be used to sign, but there are reasons, of both efficiency and security, to be careful about the structure of the signed document and/or to use specialized signature algorithms.

It is not efficient to run an algorithm on a whole text just to sign it. Imagine it is a 10 MByte contract! The message digests that we have studied solve the problem, yielding a *message-digest public-key signature* protocol. The principle can be understood by looking again at Figure 17.8, but considering now that E_K is a public key (asymmetric) cipher, and K is the private key of the signer. The protocol is explained in Figure 17.10, where Alice wants to send a signed text M to Bob.

		Action	Description
1	A	$h_m = H(M)$ $s_m = S_a(h_m)$	Alice computes the message digest and signs the 128-bit digest with her key
2	A $\rightarrow$ B	$\langle M, s_m \rangle$	She sends both the text and the signature to Bob
3	B	$h_m = H(M)$ $v_m = V_a(s_m)$ $(v_m = h_m)$?	Bob verifies the signature using the following procedure: he hashes the message; then he verifies the signature using Alice's public key; if the result is equal, then M is ok, and was signed by Alice

Figure 17.10. Message-digest Public-key Signature

Let us discuss now the effectiveness of digital signatures in real-life applications. For example, Bob cannot *reuse* the signature to append in another document, but he can reuse the whole document (it is a file!). This is not convenient if it is a bank check or digital note. In order to completely assure the non-reutilization property when the whole document cannot be reused, the document should include unique sequence numbers, timestamps and/or expiry

dates before it is signed. The verification process will include checking if the sequence number exists, if the expiry date was not exceeded, etc.

There is another problem, concerning **repudiation**, that also exists with hand-made signatures and negotiation protocols (e.g., credit cards). It is explained as follows: Bob signs a document at 3:00pm. Then he later complains to the police that someone early that morning stole the diskette where he held a copy of the private key, and gives forged evidence that this might have happened as early as 12:00am. In consequence, he denies all signatures that he made since 12:00am, including the document in question. There will always be a window of uncertainty in these operations. In other words, it is a fundamental problem: it can be reduced, but it cannot be eliminated, unless the model is changed. In fact, we can eliminate it if we timestamp and certify all transactions, but that requires on-line access to a mediator.

The security of public-key signature with one-way hashing lies on two facts besides the resilience of the signature and hashing algorithms: the secrecy of the signer's key, both physical and in terms of length (e.g., 1024 bits yield an enormous search space); unfeasibility that another message has the same hash as the original one (e.g., 128 bits yields a probability of 2^{-128}). This signature approach has the authenticity, unforgeability, integrity, non-reutilization and non-repudiation properties. In that sense it is a fully-fledged signature.

There is an additional advantage in the message-digest public-key signature scheme explained above, which concerns *multiple signature*. Suppose that the contract of our last example was to be signed by n principals. Instead of doing n whole-text signatures, each principal signs a copy of h_m, and the signed text is $\langle M, s_m^1, ..., s_m^n \rangle$. Each signature can be verified separately.

Still another advantage is for *notary* purposes: Alice wants to archive a document with a notary, who dates it and certifies that Alice produced it before that date. In the classical procedure, the notary would have to see and copy the document, and that can be inconvenient. With this method, the notary only certifies the digest of the document, and so Alice can keep the privacy of her document and only reveal it if it is ever necessary to prove that the certification corresponds to it. The probabilities that there is another meaningful text that hashes to the same value are negligible.

Signing with symmetric cryptography is worthwhile mentioning. Since a symmetric key would have to be shared between two principals that do not trust each other (the signer and the verifier), this cannot be done directly, but rather through an **arbiter**. This is obviously bothersome.

17.6.3 DSA

The *Digital Signature Algorithm (DSA)*(DSS, 1994) is an asymmetric algorithm for signing. Most of the market used RSA for signature until the *Digital Signature Standard (DSS)*, featuring DSA, came out in 1994.

We will give an overview of the protocol, skipping the details. The signer has a long-term pair of keys: a private key Kr chosen at random and a public key Ku computed from some public numbers and the private key. The key

length is variable from 512 to 1024. For each signature, a new pair of keys is generated, let us call them single-use keys: private Kr_s chosen at random and public Ku_s computed from the public numbers and the private key. A text M is previously hashed and then signed with a function using Kr_s, Ku_s, and Kr. The signed text is $\langle M, s_m, Ku_s \rangle$, where s_m is the signature, and Ku_s the public single-use key for that text. The security of DSA relies on: generation of "good" public numbers; Kr being kept secret; and Kr_s not being reused.

17.6.4 Blind Signature

Blind signatures were invented by Chaum (Chaum, 1983), with the purpose of authenticating an object without revealing the identity or whereabouts of its owner. If the object is digital money, this provides for **untraceability**, a desirable property that real money has, but is very difficult to achieve with digital money, without impairing other properties (*see* Section 17.7).

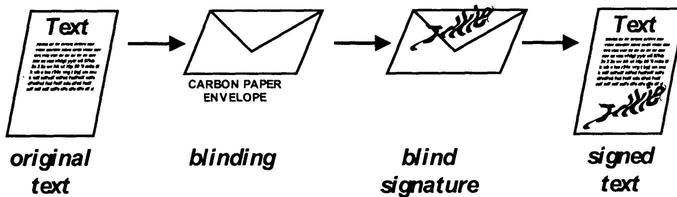

Figure 17.11. Generating a Blind Signature

The metaphor behind the blind signature concept is illustrated in Figure 17.11. The problem is the following: Alice has a document to be signed by Trent, but she does not want Trent to read it. She puts the document inside an envelope with carbon paper lining, and asks Trent to sign the envelope on the outside. The signature prints on the document as well because of the carbon paper. Alice opens the envelope and has the document signed by Trent. The algorithm is described in Figure 17.12. Alice wants Trent to sign document M for her. Trent has public and private keys u and r, and public modulus n. It works because the blinding and signing operations are commutative.

This completely blind signature is uncomfortable for Trent, since he does not know if Alice is giving him something nasty to sign (such as "I owe Alice a million EURO"). A protocol combining blind signature with a cut-and-choose technique introduces faiirness in the process. The protocol, shown in Figure 17.13, is a generalization of the protocol of Figure 17.12. The cut-and-choose metaphor is explained as follows:

Cut-and-Choose - Alice and Bob have a bucket full of fish that they caught, to divide in half. Alice divides the fish in two piles (cutting) and Bob blindly picks one of them (choosing). Bob thinks this is fair because his odds of getting the best pile are 50%. On the other hand, Alice should make the division fair, since she has no advantage

		Action	Description
1	A	$\widehat{M} = Mk^u \bmod n$	Alice prepares document M, and blinds it, obtaining $\widehat{M}$, by multiplying by a quantity depending on the blinding factor (random number k) and Trent's public key (u)
2	T	$\widehat{S(M)} = S(\widehat{M}) = (Mk^u)^r \bmod n = M^r k \bmod n$	Trent signs the blinded document $\widehat{M}$
3	A	$S(M) = \widehat{S(M)}/k = M^r \bmod n$	Alice unblinds $S(\widehat{M})$, obtaining document M signed by Trent with r

Figure 17.12. Blind Signature Protocol

in cheating by making one pile bigger than the other— her odds are 50% as well.

		Action	Description
10		A → B	Alice prepares p blinded forms of notes of value V, blinds each of them with a different blinding factor and sends them to Bob, the banker
20		B	Bob selects $p-1$ forms, asks Alice for their blinding factors, removes the blinding factors from the forms, and verifies they request V
30		B → A	Bob signs the remaining note form, N, hands it to Alice and debits V from Alice's account
40		A	Alice removes the blinding factor from note N, but Bob's signature remains on it. She can spend it in a shop now

Figure 17.13. Basic Digital Cash Minting Protocol

The chances that the last copy of the note form contains something different are 1 in p, and of course, this probability can be made as small as wished, since it is dictated by the number of blinded copies Alice has to generate. By the properties of digital signatures, Bob can later recognize his signature and is bound to it, even if he has never seen the text that he signed.

17.7 DIGITAL CASH

Digital cash is the materialization of the "money" metaphor onto computer and digital systems. It is an enabling paradigm for emerging technologies that will certainly play an extremely important role in the near future, such as digital

payment systems, electronic transactions and electronic commerce. Digital cash depends on a number of cryptographic principles that we have already discussed, namely digital signatures, and very specially blind signatures.

17.7.1 Properties of Digital Money

Perhaps the two most frightening nightmares about digital money are: for the user, that it evaporates somewhere inside a network or computer; for the authorities, that someone discovers how to counterfeit it. This is nothing that could not happen with real money though: it could burn under the mattress in a house fire or evaporate in the bank under a depression; it can be counterfeited with several degrees of perfection.

It is extremely important however, to formalize a set of properties for the digital money concept. Something against which algorithms, protocol and system designs can validated. The following desirable properties, almost sic from (Okamoto and Ohta, 1992), comprehensively define digital money:

Independence	properties of digital money do not depend on its location
Uniqueness	digital money items cannot be copied or reused
Untraceability	digital money items cannot be traced
Off-line Validity	digital money items have standalone value
Transferability	digital money items can be transferred between users
Divisibility	digital money items can be subdivided

The independence property stipulates that digital money does not get less secure when pieces get out of the user's wallet and into the merchant's terminal, for example. Uniqueness stipulates that you cannot counterfeit money by copying an item or by reusing it. Untraceability is a very important property, though many existing systems do not provide it: it guarantees, such as with real money, that one cannot trace where the money was spent or who spent it. Off-line validity assures that money can be spent without need for connection to any central system. Transferability and divisibility ensure that users can pay things or give money to each other, instead of only to the merchant, and that pieces can be subdivided into smaller ones.

Existing digital cash systems fulfill only some of the properties, while an experimental system proposed in (Okamoto and Ohta, 1992) satisfies them all. The need for particular properties and the impact of their absence will be discussed further in Section 18.11.

17.7.2 Generating and Using Digital Cash

To introduce a digital cash payment system, we will use as our point of departure, the cash minting protocol using blind signatures and cut-and-choose that we studied in Figure 17.13. After Alice has note N (step 40 of Figure 17.13)

she goes on and spends it at Mike's shop. The basic protocol for payment with digital cash is shown in Figure 17.14.

	Action	Description
50	A → M	Alice spends N in Mike's shop
60	M	Mike the merchant checks Bob's signature
70	M → B	If everything is Ok, he accepts payment and sends N to Bob
80	B	Bob checks the signature and credits V to Mike's account

Figure 17.14. Basic Digital Cash Payment Protocol

This protocol lets Alice or Mike *reuse* the note, although they *cannot forge* it. This is called *double spending*, and a simple modification addresses the problem. It consists of having Alice concatenate each form of note with a random *uniqueness string (Un)*. Bob now also verifies that the Un strings are all different, when unblinding the forms. The note finally gets back to Bob, after Alice bought her merchandise. Bob, besides checking the signature, also checks that Un does not exist yet in his "spent" database list. If it does, then there has been double spending. There is a final problem to be solved: double spending is indeed detected, but the guilty may either be Alice or Mike, that is, there is an *imperfect detection.* One possible remedy would be for Mike to ask Alice to write a random non-erasable *identity string (Id)* on N. Then, Bob compares the identity string in the database record with the one in the note: if it is the same, Mike is the guilty one, if not, it is Alice. However, Alice could have forged the Un random number generation, giving the same identity string in the second time, to frame Mike. In consequence, this remedy and the previous one are only safe for *on-line* spending.

Off-line operation requires more sophisticated techniques to create Id, shown in the final protocol in Figure 17.15, where we consolidate the protocols of Figures 17.13 and 17.14 and show the modified or added lines in bold. Recall that Un is a uniqueness string randomly generated, long enough that there are no two notes with the same Un. Before we proceed, let us introduce two more cryptographic operations:

Secret Splitting - division of an item M of data in two parts such that either of them alone reveals no knowledge about M; joining of the two is a public operation that reveals M

Bit Commitment - processing of an item M of data such that M can no longer be changed and the result reveals no knowledge about M; M can be publicly revealed when the owner reveals a secret

Detail on the two can be found in (Schneier, 1996). The preparation of the Id strings in step 10 is the following: Id is an ASCII string revealing Alice's

	Action	Description
10	A → B	**Alice prepares p blinded forms of notes of value V, concatenates each form with one *uniqueness string (Un)* and p *identity strings (Id)*, such that for each one, $Id_j = Id_{j_L} \| Id_{j_R}$, and sends them to Bob, the banker**
20	B	Bob unblinds $p-1$ forms and verifies that they request V
21	B	**Bob also verifies that the Un strings are all different and that the Id strings identify Alice**
30	B → A	Bob signs the remaining form, N, hands it to Alice and debits V from Alice's account
40	A	Alice removes the blinding factor from note N
50	A → M	Alice spends N in Mike's shop
60	M	Mike the merchant checks Bob's signature
52	M,A	**Mike gives Alice a binary *selection string (Ss)* of length p and requests Alice to reveal either Id_{j_L} or Id_{j_R} of each of the p Id strings of N, depending of the value (0 or 1) of Ss in that position**
70	M → B	If everything is Ok, he accepts payment and sends N to Bob
80	B	**Bob checks the signature, checks that the Un string does not exist yet in his "spent" database list, inserts the Un and Id strings in the list and credits V to Mike's account**

Figure 17.15. Robust Digital Cash Payment Protocol

identity completely; Id is secret splitted in two halves Id_{j_L} and Id_{j_R}; each one is bit committed. This ensures that: Alice cannot change them; only the two halves reveal Alice's Id. If the Un string exists, there is a problem. Then Bob checks the Id string: if it is the same, Mike is guilty, if not, it is Alice. Let us understand why:

If they are the same, it can only be because either Mike copied the note, or Alice forged two identical notes. However, if Alice spends the note again, the new merchant will give her a different selection string Ss. It is almost impossible that the two notes look the same (probability 2^{-p}), even if Alice wanted to, trying to frame Mike. On the other hand, if they look different, could it be Mike trying to forge a slightly different note and reuse it, framing Alice? That is impossible, since only Alice can reveal other sections of the Id string. In consequence, if they are different, definitely Alice is cheating. Furthermore, Alice must have revealed Id_{j_L} to one merchant and Id_{j_R} to another, in at least one of the p Id strings, and can thus be identified by Bob.

So finally, we have a protocol whereby Alice can *anonymously* spend her digital cash *off-line* without any fear of tracing, unless she cheats. However,

the protocol provides adequate safeguards to the merchant and the bank, by *detection* of fraud. Recall that the security of this protocol lies in the blind signature privacy, and in the very low probabilities: of Alice cheating the bank ($1/p$, for p initial note forms); of there existing two equal Un strings (2^{-L}, for an L-bit length); and of there existing two identical selection strings (2^{-p}, for a p-bit length).

17.7.3 Payment with Tamperproof Devices

There are two problems with the previous protocols: people can still cheat, although they are detected and prosecuted; eavesdroppers and spoofers can defraud the scheme if they have physical access to the devices involved.

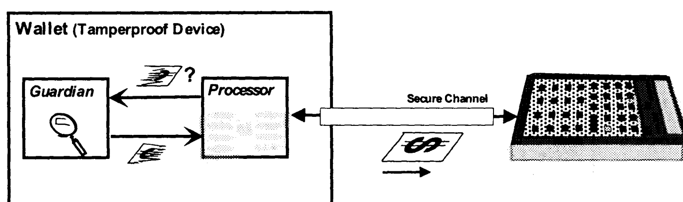

Figure 17.16. The Guardian Concept

The obvious way to avoid this problem is to give Alice and the merchant *tamperproof devicestamperproof device* where they store bank notes, and which are capable of running a secure end-to-end protocol. This way, Alice or Mike cannot tamper with the devices, and an intruder cannot penetrate the secure channel. It is possible to conceive a secure off-line payment system this way. Let us sketch a simple protocol for payment with tamperproof devices (e.g., smart-cards):

1. *Bob the banker has a public/private key pair ($Term_u$, $Term_r$) and a public/private key pair ($Note_u$, $Note_r$) to generate 1 EURO notes; Bob produces Alice's wallet and Mike's terminal, and stores the $Note_u$ and $Term_r$ keys in the terminal; he also stores $Term_u$ in Alice's wallet;*
2. *Alice loads her wallet with a sum in notes produced and signed by Bob with $Note_r$, say at an ATM (another tamperproof device);*
3. *Alice connects her wallet to Mike's terminal, and wants to pay a purchase;*
4. *The wallet authenticates the terminal with the $Term_u$ key and both establish a secure channel; the wallet transfers a sum V in notes to the terminal, which the latter validates with the $Note_u$ key;*
5. *The terminal later uploads the notes to the bank, and Bob checks his signature, verifies the note number against a "spent" list, and credits Mike.*

This is secure because the devices are trusted by the bank not to cheat, are supposedly tamperproof, and only connect to an alien device after authentication. Note that step 5 is a double check for an eventual fraud, but it implies that Bob recorded the note number upon its generation and most probably, Alice's identity as having purchased the note. When the note comes back, it

may have recorded the place where it was spent, and so Alice's steps can be traced. If the wallet, for example a smart-card, is trusted only by the bank who issued it, Alice cannot be sure that her privacy will be safeguarded. If on the other hand, it is trusted by Alice alone, then the bank or the merchant may question its security.

17.7.4 Payment with Guardians

The solution lies in the *guardian* or *observer* concept (Chaum, 1992): the **electronic wallet** consists of a tamperproof device whose processor is trusted by the user. The wallet processor executes the protocols, but needs the cooperation of another element, the *guardian*, trusted by the bank. Their relative position and interaction, as illustrated in Figure 17.16, is such that: the guardian cannot tamper with the protocol execution; the guardian cannot communicate with the outside; the processor cannot progress with the protocol execution without the repeated assistance of the guardian. Alice trusts the processor and knows that the guardian cannot disturb its operation, nor reveal secrets to the outside without the processor noticing. The bank trusts the guardian and only accepts operations in which the guardian has partaken. This is called *multi-party* security. We can make our protocol sketch evolve to payment using tamperproof devices with guardians:

1. *Alice has a wallet with Gus, the guardian from Bob the banker;*
2. *Alice loads her wallet in the bank with notes of value V;*
3. *Gus loads a down counter with V, which it decrements at each payment;*
4. *Bob generates a pair of public-private signature keys, hands the private one to the care of Gus, and gives the public one to Alice's wallet;*
5. *When Alice spends a note N, her wallet has Gus sign the note, after which it decrements the counter;*
6. *Alice hands the note N to Mike the merchant (Mike has Gus's public key). Mike checks the signature;*
7. *Mike sends N to Bob, who checks the signature again and pays Mike.*

The protocol above is just a sketch: it is too simple and naive, since anyone can generate a pair of keys and simulate Gus operation near Mike. Alice can do double spending. Besides, the protocol is sensitive to Gus being broken into. The solution lies in having Alice and Gus cooperate to generate a blinded signature book containing totally anonymous signature key certificates called **digital pseudonyms**, each of which will serve to validate one and only one spending. A full discussion, and working protocols can be found in (Brands, 1995) and (Chaum, 1992).

17.8 OTHER CRYPTOGRAPHIC ALGORITHMS AND PARADIGMS

Encryption and Digest Algorithms

IDEA, *International Data Encryption Algorithm* (Lai, 1992) deserves our attention because it is currently one of the most promising symmetric algorithms,

it is widely used and has been extensively analyzed during the past few years, without any fundamental weaknesses discovered. The most relevant protocol using it is PGP (*see* Section 19.1). It is patented and can be licensed for commercial applications. It is a block cipher of 64-bit blocks.

RC4 is a symmetric stream cipher with variable key size, widely used in several protocols. One example is the SSL protocol (*see* Section 19.1). Although it is proprietary (RSA Data Security), its sources became public domain on the Usenet years ago. RC4 has a special export status if it uses a reduced key length, which was up to 40 bits long during many years. Although becoming fragile, this was attractive to U.S. companies willing to export their secure systems. They only have to modify the protocol to use a reduced key length. What is strange is why this would be attractive to buyers in countries where there are few or no restrictions to cryptography, in detriment of alternative cryptographically-stronger products. Certainly, a different attitude would help liberalize cryptography. Merkle's Knapsack was really the first asymmetric encryption algorithm, but suffered a series of cryptanalysis that compromised its success (Merkle, 1978). El Gamal (ElGamal, 1985) is an asymmetric encryption and signature algorithm which inspired DSA. The main differences are in performance, DSA being significantly faster. The Secure Hash Algorithm (SHA), was proposed as a standard to be use together with the DSA signature standard. It produces a 160-bit hash, and makes five passes over each block, so it is in principle more secure against attacks than MD5, which has a 128-bit output and makes four passes. It is however slower than MD5.

Random Number Generators

Random numbers are a very important building block in cryptography. Operating systems sometimes do not have really random number generators: they have a period and other deterministic characteristics, and although good enough for most of our uses, they do not resist cryptanalysis. This is worrying if the security of a given algorithm depends on true randomness. Some applications, like physical circuit or link encryption (*see* Section 16.3), require that both ends have devices such as shown in Figure 17.3: they cannot be really random otherwise they could not be synchronized, but they have to look like producing random sequences. These are called *cryptographically secure pseudo-random number*generators. A good quick test of a secure sequence is that it should not be compressible. Techniques for generating good randoms are detailed in (Schneier, 1996).

Steganography

Steganography is an old paradigm for concealing data. It existed before computers, but it can assume very sophisticated forms when computer technology is available. It is based on hiding information under an apparently normal or innocent piece of data. Historical examples include: invisible ink; markings on paper, only visible under a certain light angle; tiny punctures on letters of

a printed sheet; and so forth. Steganography is not alternative to cryptography. Cryptography obscures the content of a message, but not the message, which reveals a notion of importance or secrecy to outsiders, because of being encrypted. Steganography conceals the existence of the very message, in order that outsiders do not even know that communication is taking place. In contrast, when the coding is discovered, both the existence of the message and its contents are revealed.

Key Escrow

Free use of strong cryptography has raised fears that underground forces such as terrorists, organized crime and so forth could make use of it to conceal their activities. In the U.S.A., this lead to a proposal of an **escrow** encryption system for use by the general public, where the keys of the users would be copied, split between two state agencies and safely stored in databases. The system would normally preserve confidentiality, but under a court warrant requiring the cooperation of both agencies, the key could be surrendered to the authorities, which would then be able to tap communications or decrypt files. These rules were defined in an Escrowed Encryption Standard (EES) (EES, 1994). The algorithm initially proposed for the system was a symmetric algorithm called Skipjack whose structure remained secret, and which would only be available through tamperproof hardware devices, of which at least two prototypes were produced, called Clipper and Capstone. Later, a more reasonable proposal took form, in what was called *fair cryptosystem* by its inventor (Micali, 1993). The basic ideas of the method are the following: it relies on resilient public-key cryptography; it lets users generate their own keys pairs; the private key is split in n parts and handed to *several* official agencies, with an algorithm that guarantees that the key can only be reconstructed by having at least $k \leq n$ parts. The latter condition prefigures what is called **threshold cryptography**.

17.9 AUTHENTICATION

Authentication is the process of proving the identity of a principal A, or that of proving that B acts on behalf of A. Real systems solve several facets, stronger or weaker, of this problem. For example, Alice may prove her identity as creator of a document by signing it, or as an authorized user of a service through a password. A machine Threepeeo may prove its identity to another machine by its address. These are examples of *one-way authentication*. What if Alice does not trust the server? Then, the server must also authenticate itself to Alice. This is called *mutual authentication*.

When Alice sits on Threepeeo, she delegates the authentication process on it, to execute the login program on the server. What happens if Threepeeo has been tampered with, or someone stole Alice's password? Or, what if some other machine impersonates Threepeeo's address? How can Alice or Threepeeo unequivocally prove their identity? Alice could use her cryptographic signature in the process of authentication. A PIN-protected tamperproof smart card Ar-

toodeetoo, personal to Alice, may prove its identity and indirectly that of Alice. The relationship established between Alice and Threeepeeo or Artoodeetoo are forms of what is called *delegation*. However, what if Alice lends Artoodeetoo to someone, or loses it together with the PIN? Or else, she lets her private key be stolen?

This section will discuss answers to these questions.

17.9.1 Types of Authentication

There are three basic *types of authentication*, depicted in Figure 17.17:

Unilateral	authentication is based on principal A authenticating itself to principal B
Mutual	authentication is based on principals A and B mutually authenticating themselves
Mediated	authentication is based on principal A being authenticated to principal B by a mediator T, whom they both trust

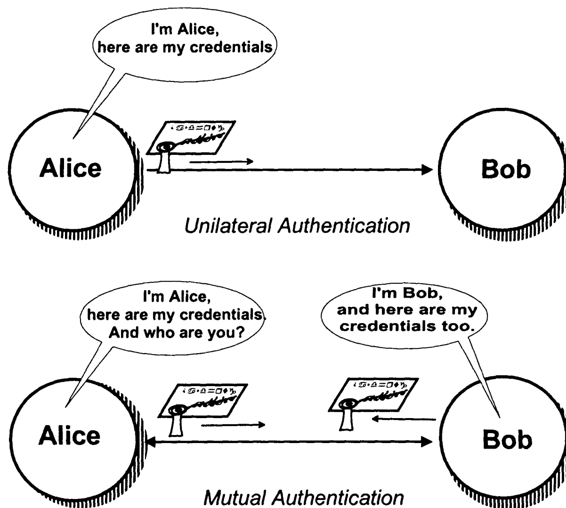

Figure 17.17. Authentication Types: (a) Unilateral; (b) Mutual

Unilateral authentication is the simplest form of authentication. A principal A (Alice in Figure 17.17a) has to produce credentials that accredit her with principal B (Bob). The term **credential** is used here in a free manner. In fact, it may take several forms, as we will see in Section 18.5. It can be a password, a signature or a cryptographic seal on one or more messages, or the proof of knowledge of any of those. Unilateral authentication is characterized by the fact that at the end of the protocol, Bob (often a server) believes it is Alice who is dialoguing with it, whereas Alice (often a user) can never be sure that

she is really talking to Bob. In certain situations, this is inappropriate, and mutual authentication should be used. Essentially, as Figure 17.17b suggests, both principals follow similar steps to persuade one another of their identities. At the end of the process, both Alice and Bob are mutually sure that they are talking to one another.

In other cases, namely in distributed systems, principals are capable of performing pair-wise mutual authentication, as defined in the previous paragraph. More precisely, all principals are capable of performing mutual authentication with a distinguished principal, a mediator. Each newcomer in the system must "learn" how to authenticate to the mediator. In turn, the mediator is capable of performing authentication (unilateral or mutual) between any two principals in the system. This scheme is obviously attractive for high-level authentication in open distributed systems. As suggested in Figure 17.17c, Trent already knows Alice and Bob, but they do not necessarily know, or trust, each other. Alice requests Trent to introduce her to Bob. Trent hands credentials to both, that will allow Alice and Bob to perform an exchange similar to that of Figure 17.17b, and get authenticated.

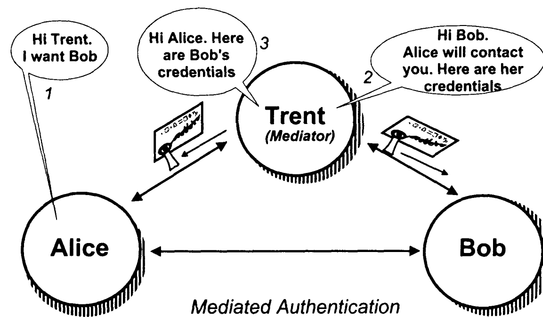

Figure 17.17 *(continued).* Authentication Types: (c) Mediated

17.9.2 Delegation

Another challenge to security related with distributed systems, is that they are modular, and geographically dispersed. In consequence, an end-user needs to have a few devices perform operations, many often in remote machines, on her behalf. The principle to achieve this correctly, that is, to authenticate other devices to act on one's behalf, is called *delegation.*

Consider the following, picking again our example from the beginning of this section: Alice is trying to login from her PC Threepeeo onto a department server S. Most of the time, all Alice does is type in her username and password. Threepeeo takes care to perform all the necessary calculations and run the necessary protocol steps. So, when Alice sits on Threepeeo, she actually *delegates* the authentication process on it, to execute the login program on the server.

The first problem is: what if Threepeeo has been tampered with? Threepeeo will do all sorts of inconvenient things using Alice's name and account, and it is hard for Alice to say she did not. This problem is caused by the fact that Alice delegated her full privileges in the organization on her computer account, *without restriction and forever.* For example, Alice is organically untitled to access the department's CD-ROM recorder, but technically, it is Alice's account on Threepeeo that has that right, always. If Alice would delegate properly, by issuing specific authorizations for specific actions, during a specific time, then she would have a powerful mechanism for securely having devices do actions on her behalf, with a much narrower window of vulnerability. Moreover, they could forward the permission to other devices, if the latter would still respect the conditions of the delegation. This is the principle of *specific delegation*, and has to do with *what* Alice authorizes others to do.

The second problem is: What if someone or something impersonates Alice or Threepeeo and issues (forges) or forwards a delegation to the wrong place, with a wrong content? This may cause trouble and will be problematic for Alice. This problem happens when the delegation techniques used by Alice do not allow her to unequivocally prove her identity, that is, authenticate herself. If Alice would authenticate properly, by issuing delegation certificates that have standalone validity, that is, remain as secure during their useful life as they were when they left Alice's hands, this problem would be highly mitigated. Use of cryptographic authentication protocols is paramount here. This is the principle of *authentication forwarding*, and has to do with ensuring that *it is* Alice who authorizes others to do things. It is the only secure way to forward a delegation.

A final problem remains, and it is: What if someone is still capable of forging the original certificate? This may cause a lot of trouble and it will be even harder for Alice, since as mechanisms get more sophisticated, Alice will have more trouble proving that they have been tampered with. The first explanation would be that the cryptography involved had been broken. The problem is however more complex than that, it has to do with what we might call the *end-to-end authentication* problem, and may occur even under the doubtful assumption that the cryptographic certificate were absolutely secure after issued. The certificate might have been forged because someone stole Alice's keys or passwords, and there is little one can do about it. Alternatively, the forgery might have taken place somewhere between Alice's keyboard and the certificate file writing, because someone might have tampered with Alice's machine in order to perform a sophisticated impersonation attack. The situation can be improved by reducing the vulnerabilities of that tiny end path from Alice, the person, and her first delegation, the device that issues certificates. The protected tamperproof device Artoodeetoo where Alice enters a PIN or a password and which issues the certificate or cooperates with her machine for that purpose, is a good advance. It is practically end to end, so nothing can get in the middle. Of course, Alice could lose Artoodeetoo, with her PIN written on the back. Well, then we would need serious end-to-end stuff. Short of implant-

ing a computer chip into Alice's brain, some advanced biometrics technology such as iris scanning can help, because it is still difficult for Alice to lend or lose her eyes. However, there is no definitive "cryptocratic" solution, because you should not forget the comments in Section 16.1: more than technology, the behavior of humans is central to achieving security.

17.9.3 Key Distribution

These are the basic authentication paradigms. Before concluding, let us introduce a problem related with authentication: key distribution. Cryptographic protocols need keys to operate, genuine keys have to get to the participants, and sometimes these keys are established just for one session. In some cases, it is convenient to combine key exchange with authentication, to avoid impersonation and forgery attacks.

Key distribution in general is a paradigm crucial to any cryptosystem, the has two facets. In fact, in all the protocols we have just discussed, we have considered that *long-term key distribution* had already taken place, that is, the keys were already with their legitimate owners and users. This includes long-term secret keys shared between two principals or public/private key pairs used for the purpose of signature and authentication. Another facet of the key distribution problem is what is called *short-term key exchange*, this particular facet of key distribution being understood as the ad hoc, frequent and on-line exchange of keys for temporary use, e.g., with the purpose of setting-up a temporary session or communication channel. These keys are called **session keys**. This is obviously a sensitive issue, since, recalling our introductory notions on *risk*, it is an operation subject to a very high level of threat. In consequence, any vulnerabilities in the protocols will be highly exposed.

The long-term key distribution problem is the bootstrap problem of most cryptographic systems, from authentication to communication systems. All must start with the first shared key or pair of asymmetric keys. Long-term public keys may be distributed by the owner to the people she interacts with, via some trusted off-line mechanism. After a few keys are in place, they can be used to sign new key files, which we call certificates, in a transitive signing chain of mutual trust. This is used for example in PGP (*see* Section 19.1). Public key certificates can travel securely over the network. Exchanging shared secret keys for symmetric cryptography is a harder problem. Unlike public key distribution, secret keys in transit not only risk corruption but also disclosure. The "primordial secret" has to be sent by some other means, for example a combination of mail, telephone, and fax. After that, principals can exchange new keys under the cover of this first key.

17.10 ACCESS CONTROL

Access control is concerned with validating the access rights of users to resources of the system. It can be seen at different levels. One may control the access of end users to database records, control the access of users and programs to

files, or control the access of program instructions to segments and pages of a computer system. The first type of control is implemented by the database management system, the second by the operating system and the third by the microprocessor. There are several approaches to access control, and systems in general possess some form of such control. The advances in computer security have contributed to a sophistication of access control mechanisms and protection models. In what follows we present the main *access control mechanisms*: *access control lists*; *capabilities*; *access control matrices*. Mechanisms are used to implement *access control models*, which essentially reflect a specification of how principals should access computer resources.

A *protection domain* is the set of resources that lie under the realm of an access control mechanism. They could for example be the O.S. resources depicted in Figure 16.4. The several ways resources can be accessed are called *access rights*: read, write, execute, lookup, create, delete, truncate, append, insert. Resources are generically called *objects*: pages, files, processes, devices. The entities trying to access are called *subjects*: humans, programs.

17.10.1 Canonical Access Control Mechanisms

The most popular access control mechanism, used by many operating systems, is the **access control list** or **ACL**. The ACL mechanism is defined by the following:

- each *object* has a list (ACL) of the subjects that may access it;
- each element of the list is a pair ⟨*subject,rights*⟩, where: *subject* is an *Id* of a user, a process, or a group; and *rights* is the enumeration of the access rights granted to this subject on that object, usually a bit mask (e.g., `xrwd` for UNIX, i.e. execute, read, write, delete);
- the ACL is protected by the system against unauthorized modification.

Among the advantages of the ACL approach, we note that the access control of an object is centralized in a single structure. Besides, when many subjects share the object, they are simply grouped in specific or generic groups (e.g., `world` in UNIX is "all subjects"). However, in security terms, grouping can present a vulnerability, since it hides access rights of individual subjects and is thus prone to granting access rights to the wrong subject, for example when increasing the access rights of a group. On the other hand, discriminating every subject's rights would be impracticable, since it would soon overload the ACL. Another well-known access control mechanism is the *capability*. The capability mechanism takes a different approach from the ACL one:

- each *subject* has a list of the objects of a protection domain which it may access;
- each entry of the list is a capability: a pair ⟨*object,rights*⟩, where: *object* is the *Id* of a resource; and *rights* is the enumeration of the access rights granted to that subject on this object, usually a bit mask;
- the capability is protected by cryptography against unauthorized modification or forging.

Compared with ACLs, capabilities are oriented to subjects rather than to objects. For that reason, they do not exhibit the problems of ambiguity of subject access rights and of overloading: a subject must have a capability for any object it wishes to access; and only has capabilities for the objects it accesses. Besides, since a capability is protected, it can be securely transferred or copied to other subjects, yielding delegation of access rights. Note that capabilities do not reside permanently with the subjects, they are requested to a control entity when necessary. This entity has a central list of subjects versus access rights to objects, which may be an ACL, or an access control matrix, that we discuss below.

17.10.2 Access Control Matrix

A formal model of access control (Lampson, 1974) requires a more complete statement of access rights, in the form of an *access control matrix* or *ACM*. The ACM mechanism is defined by the following:

- each *subject* has a list of the objects of a protection domain which it may access;
- each *object* of a protection domain has a list of subjects that may access it;
- these two lists form a matrix where each entry is a triple ⟨*subject,object,rights*⟩, where: *subject* is an *Id* of a user or process; *object* is the *Id* of a resource; and *rights* is the enumeration of the access rights granted to the subject on the object;
- the ACM is protected by the system against unauthorized modification.

An access control matrix will normally be sparse, since subjects have on average access to few objects. A simple example is shown in Table 17.3. Objects are the financial, the personnel, the stock and production files of the information system of an organization. The general manager only has read access to all files. Although she is the most important officer, she does not need to have greater rights in order to perform her function. This feature is very important, and is called the *least privilege* rule (Saltzer and Schroeder, 1975). Another very important rule in protection of information is the *need-to-know* rule: regardless of his position in the organization hierarchy, a subject should only be given access to the information needed to perform his work. The example follows this rule in general: the managers only have rights over the objects related to their functions. In consequence, a subject should be given *the least amount of rights to the least number of objects* possible in order to perform her job.

Searching a column of the matrix of the figure is equivalent to searching the ACL of an object for the rights of a given subject. Searching a row of the matrix is equivalent to finding the capability of a subject for a given object.

17.10.3 Discretionary Access Control

Until now, we said nothing about the form that the ACL, capability, or ACM may take. Who defines what a subject can do with the objects she creates or owns? Most systems do not impose any a priori restriction on such definitions.

Table 17.3. The Access Control Matrix

	Objects			
Subjects	finance	personnel	stock	production
General Mgr.	r	r	r	r
Finance Mgr.	rwcd	—	r	r
Production Mgr.	—	—	rw	rw

A subject can give whatever access rights she decides to the objects she creates or controls. We say these systems where control is subject-based follow a *discretionary access control (DACC)* policy, which is essentially a lack of policy for that matter. Note that this does not imply any lack of quality or effectiveness of the access control mechanisms just discussed, we are now at a slightly higher level of discussion: what are the rules to fill in the access control tables, that is, to decide which objects subjects or groups of subjects may access, and in what manner. The set of these rules is called the *access control policy.*

Consider subject s and object o. In a DACC policy, the access rights r granted to s on o solely depend on an ad hoc strategy F of the administrator or the owner of o, towards s, decided on a case-by-case manner:

$$DACC: \quad \langle s, o, r \rangle = F(s, o)$$

The DACC policy as defined by F can change dynamically, and can vary at the administrator's or owner's will. Discretionary access control is obviously the policy followed by most commercial-grade operating systems and database management systems.

17.10.4 Mandatory Access Control

Discretionary access control makes it impossible to enforce an access control policy, since access rights may change dynamically according to the current rights and the will of the users. If there is no formal and automatic way to check access control of authorized users, there is also no way to control malicious software, such as Trojan horses. A more restrictive policy is required, where users are not allowed to change access rights of objects, even if they own them, if that change is against some highly defined access control policy. We say these systems follow a *mandatory access control (MACC)* policy[1].

In order to follow a MACC policy, each subject or object must have a static security *class* or *label* (also called classification or clearance). Consider subject s with security class $C(s)$, and object o, with security class $C(o)$. The access

[1]The acronyms used are normally DAC and MAC, but we wish to avoid any confusion with other terms such as Message Authentication Code, MAC.

rights r granted to s on o solely depend on a static strategy F, which is a deterministic function of the security classes of both, and not of the particular subject or object. In consequence, the owner of o cannot change the access rights for s, if that violates the access control policy rules:

$$MACC : \langle s, o, r \rangle = F(C(s), C(o))$$

Now that you have understood the difference in level of abstraction between access control mechanism and access control policy, we may introduce an even higher level of abstraction: the *security policy.* A security policy is the top, human level, set of rules to enforce security in an organization. We are obviously interested in computer-supported organizations. A security policy dictates, amongst other things, the access control policy, that is, the rules to form F, which in turn will be implemented by the access control mechanisms. The access control policy is either a MACC or a DACC policy, or a MACC policy complemented by a DACC policy. This whole strategy is concerned with protection models, that will be studied in Section 18.7.

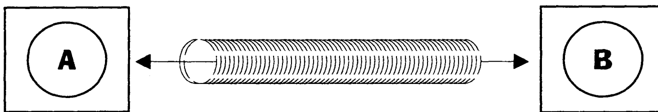

Figure 17.18. Secure Channel

17.11 SECURE COMMUNICATION

In an analogy with reliable communication, secure communication means ensuring that two or more principals (users, machines, protocols), communicate with security, despite the occurrence of malicious faults (attacks). There are essentially two ways of achieving *secure communication* over insecure media: secure channels and secure envelopes.

Indeed, a sufficient condition to achieve secure communication, is that principals are capable of building a *secure channel* between them. The channel is an abstraction that can be realized physically (*see* Figure 16.2) or virtually (*see* Figure 16.3). Observe Figure 17.18. Suppose A and B are connected by a pipe as shown, so that the pipe goes directly from A's mouth to B's ear, with no bifurcation. The pipe is completely opaque, so that no one can see what it carries. Finally, the pipe's material is also hard, so that no tool can penetrate it, inject or suck things from the inside, without that being noticed. That's a secure channel, and in fact the attributes we have just exemplified can be translated into the following properties:

authenticity - what B receives was sent by A, who cannot deny;

confidentiality - only B reads what A sent;

integrity - what A sent cannot be altered without detection.

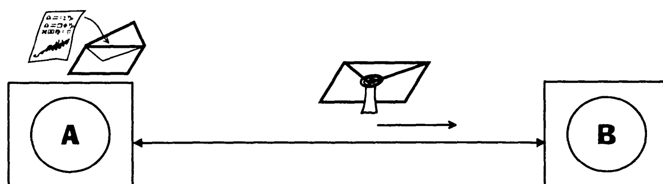

Figure 17.19. Secure Envelope

A secure channel is so to speak a form of *immediate* communication, and applies to the traditional forms of computer communication or telecommunication links, such as local area network protocols, TCP/IP, X.25, an so forth. By contrast, we identify a form of *deferred* communication, where it is desired to secure messages whose delivery is sporadic and deferred in time. Electronic mail is a perfect example of this. In that case, messages must have standalone security, that is, be wrapped in a *secure envelope*, which should enjoy the same security properties of the secure channel above: authenticity; confidentiality; integrity. Figure 17.19 depicts the principle of the secure envelope. A message is signed, put inside an opaque envelope, and the envelope is sealed. The whole packet has *standalone security* while traveling through an insecure medium: it cannot be changed without that being noticed (integrity); it cannot be read (confidentiality); and once opened, the signature can be verified (authenticity).

17.12 SUMMARY AND FURTHER READING

This chapter discussed the main paradigms concerning security. It addressed basic concepts of security, such as the trusted computing base and the foundations of modern cryptography. A detailed study was made of the main cryptographic paradigms: symmetric and asymmetric cryptography; secure hashes and message digests; digital signature; and digital cash. Authentication, access control, and secure communication complete the set of paradigms studied in this chapter. A remarkable text on secure channels, simultaneously simple and comprehensive, is Needham's chapter in (Needham, 1993). Additional cryptographic protocols can be found in (Schneier, 1996; Ryan et al., 2000; Menezes et al., 1999). A reference publication for real random numbers is the Rand Corporation million-number table (RAND, 1955). For an in depth discussion on digital cash properties, see (Okamoto and Ohta, 1992). Digital cash and guardians are detailed in (Brands, 1995), (Chaum, 1992), and (Boly et al., 1994). For further material about steganography, see (Wayner, 1993). A comprehensive treatment of authentication is found in (Kaufman et al., 1995). (Pfleeger, 1996) and (Abrams et al., 1995) do a thorough study of protection and access control mechanisms and security policies. The principles of intrusion tolerance, as opposed to prevention, are laid down in (Deswarte et al., 1991). For discussions on secure group communication see (Schneier, 1996; Reiter, 1996). Another challenging problem is maintaining keys in a group communication system, where members enter and leave (Steiner et al., 1998).

18 MODELS OF DISTRIBUTED SECURE COMPUTING

This chapter aims at providing the architect with a global view of the problem of security, by showing where the paradigms learned in the previous chapter, fit in the several models of distributed secure computing. It starts by discussing the main classes of malicious faults and errors expected in computer systems, and in distributed systems in particular— that is, attacks and intrusions. Then, it equates the main frameworks and strategies for building secure systems— authentication, secure channels and envelopes, protection and authorization, and auditing— as a form of bridging from the detail of paradigms to the global view provided by models. Finally, specific models for distributed secure computing are presented.

18.1 CLASSES OF ATTACKS AND INTRUSIONS

18.1.1 Computer Misuse

Not all types of security-related incidents are perpetrated by hackers (nonusers) through sophisticated techniques. We have already discussed the importance of negligence or occasional abuse of authorized users. *Computer misuse* designates actions and attitudes, by users and nonusers, aimed at impairing the security of computer systems. Abrams et al. introduce a comprehensive classification of computer misuse with increasing degree of severity and sophistication (Abrams et al., 1995):

P. Verissimo et al. *Distributed Systems for System Architects*
© Kluwer Academic 2001

Human error	Accidental human mistake of an authorized user causing a vulnerability that may lead to intrusion. For example, giving world read/write permissions to a confidential file, or creating an account without password
Abuse of authority	Intentional action of an authorized user abusing the authority granted by his activity. For example, a bank teller setting-up schemes of bogus transactions that leak a few cents each time to his personal account
Direct probing	Attack made by an unauthorized user (or nonuser) to a system by means of passively exploiting existing vulnerabilities with the aim of intrusion. E.g., entering through a forgotten account with default password, or using a stolen or guessed password
Software probing	Attack made by a nonuser to a system by means of passively exploiting existing vulnerabilities with the help of specially built malicious software, with the aim of intrusion. For example, use of a *Trojan horse* that pretends to be the login program, logging inadvertent users while it steals all their passwords
Penetration	Attack made by a nonuser to a system by means of actively exploiting existing vulnerabilities in the protection mechanisms of the system, normally with the help of specially built malicious software, with the aim of intrusion. E.g., sending a malicious HTTP request that confuses the HTTP server, leading it into giving an anonymous intruder total (root) control
Subversion	Attack made to a system, either at design or runtime, by designers, or by authorized or unauthorized users, by means of covert and methodical undermining of the protection mechanisms of the system, with the aim of continued intrusion. For example, by modifying operating system programs in order to introduce *trapdoor* that covertly perpetuate the intrusion

Human error and *abuse of authority* have little to do with computer technology, since a computer can hardly tell whether a legitimate user is using or abusing it. In fact, current solutions to this kind of misuse lie with ensuring users behave as they should, by means of inspection and auditing. *Probing*, whether manual or automated with the help of software, involves the discovery and/or direct use of vulnerabilities of the system. The attacker passively exploits the way certain functions are wrongly configured, or turned off, or else uses information obtained by other means (this is akin to a burglar entering a facility with a stolen key, pretending to be a legitimate user). *Penetration* bypasses the protection mechanisms, questioning their effectiveness. In consequence, it is an active and very aggressive attack against the implementation of the system's security policy (this is akin to the same burglar entering the facility, disabling the alarm, and guessing the safe combination). *Subversion* is the ultimate attack. It can be made either at design time— by one of the engineers, or at distribution time— by inserting malicious patches in down-

loadable code, or at runtime— after a successful intrusion. For example, the latter can be done by replacing crucial operating system programs— such as login, remote login, password change, auditing and monitoring, command line editor— with a complete kit of malicious programs with trapdoors and Trojan horses. Very often, a computer system is threatened at different times and in a pre-planned sequence. In generic terms, this is an **intrusion campaign**, and its consequence for the targeted organization is a *security hazard.* In general terms, intruders use a combination of the described types of computer misuse, combining human related misuse with attacks exploiting different vulnerabilities. Technically, it is usual to separate attacks between active and passive, with regard to the kind of interaction between the intruder and the target system.

18.1.2 Active Attacks

Active attacks are characterized by aggressive attempts to penetrate the system, disrupt its operation, and/or steal, modify or destroy data. Attacks may be directed at networks, machines, services or information.

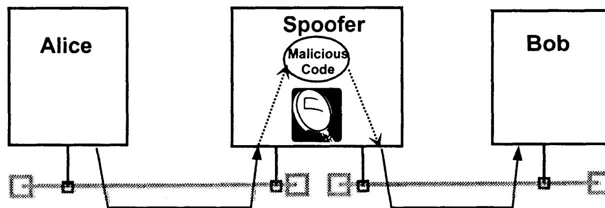

Figure 18.1. Spoofing Attack

For example, penetration attacks may be directed at an internal network by breaking through a firewall, or at a host, by defrauding its access control mechanisms, and stealing information. Penetration may instead be directed at a specific protocol, to gain control of the interaction, such as a home banking transaction. **Spoofing** is a specific form of such an attack, whereby a malicious host intercepts communications between two participants, changing its contents dynamically. The modification may take several forms: insertion/deletion or replay of whole messages, on-the-fly modification of message content. This last form is depicted in Figure 18.1. This attack can be used to penetrate cryptographic communication, or to append malicious software to downloaded programs. A well-known attack on Internet protocols is address spoofing, whereby source addresses of IP packets are modified to make them look like they are coming from somewhere else. The attack may be prepared by a previous attack on a DNS server that modifies a whole set of $\langle name, address \rangle$ pairs.

Disruption attacks are addressed at resources, for example communications jamming or bringing an operating system down. These attacks may be directed at specific services, hence the name **denial-of-service** attacks. A typical such attack is email spamming, whereby email servers are flooded with phony mes-

sages, to the point of being brought down. Modification attacks concern the contents of information repositories, e.g., databases, by modifying or destroying them. Specific Web-based services have also been attacked lately as a form of sabotage. These are either disruptive attacks, or modification attacks addressed at the content of the pages.

Attacks can be made either directly or through malicious software, such as viruses, worms, bombs, trojan horses and trapdoors. A **virus** is a software module that is appended to genuine programs. When the program executes, the virus is activated and performs the planned attack, often a mix of penetration, modification and disruption. Viruses, besides attacking the system, try to reproduce themselves, by infecting other programs in the machine.

A **worm**, unlike a virus, is an autonomous program which in general performs penetration attacks. Worms, once arrived at a host, install themselves taking advantage of vulnerabilities of the victim O.S., and prepare the assault of the next host. Besides migrating, they can copy themselves like viruses do.

A **bomb** is a disruption attack consisting of inserting a malicious software module inside a program or a system that perform some useful function, such as a database server. When activated, it will do something destructive, from blocking the server to deleting the database. The detonator is built by modifying the original program or O.S. configuration, so that the bomb is fired by a logical condition (logic bomb) or at a given date (time bomb). Bombs have been used very often for blackmailing companies.

A **Trojan horse**, or trojan for short, is a program that replaces legitimate system or user programs, and performs attacks on the system. Its name derives from the fact that it also executes the function that the original program was supposed to, in order not to be detected. A **trapdoor** is a special kind of trojan, normally a software module inserted inside a system, either during or after design, in order to subvert the access control mechanisms and let non-authorized users in.

A **trapdoor** is a piece of code inserted in a software module, either during or after design, providing a means of accessing a system other than the usual access procedure. A special kind of trojan consists of an access control mechanism with a trapdoor inserted, in order to let non-authorized users in.

A **covert channel** is a subtle form of information-theft attack. It consists of an indirect communication channel outside the reach of the access control mechanisms, which can be used to disclose information to an unauthorised user through a non-detectable means. A primitive channel would consist of having the leaking program encode the leaked information inside digital pictures served by a Web server on the same host (we briefly mentioned the underlying technique, called steganography, in Section 17.8). These channels are sometimes extremely subtle and strange, such as having the leaked information encoded as a pattern of disk usage (rythm of access, pattern of allocation), to be read and decoded by any non-privileged program in the same host.

18.1.3 Passive Attacks

In general, we term passive attacks those which, unlike active attacks, do not require an explicit action against the security mechanisms of systems or the integrity of their information. The hacker merely uses what is within reach, sometimes materialized or made possible by human error, or abuse of authority, such as reading files incorrectly open to the world, or logging in with a guessed password. Typical passive attacks are reading attacks, often aimed at gathering information for planning an active attack, for example, following the execution of a protocol, or logging login/password pairs passing on a network.

Probing, to explore basic system vulnerabilities, is the basis of many passive attack techniques. Internet IP discovery and port scanning is the basic technique to unveil vulnerabilities of networked installations. Password guessing is the basic technique against hosts using password authentication. The same idea applies to key guessing, for cryptographic protocols. These attacks are normally addressed at an encrypted form of the password or key, e.g., such as found in a password file, by exhaustively trying off-line the possible combinations, until a match is found with the encrypted item. This is called *off-line guessing*, and practical attacks are made with programs that test combinations of a glossary of probable words to be used by the password or key owner. Such a **dictionary attack** includes dictionaries of the language, and personal information that may be grabbed, for example, from the subject's Web page. These programs are as useful to hackers as they are to system managers, who may detect poor passwords and instruct the owners to change them. A popular such tool is the `crack` for UNIX.

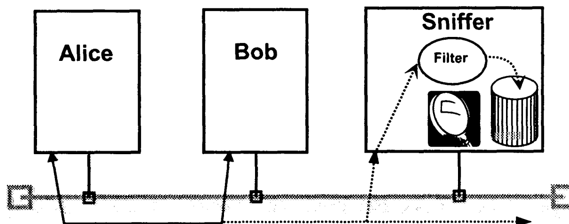

Figure 18.2. Sniffing Attack

Sniffing is the typical passive attack on a network, whereby the intruder exploits the broadcast nature of propagation of a medium, such as wireless communications or local area networks. The attack over a LAN is depicted in Figure 18.2, where the sniffer is a machine whose LAN adapter is configured to receive all the passing frames (promiscuous reception). Information received is filtered and stored in disk, to be used later. Sniffers are normally network management and debugging tools, such as `tcpdump` for the Internet. However, if misused, they become very nasty artifacts, because of their stealth nature: it is very hard to trace a sniffer, since it is absolutely passive.

File or database **snooping** is a passive attack directed at stored data, very frequently enabled by erroneous configurations and other vulnerabilities such as default passwords or wrong access permissions, easily discovered by methodical but discreet probing, also called doorknob rattling.

Although less destructive than active attacks in appearance, passive attacks are much more difficult to trace, exactly because of their frequently stealth nature, and are very often the first line of a final active attack.

18.1.4 Intrusions

In terms of effect, the results of successful attacks to computer and communication system components fall into one of the following general *intrusion categories*:

Resource Theft	Unauthorized use of computational or communication resources
Resource Disruption	Malicious disturbance of the service provided by computational or communication resources
Information Theft	Unauthorized interception, disclosure and/or use of information such as data and/or software, in computing or communication systems
Information Modification	Unauthorized disturbance of the content of information in computing or communication systems, by alteration, deletion, and/or forging

This classification emphasizes two important issues: the separation between computing or communication *resource* security, addressed by the first two, and *information* security, addressed by the last two; and the separation between *stealing* and *disturbing*.

These distinctions are important in defining strategies, since different organizations may have different policies about the relative importance of information versus resource security, or confidentiality versus integrity and availability. It is also important because different techniques apply to each category of intrusion.

18.1.5 Attacking a Cryptographic System

Let us analyze now specific attacks to cryptographic systems, which can take several forms: attacks on the algorithm; attacks on the messages; attacks on the keys; attacks on the protocol.

Any or all of these attacks can obviously be combined in an intrusion campaign against a cryptographic system. The algorithm is a crucial component of the crypto system. One of the main reasons why algorithms should be public is in order for cryptologists to know its structure, test its security, and eventually build trust on it, if it is not broken, or **cryptanalyzed**. Attacks on algorithms normally follow a few known patterns, or a combination thereof, based on what the cryptanalyst knows. In a *cyphertext-only* attack, the attacker only has ac-

cess to encrypted material. This is the basic attack, but yields very little to work with for resilient algorithms. The attacker needs more information to be effective, for example, he can manage to get hold of cyphertext blocks of which he knows the plaintext (for example, by acting as a legitimate user). This is known as a *known-plaintext* attack. Getting more sophisticated, he may try to manipulate the cryptosystem such that it encrypts selected blocks of data, and then collect the corresponding cyphertext. This is called a *chosen-plaintext* attack. This material will serve to look for vulnerabilities that invalidate one or more of the rule-of-thumb properties of a good cryptosystem listed in Table 17.1 of Section 17.2.

Attacks on messages have to do with the attacker playing with cipher blocks in transit in a secure channel or envelope, by changing, replaying or reordering whole blocks. The reuse of a signed text or digital cheque, or the forging of a financial transaction using blocks of genuine transaction, are examples of such attacks. Ciphers should be strengthened with information used only once, that is called **nonce**, such as time, sequence numbers and random quantities.

Attacks on keys have two facets. The first addresses the fundamental limit of computational ciphers, which is the feasibility of making a brute-force attack, by testing all possible combinations. The second facet has to do with guessing attacks. Long keys may render the first attack infeasible, but a cryptosystem can fall long before that limit is reached, given a poor choice of keys. Keys are susceptible to dictionary attacks as are passwords. Even brute-force attacks may become fast enough, if the key owner uses a subset of the character space (e.g., lowercase, alphabet/only), since the search space is decreased.

A crypto system is materialized by a protocol. This protocol may have logic vulnerabilities, that have nothing to do with the algorithm but with the way it is implemented in a real system. The attacker studies the protocol, and performs active attacks with the aim of confusing it and cause it to take wrong decisions, such as surrendering a shared secret key, or letting the attacker into a server without completing his authentication. For example, a spoofing attack can undermine an otherwise correct crypto system. The types of attacks described show that a cryptosystem does not live just on a good algorithm. In fact, a system using an unbreakable algorithm may fall under naive message formats, poorly chosen keys, or protocol bugs. In consequence, an architect should always have a systemic view towards solving security problems.

18.2 SECURITY FRAMEWORKS

After discussing the possible threats to system security, we spend this section analyzing the main frameworks with which the architect can work in order to build secure systems, introducing some of the models addressed later in this chapter, such as authentication and key distribution with key distribution centers and certification authorities, hybrid cryptography for secure channels and envelopes, or protection via discretionary or mandatory access control policies.

18.2.1 Secure Channels and Envelopes

Secure channels and secure envelopes are basic paradigms of secure communication studied in Section 17.11. They form a framework for setting-up distributed processing services with security, such as remote sessions, file transfer, RPC, Web servers, email, messaging services, on-line transactions, and so forth.

Secure channels are normally set up for regular communications between principals, or communications that last long enough for the concept of connection to make sense. For instance, file transfers or remote sessions, among the ones just mentioned. They live on a resilience/speed tradeoff, because they are on-line, and may use combinations of physical and virtual encryption. Secure channels adopt per-session security, and normally use symmetric communication encryption, which provides the best performance, but requires attention in terms of resilience. Practical protocols should protect their keys, and reveal as little as possible about the communication channel that may be useful for a cryptanalist.

Secure envelopes are used mainly for sporadic transmissions, such as email. They resort to per-message security and may make use of a combination of symmetric and asymmetric cryptography (also called hybrid) as a form of improving performance, especially for large message bodies. However, they are not so susceptible to the resilience/speed tradeoff, given their deferred nature. This is relevant for ultra-sensitive communication, where longer keys and asymmetric encryption may be used, to increase resilience of the ciphertext.

Section 18.4 is going to discuss practical issues of cryptographic protocols. Section 18.10 addresses the establishment of secure channels and the generation of secure envelopes, and continues building on top of that, presenting models for remote sessions, remote client-server with procedure call, and electronic mail.

18.2.2 Authentication

There are three forms of authentication paradigm, as studied in Section 17.9, and they make sense in different situations. Essentially, unilateral or mutual authentication are chosen depending on how important it is, for each principal, to identify its peer with certainty. There will be systems where authentication is compounded with the service itself, such as a centralized system with remote and distributed access. Unilateral authentication by password is the most used. However, when threats are expected, some form of cryptography should be envisaged. It is then common to provide users with a secret key shared with the server, so that they can execute such a protocol, performing either unilateral or mutual authentication. When there are several services and many users, this becomes impractical. Then, mediated authentication is one solution, where a distinguished service is created just for the purpose of authentication: a key distribution center (KDC). Now, users only share secrets with the KDC, and when they wish to authenticate to other services or users, they use its mediation. In large systems, an alternative may be to provide users and services with public/private key pairs, so that every public key can be found in a specialized

server, a certification authority (CA), that provides key certificates signed by it. Powerful asymmetric signature and encryption protocols can then be used for authentication, key distribution and long-term encryption.

A remaining issue is the significance of a given authentication process: who is being authenticated? When we request a call-back number in a computer dial-up, we believe that the number represents a user. When we use link encryption, we just trust the secure channel between the link extremities, not what is beyond the link. However, when we use virtual circuit encryption, we trust an end-to-end secure channel, for example between a client program and a server program. When we provide the user with authentication gadgets, such as smart cards or biometric devices, this is because we do not trust the client program to stand for the user at all times: the host can be intruded, an intruder can logon impersonating the user. We have just discussed delegation, that we will address together with authentication and key distribution in Sections 18.5 and 18.6.

18.2.3 Protection and Authorization

Protection, a fundamental framework in secure computing, is about restricting the access to and use of information and programs, to authorized users. As such, the central paradigm is *authorization*. The access control and trusted computer base (TCB) concepts form the underlying basis on which to build protection models. Access control, that we studied in Section 17.10, is performed by assessing the rights of subjects to access objects, those rights being defined by access control lists or matrices, or by capabilities. Normal operating systems however, do not provide the adequate guarantees of trustworthiness to run sensitive access control mechanisms. The TCB, that we introduced in Section 17.1, provides the necessary notion of secure, tamperproof base.

One fundamental model for protection is the reference monitor (RM) model, based on running all access control functions on a TCB, and obliging all requests for system resources to go through the RM. The RM does not necessarily stipulate how access control to objects should be given to subjects. However, in order to fulfill high-level security policies in a verifiable way, access control mechanisms should not be configured by users or administrators at their own will, which is how they are often implemented, but instead dictated by access control policies (sets of rules), such as the statement that "an unclassified subject cannot read from a top-secret object". The former and the latter policies are called respectively discretionary and mandatory access control policies.

Despite what we just said, protection (taken in a broad sense) starts with good architectural procedures. Physical separation by architecture and topology of the network is the first step to protecting a distributed system. Secondly, machine protection, further reducing the level of threat by reducing the exposure of services, for example, by putting restrictions on remote access, dial-up, or the way administrators perform sensitive services. Firewalls come next, as a means to enforce both physical and logical separation, since containment is achieved at both architectural and protocol levels. These measures can be further refined through the separation of concerns between machine protection

and data protection. It should still be difficult for an intruder that breaks into a machine, to further get to users' or services' data. This has to do with the logical architecture of services and applications, and the adequate use of distribution, fragmentation, replication and cryptography. All these notions will be expanded in Sections 18.7, 18.8 and 18.9, where we will study models for protection of systems, from architecture, including firewalls, to formal models.

18.2.4 Auditing and Intrusion Detection

Logging system actions and events, or in other words, performing an **audit trail** of the system, is a good management procedure, and is routinely done in several operating systems. It allows a posteriori diagnosis of problems and their causes, by analysis of the logs. Audit trails are a crucial framework in security. Not only for technical, but also for accountability reasons, it is very important to be able to trace back the events and actions associated with a given time interval, subject, object, service, or resource. Furthermore, it is crucial that all activity can be audited, instead of just a few resources. Finally, the granularity with which auditing is done should be related with the granularity of possible attacks on the system. Since logs may be tampered with by intruders in order to delete their own traces, logs should be tamperproof, which they are not in many operating systems. A broader perspective on this subject is provided by **intrusion detection systems (IDS)** (Debar et al., 1999). An IDS system is a supervision system that follows and logs system activity, in order to detect and react against attacks and intrusions, preferably in real-time, that is, with low latency.

18.3 STRATEGIES FOR SECURE OPERATION

Strategy is conditioned by several factors, such as: type of operation, acceptable risk, price, performance, available technology. Technically, besides a few fundamental tradeoffs that should always be made in any design, the strategy of the architect for the design of a secure system develops along a few main lines that we discuss in this section.

18.3.1 Fundamental Tradeoffs

There are a few fundamental tradeoffs to be considered by the systems architect:

- cost vs. effectiveness
- performance vs. security
- robustness vs. lifetime
- degree of vulnerability vs. level of threat
- cost of securing vs. cost of intruding
- prevention vs. tolerance of attacks
- prevention vs. detection of modification
- detection/recovery vs. prevention of fraud

- cost of security vs. cost of breaking

The cost versus effectiveness tradeoff may be patent for example in how far goes the effort to implement a real TCB, since there is a spectrum of alternatives from purposely made, high-coverage security kernels, to adapted, and necessarily more fragile, commercial operating systems. Security can mean adverse performance in several instances. For example, robustness of a cryptographic channel depends on the key length, but the longer the key, the slower the channel. Asymmetric cryptography is deemed more robust than its symmetric counterpart, however it is much slower, by approximately three orders of magnitude (*see* Table 18.1). The cost versus effectiveness tradeoff yields to the need for performance when both speed and robustness are required, implying the use of cryptographic hardware, versus software-based cryptography, inexpensive and robust, but much slower (*see* Table 18.1 for typical speed-up factors). On the other hand, robustness of a channel is inversely proportional to its lifetime, since more material can be revealed to an attacker. Long-lasting channels should refresh their context regularly (e.g. keys).

The degree of vulnerability of system components cannot be reduced arbitrarily. In other words, *vulnerability removal* must be balanced with the reduction of the level of threat the system is subjected to, in order that the risk of operation remains acceptable. Reducing the level of threat is the path of *preventing attacks*, whereby the system gets protected from certain attacks. Attack prevention is advised for long-term secrets, that is, information that must remain confidential for a long time. In this case, even a cyphertext-only attack presents a risk if one is not certain that it will resist a brute-force analysis during the lifetime of the confidential information. However, most of the time acceptable risk does not imply absolute intrusion-free operation, the cost of which may be overwhelming. Instead, it is acceptable for the cost of intruding to be significantly higher than the value of the service being offered. This may justify the use of weaker but faster, cheaper or simpler cryptosystems, for example. However, this approach has its limitations: it is impossible to guarantee that all attacks are prevented, given the unpredictable nature of attackers; it is sometimes not possible to prevent attacks at all, given the open nature of some applications. So, we better seek for an alternative to vulnerability removal plus attack prevention, which we may call *intrusion prevention.* Essentially, this means admitting that attacks will take place and lead to intrusions, what we might call *intrusion tolerance.* The latter consists of letting the system be attacked and intruded upon, but providing it with the means to resist and avoid failure of the security properties.

Which properties to secure represents another class of tradeoffs. For example, integrity can mean prevention of modification, or its detection. The latter is ensured with cryptographic techniques. However, the former requires protection, and sometimes redundancy. Computer fraud has legal implications that make the tradeoff between detection/recovery and prevention a delicate one. Fraud prevention may reveal itself cumbersome (more complex and slower

Table 18.1. Figures of Merit of Several Cryptosystems

	Variants	
Algorithm	**SW: Speed(Pr/Key)**	**HW: Speed(Pr/Key)**
DES	1.2Mb/s (66M4/56b)	512Mb/s (32M/56b)
DES	9.3Mb/s (100M5/56b)	640Mb/s (-/56b)
3DES	0.4Mb/s (66M4/56b)	214Mb/s (-/56b)
IDEA	2.4Mb/s (66M4/128b)	177Mb/s (25M/128b)
RSA encr	5Kb/s (66M4/512b)	220Kb/s (-/512b)
RSA decr	320Kb/s (66M4/512b)	
RSA sign	0.16s (S2/512b)	
RSA ver	0.02s (S2/512b)	
DSA sign	0.20s (S2/512b)	
DSA ver	0.35s (S2/512b)	
MD5	5.9Mb/s (66M4/-)	315Mb/s (-/-)
SHA	2.6M/s (66M4/-)	253Mb/s (-/-)

protocols), whereas fraud detection is followed by a recovery process which normally takes place outside the system and is thus lengthy.

Table 18.1 gives some useful insight on what is behind some fundamental tradeoffs just discussed. Speeds of cryptographic material are presented (either in Mb/s or sec.), differentiating between types of algorithms, and hardware and software implementations for several key lengths (main sources: (Garfinkel and Spafford, 1997; Lampson, 1993; Schneier, 1996)). Processor codes are in the form xMy, where x is speed in MHz and y is: 4- i486; 5-iPentium; S2- SUN Sparc II. Key lengths in bits. We have tried as much as possible to harmonize figures, and ended up using the today obsolete 66MHz i486 (66M4). However, we compare with DES figures for the 100MHz Pentium (100M5), to give you an idea of the evolution. We would have to review the book every 6 months to cope with technical progress. It should be easy to extrapolate to faster machines, if you have good comparative benchmarks at hand. Look for computation intensive ones. RISC architectures, for example, do remarkably well because of pipelining and internal parallelism: note that the speed-up in DES from 66M4 to 100M5 is almost 8 times, quite a bit higher than the clock speed-up.

18.3.2 On Keys and Passwords

Keys and passwords are the most fragile components of a secure system, and we know a system breaks by its weakest link. So, this section gives a few hints on their correct use. We will use 'key' to denote both key and password here, unless a distinction is necessary.

A brute-force attack is the ultimate barrier on a key or password. A pool of 200 hardware chips featuring 256M encryptions/sec. will break a 56-bit key (e.g. DES) block cipher in 2 weeks. To make things worse, hardware speed and power are going up at an incredible pace. However, that time goes up to

an unfeasible 2×10^{20} years, if the key length is 128 bits (e.g. IDEA). Long-term asymmetric keys provide even more credible protection. For example, a 1024-bit key cipher is un-attackable just by brute-force if the key is random.

A key is weak if it has redundant information. This has the effect of reducing its equivalent length. For example, a 56-bit key using only lowercase letters and digits "shrinks" to around 40 bits. The attack exemplified above would succeed after a dangerously short 20 seconds. If keys have lexical content, they become amenable to dictionary attacks, which can make an attack even faster. (Please, do not use "dowjones" or "dragonball" for a key or password). Incidentally, 40 bits is the key length of several U.S. commercial encryption products (e.g., the export version of U.S.-made SSL), derived from the export restrictions. These were partially lifted in the end of 1999 (e.g., full Netscape and PGP are now freely exportable). The restrictions have been partially lifted. For example, SSL can now work with 128-bit cryptography all over the world.

How to generate a good key or password then? Good cryptographic systems have resorted to the **passphrase** stratagem. A passphrase is a long, intelligible sentence, known only to the user. When the user enters it, the system applies a cryptographic checksum to it, generating a high quality key or password. A rule of thumb for practically random content is one passphrase letter per password bit, but shorter phrases are normally enough. For example,

"My dear friends, who on earth would believe this is my passphrase?"
might yield after hashing something like the 64-bit hexadecimal quantity
`E6C1 0A9B 894E 03AF`
a good albeit hard to memorize password

Keys can be strengthened by combination. Long asymmetric keys may protect shorter but faster symmetric keys, the former being called **key-encrypting keys**. Practical protocols will perform *key rollover* as a routine, that is, change session keys often, even during a session. The robustness of fixed key-length ciphers (e.g. DES, IDEA) can be increased to approximately twice by running the protocol three times, which is known as *encrypt-decrypt-encrypt*. Finally, keys should be made both to the measure of the attacks they must withstand and to the lifespan of the data they are to protect. Keys may be long- or short-term. Data may be long- or short-lived. A key should be planned to resist an attack that can be made during its lifetime. There are a few exceptions: a short-term key protecting long-lived data inherits the lifetime of the data; the lifetime of a long-term key protecting long-lived data, or protecting short-term keys that protect long-lived data, must be an upper bound of all data lifetimes. Some of these issues will be detailed in Section 18.4.

18.3.3 Vulnerability versus Threat

A four-digit PIN is not a weak password if it is to be keyed in by hand at an ATM, validated by a token such as a card. A brute-force attack on the 10^4 possible combinations is practically infeasible with this technology: it would take too long; it would be too conspicuous; the machine would confiscate the card after a number of attempts. However, suppose that the technology was

directly transposed to an environment where the card would be read in a PC and the card Id together with the PIN would be submitted via a network to the bank. Then we would have an extremely fragile system, because: an automated program would be able to test all the combinations very fast; and do it remotely, under the cover of anonymity, without any possibility of physical action on the hacker or the card.

What did we want to show with this example? It showed that vulnerabilities are sensitive to the level of threat. It also stressed a more fundamental issue that we had not mentioned yet. Security existed long before the widespread use of computers and communication facilities. There is a risk that procedures that were perfectly safe in a manual, electrical or mechanical world will completely fall apart in the high-speed world of modern computers and the pervasive Web of network links. A classical example of the modern era is the DES cryptographic algorithm: the U.S. authorities, when it was introduced in 1977, imposed a key length which they believed could be cracked by brute-force (testing approximately 10^{16} combinations of the 56 bits of the key) only by their own powerful machinery, and no one else's, for many years to come. Today, that task is within the reach of amateurs with good computational resources. So, beware of technological changes.

Level of threat and degree of vulnerability are not precise quantitative measures, but awareness of these two facets is a fundamental strategic issue: the architect must balance her design options in order to achieve the desired risk of operation. We can give examples of what may go wrong when one of the terms is not taken into consideration, leading to a false impression of security:

Example 1: Many mainframe-based OLTP[1] systems rely on private networks of point-to-point leased lines. We have found many often, among architects of these systems, a false feeling of security of the leased line (wrongly assumed low level of threat), which can be tapped and intercepted as any other. However, this feeling relaxes the requirements on security of the user-to-database interaction (high degree of vulnerability), leading to a high risk of operation.

Example 2: The Secure Sockets Layer (SSL) protocol reportedly ensures secure client-server interactions between browsers and WWW servers. Users have tended to accept that the interactions are secure (assumed low degree of vulnerability), without quantifying *how secure*. Netscape's implementation of the SSL protocol was broken because of a bug that allowed to replicate session keys and thus decrypt any communication. The corrected version was then broken at least twice through brute-force attacks on the export version (at that time, the only one available to companies outside the U.S.A.), which used short (40-bit) keys. Several companies have built their commerce servers around SSL, some of them to perform financially significant transactions such as home banking. Some of these servers, because of the assumption of low vulnerability, were put on the Internet for spontaneous transactions with few or no restrictions to probing and access anonymity. Financial value and openess foreshadow a situation of high level of threat, leading once more to a high risk of operation.

---

[1] On-Line Transaction Processing.

No feasible system is 100% risk-free or secure. The level of threat and degree of vulnerability can be made to decrease in an asymptotic-like manner, where the limit is, respectively, a useless or an extraordinarily expensive system. Here is a good question an architect would like to see answered— *"What is the right balance?"*:

- There is no universal answer, but the level of threat is mostly dictated by the type and function of the system in question: Does it have to stand on the Internet or can it be behind a firewall? Must accesses be completely anonymous or can have clients identify themselves?
- The degree of vulnerability is dictated by the hardware and software used, and their configuration: Will a normal operating system be used, or a security-hardened one? Will strong or weak cryptography be used? Will strong authentication be made, or just O.S. access control?

A useful figure to equate the risk related with the operation of a given system is an estimate of the **cost to intrude** it under given conditions. For instance, when Netscape released the above-mentioned second version of the SSL implementation, they reported that it would cost at least USD10,000 to break an Internet session, in computing time. The cost of intruding a system versus the value of the service being provided allows the architect to make a risk assessment. Someone who spends 10 000 EURO[2] to break into a system and get 100 EURO worth of bounty, is doing a bad business. Unfortunately, these estimates may fail: shortly after Netscape's announcement, a student using a single but powerful desktop graphics workstation, broke the export version for just USD600. However, what went wrong here was not the principle and the attitude of Netscape, just the risk assessment they made, which was too optimistic. These estimates may also fail when values other than economic are at stake, such as political, for a potential attacker.

18.3.4 Open System Security Policies

The main vulnerabilities of open distributed systems lie in:

- Networks:
  - bugs of communication protocols
  - broadcast or wireless nature of networks
  - openess to anonymous access
  - the human element (administrator and user)
- Hosts:
  - bugs of the O.S. and widespread application software
  - wrong configurations (backdoors)
  - human element (administrator and user)

[2]The EURO is the currency of the European Union. It is worth approximately one USD.

There is a major threat to current operation of open systems, which is the general lack of strong authentication in current network and O.S. technology, leading to impersonation threats that can undermine the best cryptosystem, or the most conservative access control policy. In detail, the main threats to a networked system are the possibility of: scanning for resource and vulnerability discovery; eavesdropping with sniffers; active attacks with spoofers. On the other hand, the main threats on hosts are: the wide availability of exploits for almost any O.S.; automated penetration attacks such as viruses and trojans; dictionary attacks on passwords. A security policy, informal as it may be, is the key to securing any system. We suggest and discuss here a simple strategy for implementing a single-level security policy for an open system:

- selecting the system components;
- evaluating their vulnerabilities;
- adjusting threats to the desired risk according to a security policy;
- checking whether the right balance was achieved;
- iterate once more if not;
- implement the policy.

The **4P policies** are simple enough to be understood and implemented without much effort. They divide the strategies for protection into four possible choices, one of which is selected as the policy for the system in question. The 4P policies are[3]:

Paranoid	all is forbidden
Prudent	all that is not explicitly allowed is forbidden
Permissive	all that is not explicitly forbidden is allowed
Promiscuous	all is allowed

A *paranoid* policy is very simple to implement, because it means to completely isolate the system. It would be the implicit policy of enterprises in the old days. However, short of specific critical subsystems in an organization, it is not applicable to a whole organization in these times of connectivity and openess. The *promiscuous* policy is also simple to implement, because it lies in not making any access restrictions. It used to be the implicit policy of academic organizations in the old days of an "academic" and friendly Internet. It is doubtful that an organization's system following the promiscuous policy would survive long enough to do anything useful in these times of hackers and threat. The prudent and permissive policies will be the main workhorses in configuring the majority of systems. The *prudent policy* assumes that everything is forbidden unless explicitly allowed. It is the most difficult to implement, but the most effective. The difficulty lies in the nuisance it presents to users until it is tuned, if that state is ever reached. Its implementation consists of denying

[3]The credit for these mnemonic designations is attributed to BBN Cambridge.

all accesses by default, and explicitly opening all the desired accesses in whatever desired modes, one by one. Until the system stabilizes, users experience difficulty in accessing services not specified, which would normally be allowed in a more permissive configuration. The *permissive policy* assumes that everything is allowed unless explicitly forbidden. It is simple to implement, based on allowing all accesses by default, and selectively deny a number of accesses listed in a first specification of 'don'ts'. It is normally tuned at the cost of suffering attacks and closing the respective backdoors or suspicious accesses. A real system will normally be divided in subsystems, to which different policies may be allocated. Additionally, the starting point of the implementation of either the prudent or permissive policies may be a semi-allowed or semi-denied state, depending on the viewpoint.

18.3.5 Is a TCB Implementable?

The most effective protection or cryptographic mechanism can be defeated by a vulnerability, not in itself, but in the O.S. of the host if it is supposed to run securely that is, in a trusted computing base (TCB). The approaches to implement a TCB fall into two classes:

security kernel- designing an operating system kernel intended to be secure from the start and building the rest of the O.S. around it;

security enhancement- starting with an existing operating system and designing security in, by retrofitting or simply configuring the O.S.

The central question is *coverage*, exactly in the same sense that it was studied in Section 6.2 of the Fault Tolerance part: the probability with which the TCB properties hold. The *security kernel* approach is the only road to achieving extremely high coverage of the TCB properties: interposition, shielding, and validation. Alternatively, an operating system designer may select an existing operating system and improve it by *retrofitting* security into it, getting what is called a *security-enhanced kernel.* A milder form of security enhancement that does not involve source code modification is achieved by *configuring* an existing commercial operating system.

When attacks just take the form of probing, direct or by software, or the risk of sporadically successful penetration attacks (for unlikely) is accepted, a security-enhanced system will be adequate. Then, the choice between retrofitted or configured kernels depends on the balance between: price, performance, and accepted risk. Configured kernels may represent an adequate solution when: assets are of moderate value and removal of vulnerable functions does not compromise effectiveness of the service; or for protecting a first rampart, like a firewall, when there are complementary measures, such as intrusion detection. For systems which, despite the high value of the assets at stake, have to endure a high level of threat, such as open financial transactional servers, retrofitted kernels may be a sensible decision, in order to keep the risk of operation at acceptable levels. The security kernel approach for a TCB is definitely the solution to consider when the system holds very sensitive data and may be subjected to attacks at the level of penetration or subversion.

18.3.6 Preventing Attacks

The TCB concept attempts at eliminating vulnerabilities. It is part of a prevention strategy: if vulnerabilities disappear, attacks cannot be successful. This is feasible for very focused components, implementing critical functions, but is hardly a good strategy for the whole of the system. However, we can specialize the strategy to preventing attacks, the next step down the ladder, i.e., the malicious faults that take advantage from vulnerabilities. This has two facets: not letting attacks be produced; not letting them cause intrusions.

The first approach concerns all techniques for placing the system or the desired components out of the reach of attacks, that is, the direct reduction of the level of threat. For example, placing a sensitive database behind a firewall, without access from the outside. The second approach has to do with letting the attack be done, detecting it and acting after it is performed and before it succeeds in causing an intrusion. For example: by defusing a time bomb or deleting a virus before they are activated; by detecting scanning from a suspect host and blocking that host.

18.3.7 Detecting and Reacting to Intrusion

The prevention strategy is not always adequate. In this case, we must redefine the strategy to act one degree further down the ladder: attacks will be there, and cause intrusions. We can: detect, recover from, or mask intrusions. In fault tolerance terms, we might call this strategy *attack tolerance.* Tolerance and prevention are often used in combination.

Intrusion detection is a well known field. Intrusion detection systems (IDS) aim at detecting attacks and intrusions. They are based on native logging mechanisms of O.S.s and networks, and on specific instrumentation (sensors and actuators) placed wherever appropriate in the system, such as network sniffers, and system-call interceptors. Depending on the type of IDS, its detectors may be sensitive to different stimuli. Some are based on learning the *patterns* of attacks, and using a *knowledge* base of known attacks to detect them. Others learn the normal *behavior* modes of the system, and base intrusion detection on the occurrence of *anomalies* in that behavior. The latter are more adaptive, both to the evolving behavior of the system, and to new attacks, since they do not have to know attacks at all. Subsequent to detection, there is reaction, which may go from event alarms to automated or semi-automated countermeasures. Countermeasures take several forms, and are specific of the kind of attack and the make of the IDS. They may involve sanitizing attacked components (neutralizing viruses and Trojan horses), or neutralizing outside attackers, by barring their way in a firewall, for example.

Detection is not always accurate, or fast enough. The risk is for intrusions to take place that may cause failure before being detected. In this case, there is room for more sophisticated techniques involving *intrusion masking.* The principle is based on providing the system with redundancy such that in the presence of intrusions it continues to work correctly. For example, if a file is

encrypted, fragmented and scattered through several hosts, the intrusion of one such host will not reveal the file, regardless of whether the intrusion is detected or not (Deswarte et al., 1991). Threshold cryptography is also resilient to a number of malicious participants.

Past the real-time reaction to an attack, there is often the need to repair the system: removing the remnants of the attack (e.g., deleting Trojan horses, which may involve formatting disks); hole-plugging or removing the vulnerability that caused the attack (e.g., an account with a broken password). These actions are also called fault treatment in a fault tolerance sense.

18.3.8 Avoiding Disruption

Disruption attacks attempt at damaging data or causing denial of service. They affect the properties of integrity and availability. It is generally recognized that availability and integrity preservation are hard to achieve. Integrity preservation may start with attack prevention by limited physical separation. Next, it requires access control. However, replication should be used in order to tolerate successful attacks, that may damage some but not all the replicas.

On the other hand, the classical techniques of access control and cryptography provide little help against denial of service attacks. Some of them are external, and may come from spoofed hosts, making countermeasures extremely difficult. Authentication plays a role here, because it allows filtering out and even fight back the malicious external users. In consequence, this also suggests that allowing completely anonymous accesses to public servers favors denial-of-service attacks. Replication, inasmuch as it is used for availability with respect to accidental faults, may also yield positive results, if set up in a way that common-mode attacks to all replicas are prevented.

18.4 USING CRYPTOGRAPHIC PROTOCOLS

We studied four main cryptographic building blocks: symmetric encryption; asymmetric encryption; secure hash; and signature. Their functionality serves different purposes. Their relative speed also varies quite a lot: if we wanted to give a gross estimate of their relative performance, we would say that if asymmetric algorithms run at speed 1, then signature ones also run at speed 1, while symmetric run at up to speed 1000, and hashes run at speed 3000. Combined in several ways, they yield protocols that perform a number of interesting functions.

18.4.1 Protocol Types

Cryptographic protocols use and combine cryptographic algorithms in several forms. Generally speaking, there are two basic *protocol,types*, depicted in Figure 18.3, with regard to the approach taken to enforce correct behavior in the presence of malicious faults or attacks:

Self-enforcing	in which correct behavior is guaranteed by the protocol, which ensures trust among the participants
Trusted-third-party	in which correct behavior is guaranteed by a third party, which builds trust among the participants; trusted-third-party protocols assume three facets, depending on *when* trust is built:
adjudicated:	in which correct behavior is guaranteed a posteriori, by an adjudicator who in case of a fault will determine the responsible participant(s), by analyzing information (evidence) collected during the execution of the protocol; the participants trust that whatever may go wrong will be corrected by the adjudicator
arbitrated:	in which correct behavior is guaranteed during the execution, by an arbiter who ensures that the protocol executes correctly by participating of every execution; the participants trust that whatever might go wrong is prevented from happening by the arbiter
certified:	correct behavior is guaranteed a priori, by a certification authority who will issue and provide participants with certificates appropriate to build trust between them; the participants trust that nothing may go wrong that is guaranteed by the certification authority

In essence, a self-enforcing protocol, depicted in Figure 18.3a, guarantees that the properties of the service are achieved solely by the mutual interactions between the principals. Whatever faults occur, they are tolerated from within the protocol. That is, the protocol is a error-containment domain, hence its name. Since nothing comes free, the complexity normally migrates to the underlying algorithms.

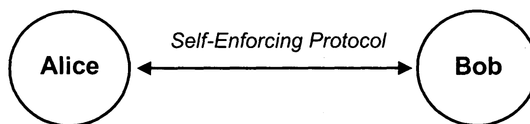

Figure 18.3. Protocol Types: (a) Self-Enforcing

The other two types both have in common the fact that they resort to the assistance of what is called a **trusted third party** (TTP). A TTP is an entity which is trusted by all principals to perform its functions correctly and impartially. As such, typical functions of a TTP are: adjudication, arbitration, and certification. Depending on the type of TTP, we call TTP-protocols adjudicated, arbitrated, or certified, as depicted in Figure 18.3b.

The decision about which one to use lies in the answer to the following questions: can we afford an incorrect behavior at all? Can we build trust beforehand (off-line), or do we need to watch every execution (on-line)? If a country's court system is efficient, we may go ahead in a business, knowing that if something goes wrong, we will be compensated. However, if it is slow

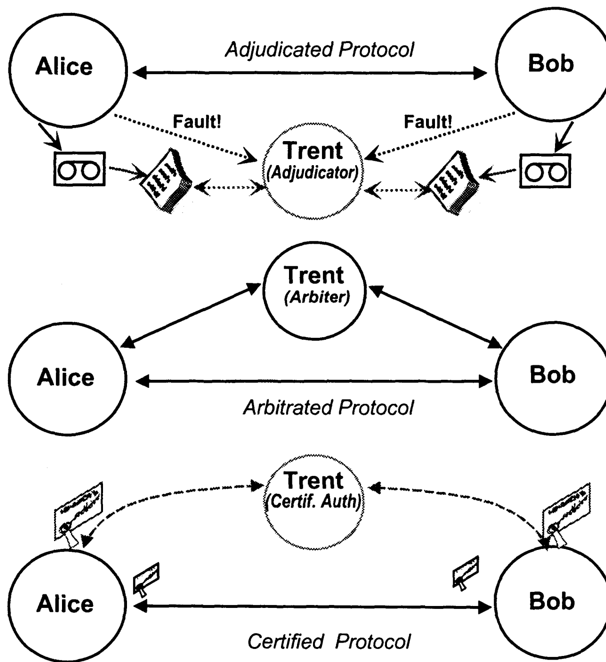

Figure 18.3 *(continued)*
. Protocol Types: (b) Trusted-Third-Party— adjudicated; arbitrated; certified

and unreliable, perhaps we had better not be cheated and take all necessary precautions: if we believe in the credentials our partners present, that is enough; otherwise, we should have someone trusted by both follow the negotiation.

Technically speaking, adjudicated protocols take the first approach, performing fault tolerance with long error latency in error recovery, whereas arbitrated and certified protocols take the second, performing fault prevention. There is a substantial difference in overhead between them: arbiters must always be on-line and thus they are a bottleneck and a single point of failure; certification authorities act prior to the execution of the protocol, and thence they may even be off-line, if certificates are obtained beforehand; adjudicators act in background and only when one of the parties suspects anything. The only constant overhead for the latter is the collection of evidence during the execution of the protocol.

18.4.2 Block Cipher Modes

There are several modes to use block ciphers. We will present them using DES as an example, but they can be applied to any block cipher. At first sight, the obvious way to use DES would be: to break the cleartext in 64-bit (or 8-byte) chunks; to do *padding* with some pattern to fill in the last block if the text is

not a multiple of 64 bits; to encrypt the blocks one at a time and concatenate the result. This is called the *Electronic Code Book (ECB)* mode. ECB mode is susceptible to analysis and replay attacks, since an 8-byte cleartext chunk always encrypts to the same cyphertext.

The systematic solution is to make blocks depend on one another. *Cipher Block Chaining (CBC)* applies feedback to a block cipher: the plaintext of block i is XORed with ciphertext of block $i-1$ and then encrypted. This way, Mallory cannot cut and paste a message arbitrarily. However, this is not perfect yet: he can still analyze/replay whole messages— such as encrypted password messages— because with CBC, two identical messages get the same ciphertext. The solution is to encrypt a block of random data as the first block, called *Initialization Vector (IV)*. The initial difference propagates to the whole message because of the feedback and so the same message never yields the same ciphertext twice.

Interestingly enough, block ciphers can be implemented as stream ciphers, in a mode called *Cipher Feedback (CFB)*, which allows encryption of an arbitrary stream of bits. *Output Feedback (OFB)* is yet another streaming block cipher, with interesting properties. It uses DES to generate a pseudo one-time pad, by feeding it with the ciphertext of the previous block (starting with IV).

MACs can be generated with Block Chipers. There is a standard MAC protocol, ANSI X9.9 (ANSI X9.9, 1986), which uses the *CBC residue* approach: the message is encrypted by DES in CBC mode, but only the last block is used as a 64-bit hash (*see* Section 18.4). The recipient verifies in the same way. Internet protocols, such as SNMP, use the HMAC protocol (RFC2104).

18.4.3 Double and Triple Encryption

It is generally believed that encrypting more than once with different keys improves the security of a block cipher. This is not always so, and in fact for DES double encryption ($C = E_{K_b}(E_{K_a}(M))$) would be much like using a 57-bit key, because of an attack known as **meet-in-the-middle** (Merkle and Hellman, 1981). However, triple encryption technique works, and can be very robust with a subtlety: the middle operation is decrypt, using two keys in the following manner: $C = E_{K_a}(D_{K_b}(E_{K_a}(M)))$. It is called *Encrypt-Decrypt-Encrypt (EDE)*. If a single key is n bits, the equivalent key length is $2n$. DES is an obvious candidate: a 112-bit 3DES sidesteps worries discussed earlier about the fragility of DES.

18.4.4 Signing and Encrypting

When transmitting a document it may be desirable to enforce both authenticity and integrity on one hand, and confidentially on the other hand. This is achieved by signing and encrypting. Now, how should it be done? Sign then encrypt? Encrypt then sign? If we encrypt first, we will be signing something unintelligible. This is not very wise. Besides, it is cryptographically vulnerable (Anderson and Needham, 1995).

The protocol can be derived from the message-digest public-key signature protocol (*see* Figure 17.10). Alice wants to send a signed and confidential message M to Bob. The protocol is explained in Figure 18.4.

		Action	Description
1	A	$h_m = H(M)$ $s_m = S_a(h_m)$	Alice computes the message digest and signs the 128-bit digest with her key
2	A $\rightarrow$ B	$\langle E_b(M, s_m)\rangle$	Alice encrypts both the signature and the message with Bob's public key, and sends it to Bob
3	B	$D_b(M, s_m)$	Bob decrypts first with his private key
4	B	$h_m = H(M)$ $v_m = V_a(s_m)$ $v_m = h_m?$	Bob verifies the signature using the following procedure: he hashes the message, then runs V_a on the signature; if the result is equal, then M is ok, and was signed by Alice

Figure 18.4. Public-key Signature and Encryption

A public key protocol such as RSA can be used for both functions, encryption and signature. It is even tempting to use the same keys. However, there are attacks that can take advantage of these vulnerabilities, so care must be taken on how to design and use the protocol. Good sense advises: not to use the same keys for signing and encrypting; if possible, not to use the same algorithm for signing and encrypting; sign before encrypting, and in any case, never put the user in a position of signing unknown things.

18.4.5 Hybrid Cryptography

Nothing could be more wrong than considering that symmetric and asymmetric cryptography are alternative or competitive encryption approaches. In fact, they are complementary, and there is a current trend to associate symmetric cryptography with payload encryption, and asymmetric cryptography with key and signature encryption, in what is called *hybrid cryptography.* We are going to study two examples of approaches used in most current systems: hybrid cryptographic channels and envelopes. Alice and Bob can execute symmetric encryption/decryption, although they don't currently share any key. They also rely on public cryptography: Alice has private key Kr_a, Bob has Alice's public key Kp_a, and vice-versa for Kr_b and Kp_b.

Let us look at how the first protocol could work. Alice and Bob wish to communicate and will establish a *hybrid cryptographic channel* for the purpose: a secure channel where the session key for the channel is exchanged by public-key cryptography, and communication is done with symmetric cryptography using the session key. The protocol is explained in Figure 18.5. Signing may appear to be an overkill, but in fact it foils attacks by impersonation: anyone

		Action	Description
1	B	$s_m = S_b(K_{ss})$	Bob generates a random key K_{ss} and signs it with his private key
2	B → A	$E_a(s_m)$	Bob further encrypts the signed session key with Alice's public key, and sends it to Alice
3	A	$s_m = D_a(E_a(s_m))$	Alice retrieves the signed key s_m by decrypting with her private key
4	A	$v_m = V_b(s_m)$ v_m $OK?$ $K_{ss} = v_m$	Alice verifies Bob's signature with his public key, retrieving K_{ss}
5	A,B	use K_{ss}	Alice and Bob communicate with symmetric encryption using K_{ss} as the key

Figure 18.5. Hybrid Cryptographic Channel

could encrypt something for Alice. This very simple protocol explains the principle of hybrid cryptographic channels. More sophisticated short-term key-exchange protocols with public-key and hybrid cryptography will be studied in Section 18.6.

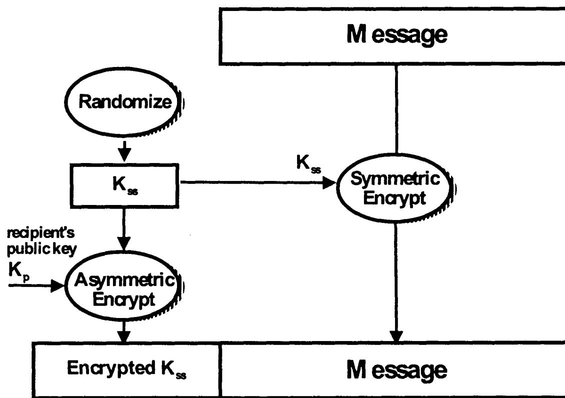

Figure 18.6. Hybrid Cryptographic Envelope

The second approach, that we call hybrid cryptographic envelope, uses the key-encrypting key principle: the message is encrypted with symmetric cryptography, and the encryption key goes along with the envelope, encrypted by public-key cryptography. This second protocol is depicted in Figure 18.6. Alice wishes to send sporadic messages to Bob, and for that purpose, she wraps them in a *hybrid cryptographic envelope.* Alice generates a random symmetric key,

encrypts the message with it, and then encrypts the session key with Bob's public key and concatenates it with the message.

18.5 AUTHENTICATION MODELS

In Section 17.9 we addressed the authentication paradigm. We understood that there are essentially three types of authentication: unilateral- principal A authenticates itself to principal B; mutual- principals A and B mutually authenticate themselves; mediated- principal A is authenticated to B by principal T, whom they both trust. These types are implemented through essentially three *mechanisms for authentication* of a principal A with a principal B:

Password	authentication is based on A submitting to B a unique pair ⟨ `username,password` ⟩ that B recognizes
Shared-secret	authentication is based on A proving to B that A knows a secret K that they and no one else share, without showing the secret
Signature	authentication is based on A proving to B to have signed something using its digital signature S, that no one else can produce

In what follows, we discuss several fully-fledged authentication models, which are essentially instantiations of the several authentication types materialized with combinations of the mechanisms just described. We analyzed them under the light of two groups of threats: reading of the authentication service state; eavesdropping on remote interactions.

18.5.1 Password-based Authentication

The simplest way to authenticate an authorized user in a system is password-based authentication. The problem is the following: a principal A submits a pair ⟨`username,password`⟩ to an authentication service S, and the pair should uniquely identify it. However, this method is not very secure. The intruder may read the state of S, so the password file should be protected: by obscuring its contents, for example by replacing the password with its hash or with a cryptographic checksum depending on the password, so that it cannot be reversed (*one-way encryption*); by making it difficult for the intruder to read the password file (*shadowing*). Even so, the intruder can decipher the password of a user by guessing, what is also called *cracking* in hacker lingo. He can try them directly, exhaustively or by selecting probable passwords, called *on-line password guessing*, which is infeasible if passwords are long enough. Else, he can do *off-line password guessing*, if he gets access to the hashed password file. He runs each candidate password through the hash algorithm and checks if the result is the same as in the password file. The attack can be made quicker by using a dictionary to base the search on (usual words in the language, words related with the user, her working field and her organization, etc.). This is called a **dictionary,attack**, and it is surprising how many passwords can be

guessed in a normal organization. This is why the password creation, change and deletion procedures discussed in Section 18.3.2 are important, in order to harden the passwords against guessing.

The second problem with password authentication is that it is very insecure when login is remote. The password is submitted in cleartext, and so an eavesdropper may read it and reuse later. Compared to the trouble of cracking a password file, this is a lot easier, and only requires that the hacker controls one machine in a network, so beware of what has become a very common threat. Since eavesdropping or sniffing on a network can go unnoticed, there is no remedy for this but changing the approach: encrypting whatever is exchanged or exchanging only public or trivial things. Later in this section we will learn how to do more resilient authentication exchanges.

Password File Protection An interesting technique for password file protection is employed by UNIX and other systems. The generic mechanism works as follows:

1. *the 8-character (7-bit ASCII)* password *is converted into a 56-bit key to be used in a DES-derived encryption function called* crypt*; crypt is also parameterized with a 12-bit* salt *value, a randomized quantity that makes two entries of the same password always look different;*
2. `crypt` *is a cryptographic checksum algorithm: it starts using an all-zero block as input, uses the result as the input of the next round, and performs 25 rounds, using the key and the salt;*
3. *the result is translated to an 11-character printable ASCII* string*, and the password entry of a user in the password file becomes the triplet* ⟨userId, salt, string⟩*;*
4. *when a user logs in, she supplies* ⟨userId, password⟩*; the operating system indexes the password file with* userId *and grabs* ⟨salt, string⟩*;*
5. *the cryptographic checksum is performed on* ⟨salt, password⟩ *and the result compared with* string*. If they match, the user is authenticated.*

One-Time Passwords Systems offering one-time passwords (OTP) change the password each time it is used. In consequence, this approach is of the "exchanging trivial things" type: the plaintext passwords the eavesdropper sees passing are useless by the time he collects them. Existing one-time password systems fall in the class of the *challenge-response* systems: the authenticator sends a challenge in the form of *"give me password number x"*; the user should reply with the appropriate password. Passwords may be computed interactively by the user upon receiving the challenge, which requires that her machine can run the computation or that she is assisted by a device such as a pocket-calculator password generator. Else, a ⟨*challenge,password*⟩ list, with all the passwords corresponding to every challenge, is given in advance to the user.

The essential property of the mathematical function used to compute the challenge is that the password should be deduced neither from the challenge nor from past passwords. It would seem that one-way hashes would be extremely appropriate for this. Lamport invented a function for a one-time pass-

word scheme (Lamport, 1981) based on generating a sequence of n passwords by hashing a password n times. Of course, the system must start from password p_n and work backwards, to avoid that password p_{n+1} is deduced from password p_n by simply hashing one more time. In conclusion, note that one-time passwords are resilient against both reading penetration and eavesdropping. One inconvenience of the scheme is that passwords eventually get exhausted, requiring the process to be re-initiated. This however is not a problem for short-term use (e.g., during trips). The S/Key system, discussed in Section 19.1, is an example of OTP system.

18.5.2 Shared-Secret Authentication

The approach of "encrypting whatever is exchanged" is appropriate for long-term use. Shared-secret authentication is one such example, where principal A (Alice) and authenticator S (Stuart) share a secret known only to both of them, let us call it K_{as}. They perform cryptographic interactions so that S is persuaded that its peer *knows* K_{as}. At this point, S *believes* that it is A on the other end, since only A could know the secret. We sketch the two basic mechanisms for doing it. Actually, mechanism (a) can use either true encryption or a cryptographic checksum (one-way encryption):

- *a) Alice sends* userId*; Stuart sends challenge "if you are Alice, encrypt X for me"; Alice sends response $E_{as}(X)$, which S checks;*

- *b) Alice sends* userId*; Stuart sends challenge "if you are Alice, decrypt $E_{as}(X)$ for me"; Alice sends response X*

18.5.3 Signature-based Authentication

Yet another "encrypting what is exchanged" approach, signature-based authentication uses public-key cryptography. Principal A (Alice) has private key Kr_a, and authenticator S (Stuart) has A's public key Kp_a. They perform cryptographic interactions so that S is persuaded that its peer knows the pair of Kp_a. At this point, S believes that it is A on the other end, since only A could know the private pair of Kp_a. There are essentially two mechanisms for doing it:

- *a) Alice sends* userId*; Stuart sends challenge "if you are Alice, sign X for me"; Alice sends response $S_a(X)$, which S verifies with Alice's public key;*

- *b) Alice sends* userId*; Stuart sends challenge "if you are Alice, decrypt $E_a(X)$ for me"; Alice decrypts with her private key and sends response X.*

18.5.4 Mutual Authentication

We have been discussing how to authenticate principal A to principal S. However, what if someone impersonates S and cheats on A? The solution to that problem is mutual authentication. None of the mechanisms discussed so far allows mutual authentication, but that can be achieved by enhancing the same mechanisms.

Shared-secret Mutual Authentication Shared-secret mutual authentication can be derived from the unilateral mechanism, by having S authenticate to A in the same way. The principle is explained in Figure 18.7.

		Action	Description
1	A → S	$\langle A \rangle$	Alice sends *userId*
2	S → A	$\langle X_s \rangle$	Stuart sends his challenge X_s for Alice to encrypt
3	A → S	$\langle E_{as}(X_s), X_a \rangle$	Alice sends the encrypted challenge in response together with her challenge X_a
4	S → A	$\langle E_{as}(X_a) \rangle$	Stuart in turn sends the encrypted response to Alice's challenge
5	A,S		Both believe they're talking to each other

Figure 18.7. Shared-secret Mutual Authentication

This protocol is only vulnerable to Mallory impersonating Stuart to collect material to do a key guessing attack, which is difficult per se. Moreover, the feasibility of this attack depends on the strength of the cipher used.

Mutual Authentication by Signature Signature-based mutual authentication can also be derived from the unilateral protocol, by having S authenticate to A in the same way. The modified protocol is explained in Figure 18.8.

		Action	Description
1	A → S	$\langle A, X_a \rangle$	Alice sends *userId* and challenge for Stuart to sign
2	S → A	$\langle X_s, S_s(X_a) \rangle$	Stuart sends his challenge X_b for Alice to sign, and signs Alice's challenge
3	A → S	$\langle S_a(X_s) \rangle$	Alice sends the signed challenge in response
4	A,S		Both believe they're talking to each other

Figure 18.8. Mutual Authentication by Signature

Alice believes it is Stuart on the other end, because she could verify his signature on her challenge. Only Stuart could have signed the challenge. Stuart believes it is Alice on the other end, exactly for the same reasoning applied to Alice's signature.

18.5.5 Mediated Authentication

Alice shares a secret K_{at} with arbiter Trent (so does Bob with K_{bt}), and can authenticate herself to Trent (so can Bob) in the sense of our discussion on shared-secret authentication. The problem now is to extend this to authentication of Alice to Bob and vice-versa, *mediated* by Trent. The principle of mediated authentication protocols is the following:

1. *Alice sends her* Id *and Bob's to Trent, requesting an authenticated session;*
2. *Trent arranges for a shared secret key K_{ab} to be distributed to both: to Alice, encrypted with her key K_{at}, and to Bob, encrypted with his key K_{bt};*
3. *Trent also ensures that it is distributed in a way that at the end of this process, they are mutually authenticated in the sense of the shared-secret principle (see Figure 18.7), where K_{ab} is the shared secret.*

		Action	Description
1	A → T	$\langle A, B, X_a \rangle$	Alice sends her and Bob's *Id* and nonce X_a to Trent
2	T	K_{ab}	Trent generates a key K_{ab} for Alice to share with Bob
3	T → A	$\langle E_a(X_a, B, K_{ab},$ *ticket*= $E_b(K_{ab}, A)) \rangle$	Trent sends Alice, encrypted with Alice's key: her nonce back, Bob's *Id*; the shared key. Also under the encryption goes a credential or ticket to Bob, encrypted with Bob's key, containing Alice's *Id* and the shared key
4	A → B	$\langle E_b(K_{ab}, A) \rangle$	Alice sends the ticket to Bob
5	B → A	$\langle E_{ab}(X_b) \rangle$	Bob retrieves the shared key, encrypts his nonce with it, and sends it to Alice
6	A → B	$\langle E_{ab}(X_b - 1) \rangle$	Alice retrieves and decrements the nonce, and sends it back encrypted with the shared key
7	A,B		Both believe they're talking to each other

Figure 18.9. Original Needham-Schroeder Authentication Protocol

Nonce-based Authentication Let us have a look at the basic Needham-Schroeder protocol (Needham and Schroeder, 1978), depicted in Figure 18.9. Recall that **nonce** is a quantity that is used only once. Sequence or random numbers or timestamps, are nonces. Nonce X_a in (1) tries to reassure Alice that she's addressing Trent, since he uses it in his reply (3) back to Alice. B in (3) reassures Alice that the shared key is meant for her connection to Bob, and no one else. The ticket cannot be seen by anyone but Bob, and reassures him that the shared key is meant for his connection to Alice. The nonce X_b sent in (5) and sent back decremented by Alice (6) proves to Bob that Alice knows K_{ab}.

Authentication based on Synchronized Clocks The Kerberos authentication protocol (Neuman and Ts'o, 1994) is depicted in Figure 18.10. It is very simple, and relies on roughly the same reasoning as the former protocol to achieve authentication. However, it further relies on principals having synchronized clocks to test each other's timestamps and detect replay attacks, within a window whose size depends on the precision of the clocks (*see Time and Clocks* in Chapter 2). Alice tests Trent's timestamp in (3) to see that it is current. Bob tests Alice's timestamp T_a in (4). Alice tests her incremented timestamp $T_a + 1$, which could only have come from the current interaction with Bob, since Alice sent message (4) at time T_a, protected with the shared key. The Kerberos Security Service (*see* Section 19.4) uses this protocol.

		Action	**Description**
1	A → T	$\langle A, B\rangle$	Alice sends her and Bob's *Id* to Trent
2	T	K_{ab}	Trent generates a shared key K_{ab} for Alice to share with Bob, and assembles a *keyId*, containing the key, its expiry time T_x, and the current timestamp T_t
3	T → A	$\langle E_a(keyId,B)$, $ticket=$ $E_b(keyId,A)\rangle$	Trent sends Alice: (i) the *keyId* and Bob's *Id*, encrypted with Alice's key; (ii) a ticket to Bob, encrypted with Bob's key, containing the *keyId* and Alice's *Id*
4	A → B	$\langle E_{ab}(A, T_a)$, $E_b(keyId,A)\rangle$	Alice sends Bob the ticket together with her *Id* and current timestamp, encrypted with K_{ab}
5	B → A	$\langle E_{ab}(T_a + 1)\rangle$	Bob retrieves the shared key and the timestamp, increments the latter, encrypts it with the shared key, and sends it to Alice
6	A,B		Both believe they're talking to each other

Figure 18.10. Kerberos Authentication Protocol

18.5.6 Distributed Authentication

The several levels of indirection in a distributed system render the problem of authentication a complex one. That is, the authentication models we have just studied must incorporate delegation, and that is hard to do in face of: autonomy, scale, heterogeneity.

Distributed authentication is about authenticating a channel along which there is a user who sits at a host that sends messages on behalf of her through a network that carries messages on behalf of the host, and so forth. It is difficult to ensure that the basic principles of delegation in such an environment are followed correctly (*see Delegation* in Section 17.9) such as *specific delegation*, *authentication forwarding*, or *end-to-end authentication*.

Lampson et al. (Lampson, 1993) introduced a logic of authentication and delegation of *channels* in distributed systems:

- A **says** s – e.g., it is true that A produces statement (requests) $\langle read\ m \rangle$
- A **speaks for** B – for example, a terminal represents whoever sits at it, i.e., $\langle K_a$ speaks for $A\rangle$
- **handoff**: A **says** B **speaks for** A – what defines the delegation, i.e., A delegates on secure channel B
- **credential**: proof that $\langle A$ speaks for $B\rangle$ – what proves the delegation, i.e., that user A delegates on host B the access to critical data owned by A; to that purpose, A gives a credential to B, maybe a password typed at the terminal, or a card swiped through the terminal card reader

A more complex delegation would be: if C **speaks for** A, and C **says** (K_{ab} speaks for A), then K_{ab} **speaks for** A. In other words, if a secure (and authentic) channel from A sends a message with a session key K_{ab} for a third principal B to speak with A, then any message to B encrypted with K_{ab} speaks for A. That is, A delegated in C, and C announced that A delegated in K_{ab}.

18.6 KEY DISTRIBUTION APPROACHES

In Section 17.9, we have explained that key distribution has two facets: *long-term key distribution*, that puts in place all the keys necessary for system bootstrap and long-term use; and *short-term key exchange*, that exchanges the keys necessary for temporary use, such as message or email deliveries, or interactive sessions. Remember that we called these keys *session keys*.

The long-term key distribution problem is mostly concerned with distribution of public key certificates. Short-term key exchange mainly addresses shared secret keys for symmetric cryptography. Ad hoc distribution of public keys is not practical for large systems, composed of unknown users, so we will study a dedicated, "official" service, specializing in supplying public key certificates, called **Certification Authority (CA)**. Likewise, managing pair-wise keys in a large system is an insurmountable challenge. In consequence, we will study a specialized service, called a **Key Distribution Center (KDC)**, which significantly eases the task of further exchanging keys in the system.

We start by addressing the general long-term key distribution problem, and then we discuss short-term key exchange, where we will consider, when necessary, that some form of long-term shared secret or asymmetric key pair is already in place.

18.6.1 Certification Authorities (CA)

In open distributed systems, for example those working over the Internet, there may be a large number of participants, who do not trust and may even not know each other. Since public key cryptography is crucial for digital signatures and these are a powerful form of authentication, an obvious solution to this problem is to provide **certificates** containing a participant's name and its signature. It is only natural for this name-to-key translation to follow a principle similar

to the name or directory service in distributed systems (*see Naming and Addressing* in Chapter 2). The server is a trusted third party called Certification Authority (CA)), and besides a well-known address as the name service, it has a *well-known public key*. The CA signs every certificate it issues, and its signature can be verified by any participant. Other functions of the CA are the revocation of certificates, and the distribution of certificate revocation lists (CRLs).

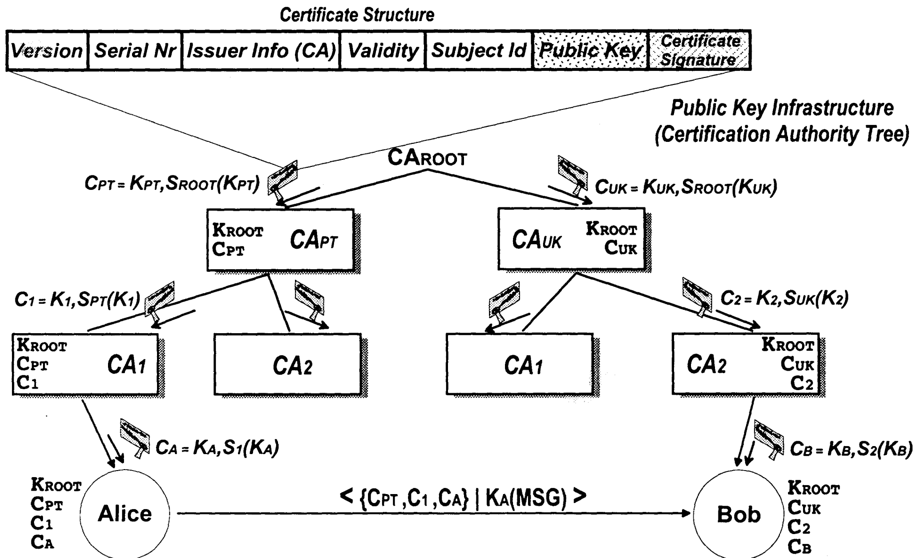

Figure 18.11. Public Key Infrastructure

Certificates are inherently secure to travel over the network, since they can neither be modified nor forged. The CA can work off-line, or have a very restricted interface (e.g., email, Web) to ease a tamperproof design. However, a CA may also be used to provide on-line proofs of identity. In a large-scale system such as the Internet, the CA's can form a hierarchy. Infrastructures for CA hierarchies are being standardized, the collective initiative is designated as the Public Key Infrastructure (PKI) and one such standard is the X.509 from the ITU (X.509, 1997). They work such that the CAs of the next level down have their public key certified (signed) by the root CA, and so forth down the hierarchy, such as represented in Figure 18.11, where part of an imaginary European X.509-like PKI is shown, with each CA keeping the certificates of all CAs in the chain up to the root.

The PKI is the primary support for wide use of public key cryptography: as defined by the X.509 standard, a certificate is a digital document that binds a public key to the identity or another attribute of its principal. As shown in Figure 18.11, a key certificate is normally composed of the identity of the principal, its public key, the timestamp of creation and validity, everything signed by the

issuing authority, and verifiable with the public key of the latter. Imagine that Alice in Portugal wishes to perform some operation with Bob in the UK, such that her public key is required— e.g., to have Bob encrypt things for her, or to check her signature. Alice has previously requested her certificate to CA_1 on the left, whose X.500 Distinguished Name is in fact $\{CA_{ROOT}, CA_{PT}, CA_1\}$, and in consequence, she keeps a chain of certificates $\{K_{ROOT}, C_{PT}, C_1, C_A\}$, which she sends Bob, alone or as part of one of the several public-key protocols available[4]. PKIs have been receiving great attention given their importance for electronic commerce. The way they are being deployed today is however not exempt from risks, as pointed out by Ellison & Schneier (Ellison and Schneier, 2000).

18.6.2 Key Distribution Centers (KDC)

A key-exchange mechanism for symmetric cryptography requires the distribution of one key per pair of participants. This is not practical, since it requires a very large amount of keys ($n\,(n-1)/2$) and establishing trust among an unnecessarily large number of pairs of people, namely in a large-scale system.

If key distribution is centralized in a special service, then only n keys, for n principals, will be required. This service is performed by a trusted third party called a Key Distribution Center (KDC). The initial bootstrap process of exchanging the first key may be part of the off-line process of registering the new user. After that, everything goes through the KDC.

The KDC presents several disadvantages. The first is that it is a serious single point of failure, for two reasons: once compromised, it can impersonate anyone to anyone; and if it crashes, everything stops. Secondly, it is a bottleneck, because unlike CAs, it will be used on-line and in the critical path of the execution of protocols. Like CAs, KDCs can be interconnected. The Kerberos Security Service (*see* Section 19.4) exemplifies a KDC.

18.6.3 Short-term Key Exchange

Key-exchange with KDC Although key exchange may be performed with pair-wise symmetric cryptography, such a mechanism is not practical, since it requires a very large amount of keys. A more practical solution is to consider that in real-life systems, Alice and Bob do not trust each other, but each of them trusts a third party, which can be a KDC as we have discussed earlier. This is a good characterization of an open distributed system. Alice and Bob use a mediated authentication mechanism such as the ones described in Section 18.5.5, to have their key distributed by the KDC.

---

[4]The Distinguished Name is a unique name in the hierarchy that identifies a principal. It is composed by concatenating the names of the hierarchy above the principal, such as done with DNS names. We wanted to emphasize this aspect by showing that the lower CAs in PT and UK can have equal names.

Key-exchange with Diffie-Hellman Assume that Alice and Bob wish to exchange a session key K_{ss}, and rely on the Diffie-Hellman algorithm to create it without having to exchange secret values. According to the basic algorithm (*see* Figure 17.5), Alice and Bob exchange their public numbers y_a and y_b, and compute the same K_ss. Now suppose that just before they exchange numbers, Mallory gets in the middle and instead gives Alice y_{b_m}, and Bob y_{a_m}. This situation is depicted in Figure 18.12, and the effect is that instead of Alice and Bob sharing some key K_{ss}, Alice and Mallory end up sharing a key K_w, different from that shared by Mallory and Bob, K_z. However, Bob and Alice do not know that. If Mallory is fast enough that he can decrypt a message M coming from Alice and re-encrypt it to Bob on-the-fly, and vice-versa, the attack will go unnoticed. Needless to say that Mallory can leisurely study Alice's and Bob's interactions and prepare something worthwhile (suppose Bob was Alice's home banker).

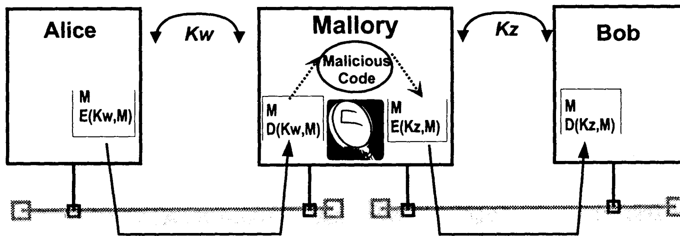

Figure 18.12. Man-in-the-Middle Attack

Mallory's attack is called a *man-in-the-middle attack*, or *bucket-brigade attack*. It belongs to the class of *spoofing attacks*, by impersonation in this case. The attack succeeds because D-H is a key exchange protocol without authentication. Next, we will see how to protect a D-H exchange with signatures or encryption.

Key-exchange with Public-Key Cryptography Alice and Bob wish to exchange a session key K_{ss}, but now they rely on public cryptography: Alice has private key Kr_a, Bob has Alice's public key Kp_a, and vice-versa for Kr_b and Kp_b. They can mutually authenticate themselves, as in the protocol of Figure 18.8.

One possibility could be to use one of the simple key-exchange mechanisms proposed for hybrid cryptography in Section 18.4.5: Bob generates a random key K_{ss} and sends it to Alice encrypted with her public key, $E_a(K_{ss})$; or Bob further signs the encrypted session key with his private key, $S_b(E_a(K_{ss}))$. The first approach would fail under a spoofing attack by impersonation. The second is robust, but a moderate problem is that it still is sensitive to read penetration attacks. If Mallory reads Alice's or Bob's state (i.e. $Kr_{a|b}$), he can decode past conversations.

Let us look at a more resilient protocol, depicted in Figure 18.13. This protocol can be seen as an enhanced variant of the encryption version (b) of the signature-based authentication mechanisms discussed in Section 18.5, adapted to mutual authentication and with the challenges encrypted since they are used in key generation. At the end, they can use both (now secret) challenges to generate a secret session key. More sophisticated operations than XOR can be envisaged. In fact, this is a modified version of the original public-key Needham-Schroeder authentication protocol (Needham and Schroeder, 1978). That protocol was recently shown to fall under a combined impersonation attack through Id spoofing and man-in-the-middle (Lowe, 1995). Step 2 has the fix w.r.t. the original: B's *Id* goes in the message.

		Action	**Description**
1	A → B	$\langle E_b(A, X_a)\rangle$	Alice sends *Id* and challenge to Bob, encrypted with his public key
2	B → A	$\langle E_a(B, X_a, X_b)\rangle$	Bob decrypts X_a, sends *Id* and challenge X_b together with X_a to Alice, encrypted with her public key
3	A → B	$\langle E_b(X_b)\rangle$	Alice sends Bob's challenge back, encrypted
4	A,B		Both believe they're talking to each other
5	A,B	$K_{ss} = X_a\ XOR\ X_b$	Both have two secret numbers with which to generate a shared secret key

Figure 18.13. Key-exchange with Public-Key Cryptography

Another protocol, this one based on enhancing the 'key-exchange with Diffie-Hellman' mechanism just discussed, is explained in Figure 18.14. The protocol takes the best of two worlds: it relies on the robustness of the Diffie-Hellman mechanism to derive a secret key without exposing *any* communication with it; it relies on signatures to authenticate the principals involved. Each principal starts the D-H mechanism (*see* Figure 17.5), and before exchanging their public numbers (Y_a, Y_b), they sign and identify the relevant messages. They can then generate K_{ss} according to the D-H algorithm, with the guarantee that they are really talking to one another. Compared to the simple signed scheme that we gave in the beginning of this section, where the key is generated by one principal who signs and encrypts it, and then sends it to the other principal, this is more robust, since it is no longer sensitive to read penetration attacks. This is an excellent key exchange scheme.

Key-exchange with Hybrid Cryptography Now Alice wishes to exchange a session key K_{as} with Stuart, using another protocol class, combining symmetric and asymmetric cryptography. The EKE (Encrypted Key Exchange) protocol of Bellovin and Merritt (Bellovin and Merritt, 1992) is an example.

		Action	Description
1	A → B	$\langle S_a(A, Y_a)\rangle$	Alice sends *Id* and D-H number to Bob, signed with her private key
2	B → A	$\langle S_b(B, Y_b)\rangle$	Bob sends *Id* and D-H number to Alice, signed with his private key
3	A,B		Both believe they're talking to each other
4	A,B	$K_{ss} = DH(Y_a, Y_b)$	Both have the two D-H numbers with which to generate a shared secret key

Figure 18.14. Key-exchange with Signed Diffie-Hellman

It has several variants and is specially suited for when authentication and key exchange depend on a user password and are thus vulnerable to guessing attacks. The basic protocol is explained in Figure 18.15. Alice has password P, and there is password-derived secret key K_p, shared by Alice and Stuart; she also has an asymmetric key pair $\langle Ku_a, Kr_a\rangle$.

		Action	Description
1	A → S	$\langle E_p(A, Ku_a)\rangle$	Alice sends *Id* and Ku_a to Stuart, encrypted with K_p
2	S → A	$\langle E_p(E_a(K_{as}))\rangle$	Stuart generates session key K_{as}, encrypts it with Alice's public key, further encrypts with K_p, and sends the result to Alice
3	A,S	K_{as}	Alice retrieves K_{as}
4	A,S		They may proceed with mutual authentication (e.g., steps 1-3 of Fig. 18.13)

Figure 18.15. Encrypted Key-exchange

The resilience against guessing attacks comes from the fact that the information collected by the eavesdropper is practically useless: to test the guesses, he would have to break a double (symmetric-asymmetric) encryption. The protocol used in DASS (*see* Section 19.4) is another example of key-exchange with hybrid cryptography.

18.7 PROTECTION MODELS

Ensuring that an authorized principal, and only it, can access data or a service: that is what *protection* is generically about. Subjects may be explicitly identified (authentication before authorization) or implicitly assumed (e.g., address-

based authorization). There may be one or more levels of protection. The system may provide *single-level security*, where a *security perimeter* is defined, and all the entities inside the perimeter are considered benign. Alternatively, when protection against *insider attacks* is desired, the need-to-know rule is enforced and several classes of access control are defined, in what is called *multi-level security.* Protection may impact all resources or just a few of them, that is, all system operations may or not go through a *reference monitor*. Access control rules may be implemented in an ad hoc manner or according to some formal rule set, that is, the system may follow a discretionary or a mandatory access control policy, in which case the policy is normally dictated by a *formal security policy model*. Underlying architectural measures further strengthen the protection mechanisms. These are the several facets of the protection problem, that we study in the next few sections.

18.7.1 Authorization

Authorization of access for a principal may or not assume previous authentication, since in some forms of interaction, the user is authorized access simply because she is in a particular situation (such as sitting in front of a given terminal, knowing the secret phone number of a dial-up connection, or having an given IP source address or port). In others, authentication has been performed previously, and the user gets an object (e.g., a cryptographic credential) that proves authorization without revealing identity (*see* Section 18.5).

In order to be able to be automated, and also verifiable, authorization must be dictated by a *security policy.* A security policy consists of specifying: the security classification of users (subjects), that is, their ranking in terms of the sensitivity of information they can access; the access classification of data, system services, resources in general (objects), that is, the sensitivity of these resources, in terms of the organization activity; and the access control policy.

The *access control policy* is the specification of the way subjects can access objects, according to each other's classification, the sensitivity of the objects in terms of confidentiality or integrity, and the use of the least privilege and need-to-know principles. Common *security classifications* are, in increasing order of sensitiveness: *unclassified, restricted, confidential, secret, top secret*. These originated in the military and although usable in other settings, commercial applications may follow a more suitable hierarchy: *public, proprietary, internal*. In the end, we may see the security policy translated into the triples $\langle s, o, r \rangle$ that form the access control mechanism (*see* Section 17.10), by following the rules dictated by the classifications and the access control policy.

18.7.2 Reference Monitor Model

We have already discussed access control. However, for it to be effective under a system perspective, a few additional questions arise: Does access control apply to all subjects and objects or only to some? Are the rules deterministic or can