to its specifications if

$$\forall t : (1-\rho) \leq \frac{F(t)}{F} \leq (1+\rho)$$

By using hardware interrupts we are directly coupling a software clock to the hardware clock, and thus also its clock drift rate. In particular, we have that

$$C_p(t) = \frac{1}{F}\int_0^t F(t)dt, \text{ and thus: } \frac{dC_p(t)}{dt} = \frac{F(t)}{F}$$

which brings us to our ultimate goal, namely keeping the software clock drift rate also bounded to ρ:

$$\forall t : 1-\rho \leq \frac{dC_p(t)}{dt} \leq 1+\rho$$

Slow, perfect, and fast clocks are shown in Figure 6.4.

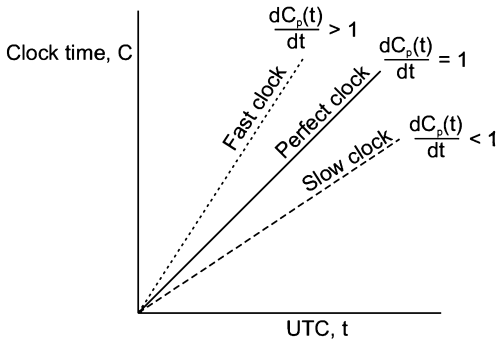

Figure 6.4: The relation between clock time and UTC when clocks tick at different rates.

If two clocks are drifting from UTC in the opposite direction, at a time Δt after they were synchronized, they may be as much as $2\rho \cdot \Delta t$ apart. If the system designers want to guarantee a precision π, that is, that no two clocks ever differ by more than π seconds, clocks must be resynchronized (in software) at least every $\pi/(2\rho)$ seconds. The various algorithms differ in precisely how this resynchronization is done.

Network Time Protocol

A common approach in many protocols and originally proposed by Cristian [1989], is to let clients contact a time server. The latter can accurately provide the current time, for example, because it is equipped with a UTC receiver or an accurate clock. The problem, of course, is that when contacting the server, message delays will have outdated the reported time. The trick is to

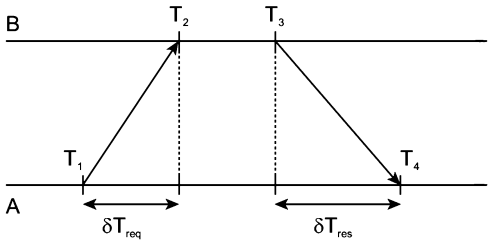

Figure 6.5: Getting the current time from a time server.

find a good estimation for these delays. Consider the situation sketched in Figure 6.5.

In this case, A will send a request to B, timestamped with value T_1. B, in turn, will record the time of receipt T_2 (taken from its own local clock), and returns a response timestamped with value T_3, and piggybacking the previously recorded value T_2. Finally, A records the time of the response's arrival, T_4. Let us assume that the propagation delays from A to B is roughly the same as B to A, meaning that $\delta T_{req} = T_2 - T_1 \approx T_4 - T_3 = \delta T_{res}$. In that case, A can estimate its offset relative to B as

$$\theta = T_3 + \frac{(T_2 - T_1) + (T_4 - T_3)}{2} - T_4 = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

Of course, time is not allowed to run backward. If A's clock is fast, $\theta < 0$, meaning that A should, in principle, set its clock backward. This is not allowed as it could cause serious problems such as an object file compiled just after the clock change having a time earlier than the source which was modified just before the clock change.

Such a change must be introduced gradually. One way is as follows. Suppose that the timer is set to generate 100 interrupts per second. Normally, each interrupt would add 10 msec to the time. When slowing down, the interrupt routine adds only 9 msec each time until the correction has been made. Similarly, the clock can be advanced gradually by adding 11 msec at each interrupt instead of jumping it forward all at once.

In the case of the **network time protocol** (**NTP**), this protocol is set up pairwise between servers. In other words, B will also probe A for its current time. The offset θ is computed as given above, along with the estimation δ for the delay:

$$\delta = \frac{(T_4 - T_1) - (T_3 - T_2)}{2}$$

Eight pairs of (θ, δ) values are buffered, finally taking the minimal value found for δ as the best estimation for the delay between the two servers, and subsequently the associated value θ as the most reliable estimation of the offset.

Applying NTP symmetrically should, in principle, also let B adjust its clock to that of A. However, if B's clock is known to be more accurate, then such an adjustment would be foolish. To solve this problem, NTP divides servers into strata. A server with a **reference clock** such as a UTC receiver or an atomic clock, is known to be a **stratum-1 server** (the clock itself is said to operate at stratum 0). When A contacts B, it will adjust only its time if its own stratum level is higher than that of B. Moreover, after the synchronization, A's stratum level will become one higher than that of B. In other words, if B is a stratum-k server, then A will become a stratum-$(k+1)$ server if its original stratum level was already larger than k. Due to the symmetry of NTP, if A's stratum level was *lower* than that of B, B will adjust itself to A.

There are many important features about NTP, of which many relate to identifying and masking errors, but also security attacks. NTP was originally described in [Mills, 1992] and is known to achieve (worldwide) accuracy in the range of 1–50 msec. A detailed description of NTP can be found in [Mills, 2011].

The Berkeley algorithm

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

In Figure 6.6(a) at 3:00, the time daemon tells the other machines its time and asks for theirs. In Figure 6.6(b) they respond with how far ahead or behind the time daemon they are. Armed with these numbers, the time daemon computes the average and tells each machine how to adjust its clock [see Figure 6.6(c)].

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