CptS 464/564
Lecture 24

6th December 2004

Real-time distributed systems, continued

Time-value duality

Suppose that $E()$ is the time-varying function representing the state of some physical entity. A specification says "At time T_A produce value $V_A = E(T_A)$", and that the system actually produces value v at time t. How can we talk about how well the system meets its specification?

If $t = T_A$ then there is a *value error* $|v - V_A|$. If, on the other hand, $v = E(T_A)$ but $t > T_A$ what can we say? There is clearly a timing error: the value has been produced late. But is it *only* a timing error? We would often say "no" because we want to act on the state of the physical entity as of the time it was measured. So let's look at what we actually measure and when. It will be useful to consider all observations as being timestamped, so an observation looks like $\langle (r(E))(t), T \rangle$ indicating that we have recorded the value of the representative of E at some time t and time-stamped it with T.

The *observation error in the value domain* is defined to be $|(E(T) - E(t)) + (E(t) - (r(E))(t))|$. Intuitively this makes sense: $\langle (r(E))(t), T \rangle$ is a statement that at time T the value of $E(T)$was $(r(E))(t)$. The error in this statement is simply the amount by which $(r(E))(t)$ differs from $E(T)$, that is $|E(T) - (r(E))(t)|$. Notice that we cannot measure the error directly – we will have to argue as to its potential magnitude from the design and implementation of the system.

The *observation error in the time domain* is defined to be $|T - t|$. That is, by how much is the timestamp on the observation wrong.

In the specification, design and implementation of a real-time system, you have to be aware of both of these aspects: I find it very helpful to use the notion of timestamped observations from the outset in the design. Doing so focuses one's attention on *both* ways that an observation can be in error.

The next problem to consider is *when does an observation become available for use?* An observation $\langle v, T \rangle$becomes less and less valuable for real-time control as time passes. At time t_a, where $t_a > T$, the *temporal inconsistency* of an observation $\langle v, T \rangle$ is measured by the error term $|E(t_a) - E(T)|$. Notice that temporal consistency does not depend on the value in the observation but only on the behavior of $E()$. If $E()$is well-behaved we may be able to establish a time interval over which observations are known to have bounded temporal inconsistency.

Real-time communications

Real-time communications requires having *known* and *bounded* message delivery delay. The delay should be considered end-to-end, from the sending process to the receiving process.

The bounds must remain known and bounded within all behaviors of the environment contemplated by the coverage assumptions: other traffic in the system, variable loads, faults, etc.

Achieving bounded delay involves bounding the delays on various steps along the way: delay from the sending process to the network interface of the source system; from the network interface onto the communication medium; from the receiving network interface to the receiving process. The first and last are the domain of real-time operating systems. The middle one is intimately tied to the design of the communication network itself. For example, CSMA/CD ethernet, which uses a randomized contention algorithm for medium access, is not well-suited to hard real-time applications.

Simultaneously handling multiple kinds of traffic with different urgency requires use of some sort of prioritization mechanism to handle more urgent traffic sooner.

Finally, the problem of independent failures again haunts us: what if the network loses packets while the computers attached to it continue to function. Networks, exposed as they are to the physical world and to use by potentially many applications, are less reliable than the computers. Fault-tolerance techniques are required to increase the overall system reliability beyond the basic reliability of the network – and this fault-tolerance must be achieved within the time bounds set for the system.

With all this in mind we come to our list of steps for achieving real-time communications:

1. compute the network load budget in the worst-case scenarios. Determine how traffic is to be prioritized
2. make the network as reliable as possible and reduce partitioning as much as possible (notice that this is insufficient by itself: the network is still unreliable and may still partition; doing this step may make our lives easier in the later steps, however.
3. achieve bounded time to send a frame (OS, network architecture)
4. introduce mechanisms to tolerate omission faults up to a maximum omission frequency. This may involve transmitting multiple messages over different paths (so-called *spatial redundancy)*, or sending multiple messages separated in time over the same path (*temporal redundanc*y) or both. The time bounds of #3 are very important here because they limit how much redundancy can be achieved, thus how many faults can be tolerated, while still meeting the time bounds for delivery of messages.
5. control system load – make sure that the budget set in #1 continues to hold in real-time. As we have previously mentioned, this step involves some sort of admission control. It is thus important to consider the business, social and political context for these decisions because people may view the "System" as being much more than the "system" being implemented. (E.g. 911 – load shedding outside the technological system would be seen as a failure of the 911 system).

Real-time scheduling

Recall that RT is not the same as "fast". RT is about meeting deadlines: fast helps, but it isn't sufficient. We also have to be clever.

Consider the tortoise and the hare: the requirement is to carry two boxes from the starting point, one must be taken east by 270 feet within 290 seconds, the other must be taken north 15 feet within 28 seconds.

The tortoise moves 1 foot/s and takes 1 second to turn 90 degrees; the hare moves 10 feet/s and turns in 0.01 seconds. Both start facing east.

- The tortoise's strategy: turn north and deliver that box: 1 second to turn, 15 seconds to deliver; box done at t=16 (on time); turn east for 1 second then travel east 270 seconds finishing at t=287, on time. (I guess it wasn't a tortoise but rather a box turtle)
- The hare's strategy: travel east for 27 seconds, delivering the first box at t=27; turn north, travel north 1.5 seconds delivering the second box at t=28.51 seconds – oops! too late.

So, how can we be clever?

First, we must recognize that some tasks can't be scheduled in real-time. For example, the aperiodic tasks that we mentioned previously.

On the other hand, periodic tasks are relatively easy: we can analyze the periods of the various tasks then statically create a schedule that meets all the deadlines. The big challenge comes with sporadic tasks: their behavior is not constant but there are known bounds on their behavior.

Scheduling terms:

static scheduling schedule is created ahead of time, off-line (works for periodic tasks, not sporadic)

dynamic scheduling schedule is created at run-time, as the tasks are executing (required for sporadic, also OK for periodic)

fixed-priority scheduling tasks are assigned a priority which they keep forever

dynamic-priority scheduling task priorities change with time; for example, as deadlines approach

preemptive scheduling one task's execution may be interrupted so that a different task can run

non-preemptive scheduling once a task begins executing it runs until it is finished

centralized scheduling scheduling decisions are made at a single location

distributed scheduling scheduling decisions are made at different locations (matches distributed systems better)

There are at least two time scales with which we are concerned in scheduling. The most obvious one is the instantaneous scale: the scheduler must at each instant arrange for the "correct" process to use the processor. This scheduling typically uses a simple rule such as "run the highest priority ready process". Then, we need some way to assign priorities so that the simple rule of the instantaneous scheduler meets the long-term goal of meeting real-time deadlines.

The first question we should ask, given a set of real-time requirements is "can the set of tasks be scheduled to meet all of their deadlines?" That is, is there a *feasible schedule* for these tasks? There are several obvious tests for *necessary* properties of the task set.

- For each task in the set, $T_{Xmax} \geq T_{WCET}$ – the deadline must be at least as big as the worst-case execution time
- For each task in the set, the minimum interarrival time must be bigger than the deadline, $T_{Rmin} \geq T_{Xmax}$
- Utilization: $\sum_{i=1}^{n}(T_{WCET_i}/T_{R_i} \leq U_{max} \leq 1$. ($T_R$ is the period of a task.) Obviously, the CPU utilization cannot exceed 100%, but for some scheduling disciplines smaller bounds are known (U_{max}). And for some disciplines there are known U_{max}that will assure that this utilization test is a *sufficient* test as well.

Static scheduling and the rate-monotonic approach

Recall that static scheduling is done off-line for periodic tasks. Periodic tasks are characterized at least by their period T_R and their worst-case execution time T_{WCET}. If the periods of a set of tasks are all multiples of the smallest period a *rate monotonic scheduler* can achieve 100% utilization with $T_{Xmax} = T_{WCET}$ for each task.

In a rate monotonic scheduler the priority of each task is its period, T_R (smaller number, higher priority). The rule is run the highest priority ready task (with preemption).

Example. Fig. 12.10

Dynamic scheduling of sporadic tasks

The *earliest-deadline-first* scheduler is commonly used for sporadic tasks. Under certain constraints on the task set, it can be shown to be optimal. The idea is that each time the scheduler chooses a task, it chooses the one with the shortest deadline.

Skipping - priority inversion and distributed scheduling; clock synchronization

Fault tolerance for acuators and sensors

We have mentioned fault tolerance as a necessity for real-time computing. CptS 562 takes up the subject of fault-tolerant computing in greater detail at an advanced graduate level. That course will be taught next Spring semester if you're still around. (Not in Spring 2005.)

In the context of real-time control systems, fault tolerant computing is not all that is required: we also need fault-tolerant sensors and actuators. Our previous look at consensus suggests an approach to constructing fault-tolerant sensors. If we build multiple sensors and sensor representatives we can apply consensus or voting techniques to determine a value for the sensed quantity that is correct even in the presence of faults.

Reliable actuators are fun to look at, such as the reliable valve in Fig. 12.17. It opens or closes reliably even if one of the component valves is stuck at any position between open and closed, or even if it works completely backwards (opens when commanded to close, closes when commanded to open).

Next time: Chapt 16 - Security