How Does Control Timing Affect Performance?

Cervin, Anton; Henriksson, Dan; Lincoln, Bo; Eker, Johan; Årzén, Karl-Erik

Published in:
Control Systems Magazine

DOI:
10.1109/MCS.2003.1200240

2003

FEATURE

How Does Control Timing Affect Performance?

Analysis and Simulation of Timing Using Jitterbug and TrueTime

Control systems are becoming increasingly complex from both the control and computer science perspectives. Today, even seemingly simple embedded control systems often contain a multitasking real-time kernel and support networking. At the same time, the market demands that the cost of the system be kept at a minimum. For optimal use of computing resources, the control algorithm and the control software designs need to be considered at the same time. For this reason, new computer-based tools for real-time and control codesign are needed.

Many computer-controlled systems are distributed systems consisting of computer nodes and a communication network connecting the various systems. It is not uncommon for the sensor, actuator, and control calculations to reside on different nodes, as in vehicle systems, for example. This gives rise to networked control loops (see [1]). Within the individual nodes, the controllers are often implemented as one or several tasks on a microprocessor with a real-time operating system. Often the microprocessor also contains tasks for other functions (e.g., communication and user interfaces). The operating system typically uses multiprogramming to multiplex the execution of the various tasks. The CPU time and the communication bandwidth can hence be viewed as shared resources for which the tasks compete.

Digital control theory normally assumes equidistant sampling intervals and a negligible or constant control delay from sampling to actuation. However,

©MASTERSERIES

By Anton Cervin, Dan Henriksson, Bo Lincoln, Johan Eker, and Karl-Erik Årzén

Cervin (anton@control.lth.se), Henriksson, Lincoln, Eker, and Årzén are with the Department of Automatic Control, Lund Institute of Technology, Box 118, SE-221 00 Lund, Sweden.

0272-1708/03/$17.00©2003IEEE

this can seldom be achieved in practice. Within a node, tasks interfere with each other through preemption and blocking when waiting for common resources. The execution times of the tasks themselves may be data dependent or may vary due to hardware features such as caches. On the distributed level, the communication gives rise to delays that can be more or less deterministic depending on the communication protocol. Another source of temporal nondeterminism is the increasing use of commercial off-the-shelf (COTS) hardware and software components in real-time control (e.g., general-purpose operating systems such as Windows and Linux and general-purpose network protocols such as Ethernet). These components are designed to optimize average-case rather than worst-case performance.

The temporal nondeterminism can be reduced by the proper choice of implementation techniques and platforms. For example, time-driven static scheduling improves determinism, but at the same time it reduces the flexibility and limits the possibilities for dynamic modifications. Other techniques of a similar nature are time-driven architectures such as TTA [2] and synchronous programming languages such as Esterel, Lustre, and Signal [3]. Even with these techniques, however, some level of temporal nondeterminism is unavoidable.

The delay and jitter introduced by the computer system can lead to significant performance degradation. To achieve good performance in systems with limited computer resources, the constraints of the implementation platform must be taken into account at design time. To facilitate this, software tools are needed to analyze and simulate how timing affects control performance. This article describes two such tools: Jitterbug (http://www.control.lth.se/~lincoln/jitterbug) and TrueTime (http://www.control.lth.se/~dan/TrueTime).

The Software Tools

Jitterbug is a MATLAB-based toolbox that computes a quadratic performance criterion for a linear control system under various timing conditions. The tool can also compute the spectral density of the signals in the system. Using the toolbox, one can easily and quickly assert how sensitive a control system is to delay, jitter, lost samples, etc., without resorting to simulation. The tool is quite general and can also be used to investigate jitter-compensating controllers, aperiodic controllers, and multirate controllers. The main contribution of the toolbox, which is built on well-known theory (linear quadratic Gaussian (LQG) theory and jump linear systems), is to make it easy to apply this type of stochastic analysis to a wide range of problems.

The use of Jitterbug assumes knowledge of sampling period and latency distributions. This information can be difficult to obtain without access to measurements from the true target system under implementation. Also, the analysis cannot capture all the details and nonlinearities (especially in the real-time scheduling) of the computer system. A natural approach is to use simulation instead. However, today's simulation tools make it difficult to simulate the true temporal behavior of control loops. Normally time delays are introduced in the control loop representing average-case or worst-case delays. Taking a different approach, the MATLAB/Simulink-based tool TrueTime facilitates simulation of the temporal behavior of a multitasking real-time kernel executing controller tasks. The tasks are controlling processes that are modeled as ordinary Simulink blocks. TrueTime also makes it possible to simulate simple models of communication networks and their influence on networked control loops. Different scheduling policies may be used (e.g., priority-based preemptive scheduling and earliest-deadline-first (EDF) scheduling). (For more on real-time scheduling, see [4].)

Jitterbug is a MATLAB-based toolbox that computes a quadratic performance criterion for a linear control system under various timing conditions.

TrueTime can also be used as an experimental platform for research on dynamic real-time control systems. For instance, it is possible to study compensation schemes that adjust the control algorithm based on measurements of actual timing variations (i.e., to treat the temporal uncertainty as a disturbance and manage it with feedforward or gain scheduling). It is also easy to experiment with more flexible approaches to real-time scheduling of controllers, such as feedback scheduling [5]. There the available CPU or network resources are dynamically distributed according to the current situation (CPU load, the performance of the different loops, etc.) in the system.

Comparison of the Tools

Jitterbug offers a collection of MATLAB routines that allow the user to build and analyze simple timing models of computer-controlled systems. A control system is built by connecting a number of continuous- and discrete-time systems. For each subsystem, optional noise and cost specifications may be given. In the simplest case, the discrete-time systems are assumed to be updated in order during the control period. For each discrete system, a random delay (described by a discrete probability density function) can be specified that must elapse before the next system is updated. The total cost of the system (summed over all subsystems) is computed algebraically if the timing model system is periodic or iteratively if the timing model is aperiodic.

To make the performance analysis feasible, Jitterbug can only handle a certain class of system. The control system is built from linear systems driven by white noise, and the performance criterion to be evaluated is specified as a quadratic, stationary cost function. The timing delays in one period are assumed to be independent from the delays in the previous period. Also, the delay probability density functions are discretized using a time-grain that is common to the whole model.

Even though a quadratic cost function can hardly capture all aspects of a control loop, it can still be useful when one wants to quickly judge several possible controller implementations against each other. A higher value of the cost function typically indicates that the closed-loop system is less stable (i.e., more oscillatory), and an infinite cost means that the control loop is unstable. The cost function can easily be evaluated for a large set of design parameters and can be used as a basis for the control and real-time design.

TrueTime makes it possible to study more general and detailed timing models of computer-controlled systems. The toolbox offers two Simulink blocks: a real-time kernel block and a real-time network block. The delays in the control loop are captured by simulation of the execution of tasks in the kernel and the transmission of messages over the network.

Being a simulation tool, TrueTime is not restricted to the evaluation of a quadratic performance criterion but can be used to evaluate any time-domain behavior of the control loop. If there are many random variables, however, very long simulations may be needed to draw conclusions about the system.

The Simulink blocks are event driven, so there is no need to specify a time-grain for the model. The execution of a task can be simulated on an arbitrarily fine time scale by dividing the code into segments. Typically, it is enough to divide a control task into a few segments (for instance, Calculate and Update) to capture its temporal behavior. The code segments can be likened to the discrete-time subsystems in Jitterbug. A difference is that they can contain any user-written code (including calls to real-time primitives) and not just linear update equations.

Finally, although Jitterbug can only analyze the stationary behavior of a control loop, TrueTime can be used to investigate transient responses in conjunction with, for example, temporary CPU overloads. It can also be used to study systems where the controller and scheduling parameters are adapted to the current situation in the real-time control system.

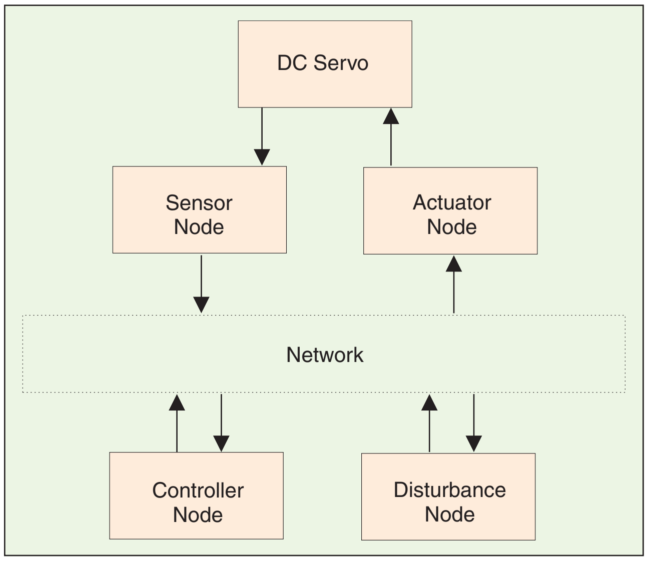


Figure 1. *The networked control system is used as a recurring example in the article.*

Networked Control System

As a recurring example in this article (among other examples), we will study a control loop that is closed over a communications network. Closing control loops over networks is becoming increasingly popular in embedded applications because of its flexibility, but it also introduces many new problems. From a control perspective, the computer system will introduce (possibly random) delays in the control loop. There is also the potential problem of lost measurement signals or control signals. From a real-time perspective, the first problem is figuring out the temporal constraints (deadlines, etc.) of the different tasks in the system and then scheduling the CPUs and the network such that all constraints are met during runtime.

In the example, we will study the setup shown in Figure 1. In our control loop, the sensor, the actuator, and the controller are distributed among different nodes in a network. The sensor node is assumed to be time driven, whereas the controller and actuator nodes are assumed to be event driven. At a fixed period h, the sensor samples the process and sends the measurement sample over the network to the controller node. There the controller computes a control signal and sends it over the network to the actuator node, where it is subsequently actuated. This kind of setup was studied in [6], where an optimal, delay-compensating LQG controller was derived. Here we are more interested in the interplay between control and real-time design and choose to study a simple process and controller.

We will assume that the process to be controlled is a dc servo and that the controller is a simple proportional-differential (PD) controller. In the Jitterbug section, we will study the impact of sampling period, delay, and jitter on the control-loop performance. A simple jitter-compensating controller is introduced where the parameters of the PD controller are adjusted according to the actual measured delay from the sensor node to the controller node. The delay model at this point is very simple: the delay from one node to another is described by a uniformly distributed random variable. In the TrueTime section, a more detailed delay model is obtained by simulating the execution of tasks in the nodes and the scheduling of messages in the network.

Long random delays are caused by interfering traffic generated by a disturbance node in the network. It will be seen that the behavior in the simulations agrees with the results obtained by the more simplistic analysis.

Analysis Using Jitterbug

In Jitterbug, a control system is described by two parallel models: a signal model and a timing model. The signal model is given by a number of connected, linear, continuous- and discrete-time systems. The timing model consists of a number of timing nodes and describes when the different discrete-time systems should be updated during the control period.

An example of a Jitterbug model is shown in Figure 2, where a computer-controlled system is modeled by four blocks. The plant is described by the continuous-time system G, and the controller is described by the three discrete-time systems H_1, H_2, and H_3. The system H_1 could represent a periodic sampler, H_2 could represent the computation of the control signal, and H_3 could represent the actuator. The associated timing model says that, at the beginning of each period, H_1 should first be executed (updated). Then there is a random delay τ_1 until H_2 is executed, and another random delay τ_2 until H_3 is executed. The delays could model computational delays, scheduling delays, or network transmission delays.

Signal Model

A *continuous-time system* is described by

$$\begin{aligned}\dot{x}_c(t) &= Ax_c(t) + Bu(t) + v_c(t)\\ y(t) &= Cx_c(t),\end{aligned}$$

where A, B, and C are constant matrices, and v_c is a continuous-time white noise process with covariance R_{1c}. (In the toolbox, it is also possible to specify discrete-time measurement noise. This will be interpreted as input noise at any connected discrete-time system.) The cost of the system is specified as

$$J_c = \lim_{T\to\infty} \frac{1}{T}\int_0^T \begin{pmatrix} x_c(t) \\ u(t)\end{pmatrix}^T Q_c \begin{pmatrix} x_c(t) \\ u(t)\end{pmatrix} dt,$$

where Q_c is a positive semidefinite matrix.

A *discrete-time system* is described by

$$\begin{aligned}x_d(t_{k+1}) &= \Phi x_d(t_k) + \Gamma u(t_k) + v_d(t_k)\\ y(t_k) &= Cx_d(t_k) + Du(t_k) + e_d(t_k),\end{aligned}$$

where Φ, Γ, C, and D are possibly time-varying matrices (see below). The covariance of the discrete-time white noise processes v_d and e_d is given by

$$R_d = \mathbf{E}\begin{pmatrix} v_d(t_k) \\ e_d(t_k)\end{pmatrix}\begin{pmatrix} v_d(t_k) \\ e_d(t_k)\end{pmatrix}^T.$$

The input signal u is sampled when the system is updated, and the state x_d and the output signal y are held between updates. The cost of the system is specified as

$$J_d = \lim_{T\to\infty} \frac{1}{T}\int_0^T \begin{pmatrix} x_d(t) \\ u(t)\end{pmatrix}^T Q_d \begin{pmatrix} x_d(t) \\ u(t)\end{pmatrix} dt,$$

where Q_d is a positive semidefinite matrix. Note that the update instants t_k need not be equidistant in time and that the cost is defined in continuous time.

The *total system* is formed by appropriately connecting the inputs and outputs of a number of continuous- and discrete-time systems. Throughout, multi-input, multi-output formulations are allowed, and a system may collect its inputs from a number of other systems. The total cost to be evaluated is summed over all continuous- and discrete-time systems:

$$J = \sum J_c + \sum J_d.$$

Timing Model

The timing model consists of a number of timing nodes. Each node can be associated with zero or more discrete-time systems in the signal model, which should be updated when the node becomes active. At time zero, the first node is activated. The first node can also be declared to be periodic (indicated by an extra circle in the illustrations), which means that the execution will restart at this node every h seconds. This is useful for modeling periodic controllers and also greatly simplifies the cost calculations.

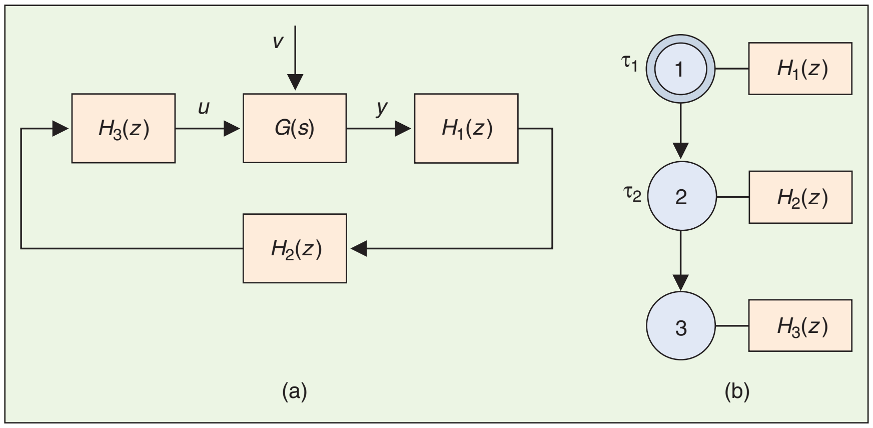


Figure 2. *A simple Jitterbug model of a computer-controlled system: (a) signal model and (b) timing model. The process is described by the continuous-time system $G(s)$, and the controller is described by the three discrete-time systems $H_1(z)$, $H_2(z)$, and $H_3(z)$, representing the sampler, the control algorithm, and the actuator. The discrete systems are executed according to the periodic timing model.*

Each node is associated with a time delay τ, which must elapse before the next node can become active. (If unspecified, the delay is assumed to be zero.) The delay can be used to model computational delay, transmission delay in a network, etc. A delay is described by a discrete-time probability density function

$$P_\tau = [P_\tau(0) \quad P_\tau(1) \quad P_\tau(2) \quad \ldots],$$

where $P_\tau(k)$ represents the probability of a delay of $k\delta$ seconds. The time-grain δ is a constant that is specified for the whole model.

TrueTime facilitates simulation of the temporal behavior of a multitasking real-time kernel executing controller tasks.

In periodic systems, the execution is preempted if the total delay $\sum \tau$ in the system exceeds the period h. Any remaining timing nodes will be skipped. This models a real-time system where hard deadlines (equal to the period) are enforced and the control task is aborted at the deadline.

An aperiodic system can be used to model a real-time system where the task periods are allowed to drift if there are overruns. It could also be used to model a controller that samples "as fast as possible" instead of waiting for the next period.

Node- and Time-Dependent Execution

The same discrete-time system may be updated in several timing nodes. It is possible to specify different update equations (i.e., different Φ, Γ, C, and D matrices) in the various cases. This can be used to model a filter where the update equations look different depending on whether or not a measurement value is available. An example of this type is given later.

It is also possible to make the update equations depend on the time since the first node became active. This can be used, for example, to model jitter-compensating controllers.

Alternative Execution Paths

For some systems, it is desirable to specify alternative execution paths (and thereby multiple next nodes). In Jitterbug, two such cases can be modeled:

- A vector n of next nodes can be specified with a probability vector p. After the delay, execution node $n(i)$ will be activated with probability $p(i)$. This can be used to model a sample being lost with some probability.
- A vector n of next nodes can be specified with a time vector t. If the total delay in the system since the node exceeds $t(i)$, node $n(i)$ will be activated next. This can be used to model time-outs and various compensation schemes.

Computation of Cost and Spectral Densities

The computation of the total cost is performed in three steps. First, the cost functions, the continuous-time noise, and the continuous-time systems are sampled using the time-grain of the model. Second, the closed-loop system is formulated as a jump linear system, where Markov nodes are used to represent the time steps in and between the execution nodes. Third, the stationary variance of all states in the system is calculated.

For periodic systems, the Markov state always returns to the periodic execution node every h/δ time steps. The stationary variance in the periodic execution node can then be obtained by solving a linear system of equations. The cost is then calculated over the time steps in one period. In this case, the cost calculation is fast and exact. It is also straightforward to compute the spectral densities of all outputs as observed in the periodic timing node. For systems without a periodic node, the variance must be computed iteratively. In both cases, the toolbox will return an infinite cost if the total system is not stable (in the mean-square sense). More details about Jitterbug's internal workings can be found in [7].

Networked Control System

The first example we will look at is the networked control system introduced earlier. We will begin by investigating how sensitive the control loop is to slow sampling and delays, and then we will look at delay and jitter compensation.

The Jitterbug model of the system was shown in Figure 2. The dc servo process is given by the continuous-time system

$$G(s) = \frac{1000}{s(s+1)}.$$

The process is driven by white continuous-time input noise. There is assumed to be no measurement noise.

The process is sampled periodically with the interval h. The sampler and the actuator are described by the trivial discrete-time systems

$$H_1(z) = H_3(z) = 1,$$

and the discrete-time PD controller is implemented as

$$H_2(z) = -K\left(1 + \frac{T_d}{h}\frac{z-1}{z}\right),$$

where the controller parameters are chosen as $K = 1.5$ and $T_d = 0.035$. (A real implementation would include a low-pass filter in the derivative part, but that is ignored here.)

The delays in the computer system are modeled by the two (possibly random) variables τ_1 and τ_2. The total delay from sampling to actuation is thus given by $\tau_{\text{tot}} = \tau_1 + \tau_2$. It is assumed that the total delay never exceeds the sampling period (otherwise Jitterbug would skip the remaining updates).

Finally, we need to specify the control performance criterion to be evaluated. As a cost function, we choose the sum of the squared process input and the squared process output:

$$J = \lim_{T\to\infty} \frac{1}{T}\int_0^T \left(y^2(t)+u^2(t)\right) dt. \tag{1}$$

An outline of the MATLAB commands needed to specify the model and compute the value of the cost function are given in Figure 3.

Sampling Period and Constant Delay

A control system can typically give satisfactory performance over a range of sampling periods. In textbooks on digital control, rules of thumb for sampling period selection are often given. One such rule suggests that the sampling interval h should be chosen such that

$$0.2 < \omega_b h < 0.6,$$

where ω_b is the bandwidth of the closed-loop system. In our case, a continuous-time PD controller with the given parameters would give a bandwidth of about $\omega_b = 80$ rad/s. This would imply a sampling period of between 2.5 and 7.5 ms. The effect of computational delay is typically not considered in such rules of thumb, however. Using Jitterbug, the combined effect of sampling period and computational delay can be easily investigated. In Figure 4, the cost function (1) for the networked control system has been evaluated for different sampling periods in the interval 1 to 10 ms and for constant total delay ranging from 0 to 100% of the sampling interval. As can be seen, a one-sample delay gives negligible performance degradation when $h = 1$ ms. When $h = 10$ ms, a one-sample delay makes the system unstable (i.e., the cost J goes to infinity).

Random Delays and Jitter Compensation

If system resources are very limited (as they often are in embedded control applications), the control engineer may have to live with long sampling intervals. Delay in the control loop then becomes a serious issue. Ideally, the delay should be accounted for in the control design. In many practical cases, however, even the mean value of the delay will be unknown at design time. The actual delay at runtime will vary from sample to sample due to real-time scheduling, the load of the system, etc. A simple approach is to use gain scheduling—the actual delay is measured in each sample, and the controller parameters are adjusted according to precalculated values that have been stored in a table. Since Jitterbug allows time-dependent controller parameters, such delay compensation schemes can also be analyzed using the tool.

In the Jitterbug model of the networked control system, we now assume that the delays τ_1 and τ_2 are uniformly distributed random variables between 0 and $\tau_{\max}/2$, where $\tau_{\max}$ denotes the maximum round-trip delay in the loop. A range of PD controller parameters (ranging from $K = 1.5$ and

```
G = 1000/(s*(s+1));                              Define the process
H1 = 1;                                          Define the sampler
H2 = -K*(1+Td/h*(z-1)/z);                     Define the controller
H3 = 1;                                         Define the actuator

Ptau1 = [ ... ];         Define delay probability distribution 1
Ptau2 = [ ... ];         Define delay probability distribution 2

N = initjitterbug(delta,h);                  Set time-grain and period
N = addtimingnode(N,1,Ptau1,2);                 Define timing node 1
N = addtimingnode(N,2,Ptau2,3);                 Define timing node 2
N = addtimingnode(N,3);                         Define timing node 3

N = addcontsys(N,1,G,4,Q,R1,R2);  Add plant, specify cost and noise
N = adddiscsys(N,2,H1,1,1);                 Add sampler to node 1
N = adddiscsys(N,3,H2,2,2);              Add controller to node 2
N = adddiscsys(N,4,H3,3,3);                Add actuator to node 3

N = calcdynamics(N);                 Calculate internal dynamics
J = calccost(N);                         Calculate the total cost
```

Figure 3. *This MATLAB script shows the commands needed to compute the performance index of the networked control system using Jitterbug.*

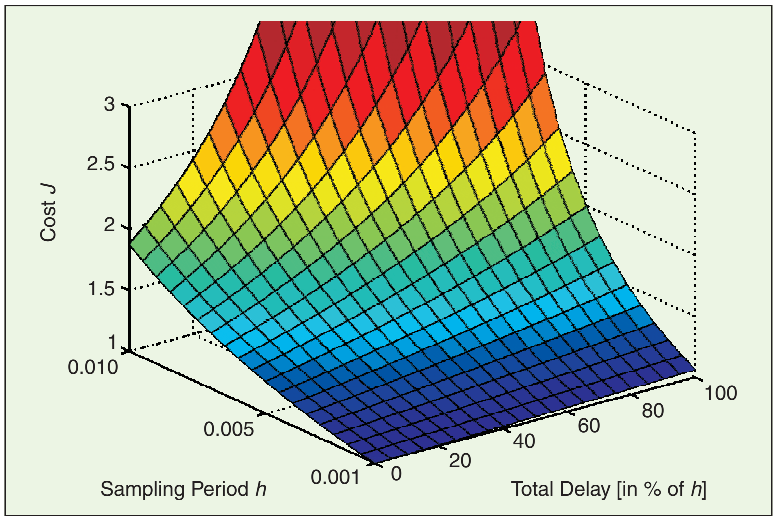


Figure 4. *Example of a cost function computed using Jitterbug. The plot shows the cost as a function of sampling period and delay in the networked control system example.*

$T_d = 0.035$ for zero delay to $K = 0.78$ and $T_d = 0.052$ for 7.5 ms delay) are derived and stored in a table. When a sample arrives at the controller node, only the delay τ_1 from sensor to controller is known, however, so the remaining delay is predicted by its expected value of $\tau_{\max}/4$.

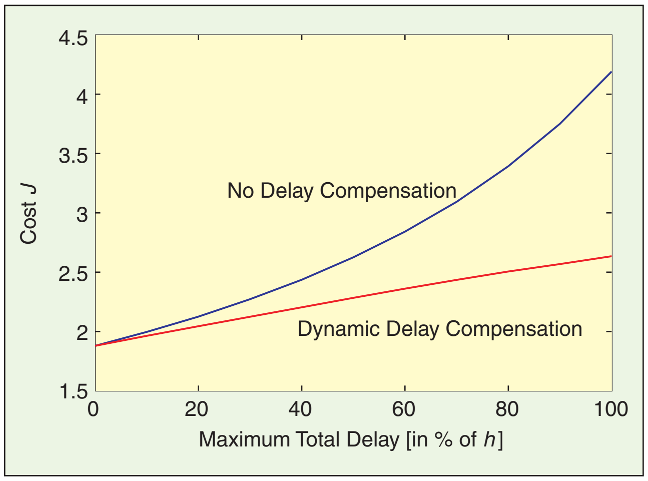


Figure 5. *Cost as a function of maximum delay in the networked control system example with random delays.*

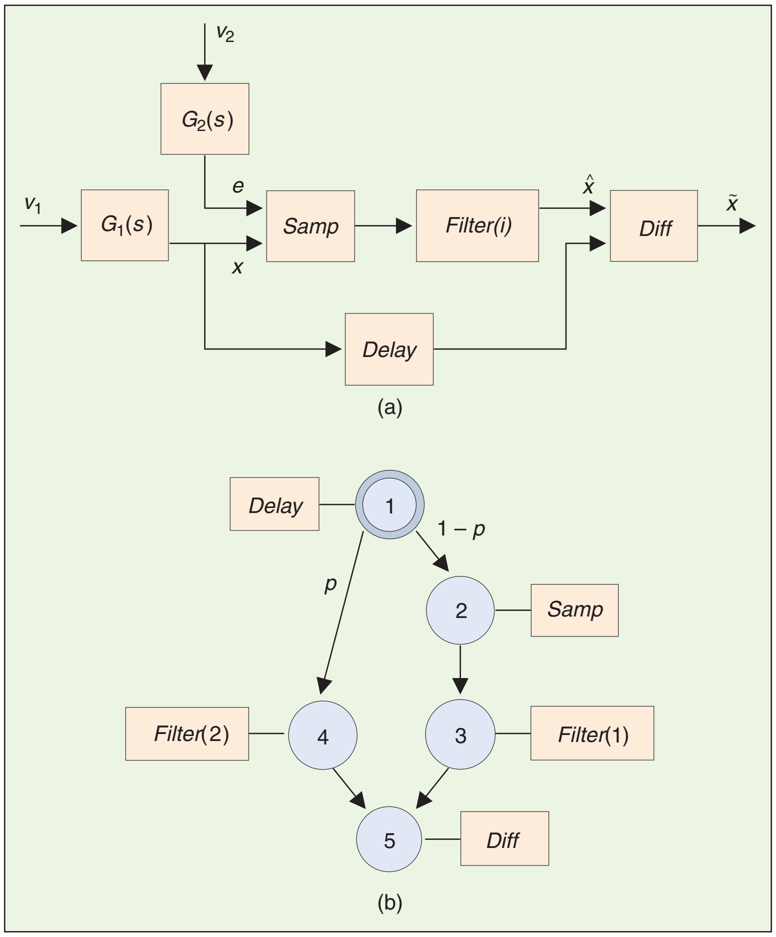


Figure 6. *Jitterbug model of the signal processing application: (a) signal model and (b) timing model.*

The sampling interval is set to $h = 10$ ms to make the effects of delay and jitter clearly visible. In Figure 5, the cost function (1) has been evaluated with and without delay compensation for values of the maximum delay ranging from 0 to 100% of the sampling interval. The cost increases much more rapidly for the uncompensated system. The same example will be studied in more detail later using the TrueTime simulator.

Signal Processing Application

As a second example, we will look at a signal processing application. Cleaning signals from disturbances using notch filters is important in many control systems. In some cases, the filters are very sensitive to lost samples due to their narrow-band frequency characteristics, and in real-time systems lost samples are sometimes inevitable. In this example, Jitterbug is used to evaluate the effects of lost samples in different filters and possible compensation techniques.

The setup is as follows. A good signal x (modeled as low-pass-filtered white noise) is to be cleaned from an additive disturbance e (modeled as band-pass-filtered white noise). An estimate $\hat{x}$ of the good signal should be found by applying a digital filter with the sampling interval $h = 0.1$ to the measured signal $x + e$. Unfortunately, a fraction p of the measurement samples are lost.

A Jitterbug model of the system is shown in Figure 6. The signals x and e are generated by filtered continuous-time white noise through the two continuous-time systems G_1 and G_2. The digital filter is represented as two discrete-time systems: *Samp* and *Filter*. The good signal is buffered in the system *Delay* and is then compared to the filtered estimate in the system *Diff*.

In the execution model, there is a probability p that the *Samp* system will not be updated. In that case, an alternative version, *Filter*(2), of the filter dynamics will be executed and used to compensate for the lost sample.

Two different filters are compared. The first filter is an ordinary second-order notch filter with two zeros on the unit circle. It is updated with the same equations even if no sample is available. The second filter is a second-order Kalman filter, which is based on a simplified model of the signal dynamics. In the case of a lost sample, only prediction is performed in the Kalman filter.

The performance of the filters is evaluated using the cost function

$$J = \lim_{T\to\infty} \frac{1}{T}\int_0^T \tilde{x}^2(t)\ dt,$$

which measures the variance of the estimation error. In Figure 7, the cost has been plotted for different probabilities of lost samples. The figure shows that the ordinary notch filter performs better in the case of no lost samples, but the Kalman filter performs better as the probability of lost samples increases. This is because the Kalman filter can perform prediction when no sample is available.

Simulation Using TrueTime

Analysis using Jitterbug can be used to quickly determine how sensitive a control system is to slow sampling, delay, jitter, and so on. For more detailed analysis as well as systemwide real-time design, the more general simulation tool TrueTime can be used.

In TrueTime, computer and network blocks are introduced. The computer blocks are event driven and execute user-defined tasks and interrupt handlers representing, e.g., I/O tasks, control algorithms, and network interfaces. The scheduling policy of the individual computer blocks is arbitrary and decided by the user. Likewise, in the network, messages are sent and received according to a chosen network model.

The level of simulation detail is also chosen by the user; it is often neither necessary nor desirable to simulate code execution on instruction level or network transmissions on bit level. TrueTime allows the execution time of tasks and the transmission times of messages to be modeled as constant, random, or data-dependent. Furthermore, TrueTime allows simulation of context switching and task synchronization using events or monitors.

TrueTime can be used in several ways:

- to investigate the effects of timing nondeterminism, caused, for example, by preemption or transmission delays, on control performance
- to develop compensation schemes that adjust the controller dynamically based on measurements of actual timing variations
- to experiment with new, more flexible approaches to dynamic scheduling, such as feedback scheduling of CPU time and communication bandwidth and quality-of-service (QoS)-based scheduling approaches
- to simulate event-driven control systems (e.g., engine controllers and distributed controllers).

Simulation Environment

The interfaces to the computer and network Simulink blocks are shown in Figure 8. Both blocks are event driven, with the execution determined by both internal and external events. Internal events are timely and correspond to events such as "a timer has expired," "a task has finished its execution," or "a message has completed its transmission." External events correspond to external interrupts, such as "a message arrived on the network" or "the crank angle passed 0°."

The block inputs are assumed to be discrete-time signals, except for the signals connected to the A/D converters of the computer block, which may be continuous-time signals. All outputs are discrete-time signals. The schedule and monitors outputs display the allocation of common resources (CPU, monitors, network) during the simulation.

The blocks are variable-step, discrete, MATLAB S-functions written in C++, the Simulink engine being used only for timing and interfacing with the rest of the model (the continuous dynamics). It should thus be easy to port the blocks to other simulation environments, provided these environments support event detection (zero-crossing detection).

The Computer Block

The computer block S-function simulates a computer with a simple but flexible real-time kernel, A/D and D/A converters, a network interface, and external interrupt channels.

Internally, the kernel maintains several data structures that are commonly found in a real-time kernel: a ready queue, a time queue, and records for tasks, interrupt handlers, monitors, and timers that have been created for the simulation.

The execution of tasks and interrupt handlers is defined by user-written code functions. These functions can be written either in C++ (for speed) or as MATLAB m-files (for ease

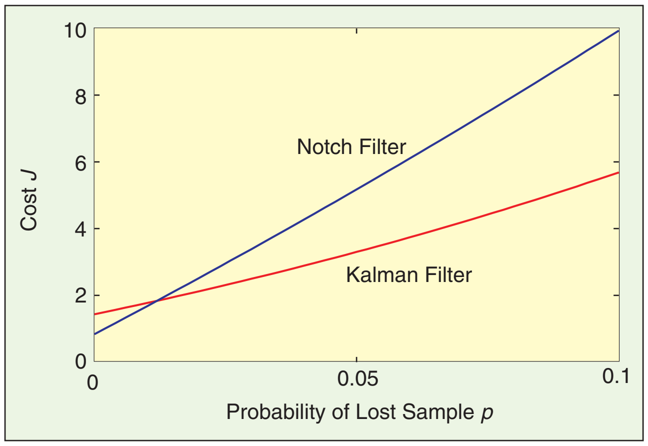
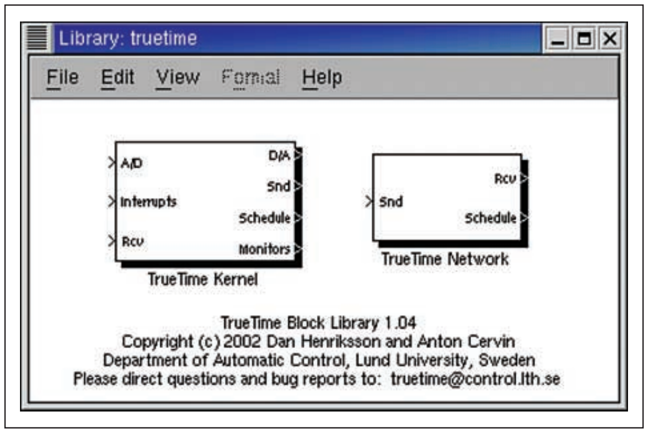


Figure 7. *The variance of the estimation error with the different filters as a function of the probability of lost samples.*



TrueTime Block Library 1.04
Copyright (c) 2002 Dan Henriksson and Anton Cervin
Department of Automatic Control, Lund University, Sweden
Please direct questions and bug reports to: truetime@control.lth.se

Figure 8. *The TrueTime block library. The Schedule and Monitor outputs display the allocation of common resources (CPU, monitors, network) during the simulation.*

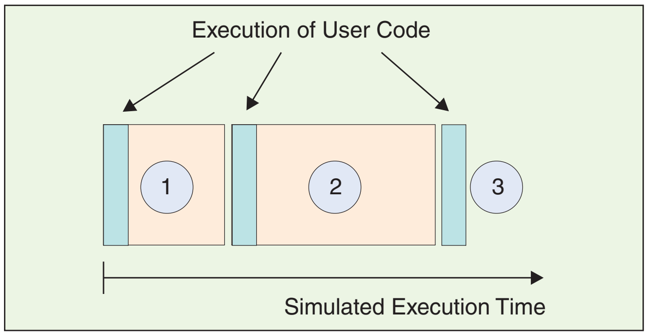


Figure 9. *The execution of the code associated with tasks and interrupt handlers is modeled by a number of code segments with different execution times. Execution of user code occurs at the beginning of each code segment.*

```
function [exectime, data] = myController(seg, data)
switch seg,
   case 1,
      data.y = ttAnalogIn(1);
      data.u = calculateOutput(data.y);
      exectime = 0.002;
   case 2,
      ttAnalogOut(1, data.u);
      updateState(data.y);
      exectime = 0.003;
   case 3,
      exectime = -1; % finished
end
```

Figure 10. *Example of a simple code function.*

Table 1. Examples of kernel primitives (pseudo syntax) that can be called from code functions associated with tasks and interrupt handlers.

`ttAnalogIn(ch)`	Get the value of an input channel
`ttAnalogOut(ch, val)`	Set the value of an output channel
`ttSendMsg(rec,data,len)`	Send message over network
`ttGetMsg()`	Get message from network input queue
`ttSleepUntil(time)`	Wait until a specific time
`ttCurrentTime()`	Current time in simulation
`ttCreateTimer(time,ih)`	Trigger interrupt handler at a specific time
`ttEnterMonitor(mon)`	Enter a monitor
`ttWait(ev)`	Await an event
`ttNotifyAll(ev)`	Activate all tasks waiting for an event
`ttSetPriority(val)`	Change the priority of a task
`ttSetPeriod(val)`	Change the period of a task

of use). Control algorithms may also be defined graphically using ordinary discrete Simulink block diagrams.

Tasks

The task is the main construct in the TrueTime simulation environment. Tasks are used to simulate both periodic activities, such as controller and I/O tasks, and aperiodic activities, such as communication tasks and event-driven controllers.

An arbitrary number of tasks can be created to run in the TrueTime kernel. Each task is defined by a set of attributes and a code function. The attributes include a name, a release time, a worst-case execution time, an execution time budget, relative and absolute deadlines, a priority (if fixed-priority scheduling is used), and a period (if the task is periodic). Some of the attributes, such as the release time and the absolute deadline, are constantly updated by the kernel during simulation. Other attributes, such as period and priority, are normally kept constant but can be changed by calls to kernel primitives when the task is executing.

In accordance with [8], it is furthermore possible to attach two overrun handlers to each task: a deadline overrun handler (triggered if the task misses its deadline) and an execution time overrun handler (triggered if the task executes longer than its worst-case execution time).

Interrupts and Interrupt Handlers

Interrupts may be generated in two ways: externally or internally. An external interrupt is associated with one of the external interrupt channels of the computer block. The interrupt is triggered when the signal of the corresponding channel changes value. This type of interrupt may be used to simulate engine controllers that are sampled against the rotation of the motor or distributed controllers that execute when measurements arrive on the network.

Internal interrupts are associated with timers. Both periodic timers and one-shot timers can be created. The corresponding interrupt is triggered when the timer expires. Timers are also used internally by the kernel to implement the overrun handlers described in the previous section.

When an external or internal interrupt occurs, a user-defined interrupt handler is scheduled to serve the interrupt. An interrupt handler works much the same way as a task, but it is scheduled on a higher priority level. Interrupt handlers will normally perform small, less time-consuming tasks, such as generating an event or triggering the execution of a task. An interrupt handler is defined by a name, a priority, and a code function. External interrupts also have a latency during which they are insensitive to new invocations.

Priorities and Scheduling

Simulated execution occurs at three distinct priority levels: the interrupt (highest priority), kernel, and task (lowest priority) levels. The execution may be preemptive or nonpreemptive; this can be specified individually for each task and interrupt handler.

At the interrupt level, interrupt handlers are scheduled according to fixed priorities. At the task level, dynamic-priority scheduling may be used. At each scheduling point, the priority of a task is given by a user-defined priority function, which is a function of the task attributes. This makes it easy to simulate different scheduling policies. For instance, a priority function that returns a priority number implies fixed-priority scheduling, whereas a priority function that returns a deadline implies deadline-driven scheduling. Predefined priority functions exist for most of the commonly used scheduling schemes.

Code

The code associated with tasks and interrupt handlers is scheduled and executed by the kernel as the simulation progresses. The code is normally divided into several segments, as shown in Figure 9. The code can interact with other tasks and with the environment at the beginning of each code segment. This execution model makes it possible to model input-output delays, blocking when accessing shared resources, etc. The simulated execution time of each segment is returned by the code function and can be modeled as constant, random, or even data-dependent. The kernel keeps track of the current segment and calls the code functions with the proper arguments during the simulation. Execution resumes in the next segment when the task has been running for the time associated with the previous segment. This means that preemption from higher-priority activities and interrupts may cause the actual delay between the segments to be longer than the execution time.

Figure 10 shows an example of a code function corresponding to the time line in Figure 9. The function implements a simple controller. In the first segment, the plant is sampled and the control signal is computed. In the second segment, the control signal is actuated and the controller states are updated. The third segment indicates the end of execution by returning a negative execution time.

The functions `calculateOutput` and `updateState` are assumed to represent the implementation of an arbitrary controller. The data structure `data` represents the local memory of the task and is used to store the control signal and measured variable between calls to the different segments. A/D and D/A conversion is performed using the kernel primitives `ttAnalogIn` and `ttAnalogOut`.

Besides A/D and D/A conversion, many other kernel primitives exist that can be called from the code functions. These include functions to send and receive messages over the network, create and remove timers, perform monitor operations, and change task attributes. Some of the kernel primitives are listed in Table 1.

Graphical Controller Representation

As an alternative to textual implementation of the controller algorithms, TrueTime also allows for graphical representation of the controllers. Controllers represented using ordinary discrete Simulink blocks may be called from within the code functions using the primitive `ttCallBlockSystem`. A block diagram of a PI controller is shown in Figure 11. The block system has two inputs, the reference signal and the

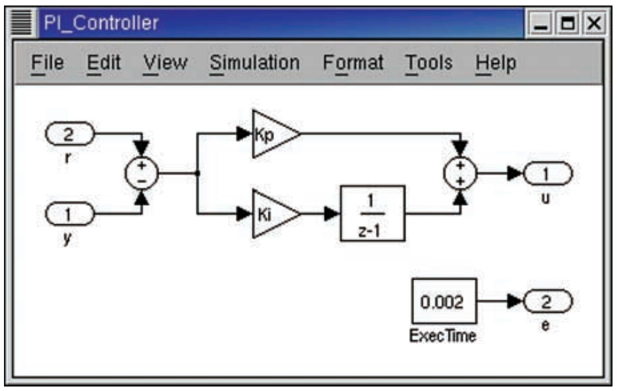
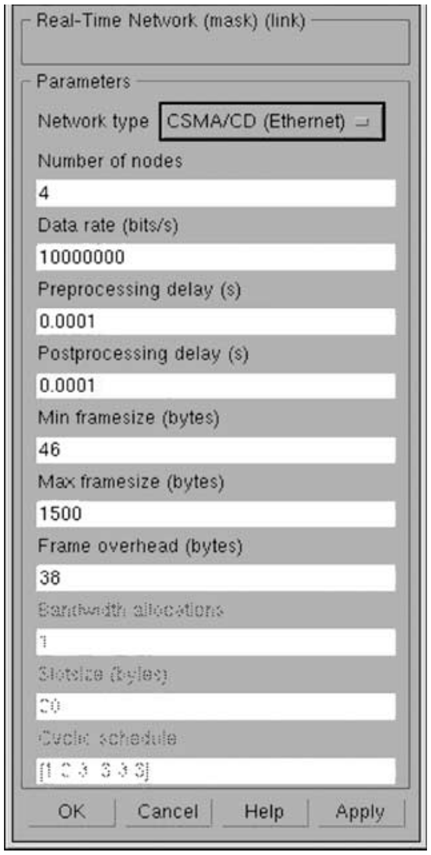


Figure 11. *Controllers represented using ordinary discrete Simulink blocks may be called from within the code functions. The example above shows a PI controller.*


Real-Time Network (mask) (link)

Parameters

Network type: CSMA/CD (Ethernet)

Number of nodes: 4

Data rate (bits/s): 10000000

Preprocessing delay (s): 0.0001

Postprocessing delay (s): 0.0001

Min framesize (bytes): 46

Max framesize (bytes): 1500

Frame overhead (bytes): 38

Bandwidth allocations: 1

Slotsize (bytes): 20

Cyclic schedule: [1 2 3 3 3 3]

Figure 12. *The dialog of the TrueTime Network block.*

process output, and two outputs, the control signal and the execution time.

Synchronization

Synchronization between tasks is supported by monitors and events. Monitors are used to guarantee mutual exclusion when accessing common data. Events can be associated with monitors to represent condition variables. Events may also be free (i.e., not associated with a monitor). This feature can be used to obtain synchronization between tasks where no conditions on shared data are involved.

Output Graphs

Depending on the simulation, several different output graphs are generated by the TrueTime blocks. Each computer block will produce two graphs, a computer schedule and a monitor graph, and the network block will produce a network schedule. The computer schedule will display the execution trace of each task and interrupt handler during the course of the simulation. If context switching is simulated, the graph will also display the execution of the kernel. If the signal is high, it means that the task is running. A medium signal indicates that the task is ready but not running (preempted), whereas a low signal means that the task is idle. In an analogous way, the network schedule shows the transmission of messages over the network, with the states representing sending (high), waiting (medium), and idle (low). The monitor graph shows which tasks are holding and waiting on the different monitors during the simulation. Generation of these execution traces is optional and can be specified individually for each task, interrupt handler, and monitor.

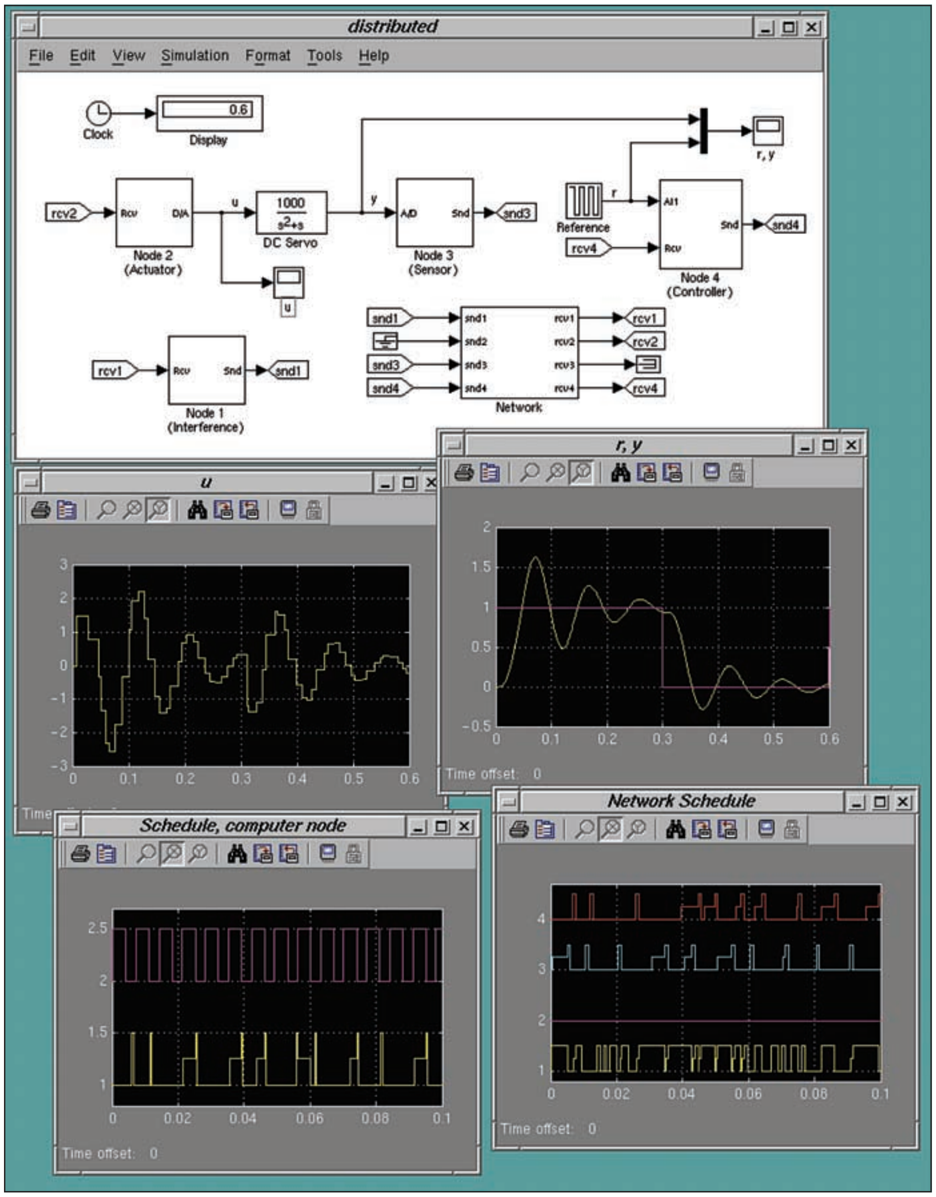


Figure 13. *TrueTime simulation of the networked control system. The poor control performance is a result of delays caused by colliding network transmissions and preemption in the controller node.*

The Network Block

The network model is similar to the real-time kernel model, albeit simpler. The network block is event driven and executes when messages enter or leave the network. A message contains information about the sending and receiving computer node, arbitrary user data (typically measurement signals or control signals), the length of the message, and optional real-time attributes such as a priority or a deadline.

In the network block, it is possible to specify the transmission rate, the medium access control protocol (CSMA/CD, CSMA/CA, round robin, FDMA, or TDMA), and a number of other parameters; see Figure 12. A long message can be split into frames that are transmitted in sequence, each with an additional overhead. When the simulated transmission of a message has completed, it is put in a buffer at the receiving computer node, which is notified by a hardware interrupt.

Networked Control System

As a first example of simulation in TrueTime, we again turn our attention to the networked control system. Using TrueTime, general simulation of the distributed control system is possible wherein the effects of scheduling in the CPUs and simultaneous transmission of messages over the network can be studied in detail. TrueTime allows simulation of different scheduling policies of the CPU and network and experimentation with different compensation schemes to cope with delays.

The TrueTime simulation model of the system contains one computer block for each node and a network block (see Figure 13). The time-driven sensor node contains a periodic task, which at each invocation samples the process and sends the sample to the controller node over the network. The controller node contains an event-driven task that is triggered each time a sample arrives over the network from the sensor node. Upon receiving a sample, the controller computes a control signal, which is then sent to the event-driven actuator node, where it is actuated. Finally, the interference node contains a periodic task that generates random interfering traffic over the network.

Initialization of the Actuator Node

Figure 14 shows the complete code needed to initialize the actuator node in this particular example. The computer block contains one task and one interrupt handler, and their execution is defined by the code functions `actcode` and `msgRcvHandler`, respectively. The task and interrupt handler are created in the `actuator_init` function together with an event (`packet`) used to trigger the execution of the task. The node is "connected" to the network in the function `ttInitNetwork` by supplying a node identification number and the interrupt handler to be executed when a message arrives at the node. In the `ttInitKernel` function, the kernel is initialized by specifying the number of A/D and D/A channels and the scheduling policy. The built-in priority function `prioFP` specifies fixed-priority scheduling. Other predefined scheduling policies include rate monotonic (`prioRM`), earliest deadline first (`prioEDF`), and deadline monotonic (`prioDM`) scheduling.

Simulations

In the following simulations, we will assume a CAN-type network where transmission of simultaneous messages is decided based on priorities of the packages. The PD controller executing in the controller node is designed assuming a 10-ms sampling interval. The same sampling interval is used in the sensor node.

In a first simulation, all execution times and transmission times are set equal to zero. The control performance resulting from this ideal situation is shown by the green curves in Figure 15.

Next we consider a more realistic simulation where execution times in the nodes and transmission times over the network are taken into account. The execution time of the controller is 0.5 ms, and the ideal transmission time from one node to another is 1.5 ms. The ideal round-trip delay is thus 3.5 ms. The packages generated by the disturbance node have high priority and occupy 50% of the network bandwidth. We further assume that an interfering, high-priority task with a 7-ms period and a 3-ms execution time is executing in the controller node. Colliding transmissions and preemption in the controller node will thus cause the round-trip delay to be even longer on average and time varying. The resulting degraded control performance is shown

```
%% Code function for the actuator task

function [exectime, data] = actcode(seg, data)

  switch seg,
    case 1,
      ttWait('packet');
      exectime = 0.0;
    case 2,
      data.u = ttGetMsg;
      exectime = 0.0005;
    case 3,
      ttAnalogOut(1, data.u);
      ttSetNextSegment(1); % wait for new msg
      exectime = 0.0;
  end

%% Code function for the network interrupt handler

function [exectime, data] = msgRcvHandler(seg, data)

  ttNotifyAll('packet');
  exectime = -1;

%% Initialization function
%% creating the task, interrupt handler and event

function actuator_init

  nbrOfInputs = 0;
  nbrOfOutputs = 1;
  ttInitKernel(nbrOfInputs, nbrOfOutputs, 'prioFP');

  priority = 5;
  deadline = 0.010;
  release = 0.0;
  ttCreateTask('act_task', deadline, priority, 'actcode');
  ttCreateJob('act_task', release);

  ttCreateInterruptHandler('msgRcv', 1, 'msgRcvHandler');
  ttInitNetwork(2, 'msgRcv'); % I am node 2
  ttCreateEvent('packet');
```

Figure 14. *Complete initialization of the actuator node in the networked control system simulation.*

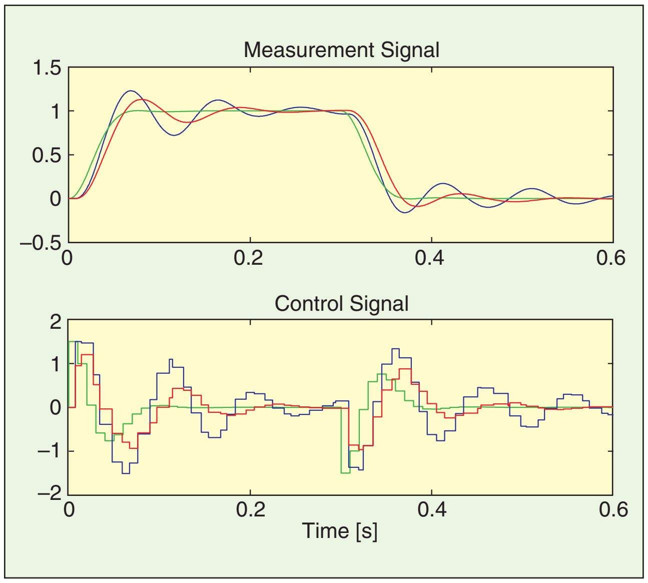


Figure 15. *Control performance for the networked control system in the ideal case (green), with interfering network messages and an interfering task in the controller node without compensation (blue) and with delay compensation (red).*

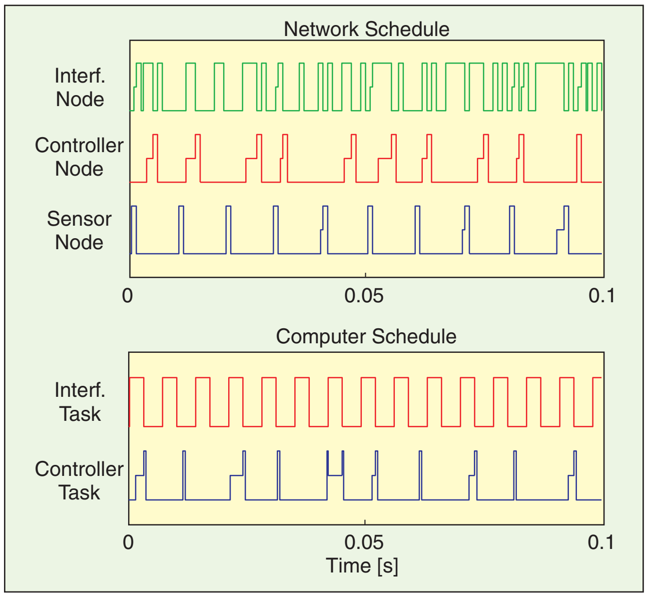


Figure 16. *Close-up of schedules showing the allocation of common resources: network (top) and controller node (bottom). A high signal means sending or executing, a medium signal means waiting, and a low signal means idle.*

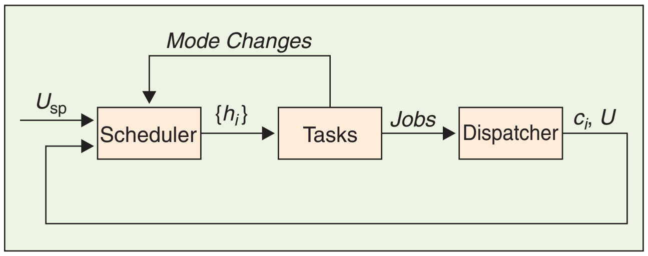


Figure 17. *The feedback scheduling structure.*

by the blue curves in Figure 15. The execution of the tasks in the controller node and the transmission of messages over the network can be studied in detail (see Figure 16).

Finally, a simple compensation is introduced to cope with the delays. The packages sent from the sensor node are now time stamped, which makes it possible for the controller to determine the actual delay from sensor to controller. The total delay is estimated by adding the expected value of the delay from controller to actuator. The control signal is then calculated based on linear interpolation among a set of controller parameters precalculated for different delays. Using this compensation, better control performance is obtained, as shown by the red curves in Figure 15.

Feedback Scheduling

As a second example, we will look at a feedback scheduling application. Some controllers, including hybrid controllers that switch between different modes, can have highly varying execution-time demands. This makes the real-time scheduling design for this type of controller difficult. Basing the real-time design on worst-case execution time (WCET) estimates may lead to low utilization, slow sampling, and poor control performance. On the other hand, basing the real-time design on average-case assumptions may lead to temporary CPU overloads and, again, poor control performance.

One way to solve the problem is to introduce feedback in the real-time system. The CPU overload problem can be resolved by online adjustment of the sampling frequencies of the hybrid controllers based on feedback from execution-time measurements. The scheduler may also use feedforward information from control tasks that are about to switch mode. The scheme was originally presented in [9] and is illustrated in Figure 17.

In this example, we consider feedback scheduling of a set of double-tank controllers. The double-tank process is described by nonlinear state-space equations of the form

$$\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} = \begin{pmatrix} -\alpha\sqrt{x_1} + \beta u \\ \alpha\sqrt{x_1} - \alpha\sqrt{x_2} \end{pmatrix}.$$

The objective is to control the level of the lower tank, x_2, using the pump, u. A hybrid controller for the double-tank process was presented in [10]. The controller consisted of two subcontrollers: a time-optimal controller for set-point changes and a proportional-integral-differential (PID) controller for steady-state regulation.

Measurements on the controller showed that in optimal control mode, the execution time was about three times longer than in PID control mode. The problem becomes pronounced when several hybrid controllers share a common computational unit. In the worst case, all controllers will be in optimal control mode at the same time, and the CPU load can become very high.

Simulations

It is assumed that three hybrid double-tank controllers should be scheduled on the same computer. The tanks have different time constants, $(T_1, T_2, T_3) = (210, 180, 150)$, and the corresponding controllers are therefore assigned different nominal sampling periods $(h_{\text{nom1}}, h_{\text{nom2}}, h_{\text{nom3}}) = (21, 18, 15)$ ms. Each controller is implemented as a separate TrueTime task. The simulated execution time of a controller in PID mode is $C_{\text{PID}} = 2$ ms and the simulated execution time of a controller in optimal control mode is $C_{\text{Opt}} = 10$ ms.

First, ordinary rate-monotonic scheduling is attempted. According to this scheduling principle, the task with the longest period gets the lowest priority. In the worst case, when all controllers are in optimal control mode, the utilization will be $U = \sum(C_i / h_i) = 1.7$ and the lowest-priority task (Controller 1) will be blocked. Simulation results are shown in Figures 18 and 19, displaying the control performance of the low-priority controller task and a closeup of the computer schedule. The performance of Controller 1 is very poor due to preemption from the higher-priority tasks.

Next, a feedback scheduler is introduced. The feedback scheduler is implemented as a task executing at the highest priority with a period of $h_{\text{FBS}} = 100$ ms and an execution time of $C_{\text{FBS}} = 2$ ms. It also executes an extra time whenever a task switches from PID to optimal mode. The feedback

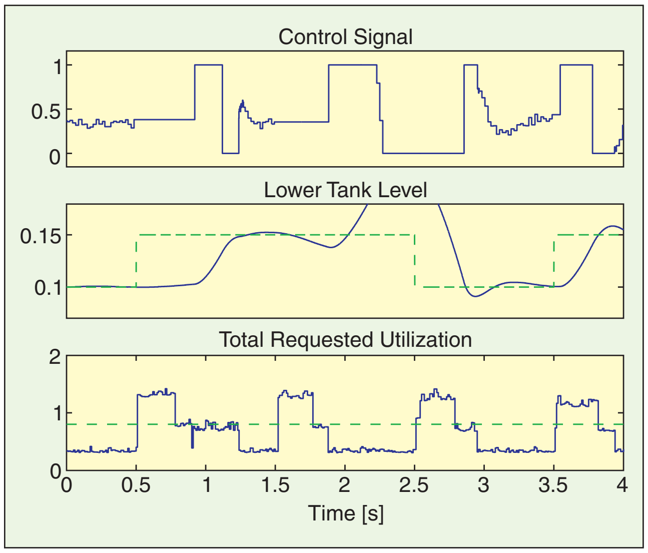


Figure 18. *Performance of Controller 1 under ordinary rate-monotonic scheduling. The CPU becomes overloaded and the controller is blocked, which deteriorates the performance.*

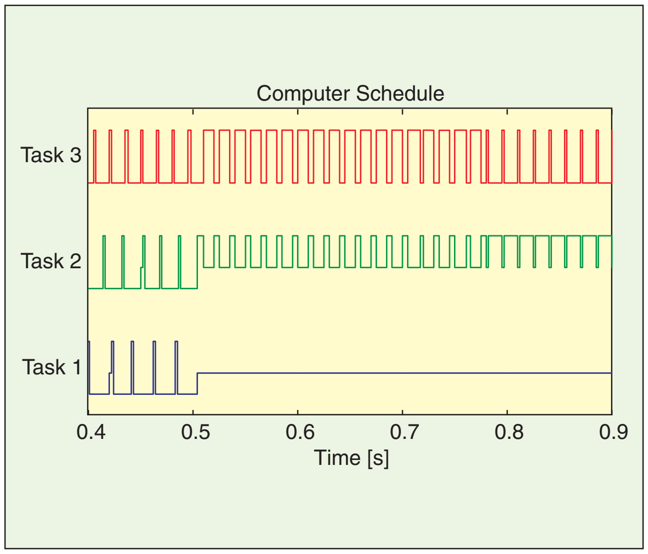


Figure 19. *Closeup of the computer schedule during ordinary rate-monotonic scheduling. When the system becomes overloaded, the low-priority controller is preempted for a significant amount of time.*

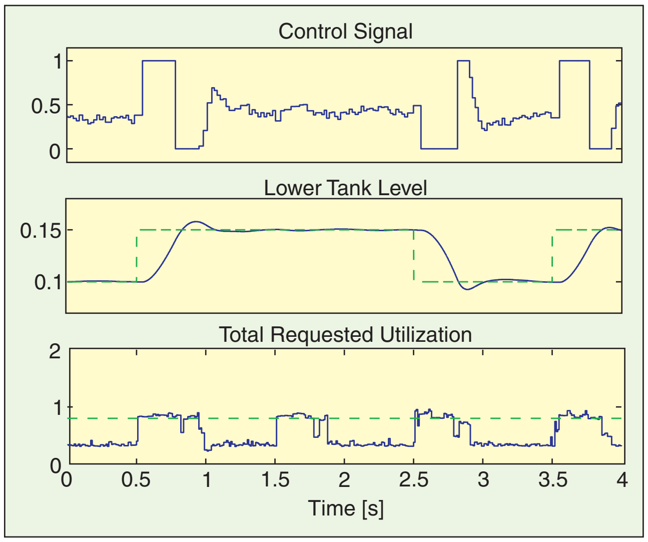


Figure 20. *Performance of Controller 1 under feedback scheduling. The CPU utilization is controlled to never exceed 0.8, and the control performance is good throughout.*

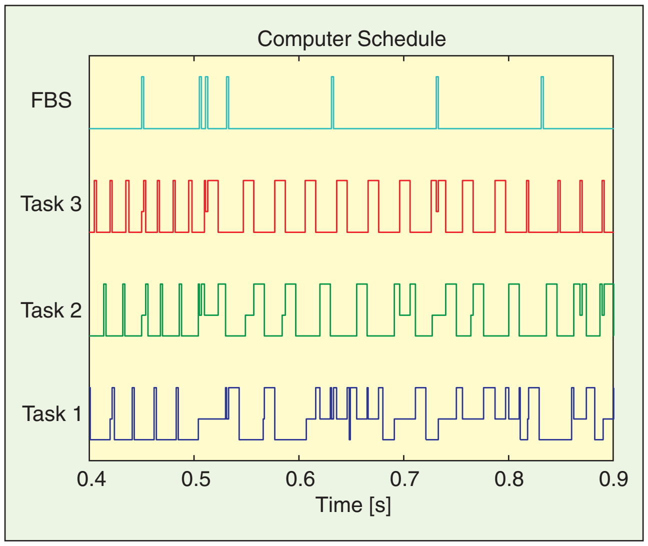


Figure 21. *Closeup of the computer schedule during feedback scheduling. The sampling intervals of the tasks are rescaled to avoid overload.*

scheduler estimates the workload of the controllers and rescales the task periods, if necessary, to achieve a utilization level of, at most, $U_{sp} = 0.8$. Results from a simulation are shown in Figures 20 and 21. The performance of Controller 1 is much better, even though it cannot always execute at its nominal period.

Conclusion

Designing a real-time control system is essentially a codesign problem. Choices made in the real-time design will affect the control design and vice versa. For instance, deciding on a particular network protocol will give rise to certain delay distributions that must be taken into account in the controller design. On the other hand, bandwidth requirements in the control loops will influence the choice of CPU and network speed. Using an analysis tool such as Jitterbug, one can quickly assert how sensitive the control loop is to slow sampling rates, delay, jitter, and other timing problems. Aided by this information, the user can proceed with more detailed, systemwide real-time and control design using a simulation tool such as TrueTime.

Jitterbug allows the user to compute a quadratic performance criterion for a linear control system under various timing conditions. The control system is described using a number of continuous- and discrete-time linear systems. A stochastic timing model with random delays is used to describe the execution of the system. The tool can also be used to investigate aperiodic controllers, multirate controllers, and jitter-compensating controllers.

TrueTime facilitates event-based cosimulation of a multitasking real-time kernel containing controller tasks and the continuous dynamics of controlled plants. The simulations capture the true, timely behavior of real-time controller tasks and communication networks, and dynamic control and scheduling strategies can be evaluated from a control performance perspective. The controllers can be implemented as MATLAB m-functions, C++ functions, or ordinary discrete-time Simulink blocks.

Acknowledgments

This work has been sponsored by ARTES (A network for Real-Time research and graduate Education in Sweden, http://www.artes.uu.se) and LUCAS (Lund University Center for Applied Software Research, http://www.lucas.lth.se).

References

[1] *Special Section on Networks and Control, IEEE Contr. Syst. Mag.*, vol. 21, Feb. 2001.

[2] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Boston, MA: Kluwer, 1997.

[3] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Boston, MA: Kluwer, 1993.

[4] J.W.S. Liu, *Real-Time Systems*. Upper Saddle River, NJ: Prentice-Hall, 2000.

[5] J. Eker, P. Hagander, and K.-E. Årzén, “A feedback scheduler for real-time control tasks,” *Contr. Eng. Practice*, vol. 8, no. 12, pp. 1369-1378, 2000.

[6] J. Nilsson, “Real-time control systems with delays,” Ph.D. dissertation, ISRN LUTFD2/TFRT-1049-SE, Dept. of Automatic Control, Lund Inst. Technol., Sweden, Jan. 1998.

[7] B. Lincoln and A. Cervin, “Jitterbug: A tool for analysis of real-time control performance,” in *Proc. 41st IEEE Conf. Decision and Control*, Las Vegas, NV, 2002, pp. 1319-1324.

[8] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull, *The Real-Time Specification for Java*. Reading, MA: Addison-Wesley, 2000.

[9] A. Cervin and J. Eker, “Feedback scheduling of control tasks,” in *Proc. 39th IEEE Conf. Decision and Control*, Sydney, Australia, 2000, pp. 4871-4876.

[10] J. Eker and J. Malmborg, “Design and implementation of a hybrid control strategy,” *IEEE Contr. Syst. Mag.*, vol. 19, pp. 12-21, Aug. 1999.

Anton Cervin received an M.Sc. in computer science and engineering from the Lund Institute of Technology, Sweden, in 1998. Since then, he has been a Ph.D. student in the Department of Automatic Control at Lund Institute of Technology. His research interest is real-time control systems, and his thesis work is about the integration of control and real-time scheduling.

Dan Henriksson received an M.Sc. in engineering physics from the Lund Institute of Technology, Sweden, in 2000. He is currently a Ph.D. student in the Department of Automatic Control at Lund Institute of Technology. His research interest is real-time control systems, involving flexible approaches to real-time control and scheduling design.

Bo Lincoln received an M.Sc. in computer science and engineering from the Linköping Institute of Technology, Sweden, in 1999, and he has been a Ph.D. student in the Department of Automatic Control at Lund Institute of Technology since then. His research interests include networked control systems and optimal control.

Johan Eker received a Ph.D. in automatic control from the Lund Institute of Technology, Sweden, in 1999. After completing a postdoctoral research position at the University of California, Berkeley, he will join the Ericsson Mobile Platforms research group in 2003. His interests are real-time control, software engineering, and programming language design, and he is currently working on the Cal actor language.

Karl-Erik Årzén received a Ph.D. in automatic control from the Lund Institute of Technology, Sweden, in 1987. He has been a professor in the Department of Automatic Control at Lund Institute of Technology since 2000. His research interests are real-time systems, real-time control, and programming languages for control applications.