TrueTime: Simulation of Networked and Embedded Control Systems

Anton Cervin

Department of Automatic Control
Lund University
Sweden

Contributions from Dan Henriksson, Martin Ohlin, and Karl-Erik Årzén

Outline of Lecture

1. Time and scheduling
2. Interrupt handlers and task synchronization
3. The network blocks
4. Summary
5. Specification of “mini-project”

Time primitives

- Simulink provides a global time base
- Each kernel block has its own local clock, with possible offset and drift
- Tasks may self-suspend (sleep)

```
ttCurrentTime
ttCurrentTime(time)
ttSleep(duration)
ttSleepUntil(time)
```

Priority functions

- The ready queue of the kernel is sorted by a priority function, which is a function of the task attributes
- Pre-defined priority functions exist for fixed-priority (`prioFP`), deadline-monotonic (`prioDM`), and earliest-deadline-first scheduling (`prioEDF`)
- Individual tasks can be made non-preemptible (`ttNonpreemptible`)
- Example: the EDF priority function (C++):

```
double prioEDF(UserTask* t)
   return t->absDeadline;
}
```

Scheduling Hooks (C++ only)

Pieces of user code that are executed at different stages during the execution of a task

- Arrival hook – when a job is created
- Release hook – when the job is first inserted in the ready queue
- Start hook – when the job executes its first segment
- Suspend hook – when the job is preempted, blocked or voluntarily goes to sleep
- Resume hook – when the job resumes execution
- Finish hook – when the job has executed its last segment

```
ttAttachHook(char* taskname, int ID, void (*hook)(UserTask*))
```

Constant Bandwidth Servers (CBS)

- Version 2.0 has built-in support for CBS scheduling [Abeni and Buttazzo, 1998]
- Assumes an EDF kernel (`prioEDF` must be selected)
- A CBS is characterized by
  - a period T_s
  - a budget Q_s
- A task associated with a CBS cannot execute more than the server budget period each server period (“sandboxing”)
- Implemented using scheduling hooks

```
ttCreateCBS(name, Qs, Ts, type)
ttAttachCBS(taskname, CBSname)
ttSetCBSParameters(name, Qs, Ts)
```

Multicore Scheduling

- Version 2.0 supports partitioned multicore scheduling
  - One ready queue per core
  - The same local scheduling policy in each core
- Tasks can be migrated between cores during runtime

```
ttSetNumberOfCPUs(nbr)
ttSetCPUAffinity(taskname, CPUnbr)
```

Data Logging

- Arbitrary events, intervals and values may be logged from the user code
- Written to MATLAB workspace when the simulation terminates
- Automatic task attribute logging provided for
  - Response time
  - Release latency
  - Start latency
  - Task execution time

```
ttCreateLog(logname, variable, size)
ttLogNow(logname)
ttLogStart(logname)
ttLogStop(logname)
ttLogValue(logname,value)
ttCreateLog(taskname, type, variable, size)
```

Example: Three Controllers on one CPU

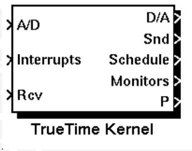
- Three controller tasks controlling three different DC-servo processes
- Sampling periods $h_i = [0.006 \;\; 0.005 \;\; 0.004]$ s
- Execution time of 0.002 s for all three tasks for a total utilization of $U = 1.23$
- Evaluate the effect of various scheduling policies on the control performance
- Use the logging functionality to monitor the response times and sampling latency under the different scheduling schemes

Outline of Lecture

1. Time and scheduling
2. Interrupt handlers and task synchronization
3. The network blocks
4. Summary
5. Specification of “mini-project”

Interrupt Handlers

- Code executed in response to interrupts
- Scheduled on a higher priority level than tasks, using fixed priorities
- Interrupt types
  - Timers (periodic or one-shot)
  - External (hardware) interrupts
  - Task overruns
  - Network interface




```
ttCreateHandler(hdlname, priority, codeFcn, data)
ttCreateTimer(timername, time, hdlname)
ttCreatePeriodicTimer(timername, start, period, hdlname)
ttRemoveTimer(timername)
ttAttachTriggerHandler(trignbr, hdlname)
```

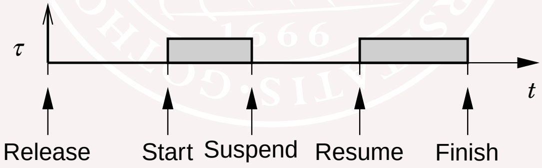
Overrun Handlers

- Two special interrupt handlers may be associated with each task (similar to Real-time Java)
  - A deadline overrun handler
  - An execution time overrun handler
- Can be used to dynamically handle prolonged computations and missed deadlines
- Implemented by internal timers and scheduling hooks

```
ttAttachDLHandler(taskname, hdlname)
ttAttachWCETHandler(taskname, hdlname)
```

Hooks for Overrun Handling

- **Release hook**: Set up a timer to expire at the absolute deadline of the task. The associated deadline overrun handler is called if the timer expires
- **Start hook**: Set up a timer corresponding to the worst-case execution time (WCET) of the task
- **Suspend hook**: Update execution time budget and remove WCET timer
- **Resume hook**: Set up the WCET timer for the remaining budget
- **Finish hook**: Remove both overrun timers



Task synchronization and communication

Four different simulated mechanisms of synchronization and communcation are supported:

- Monitors
- Events
- Mailboxes
- Semaphores

Monitors

- Monitors are used to model mutual exclusion between tasks that share common data
- Tasks waiting for monitor access are arranged according to their respective priorities
- The implementation supports standard priority inheritance to avoid priority inversion

`ttCreateMonitor(name)`

`ttEnterMonitor(name)` (blocking)

`ttExitMonitor(name)`

Events

- Events are used for task synchronization and may be free or associated with a monitor (condition variable)
- `ttNotifyAll` will move all waiting tasks to the monitor waiting queue or directly to the ready queue (if it is a free event)
- Events may, e.g., be used to trigger event-based controllers from an interrupt handler

`ttCreateEvent(name, monitorname)`

`ttWait(name)` (blocking)

`ttNotifyAll(name)`

Mailboxes

- Communication of data between tasks is supported by mailboxes
- A ring buffer is used to store incoming messages

```
ttCreateMailbox(name, maxsize)
ttTryPost(name, msg)
ttPost(name, msg) (blocking)
msg = ttTryFetch(name)
ttFetch(name) (blocking)
msg = ttRetrieve(name)
```

Semaphores

- Tasks can also be synchronized using counting semaphores

```
ttCreateSemaphone(name, initval, maxval)
ttTake(name) (blocking)
ttGive(name)
```

Readers–Writers Example

- Shared memory area must be protected using mutual exclusion
- Writers must wait for the buffer to be non-full
- Readers must wait for the buffer to be non-empty
- More sophisticated versions are possible (e.g. allowing multiple readers but only one writer)

Outline of Lecture

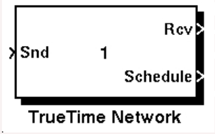
1. Time and scheduling
2. Interrupt handlers and task synchronization
3. The network blocks
4. Summary
5. Specification of “mini-project”

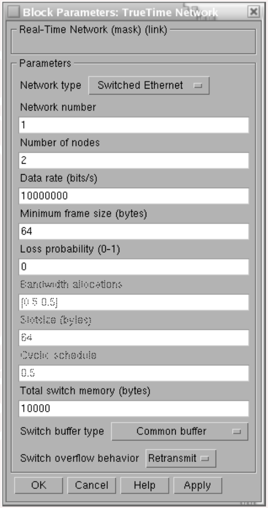
The Network Blocks

- Wired, wireless and ultrasound network blocks
- Each network is identified by a unique number
- Automatic connections between the kernel and network blocks (via hidden Goto blocks)
  - Needed only to trigger the blocks – no data is passed through the Simulink block connections

The Wired Network Block

- Supports eight common MAC layer policies:
  - CSMA/CD (Ethernet)
  - CSMA/AMP (CAN)
  - Round Robin (Token bus)
  - FDMA
  - TDMA
  - Switched Ethernet
  - Flexray
  - PROFINET IO
- Policy-dependent network parameters
- Generates a transmission schedule





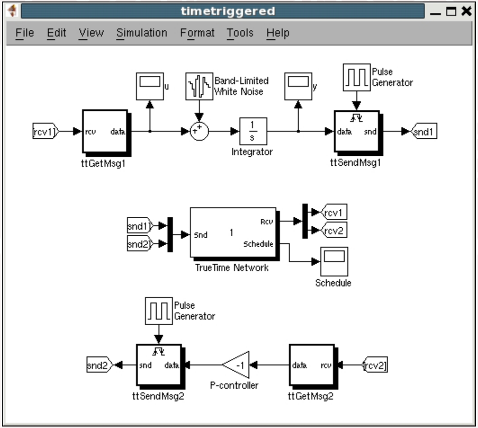
Network Communication

- Each node (kernel block) may be connected to several network blocks
- A dedicated interrupt handler may be associated with each network
  - Triggered as a packet arrives
- The actual message data can be an arbitrary MATLAB variable (scalar, struct, cell array, etc)
- Broadcast by specifying receiver number 0

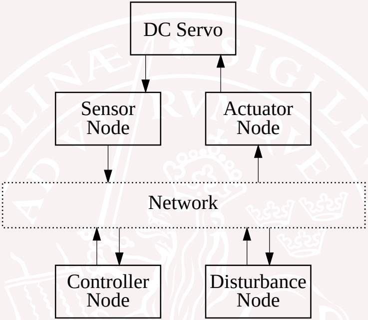
```
ttSendMsg([network receiver], data, length, priority)
ttGetMsg(network)
ttAttachNetworkHandler(network, hdlname)
```

Stand-Alone Network Interface Blocks

- Eliminates the need of Kernel blocks
- Event-triggered transmission of vector values



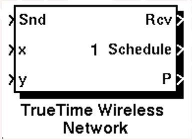
Example: Networked Control Loop

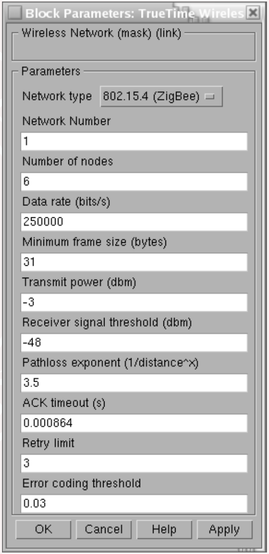


- Sensor/actuator node with time-driven sampler and event-driven actuator
- Event-driven controller node
- Disturbance node generating high-priority traffic

The Wireless Network Block

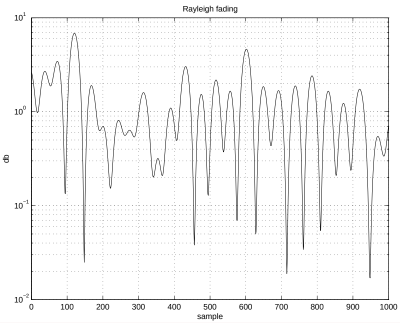
- Used in basically the same way as the wired network block
- Supports two common MAC layer policies:
  - 802.11b/g (WLAN)
  - 802.15.4 (ZigBee)
- Variable network parameters
- x and y inputs for node locations
- Generates a transmission schedule





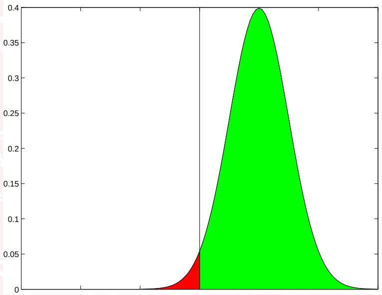
The Wireless Network Model

- Isotropic antennas
- Default path-loss formula: $$\frac{1}{d^a}$$
  - d – distance between nodes $\left(= \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}\right)$
  - a – environment parameter (e.g., 2–4)
- User-defined path-loss formula can be used to simulate e.g. Rayleigh fading



Transmission Errors

- The signal-to-interference ratio in the receiver is calculated
- Assuming additive Gaussian noise, the number of bit errors is drawn from a probability distribution



- If the number of bit error exceeds the specified error coding threshold, the packet is lost

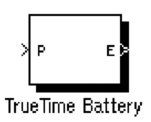
Wireless Network Parameters

- Data rate (bits/s)
- Transmit power (dBm)
  - configurable on a per node basis, can be reconfigured during run-time
- Receiver sensitivity (dBm)
- Path-loss exponent *or* used-defined path-loss function
- ACK timeout (s)
- Maximum number of retransmissions
- Error coding threshold

Higher-level network protocols

- Layers above MAC are not supported by TrueTime
- Higher-level protocols can however be implemented as applications
- Some examples we have implemented:
  - TCP with random early detection (RED)
  - AODV routing

The Battery Block



- Simulation of battery-powered devices
- Simple integrator model
  - Discharged or charged
- Energy sinks:
  - Computations, radio transmissions, usage of sensors and actuators, . . .
- If the power input to a kernel block reaches zero, the kernel freezes

```
ttSetKernelParameter('energyconsumption',value)
```

Dynamic Voltage Scaling

- The kernel CPU speed can be changed from the application (e.g., to consume more or less power)
- Independent from the local clock

```
ttSetKernelParameter('cpuscaling',value)
```

The Ultrasound Network Block

- Works similar to the other network blocks, but also simulates a propagation delay (speed of sound)
- Cannot send messages, but only broadcast ultrasound pings

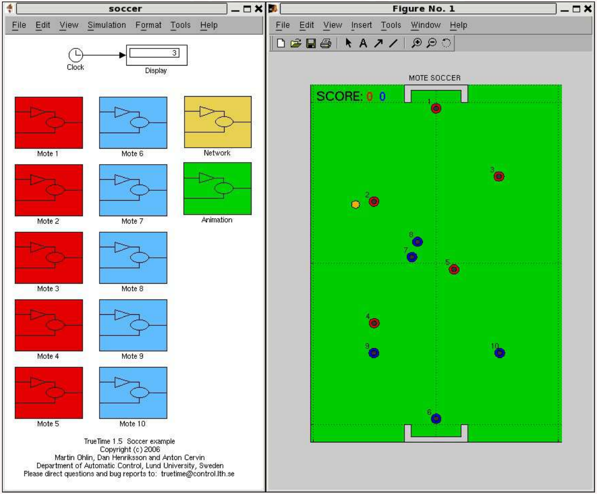
```
ttUltrasoundPing(network)
```

Example: Power Control

- Control of the DC-servo over a wireless network
- Two nodes:
  - sensor/actuator node
  - controller node
- Communication using wireless radio
- Dynamic control of radio transmission power
  - increase or decrease transmission power depending on link quality

Example: Soccer

- $5+5$ mobile robots communicating over a wireless network



Outline of Lecture

1. Time and scheduling
2. Interrupt handlers and task synchronization
3. The network blocks
4. Summary
5. Specification of “mini-project”

TrueTime – Summary

- Co-Simulation of:
  - computations inside the nodes
  - tasks, interrupt handlers, scheduling hooks
  - wired or wireless communication between nodes
  - sensor and actuator dynamics
  - mobile robot dynamics
  - dynamics of the environment
  - dynamics of the physical plant under control
  - the batteries in the nodes
- Control performance assessment
  - time domain
  - cost function calculations (via Monte Carlo simulations)

Some Limitations

- Developed as a research tool rather than as a tool for system developers
- Cannot express tasks and interrupt handlers directly using production code
  - Code must be divided into code segments and execution times must be assigned
- The zero-crossing functions generate quite a few events $\Rightarrow$ large models tend to be slow
- No built-in support for, e.g.,
  - higher-level network protocols
  - task migration between kernel blocks
  - ...

Outline of Lecture

1. Time and scheduling
2. Interrupt handlers and task synchronization
3. The network blocks
4. Summary
5. Specification of “mini-project”

“Mini-project”

- Simulation of a distributed consensus algorithm
- 6 mobile nodes should agree upon a location in the plane to meet
- Communication over a wireless network
- Each node is modeled by
  - A Kernel block
  - Two integrator blocks (for the x and y positions)

Distributed consensus

A simple algorithm to reach consensus regarding the state of n integrator agents with dynamics $\dot{z}_i = u_i$ can be expressed as

$$u_i(t) = K \sum_{j \in \mathcal{N}_i} \big(z_j(t) - z_i(t)\big) + b_i(t)$$

where K ăis a gain parameter and $\mathcal{N}_i$ are the neighbours of agent i (i.e., the nodes within communication range)

The bias term $b_i(t)$ should be zero if the nodes are to meet at a common location.

TrueTime Model

- A model with six integrator agents connected to a wireless network is provided (`consensus.mdl`)
- Each kernel block has to be configured – use the same initialization function and code function(s) for all blocks
- The consensus algorithm can be implemented as a simple periodic task (`ttCreatePeriodicTask`):
  - Collect all x and y values sent from your neighbours during the last period (repeated calls to `ttGetMsg`)
  - Read your own x and y coordinates (`ttAnalogIn`)
  - Compute the control signals in the x and y directions according to the formula
  - Output the control signals (`ttAnalogOut`)
  - Broadcast your own x and y position to your neighbours (`ttSendMsg` with receiver 0)

Extensions

- Add bias terms to the consensus algorithm to simulate “formation flight”
- Let one node lead (by not running the consensus algorithm) and let the other ones follow