Class in the Fall

- CSE 521S Wireless Sensor Networks
  - More properly: Internet of Things

- https://www.cse.wustl.edu/~lu/cse521s/
Adaptive QoS Control for Real-Time Systems

Chenyang Lu
CSE 520S
Challenges

- Classical real-time scheduling theory relies on accurate knowledge about workload and platform.

- New challenges under uncertainties
  - Maintain robust real-time properties in face of
    - unknown and varying workload
    - system failure
    - platform upgrade
  - Tuning, testing and certification of adaptive real-time systems
Challenge 1: Workload Uncertainties

- **Task execution times**
  - Heavily influenced by sensor data or user input
  - Unknown and time-varying

- **Disturbances**
  - Aperiodic events
  - Resource contention from subsystems
  - Denial of Service attacks

- **Examples:** power grid management, autonomous vehicles.
Challenge 2: System Failure

- Only maintaining **functional** reliability is not sufficient.
- Must also maintain robust real-time properties!

1. Norbert fails.
2. Move its tasks to other processors.
   - **hermione & harry** are overloaded!
Challenge 3: System Upgrade

- **Goal:** Portable application across hardware/OS platforms
  - Same application should “work” on multiple platforms

- **Existing real-time middleware**
  - Support functional portability
  - Lack QoS portability: must manually reconfigure applications on different platforms to achieve desired QoS
    - Profile execution times
    - Determine/implement allocation and task rate
    - Test/analyze schedulability
  - Time-consuming and expensive!
Example: nORB Middleware

**Application**

- CORBA Objects
  - Server
    - Worker thread
    - Conn. thread
  - Client
    - Timer thread
    - Priority queues
    - Conn. thread

**nORB**

- T1: 2 Hz
- T2: 12 Hz

**Operation Request Lanes**

*Manually set offline*
Challenge 4: Certification

- Uncertainties call for **adaptive** solutions.
  But...
- Adaptation can make things **worse**.
- Adaptive systems are difficult to test and certify.

![Graph showing CPU utilization over time](image)

An unstable adaptive system
Adaptive QoS Control

- Develop software feedback control in middleware
  - Achieve robust real-time properties for many applications
- Apply control theory to design and analyze control algorithms
  - Facilitate certification of embedded software

Sensor/human input? Disturbance?

Applications

Adaptive QoS Control Middleware

Drivers/OS/HW?

Available resources? HW failure?

Maintain QoS guarantees
- w/o accurate knowledge about workload/platform
- w/o hand tuning
Adaptive QoS Control

- FCS/nORB: feedback control scheduling for a single server
- FC-ORB: control for end-to-end tasks in distributed systems
Feedback Control Real-Time Scheduling

- Developers specify
  - **Performance specs**
    - CPU utilization = 70%; Deadline miss ratio = 1%.
  - **Tunable parameters**
    - Range of task rate: digital control loop, video/data display
    - Quality levels: image quality, filters
    - Admission control

- Guarantee specs by tuning parameters based on feedbacks
  - **Automatic**: No need for hand tuning
  - **Transparent** to developers
  - **Performance portability!**
Feedback Control Loop

Software Feedback Control Loop

Controller → Actuator
  control input

Actuator → Monitor
  change

Monitor → Controller
  sample

Reference

Computing System

Manipulated variable

Controlled variable
A Feedback Control Loop

Specs
$U_s = 70\%$

Tunable parameters
$R_1: [1, 5]\text{ Hz}$
$R_2: [10, 20]\text{ Hz}$

Controller

Actuator

Monitor

FC-U

{ $R_i(k+1)$ }

$U(k)$

Sensors, Inputs

Application?

Middleware

Drivers/OS?

HW?
The FC-U Algorithm

$U_s$: utilization reference
$K_u$: control parameter
$R_i(0)$: initial rate

1. Get utilization $U(k)$ from Utilization Monitor.
2. Utilization Controller:
   \[ B(k+1) = B(k) + K_u(U_s - U(k)) \] /* Integral Controller */
3. Rate Actuator adjusts task rates
   \[ R_i(k+1) = \frac{B(k+1)}{B(0)} R_i(0) \]
4. Inform clients of new task rates.
The Family of FCS Algorithms

- **FC-U** controls utilization
  - Performance spec: \( U(k) = U_s \)
  - Meet all deadlines if \( U_s \leq \) schedulable utilization bound
  - Relatively low utilization if utilization bound is pessimistic

- **FC-M** controls miss ratio
  - Performance spec: \( M(k) = M_s \)
  - High utilization
  - Does not require utilization bound to be known \textit{a priori}
  - Small but non-zero deadline miss ratio: \( M(k) > 0 \)

- **FC-UM** combines FC-U and FC-M
  - Performance specs: \( U_s, M_s \)
  - Allow higher utilization than FC-U
  - No deadline misses in “nominal” case
  - Performance bounded by FC-M
Dynamic Response

Steady State

Steady state error

Stability

Settling time

Transient State

Controlled variable

Reference

Time

Steady State
Control Analysis

- Rigorously designed based on feedback control theory

- **Analytic guarantees** on
  - Stability
  - Steady state performance
  - Transient state: settling time and overshoot
  - Robustness against variation in execution time

- Do **not** assume accurate knowledge of execution time
FCS/nORB Architecture

CORBA Objects

Server
- worker thread
- conn. thread
- feedback lane
- Operation Request Lanes

Client
- Timer thread
- Priority Queues
- conn. thread
- rate modulator

FCS/nORB

Application
Implementation

- Running on top of COTS Linux

- Deadline Miss Monitor
  - Instrument operation request lanes
  - Time-stamp operation request and response on each lane

- CPU Utilization Monitor
  - Interface with Linux /proc/stat file
  - Count idle time: “Coarse” granularity at jiffy (10 ms)

- Only controls server delay
Offline or Online?

- Offline
  - FCS is executed in testing phase on a new platform
  - Turned off after entering steady state
  - No run-time overhead
  - Cannot deal with varying workload

- Online
  - Run-time overhead (actually small…)
  - Robustness in face of changing execution times
Set-up

- OS: Red Hat Linux

- Hardware platform
  - Server A: 1.8GHz Celeron, 512 MB RAM
  - Server B: 1.99GHz Pentium 4, 256 MB RAM
  - Same client
  - Connected via 100 Mbps LAN

- Experiment
  - Overhead
  - Steady execution time (offline case)
  - Varying execution time (on-line case)
Server Overhead

- Overhead: FC-UM > FC-M > FC-U
- FC-UM increases CPU utilization by <1% for a 4s sampling period.
Performance Portability
Steady Execution Time

- **Same** CPU utilization (and no deadline miss) on different platforms w/o hand-tuning!

**FC-U on Server A**
- 1.8GHz Celeron, 512 MB RAM

**FC-U on Server B**
- 1.99GHz Pentium 4, 256 MB RAM

$U_s = 70\%$
Steady-state Deadline Miss Ratio

- FC-M enforces miss ratio spec
- FC-U, FC-UM causes no deadline misses

Average Deadline Miss Ratio in Steady State

- $M_s = 1.5\%$
Steady-State CPU Utilization

- FC-U, FC-UM enforces utilization spec
- FC-M achieves higher utilization

Average CPU Utilization in Steady State

- FC-U: 70.01%
- FC-M: 98.93%
- FC-UM: 74.97%

\[ U_s = 70\% \quad \text{and} \quad U_s = 75\% \]
Robust Guarantees

Varying Execution Time

**Same** CPU utilization and no deadline miss in steady state despite changes in execution times!
Tolerance to Load Increase

- **Surprise**
  - Server crashes under FC-M when execution time increases
  - FCS/nORB threads run at real-time priority
  - Kernel starvation when CPU utilization reaches 100%

- **Tolerance margin of load increase**
  - FC-U, FC-UM: margin = 1/\(U_s\)-1
    - \(U_s=70\%\) → can tolerate \((1/0.7-1)=43\%\) increase in execution time
  - FC-M: small and unknown margin
    - Unsuitable when execution time can increase unexpectedly
Summary of Experimental Results

- FCS enforces specified CPU utilization or miss ratio in steady states
  - Experimental validation of control design and analysis

- Performance portability: FCS/nORB achieves same performance guarantees when
  - platform changes
  - execution time changes (within tolerance margin)

- Overhead acceptable → FCS can be used online
Summary: FCS/nORB

- Enable robust, performance-portable real-time software
- Program application once → runs on multiple platforms with robust performance guarantees!
References


Adaptive QoS Control Middleware

FCS/nORB: Single server control

FC-ORB: Distributed systems with end-to-end tasks
- Handle end-to-end tasks
- Fault tolerance
End-to-End Task Model

- Periodic task $T_i = \text{chain of subtasks } \{T_{ij}\} \text{ on different hosts}$
  - All subtasks run at a same rate
  - End-to-end deadline

- Task rate can be adjusted within a range
  - Trade-off between video quality and rate
  - Higher rate $\rightarrow$ better video quality & higher CPU utilization

![Diagram of task dependencies]

- Precedence Constraints
- Subtask
End-to-End Utilization Control

- CPU utilization
  - Too high → system overload → crash
  - Too low → poor performance (e.g. poor video quality)
  - Utilization < schedulable bound → meet deadlines

- Uncertainties: varying task execution times
  - Adjust task rates to compensate for variations

![Diagram showing precedence constraints between tasks T1, T11, T12, T13, and subtasks P1, P2, P3. The diagram illustrates precedence constraints with dotted arrows.]

Precedence Constraints
Subtask
Challenges

- **Multi-Input-Multi-Output (MIMO) control**

- Utilizations are **coupled** due to end-to-end tasks
  - Rate change affects all CPUs in the task chain

- **Constraints** on task rates

- **Stability** assurance
EUCON – End-to-end Utilization CONtrol

- Centralized control
- Designed based on Model Predictive Control (MPC) theory
- Invoked periodically to control the utilizations of all processors

Desired utilization bounds:

\[
\begin{bmatrix}
B_1 \\
\vdots \\
B_n
\end{bmatrix}
\]

Allowed rate range for tasks (constraints):

\[
\begin{bmatrix}
R_{\text{min},1} & R_{\text{max},1} \\
\vdots & \vdots \\
R_{\text{min},m} & R_{\text{max},m}
\end{bmatrix}
\]

Manipulated variables: Task rate changes

\[
\begin{bmatrix}
\Delta r_1(k) \\
\vdots \\
\Delta r_m(k)
\end{bmatrix}
\]

Controlled Variables: CPU utilizations

\[
\begin{bmatrix}
u_1(k) \\
\vdots \\
u_n(k)
\end{bmatrix}
\]
Control Theoretic Methodology

1. Model the controlled system
2. Design a controller
3. Analyze stability
Dynamic Model: One Processor

\[ u_i(k) = u_i(k - 1) + g_i \sum_{T_{jl} \in S_i} c_{jl} \Delta r_j(k - 1) \]

- **Sₖ**: set of subtasks on Pₖ
- **cₖ**: estimated execution time of Tᵢₖ running on Pₖ
  - may not be correct
- **gₖ**: utilization gain of Pₖ
  - ratio between actual and estimated change in utilization
  - unknown: models uncertainty in execution times
Dynamic Model: Multiple Processors

\[ u(k) = u(k-1) + GF\Delta r(k-1) \]

- **G**: diagonal matrix of utilization gains
- **F**: subtask allocation matrix
  - models the **coupling** among processors
  - \( f_{ij} = c_{jl} \) if task \( T_j \) has a subtask \( T_{jl} \) on processor \( P_i \)
  - \( f_{ij} = 0 \) if \( T_j \) has no subtask on \( P_i \)

\[
F = \begin{bmatrix}
  c_{11} & c_{21} & 0 \\
  0 & c_{22} & c_{31}
\end{bmatrix}
\]
Model Predictive Control

At a sampling instant

- Compute inputs for several future sampling periods $\Delta r(k), \Delta r(k+1), \ldots \Delta r(k+M-1)$
  to minimize a cost function in the future
- Cost in the future is predicted using
  i) feedback $u(k-1)$
  ii) approximate dynamic model
- Apply $\Delta r(k)$ to the system

At the next sampling instant:

- Shift time and re-compute $\Delta r(k+1), \Delta r(k+2), \ldots \Delta r(k+M)$
  based on feedback $u(k)$
Model Predictive Controller in EUCON

The Model Predictive Controller (MPC) in EUCON is a control strategy that uses a model of the system to predict future behavior and optimize control actions. The MPC system consists of several key components:

- **System Model**: Describes the dynamics of the system.
- **Rate Constraints**: Imposes limits on the rate of change of the control variables.
- **Least Squares Solver**: Solves the optimization problem to find the control inputs that minimize a cost function.
- **Cost Function**: Measures the deviation from the desired control path.
- **Reference Trajectory**: The ideal path that the system should follow.
- **Model Predictive Controller**: The overall system that integrates these components to control the system.

The control input is calculated as follows:

\[
\begin{bmatrix}
B_1 \\
\vdots \\
B_n \\
\end{bmatrix} 
\begin{bmatrix}
u_1(k) \\
\vdots \\
u_n(k) \\
\end{bmatrix} 
\]

\[
\Delta r_1(k+1) \\
\vdots \\
\Delta r_m(k+1) 
\]

The difference from the reference trajectory is calculated as:

\[
\Delta r_i(k+1) 
\]

The desired trajectory for \( u(k) \) to converge to \( B \) is shown in the diagram.
Stability Analysis

- **Stability**: system converges to equilibrium point from any initial condition
  - Equilibrium point = utilization set points $B$
  - Utilization of all processors $\rightarrow$ their set points whenever feasible

- Derive stability condition in term of $G$
  - Tolerable range of variation in execution times

**Guarantees on utilization despite uncertainty!**
Simulation: Stable System

![Graph showing CPU utilization over time](image)

- **Execution time factor = 0.5**
- **Actual execution times = \( \frac{1}{2} \) estimates**
Simulation: Unstable System

**Execution Time Factor = 7**

(actual execution times = 7x estimates)
average execution time of every subtask is seven times its estimation. In the beginning, the processors were fully utilized because of the long task execution times. At around time $30T_s$, the utilization drops sharply to almost zero and starts to oscillate. The utilization on $P_2$ also oscillates significantly. The system fails to converge to the utilization set point. This result is also consistent with our stability analysis that predicts the system will be unstable when the system gains exceed 5.95.

We plot the mean and standard deviation of utilization on $P_1$ during each run in Fig. 3a. Every data point is based on the measured utilization $u(k)$ from time $100T_s$ to $300T_s$ to exclude the transient response in the beginning of each run. The system performance is considered acceptable if the average utilization is within $[0.02]$ to the utilization set point, and the standard deviation is less than 0.05. Satisfying the requirement on average utilization ensures that the system achieves the desired utilization. Satisfying the requirement on standard deviation ensures that the utilization does not oscillate significantly. While the thresholds for acceptable performance depend on specific applications, the general conclusions drawn in this section are applicable to many applications. As shown in Fig. 3a, the average utilization remains close to the set point for execution-time factors between 0.20 and 5.95, and it starts deviating from the set point and increases linearly when the execution-time factor exceeds 6.00. When execution-time factor = 5.95, the average utilizations on $P_1$ and $P_2$ are 0.828 and 0.829, respectively. When execution-time factor increases to 6.00, however, the average utilization on $P_1$ and $P_2$ become 0.828 and 0.833, respectively. Based on the set point of 0.828 on both processors, the system becomes unstable (on $P_2$) when execution-time factor is in the range $[5.95; 6.00]$. This empirical result is close to the analysis which shows the system should remain stable when the gain is below 5.95 (see Section 5).

The standard deviation of utilization indicates the intensity of oscillation. As the execution-time factor increases from 0.2 to 3, the standard deviation remains less than 0.05 and the average utilization remains within $[0.02]$ to the set point. These results demonstrate that EUCON can enforce the same utilization guarantees when execution times deviate from the estimates as long as the execution-time factor remains below 3. However, the standard deviation is higher than 0.05 for execution-time factors between 4 and 6, although the system is analytically stable in this range. This result is consistent with our analysis in Section 5 that pessimistic estimation on execution times will reduce oscillation without underutilizing the CPUs.
FC-ORB

Feedback Controlled Object Request Broker

- End-to-end utilization control
  - Maintains desired utilizations on all CPUs

- End-to-end ORB architecture
  - Specialized for rate adaptation

- Task migration
  - Reliability in terms of both functionality and real-time performance
End-to-End Utilization Control Service

- Implements EUCON (End-to-end Utilization CONtrol)
- Provides functional and performance portability
End-to-End Object Request Broker

- **Release guard** for end-to-end tasks
- **Priority management**
  - Rate adaptation $\rightarrow$ continuous priority changes
  - Thread-per-priority $\rightarrow$ high overhead
  - Thread-per-subtask: change priority only when the order of task rates changes
Task Migration

- **Fault model:** permanent processor failure
- **Subtasks** have backups on different processors
- **Utilization control + fault tolerance**
  - Automatic controller reconfiguration
  - Handle overload caused by task migration

Utilizations: $\begin{bmatrix} u_1(k) \\ u_1'(k) \\ u_2(k) \\ u_2'(k) \\ u_3(k) \end{bmatrix}$

Rate changes: $\begin{bmatrix} \Delta r_1(k) \\ \Delta r_2(k) \end{bmatrix}$

Rate Modulator

Priority Manager

Utilization Monitor

Remote request lanes

Model Predictive Controller

Priority Manager

Utilization Monitor
FC-ORB Implementation

- Implemented based on FCS/nORB, nORB and ACE
- Specialized for memory constrained distributed real-time systems
- 7017 lines of C++ code
- Controller is implemented as a Dynamic Link Library (DLL) generated by MATLAB
Experimental Setup

- 12 tasks (25 subtasks) and 4 Pentium IV processors
- KURT Linux 2.4.22
- Rate Monotonic Scheduling
- Subtasks on Norbert have backups on other processors
Goal 1: Robust Utilization Control

Execution times change at runtime

Desired utilization: 73% (0.73)

Disturbance from external resource contention
Goal 2: Performance Portability

**Same utilization: portable performance on systems with different capacities**

exec time = 2x expected (running on slow machines)

exec times = expected/4 (running on fast machines)

**Desired utilization: 73% (0.73)**
Goal 3: Fault Tolerance

1. Norbert fails.
2. Move its tasks to other processors.
3. Reconfigure controller.
4. Control utilization by adjusting task rates.
Robust utilization control, despite
- unknown or varying execution times
- external disturbances

Performance portability

Fault tolerance, in terms of
- functionality
- real-time performance
Conclusion: Adaptive QoS Control

- **Feedback control**: robust real-time performance under uncertainty
- **Middleware**: provides reusable control services to real-time applications
- **Control analysis**: tuning and certification of adaptive software

**More**
- Advanced control: event-driven, discrete configurations.
- Coordination of multiple control policies
- Sophisticated fault tolerance techniques
- Certification/testing methodologies
Reading

- **Control of a single server**

- **Centralized control of distributed systems**

- **Feedback control theory for computing systems**
  - Tutorials: [https://www.cse.wustl.edu/~lu/control-tutorials/im09/](https://www.cse.wustl.edu/~lu/control-tutorials/im09/)