Overload Protection for Real-time CORBA Systems

Niklas Widell
Department of Communication Systems
Lund University
Sweden
niklasw@telecom.lth.se

LUND
UNIVERSITY

Outline of presentation

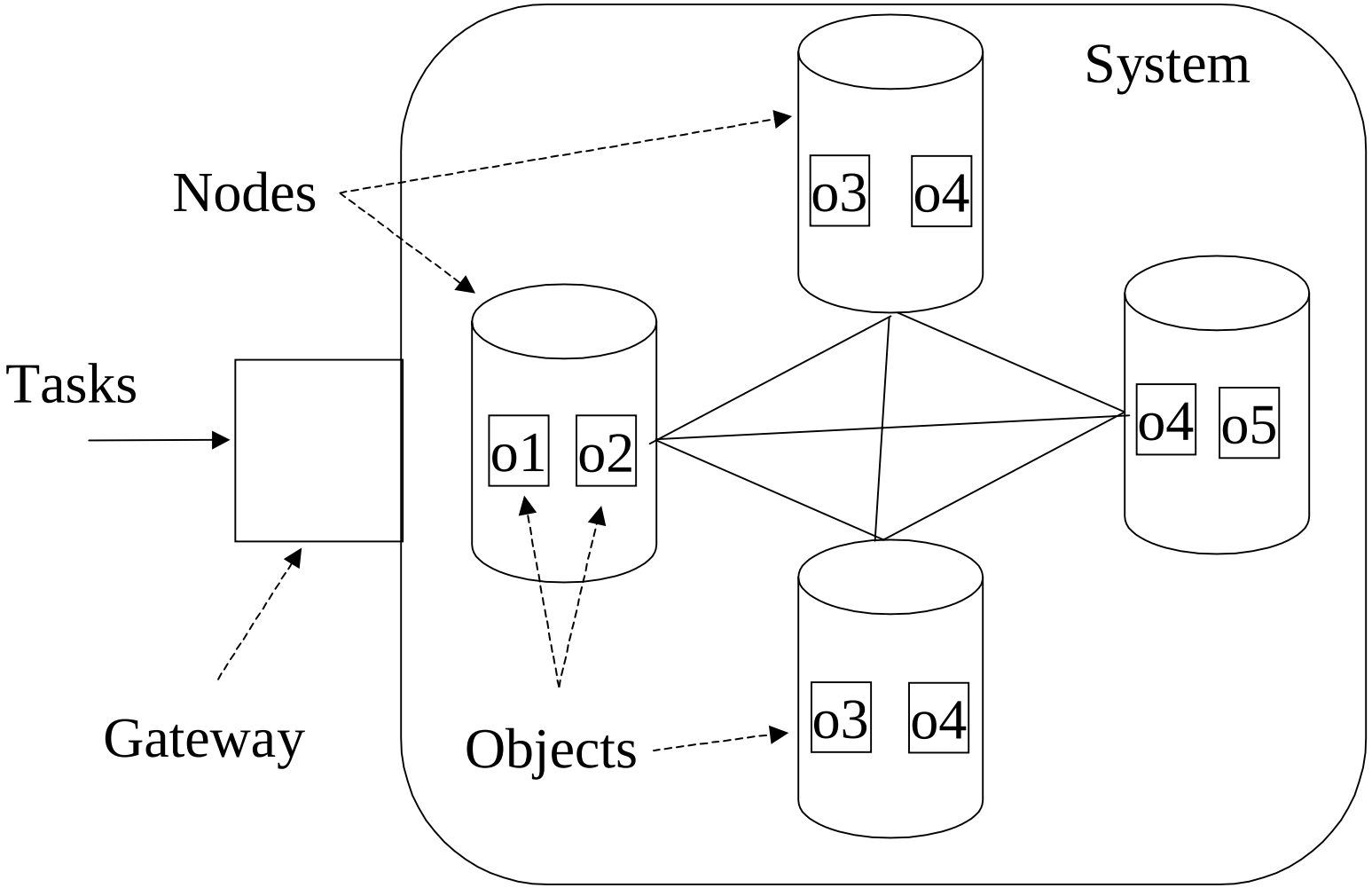
- Introduction
- System model
- Present real-time CORBA improvements
- System overload
- Overload protection
- Implementation
- Conclusions

Introduction

- Real-time distributed object oriented systems used for a multitude of applications, such as:
  - web servers
  - telecommunication service logic
  - computing farms
- Applications typically large, highly dynamic and with high stability requirements

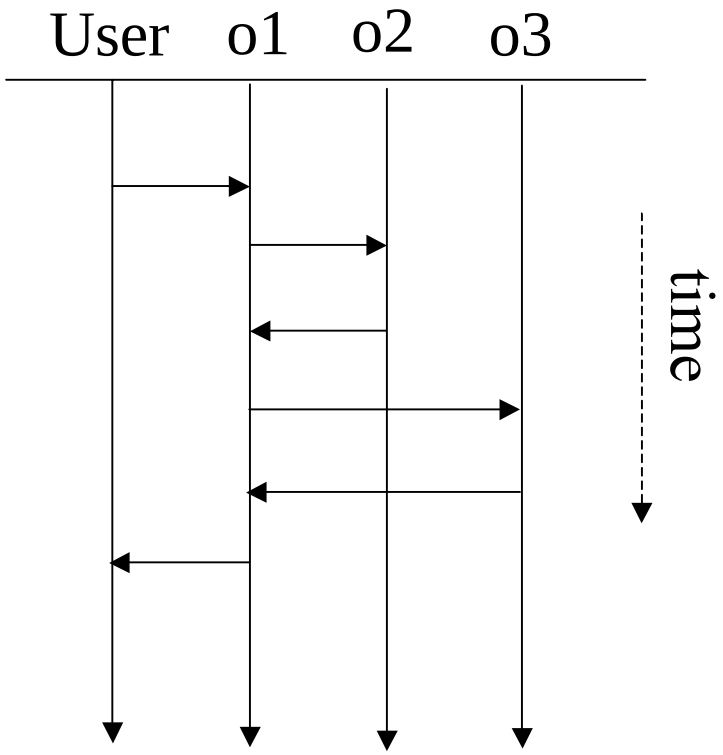
System model

- System
- Nodes
- Gateway
- Tasks
- Objects



System model, cont.

- A task (activity) is a sequence of method invocations.
- Task parameters:
  - deadline
  - priority
  - importance
- Using profiling, the expected execution time for a task can be predicted
- Tasks initiated by threads or users



System model, cont.

- Static request patterns - predictable arrivals, known processing duration and deadlines. Optimal scheduling can be achieved and overload be avoided by correct dimensioning.
- Dynamic request patterns - no *a-priori* scheduling information available. Overload is possible.

Existing real-time CORBA performance improvements

- Distribution
- Replication
- Load balancing

⇒ Improves capacity of a system from a finite value *C* to a (larger) value *C'*.

⇓ What happens when workload generated by arriving tasks approach or even exceed *C*?

System overload

- Tasks arriving to the system generate workload on processors
- load = arrival_rate * service_time
- Loads increase => delays increase => performance degradation =>harder/impossible to meet deadlines
- Delay typically proportional to 1/(1-load)
- Long term solution: increase capacity of system (buy more processing capacity, improve load balancing).
- Short term solution: block some arriving tasks

Overload protection

- Objective: maximise *useful* work done by system
- By *useful* work is meant work that is done to tasks that meet their deadlines
- A task that is finished, but fails to meet it deadline, means that resources have been *wasted*.
- Thus, if a task that is likely to miss it´s deadline is rejected before it starts execution, the resources that would have been used for the task can be used for other tasks

Overload protection, cont.

- Node level overload protection:
  - Can reject task during execution
  - Node has better load information than gateway/scheduler
  - May be non-trivial to shed a task that has begun execution, and work is wasted
- System level overload protection:
  - Reject task at gateway before execution begins
  - No/limited wasted resources
  - May block task that could have been schedulable

Overload protection, cont.

- Simple
- Low overhead
- Little *a-priori* knowledge about system parameters
- Adaptive
- Stable
- Robust

Detecting overload

- Existing load monitors can used
- Measuring:
  - resource load (processor, network, memory,etc)
  - arrival rate
  - deadline miss rate
  - queue lengths
  - admitted predicted load

Protection mechanisms

- Window based - allow only a limited number of tasks to be executed at any time
- Rate based - keep the arrival rate (tasks per time unit) of new tasks at a level where no overload situations occur
  - percent blocking
  - call gapping
  - token rate

Measuring success

- Throughput
- Deadline miss-rate
- Response time

- Relating throughput and deadline miss-rate to importance of tasks

Implementation

- General:
  - Shed tasks according to some policy
  - Resource release after task has been shedded
- Application specific:
  - Monitoring parameters to support shedding rules (user sessions, call completion rates etc.)
  - Shedding policy
- Fault tolerance

Conclusions

- To support QoS for systems with dynamic request patterns overload protection mechanisms are needed to protect the system from overload
- Implementation cannot be entirely application independent