On the Integration of Real-Time and Fault-Tolerance in P2P Middleware

Rolando Martins

Scientific Advisors:
Luís Lopes, Faculty of Science - University of Porto
Fernando Silva, Faculty of Science - University of Porto
Target Systems

- EFACEC’s Oporto light-train deployment
  - 5 lines, 70 stations, trains multiplexed over 5 lines
    - 70+ computational nodes (peers), 200+ sensors, arbitrary topology
  - Traffic comprised of normal operations, critical events, alarms
  - Tight timing, e.g., 2s for end-to-end response time

- Deployments across cities/regions can be overwhelmingly large

- What is needed to support such systems?
  - Peer-to-peer (P2P) infrastructure that mirrors physical deployment
  - Combined real-time and fault-tolerance guarantees
  - Hierarchical abstraction (cells) to scale to large deployments
In Search of a Solution

Video Streaming

Distributed storage

Pastry

CORBA RT FT

RT

RT+FT

RT+P2P

RT+FT+P2P

P2P

FT

FT+P2P

CORBA FT

Stheno

DDS
Research Challenges and Opportunities

▶ Challenges
▶ FT mechanisms consume additional resources
▶ FT mechanisms add overhead (e.g., additional latency)
▶ Different traffic types have different soft-RT requirements
▶ Different traffic types may require different FT configurations
▶ RT requirements must continue to be met even under faults

▶ Opportunities
▶ P2P infrastructures have network-aware resilience
▶ COTS operating systems have priority-based scheduling, multi-threading and resource-reservation mechanisms
▶ Proven FT configuration options exist (replication styles)
Research Question

Can we opportunistically leverage and integrate these proven strategies to simultaneously support soft-RT and FT to meet the needs of our target systems?
Scope

▶ Non-Goals
  ▶ Handling value faults and byzantine faults
  ▶ Formal specification and verification of the system
  ▶ Support for hard real-time
  ▶ Fully optimized implementation
  ▶ Testing in production (not yet)

▶ Assumptions
  ▶ Fault model: crash of a peer, message loss
  ▶ Resource-reservation mechanisms are always available
Stheno: System Architecture

Applications and Services

Core

P2P Layer and FT Configuration

Support Framework

Operating System

Stheno
Stheno: Operating-System Interface

- **Problem:** Control and monitor resource usage from userspace
- **Solution:**
  - Leverage threads, priorities, /proc
  - Resource reservation
  - CPU partitioning
- **Example:**
  - Highly critical surveillance feed has reserved amount of CPU for processing
Stheno: Support Framework

- Problem: Tasks have different RT requirements
- Solution:
  - Leverage threading policies
  - QoS Daemon
- Example:
  - Thread-per-Connection used for critical events in our target system to achieve low latency
Stheno: P2P Overlay and FT Configuration

- **Problem**: Tailor choice of P2P overlay and FT configuration to application needs
- **Solution**:
  - High-level API to support alternative overlays, e.g., P3, Pastry
  - Leverage proven replication styles, e.g., active, semi-active, passive
  - Configure replication properties, e.g., number and placement of replicas
  - Support service discovery
- **Example**:
  - P3 mirrors regional hierarchy of target system
  - Active replication for critical tasks needing instantaneous fail-over
Stheno: Core

- Problem: Manage services with different RT and FT requirements
- Solution:
  - QoS daemon proxy
  - Service repository
  - Creator and coordinator of service instances and clients
  - Delegation of service discovery to the P2P layer
- Example:
  - Service repository could include RPC, streaming service, etc
Stheno: Application and Services

- Problem: Expose system functionalities and configuration options to the user
- Solution:
  - High-level APIs for querying and configuring different layers
- Example:
  - Create a video streaming service from light-train station and set the frame rate and replication style
Stheno: Interaction between Layers

Peer

Apps
User
Service

Runtime Interface

Core
QoS Controller
QoS Client

Core Interface

QoS Daemon

Overlay Interface

Overlay
Proof-of-Concept Prototype

- First prototype implementation in Java had more than 50k SLOC
- Current (unoptimized) prototype implementation in C/C++ with more than 60k SLOC
- P^3 overlay plugin implementation
- CPU resource reservation
- Thread priorities: three classes corresponding to low, medium and high criticality
- Threading policies: Thread-per-Connection, Thread-per-Request, Leader-Followers
- Semi-active replication style
Empirical Evaluation

- Goals: To quantify
  - Overhead of fault-tolerance mechanisms with/without faults
  - Impact of background workload and faults on end-to-end latency
- Metrics:
  - End-to-end latency, jitter, recovery time
- Experimental setup:
  - 20 nodes, each quad-core AMD Phenom with 4GB RAM
  - 100 Mbit/s switch
- Experimental procedure:
  - Used a P³-based overlay, semi-active replication
  - Run of 1000 invocations
  - Fault-injection mid-way through each run
End-to-End Latency Results

- 4 replicas, without resource reservation: max time of 1s/invocation
- 4 replicas, With resource reservation: max time of 1ms/invocation

(a) Without resource reservation.

(b) With resource reservation.

- Stheno’s RT+FT support meets and exceeds target system requirements (2s end-to-end response time, even under a fault)
### Fail-over Latency Results

- Without resource reservation: max fail-over time of 3s
- With resource reservation: max fail-over time of 30ms

![Graph](image1)

(a) Without resource reservation.

![Graph](image2)

(b) With resource reservation.

- **Stheno’s RT+FT provides low fail-over latency that meets target system requirements**
Thesis Contributions

- Stheno, an RT+FT+P2P middleware
  - Motivated by the timing, reliability and physical deployment characteristics of our target systems
- To the best of our knowledge, Stheno is the first system that
  - Supports traffic types with different soft-RT requirements
  - Supports different FT configurations
  - Supports configurability at multiple levels: P2P, RT and FT
  - Continues to meet RT requirements even under faults
- Implementation of a proof-of-concept prototype
- Empirical evaluation demonstrates that
  - Stheno meets and exceeds target system requirements for end-to-end latency and fail-over latency
Thank You

Stheno, in Greek mythology, was the eldest of the three Gorgons. She was known to be the most independent and ferocious, having killed more men than both of her sisters combined. (source Wikipedia)

In many ways, Stheno represents the complexity of the problem that we set out to solve.
Publications


Replication Groups Over Group Communications

(a) Semi-active
(b) Passive
Resource Reservation Daemon

MEM

RAM

QoS Deamon

CPU

Core 0 - Best-Effort

Core \{1,2,3\} - RT

NET

ETH0

CPU

Core 0 - Best-Effort

Core \{1,2,3\} - RT

ETH1

IO

Disk 1
Multicore: Examples of CPU Partitioning.

<table>
<thead>
<tr>
<th>Core 0 OS</th>
<th>Cores {2,3} BE &amp; RT</th>
<th>Core 5 Isolated RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 1 Isolated RT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) Quad-core partitioning.

<table>
<thead>
<tr>
<th>Cores {0,1} OS</th>
<th>Cores {2,3,4} BE &amp; RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores {6,7} Isolated RT</td>
<td></td>
</tr>
</tbody>
</table>

(b) Six-core partitioning.

<table>
<thead>
<tr>
<th>Cores {0,1} OS</th>
<th>Cores {2,3,4,5} BE &amp; RT</th>
<th>Cores {6,7} Isolated RT</th>
</tr>
</thead>
</table>

(c) Eight-core partitioning.

- **Core Os**: Threads belonging to the operating system
- **BE**: Threads served by SCHED_OTHER scheduling policy
- **RT**: Threads served by SCHED_{FIFO,RR} scheduling policies
- **Isolated RT**: Isolated RT threads that are isolated from all other threads present in the system
RT Support: Object-to-Object interactions.

(a) Direct calling with different partitions.

(b) Direct calling within the same partition.

(c) Deferred calling with different partitions.
Threading Strategies

Leader-Followers

Thread-Pool

Thread-Per-Request
Minimizing Priority Inversion Through Traffic Demultiplexing
Minimizing Priority Inversion Through Traffic Demultiplexing
Putting It All Together

RPC service

1. invocation
2. reply

Core 0
- Best-Effort

Core (1)
- Isolated RT

Core (2,3)
- RT

QoS Deamon

CPU

RPC client

server

server adapter

socket

Core 1
- Isolated RT

Core 2
- BE & RT

Rolando Martins

On the Integration of RT & FT in P2P

May 7, 2012

28
Putting It All Together (Continuation)
Execution Context/Execution Model (ECEM) Design Pattern

Execution Model 0

Execution Model 1

Threading Strategy

Op. sync?
no, async
no, error

Last EM allows join?
true
false
Queueable?
yes, queue()

Environment stack

{EM1, event1, timeout1}
{EM0, event0, timeout0}

Future

wait()
Comparison with other Middlewares (RPC)

Our approach enable us to provide a 200us latency even in the presence of a 95% CPU workload
Related Work

1 - Decentralized scalability:

2 - Modular FT:

3 - Resource reservation + CPU partitioning:

4 - Real-time support: