Flexible Protocol Stack Framework : Design, Validation and Performance

Tim Farnham[1], Thorsten Schöler[2]

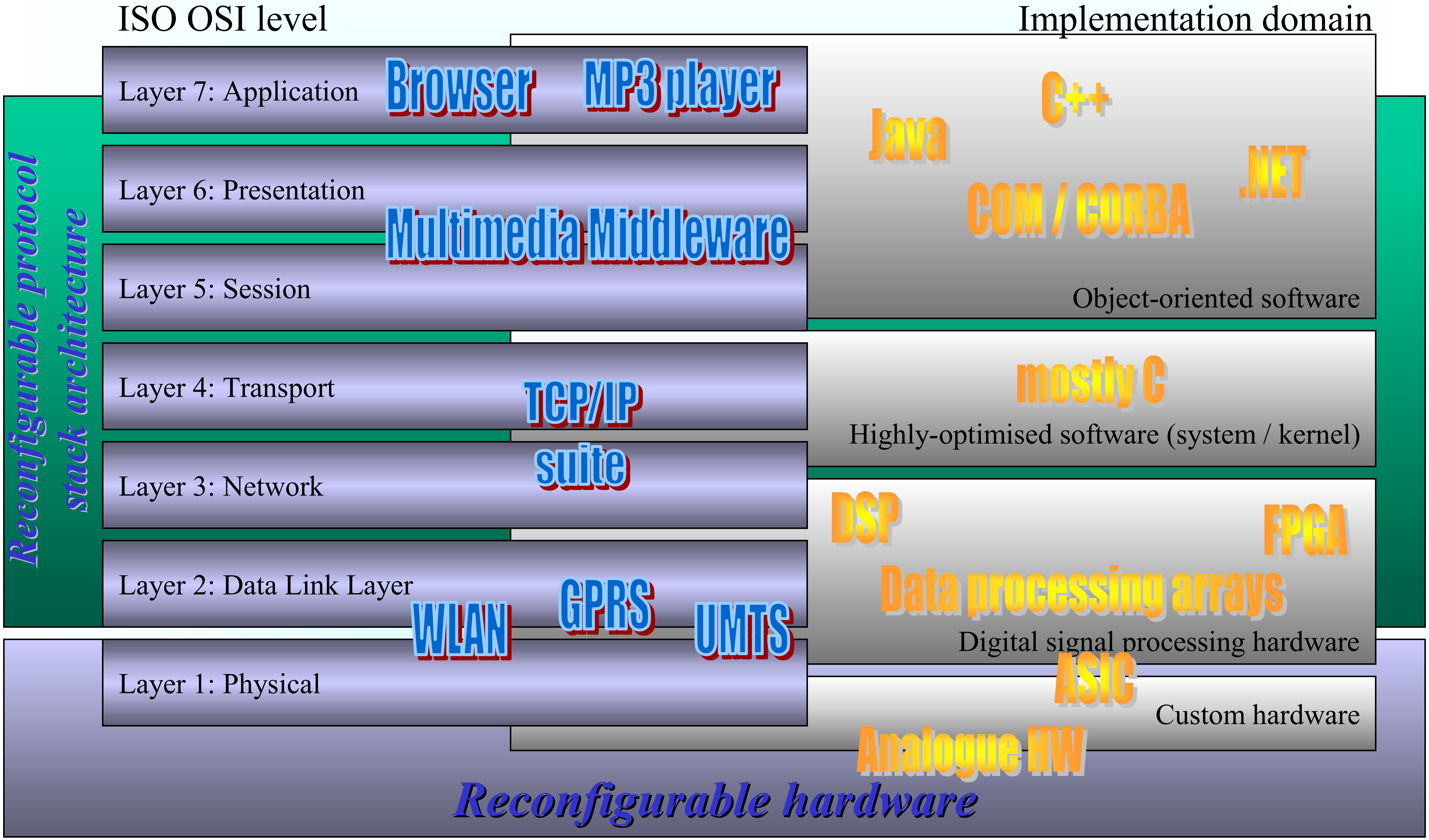
SDR Forum Technical Conference November 2003

1 Toshiba Research Europe Ltd
2 Siemens AG

Contents

- Introduction
- Terminal architecture
- Flexible protocol stack framework
  - Design
  - Validation
  - Performance
- Conclusions

Terminal architecture



ISO OSI level

Implementation domain

Reconfigurable protocol stack architecture

Layer 7: Application

Layer 6: Presentation

Layer 5: Session

Layer 4: Transport

Layer 3: Network

Layer 2: Data Link Layer

Layer 1: Physical

Browser

MP3 player

Multimedia Middleware

TCP/IP suite

WLAN

GPRS

UMTS

Java

C++

.NET

COM / CORBA

Object-oriented software

mostly C

Highly-optimised software (system / kernel)

DSP

FPGA

Data processing arrays

Digital signal processing hardware

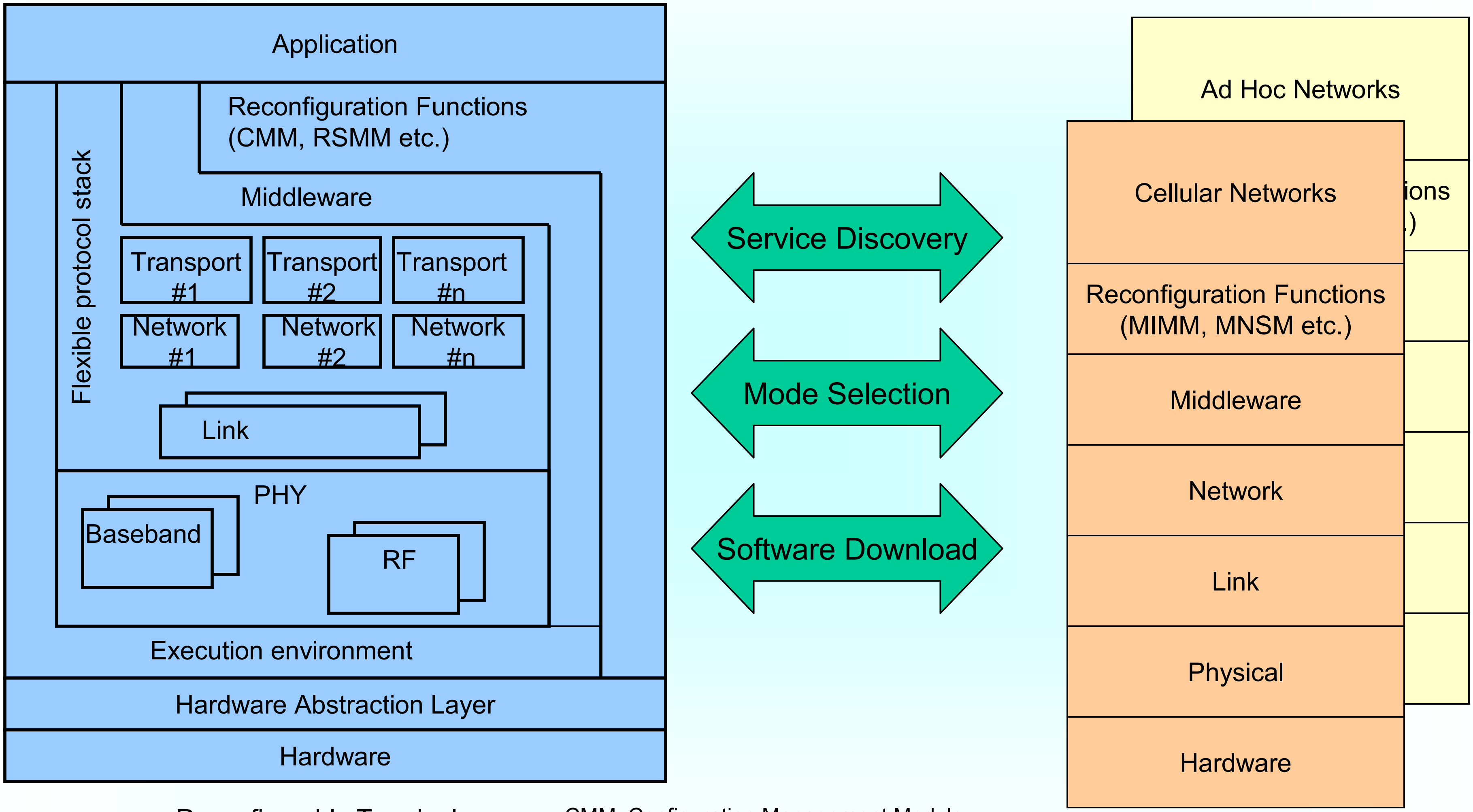
ASIC

Custom hardware

Analogue HW

Reconfigurable hardware

Network Centric Support for Reconfiguration



Application

Reconfiguration Functions
(CMM, RSMM etc.)

Middleware

Flexible protocol stack

Transport #1
Transport #2
Transport #n

Network #1
Network #2
Network #n

Link

PHY

Baseband

RF

Execution environment

Hardware Abstraction Layer

Hardware

Service Discovery

Mode Selection

Software Download

Ad Hoc Networks

ions

.)

Cellular Networks

Reconfiguration Functions
(MIMM, MNSM etc.)

Middleware

Network

Link

Physical

Hardware

Reconfigurable Terminal

CMM: Configuration Management Module
RSMM: Resource System Management Module
MIMM: Mode Identification and Mode Monitoring
MNSM: Mode Negotiation and Switching Module

Requirements and Solution Features for Flexible Protocol Stacks

- Platform Independence
  - Multiple CPU / execution environment and language support
- High reliability / availability
  - Fallback states, etc.
  - Validation of Stack Configuration and Implementation
- Secure operation
  - Mechanisms to prevent unauthorised interception, manipulation
- Multi-vendor sourcing
  - Manufacturer, operator, service provider and third party
  - Open interfaces
- Dynamic optimisation
  - Depending on resource availability, execution environment and service requirements
  - Mechanisms for active protocol stack reconfiguration
- Customisation and enhancement
  - Mechanisms to allow incremental upgrading

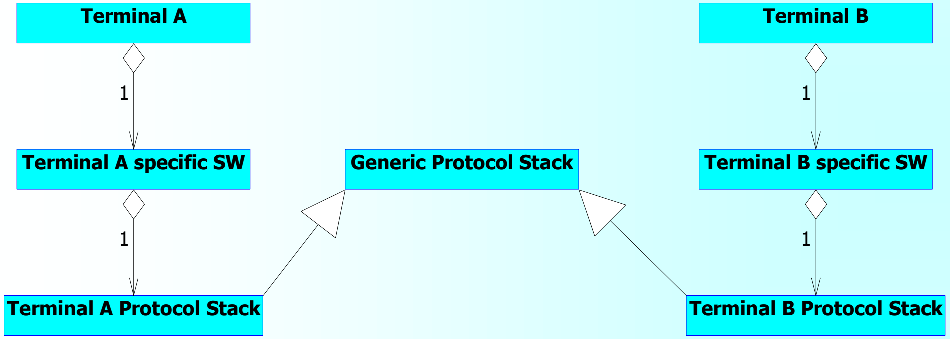
Contents

- Introduction
- Terminal architecture
- Flexible protocol stack framework
  - Design
  - Validation
  - Performance

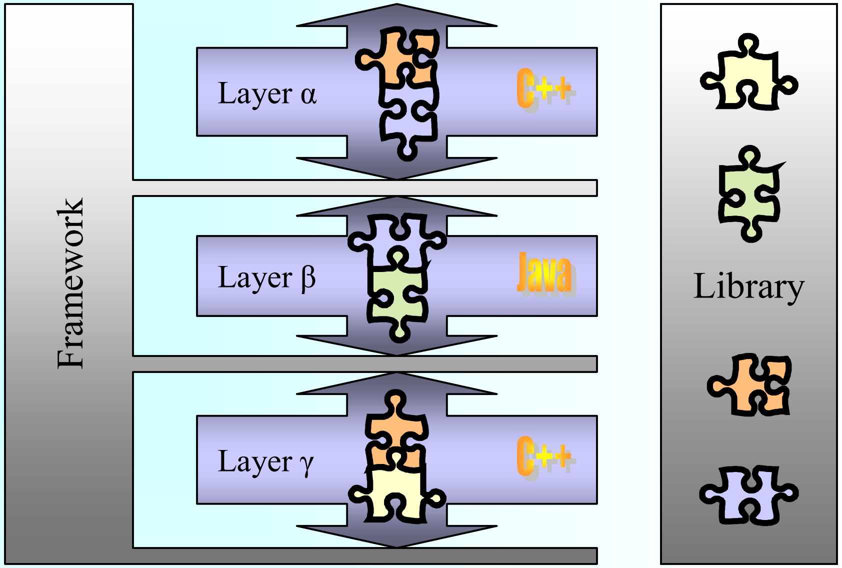
- Conclusions

State of the art in modular protocol stacks

Customisable protocol stacks (design time)



Composable protocol stacks (run time)



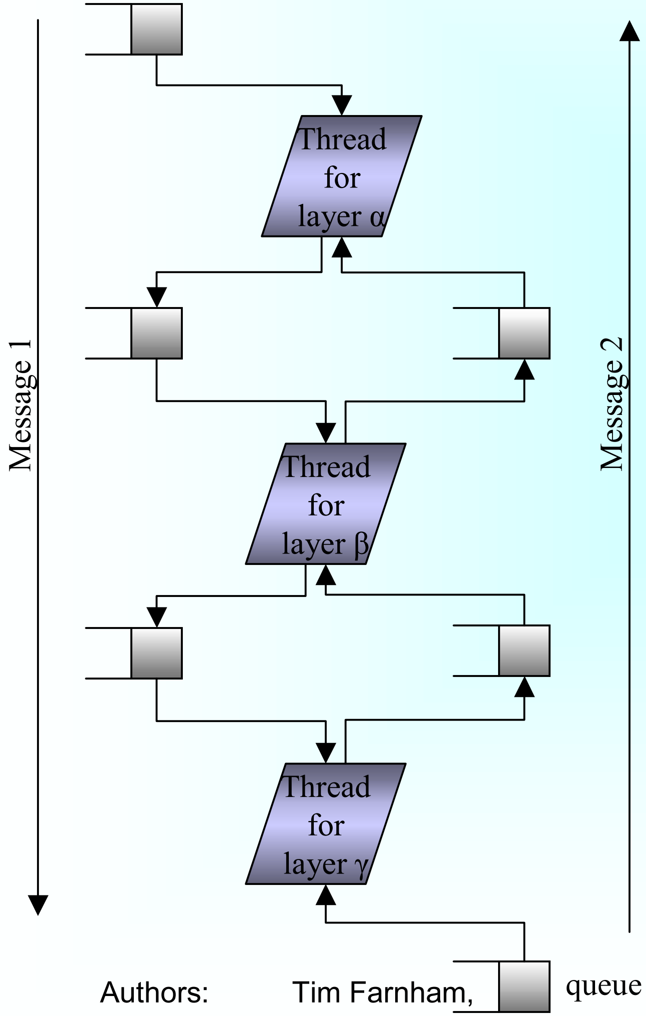
X-Kernel – Composable (at compile time) framework with configurable virtual protocol layers

OPtIMA – Java based, composable and (run-time) customisable framework with configurable active programming interfaces

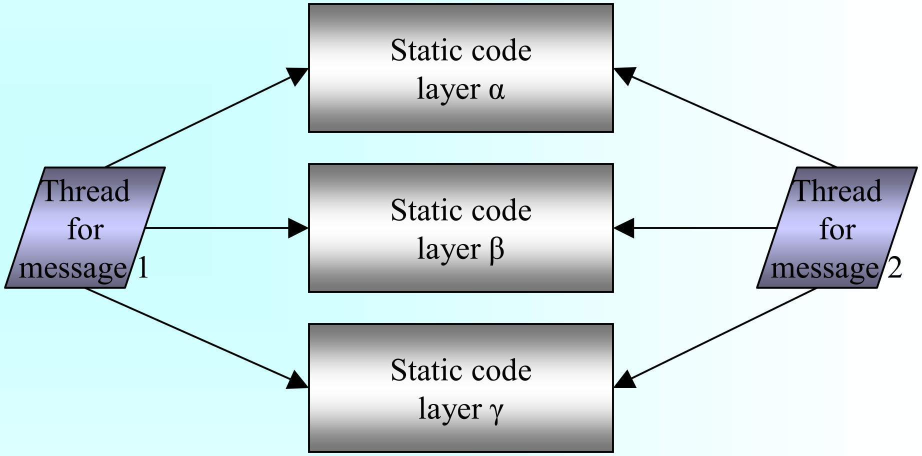
DIMMA – C++ based, customisable framework which is derived from X-kernel framework

Protocol stack computation models

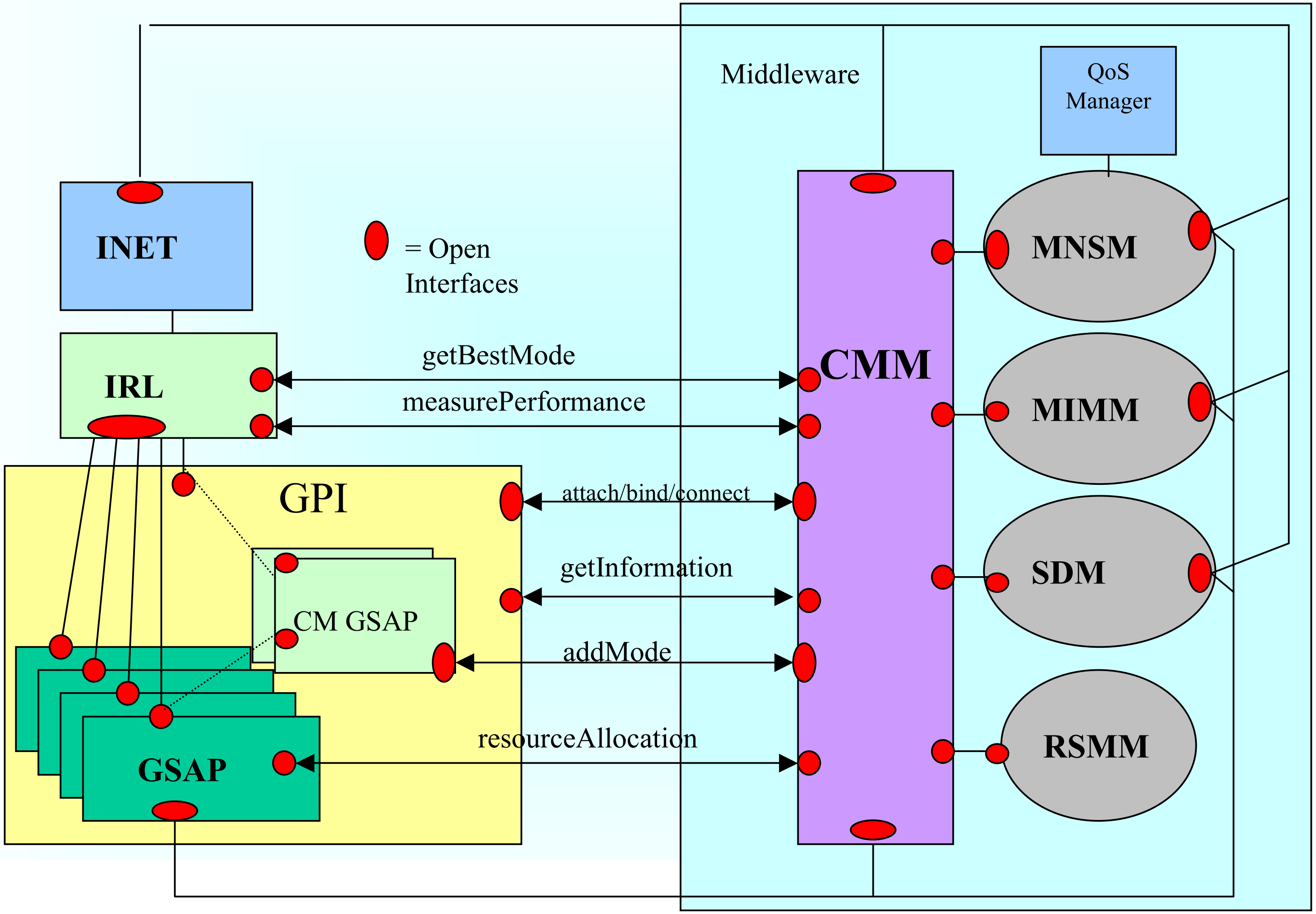
Process per protocol



Process per message



Proposed Flexible Protocol Stack Framework

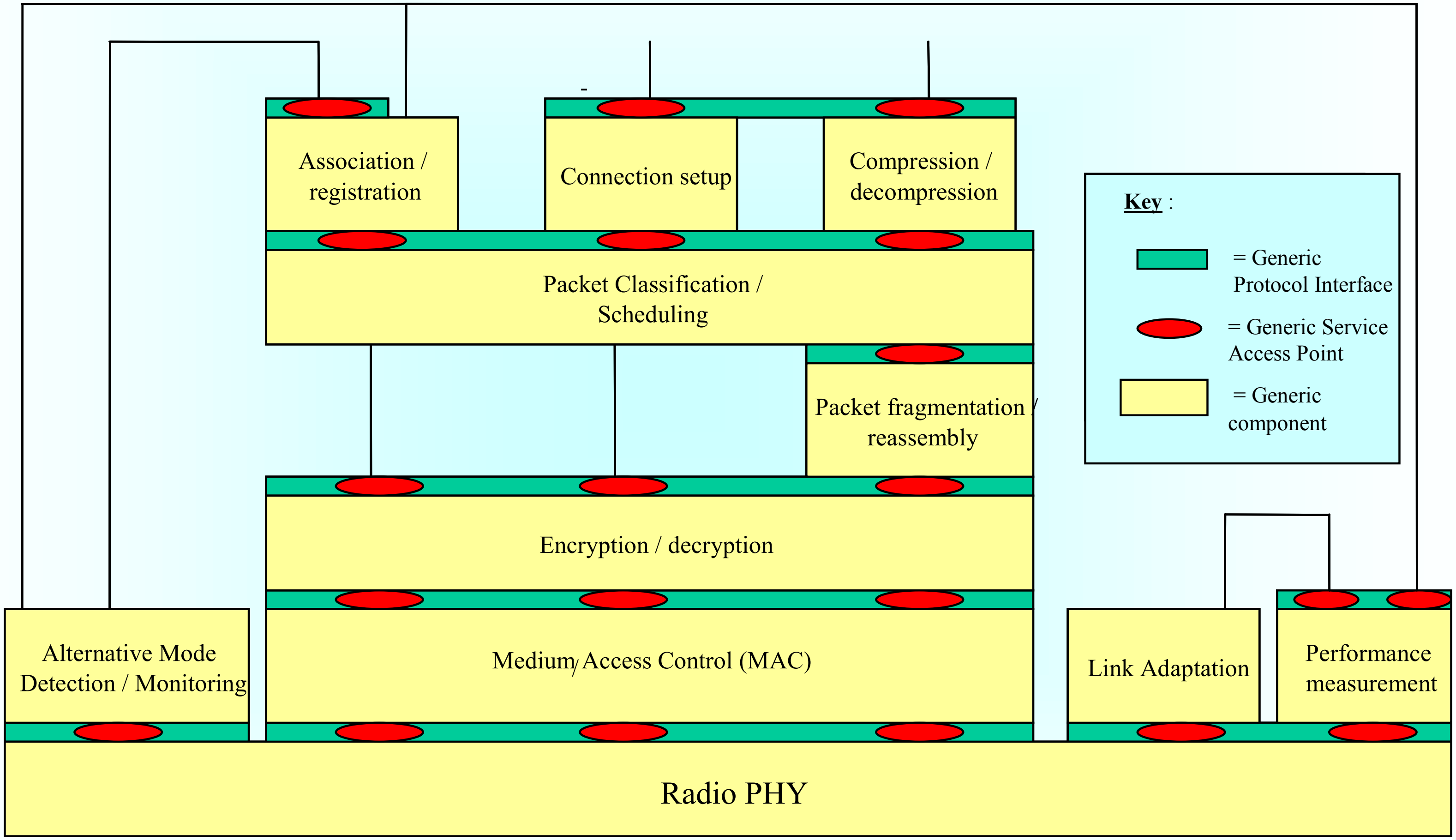


Note : MIMM = Mode Identification and Monitoring Module (or Mode Identification and Monitoring), MNSM = Mode Negotiation and Switching Module (or Mode Switching Module), SDM = Software Download Module, RSMM = Resource System Management Module (or Resource Management System), GPI = Generic Protocol Interface, GSAP = Generic Service Access Point, CM-GSAP = Connection Management GSAP, CMM = Configuration Management Module, INET = Internet TCP/IP Stack, IRL = Intelligent Routing Layer.

Framework Key Features

- Generic Protocol Interface (GPI)
  - Language and platform independent
  - Radio access technology independent
- Generic Service Access Points (GSAPs)
  - Dynamically bound and rebound
  - Secure interaction between layer instances
  - Extensible message data format
  - Execution environment neutral
- Intelligent Routing Layer (IRL)
  - Supporting dynamic mode selection

Generic Protocol Stack Example



Association / registration

Connection setup

Compression / decompression

Packet Classification / Scheduling

Packet fragmentation / reassembly

Encryption / decryption

Alternative Mode Detection / Monitoring

Medium Access Control (MAC)

Link Adaptation

Performance measurement

Radio PHY

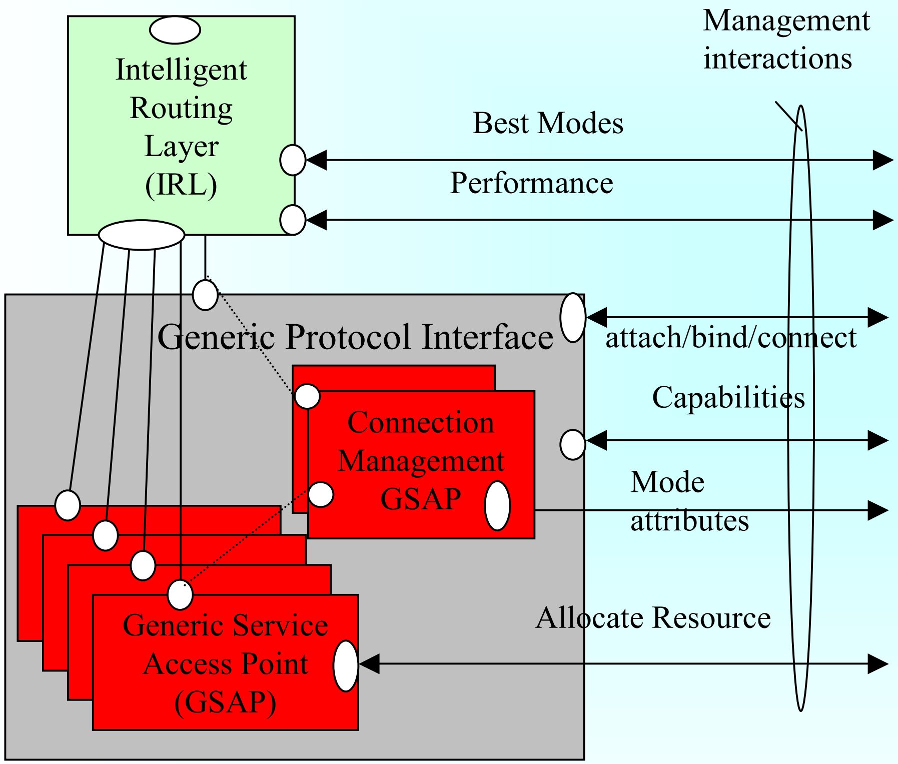
Key :

🟩 = Generic Protocol Interface

🔴 = Generic Service Access Point

🟨 = Generic component

Reconfiguration Management Interactions



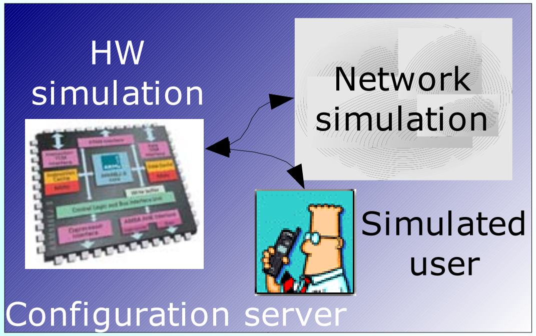
- Open interfaces allow reconfiguration of protocol stack to exploit the capabilities of the platform execution environments and customisation and enhancement options within protocol software.

Contents

- Introduction
- Terminal architecture
- Flexible protocol stack framework
  - Design
  - Validation
  - Performance
- Conclusions

Protocol stack validation process

- Network-based validation
  - Off-line validation
    - Virtual prototyping
      - HW, SW, Network simulation
      - Simulation of actual stack implementation
      - Assertions for validation of software correctness

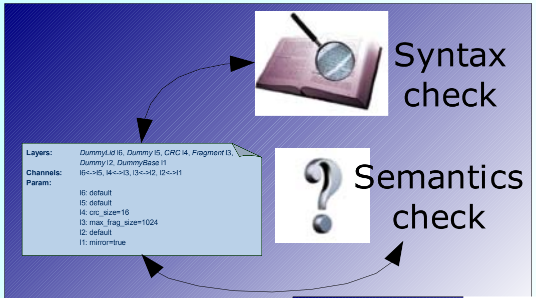

HW simulation

Network simulation

Simulated user

Configuration server

- Terminal-based validation
  - On-line validation
    - Check of protocol stack configuration
      - Syntax and semantics

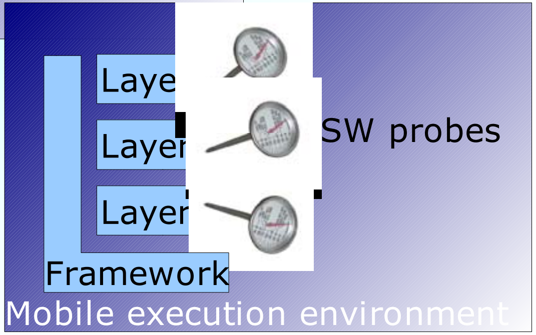

Syntax check

Semantics check

```
Layers:     DummyLid l6, Dummy l5, CRC l4, Fragment l3,
            Dummy l2, DummyBase l1
Channels:   l6<->l5, l4<->l3, l3<->l2, l2<->l1
Param:
            l6: default
            l5: default
            l4: crc_size=16
            l3: max_frag_size=1024
            l2: default
            l1: mirror=true
```

Mobile terminal

  - Run-time validation
    - In protected execution environment
    - Software probes in protocol stack software
      - Assertion-based


Laye

Layer

Layer

SW probes

Framework

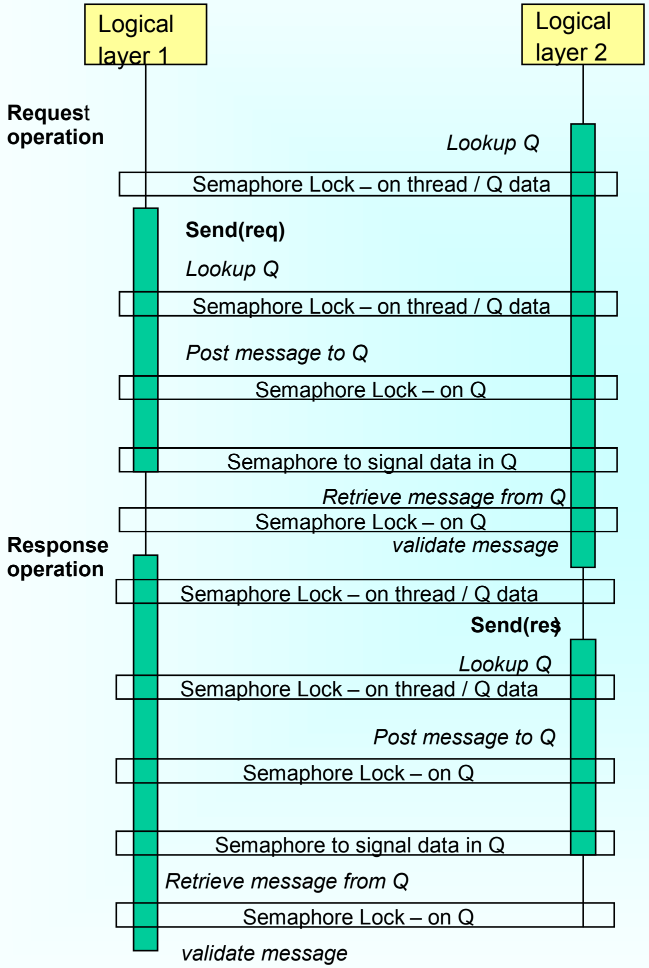
Mobile execution environment

Authors: Tim Farnham, Thorsten Schöler

Contents

- Introduction
- Terminal architecture
- Flexible protocol stack framework
  - Design
  - Validation
  - Performance
- Conclusions

Secure Asynchronous Messaging



Logical layer 1

Logical layer 2

Request operation

Lookup Q

Semaphore Lock – on thread / Q data

Send(req)

Lookup Q

Semaphore Lock – on thread / Q data

Post message to Q

Semaphore Lock – on Q

Semaphore to signal data in Q

Retrieve message from Q

Semaphore Lock – on Q

validate message

Response operation

Semaphore Lock – on thread / Q data

Send(res

Lookup Q

Semaphore Lock – on thread / Q data

Post message to Q

Semaphore Lock – on Q

Semaphore to signal data in Q

Retrieve message from Q

Semaphore Lock – on Q

validate message

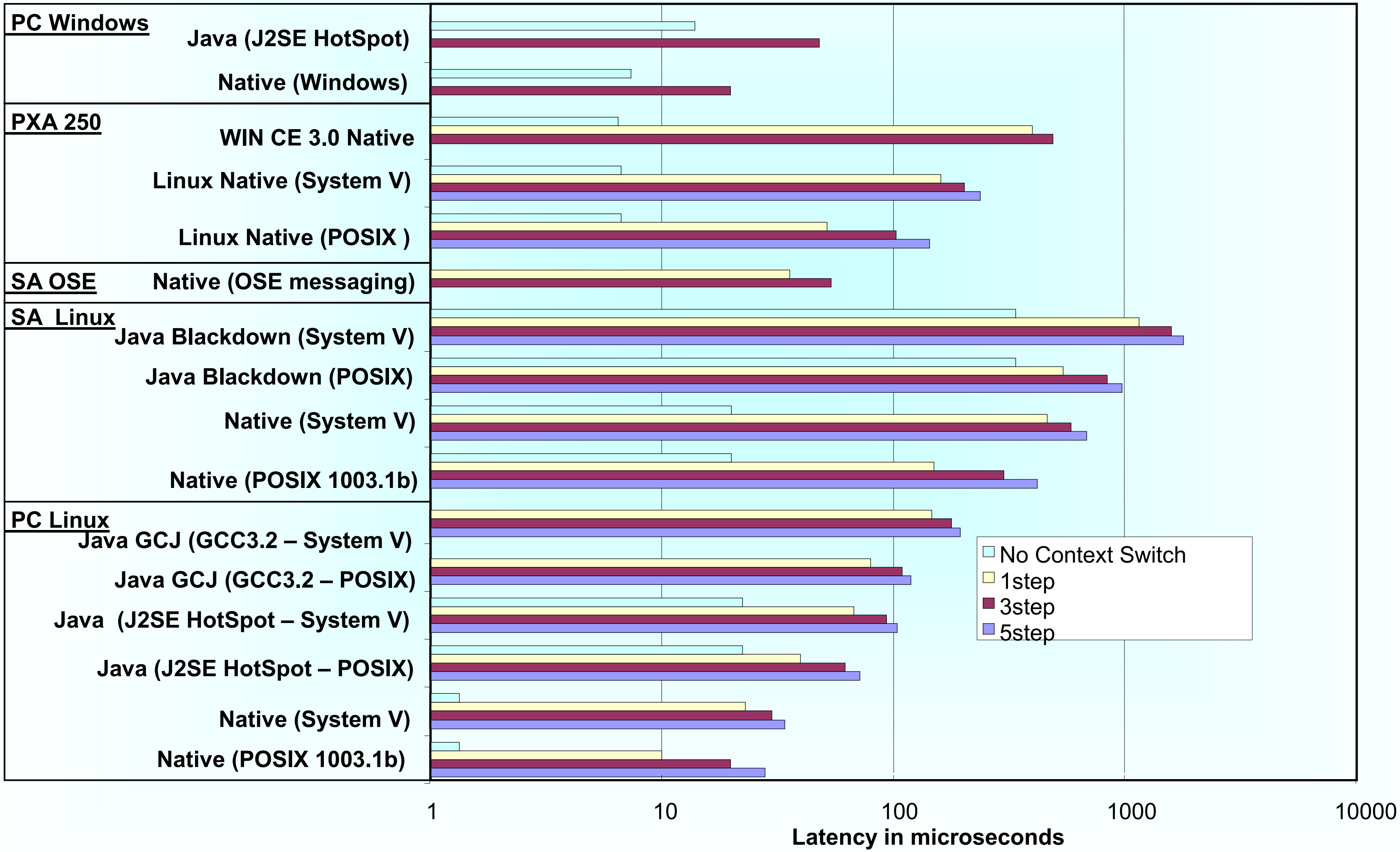
- Execution environments provide protection between logical protocol layer instances
- Interaction between instances authorised to prevent rogue behaviour
- Different steps to accommodate different execution environments and computational models

Benchmarking Platforms

Name	**Type**
PC Linux	1GHz Intel P3 PC with Linux 2.5.54
PC Windows	1GHz Intel P3 PC with Windows 2000
SA Linux	Intel StrongARM 200MHz with Linux 2.4.18
SA OSE	Intel StrongARM 200Mz with Enea OSE Delta RTOS
PXA Linux	Intel PXA 250 400MHz with Linux 2.4.18
PXA WinCE	Intel PXA 250 400MHz with Pocket PC 2002

- Three hardware platforms and four different operating systems
- Java Virtual Machines also considered on Linux operating and Windows systems

Benchmark Results



Results Summary

- Operating different logical layer instances in separate execution environments is attractive to exploit heterogeneous execution environments
  - Native threads and processes
  - JVM threads and processes
- The overhead in performing context switching must be considered when partitioning the protocol stack between execution environments
- Thread context switching and asynchronous messaging can actually be less computationally intensive than Java native calls using Java Native Interface (JNI)

Results Summary - continued

- Computationally intensive operations such as CRC calculation within a JVM protocol module can present much higher latency than thread or process messaging and native processing
- Memory requirements of Java implementation considerably higher than native implementation
- Performance variation across different platforms and computational models considerable
  - 1000 to 1 variation in benchmarks
- Context switching can be avoided if a single execution environment provides the necessary performance and security, but this will not generally be the case

Conclusions

- Flexible protocol stack framework based on open interfaces and generic service access points is an attractive approach
- Different execution environments and computational models are also attractive to provide best use of resources
- Validation can be most efficiently performed in a combined off-line, on-line and at run-time manner
- Performance results indicate that secure asynchronous messaging is a viable and lightweight solution for supporting the framework