Flexible Protocol Stack Framework: Design, Validation and Performance

Tim Farnham\textsuperscript{1}, Thorsten Schöler\textsuperscript{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

<table>
<thead>
<tr>
<th>ISO OSI level</th>
<th>Implementation domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer 7: Application</td>
<td>Browser, MP3 player, Java, C++</td>
</tr>
<tr>
<td>Layer 6: Presentation</td>
<td>Multimedia Middleware, COM / CORBA, .NET</td>
</tr>
<tr>
<td>Layer 5: Session</td>
<td>C++</td>
</tr>
<tr>
<td>Layer 4: Transport</td>
<td>Mostly C</td>
</tr>
<tr>
<td>Layer 3: Network</td>
<td>Highly-optimised software (system / kernel)</td>
</tr>
<tr>
<td>Layer 2: Data Link Layer</td>
<td>TCP/IP suite, FPGA</td>
</tr>
<tr>
<td>Layer 1: Physical</td>
<td>DSP, Data processing arrays, ASIC</td>
</tr>
</tbody>
</table>

### Reconfigurable protocol stack architecture

- **WLAN, GPRS, UMTS**
- **TCP/IP suite**
- **Multimedia Middleware**
- **Browser, MP3 player**

### Reconfigurable hardware

- **Digital signal processing hardware**
- **Custom hardware**

---

**Authors:** Tim Farnham, Thorsten Schöler  
**Date:** 30/09/2003  
**Page:** 3
Network Centric Support for Reconfiguration

Authors: Tim Farnham, Thorsten Schöler

Date: 30/09/2003
Page: 4
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

Authors: Tim Farnham, Thorsten Schöler
Date: 30/09/2003
Page: 5
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)**

- Terminal A specific SW
- Terminal B specific SW
- Terminal A Protocol Stack
- Terminal B Protocol Stack
- Generic Protocol Stack

**Composable protocol stacks (run time)**

- Layer α
  - C++
- Layer β
  - Java
- Layer γ
  - C++

**Frameworks**

- 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

Authors: Tim Farnham, Thorsten Schöler

Date: 30/09/2003

Page: 7
Protocol stack computation models

Process per protocol

- Thread for layer α
- Thread for layer β
- Thread for layer γ

Process per message

- Static code layer α
- Static code layer β
- Static code layer γ

Authors: Tim Farnham, Thorsten Schöler

Date: 30/09/2003
Page: 8
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

Key:
- = Generic Protocol Interface
- = Generic Service Access Point
- = Generic component

Authors: Tim Farnham, Thorsten Schöler
• 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
  - Terminal-based validation
    - On-line validation
      • Check of protocol stack configuration
        – Syntax and semantics
    - Run-time validation
      • In protected execution environment
      • Software probes in protocol stack software
        – Assertion-based

Authors: Tim Farnham, Thorsten Schöler
Contents

• Introduction
• Terminal architecture
• Flexible protocol stack framework
  – Design
  – Validation
  – Performance
• Conclusions
Secure Asynchronous Messaging

• 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC Linux</td>
<td>1GHz Intel P3 PC with Linux 2.5.54</td>
</tr>
<tr>
<td>PC Windows</td>
<td>1GHz Intel P3 PC with Windows 2000</td>
</tr>
<tr>
<td>SA Linux</td>
<td>Intel StrongARM 200MHz with Linux 2.4.18</td>
</tr>
<tr>
<td>SA OSE</td>
<td>Intel StrongARM 200Mz with Enea OSE Delta RTOS</td>
</tr>
<tr>
<td>PXA Linux</td>
<td>Intel PXA 250 400MHz with Linux 2.4.18</td>
</tr>
<tr>
<td>PXA WinCE</td>
<td>Intel PXA 250 400MHz with Pocket PC 2002</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Platform</th>
<th>Configuration</th>
<th>Latency (microseconds)</th>
<th>Context Switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC Windows</td>
<td>Java (J2SE HotSpot)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Native (Windows)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PXA 250</td>
<td>WIN CE 3.0 Native</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Linux Native (System V)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SA OSE</td>
<td>Native (OSE messaging)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SA Linux</td>
<td>Java Blackdown (System V)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Java Blackdown (POSIX)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Native (System V)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Native (POSIX 1003.1b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC Linux</td>
<td>Java GCJ (GCC3.2 – System V)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Java GCJ (GCC3.2 – POSIX)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Java (J2SE HotSpot – System V)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Java (J2SE HotSpot – POSIX)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Native (System V)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Native (POSIX 1003.1b)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Authors: Tim Farnham, Thorsten Schöler

Date: 30/09/2003
Page: 18
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