X[ML]-Rated Architectures

Eoin Woods
OT2002 – Oxford, UK
April 2002

INTERTRUST®
:: we pioneered
digital rights management™

Contents

Introductions

InterTrust Technologies

Mission: Develop and commercialize fundamental new technology for digital commerce and information exchange.

History: Founded in 1990, commercial software development started in 1995, FCS in 1998. Headquartered in Santa Clara, CA, USA.

Partners: 61 partners across music, video, publishing, and enterprise markets.

Introductions

InterTrust Technologies – DRM pioneers

• **STAR Labs –** The longest existing Research and Development organization dedicated to Digital Rights Management.

• **21 Patents –** Covering a broad range of intrinsic DRM concepts and technologies with 80 additional patents pending.

• **Acquisitions –** Obtained key complimentary technologies and service offerings through acquisitions of XAudio, PassEdge and PublishOne.

Introductions

Eoin Woods

- Software architect.
- Ex Bull, LBMS and Sybase.
- Currently “chief” architect in the systems research group at InterTrust.
- Main professional interests are software architecture and large scale, reliable, distributed systems.

Contents

Background – The Application Domain

Digital Rights Management

- What is digital rights management?

  *DRM is the **persistent protection** of digital information combined with **trusted governance** of the rules associated with its use.*

- Why digital rights management?

  Flexible business rules enable you to explore a variety of business models: rent, own, pay-per-view, memberships, subscription, etc…

  Persistent protection shields your digital assets from piracy.

Background - The Product

Rights|System 1.0 Features

- General purpose DRM platform.
  - Multiple content types (video, music, documents, ...)
  - Multiple business rules (subscription, r-to-o, purchase)
  - Multiple device types (PC, mobile, STB)
- Easy Integration into existing technology.
- Scalable architecture.
- Portable rights and content.

Background - The Product

Rights|System 1.0 Components

Rights|System™

Packagers
- *Standard Packager*
- *Stream Packager**
- *Format Plug-ins*
  - *PAF*
  - *PDF*
  - *MP4**
  - *MP4 Streaming**
  - *Custom formats**

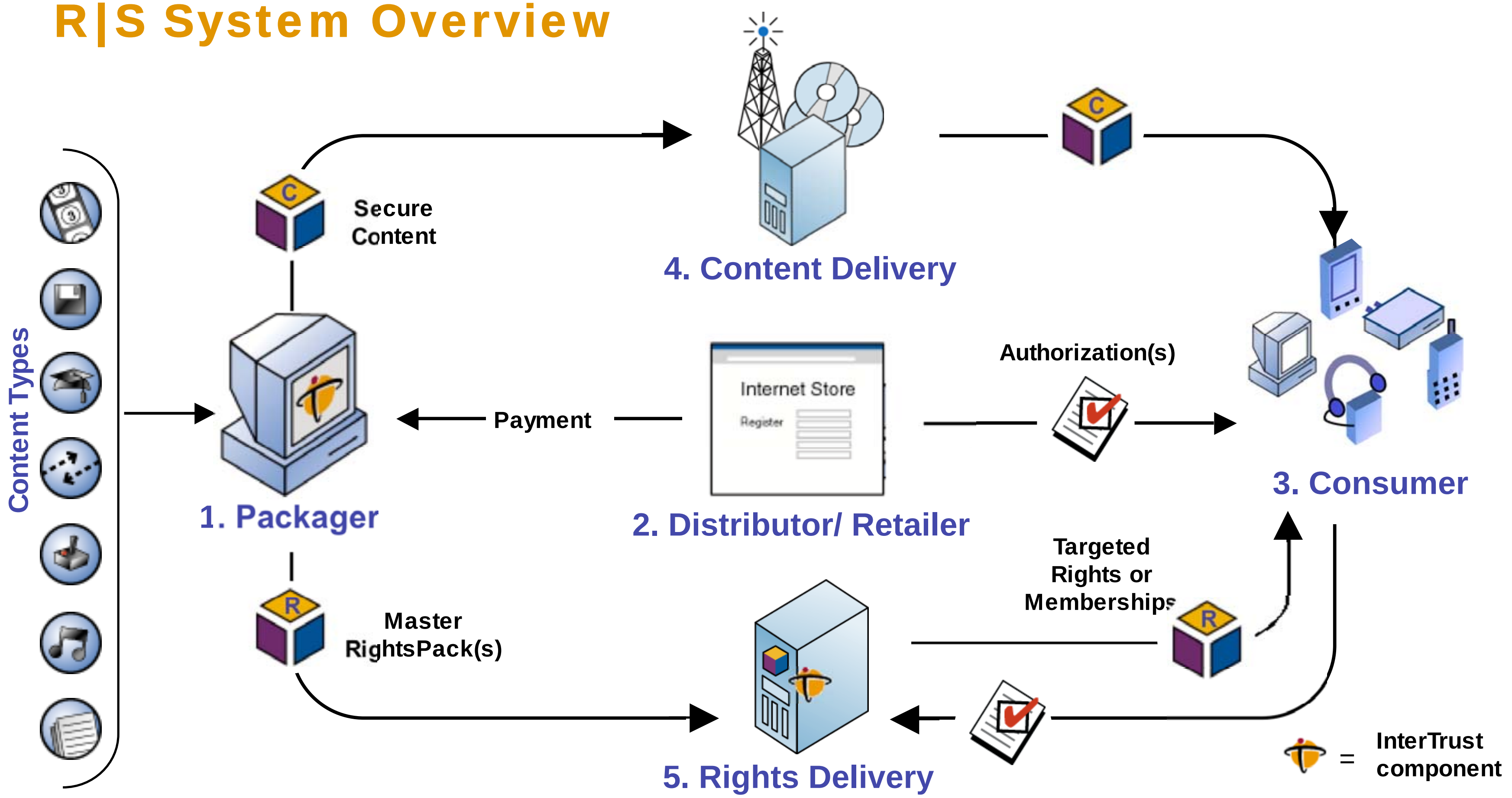
Servers
- *Authorization Generator*
- *Content Rights Server*
- *Membership Rights Server**
- *System Operations Server*

Clients
- *Software*
  - *Rights|Desktop*
  - *Rights|PD*
  - *Rights|Phone*
  - *Rights|TV*
- *Hardware*
  - *Rights|Chip*
  - *TrustChip*

Toolkits
- *Rights|Audio SDK*
- *Rights|Video SDK**
- *Rights|Desktop SDK*
- *Packager SDK**

Background - The Product

R|S System Overview



Background - Definitions

Software Architecture

- The structures of a system which comprise software components, relationships and externally visibly properties.

Product Line Architecture

- A software architecture that defines a set of common assets and how they are combined in order to create a family of closely related systems.

Contents

The Problem

Where did we start?

- First generation product (called “Commerce”) needed replaced.
- Needed a new “feel” to the server products.
- Needed to reduce proprietary nature of the server products.
- Needed to get to market in 6 months or less.
- Needed to gel a new distributed engineering team.

The Problem

What did we need?

- Better product quality (less defects).
- Reduced time to market.
- Much higher individual developer productivity.
- Less overall development effort.
- Good runtime scaling (to millions of transactions per day).
- Good runtime reliability (to < l failure per 10 x 10^6 txns).
- High commonality across all the servers (a “family”).

The Problem

Our solution ….

- Steal as much as possible!
  - *Technology*: 3rd party software.
  - *Experience*: tried and tested technology and approaches.
- Reuse as much as possible (of our own).
  - *Process*: product line architecture.

Contents

The Plan!

Overall approach …

- Create a product line architecture based on common services.
- Prototype each aspect of the product line architecture to find out what works before starting development.
- Create a technical framework to implement the product line architecture and capture the lessons learned.
- Implement all servers as instantiations of the product line architecture.

The Plan!

Design principles …

- Everything as simple as possible (a la XP).
- Look for an existing large scale use of everything.
- Use standard technology where ever possible.
- Don’t force servers to communicate synchronously.
- Achieve reliability and scaling by replication.
- Build in administrative functions from the beginning.
- Allow developers to focus on DRM not middleware.

The Plan!

Technology choices …

- Java with JNI to C++ security software.
- J2EE (but only JDBC and Servlets).
- XML for data encoding (particularly messages).
- Simple RDBMS persistence via Oracle.
- Use of 3rd party libraries.
  - Connection pooling used PoolMan then JDBCPool.
  - XML parsing used Apache's Xerces.
  - Logging used Apache's Log4J.

Contents

The Risks - Process

Primary design and process risks we perceived …

- New process causing disruption.
- Too much initial effort required to justify the product line approach (for our environment).
- New approaches distracting from the real problem (developing a compelling product).
- Acceptance of change and new approaches by key engineering staff.

The Risks - Technology

Primary technical risks we perceived …

- Java could be too slow or bad for servers.
- XML is large as a data encoding.
- Parsing/assembling XML could be a significant overhead.
- Accessing C++ from Java via JNI may have problems.
- Third party libraries may cause problems.
- Immature technology.
- Technology that was new to the organisation.

The Risks – Mitigating Them

How we assessed the process risks …

- Early explanation of our plans.
- Evangelization from the architecture group to the development groups.
- Inclusion of some development engineers in the product line definition.
- Obtaining senior management approval and backing.
- Ruthlessly managing schedule.

The Risks – Mitigating Them

How we assessed the technical risks …

- 4 weeks of prototyping.
  - Servlet containers.
  - Performance / load testing.
  - XML processing.
  - Calculation of likely XML message sizes.
  - Prototype C++/Java interface created.
- A product line architecture!

Contents

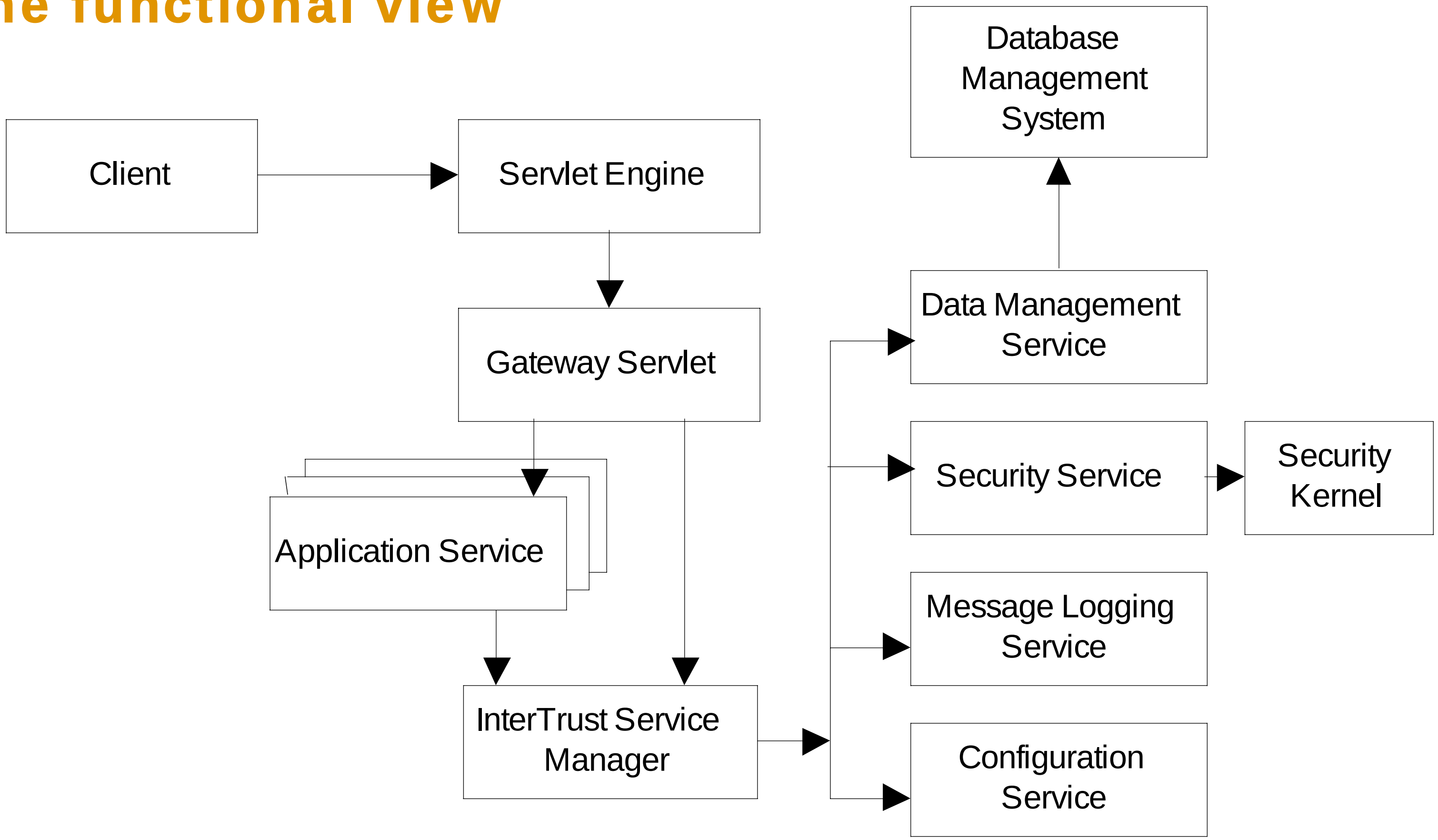
The Product Line Architecture

What did we expect of it?

- Better, faster, cheaper!
  - —Better product because people could focus on DRM, not middleware.
  - —Better product because underlying platform provided proven services.
  - —Faster development because less mistakes, reused knowledge and reuse of software.
  - —Cheaper due to less overall work being needed because of sharing and reduced rework.

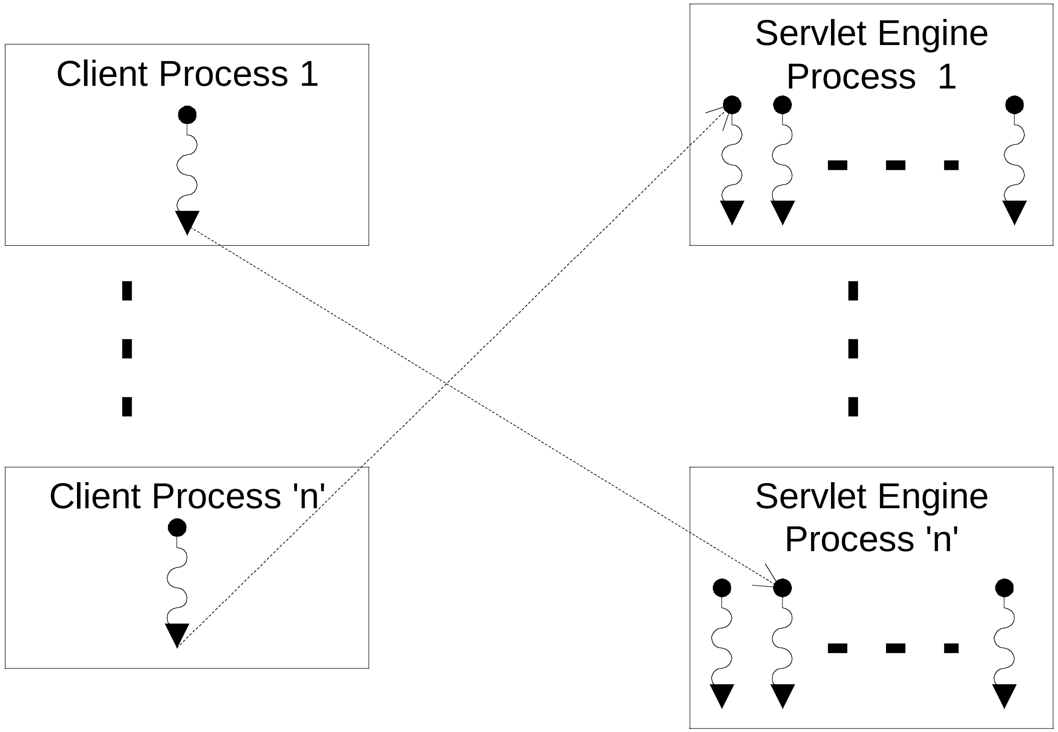
The Product Line Architecture

The functional view



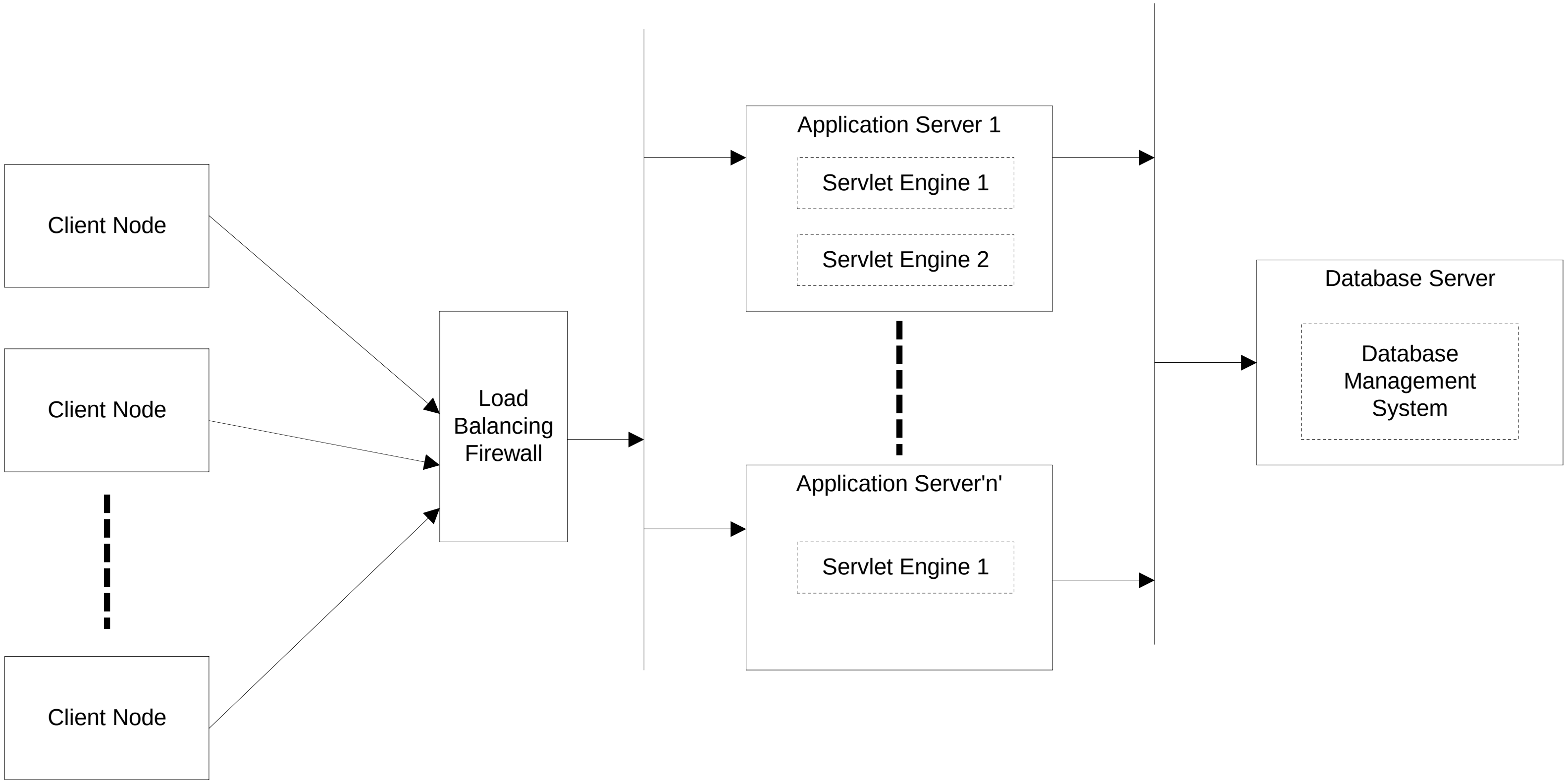
The Product Line Architecture

The concurrency view



The Product Line Architecture

The deployment view



Contents

What *Really* Happened!

The process proceeded as ….

- Product line architecture published – April 2001.
- Reviewed widely and generally accepted (minor changes).
- Development begins – May 2001.
- Product line architecture initially met resistance from the (new) server engineering team.

What *Really* Happened!

Why was the product line architecture rejected?

- Not “ours”.
- Why bother?
- Too complex!
- Why services? Just use libraries.
- The framework will be wrong and we’ll have to work around it.
- Need to start coding the servers now!
- Culture / trust between two development sites

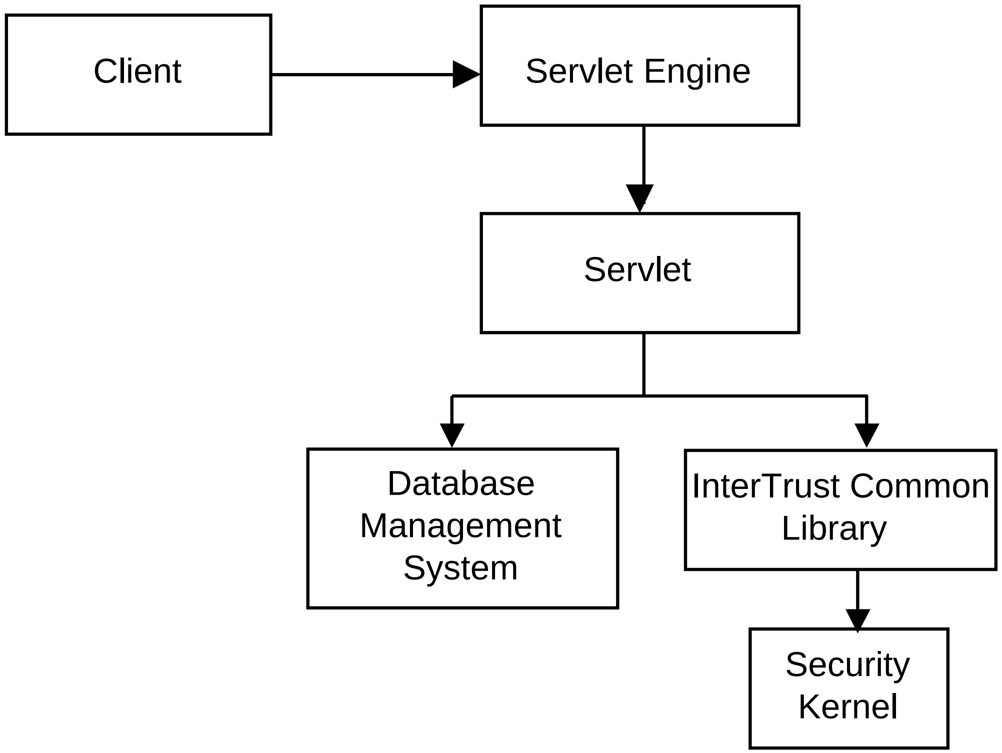
What *Really* Happened!

Frantic negotiation lead to a compromise of …

- The architecture accepted in spirit, but not detail.
- The framework is ditched in favour of.
  - A set of development guidelines.
  - Agreement over common structure, standard processing, code standards and use of libraries.

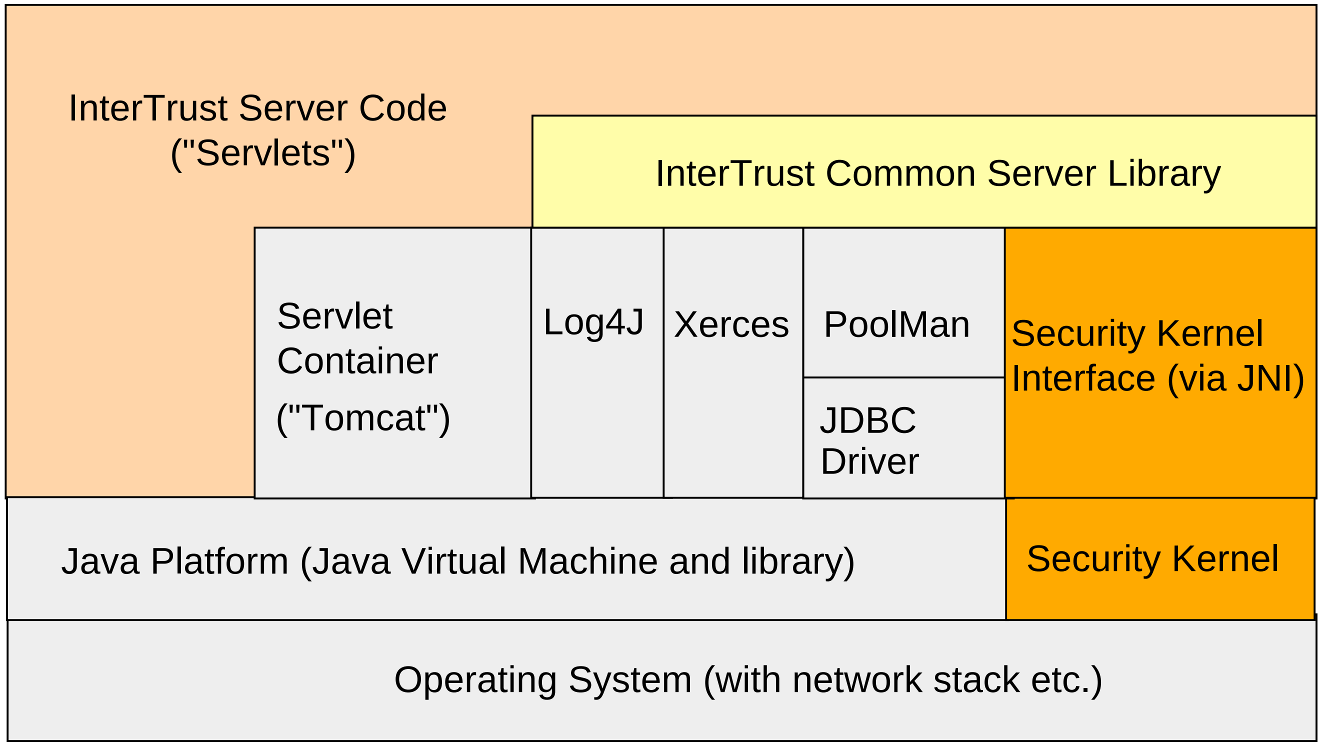
What *Really* Happened!

The functional view



What *Really* Happened!

The development view



Server specific code | Server shared code | Security code | 3rd party code

What *Really* Happened!

Results - milestones and metrics ...

- 6 servers completed by September 2001 (5 months).
- ~30 KLOC of unit tested Java and C++ delivered to QA.
- ~800 man days of *development* effort expended.
- Average of 20 txn/sec on reference hardware.
- A number of 10^6 transaction tests complete without problems.
- Estimate of 80% external server commonality.
- Estimate of 30% internal server commonality.

What *Really* Happened!

Results - success points

- Time to market (< 6 months).
- Client/server interfaces.
- Quality (measured as defects).
- Reliability.
- Performance goals broadly met.
- External server commonality.

What *Really* Happened!

Results - failure points

- Internal services.
- Internal server commonality.
- Rework required during development (for commonality).
- Run time efficiency (resource usage).

Contents

Lessons Learned - Process

Product line architectures

- Product lines are hard! (For us anyway)
  - —Not invented here.
  - —Lack of knowledge about architecture and product lines.
  - —Finding resources to work on it.
  - —Work scheduling more complex (e.g. framework dependencies).
  - —Worries about framework flexibility / overhead.
  - —Communicating the approach can be hard.
  - —When time is tight, people want to code not share!

Lessons Learned - Process

Product line architectures

- Our solutions were:
  - —Involving more key developers earlier (but resourcing is the new problem here!).
  - —Introduce it in a piecemeal way (guidelines, then small framework, then larger framework, …).
  - —Enforce change management discipline on shared (product line) assets (e.g. sync points, daily build, JUnit testing).
  - —Encouraging personal education in software architecture.

Lessons Learned - Technology

Java language and runtime – the good news …

- Java ***is*** slow … but acceptably so for many jobs!
  - Example: up to 10 times slower for pure computation (e.g. crypto) but only 20% slower for database access work.
  - We do all our security related processing in C++ (accessed via JNI) for security and performance reasons.
- JVM SMP server scaling isn't great – but acceptable for us.
  - 2 SPARC CPUs for Java + 2 for (C++) crypto OK.

Lessons Learned - Technology

Java language and runtime – the good news …

- Highly productive when compared to C++.
  - Wild Estimate: roughly twice as good for productivity.
- Very few runtime defects when compared to C++.
- J2EE provides an effective abstraction layer for engineers.

Lessons Learned - Technology

Java language and runtime – the bad news …

- Lots of memory required.
  - 60Mb – 100Mb+ servers.
- GC can often be a real pain to manage.
  - Server JVM is meant to help (but may or may not)
  - JNI seems to complicate GC.
  - Server stalls are common and can need a lot of work to reduce.
  - Object creation/destruction isn't free!

Lessons Learned - Technology

XML as a data encoding – the good news …

- XML is easy to trap, read and debug.
- Lots of good support software exists (IBM, Apache, MS).
- XML is easy to process programmatically (partly due to design, partly due to support software available).
- Engineers “get” it – make relatively few mistakes after short learning curve (e.g. no trailing nulls !).
- XML compresses well (thank goodness) .
  - ~50 % for our requests (40% tags, 60% data).

Lessons Learned - Technology

XML as a data encoding – the bad news …

- XML ***is*** big (due to tags, redundancy and base 64 data).
  - Example: Michi Henning's "StockQuote" example of 60 bytes for IIOP vs 360 for SOAP on comp.object.corba.
- Parsing ***is*** slow and validation adds significant overheads.
  - Example: 35 ms vs. 20 ms to parse request.
- Humans can't write XML – even with tools.

Lessons Learned - Technology

XML with Java for web services – the good news …

- Encourages loose coupling.
- Engineers “get” it – make v. few protocol mistakes.
- Easy to debug (very visible).
- Works well over LANs ***and*** WANs (e.g. firewalls).
- Flexible and cheap to change (e.g. no stubs/skeletons).
- Java has ***lots*** of XML processing facilities.
- GC in Java is ideally suited to parse/create type processing.
- J2EE provides a great “server container” for web services.

Lessons Learned - Technology

XML with Java for web services – the bad news …

- There is definitely overhead when compared to CORBA.
  - Example: C++ ORB server "hello" response 1msec, Java servlet equivalent response time 4 msec.
- XML's bulk can cause problems.
  - Example: 4K message size => wireless problems.
  - Example: 10^6 req x 4K ~= 8 **Gb** bandwidth needed.
- Current standards are (were) quite daunting.
  - XML Schema [complex].
  - WS-Security [brand new, complex].

Lessons Learned - Technology

Web Services in general …

- The loose coupling and call overhead means very coarse grained components need used.
- As with CORBA, standards are based on interface syntax which makes inter-organisation composition hard/risky.
- Security/policy management is v. vague at present.
- Client/server interaction needs planned more carefully than ever.
- No “native” support for session oriented protocols.
- At present no practical way to “call back” from the server (which forces a “polling” model for some applications).

Contents

Summary

We redeveloped a product family using a product line architecture …

- Overall, we had a lot of success and would do it again.
- Product lines need a lot of careful management to make them work. Gradual introduction may work best.
- Java has been a big success for our servers.
- XML has been a success too, but perhaps less dramatically. The future here looks complex!
- Both Java and XML come with costs, but for us they are acceptable. Understand these costs carefully before proceeding!

Discussion Points

Architecture

- Do you use a product line architecture?
- Have you had problems introducing s/w architectures?

Server Side Java

- Have you used it? Did you have similar experiences to us? If not, were your experiences better or worse?

Web Services

- Do you think web services style architectures will be widely adopted?
- If so, what are the top challenges you think we'll face?

RIGHTS|SYSTEM

INTERTRUST®

:: we pioneered
digital rights management™