

Java Card Evolution toward Service-oriented Architectures e-Smart 2004

Laurent Lagosanto
Systems Research Labs
Gemplus

Service Oriented Architectures Definition

- Beyond the Hype:
 - It's more than Web Services, (XML, WSDL, and SOAP included ;-))
 - It's older than Web Services (think to DCOM, CORBA, Jini,...)
- A service
 - Is a unit of work performed by a service provider and consumed by a service consumer
 - Is defined by its interface, i.e. the set of messages it accepts (methods in the OO world)
- A service oriented architecture
 - Binds services producers and consumers
 - Usually relies on a discovery service
 - Places requirements on the participants

Requirements of SOA

- Some are explicit
 - Consumers and producers agreed on the interoperability level
 - Web Services : XML for interfaces and SOAP for protocol
 - CORBA : IDL for interfaces and IIOP for protocol
 - Jini : Java for interfaces and mobile code for protocol
 - Consumers and producers are on the same network
 - Usually, it's an IP-based network (Internet, or intranets)
- Some are implicit
 - Services are developed with OO languages, with rich set of API
 - Java, C# but rarely C, or C++
 - Service containers are highly capable platforms
 - Services run in parallel
 - Services are developed by independent third parties

How Java Card fits into this space ?

- Let's be fair: It wasn't intended to
- Java Card 2.x objectives:
 - Bring WORA to the smart card
 - well-defined APIs
 - binary interoperability
 - Emulation of already existing smart card applications
 - the first deployed applications were native applications ported to Java
 - a Java-based SIM is an ETSI compliant SIM, just as a native one

The evolution has started...

Introduced with 2.1: the Shareable interface

- Defines a way to offer services *inside* the card:
 - Producer and Consumer agree on a Service Interface
 - The producer implements this interface within it's applet...
 - ... and export it to other applets by registering it to the system
 - Consumer applets lookup for the shared service into the registry...
 - ... and use the service through the Shareable interface
- While this is not visible from outside the card,
 - It permits secure deployment of service-oriented application inside the card, with all the benefits of a SOA

The evolution has started...

Introduced with 2.2: Java Card Remote Method Invocation

- Defines a way to offer services *outside* the card:
 - Producer and Consumer agree on a Service Interface
 - The producer implements this interface within it's applet...
 - ... and export it to remote clients ...
 - ... and gives proxy of the service to the consumer
 - Consumer application instantiate the service proxy on the terminal...
 - ... and use the service through the Remote interface
- Leveraged by JSR 177
 - Provides a way to uses JCRMI in J2ME applications
 - MIDlets (i.e. phone applications) can uses services offered by the card

Where do we stand ?

- Java Card is successful, it has reached its objectives
 - Native application are emulated
 - An onboard secure service sharing mechanism has been introduced
 - The platform can be dynamically extended at post-issuance by downloading new packages of Java Card classes
- Java Card is a mature platform
 - The 2.x versions are binary compatible
 - The set of APIs has become consistent
 - Some Vertical standards are relying on Java Card:
e.g. the 2G and 3G (u)SIM cards and their JC APIs

Is this sufficient ?

- Is Java Card mature/powerful enough to participate to SOAs ?
 - As a service producer ?
 - As a service consumer ?
 - As an other part of the infrastructure ?
- There are a couple of areas where Java Card doesn't meet the requirements to be a service consumer or producer:
 - The richness of the language, and APIs
 - The limitations of the runtime environment
 - Its "exotic" communication protocol
- (How) should we compete ?
 - J2ME is clearly dominating the embedded market
 - Web Services (therefore SOAs) are entering this space

Let's Have a look on a set of desirable evolutions...

Runtime environment evolutions

- Multithreaded execution engine
 - The VM can run multiple Java threads
 - Doesn't mean that an application can create threads on its own (creating a thread is subject to a permission, that only the system or framework may be granted)
 - *This, coupled with a better communication stack, may avoid an application performing a long computation to block the whole card, and would ease the development of independent applications*
- Automatic garbage collector
 - A GC is said to be automatic when it doesn't have to be called explicitly, but does its job when required
 - *On the contrary to the 2.x "on-demand" GC, an automatic GC provides a more convenient programming model. If coupled with an "objects in RAM" model, it can lead to an efficient memory (RAM and NVM) management*



A richer set of API, and a more flexible framework

- CLDC-like core APIs
 - Much richer than 2.x APIs, provides a “real Java” flavor
 - Useful APIs reflecting the platform capabilities (int, String, Vectors, Threads, Generic Connection Framework, ...)
 - *CLDC is a bigger API than JC2.x, but there's no point in reducing the system by limiting the capabilities offered to the application: this limits the range of feasible applications, and usually leads to bigger applications (every one re-inventing the wheel)*
- Flexible framework, supporting multiple application models
 - Java Card 2.x is tied to the javacard.framework.Applet model
 - The new framework should be flexible enough to sustain multiple application frameworks (so called models)
 - It can be divided in two parts
 - The platform manager (code loading, code management, registry)
 - The platform API (used to write application models in Java)
 - *There's no point in being limited to the APDU processing model, especially if alternate communication means are supported*



An evolution of the communication stack

Communication stack requirements:

- To be based on standards (not re-inventing the wheel)
- To be media-agnostic (stack supported by existing and future HW links)
- To be application protocol-agnostic (applications can choose the most adequate application protocol, in a half or full duplex way)
- To support multiple applications communicating in parallel (stack supports multiple opened channels with transport of multiplexed messages on behalf of applications)
- Take into account the reliable network addressability demanded by the SIM/USIM cards (see BIP / CAT_TP)

An Internet-compliant communication stack

- IP is the common factor of every SOA, and is reaching the embedded space:
 - HTTP for SOAP, IIOP for CORBA, and RMI for Jini: they all rely on IP
 - With the emergence of GPRS and UMTS, exotic protocols tend to disappear: every phone is an IP node
 - Most operators are now looking at merging their voice and data networks, using VoIP
- The new communication stack should use the TCP and IP protocols
 - While the TCP/IP stack satisfies these requirements, there are additional advantages
 - *IP brings independence and flexibility: any application protocol can be transported by IP without any modification of IP, and any H/W link can transport IP packets, without any modification of IP*
 - *TCP brings a stream-oriented vision of communication, and provides mechanism like flow control that helps satisfying application needs while matching the card memory constraints*
 - Complex tasks like protocol or session management are no longer left to the application but taken in charge by the system (*that brings reduced application size*)



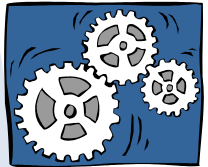
All these evolutions collected: Java Card Next ?

- Business requirements



- Should enable new (means additional) markets for Java Card
- Target 32-bit microcontroller architecture with large memories
- Closer to mainstream Java
- Communication limitations should be removed
- Support different kinds of applications

- Technological translation



- Multithreaded execution engine
- Objects in RAM
- Automatic garbage collector
- On-card linking
- CLDC-like core APIs
- Flexible framework, supporting multiple application models
- New communication stack using Internet Protocols

About compatibility and migration paths...



- Introduction of new features may require changes to the binary file format of applications:
 - This breaks the binary compatibility (hence a major version number change)
 - But feasibility of extending the existing CAP file to embed such features was not proven
 - And using an existing format allows re-use of know-how, development and deployment tools
- The JCF is working on solutions
 - application source code migration: the 2.x framework may be provided as an additional application model (but *not* as the preferred model)
 - ...
- Existing services migration:
 - services offered by the card through it's unique half-duplex link may be migrated on top of multiplexed channels and offered simultaneously on such a platform (e.g., SIM-like services can be offered through a TCP port, #1111 or #1114 for example ;-)

Conclusion

- Java Card 2.x targeted native smart card replacement market
 - Java Cards had to be seen as regular smart cards
 - First applications were native applications ported to Java Card
- Java Card Next targets:
 - Existing markets to solve coming issues (i.e. demand for more powered services and system opened to non-cooperative application from different content providers)
 - New markets where a secure runtime environment is required
- Java Card Next should compete with J2ME to participate in SOAs:
 - Not limited to be the slave of the terminal, taking its own decision, based on card issuer policy
 - Conscious of the internet protocols and ready to be integrated (as a client, as a server, as a controller, depending on the application)

Questions ?

lago@research.gemplus.com