Extensibility, Componentization, and Infrastructure

Ted Slupesky
(slupesky@us.ibm.com)

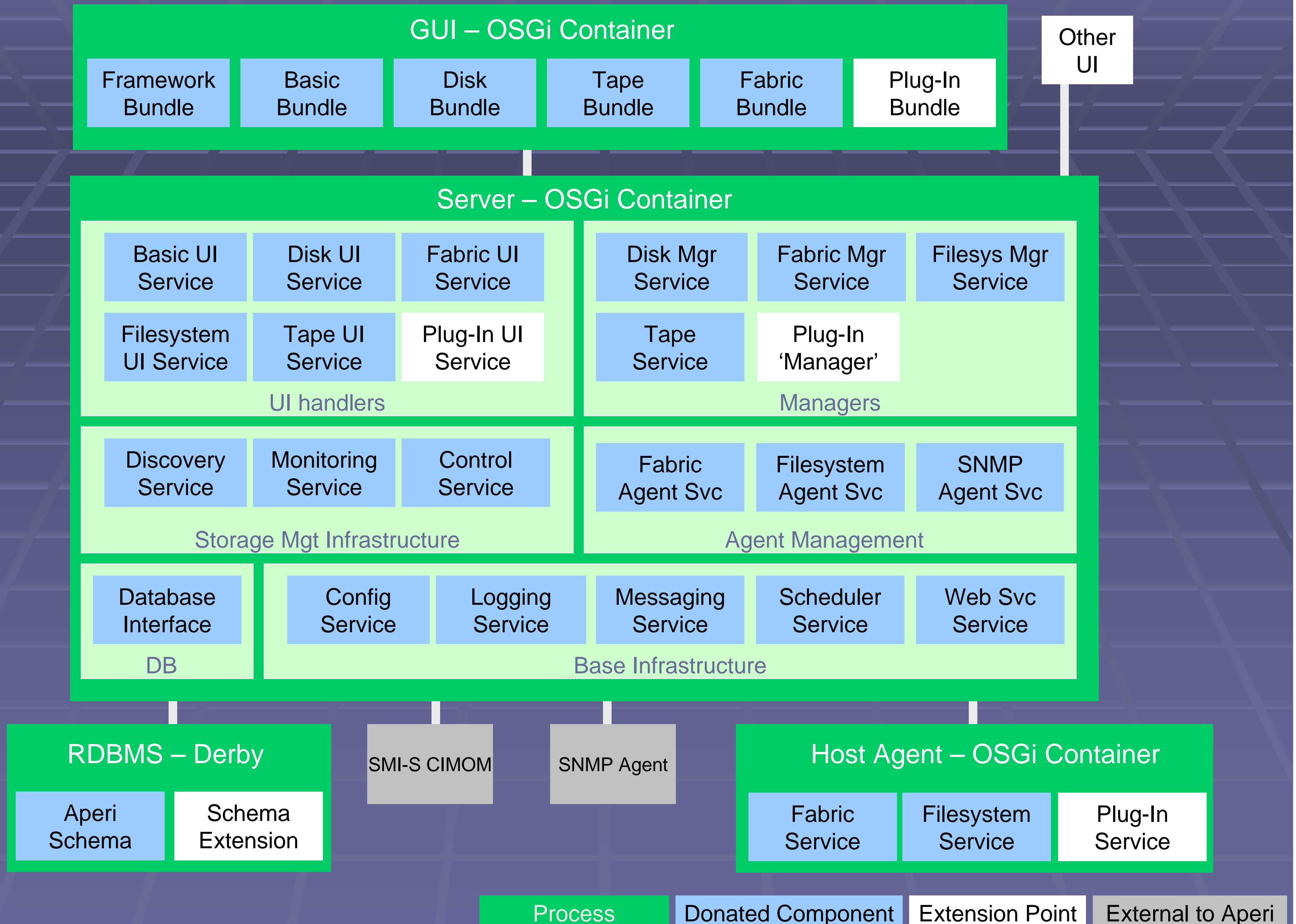
Copyright 2006 IBM Corp. Available under terms of the Eclipse Public License http://www.eclipse.org/legal/epl-v10.html

Introduction

- This is the first 'tech talk' regarding IBM's proposed donation to the Aperi community
- It covers two topics:
  - Extensibility & componentization
  - Base server infrastructure
- Fundamental technical level-set for IBM initial contribution
- The first part of this presentation (extensibility and componentization) is more forward-looking than the rest, because we think the topic is so important to get right for the Common Platform

Extensibility and Componentization

Proposed Architecture of Platform



Importance of Extensibility and Componentization

- Extensibility is a critical requirement for the Aperi Common Platform
  - The ability to insert new functions into the Aperi Common Platform at runtime, dynamically, without recompilation
    - A ‘plug-in model’
  - It enables developers to create value-added or higher-level applications and insert them at runtime into the common platform

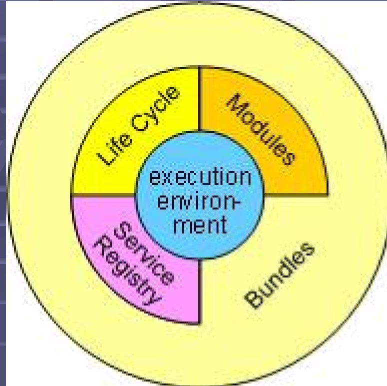
- Componentization is closely related
  - The architecture is composed of modularized components with well-defined interfaces. A component can be replaced by an alternative implementation that conforms to the required interface.
  - Supports an extensibility model
  - Allows developers to select components from the Common Platform and reuse them in their own applications

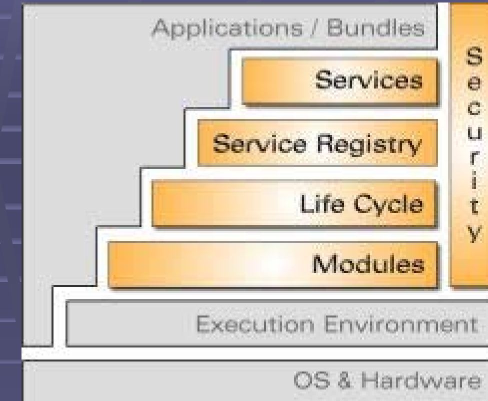
- Working together, extensibility and componentization allow:
  - A plug-in can add new function to the platform (insert new components)
  - A plug-in can replace existing function in the platform (replace existing components)

Modes of Interaction with Aperi Common Platform

- It is critical that we have consensus on what the modes of interaction are
- Users could download ‘Aperi 1.0’ as an application, install it, and use it to perform ‘basic functions’
- Developers could create value-added applications (plug-ins) that integrate into the common platform
  - Plug-ins could be open source or not
  - Users could acquire these plug-ins (either by download or by purchase) and load them into their Common Platform instance
- Developers could create (and sell) complete offerings that include the common platform plus their proprietary value-added plug-ins
- Developers could contribute changes to the base Common Platform infrastructure (to be picked up in the next release of the Common Platform)
- Developers could take pieces of the Common Platform and integrate them into their own, unrelated products
  - Subject to Eclipse license terms, which are commercially friendly
  - This reuse could be at any level: source code function, source file, whole component…

OSGi





- We propose OSGi as the foundation of our extensibility & componentization models
- Lots of information at www.osgi.org
- "The OSGi Service Platform provides a computing environment for applications, called bundles, which execute together in a single JVM. Bundles can be installed, updated, and uninstalled dynamically. Installed bundles find a rich environment to execute in."
- What is OSGi?
  - "The OSGi™ specifications define a *standardized*, *component oriented*, computing environment for *networked services*."
  - "Adding an OSGi Service Platform to a networked device (embedded as well as servers), adds the capability to manage the life cycle of the software components in the device from anywhere in the network."
  - "Software components can be installed, updated, or removed on the fly without having to disrupt the operation of the device."
  - "Software components are libraries or applications that can dynamically discover and use other components."
- OSGi is a small framework intended to run in a broad range of environments
  - Can run in J2ME on embedded devices

OSGi Architecture

- The OSGi framework is implemented as a container process that loads components (bundles)
  - A container is an OS process that runs a JVM
  - Bundles export services or contain common library code
- The framework consists of layers, three of which are mentioned here:
  - The Module Layer is responsible for reading OSGi 'bundles'
    - A bundle is a structured .jar file
    - A bundle is the 'fundamental unit of modularization'
  - The Life Cycle Layer provides an API for controlling security and life cycle
    - Install, upgrade, and remove bundles, which are versioned
    - Start and stop bundles
  - The Service Layer allows bundles to export 'service APIs'
    - The service model is a publish, find and bind model
    - A service is a normal Java object that is registered under one or more Java interfaces with the service registry
    - Bundles can register services, search for them, or receive notifications when their registration state changes
- From the developer's perspective: you write services, package them as bundles, and deploy them in the container

Extensibility and Componentization of Aperi Common Platform

- We propose that
  - Each process of the Aperi architecture (UI, server, agent) be an OSGi container
  - The components within those processes be packaged as OSGi bundles that export OSGi services
- Components can be reused in other applications
  - Naturally there will be some inter-component dependencies
- OSGi is central to Eclipse

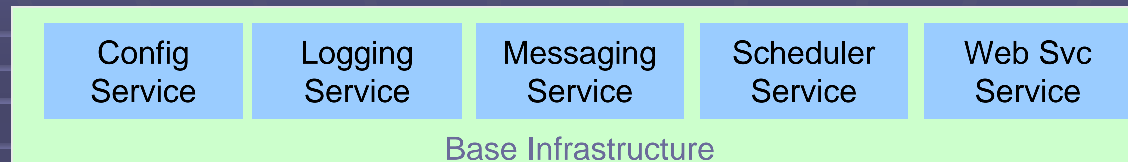
What Does It Take to Extend the Common Platform?

- Stepping back, what would it mean to extend the common platform by creating a higher-level, value-added application?

- The developer would create one or more plug-ins to the Common Platform
  - A GUI plug-in, with the code for the new panels
  - A server plug-in to handle requests from the GUI and map them to calls to other server plug-ins
  - A server plug-in that provides the basic functions of the value-added application
    - This plug-in might be responsible for additional database schema
  - Possibly, an agent plug-in, if some host-based function is required
  - The 'application' could upgrade other components, if required
    - Upgraded components need to be compatible with previous implementations to avoid compatibility problems

Base Server Infrastructure

Infrastructure



- Covers base infrastructure components that are independent of the storage management domain
  - Config Service
  - Logging Service
  - Messaging Service
  - Scheduler Service
  - Web Services Service
  - And… a discussion of the TPC 3.1 ‘Service Manager’

Config Service

- The Config Service provides access to fundamental configuration parameters
- Some live only in a .properties file (basically, port numbers and database connectivity parameters)
- The rest live in the database
- The parameters:
  - Basic info
    - Port numbers
  - Database connectivity
    - Database name, JDBC URL to access database, JDBC Username and password (encrypted)
  - Logging and tracing
    - Trace level for each component
  - Service configuration
    - Which services to start
  - General configurable parameters
    - Various tunable timeout and retry values
- API
  - APIs to get, set, remove parameters
  - Passwords get special treatment (for encryption reasons)
- Events are published when a parameter changes

Logging Service

- Manages multiple log files, with rollover and size caps
- Application creates multiple loggers
  - Per-service message files
  - Per-service trace files
  - Audit log
  - Individual ‘job logs’
- Loggers are retrieved from LogManagerFactory
- Messages are sent to a Logger via ‘message’ method via keys
  - Keys are looked up in message files, so that messages are translated
- Logging service provides abstract interface over more basic ‘logging provider’ such as JLog

Logging Examples

- Example Usage

```
msgLogger = LogManagerFactory.getMessageLogger(MyLoggerName);
traceLogger = LogManagerFactory.getTraceLogger(MyTraceLoggerName);
msgLogger.setMessageFile(MyMessageFile);
…
msgLogger.message(Level.INFO, className, methodName,
   “ErrorMessageID”);
if ( traceLogger.isLogging() )
   traceLogger.exception(Level.ERROR, className, methodName, snmpE);
```

- Example configuration properties

```
myTraceLoggerName.listenerNames=file.trace
myTraceLoggerName.LoggerType=TraceLogger
myTraceLoggerName.logging=true
MyTraceLoggerName.level=WARN
```

Messaging Service

- Based on JMS (pub/sub implementation)
- Used for loosely-coupled communication among components
- Events can be ‘subscribed’ to, by supplying the name of an Event class (as the ‘topic’) and an ID for a class that implements MessageListener
- Events can be received via onMessage()
- Events can be published via JMS createMessage()

Messaging Examples 1

- The process for subscribing to events is:
  - Create a JMS provider factory
  - Use the factory to create a subscriber

- The subscriber will specify the following parameters:

  Topic

  String defining the Topic. The general convention within the Device Server is to use the classname of an Event Class for the topic. The internal logic will send events matching this topic or topics inheriting from the topic in the Event hierarchy.

  Subscriber ID

  Optional ID identifying the subscriber. NULL if not specified.

  Filter

  For future use

  Listener

  Class implementing MessageListener that will be called when events matching the Topic and Filter are published.

  Example:

```
factory = FactoryFactory.createFactory(FactoryFactory.CURRENT_JMS_PROVIDER,
                                       null);
subscriber = factory.createSubscriber(MyEvent.class.getName(),
                    null, // subscriber ID – none specified
                    null, // filter – none specified
                    myCallback); // instance of class that implements
                                       MessageListener
```

Messaging Examples 2

- **Receiving Events**
  - When an event matching the topic and filter is published, the onMessage function of the MessageListener interface will be called.

```
public void onMessage(Message msg);
```

- It is the responsibility of the receiver to quickly execute the onMessage function to prevent impacting other listeners.

- **Unsubscribing for Events**
- Clients unsubscribe for events by closing the callback handler.

```
try
    {
        subscriber.close();
        factory.close();
    }
    catch (Exception e) {
            // handle the exception
    }
```

Messaging Example 3

- **Publishing Events**
  - The process for publishing an event is:
  - **Setup the event template**
  - The template defines the parameters that can be used in the Subscription filters, such as publisher ID.

```
factory = FactoryFactory.createFactory(FactoryFactory.CURRENT_JMS_PROVIDER, null);
template = factory.createMessage(null);
template.setStringProperty(PropertyName, PropertyValue);
```

- **Publish the event**

```
TopicPublisher publisher = factory.createPublisher(anEvent.getClass().getName(),
   template);
try {
   Message msg = factory.createMessage(anEvent);
   publisher.publish(msg);
} catch(JMSException je) {
   // handle the JMS exceptions
} catch(Exception e) {
   // handle other exceptions
}
```

- **Note: The client invocation of the onMessage calls will be performed within the thread of the publisher – the clients must quickly process and return from the onMessage call to prevent impacting other listeners.**

Scheduler Service

- Provides ability to run ‘jobs’ on a schedule
  - These are data collection jobs
- Can distribute portions of a job to specific agents
- Primary interaction is via the database rather than specific API
  - That is, the scheduler looks for work to do in a specific database table
  - When it’s time to do the work, it does so
  - When complete, it writes the status back to the database
  - So, clients interact with the scheduler by reading and writing the required database tables

Web Services Service

- Creates SOAP wrapper and WSDL for public API exposed by 'Services'
- Based on Apache SOAP
- Service implementers and clients don't have to know anything about SOAP or WSDL – can use provided 'proxy' libraries to ignore protocol details if they wish
- In TPC 3.1, this is part of the 'Service Manager' – see following slides

Service Manager

- From December presentation:
  - TPC 3.1 does not use OSGi in the server
  - It has two servers, ‘Data’ and ‘Device’
  - The Device Server has an extensibility mechanism called the ‘Service Manager’
- Implemented as a servlet
- Servlet reads config file for ‘services’ to load, then loads them
- Just a simple way to dynamically configure the services running in the server

ServiceManager Interface

```
public interface IService {
   public java.lang.String getVersion();
   public java.lang.String getName();
   public boolean saveState(); <- for future use
   public boolean restore(); <- for future use
   public boolean remove(); <- for future use
   public boolean startup();
   public boolean shutdown();
   public java.lang.String getDescription();
   public Status getStatus();
   public java.lang.String getStats(); <- for future
   use
}
```

Service Definition

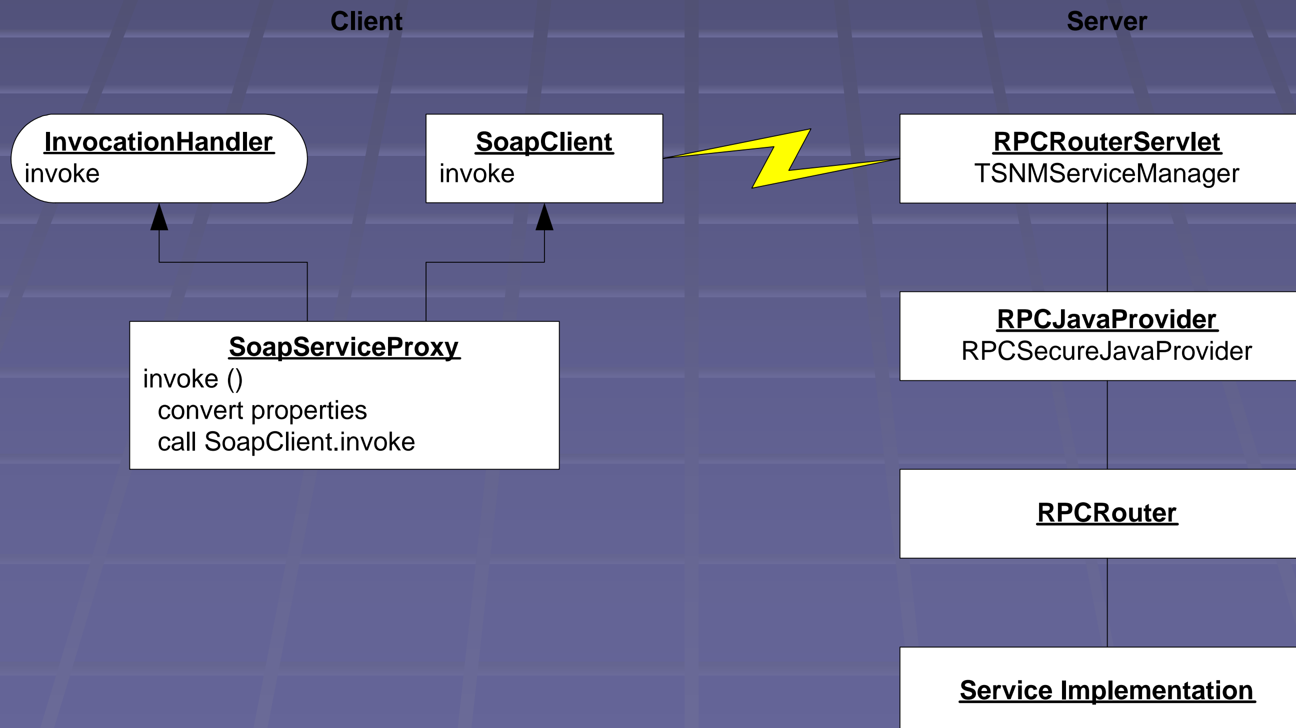
- Configuration for a service:

```
DiscoverService = com.ibm.tpc.DiscoverService application
   autostart nonstatic 6
```

  - Service Name
  - Class
  - Scope (Application vs. Session vs. Request)
    - Application = a singleton instance of the service is used to invoke all method invocations. In this case state can be maintained across all clients.
    - Request = a new instance of the service is used to invoke each method invocation. In this case no state is maintained.
    - Session = a single instance of the service is used to invoke methods in a particular http session. In this case state can be maintained per session per client.
  - Autostart (Yes or No)
  - Static (Static vs. Non-Static) – for future use
  - Order

Service Communication

- Two proxies are available to invoke services
  - Remote proxy uses SOAP
  - Local proxy makes a local call (no SOAP)
  - Proxy handles protocol issues. Caller can ignore whether the transport is SOAP or not.



Proxy Example

- Service Call

```
IDiscover discoveryProxy =
  (IDiscover) ServiceRegistry.bind(
        <url>:<port>,
        IDiscover.SERVICE_NAME,
        IDiscover.class);
```

- 'Local' Service Call

```
IDiscover discoveryProxy =
  (IDiscover) ServiceRegistry.localBind(
        IDiscover.SERVICE_NAME,
        IDiscover.class);
```

Conclusion

Conclusion

- The big question: what is the right prioritization for extensibility work versus other possible work on Aperi?