



Vision & High Level Design Overview

OpenDI Release 1

October 2008 v1.6

J. Carolan, J. Kirby, L. Springer, J. Stanford

<http://opendi.kenai.com>



Java™

Abstract

This document provides a high level overview of the motivation, problem space, and OpenDI software. The intended audience is architects, customers, and others interested in becoming familiar with the major functional areas of the OpenDI framework and how we got here. This document is not a detailed design document, however a detailed design will extend from this overview.

Copyright 2008 Sun Microsystems Inc.

Table of Contents

Section 1: Introduction.....	5
The Enterprise Cloud Imperative.....	5
Towards a Data Center Operating System.....	5
Customer Needs and Issues.....	6
Motivations – Architecture at the Right Level of Abstraction.....	9
Leading up to Clouds (Or Utility Part Deux).....	9
Section 2: Introduction to OpenDI.....	11
Five Principles of Cloud Optimization.....	11
Three Boundaries.....	11
Applications can be Optimized for Clouds	12
Resources are Controlled Via Policy and are Transient (vs. Static).....	13
There is a Dynamic Feedback Loop that Affects the Clouds Services.....	13
OpenDI Releases and Realization.....	13
Cloud Architecture Concepts.....	14
Executive Platforms – Multi-layer Orchestration Framework	18
Legacy Integration.....	20
Model-driven and Actionable.....	21
Changes.....	21
Cloud Modeling.....	21
Cloud Control and Provisioning -- A multi-dimensional problem.....	22
Granularity.....	22
Feedback – Clouds are a Two-way Street.....	24
A Deployment Example.....	25
Section 3: Design Overview.....	27
Major Design Considerations.....	27
“Be the Glue”	27
Model Driven.....	27
Distributed	27
Enterprise Context.....	28
Separation of Concerns.....	28
Policy-Centric.....	28
Controlled Transparency.....	28
Basic End To End Flows.....	28
Solution Developer.....	28
Operator	30
Release Manager.....	32
Administrator.....	32
Structural Design.....	32
Major Systems.....	35
Bootstrap.....	35
Authentication/Authorization.....	35
Task Management.....	36

Coordination.....	36
Controllers.....	36
Event Messaging.....	38
Adapters.....	39
Registration.....	39
Factory.....	42
Types.....	42
Client Proxy.....	43
Workflow.....	43
DCRM (Data Center Reference Model).....	44
Provisioning.....	45
Persistence.....	46
Policy.....	46
Monitoring.....	46
Reporting.....	47
Communication (Internal).....	47
Appendix 1.....	48
OpenDI/JBI Integration Requirements and ESB Analysis.....	48
Appendix 2.....	58
Useful Websites.....	58
Other Resources.....	58

Illustration Index

Illustration 1: Lots of Clouds -- One Deployment Model?.....	6
Illustration 2: Granularity, Scale, and Granularity of Changes.....	7
Illustration 3: Three Boundaries of Cloud Infrastructure.....	12
Illustration 4: Releases Modeled after Cybernetic Feedback.....	13
Illustration 5: Service Platform and Executive Platform.....	15
Illustration 6: The Stack.....	16
Illustration 7: Tools and Platform Example.....	16
Illustration 8: Layers and Functional Areas.....	17
Illustration 9: Orchestration.....	18
Illustration 10: Multi-Platform Orchestration.....	19
Illustration 11: Models, Tools, and Use Cases.....	22
Illustration 12: Standardized Operating Environment Granularity.....	23
Illustration 13: Deployment Example.....	25
Illustration 14: Solution Developer Workflow.....	30
Illustration 15: Operator Workflow.....	31
Illustration 16: OpenDI High-Level Structural View.....	34
Illustration 17: Message Flow.....	38
Illustration 18: Adapter Registration.....	42

Section 1

Section 1: Introduction

The Enterprise Cloud Imperative

Increasingly developers are looking elsewhere to deploy their applications. Increasingly their businesses' cost pressures are winning out – and see value in “cloud” platforms to help manage cost. These businesses also ask the question “if Amazon or Network.COM can do this – why can't I run my own infrastructure this way?”

Secondly, other systems companies are increasingly investing in the “cloud” infrastructure space, including building large data centers, partnering with cloud providers, etc.

- [IBM Plans \\$360M Cloud Data Center in NC](#)
- [Sun's On-Demand Network.com Grows Again](#)
- [HP Moving Defense Department Into The Cloud](#)
- [Sun's On-Demand Network.com Grows Again](#)
- [HP, Intel, Yahoo Team on Cloud Testbed](#)
- [Microsoft: New Design for \\$500M Iowa Site](#)

Most of these companies own management platforms that customers use to manage their own data centers. Increasingly they will use these tools to deploy to the “internal” cloud or the external cloud – only differentiated by the SLA and other criteria. The types of resources they can run one will basically be the same.

For example:

- [IBM Advances Cloud Computing w/ Tivoli Provisioning Manager](#)
- [Sun Announces Availability of Sun xVM Ops Center, Delivers on Sun's Virtualization Strategy](#)
- [HP : The Next Wave: Everything as a Service](#)
- [HP-Opware Deal Shows IT Automation Kicking Into High Gear – IT](#)

Towards a Data Center Operating System

OpenDI in a nutshell is a project based on three major releases to further the concept of the data center-wide (or even multiple data centers!) operating system. It provides a data bus, specification for events (that change or trigger changes), and an adapter framework that assist in the integration of components of the system. These components include:

- Tools (software packages) that help manage infrastructure (e.g. xVM Ops Center or CiscoWorks)
- Databases and other state management systems that help maintain the models
- Prescriptive models that help describe the policies of the systems
- Devices that provide resources and thus the services that utilize those services.

The world of utility, N1, and clouds is upon us again, this time with some major differences:

- Virtualization everywhere – from the server to the OS to the network
- Powerful servers – multi-purpose, general purpose systems that use software to provide their key differentiation rather than custom hardware, ASICS, etc.
- And the proliferation and adoption of utility models such as Amazon and Salesforce.COM.

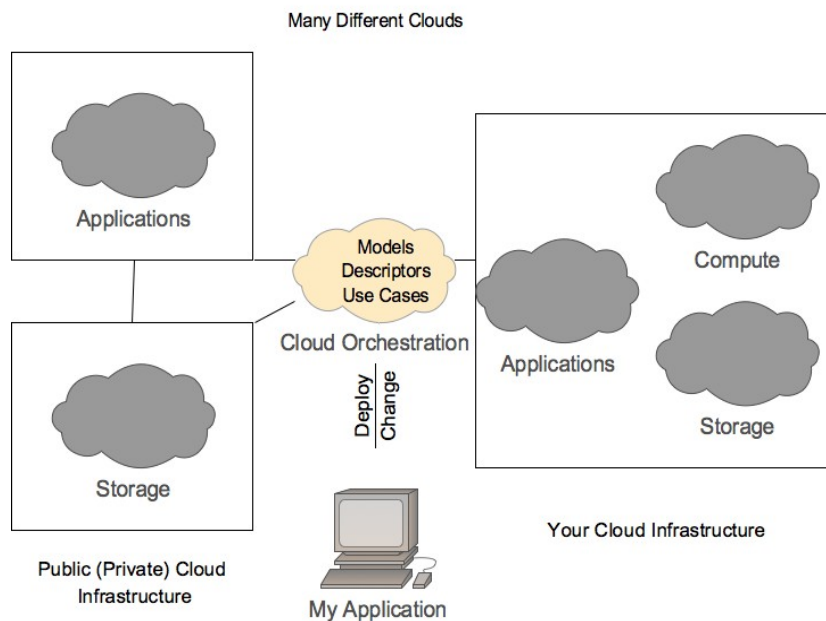


Illustration 1: Lots of Clouds -- One Deployment Model?

Customer Needs and Issues

Our customers are not only looking to develop a strategy to utilize off-site resources but also internal resources. They are looking for a unified process for dealing with requests to deploy securely applications across these types of infrastructures. They are looking for a way to rationalize and consolidate their tools used in this space, and are finding that some of their basic IT tools are unsuitable for an increasingly flexible, dynamic world.

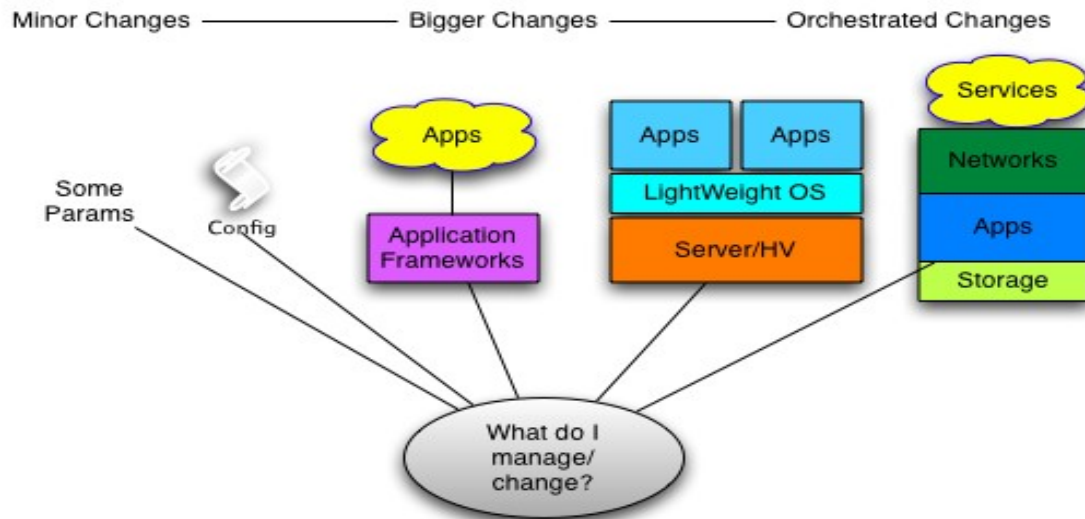


Illustration 2: Granularity, Scale, and Granularity of Changes

- 1) CMDB and Configuration Management – Many companies do not have detailed visibility into what applications are running where, system utilization, the relationship between network, storage, and compute nodes, etc. – even for environments that don't change very often. Even companies that are providing “as a service” offerings are managing “by spreadsheet” instead of “by model”. This is a critical issue for the cloudscape.
- 2) Process maturity and Time To Market – Existing operational processes and 'next gen” application deployment must merge successfully to ensure systemic qualities of the services meet target service levels. Today, many cloud prototypes allow developers a great deal of control. Conversely, traditional operations organizations move relatively slow (usually with well-defined processes) and cannot meet the temporal needs of dynamic applications. This highlights another area...
- 3) Increasing Integration Required across Data Center Management “Silos” – virtualization requires the network to be more closely integrated than in previous technology deployments. Applications are now defining the functions of the virtualized operating system, riding on top of a common VM domain. To master operational control of the cloud there must be a closer working relationship between these silos, and in turn, the automation technologies that are used. These views must all be taken into account by the cloud management system.
- 4) Business and SLAs – The business must be able to define the quality of service and IT must be able to choose both internal and external resource providers. If IT doesn't make these decisions, business units

themselves will. The process (see above) must incorporate both to ensure systemic qualities of the services are maintained at expected levels.

- 5) Governance and Integrity – Constant changes must match back to a governance policy and meet the governmental requirements of the business. This may require both internal and external resources and their hosted applications to undergo additional testing, verification, and audit.
- 6) Big changes or small changes – Customers need to be able to facilitate changes whether large or small. They already use similar change processes for these actions, but the system must be able to perform the change (based on the model) and ensure execution. Customers may choose to manage by a configuration file or a VM image, or a developer may simply want to update a WAR file. These should all be made possible by the system.

Motivations – Architecture at the Right Level of Abstraction

OpenDI was initiated as a field-based project within Sun's Global Systems Engineering. Since systems engineering for our customers usually requires a solution, we start with architecture and architecture principles and work our way to implementation.

Increasingly we have found that it is difficult to operate at a sufficient level of abstraction to provide a general solution attractive to more than one customer, thus allowing customer differences to be factored in and design trade offs or decisions as part of that solution. Technologies were changing so rapidly that we would integrate one such product by the time it was no longer being used and something else would take its place, sometimes doing better or worse.

At the same time, virtualization had not yet become a buzz word but the “dynamics” of IT infrastructure was certainly going to change and the industry had yet to develop standards and best practices related to this. Additionally, members of the original team looked at Sun's “stack” and how it would work in a virtualized and dynamic world – or [Project JxSON](#).

From JxSON we learned about and documented patterns around application development, deployment, relationships between consumer and providers, how important the network was to make virtualization work, etc. These learnings fed into [Dynamic Infrastructure](#), a set of consulting services supported by the DI reference architectures and implementations.

So why OpenDI?

- Products changing, with little value in point to point integration except for early prototypes
- Lack of overall service orchestration and understanding of complex use cases and dependencies
- Lack of abstracted models, where most models are strictly embedded into a set of tools and data could be useful elsewhere, is duplicated, not the right level of granularity, etc.
- Treat dynamic provisioning and control as a multi-dimensional problem – with varying depths

Leading up to Clouds (Or Utility Part Deux)

Over the course of the last two years, there has been serious changes to the IT landscape, punctuated by the economic climate that is now pervasive across the world. Where growth was the number one priority

everywhere, it is now measured and limited by efficiency and the need to provide highly optimized platforms at very low cost.

Another key change is the growth of “cloud” infrastructures, mostly outside the space of one's own data center. This includes force.com, AWS (Amazon web services) and others. These typically are categorized as 3 different types of services:

- Platform – broadly defined, but includes Project Caroline (Glassfish platform) or Force.com's application platform (also Google Apps), and Ning in some ways is a good example of both a Platform and Software as a Service.
- Software – specific end user functionality provided as software of the network (e.g. Salesforce.com) or a composite of the above then offered as SaaS.
- Infrastructure – broad infrastructure, including storage , typically is delimited by the OS, but including networking interconnects and routing, to run any services – usually a “VM” or container. See Joyent for example. May include storage and some basic advanced networking (though very limited.)
- Utility 2.0- For the enterprise, this is the second major attempt (and press coverage) at moving towards a common shared infrastructure. May include any of the above models.

- Shared Platforms

- Shared Infrastructure

- Software Utility as a Shared Service

The difference this time around is serious consideration and cost pressures (demands) to find creative solutions...

- External Hosting (someone-elses cloud accessible via the network)

- Internal Hosting (Utility 2.0) – this is the enterprise view of “cloud” computing when hosted internally. It looks like the N1 vision to some degree.

The industry is providing robust virtualization toolsets though focused at the infrastructure platform itself, and not the surrounding issues of complexity, deployment issues, application provisioning, network management, and more that come with it.

Section 2: Introduction to OpenDI

OpenDI provides a framework for higher level integration and automation using open standards. Versions of OpenDI (and its predecessors) have been deployed at customers for several years providing them \$100s of millions in cost savings and 80-90% better efficiency in staff costs and time to market. OpenDI is under process to open source in early FY09.

We believe this framework approach is key and particularly appropriate for Network.COM. As customers, and Network.COM continues to build new next generation platforms while continuing to integrate with legacy components. One size fits all and a single product approach will not work. A great example of this is a large scale “cloud” project Sun has been asked to participate in that has over 45 different legacy “management” tools that need some level of analysis and eventually higher level orchestration. Some may go away but not soon enough to be replaced by a single non-existent product; and then the holy grail will come under fire as some new technology is introduced.

Five Principles of Cloud Optimization

As background, OpenDI proposes five key principles in address cloud infrastructure optimization

Three Boundaries

There are three major boundaries to cloud infrastructure. The resulting layers are linked together via a common orchestration communication bus and may use common models (OpenDI in this paper).

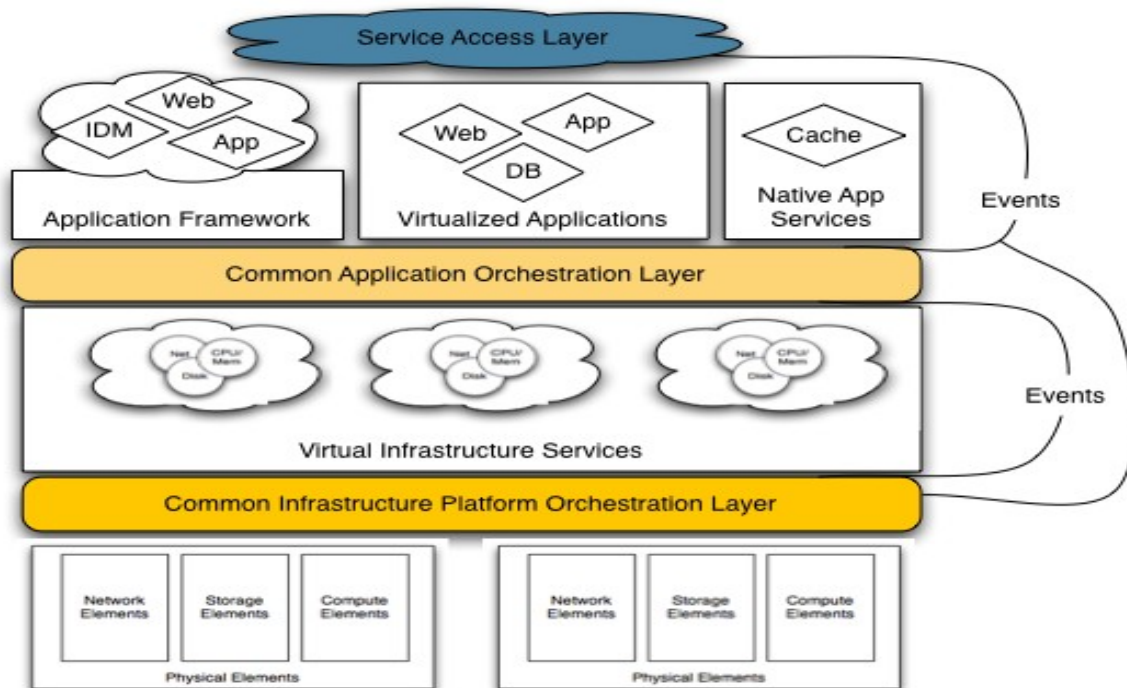


Illustration 3: Three Boundaries of Cloud Infrastructure

- 1) The Service Access Layer – defines and regulates access to the overall cloud platform services.
- 2) Application Platform Orchestration Layer – defines and regulates the application stacks and associated systems (in some ways this is A2V – application to “virtualization”)
- 3) Infrastructure Platform Orchestration Layer – defines and regulates the infrastructure platform and associated systems (most call this P2V – moving from a physical instance to a virtual one)

Applications can be Optimized for Clouds

The forces behind this are mostly around dynamics – how often do changes occur, at what granularity, etc. Complex “cloud optimized” applications appear more like application frameworks, SOAs, and clusters. Clouds also work well with simple applications, basic binaries, and integrated resource managers (e.g. GridEngine).

Clouds are Instantiated, Modified and Controlled Via Models

Models are distinct entities within the cloud. Models are independent of their tools. These models help describe the rules, configuration, changes, and policies that are allowed or not within the dynamic cloudscape.

Resources are Controlled Via Policy and are Transient (vs. Static)

Tight resource coupling while possible (and allowed by the OpenDI framework) should be avoided. This allows for “fail in place” and other continuous architecture techniques as well as overall maintainability. These policies must help maintain the “views” of the system. For example, network admins must be able to describe what functions are allowed and not allowed as part of the dynamic operation of the system.

There is a Dynamic Feedback Loop that Affects the Clouds Services

Since the cloud is dynamic and has models within which to operate, a dynamic feedback loop allows for constant refinement of resources and services based on the conditions or context. This includes the policies and models related to the services.

OpenDI Releases and Realization

OpenDI is being developed in three “releases” or areas of focus. Note the diagram below – the areas of overlap are necessary for the OpenDI vision, a typical negative feedback system describes within cybernetics compute research years ago.

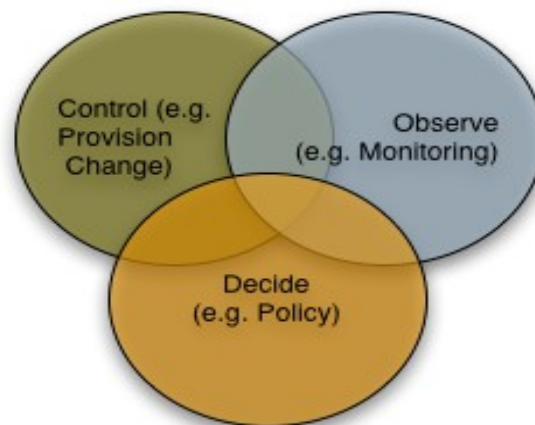


Illustration 4: Releases Modeled after Cybernetic Feedback

However, to meet our needs and time to market, we are developing a flexible architecture that can address all of these areas, and focusing on “releases” of functional capabilities – focused on the areas highlighted in the circle.

- 1) Control – the modeling and centralized control (and reference adapter) implementation
- 2) Monitoring – advanced monitoring and direction on actionable events, correlations, etc
- 3) Policy – centralized yet distributed policy management, and direction around policy implementation

We will cover all these areas in this document. However the bulk of the development efforts are focused on 1) at this time.

Cloud Architecture Concepts

There are two core issues when trying to even describe what a cloud is:

- 1) What level of abstraction (the service) is provided to the “user”?
- 2) Who is that user?

In the example of Ning:

Social network developer – interfaces with Ning via the “control” APIs to define his/her platform (a social network)

Social network user – that may not even realize (or care) that Ning was used to create and host the environment.

Both represent constituents in the cloud, but they are entirely different in the way they view and use the system.

So why is this important? It illustrates the two primary platforms that make up any cloud (or really any service as defined by SOA.)

- The Service Platform (what the user connects to in the Ning example)
- The Executive Platform (what the developer connects to in the Ning example)

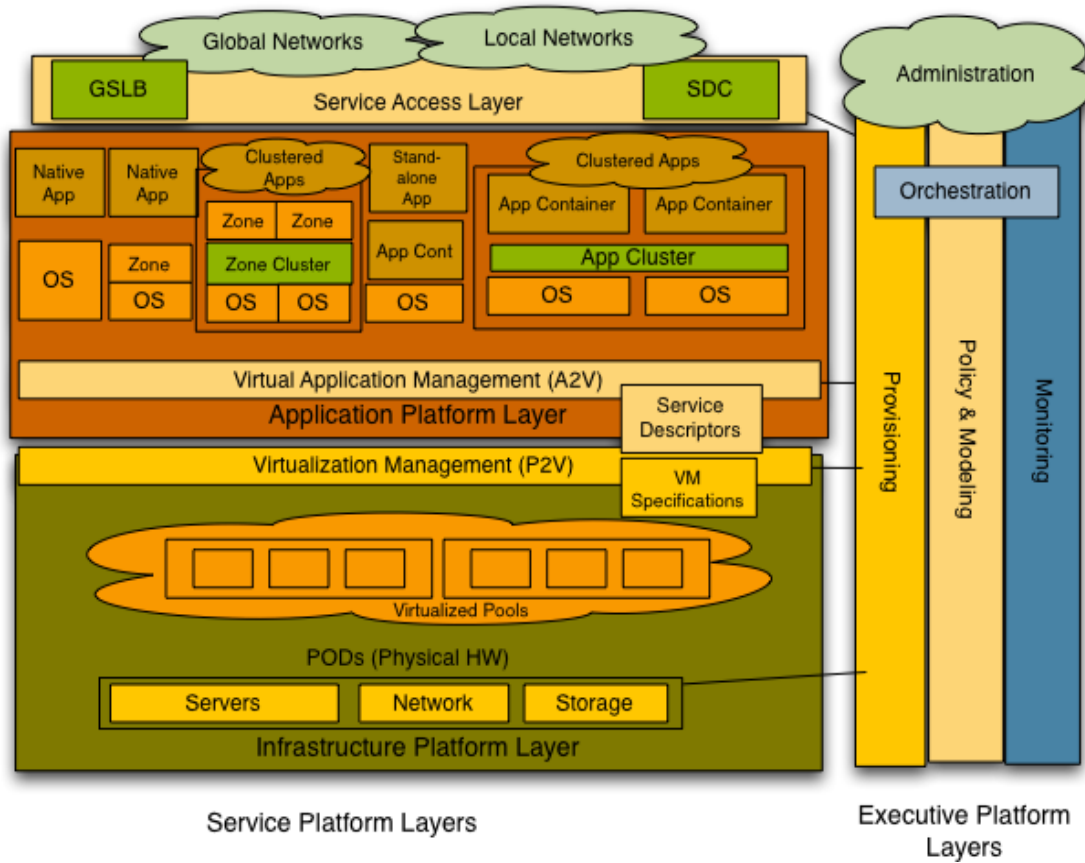
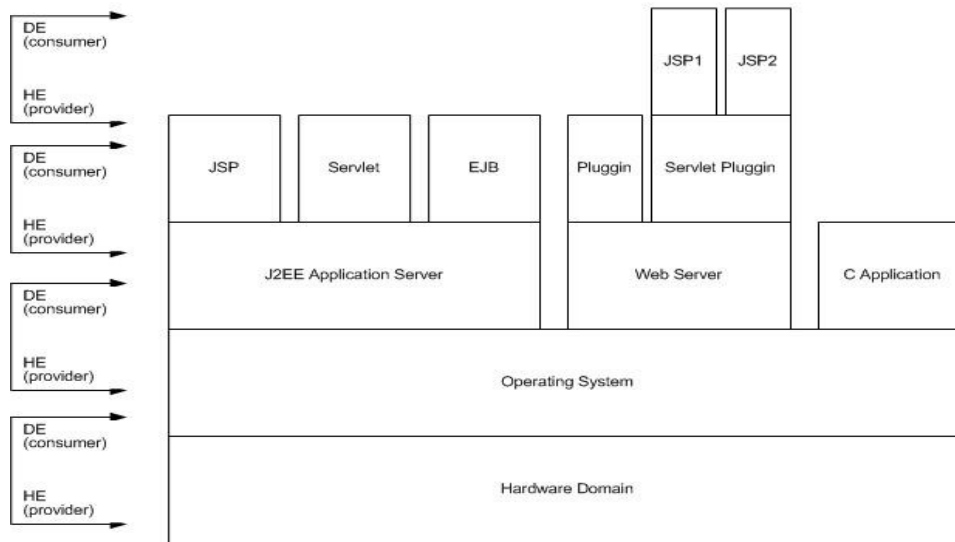


Illustration 5: Service Platform and Executive Platform

Some definitions:

Stack: A set of dependent services that are required for a running business service. Typically something like this... (from SCDE)

Runtime Services Operating Stack



Tools: COTS and other homegrown products/scripts/etc that provision, change, report, etc. (e.g. N1 SPS, BMC Patrol, xVM Ops Center) Note not all tools are created equal. Some tools (xVM OC) are a gateway to an entire compute platform that deal with additional integration and help simplify the layers.

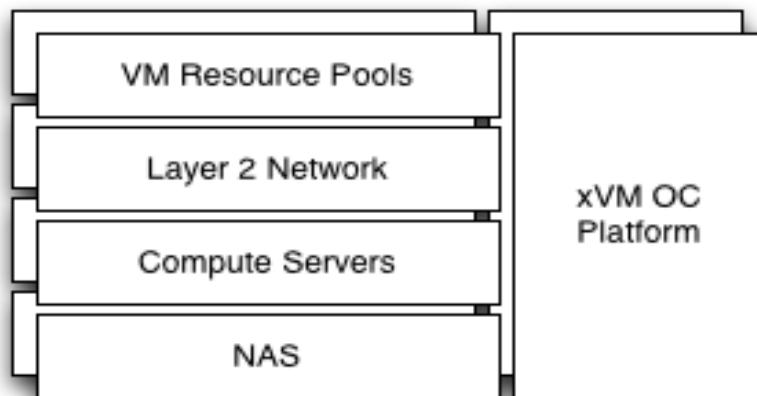


Illustration 7: Tools and Platform Example

Layers: Layers provide functionality in the platform, either exposing services for consumption or providing a mechanism for control, monitoring, etc. In the Services Layers, each can be viewed as part of a stack providing consumers and providers of resources. The Executive Platform Layers provide management capabilities of the Services Layer.

Executive Platforms sometimes are associated with a specific layer, but others overlap. For example, an intrusion detection system might monitor virtual instance, a Layer 2 switch, and a application network switch.

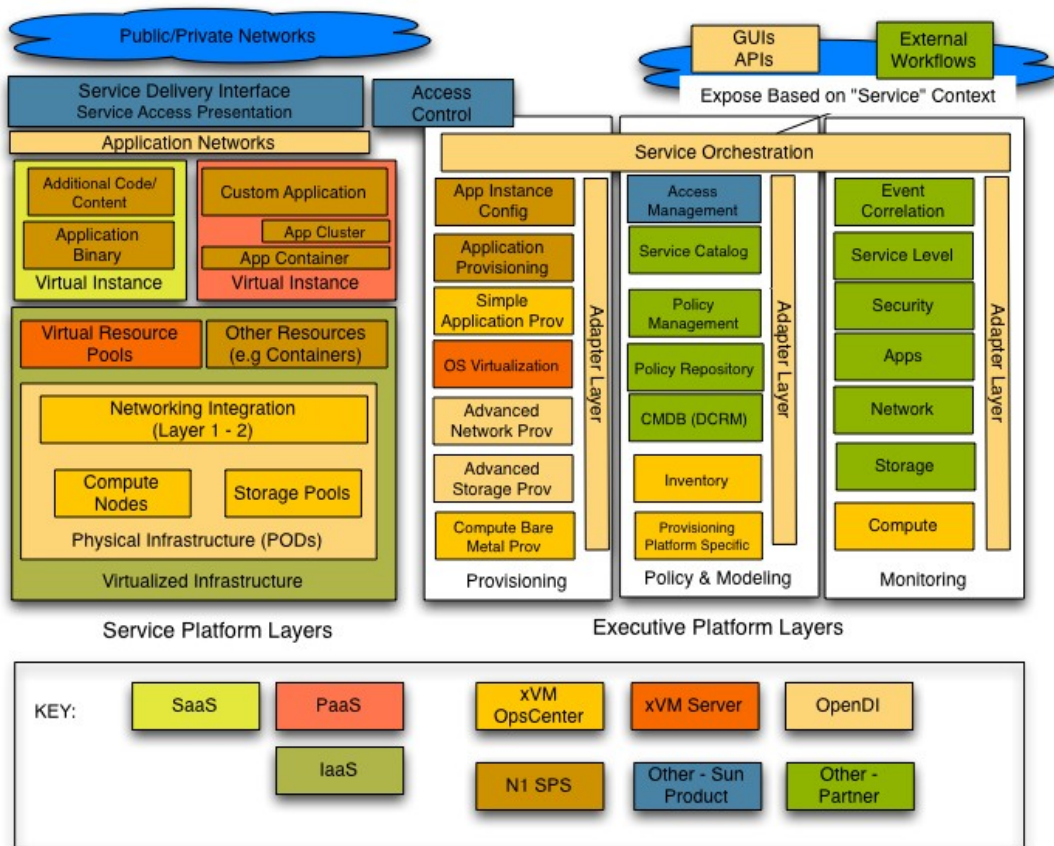


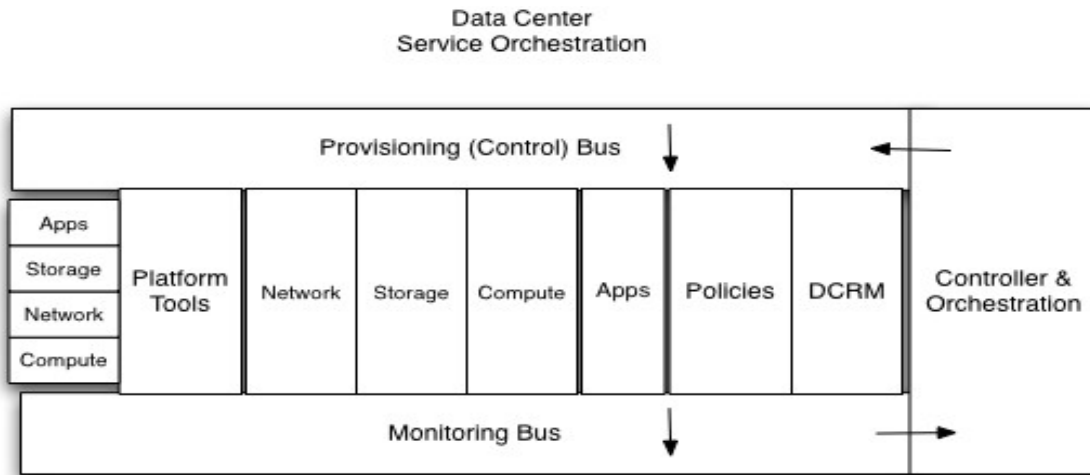
Illustration 8: Layers and Functional Areas

Executive Platforms – Multi-layer Orchestration Framework

These platforms are familiar. For the IT manager, service platforms might be xVM or VMWare or other technologies that help provide compute, storage, and network resources for consumption. These are the providers to the consumers (users) in this case.

The control platform is also familiar. It represents the technologies that IT managers use to monitor, deploy, and otherwise control the systems they manage. Examples could be Tivoli from IBM, OpenView from HP, BMC products, Sun's xVM OpsCenter, etc.

What is missing is orchestration. Today this is typically done by several humans, some disjoint tools that help filter and provide better visibility into the systems, and some manual and some limited automated scripts – all executed mostly by manual process and triggers, and some formal and some informal processes and procedures that evolve over the lifetime of the services and the entire data center operation.



NOTE: Arrows are conceptual illustration of "feedback" loop

Platform Example...

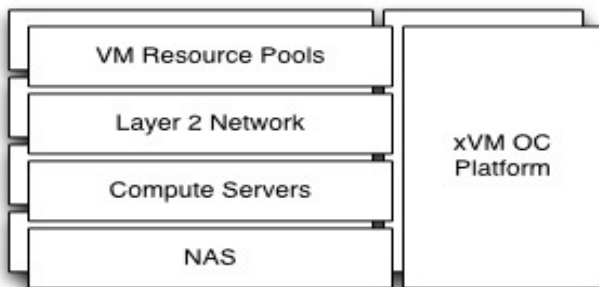


Illustration 9: Orchestration

The cloud requires orchestration across these boundaries and layers. In the example of Ning – the developer must be able to request a social network environment. This is hosted on storage, networking, and computer systems. There may be different levels of QOS associated with this request – if its just an early demo environment, then the need for resources are less than a full fledged social network that is global with thousands of users. The resources provisioned (and further grown to meet the needs, or shrunk) are essentially orchestrated via complex scripts.

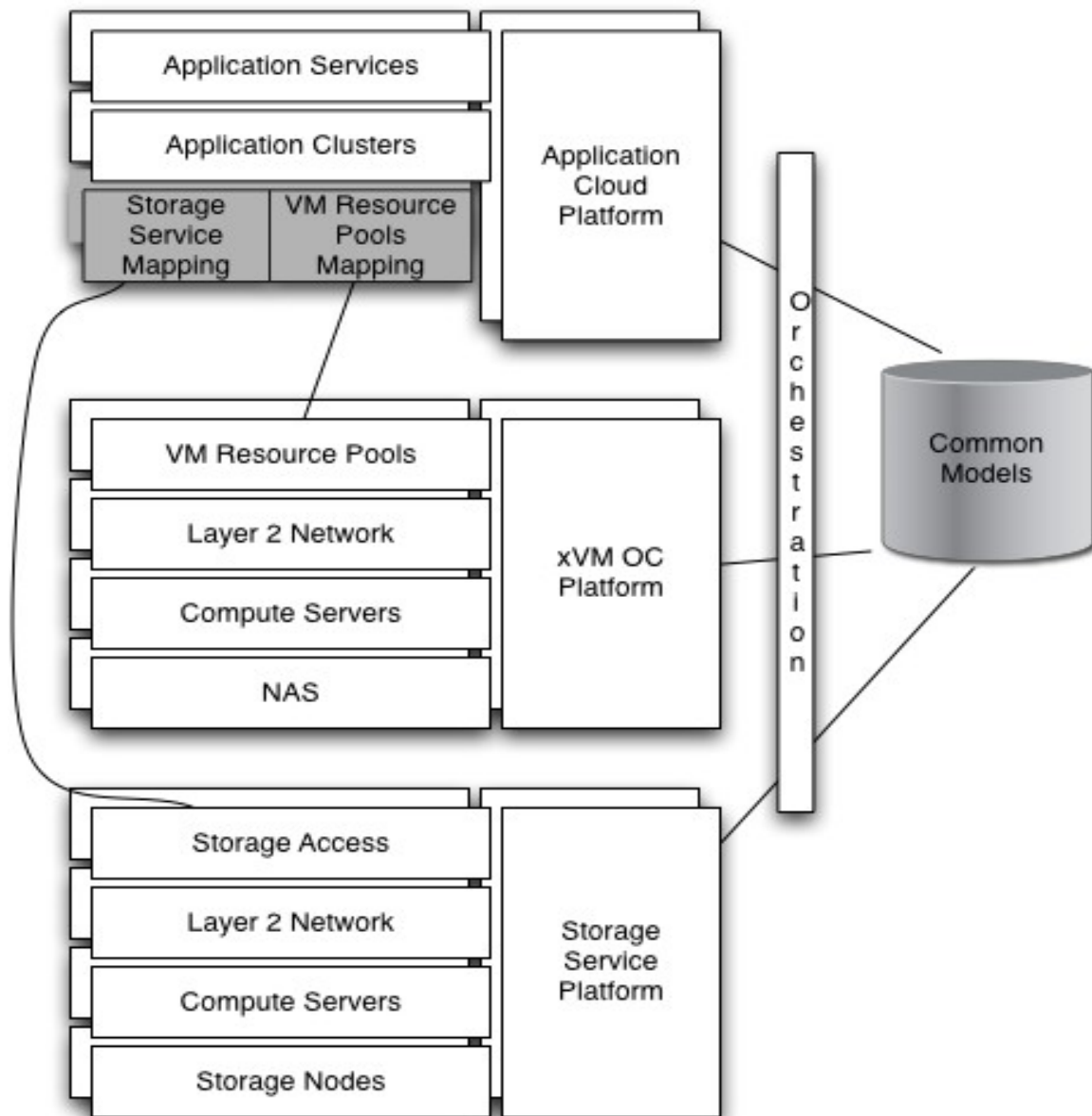


Illustration 10: Multi-Platform Orchestration

In the above example, an application service is layered on two other major infrastructure components – the compute systems and application storage, provided by Fishworks (The storage service platform.) Since all three of these systems are orchestrated via a single orchestration bus and a common model, very complex services and service relationships can be handled.

This is a key difference in the OpenDI approach. Utilizing technologies based on Java, such as OpenESB, it provides a centralized yet federated system for control, policy, transport, and abstraction of the underlying technologies necessary for these types of operations. Further, it doesn't stop at provisioning. The system can be used to react to other events....

- 1) load and capacity changes (e.g. service needs more capacity)
- 2) security issues (e.g. someone hacked a service instance)
- 3) operational issues (e.g. taking a system out of service for maintenance)

In the past, this was done generally by point to point integration, human operators running a script or performing other work at a command line, or even a multi-day process that is manually coordinated across several people.

OpenDI can take these actionable triggers, consult policy, and perform the necessary changes to the environment. It can interoperate with human workflow systems ensuring consistent process and change management policies.

Legacy Integration

Another challenge in the emerging enterprise cloud is legacy integration. The old tools and processes won't go away entirely by a new cloud orchestration platform. Changes of a “SOE” and service stack, common devices such as switches and storage servers, will still require human intervention and occasional outages.

Some example of these tools:

Monitoring: Preferred system and application monitoring platforms (e.g. BMC)

Enterprise CMDB: Enterprise asset tracking and limited change management (often reporting structure visibility)

Human Workflow: Change management, ticketing, trouble shooting, etc. (e.g. Remedy)

OpenDI must be able to provide an integration strategy across these tools and provide for acceptable interoperability, reporting, authorization, and other critical functions.

Model-driven and Actionable

Changes

Changes are modeled – before they happen. For example, if a service is allowed to grow from 10 to 100 instances, there is a model that provides the necessary structures, data, and process to make that happen. There is a policy defined that controls how much a service can grow, what it can be deployed upon, etc. The system can utilize external reporting systems to provide alerts when resources are becoming scarce, and integrate with other monitoring systems that can provide feedback and ultimately actionable triggers.

Cloud Modeling

There are several models utilized within OpenDI and the DI approach.

- 1) Architectural: Provides the higher order rule sets for architecture governance, provides guidelines and constraints within the overall system. This model is more static than the rest. An example of this is the “Model maps” within the OpenDI Release 1 system or the SDNA model.
- 2) Service: Define the business service and its components within the architectural models. Includes QOS parameters that can be realized by the system. May be represented as a “service catalog” or an ontology.
- 3) Resource: Configuration and state of the resources that provide the services. Includes service configuration at run-time. Also called the DCRM or data center resource model within the OpenDI Release 1 system.

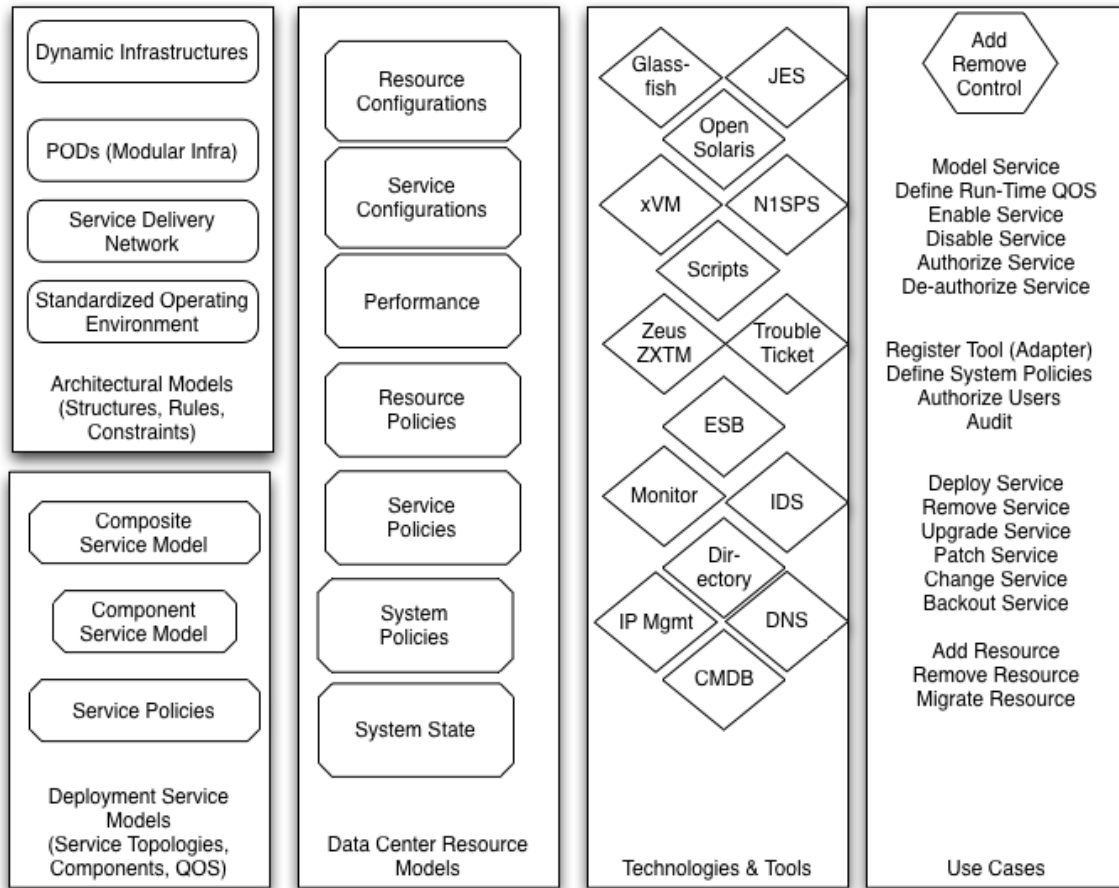


Illustration 11: Models, Tools, and Use Cases

Cloud Control and Provisioning – A multi-dimensional problem

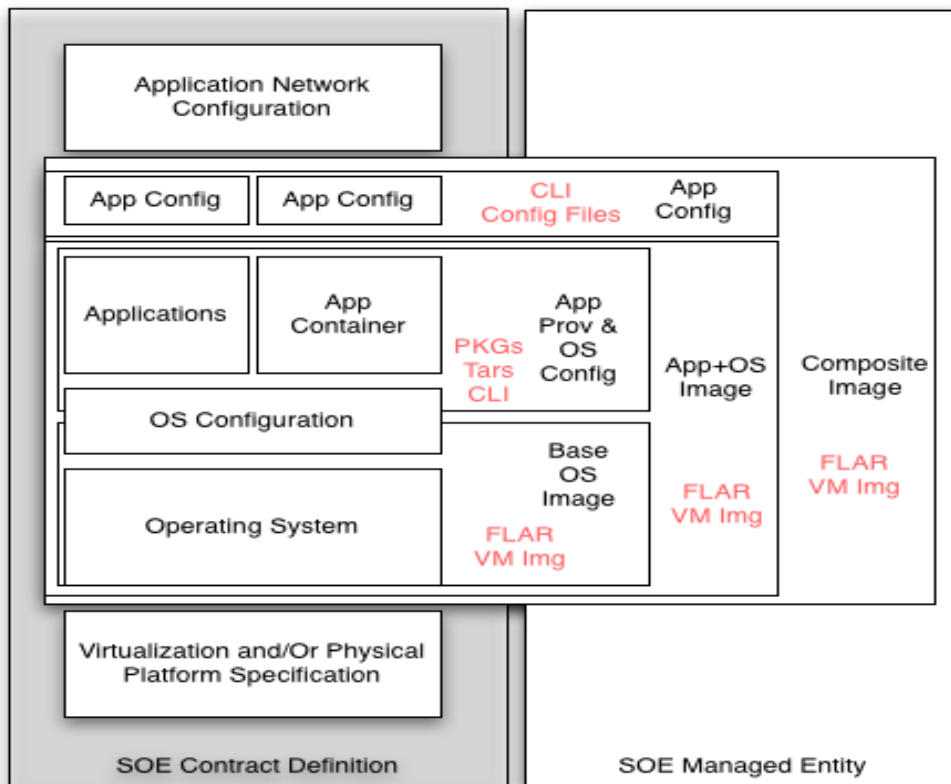
Granularity

One of the big struggling points in the "cloud" is around granularity of changes and how to facilitate them. I'm using some "OpenDI" terms in the following discussion – refer to <http://wikis.sun.com/display/OpenDI> for more info on that.

To discuss this appropriately, lets look at how systems are provisioned, using Sun's Standardized Operating Environment (SOE) as an underlying set of principles. An SOE can represent a few things....

- 1) A contract between consumer and providers that make up a service stack (in the context of computer systems.)
- 2) An image of such contract
- 3) Something else....

We are using definition 1 – the necessary components that allow are bare metal system provide a business service.



Standardized Operating Environment (SOE)
Illustration 12: Standardized Operating Environment Granularity

This illustrates how complex an SOE can be, thusly the deployment (or continuous change) of an application stack. The industry seems focused on the composite image or the image that includes the entire stack, between the network (the application network) and the hardware itself. This is in part due to the focus on OS-level virtualization. The important take away here is that this image is a composite of other entities that make up the entire stack. As you continue to move up the stack, increasing dependencies are made. As you move to the right of the diagram, the more specialized the image becomes.

There exists a “last mile” problem in this model. The final configuration of an application instance or final deployment of the application itself (for example a WAR file in a set of app container clusters) are left to other tools, not the “VM” management platform. Some tools that perform these tasks are N1 SPS or HP OpsWare, and of course, the human at a CLI (command line interface.)

Additionally, the composite model has other issues – disk space, network transport, other physics issues (time to deploy an entire 2-4 GB on a spinning disk) are problems. This is especially apparent when a “minor” change (e.g. a variable in a config file for an app server) needs to be deployed and the service restarted. One would not want to deploy 10s or 100s of complete composite images to handle this rather simple change.

Depending on the complexity of this change...

- 1) One could use an application provisioning tool to change the variable in the config file itself on the running instance and restart the service.
- 2) A bare-metal or patch provisioning tool could deploy a new config file with the new variable. Some script could be used to restart the service (or an operator could perform the task)

In both cases, the overall management system must be able to record the change so when this service is built again, it includes the new change. It also must be able to “phase out” the change, and coordinate it with the network as appropriate.

Feedback – Clouds are a Two-way Street

The services are not static and need to be able to re-act to events, for example:

- Confirmation that provisioning was successful

- A user requesting a service
- The system noticing a policy violation and responding (e.g. more capacity)
- The system noticing an intrusion and executing a strategy to reduce risk and keep the service operating.

Most importantly, these are asynchronous. They can occur at any time and are implemented in specific adapters within the OpenDI architecture framework.

A Deployment Example

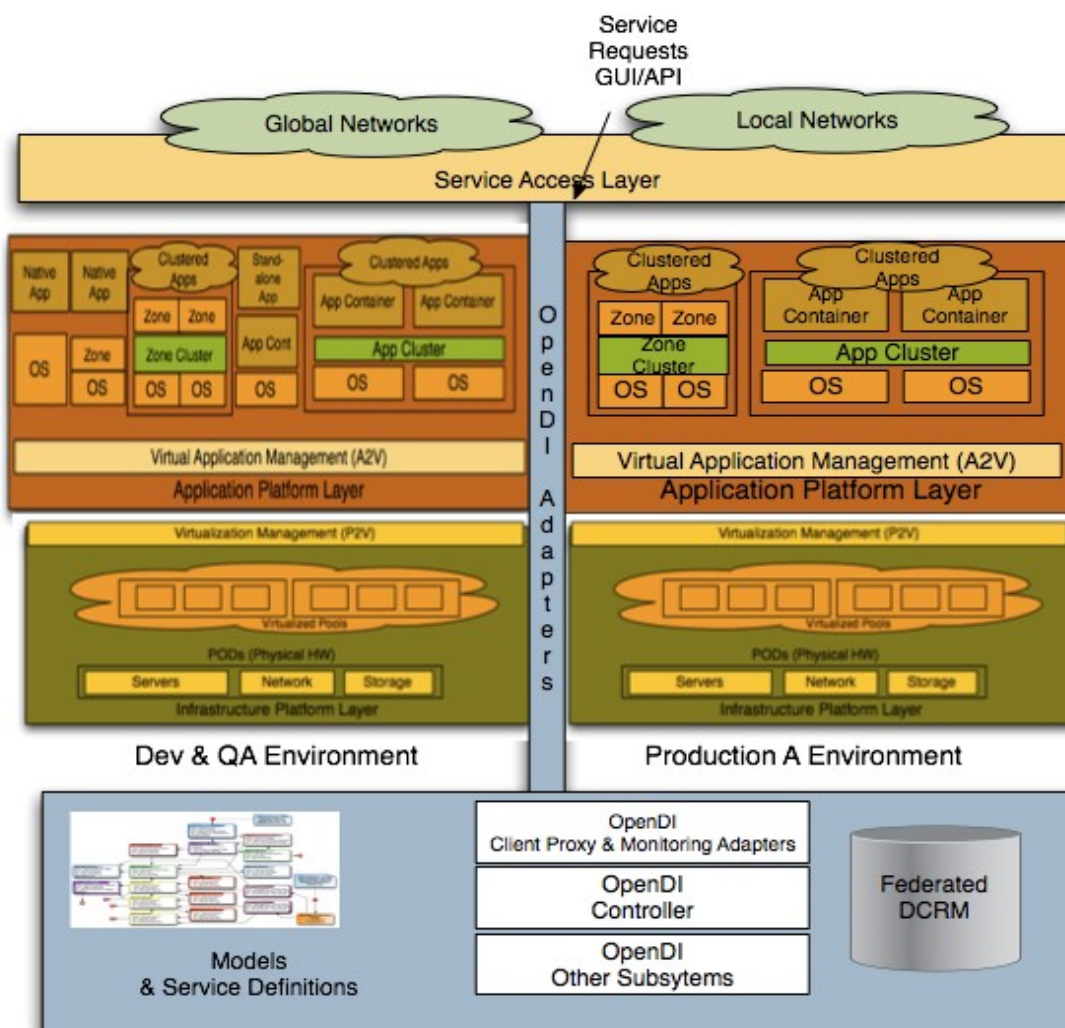


Illustration 13: Deployment Example

In the above example, there are two primary environments. Shared development and QA/Test plus Production A. Production A has limited deployment choices and supports only clustered applications since the deployment policies require high availability. Service requests are generated via a GUI and an API integrated with a development IDE which allows developers to quickly deploy applications to the dev and test environments and operations maintain higher level controls in the production environ.

Request Generator/Description	Target Environment	Detail
Developer deploys new app	Development	Via Netbeans IDE to OpenDI Adapter, facilitates file transfer and deployment coordination to app hosting environment in dev.
Developer stages new app	Production	Operations receives notification via native trouble ticket tool that new app is ready for deployment. Application must conform to the approved production deployment models (represented as service catalog items)
Operator executes change request	Production	Operations proceeds to make changes based on trouble ticket using OpenDI request GUI
Operator takes node out of service	Development	Request to take node out of service causes development services to migrate to new node. Developers notice a few minutes of connectivity issues.
Tester requests additional instances for scalability testing	QA	Requests resources via a model query and assigns the new resources, system builds new instances and deploys load generation system.

Table 1: Some Example Deployment Use Cases

Policies for the environments and resource controls have been defined to the system and are stored within the DCRM and model maps. Service changes are modeled in the service catalog and are available via GUIs, IDEs, or APIs.

The software components used in this example are described in more detail in the following overview.

Section 3

Section 3: Design Overview

Major Design Considerations

The OpenDI design is driven by some key considerations that are discussed in this section. There are many other factors influencing the design, but for the sake of conciseness, only the big ones are represented here. OpenDI is a key component of the executive platform for controlling dynamic multi-layered systems.

“Be the Glue”

OpenDI is not a standalone tool. It is a framework to connect provisioning, management and other operational tools in the enterprise, forming an “executive” platform. It does this by defining standards for a class of tool to implement. This addresses the concern that individual tools and tool implementations change over time, but the separations of concerns and the scope of responsibilities tend to be more stable.

Model Driven

All entities known by OpenDI are modeled. “Entities” include not only hardware, software, and network infrastructure, but also policies, service level definitions, and the mappings of abstract entities, such as service levels and service definitions, to acceptable concrete implementations of these services that embody the policies and constraints of the enterprise.

The model is used to understand characteristics and relationships and embody the policies, concerns and constraints of the service management stakeholders. This understanding is used to make operational decisions. The impact of those decisions is applied to the model. This provides the linchpin for service design, deployment, management, automation and control that reflects and can be traced to the operational models and concerns of the stakeholders that own the aggregated business, developer and data center management concerns.

Distributed

OpenDI is a framework for coordinating the activities of multi-vendor, multi-function tools that affect the lifecycle of technology services in the enterprise. As a framework, it should provide a set of common interfaces and taxonomy for each functional area as well as framework services to coordinate cross-function activities (e.g. task management, protocol translation and messaging).

As part of embracing a distributed philosophy, all operations are designed to be asynchronous. The framework must account for transient adapters, long-running operations, “non-blocking by default”, and other characteristics of asynchronous systems.

Enterprise Context

The OpenDI framework should be most applicable to environments that have significant size (real or virtual), and should operate efficiently in “cloud” or “superscale” distributed infrastructures as well as traditional enterprise-scale environments. The scale of these environments results in challenges related to state-awareness and resource allocation. OpenDI can be applied to smaller environments, but has a sweet spot in the very large scale.

Separation of Concerns

The design recognizes multiple types of clients of the system (administrator, solution designer, release manager, operator, analyst, etc.), and multiple technology functions (provisioning, configuration management, monitoring, reporting, etc.). Each should be reasonably protected from the others in terms of access, implementation detail, resource contention while allowing them to operate together in a controlled fashion, and to evolve their implementation without negative impact on the system as a whole.

Policy-Centric

The system should use pre-defined policy rather than interactivity. Policy should be organized in such a way that any subsystem or adapter can use it to carry out it's actions.

Controlled Transparency

Adapters and subsystems should publish information about their activities and events in such a way that other adapters and subsystems can analyze this information to refine their policies and workflows.

Basic End To End Flows

This section describe some of the major workflows of the OpenDI framework. The section is broken down by actor viewpoint. In general, all of the functionality referenced in the workflow is detailed later in the document.

Solution Developer

The role of the solution developer is envisioned to be carried out by a human actor or actors. This role defines the various solution components and represents them in the OpenDI system. There are currently three main artifacts defined for the solution developer. They are the Provisioning Scenario Descriptor (PSD) segments, the DCRM Solution Template, and the Solution Model Map (SMM).

The PSD segments encapsulate the provisioning actions that are available for a particular solution (see Provisioning Adapter section). The Solution Template represents the solution structure and operational

capabilities in the DCRM (see DCRM Adapter section). The SMM describes how the instantiated solution integrates into the rest of the DCRM (see DCRM Adapter section).

The general workflow for the solution developer is shown below. The design ensures that much of the solution developer workflow can eventually be encapsulated in an IDE to minimize the burden on the solution developer.

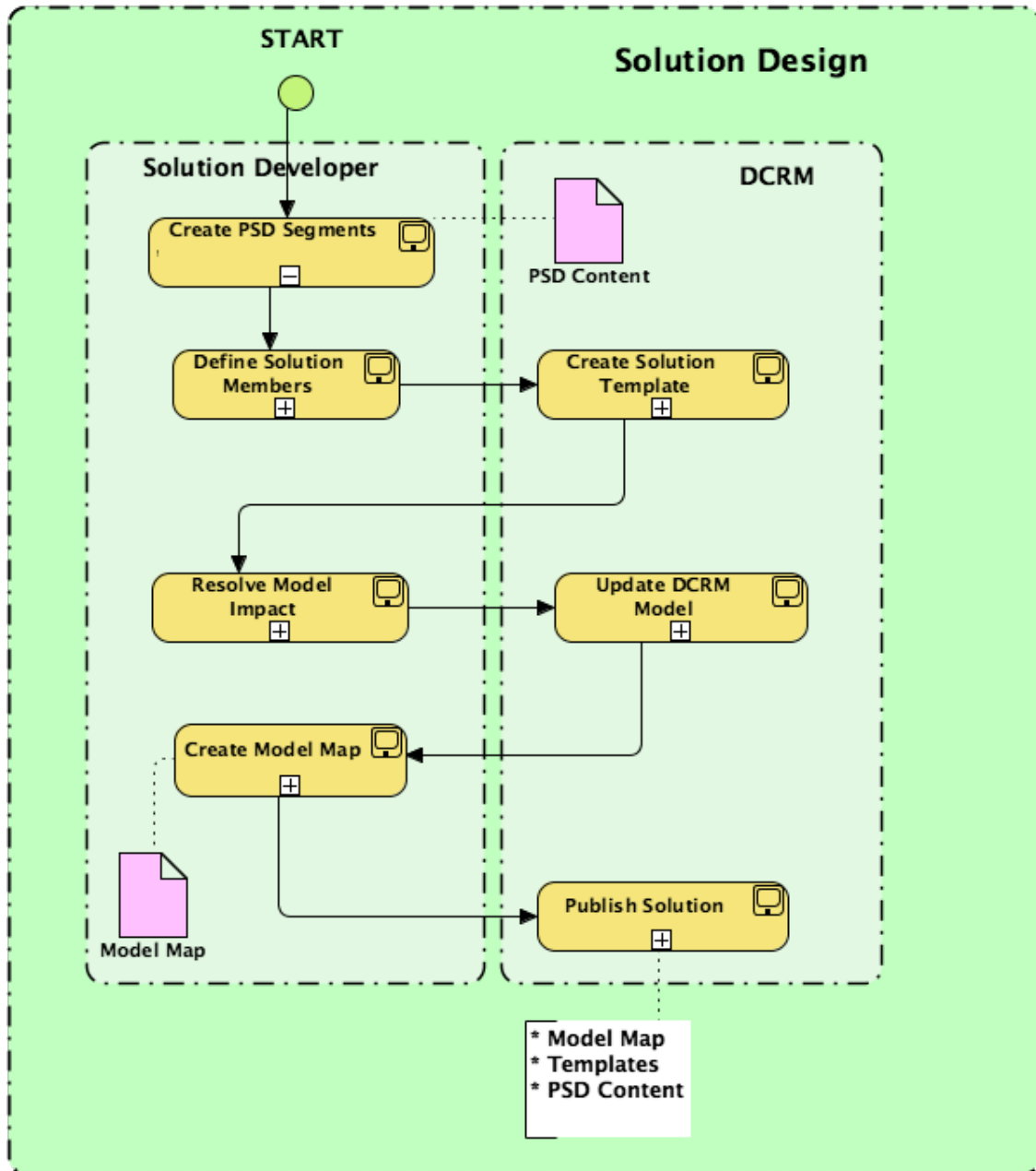


Illustration 14: Solution Developer Workflow

Operator

The role of the operator may be carried out by a human actor or possibly by another system acting as the

operator. The operator role is concerned with initiating the execution of actions upon an OpenDI solution. For example, the operator may initiate an install of an OpenDI service. The diagram below shows that high-level workflow.

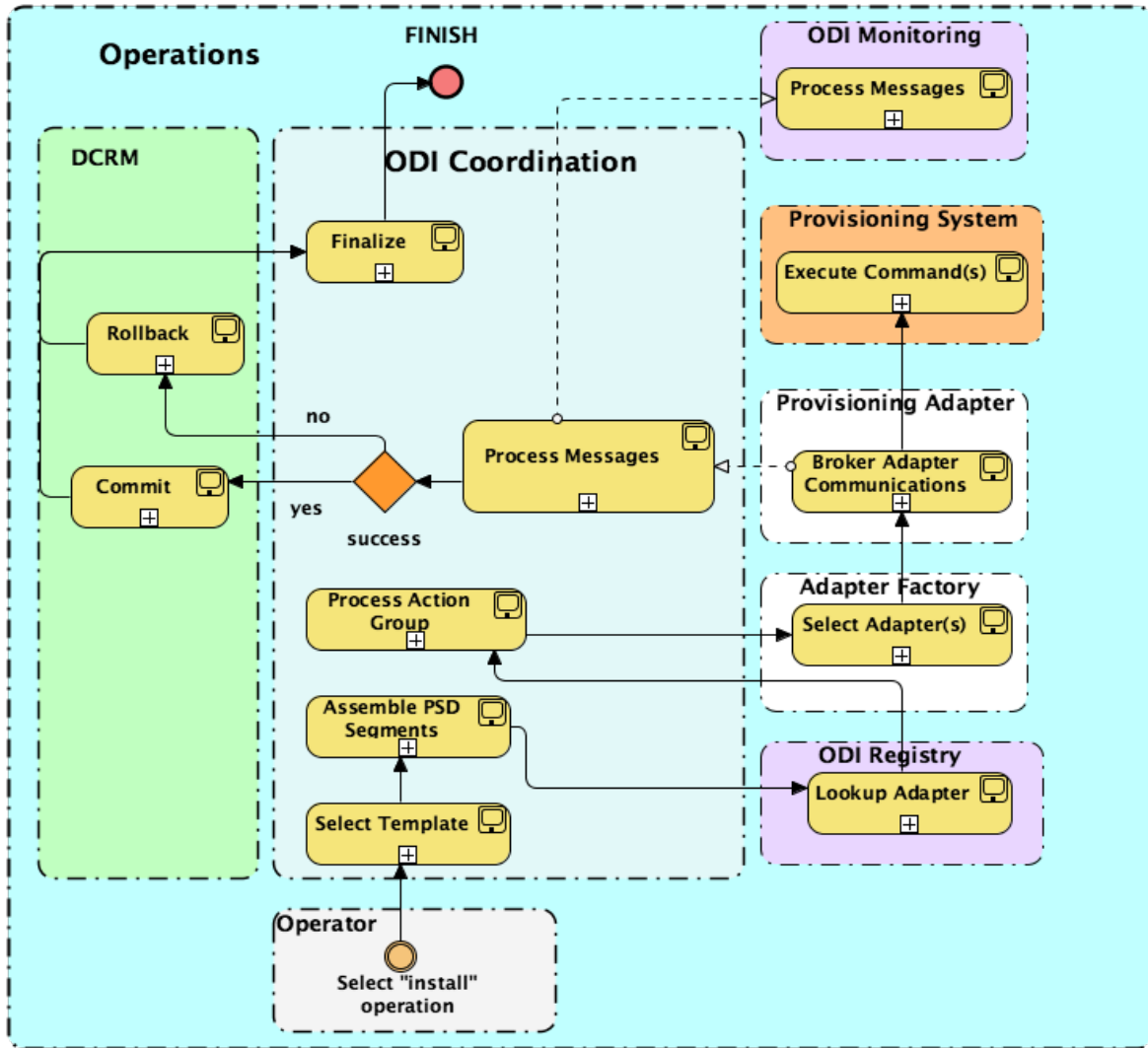


Illustration 15: Operator Workflow

Release Manager

The role of the release manager may be carried out by a human actor, or possibly by another system acting as a release manager. The function is concerned with selecting solutions for release, providing any configuration or pre-runtime values required, and defining the conditions under which the solution is to be released (e.g. execute at 11:00PM PST).

The flow is similar to that of the operator. The difference is that there is a break in the process after the assembly of the PSD segments. At this break, control is turned over to the operator and the process continues after the release requirements have been satisfied.

Administrator

The role of the administrator will be carried out by a human actor. Some of the administrative responsibilities include OpenDI user and role management, adapter management, backup and recovery, and system state management. Since there are many possible administrative actions, the flows are not presented here, however the role is important so we have represented it.

Structural Design

The diagram in this section provides a very high-level structural view of the OpenDI system. At the top of the diagram are the adapter implementations. The adapter instances cited are provided as examples. Actual adapters will vary from implementation to implementation. In addition there may be zero or more instances of each type of adapter. Adapters are generated by the adapter factory.

In order to accommodate many possible communication protocols, a communication adapter for each supported protocol is instantiated. These adapters convert communications to a common OpenDI internal communication protocol.

The front and back controller provide a point to apply inspection, security, and routing policy for traffic between the OpenDI core systems and adapters. The front controller is designated for unregistered traffic (primarily client proxies), while the back controller is designated for registered entities.

The coordination subsystem is responsible for interpreting incoming events, enforcing workflow and delegating tasks to other subsystems. To do this, it heavily leverages the task subsystem. The task subsystem is responsible for maintaining state information about all operations that occur in OpenDI. The persistence subsystem is a general purpose service provided to all OpenDI adapters, but initially it will primarily handle requests from the task subsystem. As a general purpose service, it will be available to adapters in a limited context to allow them to persist items that are too complex to fit within the confines of the task object.

The event subsystem will provide a mechanism for all adapters and internal subsystems to communicate with each other. It addresses the concepts of standardized message forms, distribution, filtering, and delivery QOS.

The diagram below provides a high-level structural view of the system¹.

¹ ESP stands for External Service Provider. This refers to products or tools in the environment that will integrate with OpenDI.

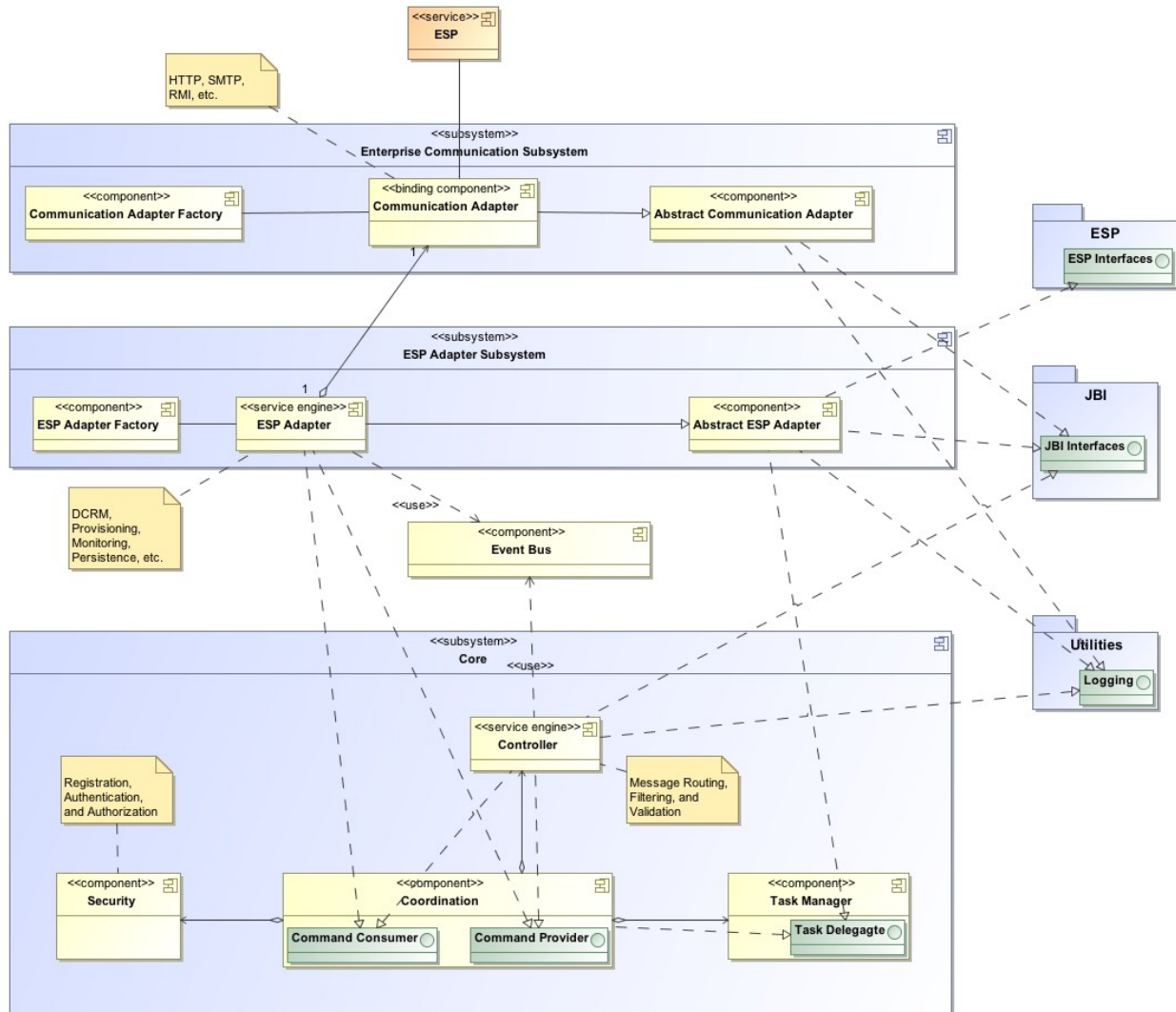


Illustration 16: OpenDI High-Level Structural View

Section 4 Major Systems

Bootstrap

The bootstrap system is concerned with initiating the OpenDI framework for the first time, or after a restart. It addresses things such as data integrity, transaction recovery, and adapter verification. Bootstrap interfaces with registration to determine which adapters to instantiate at boot time. See Registration and Factory for more information.

Authentication/Authorization

The authentication and authorization system addresses framework wide needs for access control. This system is leveraged in many places including adapter registration, user authentication, message channel subscription, and other role based access.

Adapters must be designed to run out-of-process from OpenDI proper to foster scaling and reduce synchronous dependencies between components and provide a set of mechanisms to allow for integration of tools into the OpenDI framework written in different languages with differing architectural characteristics.

OpenDI is a SOA, and all components are integrated using industry standard web service API's. The preferred mechanisms will be HTTP and SOAP, but adapter communications via JMS and other service interface designs such as RESTful services are supported.

Communications between components are secured, as appropriate, at several levels:

- Network Security and SDN – OpenDI leverages Service Delivery Network (SDN) design practices to secure and separate network traffic based on the type of traffic. Provisioning and management network traffic is always separate from public traffic.
- SSL and Encryption – network traffic exposed to significant risk from eavesdropping or unauthorized observation, such as over the Internet or between business locations, should be encrypted via SSL.
- Mutual Certificate Exchange – Adapters and OpenDI will use encryption technologies to protect the integrity of messages between adapters and OpenDI core processes. OpenDI will register the public keys of adapters to authenticate traffic from adapters, and adapters will use the OpenDI instance public key to authenticate traffic from OpenDI.
- OpenDI Specific Tokens and Handshakes – Communications protocols will include encrypted OpenDI specific tokens and handshakes that will render the theft or accidental misuse of a private keys useless without additional knowledge regarding these protocols.

OpenDI design will allow for the use of more sophisticated mechanisms, such as SAML with Security Token Services, to support federated management of more complex role-based authentication schemes for adapters deployed across multiple vendor, customer and partner installations.

OpenDI provides a framework and a set of templates for adapter design and implementation to separate adapter functional code from the OpenDI security mechanism implementation. This is intended to facilitate late binding for the choice of security mechanisms and to separate deployment from adapter design concerns.

Task Management

The task management system monitors the state of all tasks in the framework. Tasks are generally equivalent to workflows within the system. There are two types of tasks identified: “top-level tasks” and “subtasks”. Each major OpenDI operation will generate a new top-level task. Other systems or adapters may request subtasks that will be associated with the top-level task, or one of the adapters active subtasks.

There is an implied message routing relationship based on the parent of a subtask. If the parent of a subtask is a top-level task, any generated messages are routed directly to the coordination subsystem. If the parent is another subtask, the message is routed to directly to the target subsystem. The premise is that subtasks of subtasks are adapter-private operations and do not impact the coordination of a top-level task. The child subtask of the top-level task is the only subtask that coordination is responsible for.

All tasks contain a base set of information such as start time, end time, state, status, and adapter key. In addition, adapter-level subtasks can be customized by the adapter writer to track adapter specific information.

All tasks will be persisted to provide a permanent record of system activity, and to assist in transaction-level recovery in the event of a framework component failure.

Coordination

The coordination system implements the workflow and logic of the OpenDI framework. It is responsible for interpreting requests from the client proxy adapters, and satisfying those requests by invoking the various adapters in the system.

The basic end-to-end flows defined in the Overview section of this document are examples of the workflows that would be implemented in the coordination system.

All adapter traffic related to a top-level task comes through the coordination layer via the controller (assuming it is authorized and passes any filters). Any associated actions are then initiated and send out through the controller.

Controllers

The controller brokers communications between adapters or other OpenDI subsystems and the coordination system. The controller provides a centralized entry point for handling requests, invoking security services including authentication and authorization, message routing, message filtering, and detection of malformed messages.

There are three main message routes currently defined. They are:

- Adapter <--> Coordination
- Adapter <--> Adapter
- Adapter/Coordination --> Logging Channel

The first two go through the controller. Logging messages generated by adapters and by coordination do not go through a controller. They are published onto the event bus and interested (and authorized) parties can pick them up.

An interaction diagram showing basic message flow a the client proxy adapter, coordination, task management and persistence is shown below. There are similar interactions between coordination and external service provider adapters, and from adapter to adapters.

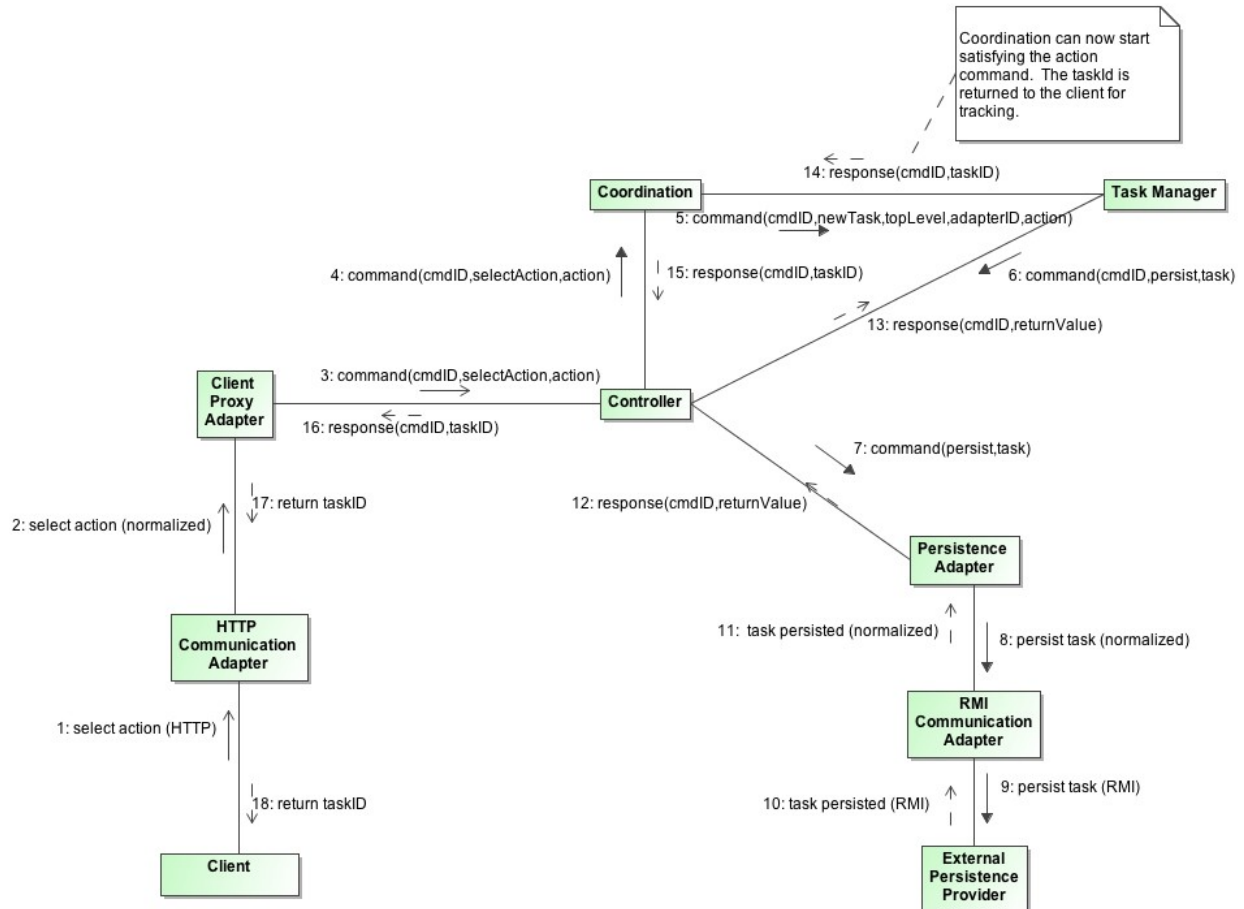


Illustration 17: Message Flow

Event Messaging

The event messaging system provides a way for all OpenDI subsystems and adapters to communicate. The system is based on a subscriber/publisher model for scalability and flexibility. Messages are broadly categorized into two groups: command messages and informational messages. As described earlier, command messages are always

handled by the controller, for filtering and routing. Informational messages are not routed, however, subsystems and adapter need to be authorized to subscribe to the information channel for a particular source.

Adapters

The adapter system allows OpenDI to be extended. There are several types of adapters known to the system. They are:

- Client Proxy Adapter
- Workflow Adapter
- Data Center Reference Model (DCRM) Adapter
- Provisioning Adapter
- Persistence Adapter
- Policy Adapter
- Monitoring Adapter
- Reporting Adapter

Each type of adapter uses prescribed interfaces that must be implemented by the adapter writer. The interfaces have been defined in general terms and are intended to be represent a common set of capabilities that many products in that particular space can implement.

Adapters are registered with OpenDI in order for them to be authenticated and used in the service management lifecycle.

Depending on the adapter type, the adapter factory may instantiate a new adapter during OpenDI bootstrap or upon first usage.

More information about registration and the adapter factory follow.

Registration

The registration system provides a mechanism to identify adapters that integrate with OpenDI. With the exception of client proxy adapter, only adapters which have been registered and verified are allowed operational capabilities in OpenDI.

Adapter registration is initiated by the administrator when a new component is ready to be integrated into the OpenDI framework. The administrator provides configuration information about the component to the

registration system, and at the appropriate time, the adapter is instantiated via the factory. The adapter state, type, and basic functionality is verified, and the DCRM model is updated to include the adapter representation.

The OpenDI adapter architecture supports a number of key requirements:

- The use of open source and industry standard integration mechanisms
- The support of multiple language and framework implementations for adapters supporting these standard mechanisms
- The support of adapters running "out of process" with core OpenDI processes, distributed across many machines potentially in differing networks, thus requiring firewall traversal and secure communications

OpenDI uses Glassfish Metro WSIT2 and the Metro XWSS3 (XML and Web Services Security) libraries (<https://metro.dev.java.net/>) to implement adapter message security and adapter communication mechanisms. Higher order integration frameworks, such as OpenESB (<https://open-esb.dev.java.net/>) may be used to facilitate orchestration of adapter processes. These frameworks rely on the underlying Metro WSIT web services security and communication mechanisms.

WSIT and XWSS as used by OpenDI support:

- Client and server implementations using Oasis compliant security mechanisms
- Integrity and non-repudiation of message exchanges
- Implementation of OpenDI adapters in different languages and running in external processes that support the Oasis WSS specifications as implemented by Metro

The OpenDI registration process uses the OASIS WSS X509 Token Profile 1.0 to secure messages between adapters. The process requires the exchange of public keys between adapters that are configured to provide and consume interfaces. The OpenDI adapter implementation uses Web Services Interoperability Technologies (WSIT) to promote interoperability of OpenDI with adapters supporting this mechanisms for its interfaces.

Adapters functions as both WSIT web service clients (consumers) and services (providers) in the OpenDI

2 <https://wsit-docs.dev.java.net/releases/m4/ClientSecurity5.htm>

3 <https://xwss.dev.java.net/servlets/ProjectDocumentList?folderID=7894&expandFolder=7894&folderID=0>

framework to enable asynchronous communication between the adapters and the OpenDI core. Although the typical communication and "wire" protocols are HTTP and SOAP, WSIT and SOAP support transparent substitution of JMS for HTTP where appropriate. SOAP also supports the routing of messages through intermediaries through the use of the "agent" attribute in the SOAP header, while maintaining message payload security.

WSIT also supports SSL encryption of traffic between service endpoints, over and above the underlying encryption and security mechanisms for the SOAP payload.

Since WSIT supports token Oasis WSS SAML Profile, OpenDI may be configured to support adapter authentication through federated identity mechanisms, such as OpenSSO, Sun Java System Access Manager and Sun Java System Identity Manager.

An interaction diagram of the adapter registration is shown below:

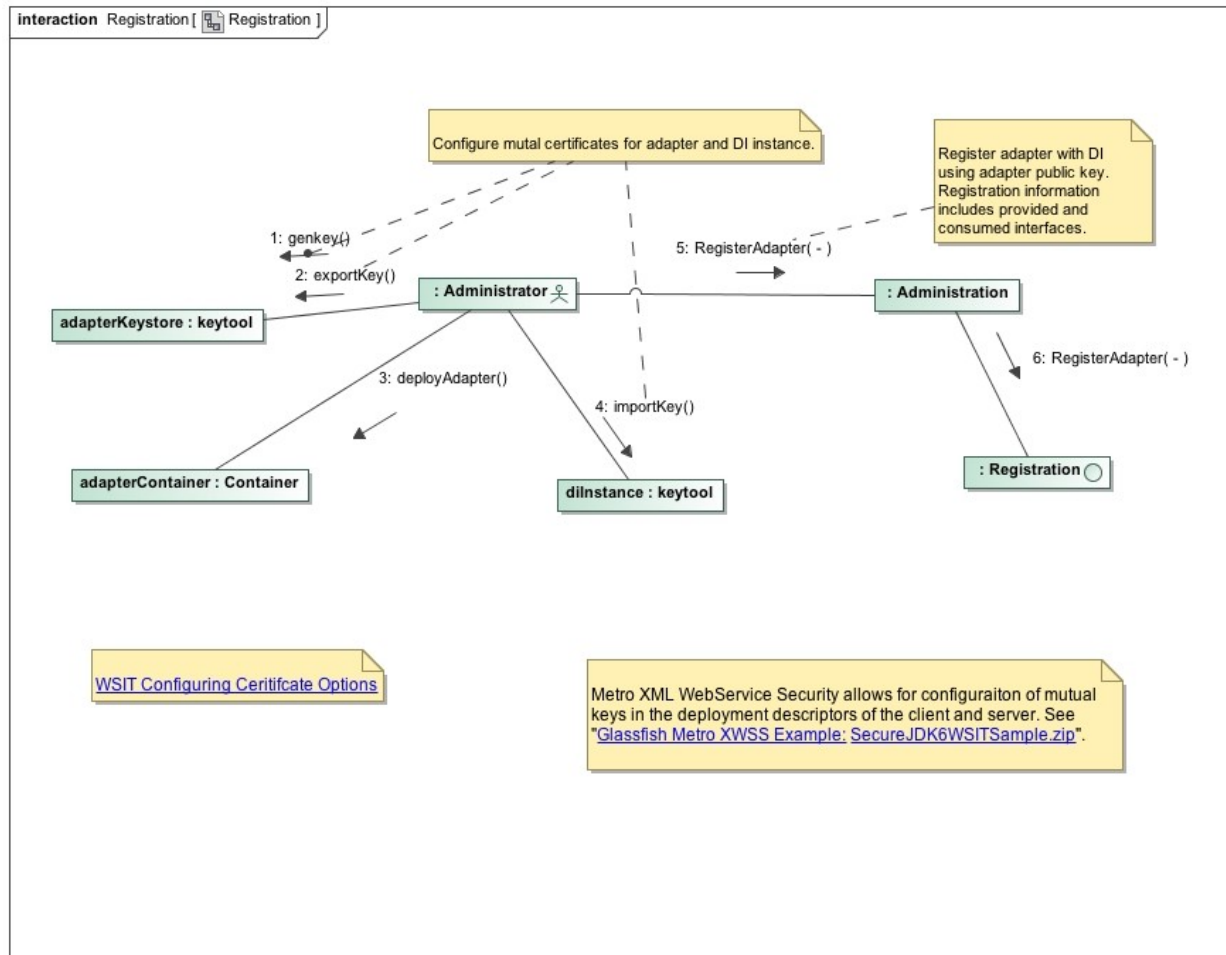


Illustration 18: Adapter Registration

Factory

The adapter factory is used to instantiate new adapters within OpenDI. The factory allows adapters to be instantiated through a common interface without knowing about any of the adapter dependencies. The registration system provides information (e.g., adapter bootstrap class) to allow both dynamic (runtime) and static (bootstrap) instantiation.

Types

Client Proxy

The client proxy adapter functions as the interface for externally influencing the OpenDI framework. There are several roles or viewpoints represented here, and they each have their own interfaces.

The administrator interfaces are equivalent to “root-level” authority within the OpenDI framework. This would include user/group management, job control, security policy, adapter registration, and other administrative tasks.

The solution designer interfaces are designed to facilitate an IDE that brings together the definition, and testing of solutions that will be used in the framework.

The operator interfaces are primarily related to initiating coordinated tasks and watching the progress of those tasks. Operators have limited visibility, and are generally only allowed to see the tasks they have initiated.

The release manager role has a solution developer facing aspect and a operator facing aspect. The solution developer does not release solutions directly into the production OpenDI framework. Instead, she indicates that a new solution is available, and the release manager deploys the solution to the production environment. With respect to the operator, the release manager may stage a task to be executed at a later time by the operator. Staging includes identifying and allocating any resources that the task will need, and indicating the criteria under which the task can be initiated.

The analyst role is related to inspecting and analyzing information maintained by the framework and it's adapters for the general purpose of improving business value.

Of these roles, it is practical that all but the administrator and solution developer could be performed completely or in part by an automated system.

Workflow

The workflow adapter is responsible for interfacing various types of workflow systems to the OpenDI framework. Depending on the workflow tool that the adapter is integrating, workflow may encompass both automated workflows such as BPEL4 specifications and human process such as BPEL4People⁵ or WS-HumanTask⁶ extensions to BPEL.

Solution developer interfaces should address the creation, replacement, retrieval, and deletion of workflows. Operator interfaces must address the execution and progress reporting of workflows if they are exposed as top-level tasks.

4 See <http://en.wikipedia.org/wiki/BPEL>

5 See <http://en.wikipedia.org/wiki/BPEL4People>

6 See <https://www.sdn.sap.com/irj/sdn/go/portal/prtroot/docs/library/uuid/a0c9ce4c-ee02-2a10-4b96-cb205464aa02>

In addition, the coordination subsystem or even other adapters may be able to maintain their internal workflows via one of the instances of a workflow adapter. In this case, the adapter or coordination subsystem is acting as a privileged operator, or solution developer.

The generalization of workflow is based on a subset of the BPMN definitions⁷, and includes:

- Flow Objects – events, activities, and gateways
- Connecting Objects – sequence flow, message flow, and association
- Artifacts – data objects, and annotations

DCRM (Data Center Reference Model)

A DCRM is responsible for providing a consistent view of the technology and service landscape as well as maintaining the structural and operational representation of services available for provisioning within OpenDI. It is also responsible for maintaining the state of resources used during tasks. The DCRM plays a larger role than just a persistent model storage in the sense that it is responsible for transforming an operation request into an actionable instruction set. For example, in the case of provisioning, the DCRM is responsible for determining model variables and returning an instance specific provisioning scenario document.

The solution developer interfaces to the DCRM should address the creation, replacement, retrieval, and deletion of template-centric OpenDI solutions. The release manager interfaces should address the staging and resource allocation of actions on managed entities or templates. The operator interface must provide methods for task execution, logging, and status messaging.

The generalization of a DCRM consists of the following definitions related to overall structure and organization:

- Catalog Items – abstract items that can be instantiated as part of a configuration
- Configuration Items – concrete instances of catalog items
- Associations – directional relationships between two configuration items
- Environments – containers for configuration items
- Templates – groups of configuration items that can be “deployed” as one to an environment
- Attachment – a user supplied file that can be associated with a configuration item

There are also definitions of structure for OpenDI integration:

⁷ See http://en.wikipedia.org/wiki/Business_Process_Modeling_Notation

- Harness Manager – a configuration item that represents the root of all adapter-specific integration into the DCRM
- Adapter Harness – a configuration item that represents an adapter-type specific integration point with the DCRM (e.g. Provisioning Harness)
- Managed Entity – the DCRM representation of an entity under OpenDI control. There are four types of managed entities: workload, payload, configuration, and target
- Action Specifier – the DCRM representation of an action that can be performed on a managed entity. The following action specifiers are identified: install, uninstall, upgrade, downgrade, control and verify
- Adapter Instructions – the DCRM must also be able to maintain adapter specific instructions related to the action specifiers. These DCRM adapter is responsible for populating values and returning these instructions when actions are called.
- Model Map – the model map is a document that describes the relationship of a DCRM template to the rest of the DCRM model. It defines the dependencies, new associations, and attribute updates that are necessary to integrate a new instance of a template into the environment.
- Provisioning Scenario Descriptor – the provisioning scenario descriptor (PSD) is a document that is returned from the DCRM during a top-level task (such as install, uninstall, control, etc.). The PSD contains the location of and instructions for all provisioning events required to execute the task.

A DCRM task has two phases: initiation and completion. The initiation phase is responsible for processing the Model Map and returning a PSD. The completion phase is responsible for committing or rolling back changes to the DCRM based on the success or failure of other OpenDI tasks related to this top-level task.

The adapter is also responsible for updating the task status per the contract with the coordination system. The adapter can also broadcast logging messages to other subsystems that subscribe to its channel.

Provisioning

A provisioning system is responsible for executing some task that does one of the following operations: install, uninstall, control, upgrade, downgrade, or verify against a DCRM template or a managed entity.

A provisioning adapter is responsible for wrapping a provisioning system and exposing interfaces for operations, and possibly solution design.

The solution developer interfaces to the provisioning adapter should address the creation, replacement, retrieval, and deletion of template-centric OpenDI solutions. The operator interface must provide methods for task execution, logging, and status messaging.

A provisioning task receives the “provisioning instruction” section of a PSD and is responsible for executing that

section. It is also responsible for updating the task status per the contract with the coordination system. The adapter can also broadcast logging messages to other subsystems that subscribe to its channel.

Persistence

The persistence adapter is responsible for maintaining object state for the OpenDI framework. It is generally concerned with state that needs to be maintained across tasks, but could be used within a task depending on implementation. In addition to providing persistence for the task management system, it may be general service available to all other OpenDI systems and adapters.

As a general service to adapters, it will still only be available in the context of supported actions. The design goal is to avoid adapters implementing their own persistence strategies when the task persistence capabilities are insufficient.

Policy

The policy adapter is responsible for managing information that supports automated decision making in the OpenDI framework. There are two main categories of policy identified in OpenDI: authorization policy and obligation policy. The policy system is a general purpose system accessible by other solution designers, the framework, and adapter developers. It provides a centralized repository for all policy affecting the ecosystem.

Much of the design for the policy system is based on the research published for Ponder and Ponder2 by the Imperial College of London⁸.

For more information on the generalization of policy in OpenDI, please refer to the OpenDI detailed design material.

Monitoring

Monitoring adapters in OpenDI can range from product specific monitors to large monitoring frameworks. The decision to create a monitoring adapter primarily depends on how the data coming from the system will be used. If the data will be used to provide analytics information to the OpenDI framework, then it may make sense to write a monitoring adapter. On the other hand, if the goal is only to provision monitors within a system as part of a top-level task, then it may be more appropriate to use an existing provisioning adapter, or create a new provisioning adapter for this monitoring system.

A monitoring adapter implements all of the interfaces of a provisioning adapter since the monitor must be

8 <http://ponder2.net>

provisioned. It also implements interfaces for logging of triggered events. When a triggered event is logged, interested parties can determine how to react based upon their policies related to the source of the event.

Reporting

Reporting adapters are intended to provide access to aggregated information. It is not the intent of OpenDI to replace point reporting solutions. The definition of the reporting adapter generalizations is still a work in progress.

Communication (Internal)

The communication adapters are responsible for translating communications between the external service provider and the core OpenDI framework. External service providers interface with a communication adapter that supports its native protocol. The communication adapter translates the information to a standard protocol used within OpenDI and passes it to the adapter associated with the external service provider. From there, the adapter can interpret the information and take the appropriate action.

Appendix 1

OpenDI/JBI Integration Requirements and ESB Analysis

John Kirby – Oct 2008

JBI (Java Business Integration) spec is a Java-based standard that defines a runtime architecture for plugins to interoperate via a mediated message exchange model. It appears an ideal model for OpenDI which strives to be a highly flexible and pluggable architecture. JBI requires that it integrate with an ESB (Enterprise Service Bus). There are different implementations of ESB solutions such as OpenESB and Apache ServiceMix which all provide compliant implementations of the JBI spec.

One of the advantages of using an ESB is that it brings flow-related concepts such as transformation and routing to a Service-Oriented Architecture. ESB can also provide an abstraction for endpoints. This promotes flexibility in the transport layer and enables pluggability of services.

Messages between components in a JBI/ESB environment are mediated by the NMR (Normalized Message Router). The NMR serves as an intermediary for routing messages between plugins (JBI Components). Plugins do not communicate directly with one another ... they only communicate with the NMR. This provides location transparency for the plugins.

The plug-in components function as service providers, or service consumers, or both. Components that supply or consume services locally (within the JBI environment) are termed “Service Engines.” Components that provide or consume services via some sort of communications protocol or other remoting technology are called “Binding Components.”

Like OpenDI the goal of JBI is to allow components and services to be integrated in a vendor independent way, allowing users and vendors to plug and play.

In a nutshell, JBI (with ESB) defines how things are realized within an infrastructure.

JBI Basics

JBI components (or plugins) come in two flavors:

- BC (Binding Components)
- SE (Service Engines).

BCs and SCs are generically referred to as JBI components. JBI components are deployed to the JBI environment (Meta Container) and simply run in memory.

In addition to the components, JBI provides APIs System Management APIs to handle installation, deployment, and monitoring.

Roles of a JBI Component

A JBI component can function as a service provider, or a service consumer, or both. These roles have distinct responsibilities:

Provider:

- Activate the service provider endpoint (service name, endpoint name).
- Provide service description data (via WSDL) describing the provided service.
- Receive message exchanges invoking or continuing an on-going operation. Respond to them according to the MEP (Message Exchange Patterns) and state of the message exchange instance.
- Perform the function requested in received operation invocations.

Consumer:

- Discover the services needed (that is, find the service description). This can occur at design-time (static service dependency), or at run-time (dynamic service dependency).
- Invoke a needed operation from the service, as dictated by internal processing within the consumer component.
- Receive message exchanges and respond to them according the MEP and state of the message exchange instance.

JBI Components

Binding Components (BCs)

A Binding Component has two purposes:

1. To communicate to the world outside of the JBI environment using remote protocols
2. To normalize/de-normalize messages it receives

Examples of remote protocols provided by BCs include HTTP/S, JMS, FTP, SMTP, XMPP, RMI, CORBA, etc.

BCs are analogous to Communication Adapters in OpenDI.

Service Engines (SEs)

A Service Engine provides the business logic inside the JBI environment and only communicates with the NMR. If a SE needs to communicate outside the JBI environment, it must send a message to a BC (via the NMR). Examples of SEs include rules engines, OpenDI Controller, OpenDI Adapters (non-communication type) BPEL engines, XSLT engines, and scripting engines.

Service Units

Service Units is another overloaded term similar to a SE which are simply processes within a container such as Orchestration (BPEL), Transformation (XSLT) and JavaEE (POJOS). They describe themselves using WSDL. Current implementation standard is HTTP/SOAP but communication is not limited to this.

The WSDL contains two parts:

- Abstract Definition (Service Interface)
- Concrete Definition (Service Implementation)

The Abstract side defines the interfaces. In terms of OpenDI, this appears where we would define our interfaces. The Abstract part defines the messages (in/out), name of services, operations provided by the service and types which define those messages exist.

The Concrete side defines where the service exists and how to talk to it. As mentioned before the convention is HTTP/SOAP, but doesn't have to. This would be interesting to see how another communication protocol would be implemented?

During deployment the Service Unit's Abstract WSDL is sent to the NMR (Normalized Message Router) for routing.

Deploying JBI Components

JBI uses Ant targets to deploy JBI Components. At the atomic level deployment is made up of one or many Service Units which comprise a Service Assembly (SA). A Service Assembly is usually in the form of ears/jars and includes a composite service deployment descriptor detailing to which component each Service Unit contained in the SA is to be deployed. Note that this service assembly concept is sometimes termed "composite service description" (CSD) or in NetBeans as Composite Application (CA).

During the deployment process the Abstract WSDL interface for each Service Unit is sent to the NMR to “register” the component. The Service Unit has a name (as defined in the WSDL) which then gets registered in the NMR service table. This allows any JBI component to call another component by simply knowing its name.

NMR (Normalized Message Router)

The NMR is the router of messages within the JVM. It does not route messages to the outside. This is where the OpenDI registry might come into play being able to “extend” the NMR to route messages to a BC (or GSCN) for OpenESB and and ActiveMQ with ServiceMix.

NMR uses a table of services to route messages. Routing table resolution is via the simple name lookup contained in the abstract definition of the Service Unit's exposed WSDL requesting the service by name. Once the NMR receives the request, it packages up the request into a normalized message and routes the message to the appropriate service. The NMR is aware of what services exist (via table of services) and what container is providing that service.

Message normalization/de-normalization is the act of converting a message from/to a protocol-specific format into a format for the JBI environment. A normalized message (envelope – structured as XML) consists of the following:

- Content – The payload message (usually XML but not required). Message format needs be constructed in a way that the consumer (container) knows how to interpret.
- Attachments - attachments to the main message object.
- Security Subject - The security subject associated with the content.
- Properties - Other message-related key/value pairs.

There is no canonical message format for JBI. Messages simply must be normalized to meet the criteria above.

What JBI Is And Isn't

IS:

- It defines how infrastructure should communicate with each other **inside the same JVM**.
- With an ESB it defines how things are realized within the infrastructure.

IS NOT:

- It does not define a standard for ESB
- Does not allow persistence of message exchanges. Currently message exchanges are not serializable; they cannot be persisted for purposes of failure recovery.
- Distributed. JBI, as currently defined, is not distributed. It does not tell how to communicate across nodes (different JVM). OpenESB provides a GSCN (Global Services Communications Network) and BCs to go across nodes. Apache ServiceMix uses ActiveMQ.

JBI Version 2

The next release of JBI (v2) plans to focus on the following capabilities. This version should be part of v3 of Glassfish.

- Clustering & Distribution
- Runtime Management Enhancements

- Interceptors (Container to Container “controller”)
- Policy Capabilities (JSR 265)
- Fault Tolerance & Reliability
- Support For POJOs
- SCA (Service Component Architecture) & OSGi Alignment. There is an overlap between SCA and JBI on how they deploys things, but SCA addresses model solutions how things are realized inside integration infrastructure.
- Service Versioning
- Removal of Dependency on WSDL

Apache ServiceMix

Apache ServiceMix is an open source ESB built inline with the JBI specs. It does have some interesting differences from OpenESB and appears to be closer to JBI Verson 2 capabilities.

The core ServiceMix services come from the ServiceMix Kernel which include features such as:

- Hot deployment: ServiceMix support hot deployment of using OSGi bundles. In addition, the Kernel also supports exploded bundles and custom deployers (a spring one is included by default).
- Dynamic configuration: Services are usually configured through the Configuration Admin OSGi service.
- Logging System: using a centralized logging back end supported by Log4j, ServiceMix Kernel supports a number of different APIs (JDK 1.4, JCL, SLF4j, Avalon, Tomcat, OSGi)
- Native OS integration: ServiceMix Kernel can be integrated into your own Operating System as a service so that the lifecycle will be bound to your Operating System.
- Extensible Shell console: ServiceMix features a nice text console where you can manage the services, install new applications or libraries and manage their state. This

shell is easily extensible by deploying new commands dynamically along with new features or applications.

- Remote access: Operations on the console can be done remotely via a secured and encrypted channel. All commands accessible from the local console can be launched from a remote computer.
- Security framework based on JAAS

On top of the ServiceMix Kernel is the ServiceMix NMR which is where the JBI spec is implemented.

It utilizes ActiveMQ (JMS) or Camel to allow clustering via JMS endpoints. ActiveMQ provides remoting, clustering, and distributed failover. It also appears to resolve the cross-JBI node limitation issues. Camel allows configurable routing and mediation rules.

JBI/OpenDI Considerations/Concerns & Functional Mappings

The major concerns so far are:

- Scalability.
- Security. Control adapter functionality and message security.
- Distributed environment capabilities. How to do transparent JBI node to node communication.
- Component registration with NMR integration in a multi-node JBI environment.

Below is a table of proposed OpenDI Components/functionality and how they could map to JBI Components and implemented via either OpenESB or Apache ServiceMix.

JBI appears to be the preferred specification to allow the pluggability and scalability required by OpenDI. JBI 1.0 is addressed under JSR 208 for JBI 2.0 under JSR 312.

OpenDI Component/Functionality	JBI	OpenESB	ServiceMix
Communication Adapter	BC	X	X
Front Controller	SE	X	X
Back Controller	SE	X	X
Authentication/Authorization	SE/BE with LDAP	X	X
Message Based Encryption	Abstract WSDL/WSIT	X	X
Task Management	Composite Application/SE	X	X
Coordination	Composite Application/BPEL	X	X
Adapters (Non-Communication) <ul style="list-style-type: none"> ● Client Proxy ● Workflow ● DCRM ● Provisioning ● Persistence ● Policy ● Monitoring ● Reporting 	SE. Will also need BC for vendor specific communications with external	X	X
Registration *	NMR/SE/BC	X	X
Transparent inter-JBI Node Communication **	NMR/SE/BC	via BC	via NMR/ JMS
Event Messaging ***	NMR	via BC (JMS) across nodes	via NMR/ JMS
Security ****	Up to the Components	X	X

* Components (Service Units) are added (or removed) to NMR's Service Table during deployment (or

undeployment). Additional meta-data about the deployment will need be added to a registration service repository. This repository will need to be accessible to all JBI nodes.

** Components need to be accessible and addressable across all JBI nodes. JBI specs don't address how to do it transparently. Vendor specific implementation. OpenESB doesn't have a mechanism (sort of through BCs). ServiceMix's NMR use JMS which appears to allow cross JBI node transparency.

*** Event Messaging within the JVM (on OpenESB) is handled by the NMR via the front/back controllers. Messages across JBI nodes will need to go via a BC. In ServiceMix the NMR is JMS based.

**** JBI 1.0 provides a basic mechanism for carrying security subject information with normalized messages, leaving other security aspects (authentication of those subjects, and authorization) up to components (and JBI implementations) to provide.

Appendix 2

Useful Websites

Dynamic Infrastructures Project Wiki:

<http://wikis.sun.com/display/DI>

DI Delivery Services :

<http://www.sun.com/service/dynamicinfrastructure/>

OpenDI :

<http://opendi.kenai.com>

OpenVM :

<http://www.openvm.org/>

Other Resources

Data Center Architecture

Data Center Reference Guide: http://www.sun.com/datacenter/reference_guide_wp.pdf