Media Player

0:00

Q&A

Refresh Now

Submit

Slides

© 2013 Carnegie Mellon University

Today's Speaker

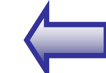
Grace Lewis
Senior member of Technical staff
Software Engineering Institute

Grace Lewis has over 20 years of professional software development experience, mainly in industry. Her main areas of expertise include service-oriented architecture (SOA), cloud computing and mobile applications.

Before joining the SEI, Lewis was Chief of Systems Development for Icesi University, where she served as project manager and technical lead for the university-wide administrative systems. Other work experience includes Design and Development Engineer for the Electronics Division of Carvajal S.A. where she developed software for communication between PCs and electronic devices; developed embedded software on the microcontroller that was used on the devices; and provided technical assistance to sales personnel during on-site visits to potential and actual clients.

© 2013 Carnegie Mellon University

Agenda: Architecture and Design of Service-Oriented Systems

Review Part 1 

SOA Infrastructure Design Considerations

Service Design Considerations

Summary

© 2013 Carnegie Mellon University

Architecture and Design of Service-Oriented Systems: Goals

Present and discuss

- Basic concepts related to software architecture design
- Impact of service orientation on system qualities
- SOA infrastructure design considerations
- Decomposition of an Enterprise Service Bus (ESB) into patterns and tactics as an example of SOA infrastructure
- Principles of service design

Provide a starting point for using quality attribute requirements to design infrastructure and services for service-oriented systems

© 2013 Carnegie Mellon University

Review Part 1

A software architecture is the earliest life-cycle artifact that embodies significant design decisions: choices and tradeoffs

Design decisions are made in the context of the architecturally significant requirements

Architectural design patterns are typically chosen to promote one or two qualities that are important to an organization

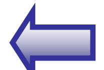
Service-orientation promotes interoperability and modifiability at the expense of performance

Service-orientation is a starting point that is often augmented by other patterns and tactics to create a complete architectural solution

© 2013 Carnegie Mellon University

Agenda: Architecture and Design of Service-Oriented Systems

Review Part 1

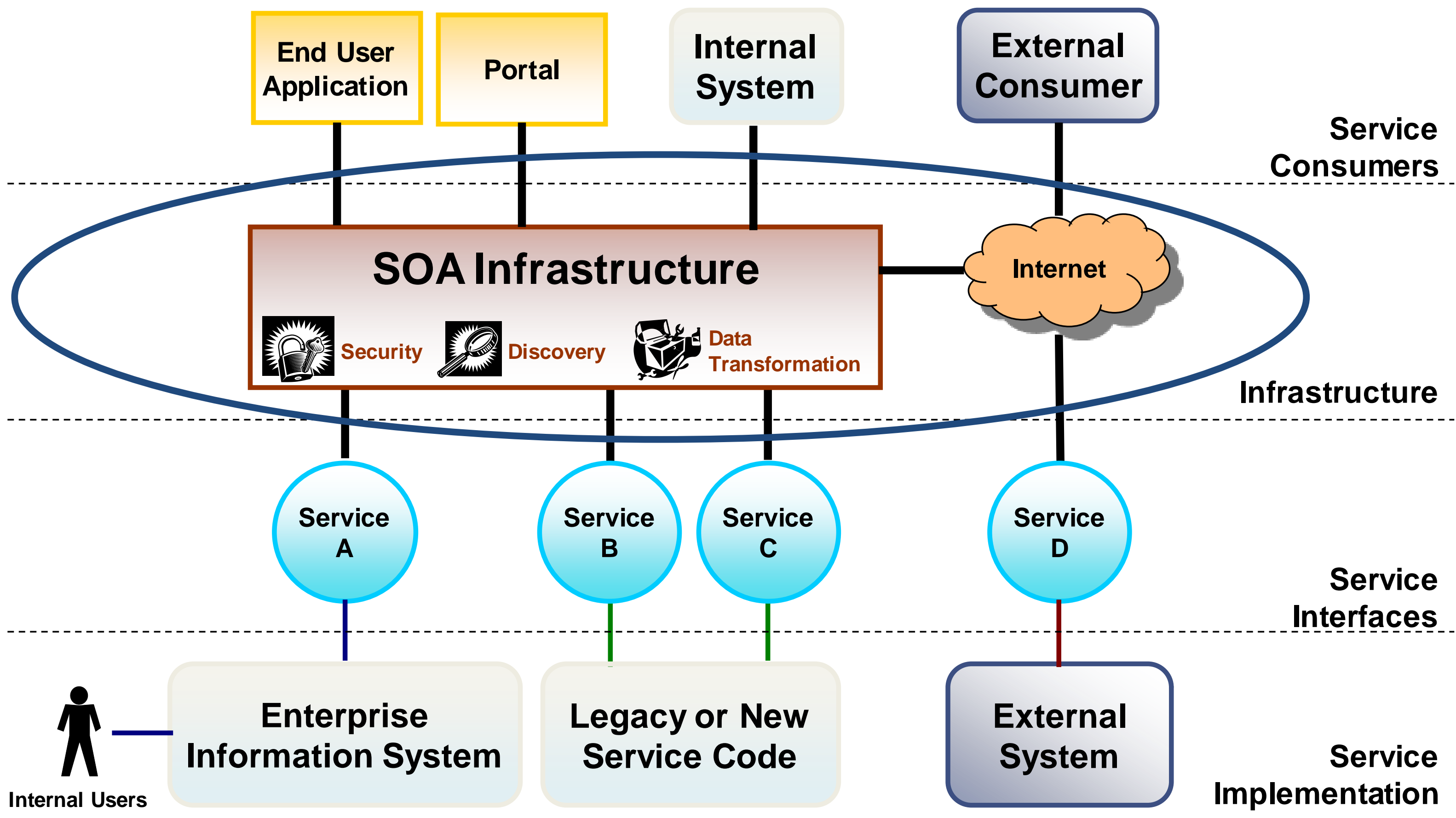
SOA Infrastructure Design Considerations 

Service Design Considerations

Summary

© 2013 Carnegie Mellon University

Focus of this Section



© 2013 Carnegie Mellon University

Integration Approach

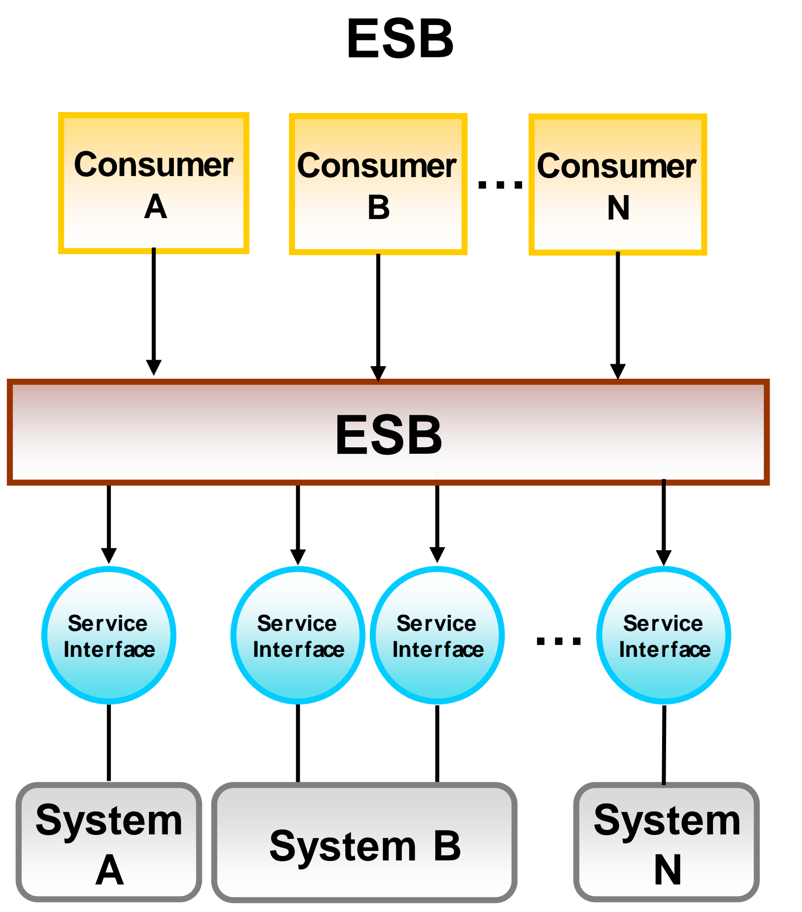
There are multiple approaches for integration of service consumers and service providers, e.g.

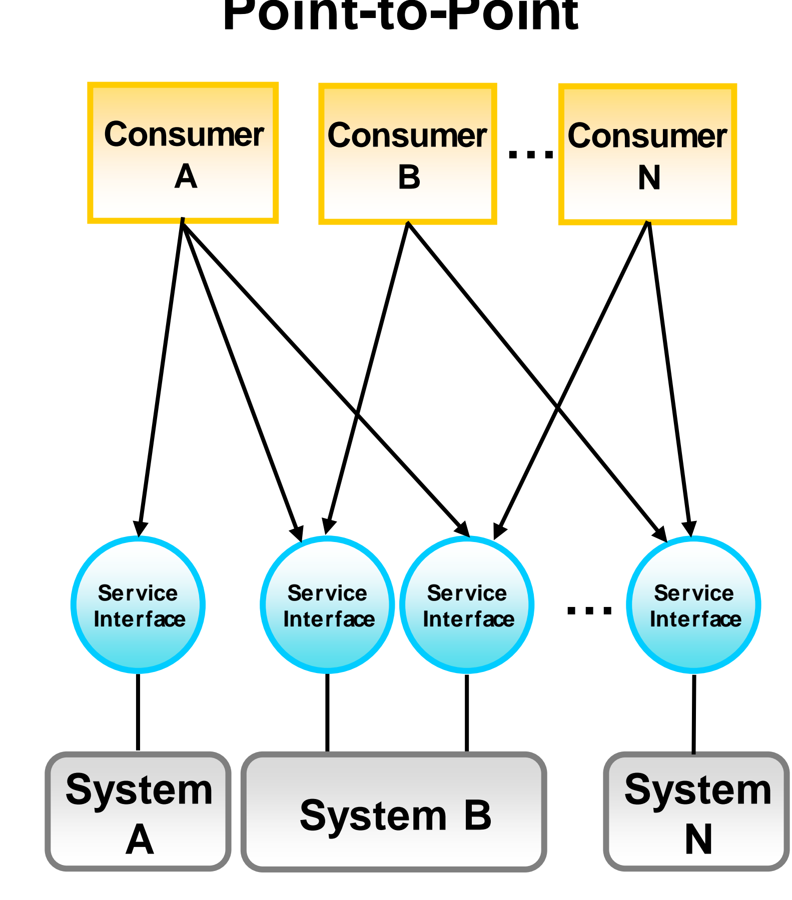
- Point-to-point
- Hub-and-spoke
- ESB (Enterprise Service Bus)*

* NOTE: Some ESB vendors contend that ESB is not hub-and-spoke. However, ESB is a logical hub-and-spoke topology where components may be distributed to eliminate performance bottleneck or single-point-of-failure (SPOF).

© 2013 Carnegie Mellon University

ESB vs. Point-to-Point





Source: Bianco et al. Evaluating a Service-Oriented Architecture. Software Engineering Institute.
http://www.sei.cmu.edu/reports/07tr015.pdf (2007)

© 2013 Carnegie Mellon University

Point-to-Point

Services are directly connected to service consumers

Each service consumer must adapt to comply with all connected service interfaces

- Interface technology (e.g. asynchronous vs. synchronous, SOAP vs. REST, versioning)
- Business service interface (e.g. call interface including arguments, semantics, exceptions, versioning)
- Security (authentication, authorization and privacy)

Point-to-point is most acceptable in environments that are

- Small in number of services and consumers
- Homogenous in technology
- Have low pace of change (business and technology)

© 2013 Carnegie Mellon University

ESB 1

Services connect to a common backbone using Web services or other standards or adapters

ESBs manage

- Interface compatibility (technology/service interface and “schema”)
- Service routing
  - Based on content, availability, load or other rules
  - May be dynamically determined
  - May be one-to-many or aggregate from many-to-one
- Data transformations (format and business semantics)

ESBs are most acceptable in environments that

- Are technologically diverse
- Rapidly changing
- Large

© 2013 Carnegie Mellon University

ESB $_2$

Tends to promote a greater degree of loose coupling / independence of connected systems

Advanced tools and techniques are provided for development and management

Specialized development, maintenance and management resources are required

© 2013 Carnegie Mellon University

Bottom Line

There is a debate often fueled by vendors with vested interest over whether to

- directly integrate applications, or
- use an ESB strategy

There is no single right answer

Most organizations have some of each

© 2013 Carnegie Mellon University

Point-to-Point vs. ESB Tradeoffs 1

	Point-to-Point	ESB
Modifiability	☹ Changes to a service induce change to all service consumers	☺ Service consumers and providers may change independent of each other. Compatibility is managed within the ESB for certain changes
Performance	☺ Tends to perform better (no transformation and routing layers)	☹ Additional layers affect performance
Security	☹ Authentication and authorization between services and consumers must be managed on a case-by-case basis	☺ Allows central management of security for each service

© 2013 Carnegie Mellon University

Point to Point vs. ESB Tradeoffs 2

	Point-to-Point	ESB
Serviceability	☹ Problem determination is spread across services—there is no central point to manage connectivity	☺ Centralized service interface management provides opportunity to centrally log/audit interactions
Reliability	☹ Strong coupling between consumers and services may result in complex failure modes and unintended dependencies	☹ Additional components add complexity and introduce failure modes ☺ Loosely-coupled approach improves overall reliability
Interoperability	☹ Consumer and service need to both support agreed-upon message protocols and data formats	☺ ESBs are designed to support diverse connection mechanisms

© 2013 Carnegie Mellon University

Enterprise Service Bus (ESB) 1

There is no consensus on what constitutes an ESB, although there is a wide agreement on many of the responsibilities

- Some consider ESB a product: "middleware product that connects and mediates all communications and interactions between service consumers and services, usually based on standards*"
- Some do not consider ESB a product, but rather a pattern for which there are multiple vendor and open source implementations — or you could implement your own

In practice, it is common to start from a vendor or open source implementation and then to add extensions or customizations to meet business needs

* Source: Wikipedia

© 2013 Carnegie Mellon University

Enterprise Service Bus (ESB) 2

Business goals for using an ESB

- Reuse of IT assets
- Agility for adding, changing and removing business partners
- Realignment of responsibilities through business reorganizations
- Integration with legacy systems

From a general perspective, an ESB is designed to reduce the complexity of connecting services with their consumers

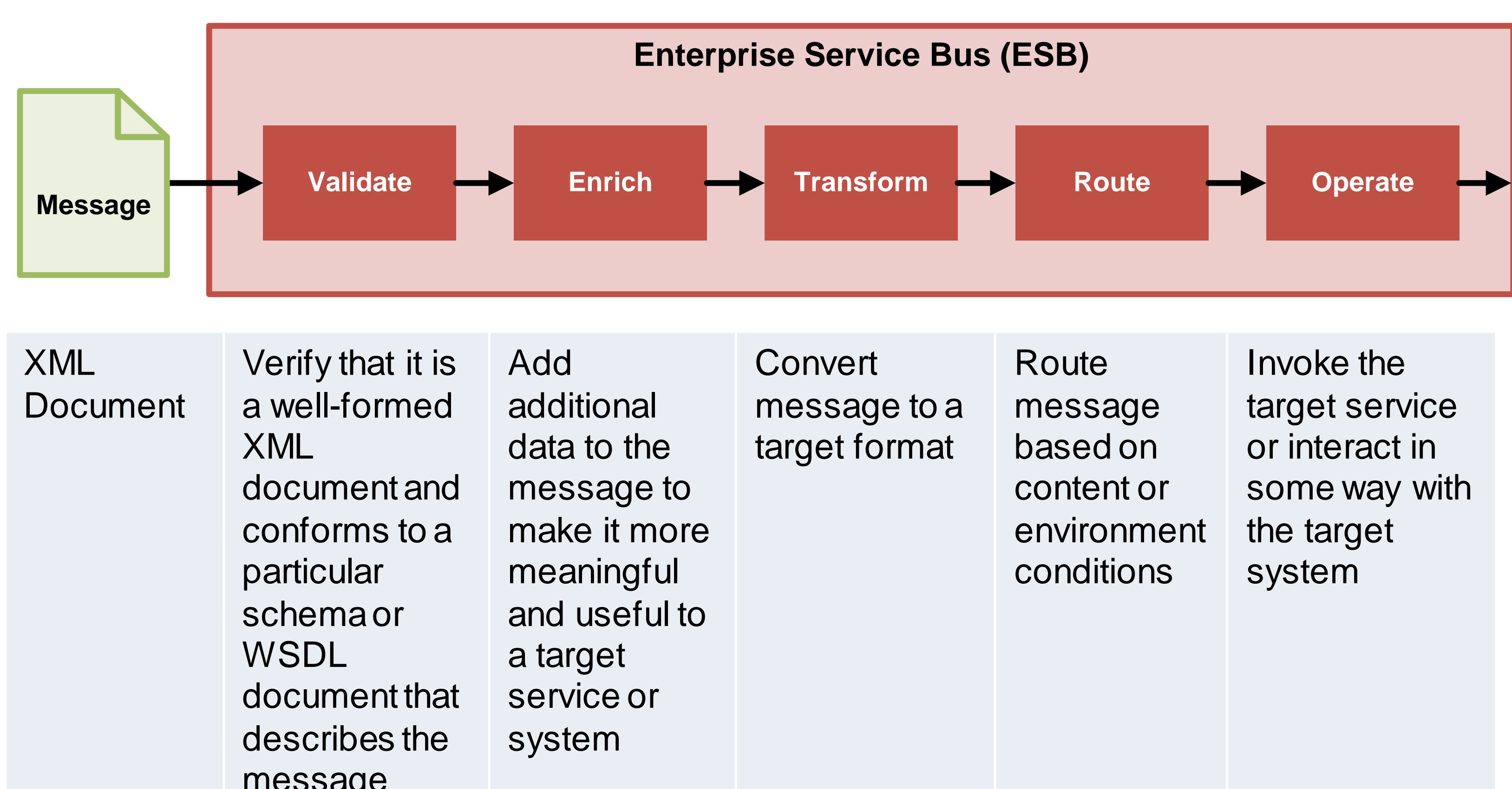
From a technical perspective, an ESB is a complex integration pattern that can be broken down into several less complex supporting patterns and tactics

- These tactics and patterns have known influence on quality attributes

© 2013 Carnegie Mellon University

Enterprise Service Bus (ESB) 3

How do ESBs work? The VETRO Pattern



Source: Dave Chappell, "Enterprise Service Bus" (O'Reilly: June 2004, ISBN 0-596-00675-6)

© 2013 Carnegie Mellon University

ESB Patterns: Broker $_1$

The Broker architectural pattern is responsible for translating protocols and data format*

Supports the interoperability driver

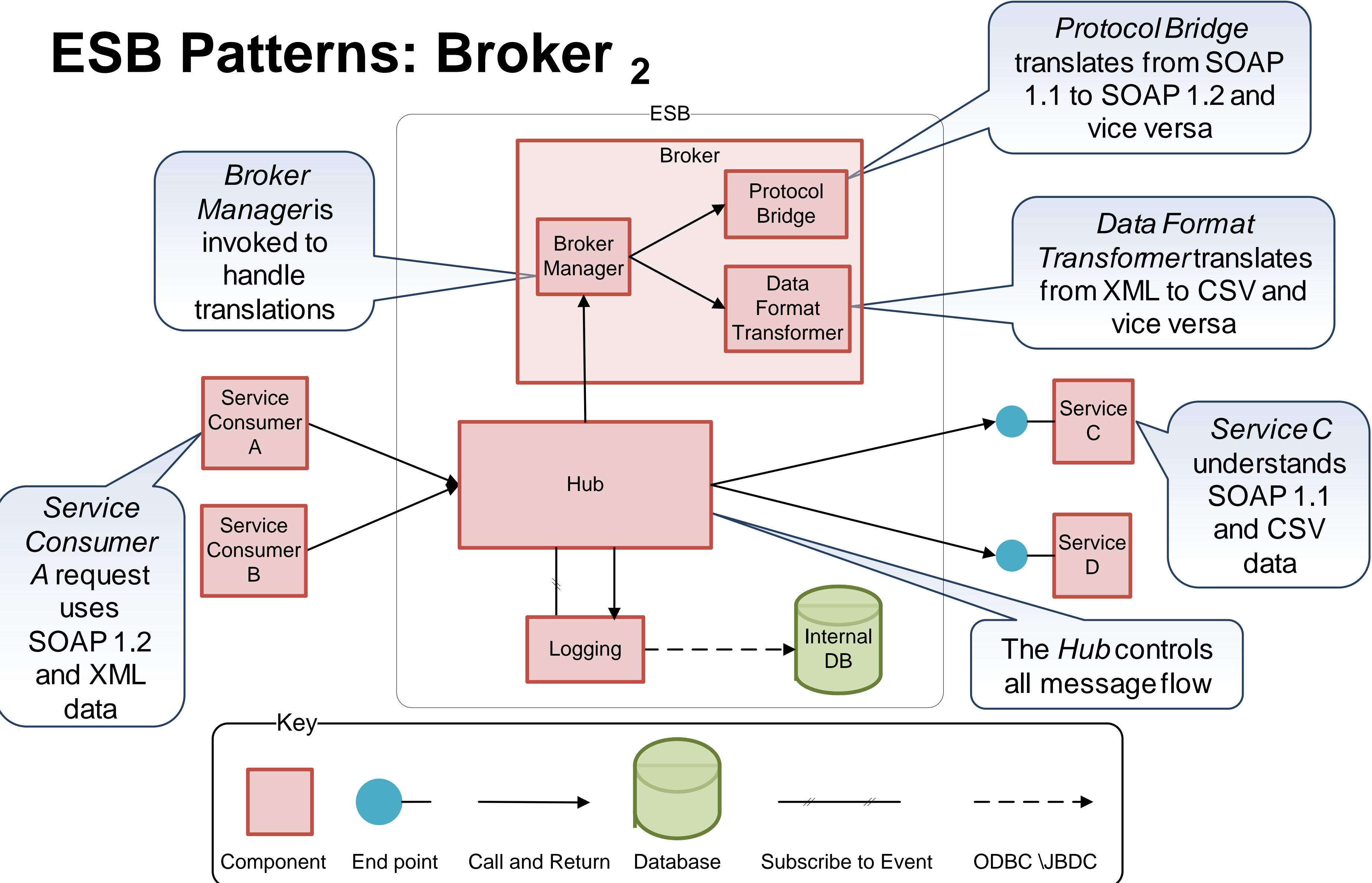
Architectural Tactics

- Abstract common services (modifiability)
- Adherence to defined protocols (interoperability, modifiability, developer usability)
- Use of an intermediary (modifiability)
- Restricted communication paths (modifiability)

* Source: Thomas Erl. SOA Design Patterns. 2009

© 2013 Carnegie Mellon University

ESB Patterns: Broker 2



© 2013 Carnegie Mellon University

ESB Patterns: Routing 1

The Routing architectural pattern is responsible for using runtime factors (current logical conditions, business rules or utilization) to route messages and to allow dynamic composition of services*

Supports the scalability driver

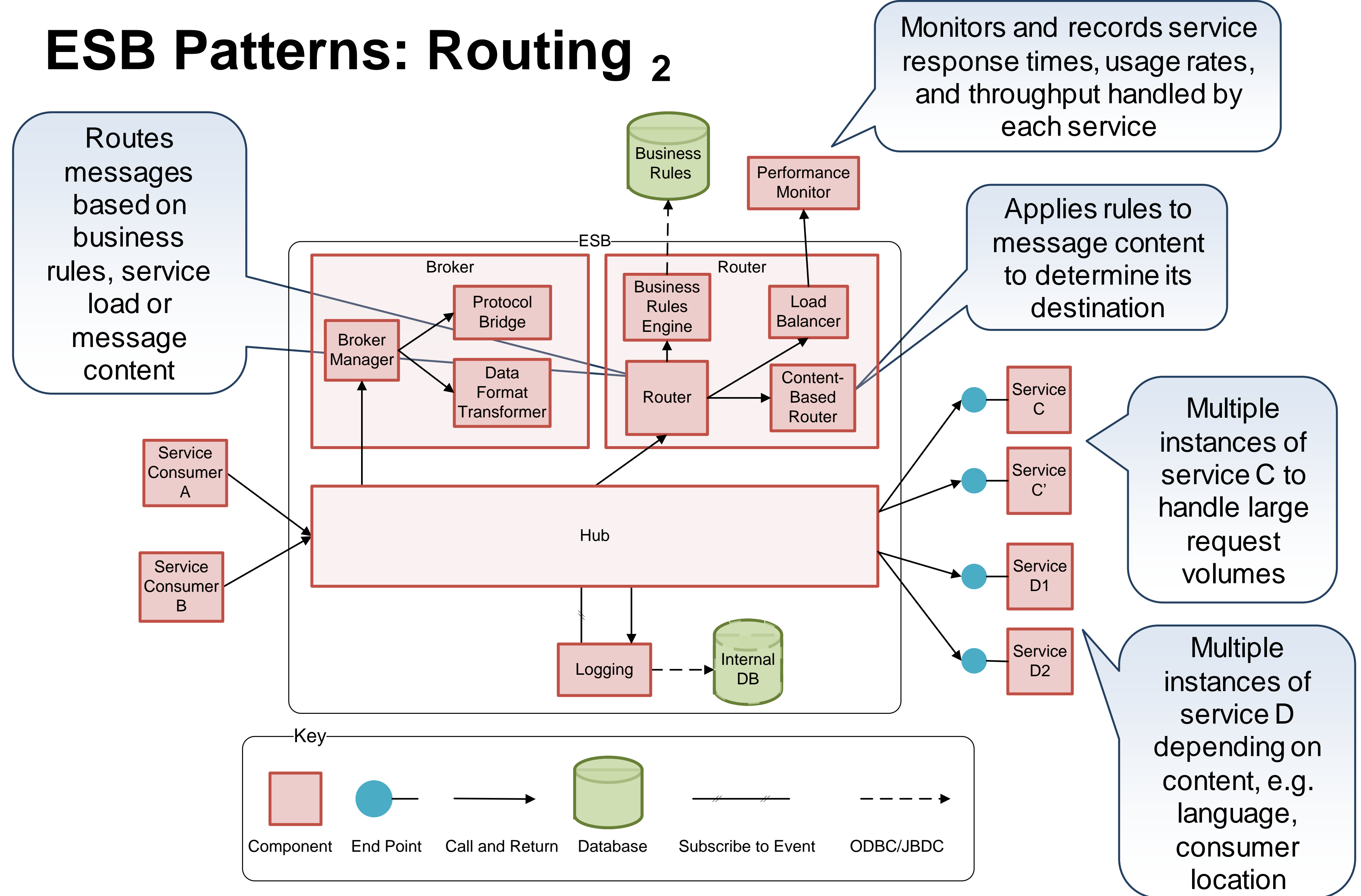
Architectural Tactics

- Abstract common services (modifiability)
- Load balancing (performance, scalability)
- Runtime binding (modifiability)
- Component replacement (modifiability)

* Source: Thomas Erl. SOA Design Patterns. 2009

© 2013 Carnegie Mellon University

ESB Patterns: Routing 2



© 2013 Carnegie Mellon University

ESB Patterns: Asynchronous Queue 1

The Asynchronous Queue architectural pattern provides an intermediate buffer that allows service providers and consumers to be temporally decoupled*

Supports the reliability driver

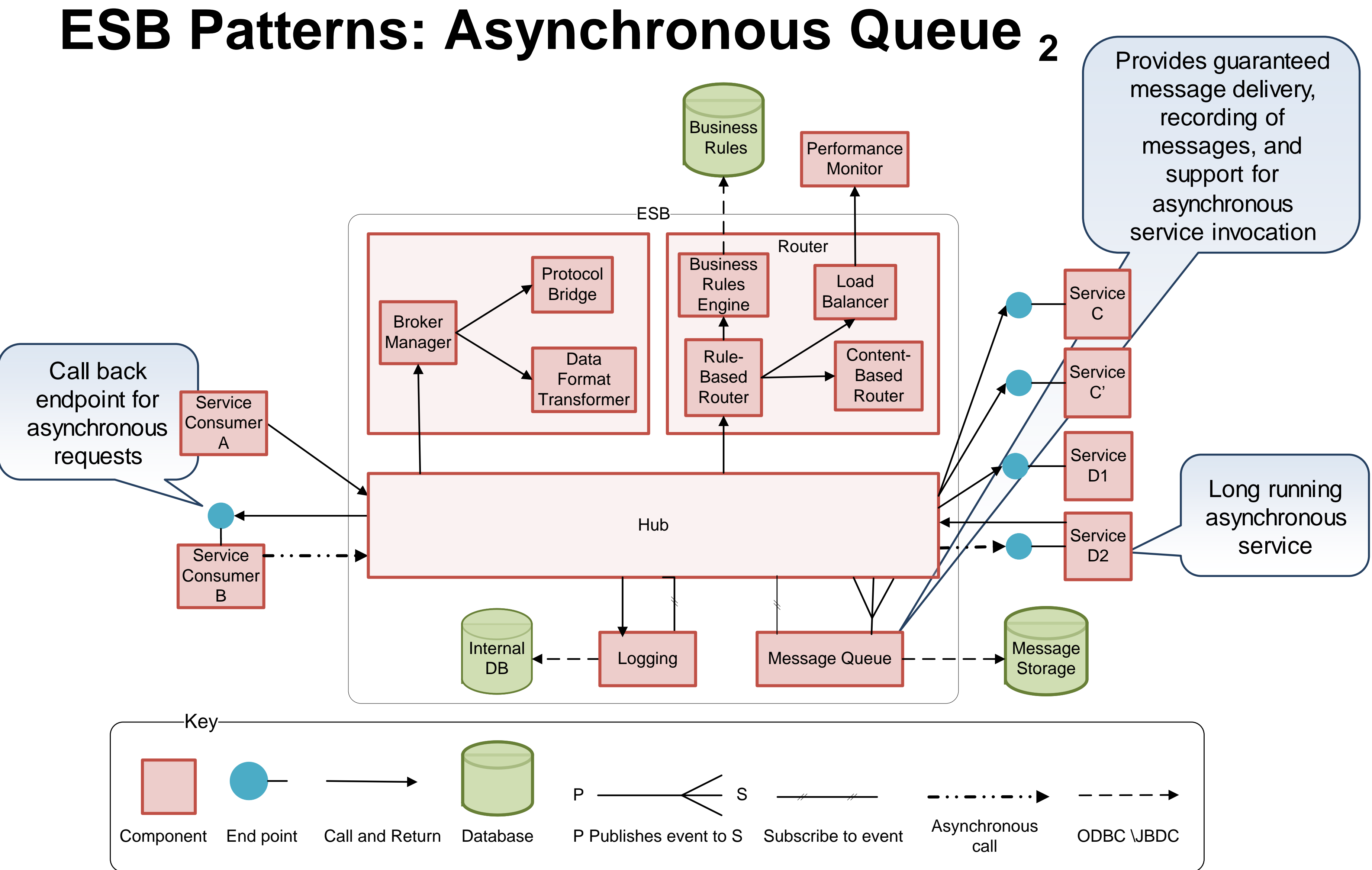
Architectural Tactics

- Adherence to standard protocols (modifiability, interoperability and developer usability)
- Increase available resources (performance)

* Source: Thomas Erl. SOA Design Patterns. 2009

© 2013 Carnegie Mellon University

ESB Patterns: Asynchronous Queue $_2$



© 2013 Carnegie Mellon University

ESB Patterns: Metadata Centralization 1

The Metadata Centralization architectural pattern provides a registry for service metadata to be formally published or registered to allow for service discovery by developers of service consumers

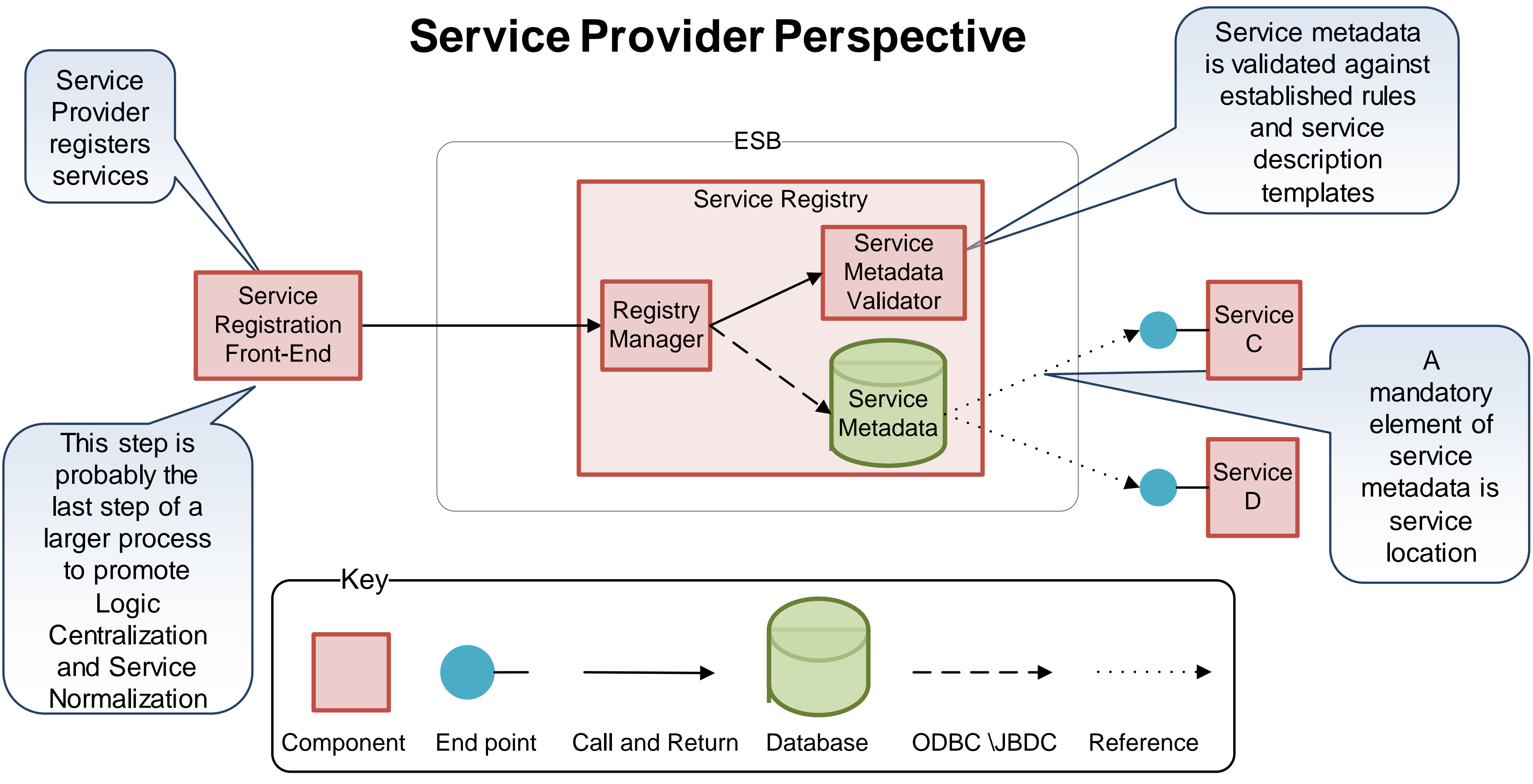
Architectural Tactics

- Adherence to standard protocols (modifiability, interoperability and developer usability)
- Use of an intermediary (modifiability)
- Maintain existing interface (modifiability)

* Source: Thomas Erl. SOA Design Patterns. 2009

© 2013 Carnegie Mellon University

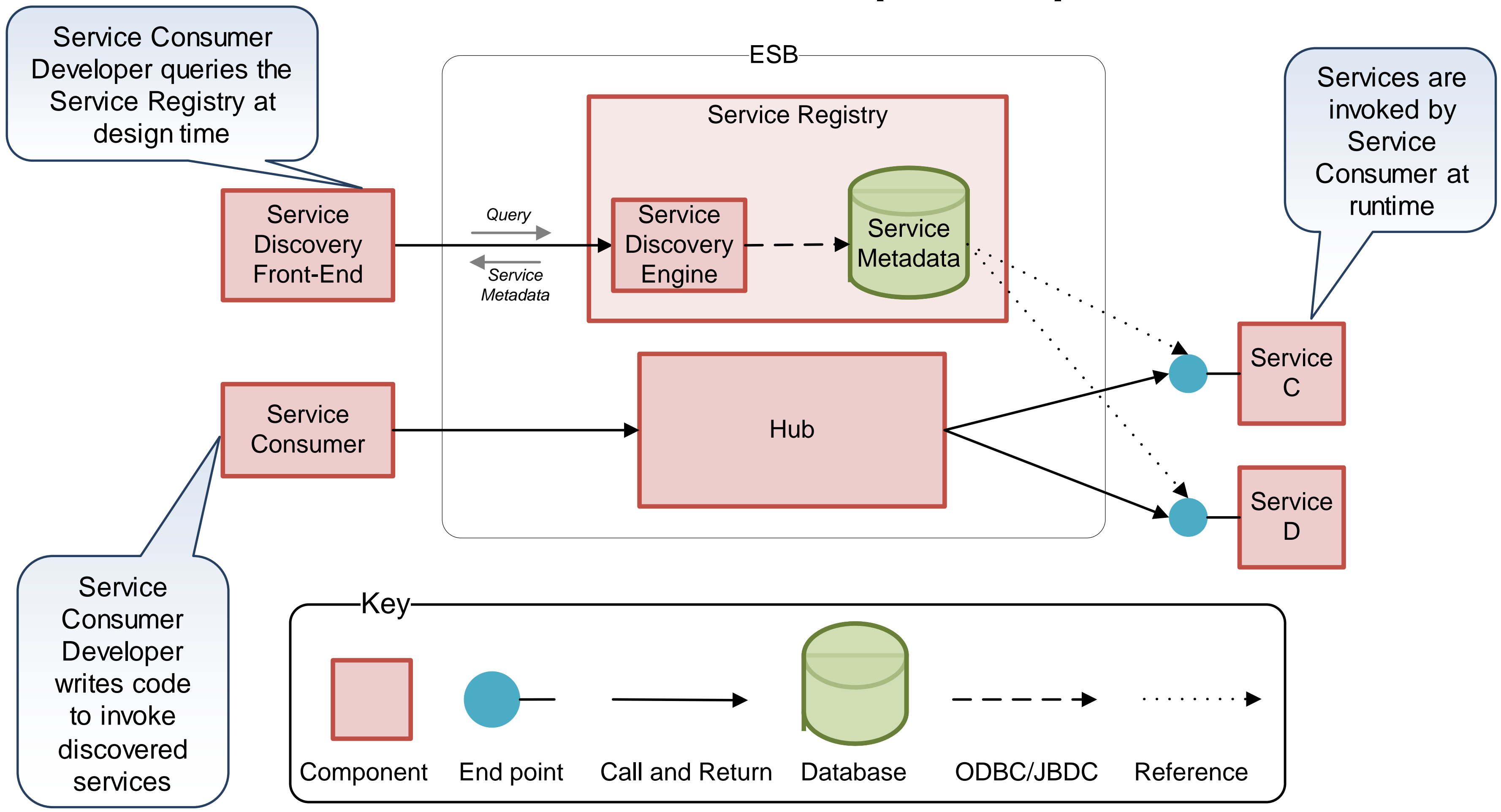
ESB Patterns: Metadata Centralization 2



© 2013 Carnegie Mellon University

ESB Patterns: Metadata Centralization 3

Service Consumer/Developer Perspective



© 2013 Carnegie Mellon University

Summary

An ESB can be broken down into several supporting tactics and patterns

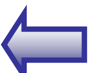
These patterns and tactics have a known influence on software quality attribute scenario response measures

© 2013 Carnegie Mellon University

Agenda: Architecture and Design of Service-Oriented Systems

Review Part 1

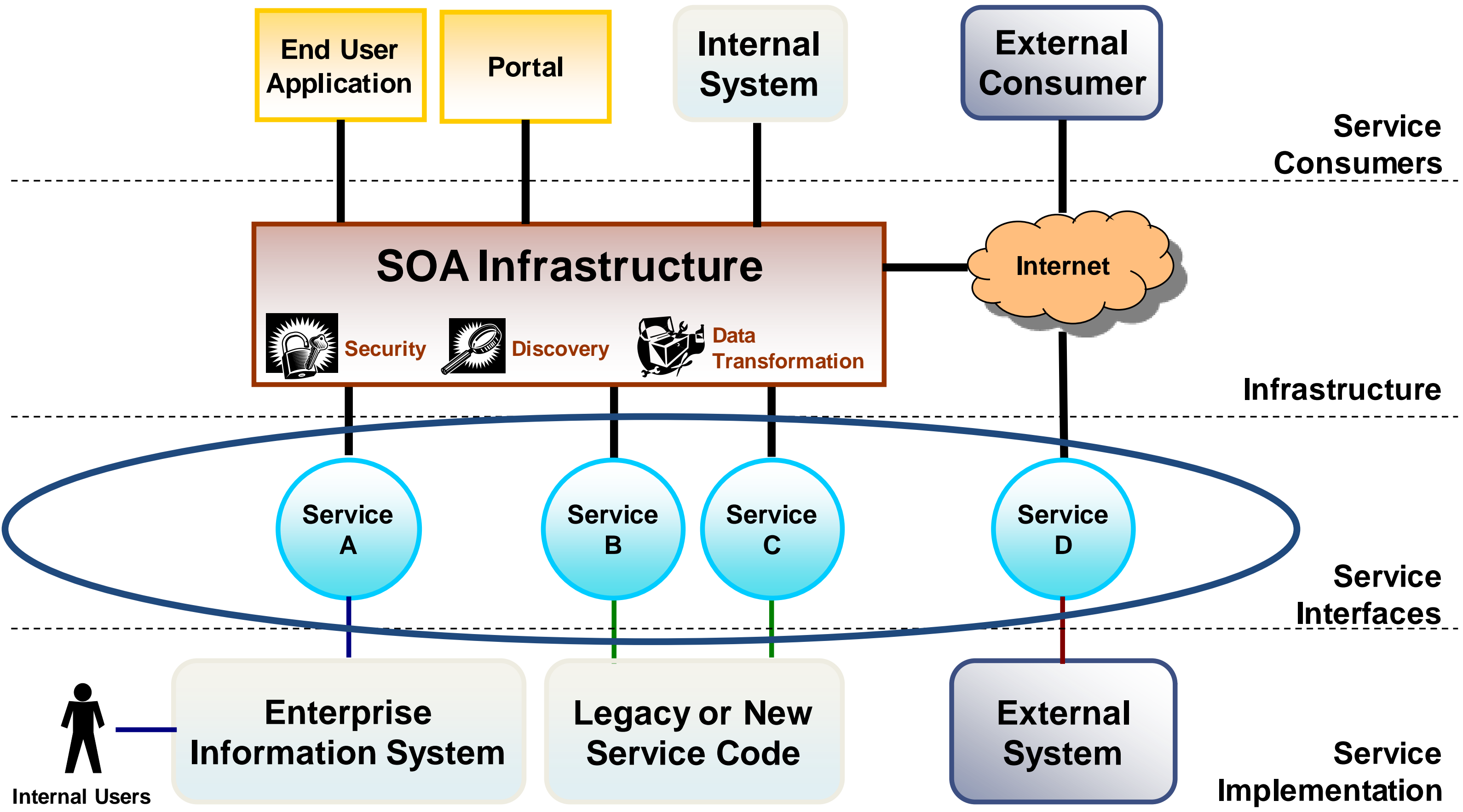
SOA Infrastructure Design Considerations

Service Design Considerations 

Summary

© 2013 Carnegie Mellon University

Focus of this Section



© 2013 Carnegie Mellon University

Principles of Service Design*

Standardized Service Contracts

Loose Coupling

Discoverability

Reusability

Autonomy

Statelessness

Composability

Abstraction

Main Question: How to determine what type of functionality makes a good service?

* Source: Thomas Erl. SOA Design Patterns. 2009

© 2013 Carnegie Mellon University

Service Identification: The Strategic Perspective

Two common approaches

- Top-Down: starting with business process inventory
  - Business processes to support business goals are identified.
  - Shared steps between business processes are identified as service candidates.
- Bottom-Up: starting with legacy system inventory
  - Systems with capabilities to support business goals are identified as migration candidates.
  - Key capabilities are identified as service candidates.

In practice it is usually a combination of both

- Service prioritization is done based on relationship to business goals for SOA adoption

© 2013 Carnegie Mellon University

Service Identification: The Technical Perspective

Common criteria for service selection

- Strategic reuse
- Functionality or data that is private and requires access control
- Functionality that needs to be highly reliable and/or highly available
- Functionality that needs to be run concurrently
- Functionality that will be accessed often (scalability)

In practice, a combination of the strategic and technical perspectives will lead to a good service portfolio

- Service prioritization should be done based on relationship to business and technical goals for SOA adoption

© 2013 Carnegie Mellon University

Service Design

Creating service layers responsible for abstracting logic based on functional type can improve reuse and promote agility

- Layers that represent generic logic (not related to functional context)
- Layers that represent single-purpose logic (related to functional context)

Three typical service layers are

- Utility
- Entity or data
- Process or task

© 2013 Carnegie Mellon University

Utility Service Layer

Provides reusable utilities services for use by other services in the inventory

Goal is to maintain strict separation of utility-based functions and specific business functionality to avoid replication of the utility-based functions across the service inventory

Examples of utility services

- Notifications
- Logging
- Auditing
- Authentication
- Data transformation

© 2013 Carnegie Mellon University

Entity or Data Service Layer

Provides services associated with the processing of business entities

Goal is to maintain strict separation of entity-based functions and specific business functionality because business entities are generally more stable

- Businesses processes change whenever organizations change the way they do business, but the entities that are operated on change less frequently

Entity services are derived from a logical or enterprise data model

- Granularity is not always determined by the underlying data model
  - Some services may operate on multiple entities and some entities may be operated on by more than one service

Examples of entity services

- Employee
- Customer
- Sales Order
- Invoice

© 2013 Carnegie Mellon University

Process or Task Layer

Usually created after the entity and utility services have been defined

- A process service will typically be a composition of business logic plus invocation of entity, utility and process services

Separating the process layer from the other layers promotes service inventory agility for business process changes

- Separating the task-specific functionality from the task-agnostic utilities reduces redundant implementation of the utilities
- The goal is to change only the process-layer internal logic and recompose with the other layers
- Separating the business entities from the specific tasks reduces governance challenges
  - The business entity expertise and the business process expertise often reside in different groups

© 2013 Carnegie Mellon University

Summary

The design of individual services has a huge impact on overall system quality

- Stateless services can be replicated to promote scalability
- Services that are discoverable potentially allow for runtime binding (modifiability)
- Services that use a standardized service contract promote interoperability

Defining services requires careful consideration

- Is the functionality useful in other contexts?
- Constraints may eliminate certain candidates, e.g. regulations, technology availability

Logically grouping services into layers can reduce the complexity of compositions and promote reuse

- Utility, entity or data, and process or task layers are a good place to start

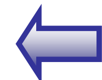
© 2013 Carnegie Mellon University

Agenda: Architecture and Design of Service-Oriented Systems

Review Part 1

SOA Infrastructure Design Considerations

Service Design Considerations

Summary 

© 2013 Carnegie Mellon University

Summary

Quality attributes have the strongest influence on architectural design decisions

- Quality attributes requirements can be captured as scenarios

SOA is a design pattern that promotes interoperability and modifiability

- SOA is not a complete architecture
- It is often combined with other patterns and tactics

There are many ways to design service-oriented systems

- A service-oriented design can be as simple as a small set of services that are integrated point-to-point
- A service-oriented design can include a complex infrastructure that helps enterprises manage rapidly evolving business processes in more agile ways

© 2013 Carnegie Mellon University

SATURN 2013

Software Engineering Institute | Carnegie Mellon

Minneapolis, Minnesota

April 29 to May 3, 2013

www.sei.cmu.edu/saturn/2013

© 2013 Carnegie Mellon University

Copyright 2013 Carnegie Mellon University

This material has been approved for public release and unlimited distribution except as restricted below.

This material is distributed by the Software Engineering Institute (SEI) only to course attendees for their own individual study. Except for the U.S. government purposes described below, this material SHALL NOT be reproduced or used in any other manner without requesting formal permission from the Software Engineering Institute at permission@sei.cmu.edu.

This material was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose this material are restricted by the Rights in Technical Data-Noncommercial Items clauses (DFAR 252-227.7013 and DFAR 252-227.7013 Alternate I) contained in the above identified contract. Any reproduction of this material or portions thereof marked with this legend must also reproduce the disclaimers contained on this slide.

Although the rights granted by contract do not require course attendance to use this material for U.S. Government purposes, the SEI recommends attendance to ensure proper understanding.

NO WARRANTY. THE MATERIAL IS PROVIDED ON AN "AS IS" BASIS, AND CARNEGIE MELLON DISCLAIMS ANY AND ALL WARRANTIES, IMPLIED OR OTHERWISE (INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE, RESULTS OBTAINED FROM USE OF THE MATERIAL, MERCHANTABILITY, AND/OR NON-INFRINGEMENT).