

# *Enterprise Applications Integration (EAI)*



## **Where functionality resides**

How networks are used?

What machines/devices are used, servers, clients, mainframes, virtual servers, "cloud"?

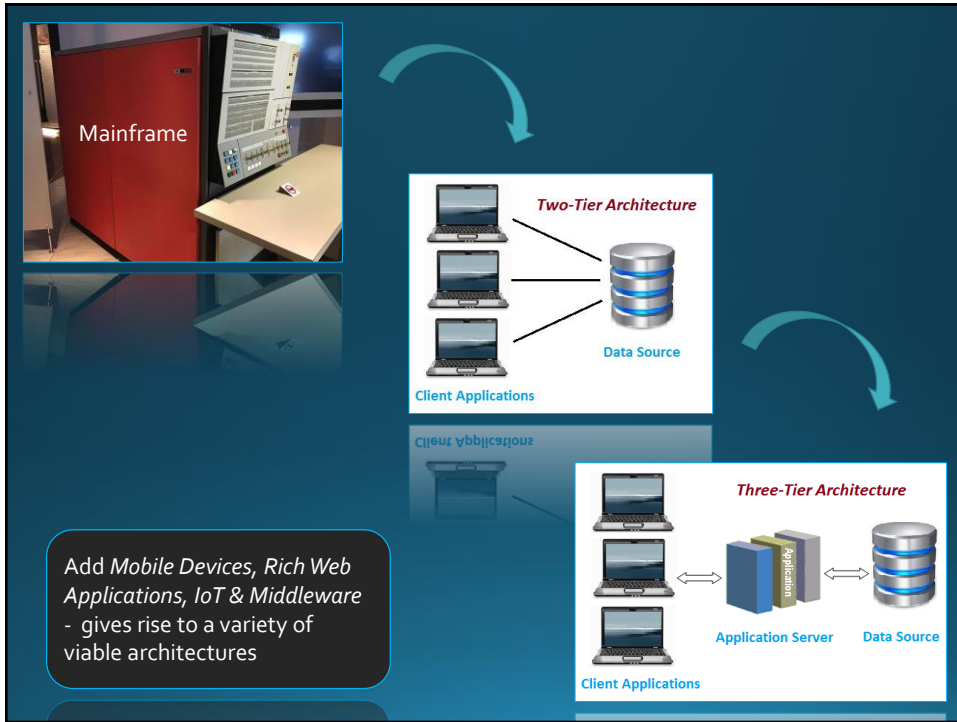


## **How the functionality is designed:**

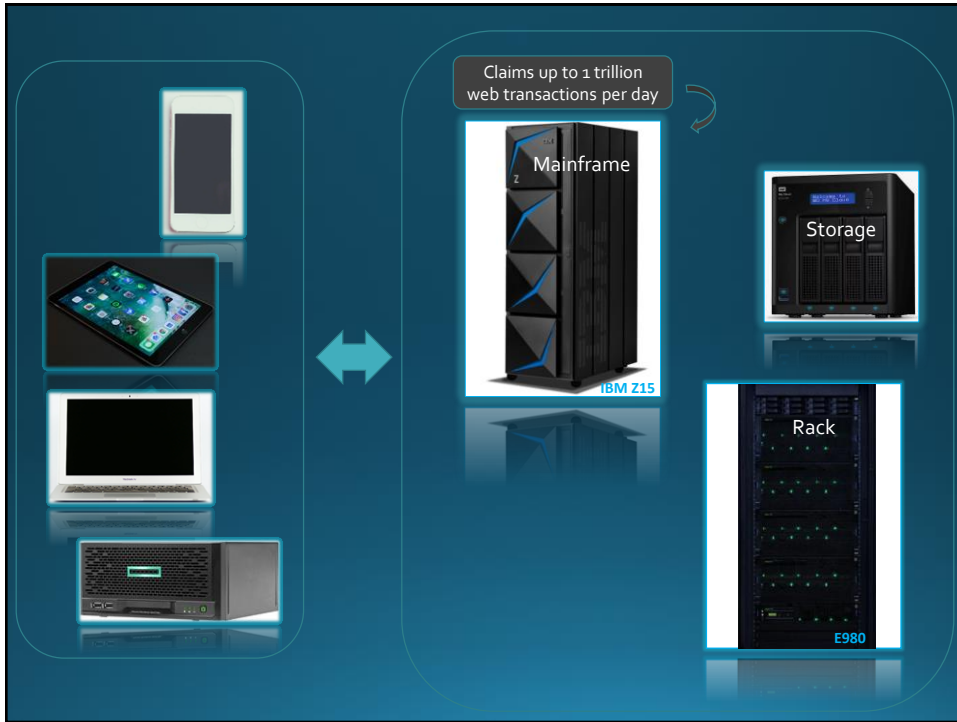
Individual programs: tightly coupled

Enterprise systems: loosely coupled

# Enterprise Architectures



IBM's Z Series



# N-Tier Architecture

# Enterprise Architecture

## *N-Tier*



### Multi-Tier / Layered

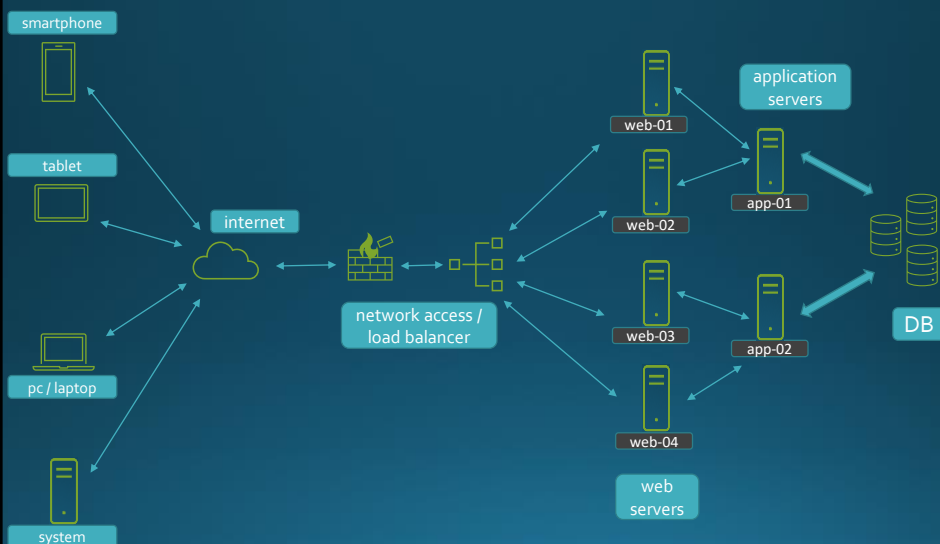
Access  
Presentation  
Business  
Persistence  
Data Repository  
Component/Service based


Multiple client  
devices /  
systems

Multiple  
datastores

Component/Service based

## A Basic Enterprise Architecture Example





## Advantages of N-Tier Architecture

- **Separation of concerns**
  - Functionality is separated into its responsible parts
  - E.g. web page rendering vs business functions
- **Maintainability**
  - Identifying relevant implementation code easier
  - Testing specific functional concerns easier
  - Separation of developer skills possible
- **Extendibility**
  - Adding new code / refactoring existing code is made easier and less error prone
- **Loose coupling**
  - Design-time: Implementation can change with minimised impact on dependant functionality
  - Run-Time: Time/Location/Protocol independence of the functions gives a more robust system
- **Functionality Access**
  - Different client devices can access the same business functionality

## Frameworks



Sets of foundation software that provide core functionality allowing developers to concentrate on the business requirements.



Frameworks

presentation (e.g. Template Tools / Angular / React)  
business logic (e.g. Spring / NodeJS / Django)  
persistence (e.g. Hibernate)

## JEE – Java Enterprise Edition

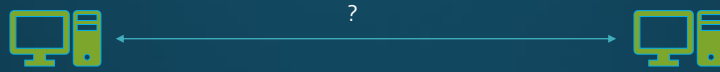


### JEE Technologies

- ✓ Access / Presentation Layer – Java Servlets / View templates / JSP / JSF
- ✓ Business Layer
  - ✓ Java - POJO / EJB
  - ✓ Web Services – JAX-RS
- ✓ Persistence Layer - JPA
- ✓ Integration Layer
  - ✓ *Java Messaging Specification (JMS)*  
*(Integration Technology)*
  - ✓ XML API's

## Message Oriented Middleware

# Loose Coupling

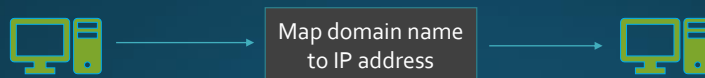


When two systems are tightly coupled there are a set of requirements that are forced into play

- ✓ Data encoding used by both systems has to be the same
- ✓ The location of the system to communicate with is known
- ✓ The system that needs to be communicated with is currently up and running
- ✓ The two systems communicate using a common protocol/format

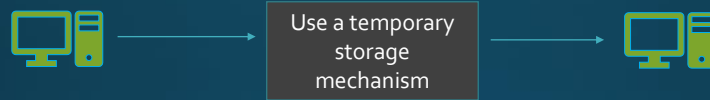
# Loose Coupling

To change from a tightly coupled situation to a loosely coupled situation we need to remove these requirements.



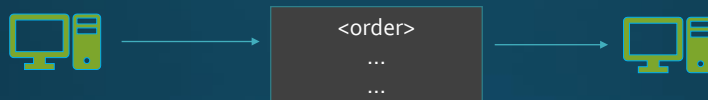
**Location** – introduce a level of indirection so that systems can be located / moved without the need for the other system to be aware.

## Loose Coupling



**Time** – introduce a temporary storage mechanism so that systems can send / receive content asynchronously. Receiving system does not need to be up and running at the time of the request but it will still be processed without error.

## Loose Coupling

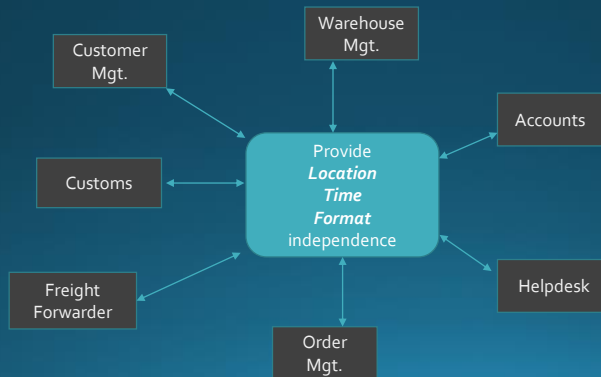


**Format** – introduce a mechanism to allow systems use their own data formats. Each system uses their own format but content is transformed / validated as needed without the systems themselves being aware.

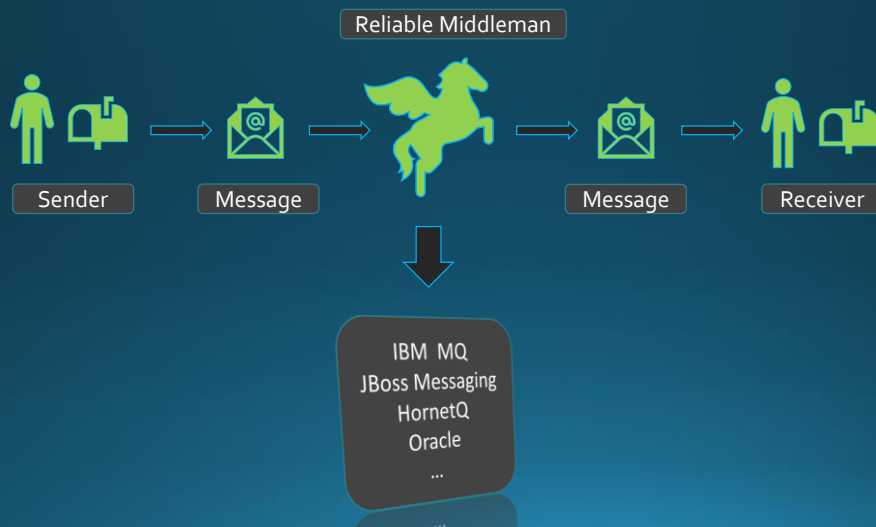


# Loose Coupling

- ✓ Reduces brittleness
- ✓ Allows scalability
- ✓ Can allow additional processing without the need for systems development

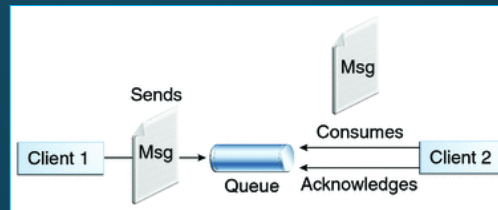


# Message Oriented Middleware



## Messaging Domains

### Point to Point



- ✓ Built around the concept of message *queues*
- ✓ Each message has only one consumer
- ✓ Can have multiple clients sending messages
- ✓ Can have multiple clients as consumers
- ✓ Only *one consumer will receive the message*

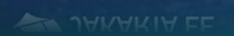
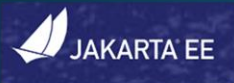
## Messaging Domains

### Publish - Subscribe



- ✓ Uses a *topic* to send and receive messages
- ✓ Each message has multiple consumers
- ✓ A Topic is a type of message destination
- ✓ Can have multiple clients publishing messages
- ✓ Each subscriber receives the message

# Java Messaging Specification

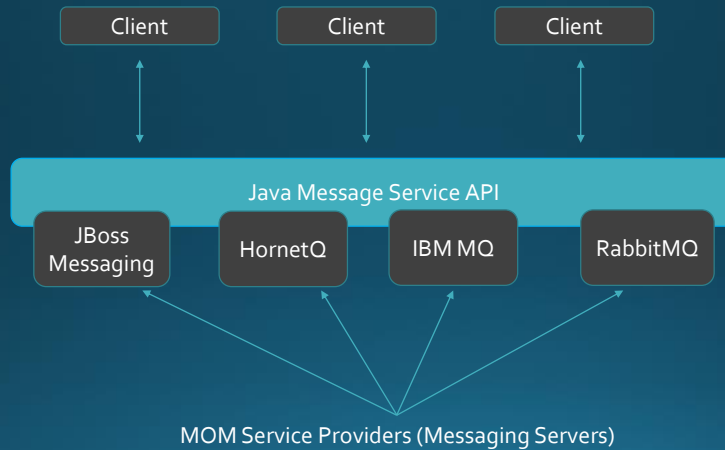


## Java Messaging Specification



- Provides a **standard java API** to Message Oriented Middleware (MOM)
- A **specification** that describes a common way for Java programs to create, send, receive and read distributed enterprise messages
- Java **clients** that connect to the message broker **are abstracted** from the specific software implementation
- **Loosely coupled** communication
- **Asynchronous** messaging
- **Reliable delivery** - A message is guaranteed to be delivered once and only once.

## Messaging – Message Oriented Middleware (MOM) & JMS



## Message Consumptions

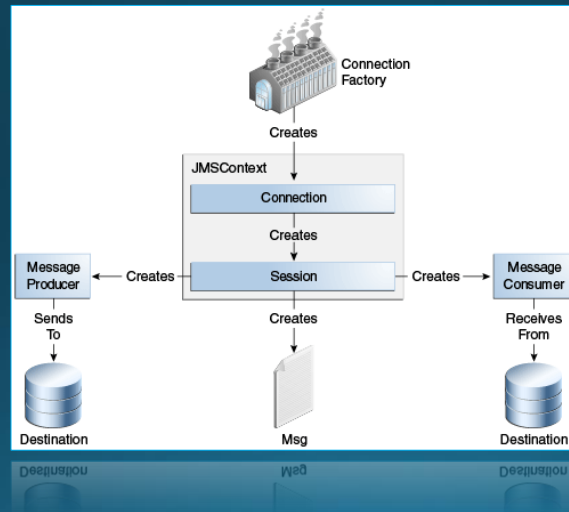
### Synchronously

- A subscriber or a receiver explicitly fetches the message from the destination by calling the receive method.
- The receive method can *block* until a message arrives or can time out if a message does not arrive within a specified time limit.

### Asynchronously

- A client can register a *message listener* with a consumer.
- Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's `onMessage()` method.

## JMS API Programming Model

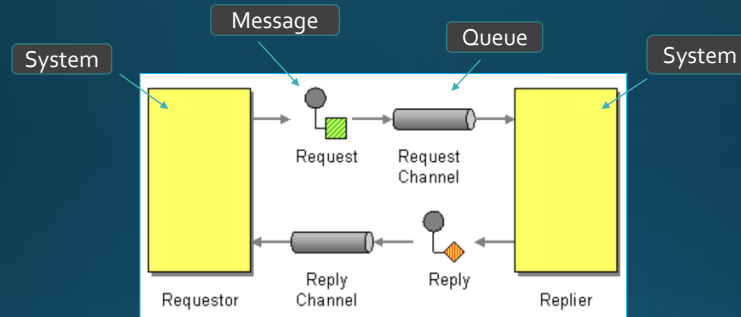


Ref: <https://docs.oracle.com/javaee/7/tutorial/jms-concept003.htm#BNCEH>

*Ref:*  
*Enterprise*  
*Integration*  
*Patterns*  
 By Gregor Hohpe  
 and Bobby Woolf

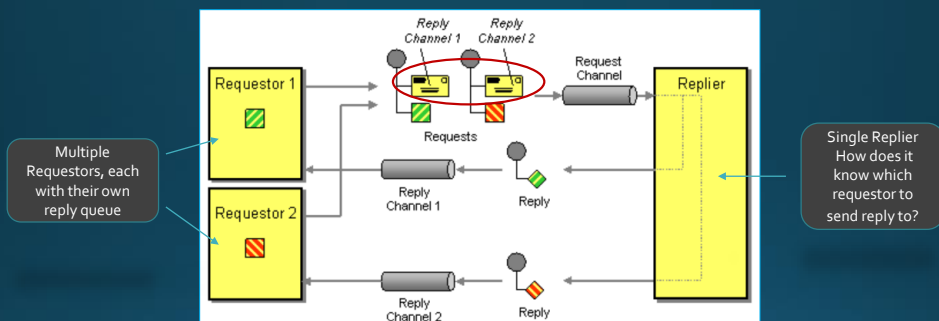
## Enterprise Integration Patterns

# Request - Reply



- A request *message* is sent from *requestor* to *replier*
- Replier responds with a separate reply message
- Communication is *asynchronous* with two *point-to-point* channels involved

# Return Address

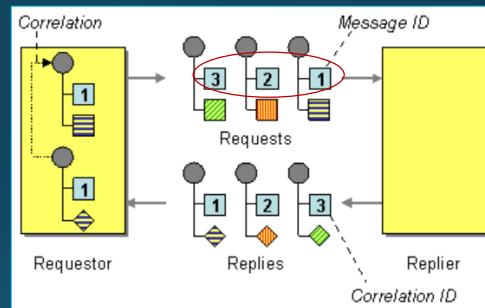


- The requestor specifies a *return address* (reply channel) as part of the message.
- The replier sends its response via the specified reply channel.

# Correlation Identifier

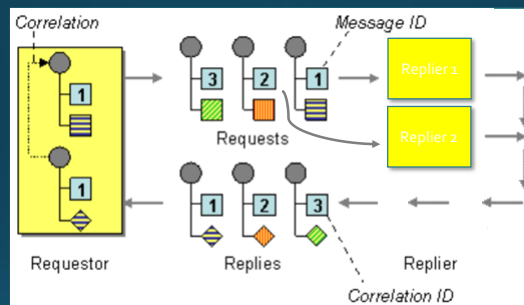
Note channels are not shown here

Multiple requests sent. How does requestor know which reply is for which request (e.g. replies could be out of order)



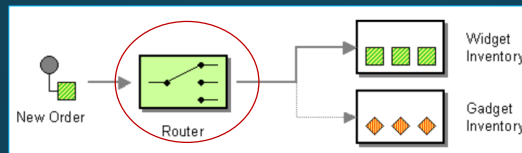
Requestor stamps unique (correlation) id and replier copies to the reply message

# Correlation ID for Load Balancing



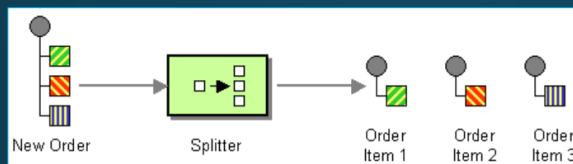
Could also be used in a load balancing scenario where there are multiple repliers in a *point-to-point* setup (only one of the repliers gets the request)

# Content Based Router



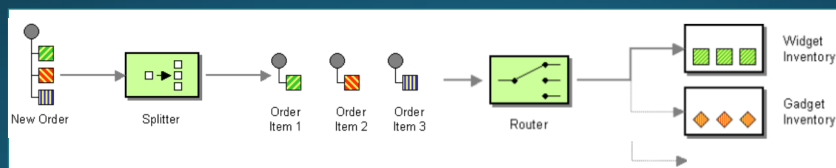
The router passes messages to different channels based on some information within the message

# Splitter



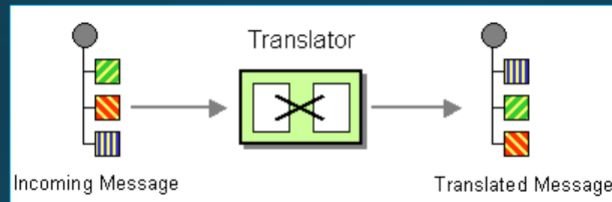
The splitter breaks a composite message into individual messages

# Combining the *Splitter* and *Router*



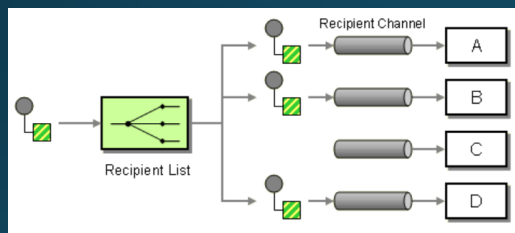


## Message Translator



Translate a message from one format to another (while in transit from one system/component to another)

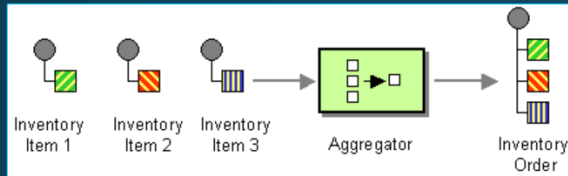
## Recipient List



Multiple recipients will each receive a copy of the message

Note, this could be implemented using a *publish and subscribe* solution also

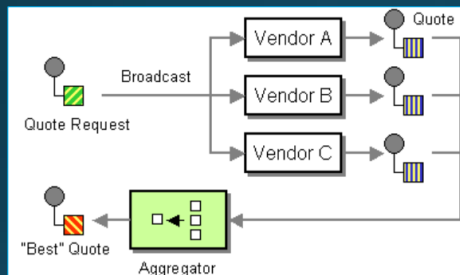
# Aggregator



Aggregator combines separate but related messages

The aggregator must have a mechanism for determining when to stop combining messages and to send the final result.

# Scatter - Gather



Here, we want to broadcast via *publish and subscribe* and receive a response via *point-to-point*

Can be done by combining *Recipient List* with *Aggregator*