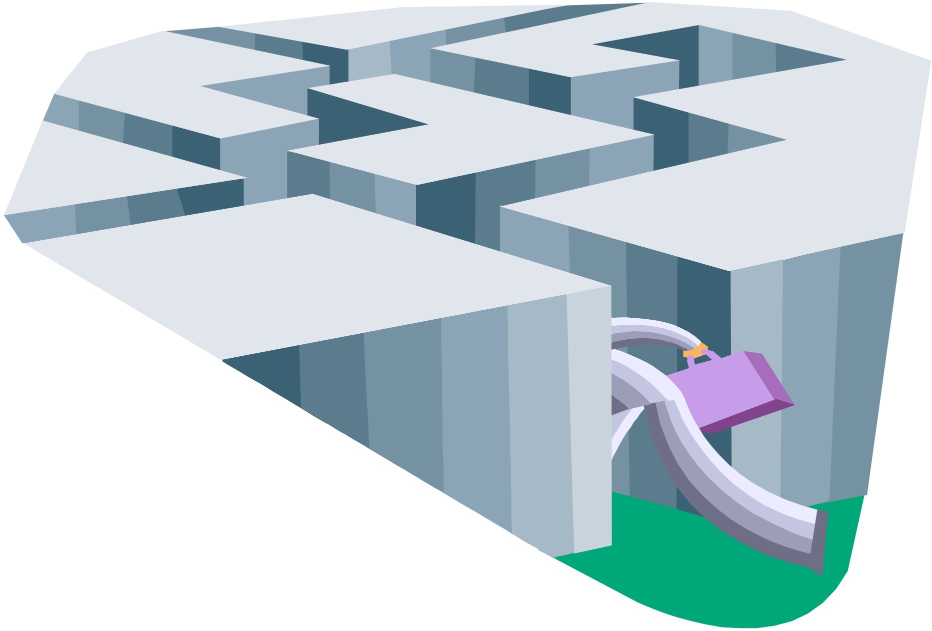
Software Architecture Patterns

*based on a tutorial of Michael Stal



Harald Gall

University of Zurich

http://seal.ifi.uzh.ch/ase

University of Zurich UZH

s.e.a.l. software evolution & architecture lab

Overview

- Goal
  - Basic architectural understanding of patterns, software architecture and middleware
- Motivation
  - Fundamental architectural principles
  - Platform Comparison
- Design Issues
- Summary

Motivation

- Driven by the Internet as well as mobile and embedded devices distributed solutions are now considered common place.
- However, building distributed applications is a non-trivial task.
- The question is: how can we efficiently build and deploy such applications?

Multi-tier approach

- Separation of concerns
  - Presentation
  - Business logic (middle-tier)
  - Data (backend system)

Multi-tier components

- Presentation Tier components:
  - they typically represent sophisticated GUI elements.
  - they share the same address space with their clients.
  - their clients are containers that provide all the resources.
  - they send events to their containers.
- Middle Tier components:
  - they typically provide server-side functionality.
  - they run in their own address space.
  - they are integrated into a container that hides all system details.

Requirements for Distributed Component-based Applications

- Transparent, platform-neutral communication
- Activation strategies for remote components
- Non-functional properties such as performance, scalability, quality of service (QoS)
- Mechanism to find and create remote components
- Keeping state persistent and consistent
- Security issues
- Data transformation
- Deployment and configuration

Using Standard OO Middleware

- What about using plain CORBA or DCOM?
  - Yes, they provide an OO-RPC
  - But developers must integrate services such as transactions or security themselves
- Integration done by developers may require more than 50% of development time
- Conclusion:
  - An object-oriented RPC helps to connect different islands of code, but is not sufficient for building the middle tier.

Architecture for Middle Tier Components

- Let us introduce step by step some of the architectural elements required to build sophisticated middle tier component infrastructures.
  - Communication

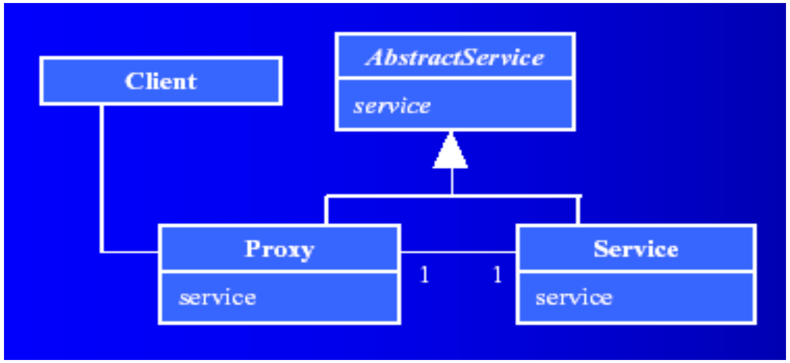
Communication

- In Multi-Tier systems we need to connect the tiers as well as the components within these tiers using communication mechanisms.
- There are four main styles of communication:
  - Collocated client/server (native procedure call)
  - Synchronous RPC
  - Asynchronous RPC
  - Message Queuing (Publish/Subscribe, Peer-to-Peer)

Handling Synchronous Communication

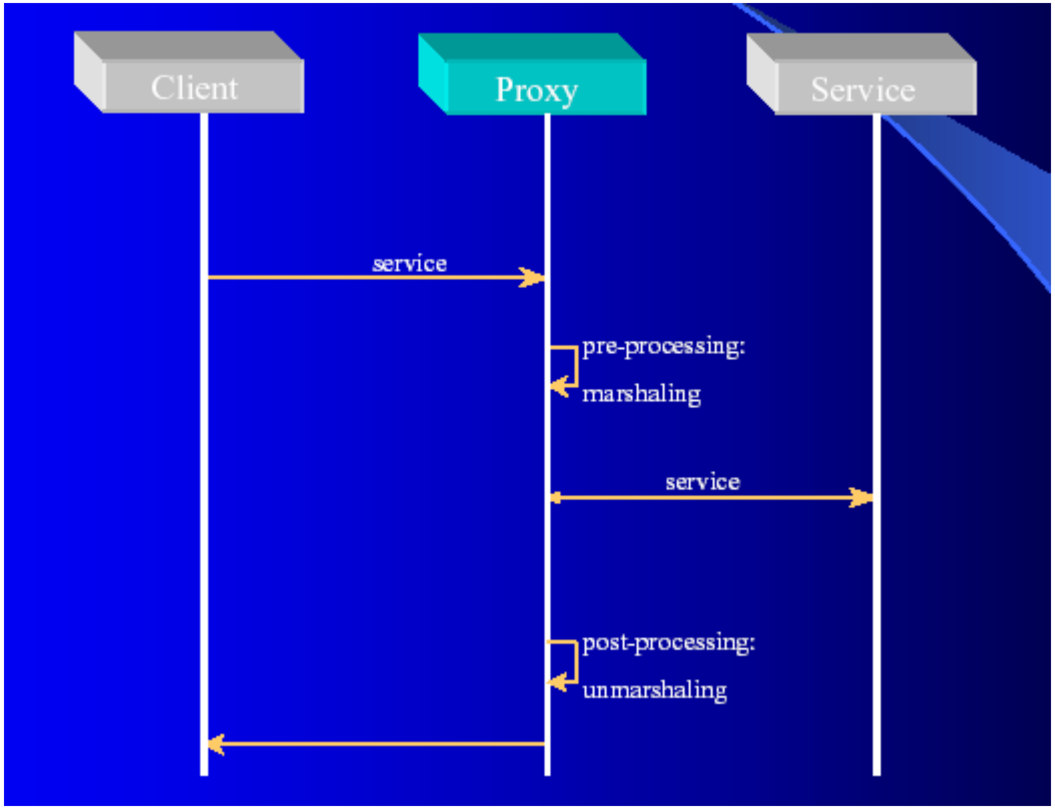
- Components should appear same as local components from a communication perspective
- Clients and servers should be oblivious to communication issues.

Proxy Pattern



- **Solution**
  - Provide a placeholder for the object through which clients can access it
  - A *Service* implements the object which is not directly accessible.
  - A *Proxy* represents the Service and ensures the correct access to it. The Proxy offers the same interface as the Service.
  - *Clients* use the Proxy to get access to the Service.

Dynamics


Client
Proxy
Service
service
pre-processing:
marshaling
service
post-processing:
unmarshaling

Benefits and Liabilities

- **Benefits**
  - Access control to originals.
  - Memory savings.
  - Performance gaining (cache proxy).
  - Separation of housekeeping and functionality.

- **Liabilities**
  - Potential overkill, if proxies include overly sophisticated functionality.
  - Level of indirection.

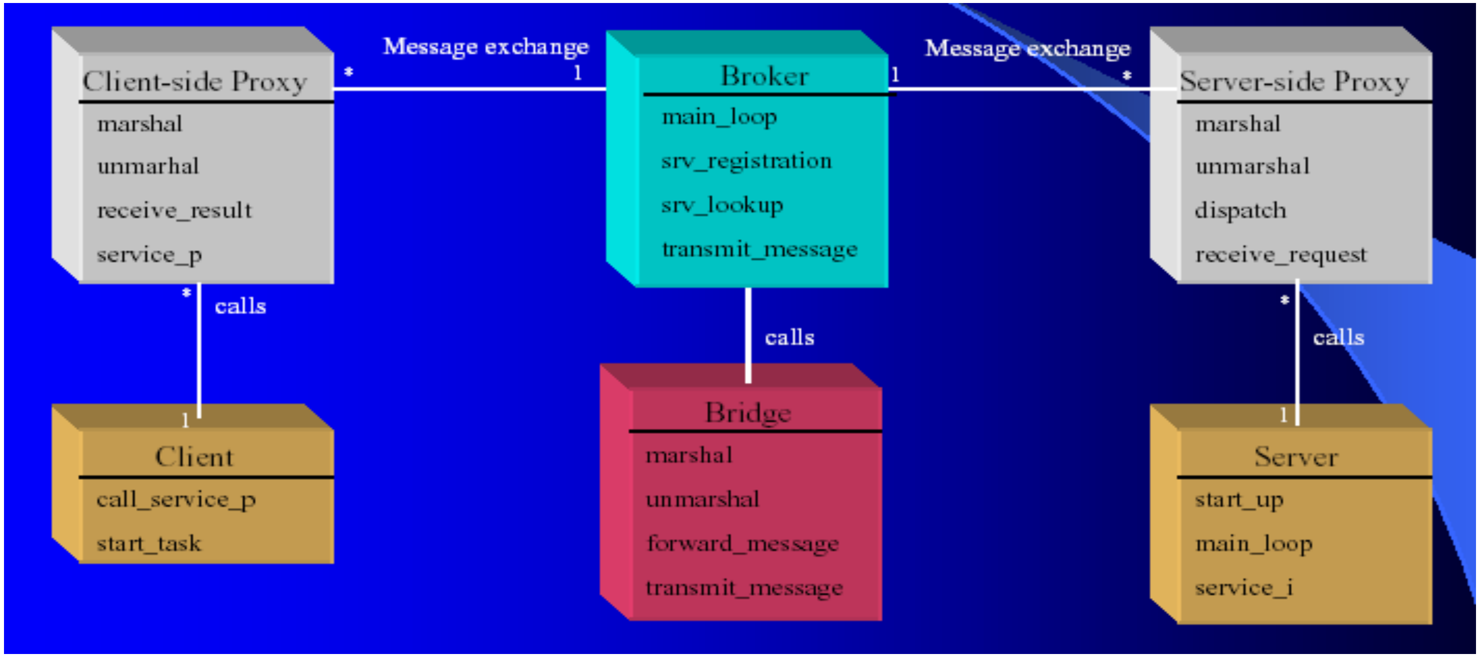
Proxy++

- However, using proxies is not sufficient:
  - How do we locate remote components?
  - How do we handle communication establishment and exchange of network packets (the protocol)?

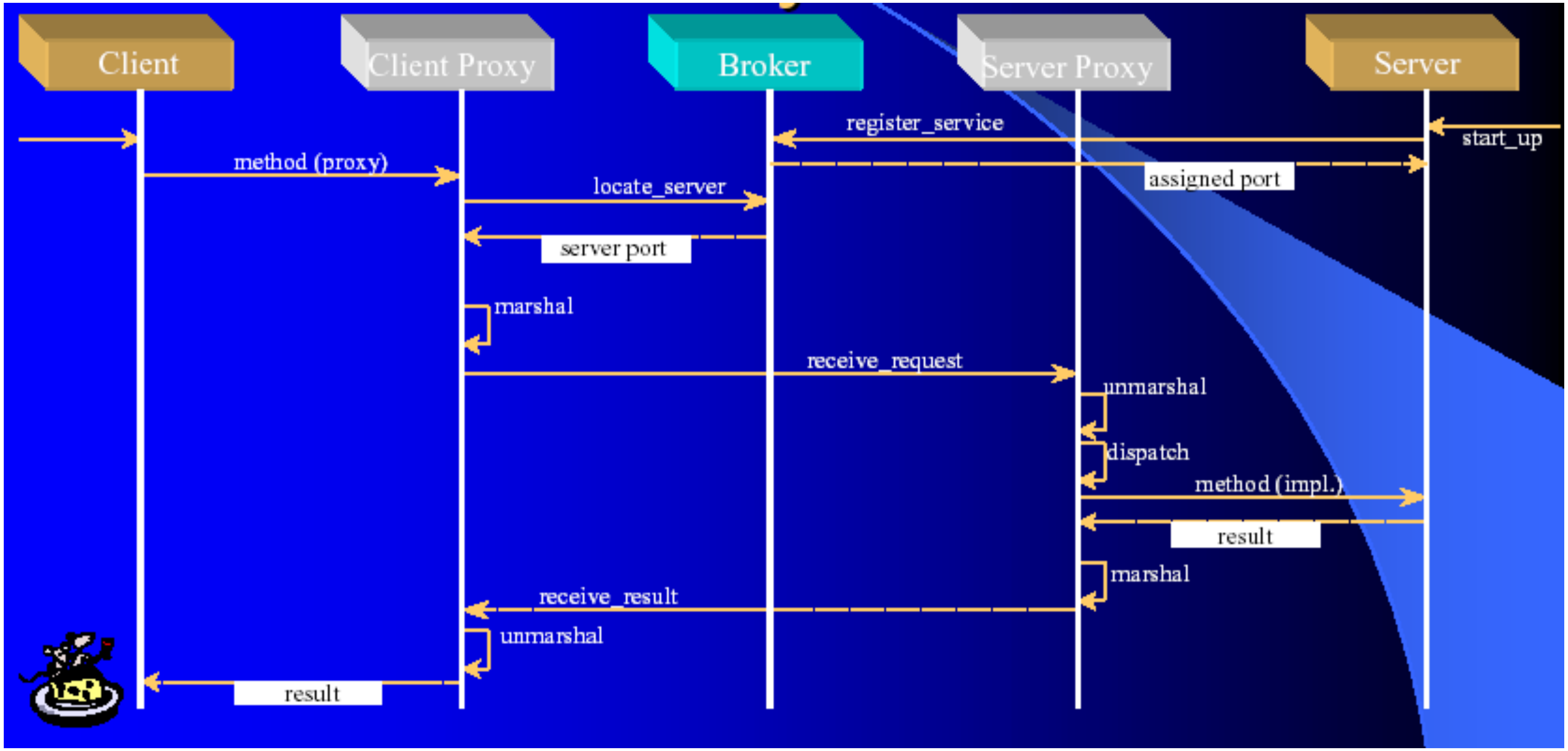
Object-Oriented Middleware Architecture

- What we need is an architecture that ...
  - supports a remote method invocation paradigm
  - provides location transparency
  - allows to add, exchange, or remove services dynamically
  - hides system details from the developer

Architectural Pattern: Broker



Broker Dynamics



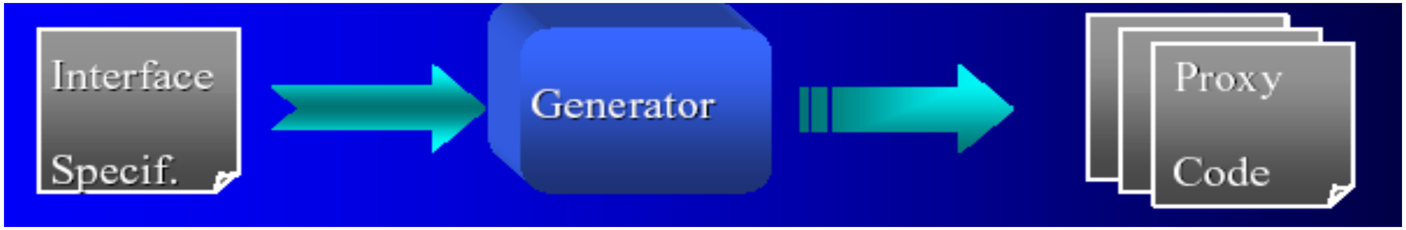
Benefits and Liabilities

- **Pros**
  - Broker and Proxies hide communication
  - Details of the OS are also hidden
  - Interoperability with other brokers (bridges)
  - Reusability of services
  - Location Transparency
  - Dynamic
  - Reconfiguration

- **Cons**
  - Restricted efficiency due to indirection
  - Multiple points of failure
  - Testing and debugging harder as with all distributed systems

Generation of „Glue“

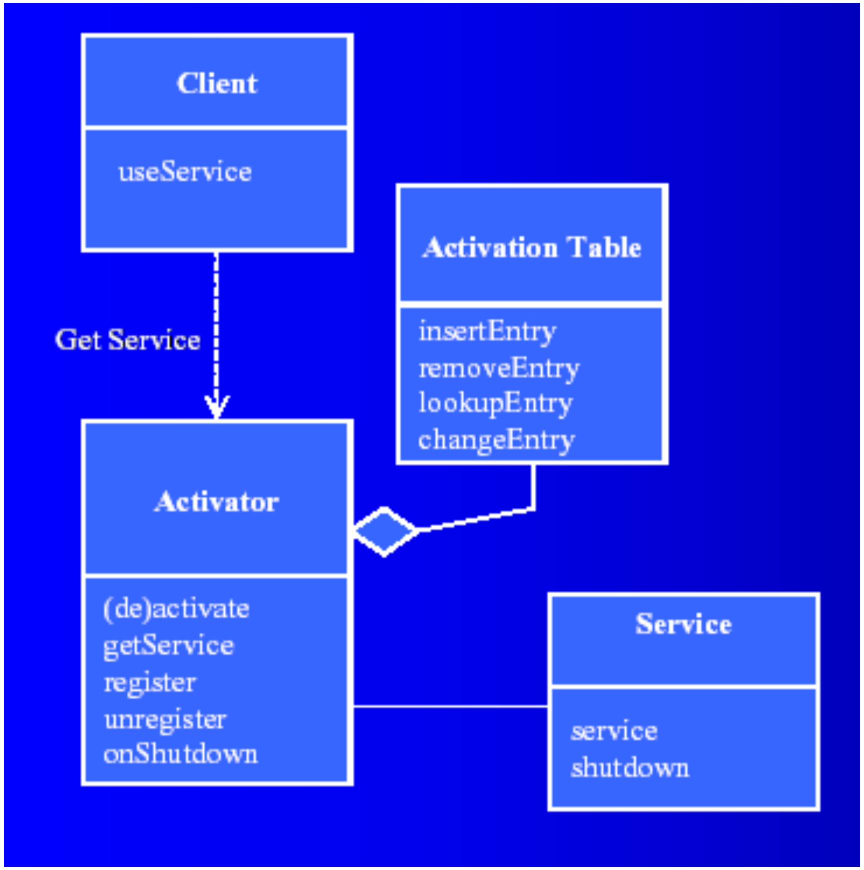
- Who is in charge to implement all of these factories, proxies, etc.?
- Generator tools provide the generation of necessary artifacts from high level language data definitions:



Problem: Activation

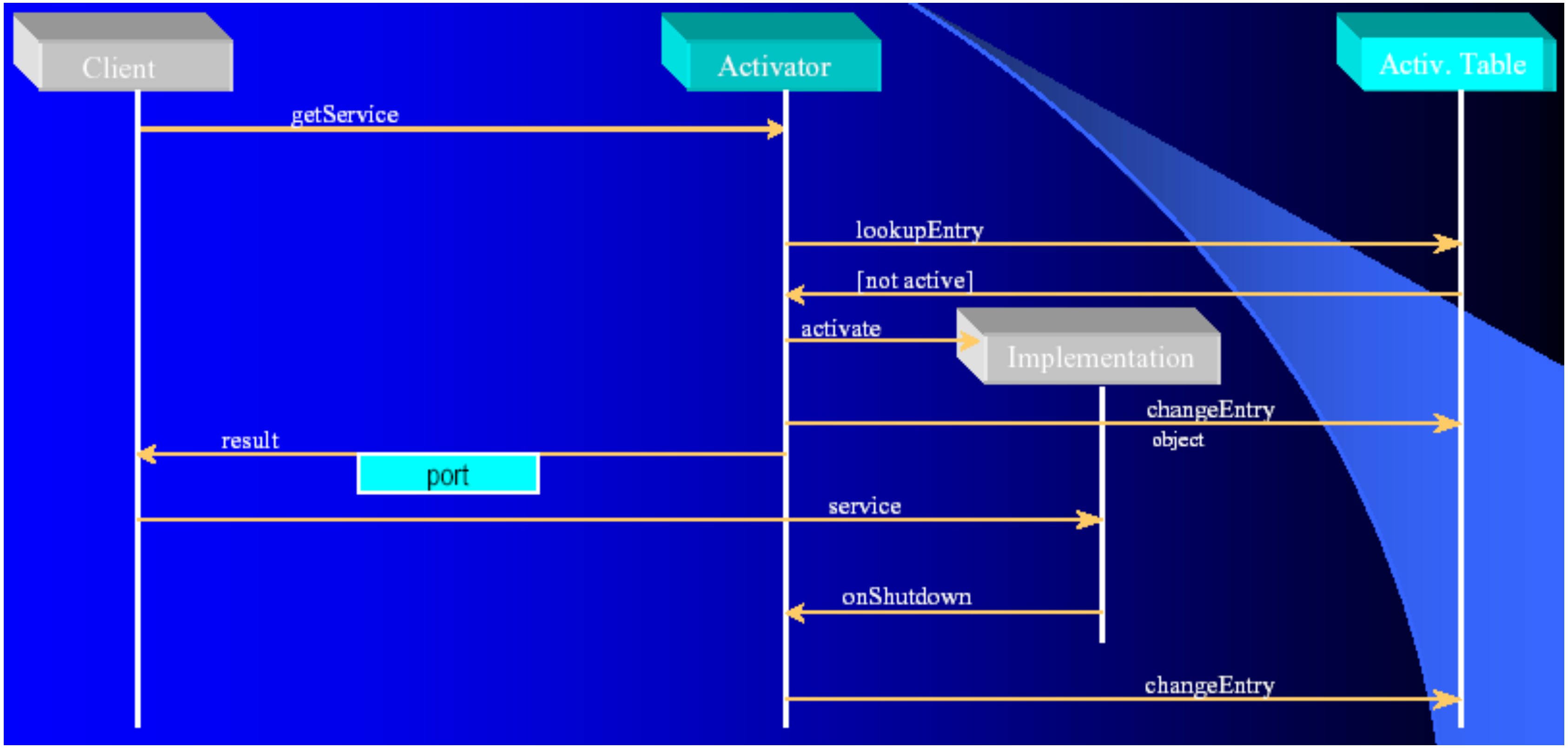
- It is not feasible to have all server implementations running all the time.
- Thus, there must be a way to activate servers on demand.
- Which activation strategy should a broker follow?

Solution



- **Integrate activation code for automatically starting up server implementations. Provide necessary information in tables**:
  - Upon incoming requests the *Activator* looks up whether a target object is already active. If the object is not running it activates the implementation.
  - The *Activation Table* stores associations between services and their physical location.
  - The *Client* uses the Activator to get service access.
  - A *Service* implements a specific type of functionality that it provides to clients.

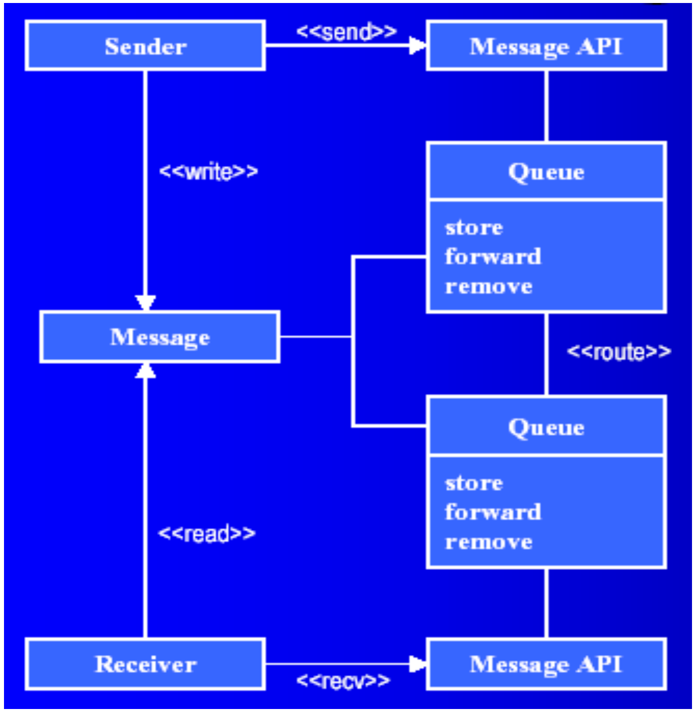
Dynamics: Sample Scenario



Asynchronous and Time Independent Communication

- Using conventional semantics brokers support only blocking communication:
  - Clients must wait until their invocation returns.
    Even if they use multi-threading the underlying brokers will block.
  - A receiver must be online. However, this cannot always be guaranteed.

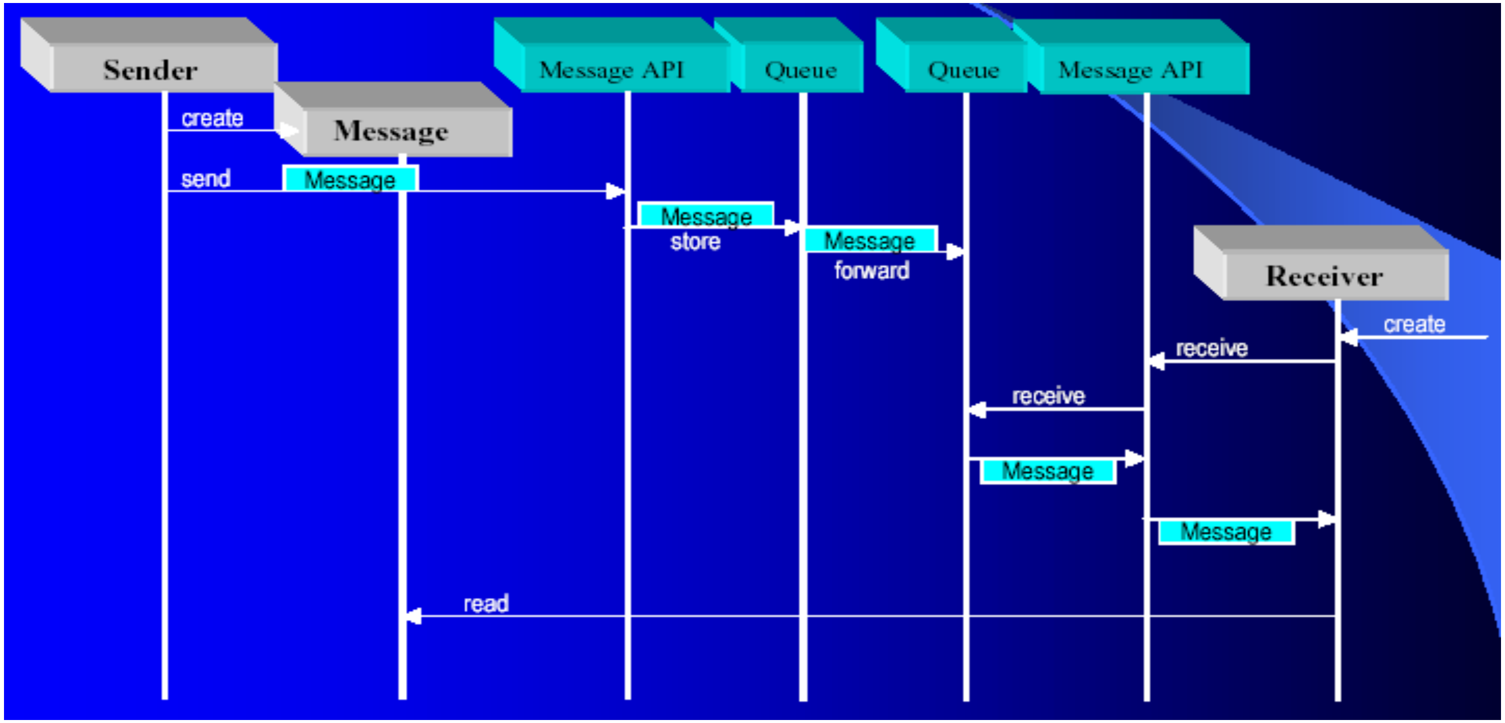
Solution



Solution

- Introduce intermediary queues between senders and receivers :
  - *Queues* are used to store messages persistently.
  - They cooperate with other queues for message routing.
  - *Messages* are objects sent from a sender to a receiver.
  - A *sender* sends messages, while a *receiver* receives them.
  - A *Message API* is provided for senders and receivers to send/receive messages.

Dynamics


Sender
Message
Message API
Queue
Queue
Message API
Receiver
create
send
Message
Message
store
Message
forward
create
receive
receive
Message
Message
read

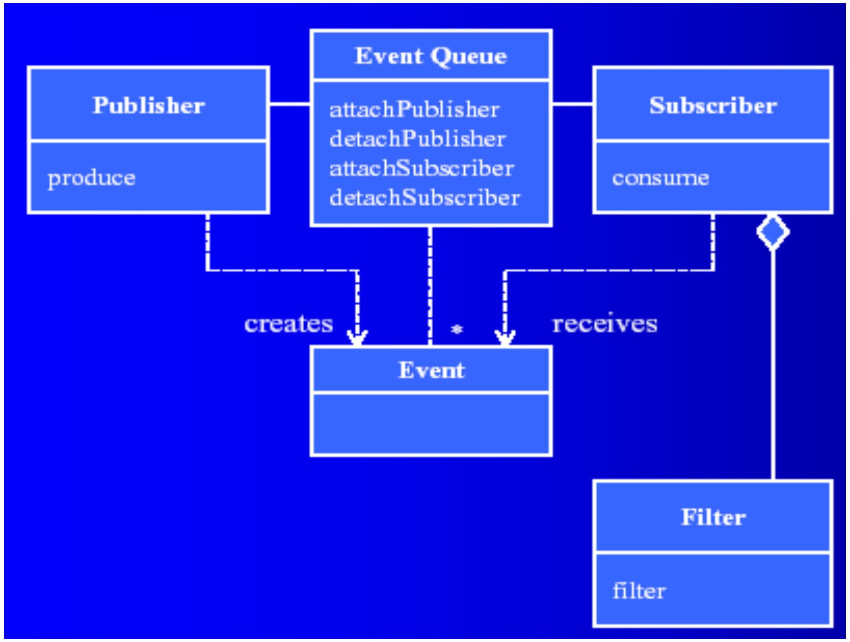
Developer's View

- OO developers prefer request/response semantics:
    - Two message types are used:
        - Requests contain in/in-out arguments
        - Results carry out/inout arguments and results
    - Callback object or Poller object to retrieve result.
    - Player/Recorder strategy: Recorder enqueues request-part of invocation, Player dispatches to actual implementation and enqueues result-part.

Publish/Subscribe

- Usually, a specific client calls a specific remote server and blocks until the result returns.
- Sometimes, this strategy is not sufficient.
  - Consider a server that reports share values.
    - A polling strategy leads to performance bottlenecks.
    - The share values could be spread across different servers.
    - More than one client is interested in the information.
- How can we decouple clients and servers?

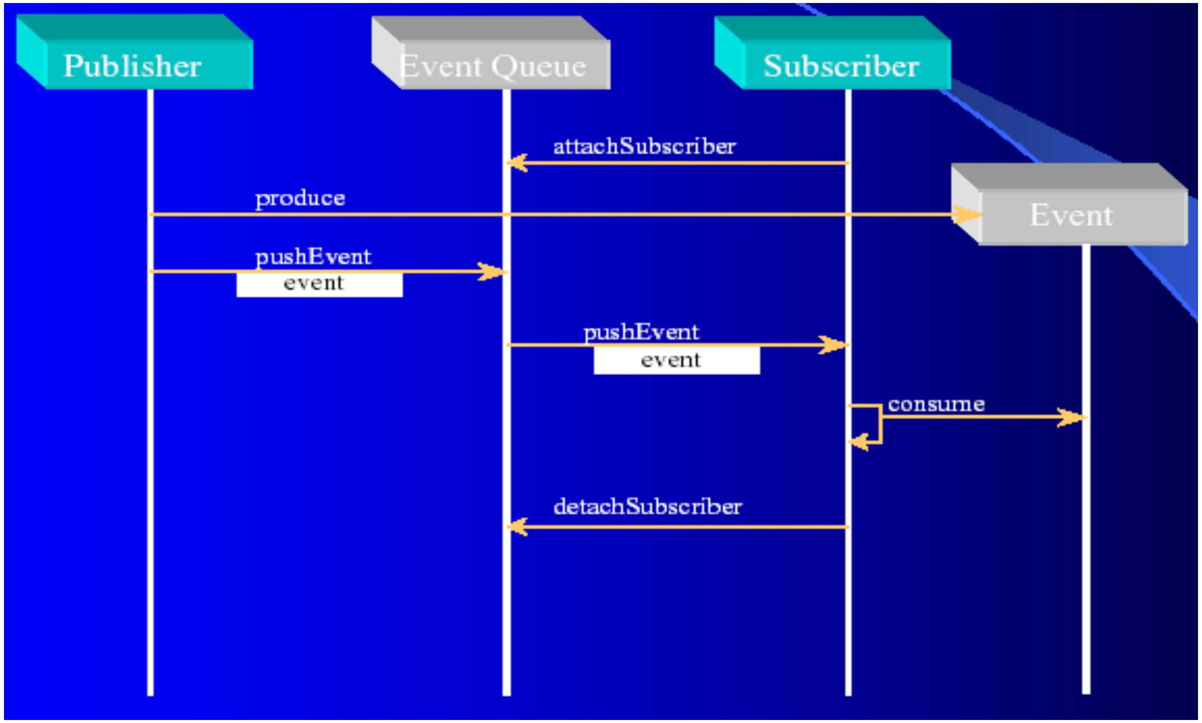
Solution



Decouple suppliers (publishers) and consumers (subscribers) of events:

- An *Event Queue* is storing events.
- *Publishers* create events and store them in an event queue with which they have previously registered.
- *Consumers* register with event queues from which they retrieve events.
- *Events* are objects used to transmit state change information from publishers to consumers.
- For event transmission *push-models* and *pullmodels* are possible.
- *Filters* could be used to filter events on behalf of subscribers.

Dynamics


Publisher
Event Queue
Subscriber
attachSubscriber
produce
Event
pushEvent
event
pushEvent
event
consume
detachSubscriber

Benefits and Liabilities

- **Benefits**
  - Decouples consumers and producers of events.
  - n:m communication models are supported.

- **Liabilities**
  - Must be careful with potential update cascades.

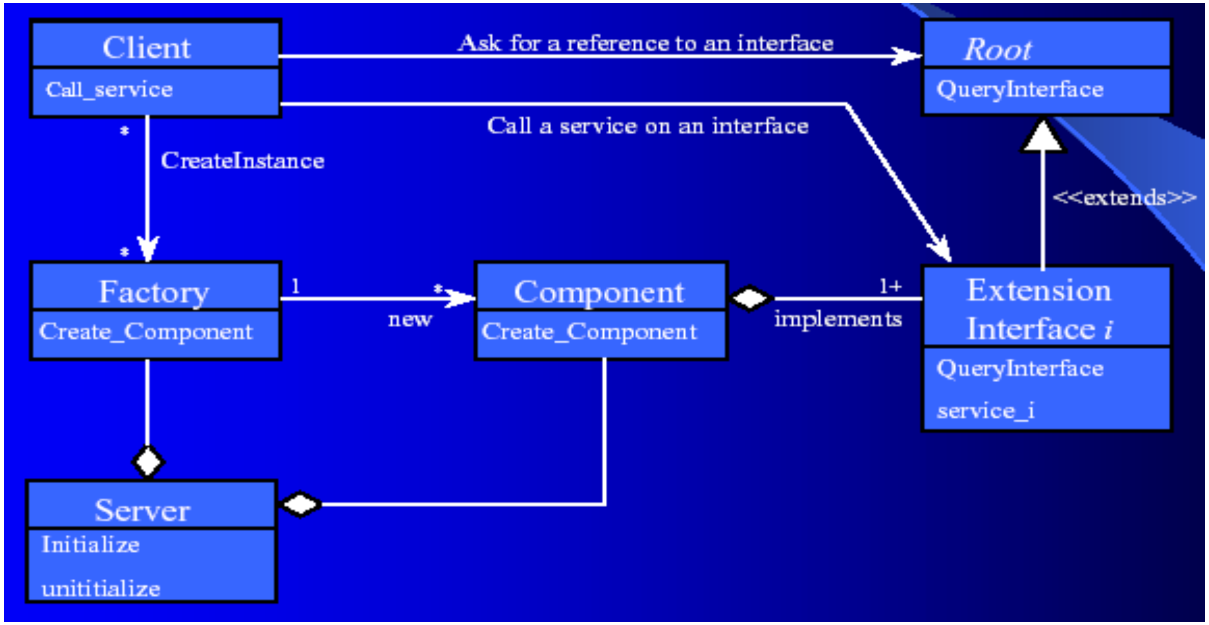
How to provide and evolve component functionality

- Every Component Model defines how components should import and export functionality.
- Components should be able to evolve over time without impact on clients.

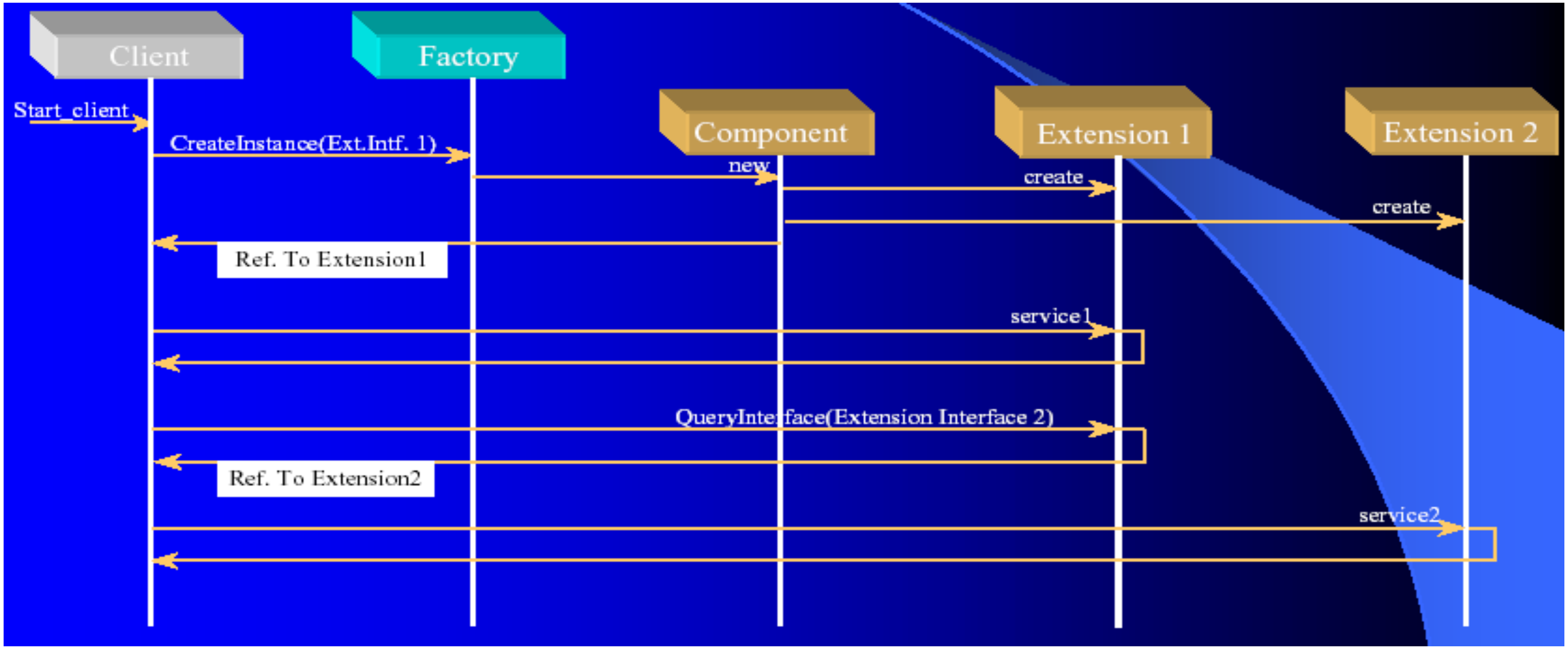
Multiple Interfaces

- Most objects have only one interface.
- Thus, if we want the banana, we get the whole gorilla.
- How can we make this more flexible?

Extension Interface Pattern



Dynamics


Client
Factory
Component
Extension 1
Extension 2
Start_client
CreateInstance(Ext.Intf. 1)
new
create
create
Ref. To Extension1
service1
QueryInterface(Extension Interface 2)
Ref. To Extension2
service2

Benefits and Liabilities

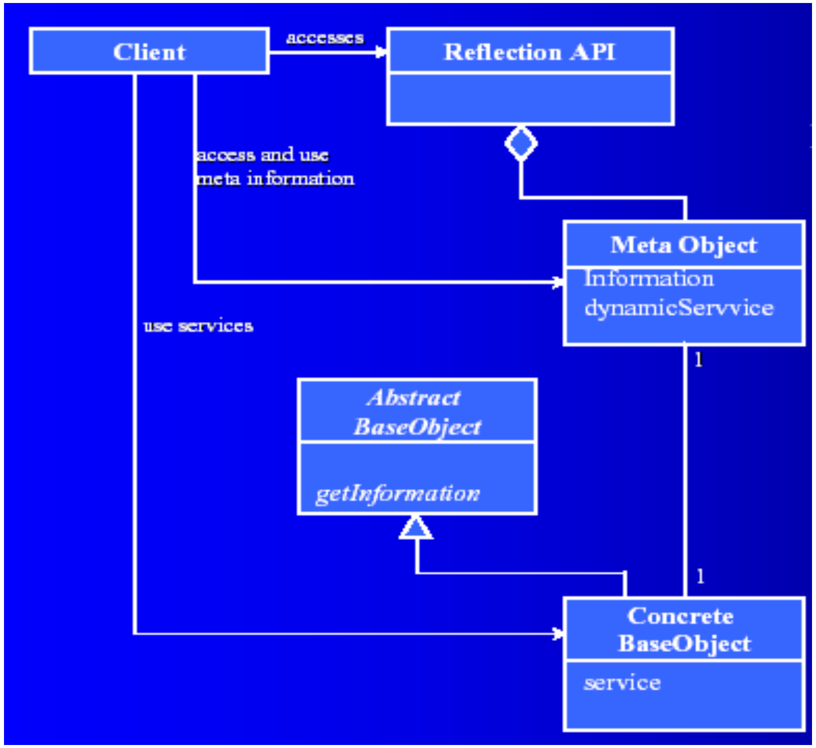
- Benefits
  - Exchangeability of components
  - Extensibility through interfaces
  - Prevention of interface bloating
  - No subclassing required
  - Separation of concerns

- Liabilities
  - Restricted efficiency due to indirection
  - Complexity and cost for development and deployment

Dynamic Environments

- Until now, clients can only access services that are available at compile-time.
- Typically, in distributed environments services must be dynamically added, exchanged, or removed.
- How can we access objects that are not known at deployment-time?

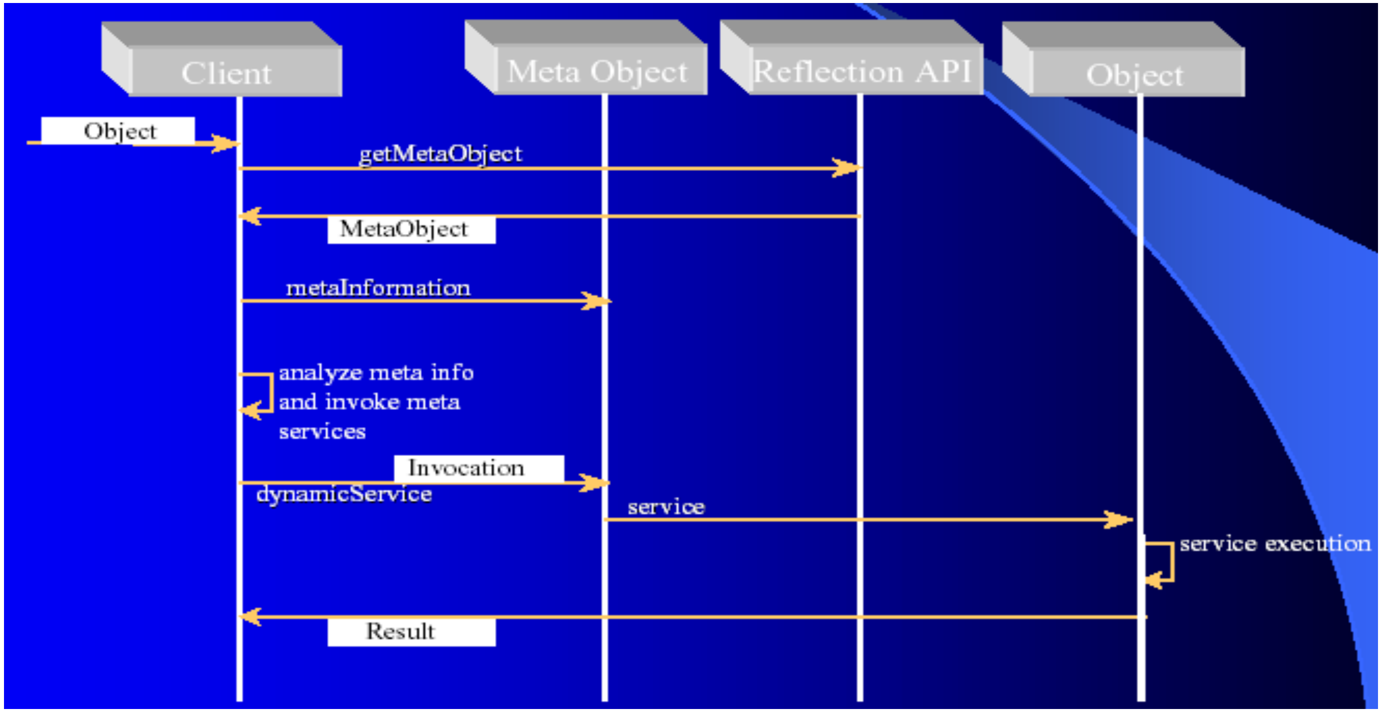
Simplified Reflection Pattern



Solution

- Provide type information through meta objects and allow clients to find and use this meta information:
  - *Meta Objects* provide information and functionality about existing base objects.
  - The *Reflection API* is responsible for creating and retrieving Reflection Objects.
  - *Base Objects* implement the application functionality.
  - *Clients* access Meta Objects to dynamically invoke base functionality.

Dynamics


Client
Meta Object
Reflection API
Object
Object
getMetaObject
MetaObject
metaInformation
analyze meta info
and invoke meta
services
Invocation
dynamicService
service
service execution
Result

Benefits and Liabilities

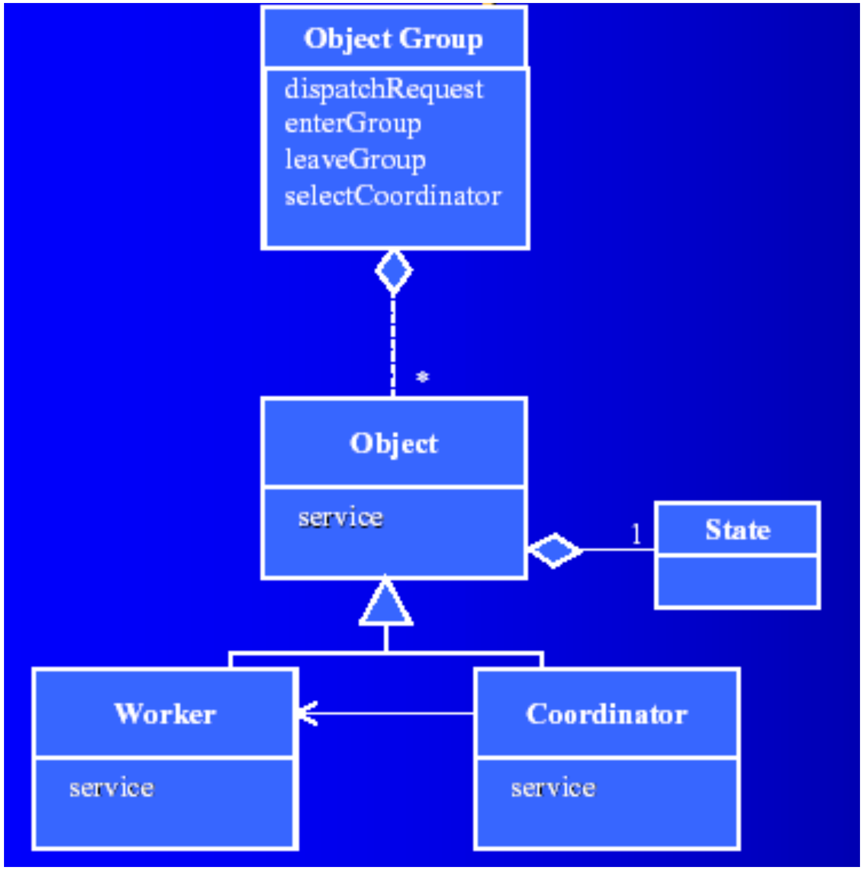
- Benefits
  - Support for change.
  - Clients may dynamically invoke objects without compile-time knowledge.
  - Support for visual builder tools and for scripting environments.
  - Base Objects may be added at run-time.

- Liabilities
  - Possible performance problems.
  - Implementation harder to maintain and understand.

Fault Tolerance

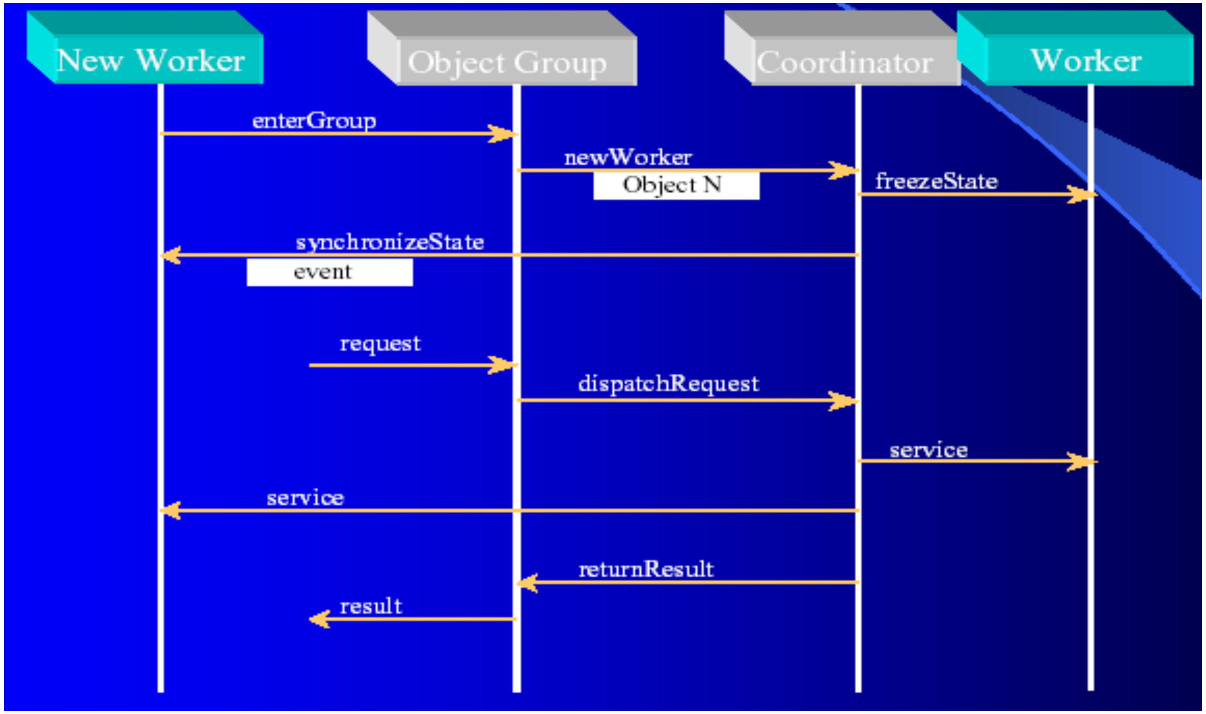
- To enhance fault tolerance, more than one server should provide the same service.
- It should be transparent to the client that replicated servers are used.
- How can we introduce this kind of transparency?

Object Group Pattern



- **Decouple suppliers (publishers) and consumers (subscribers) of events:**
  - An Object Group represents a group of objects that all provide the same service.
  - Object provides a specific service to clients.
  - The coordinator is selected by the Object Group to dispatch requests.
  - Workers receive requests from their coordinator and return results.
  - Each member of the object group maintains its own state.

Dynamics


New Worker
Object Group
Coordinator
Worker
enterGroup
newWorker
Object N
freezeState
synchronizeState
event
request
dispatchRequest
service
service
returnResult
result

Benefits and Liabilities

- **Benefits**
  - Resilience to failures in distributed object implementations.
  - Clients unaffected by object group internals.
  - State remains consistent.

- **Liabilities**
  - Overhead due to several indirection layers.

Conclusions

- Software architecture patterns as
  - a blueprint for architectural solutions
  - guidance for benefits and liabilities
  - for architecture assessment