OO Remoting

Foundations of OO remote procedure call middleware

With CORBA and .NET Remoting Examples

Markus Völter
voelter@acm.org
www.voelter.de

Patterns by Völter, Zdun, Kircher

Contents

Contents

Introduction: Sync vs. Async

- There are two main **remote communication styles**:
  - (OO-)RPC: Remote Procedure Call
  - MoM: Message oriented middleware
- Differences: **Interface and identity**
  - In **RCP-style remoting**, a caller knows the identity of the target object, it also knows its interface.
  - When calling an operation on a remote object, a caller knows that the target oject can handle the operation.
  - In **MoM-based systems**, a client posts a message (of a specific type or format) to the middleware (queues, topics, …) and does not know, which entity handles the message.

Introduction: Sync vs. Async (II)

- Differences: **Temporal coupling**
  - In **RPC-style remoting**, the invoker of an operation expects the target object to be available and handle the invocation rather instantly.
  - In **MoM-based systems**, a sender sends a message and does not necessarily know if and when it will be handled (i.e. if a receiver is online or not).
- Basic concepts are independent of the API presented to the programmer.
  - It is possible, to "emulate" MoM-based systems based on RPC-style remoting,
  - It is also possible to implement RPC-style remoting on top of a message-oriented middleware.
- This **presentation covers RPC-style remoting**, including some patterns for asynchronous APIs based on the RPC-style.

Introduction: Remoting Challenges

- Building OO RPC-style remoting frameworks involves a **set of challenges** that need to be taken into account:
- The **API towards the programmer shall be as simple as possible**. Ideally, no there should be no difference between local and remote method invocations.
- Remote invocations can lead to **new failure modes** compared to local invocations. The framework, and the API, has to take these into account.
- As in any development task, **performance should be as good as possible**. This might have different implications for different environments.

Introduction: Remoting Challenges (II)

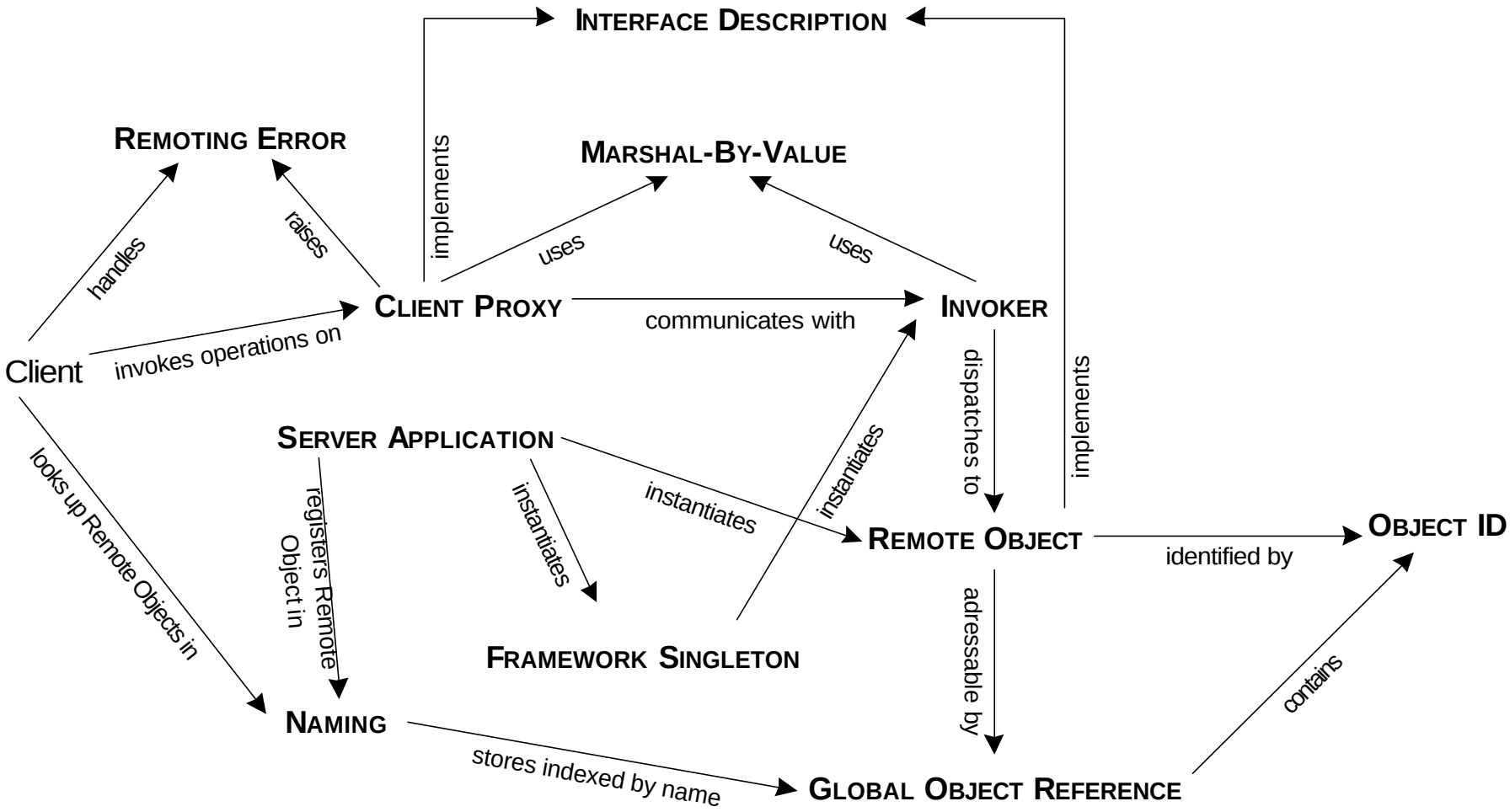
- The concepts, and perhaps even a specific framework, must be able to adapt to different **quality of service requirements**; embedded applications have different requirements than enterprise stuff.
- Clients and server developers should not need to know about the **operating system or hardware architecture** on which the communication partner runs.
- And, again as usual, the framework should be **flexible** with regards to some of its behavior or policies.

Introduction: The Pattern Language

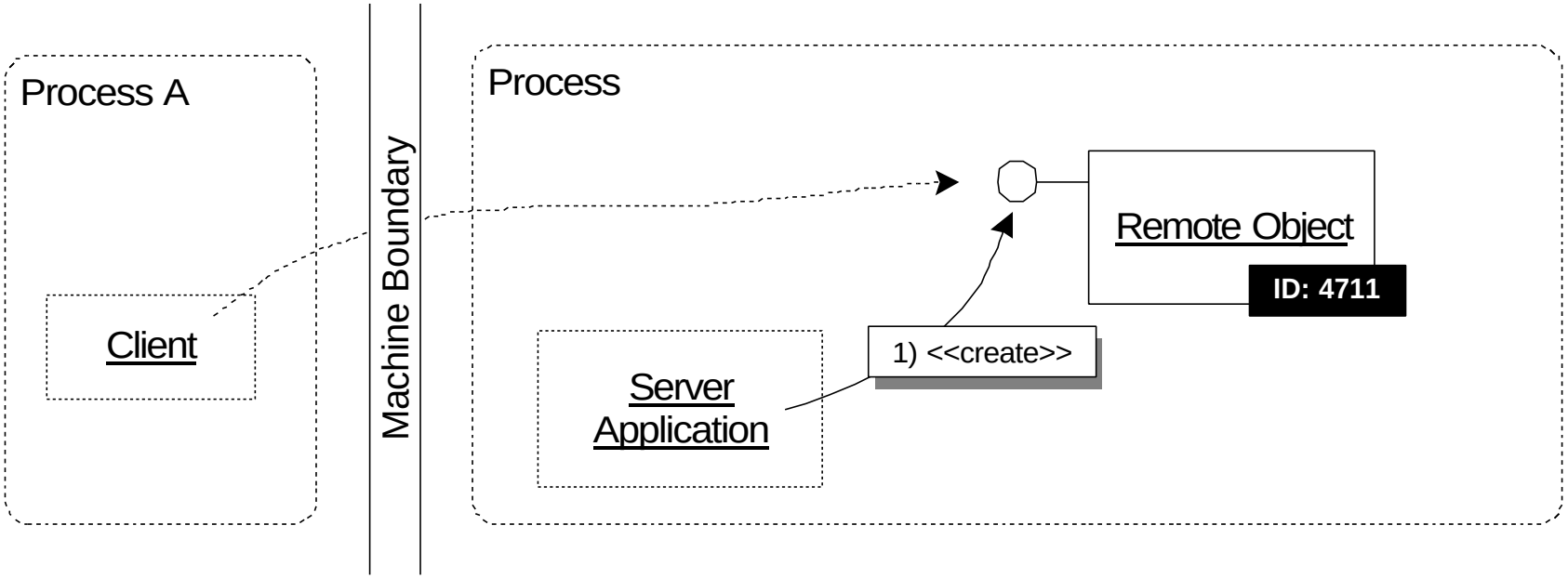
- This work is based on a **pattern language** by Markus Völter, Uwe Zdun and Michael Kircher.
- You can find further information on this topic at **www.voelter.de/remoting**
- Feedback is welcome!

Contents

Basic Remoting Patterns

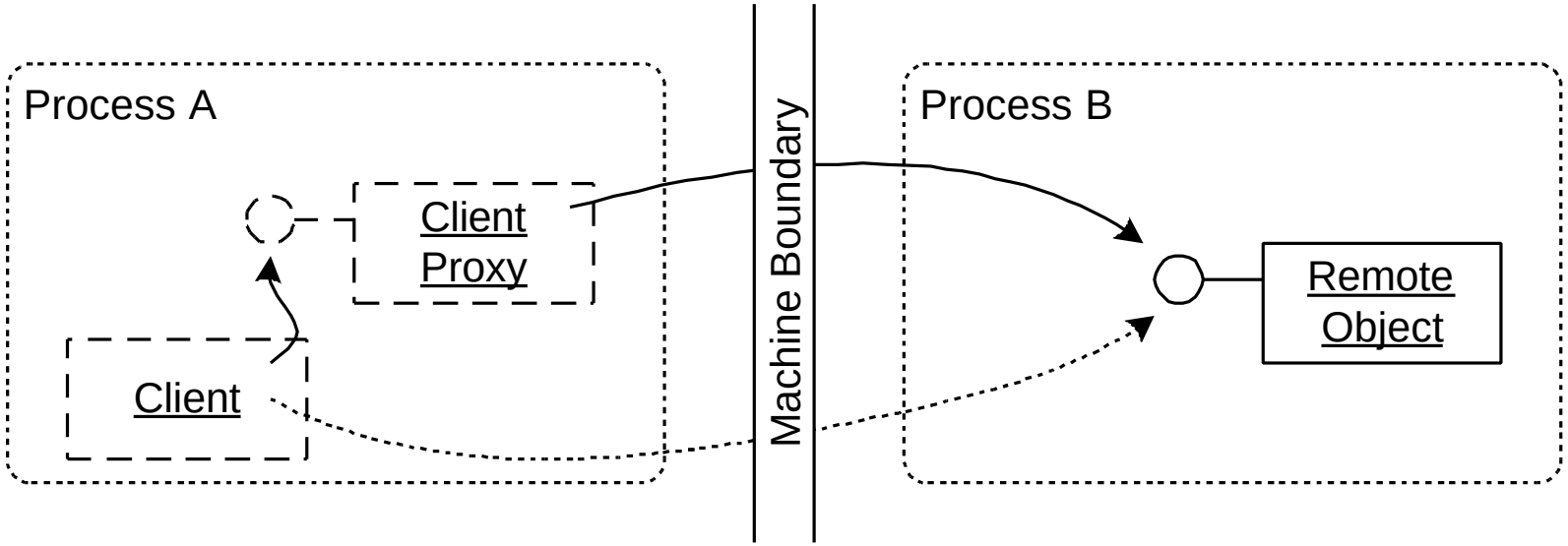


Remote Object



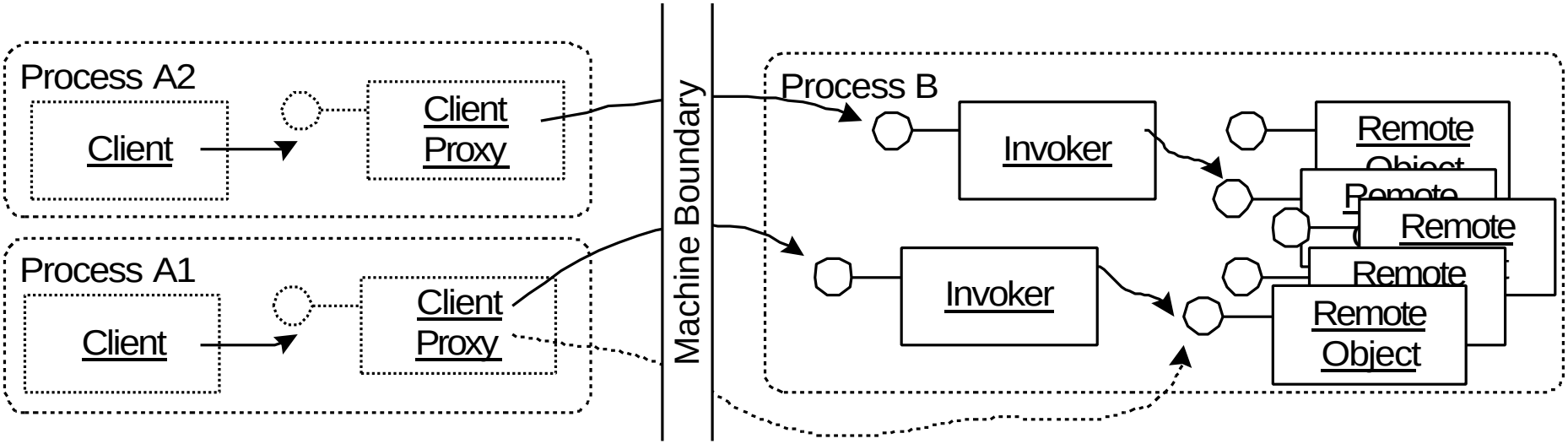
Provide REMOTE OBJECTS as the building blocks for distributed applications. They have a unique OBJECT ID in their local address space, as well as means to construct a GLOBAL OBJECT REFERENCE from this OBJECT ID. The GLOBAL OBJECT REFERENCE is unique in the "global" address space of the network. Usually, each REMOTE OBJECT has also a well-defined INTERFACE DEFINITION that is separated from the object's implementation. REMOTE OBJECTS provide facilities to allow them to be managed by the SERVER APPLICATION.

Client Proxy



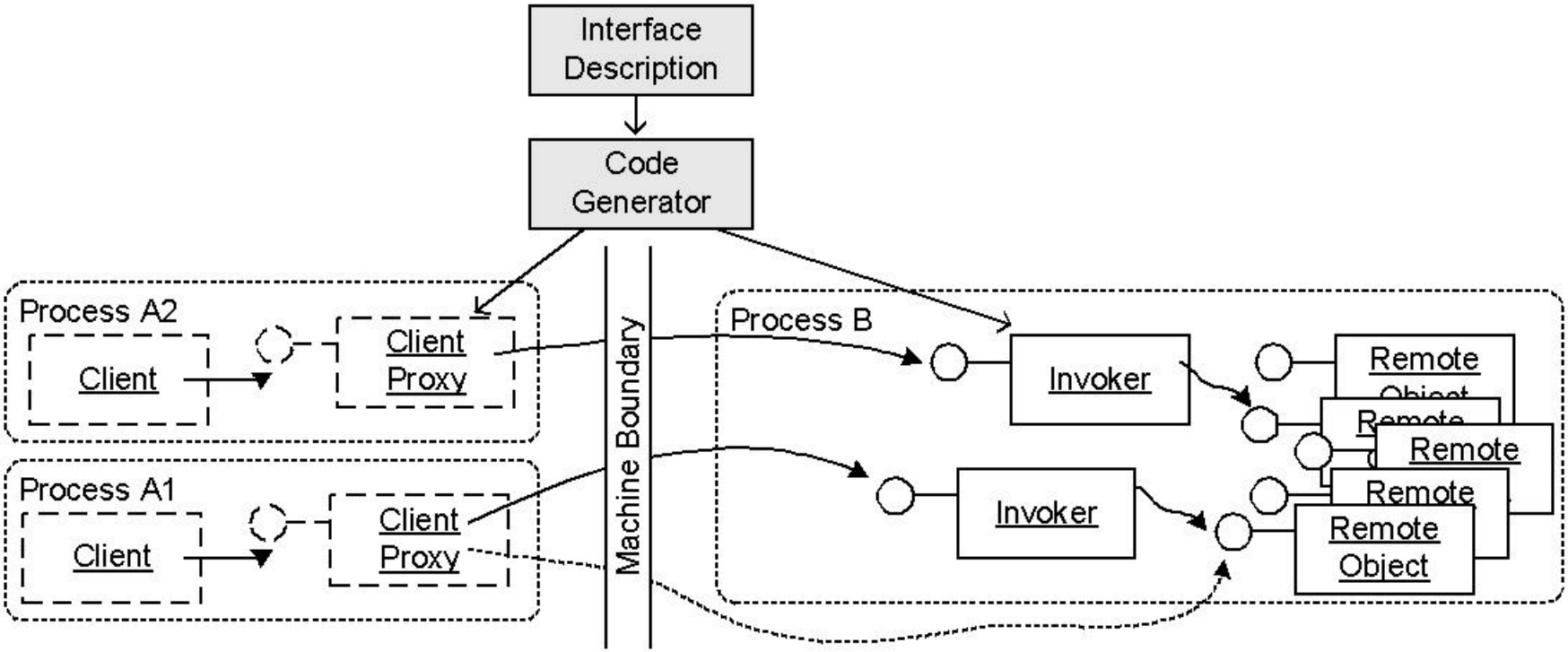
Provide a PROXY object in the client process for the REMOTE OBJECT. The PROXY object has the same interface as the REMOTE OBJECT. For remote invocations, clients only interact with that local PROXY object. The PROXY primarily forwards calls to the REMOTE OBJECT, and it is responsible for the details of accessing the REMOTE OBJECT via the distributed object system. Only those remoting details that cannot be handled automatically are visible to the client developer.

Invoker



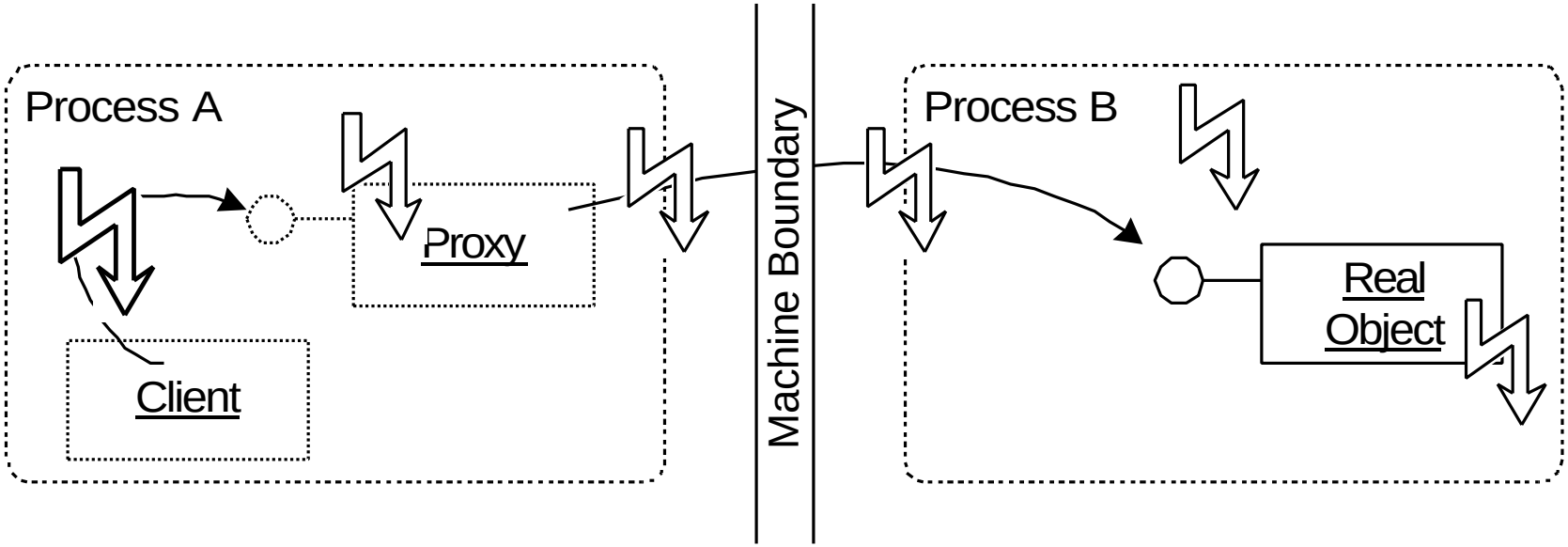
Provide an INVOKER that is remotely addressable. It receives invocations for many instances and forwards them to the actual instance. To make this possible, the data describing the invocation must contain the ID of the actual target object in an INVOCATION CONTEXT. Resolving the ID, as well as other application-specific dispatching and adaptation tasks, handled on the INVOKER, are transparent to the client.

Interface Description



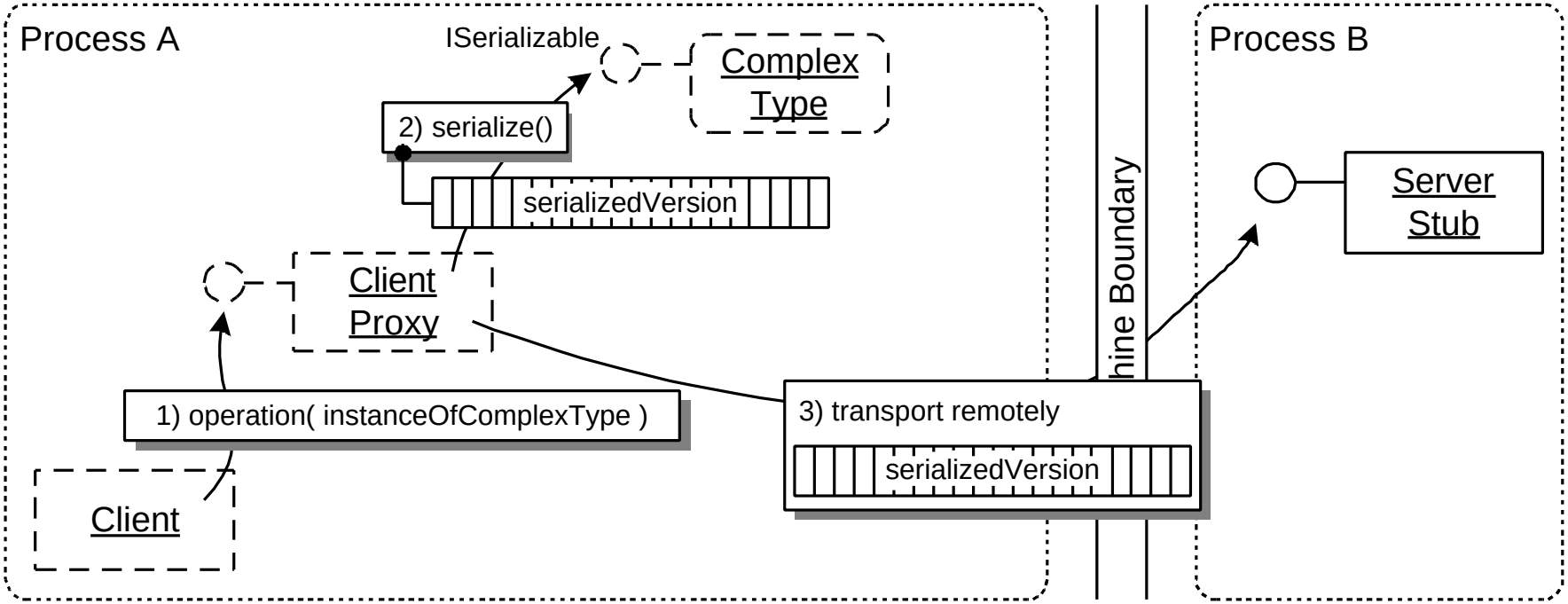
Provide an INTERFACE DESCRIPTION as a meta description in which you describe the common interfaces implemented on client side and server side once. Use a code generator to automatically produce the recurring code fragments.

Remoting Error



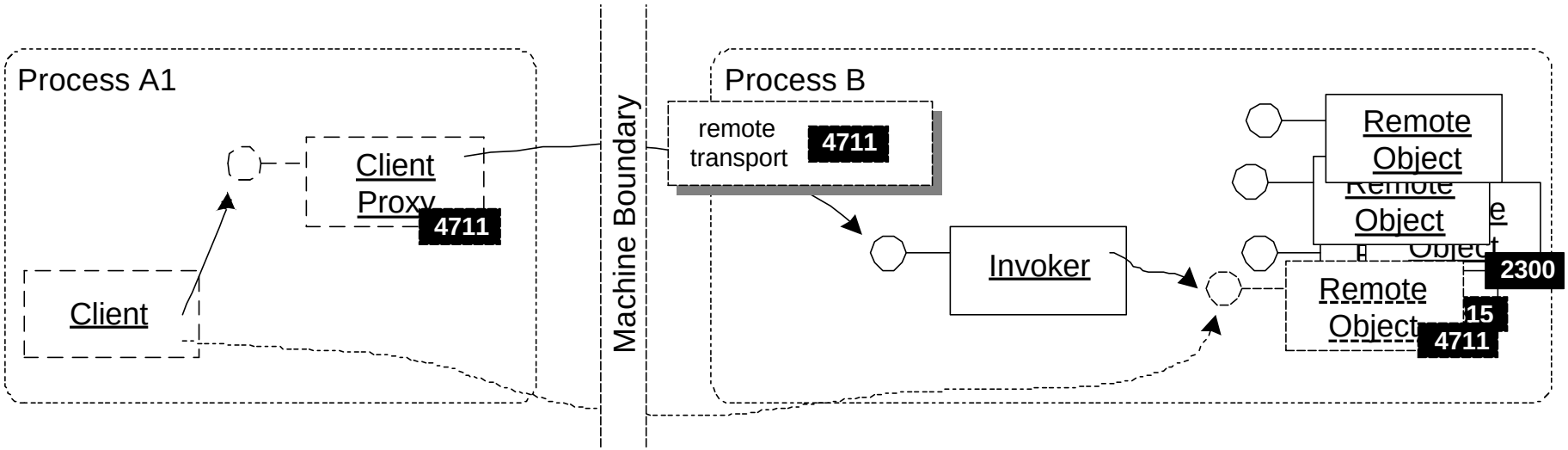
Make sure that the CLIENT PROXY throws a specific kind of exception whenever the remote server object or the network connection runs into problems. The client should be able to handle these exceptions in a catch-all style, not caring about the details. But the exception should also contain enough information to allow the client to consider alternative actions.

Marshal-By-Value



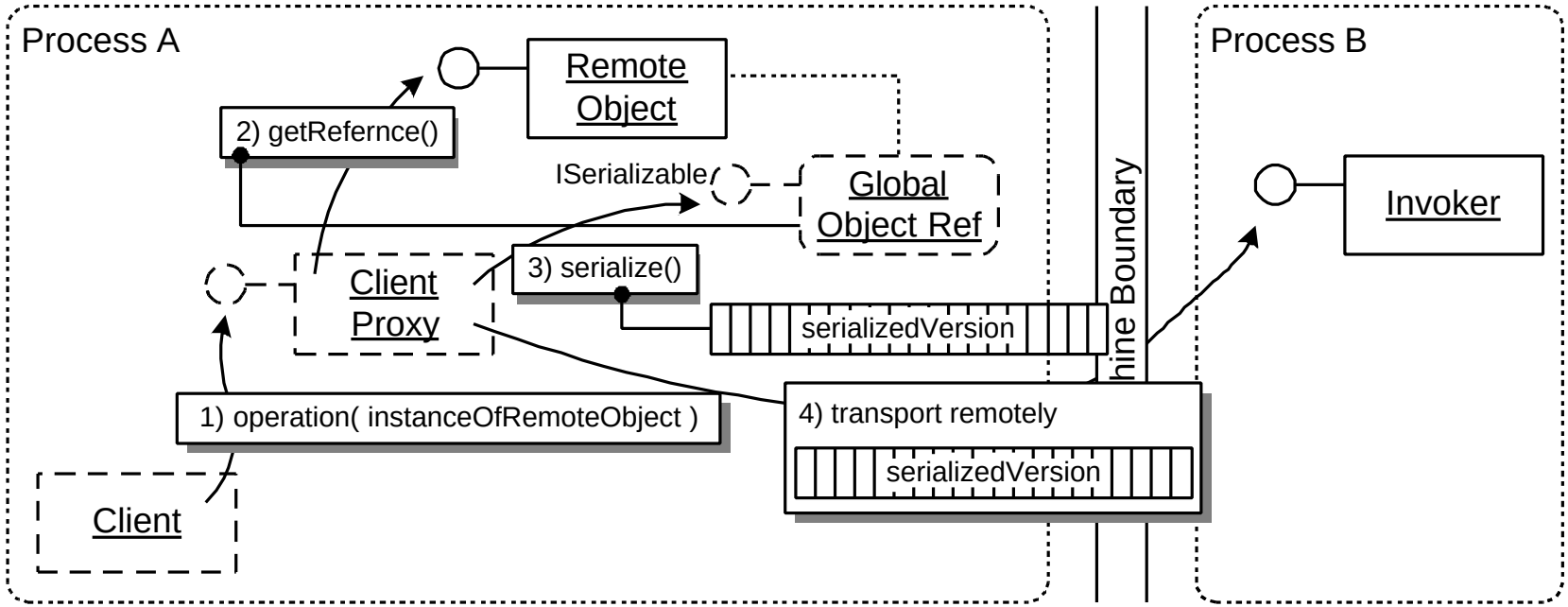
For each non-remote and non-primitive types, require that they provide a way to convert themselves into a lower level "wire format," a process known as serialization. This format is usually constructed by marshalling the complex types by value. The distributed object system will use this mechanism when automatically marshalling complex type objects.

Object ID



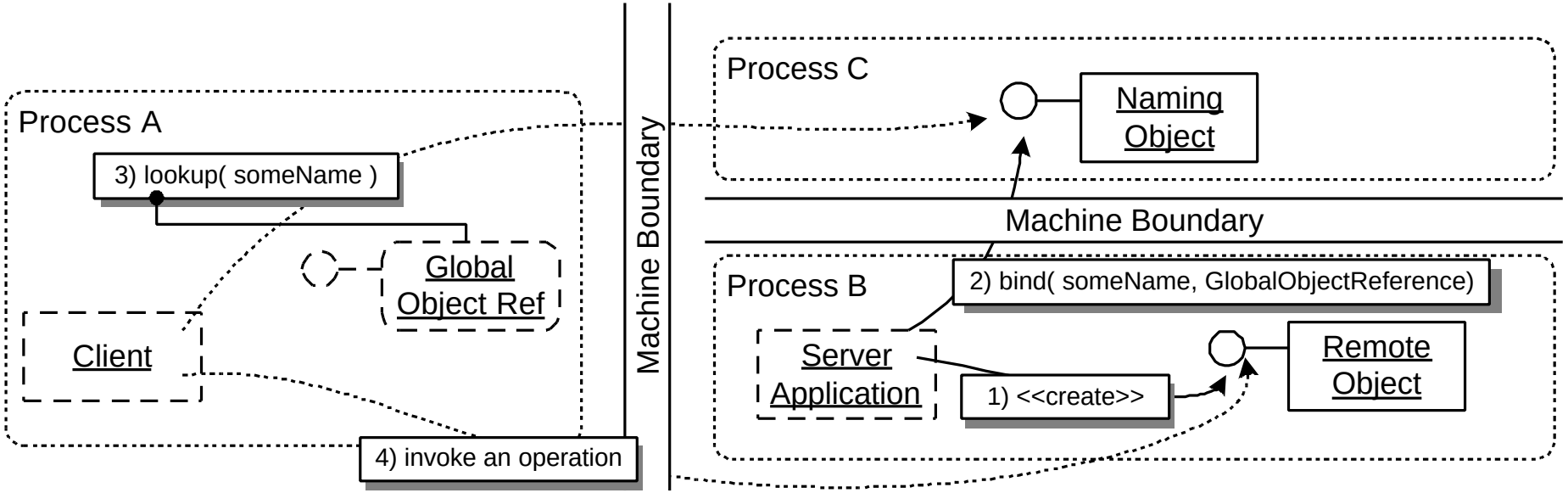
Associate each REMOTE OBJECT instance with a globally unique remote OBJECT ID. Comparisons for equality can be based on this ID. Remote invocations from the CLIENT PROXY to the INVOKER will contain this ID to allow the INVOKER to dispatch the invocation to the correct object. The CLIENT PROXY needs to know this ID in order to supply it to the INVOKER for each remote invocation.

Global Object Reference



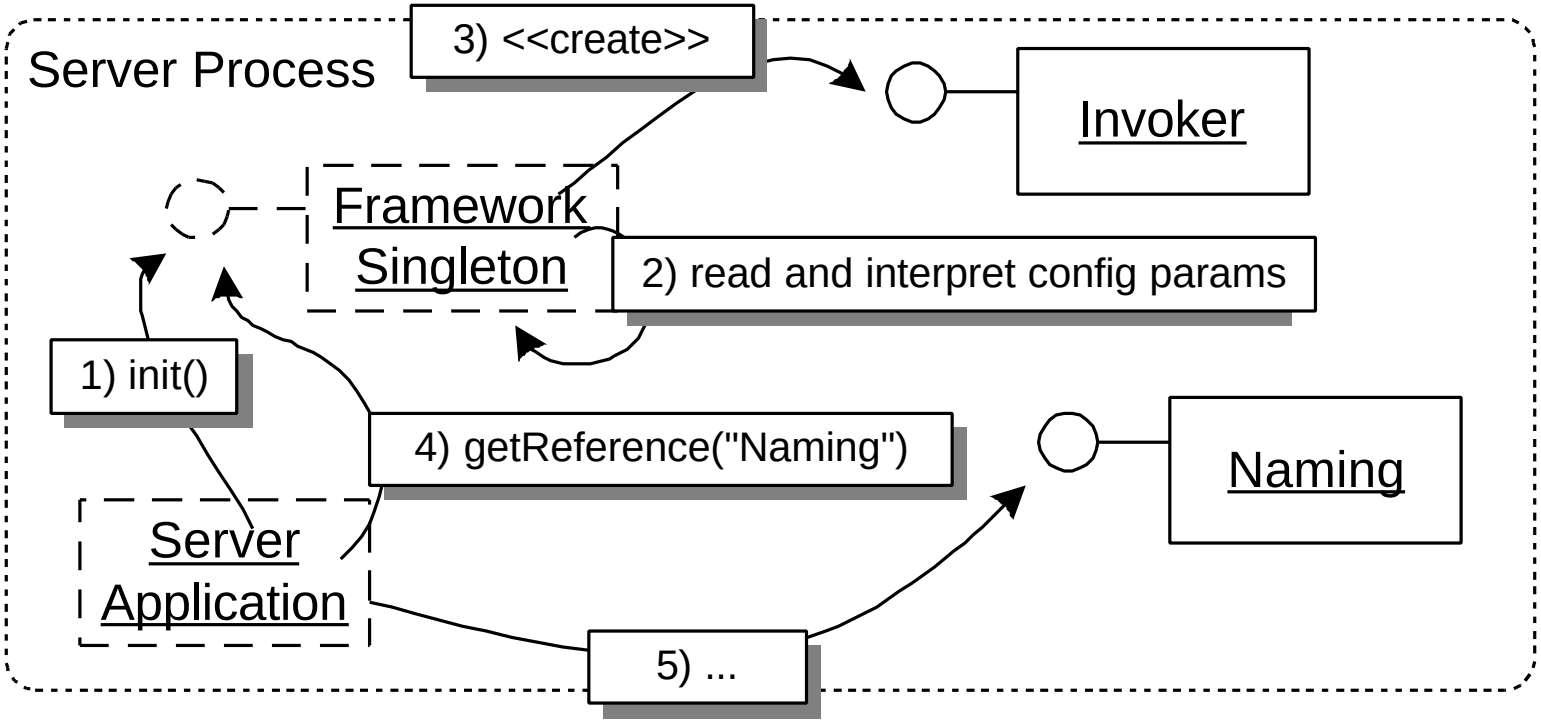
Provide GLOBAL OBJECT REFERENCES which serve as "pointers" to REMOTE OBJECTS. Such a reference needs to include all the information that is necessary for a client to reach the REMOTE OBJECT over the network, including the target object's OBJECT ID. To sent the GLOBAL OBJECT REFERENCE over the network it is MARSHALLED-BY-VALUE, but not the target object. Upon receipt of such a reference, the client can set up a network connection to the REMOTE OBJECT using the information in the GLOBAL OBJECT REFERENCE, typically by instantiating a suitable CLIENT PROXY.

Naming



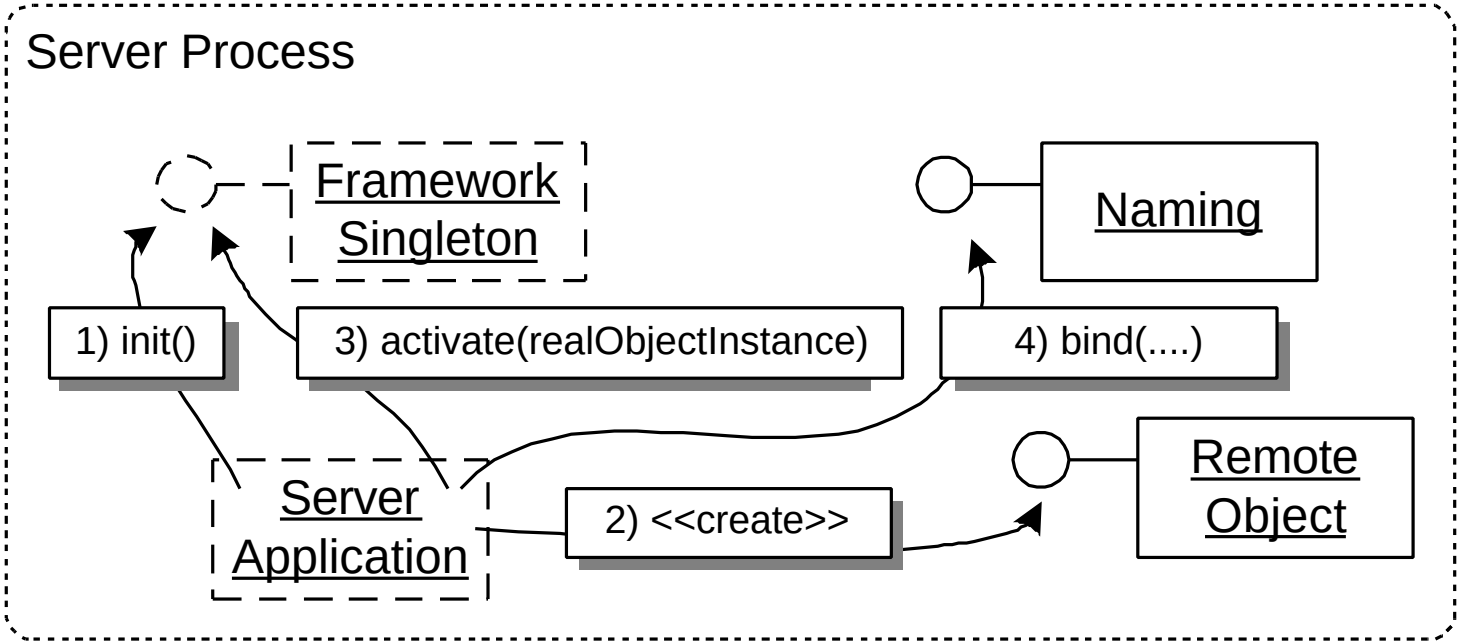
Provide a way to resolve names to GLOBAL OBJECT REFERENCES in a NAMING system. The NAMING system is typically also implemented as a REMOTE OBJECT. All clients have access to this NAMING system by default; that is, its location and reference is well known and configurable. SERVER APPLICATIONS can register ("bind") the REMOTE OBJECTS they wish to publish in NAMING. Thus clients only need to know the symbolic names of the REMOTE OBJECTS required to find out the GLOBAL OBJECT REFERENCE.

Framework Singleton



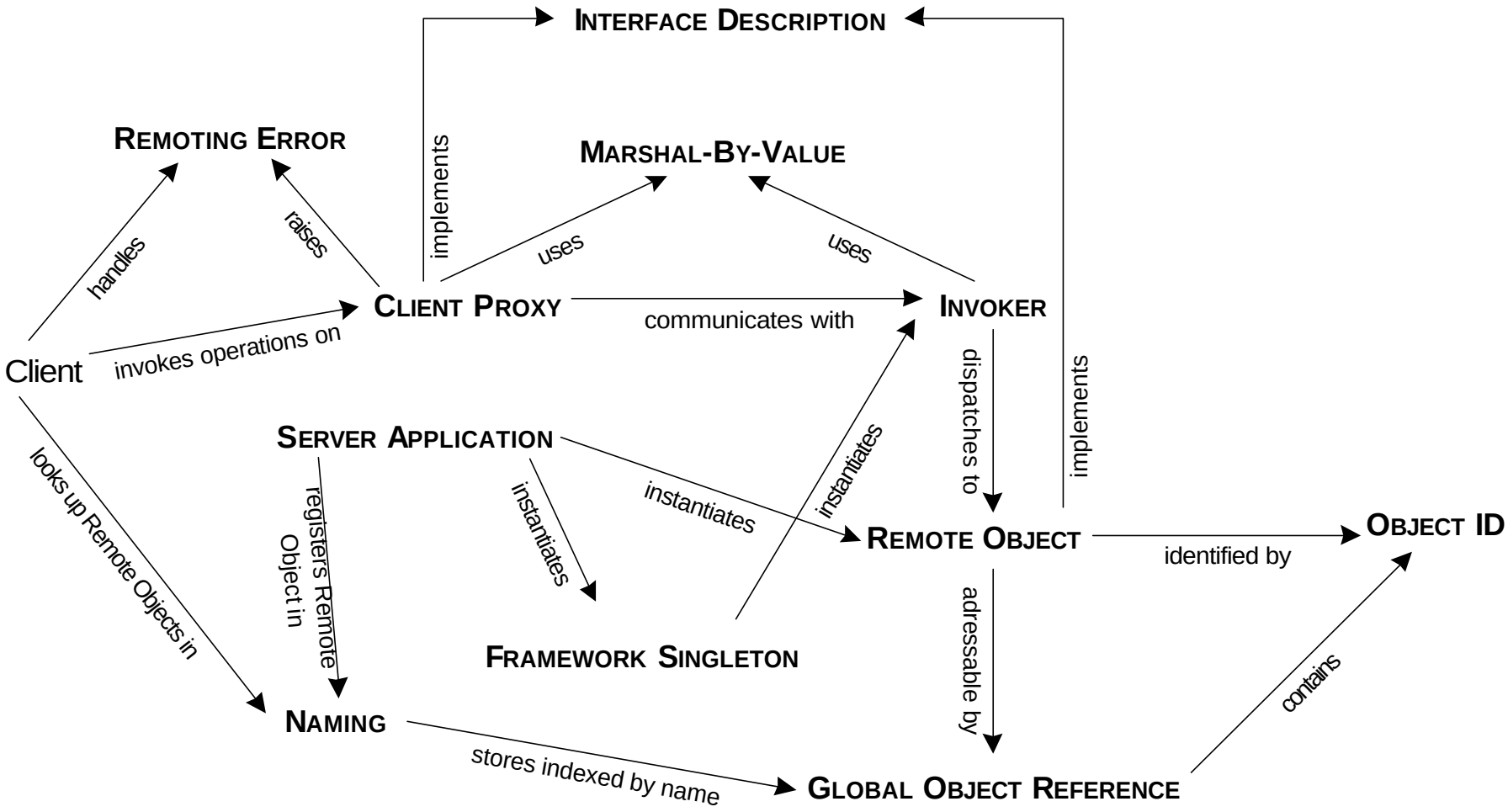
Provide a global SINGLETON that serves as a FACADE to the underlying distributed objects framework. It serves as the application developer's single access point and administration API to the framework and can be used as a REMOTE FACTORY for other objects, supplied by the distributed object system, the developer might need access to.

Server Application



Provide a SERVER APPLICATION whose job it is to initialize the distributed object system infrastructure, if it is not already initialized by another SERVER APPLICATION. The SERVER APPLICATION can do this by instantiating and configuring the FRAMEWORK SINGLETON. Moreover, it resolves initial, pre-configured references such as NAMING and instantiates REMOTE OBJECTS, or prepares for instantiation, according to the selected instantiation strategy.

Basic Remoting Patterns



Basics - CORBA

- The basic concept in CORBA are, of course, **Remote Objects**.
- Their **Interface Description** is done in CORBA IDL, a descriptive language that looks like C/C++.
  - IDL interfaces have modues (namespaces), interfaces, operations, exceptions, structs, and more.
- Implementations, so-called servants, can be provided in many different programming languages (e.g. C/C++, Java, Python, Ada, Eiffel) and on different operating systems (different Unixes, Windows *, Java, ..)
- A **Client Proxy** is automatically generated by the IDL compiler for a specific language. It is a normal programming language class/object.

Basics - CORBA

- The **Invoker** is called Skeleton in CORBA terminoloy. It is also automatically generated by the IDL compiler delivered with the ORB.
- CORBA provides exceptions to report error conditions. **Remoting Errors** are expressed using specific standardized CORBA exceptions.
  - A developer can also define their own application specific exceptions for the operations declared in the IDL interface.
  - CORBA exceptions contain a flag *completed* that specifies whether the operation on the server has been completed or not (or *maybe* ✍)

Basics - CORBA

- There are different ways to **transport (marshal) data by value**:
  - Primitive (CORBA-) data types are always transported by value; the ORB can do that natively
  - Developers can define *structs* as part of their IDL interfaces; they are marhalled by value
    - There are some limitations with regard to structs – details ommitted
  - CORBA also provides the so-called Objects-By-Value facility, which provides marshal-by-value for objects.
    - In contrast to structs, OBV valuetypes can implement IDL interfaces
  - Note that there is currently a mapping from OBV valuetypes to XML DOM in progress.
- CORBA objects (instances of implementations of IDL interfaces) are always marshalled by reference.

Basics - CORBA

- Every CORBA object (instance!) has its own unique **Object ID**. The format is not standardized, depends on the ORB and is opaque to clients.
- There are different kinds of **Object References**, depending on the underlying transport protocol (which is exchangeable).
  - For the standard protocol IIOP there are the so-called IOR (Interoperable Object References) which can be serialized to a string.
  - IORs contain the object server's adress, and information on the port the server uses.
  - Other object references may contain other „endpoint information".

Basics - CORBA

- CORBA provides a two-stage **Naming** facility:
  - Generally, objects are named using the CORBA Naming Service, a service defined in IDL that can be accessed just like any other Remote Object. It maps structured names to object references.
  - Well-known objects (such as the Naming service itself or the current transaction) can be resolved by the ORB itself.
    - Some are Pseudo Objects, such as the current transaction
    - Others (like Naming) have to be configured in the ORB manually (bootstrapping problem).
- There is also a CORBA Object Trading Service defined allows clients to lookup objects based on properties using more or less complex queries.

Basics - CORBA

- To access the ORB and ist services, CORBA provides a **Framework Singleton**, typically accessible through the *ORB* class (depends on the language mapping)

- **Server Applications** are programs in the server's language that typically initialize the ORB, lookup specific well-known objects (such as Naming), instantiate remote objects and publish them in Naming.

Basics - .NET

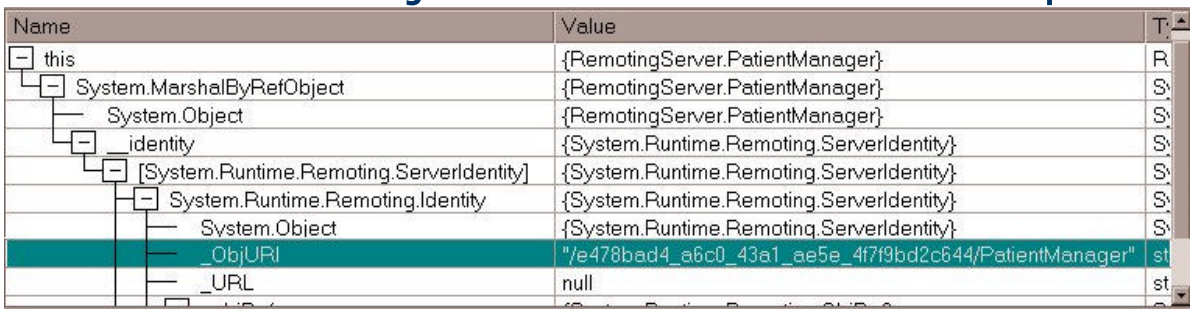
- The basic concept in .NET Remoting are, of course, also **Remote Objects**.
- Normal .NET classes become remote objects by letting them implement the MarshalByRefObject interface.
- Separate **Interface Descriptions** are not neccessary. However, it is possible and advisable (to simplify deployment) to use separately defined interfaces.
  - Normal .NET language interfaces are used here.
- The **Client Proxy** is dynamically generated at runtime using .NET's reflective features. No manual code generation step is neccessary.
  - This is only possible because all .NET programs run inside the CLR infrastructure that can take care of these issues.

Basics - .NET

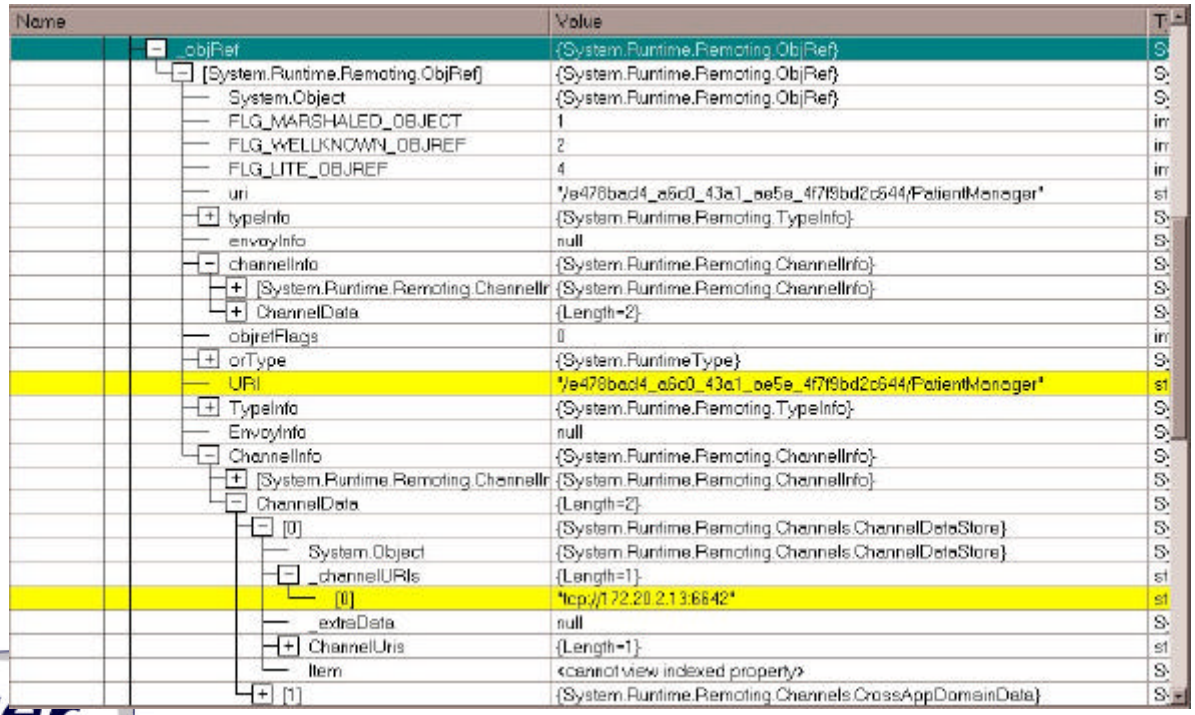
- The **Invoker** is an intrinsic part of the .NET CLR infrastructure. Again, no code generation is necessary.
- .NET also provide a proper exception concept:
  - **Remoting Errors** all inherit from *SystemException*.
  - There is a convention to let application errors inherit from *ApplicationException*.
- Objects that should be **marshalled by value** need to have the *[Serializable]* attribute.
  - You can also implement the *ISerializable* interface in addition to that to allow customized serialization support.
  - The serialization format depends on the currently configured formatter (can be binary, XML, or something else).

Basics - .NET

- .NET remote object instances have unique IDs:



- Global Object References contain communication endpoint info and the target object ID.

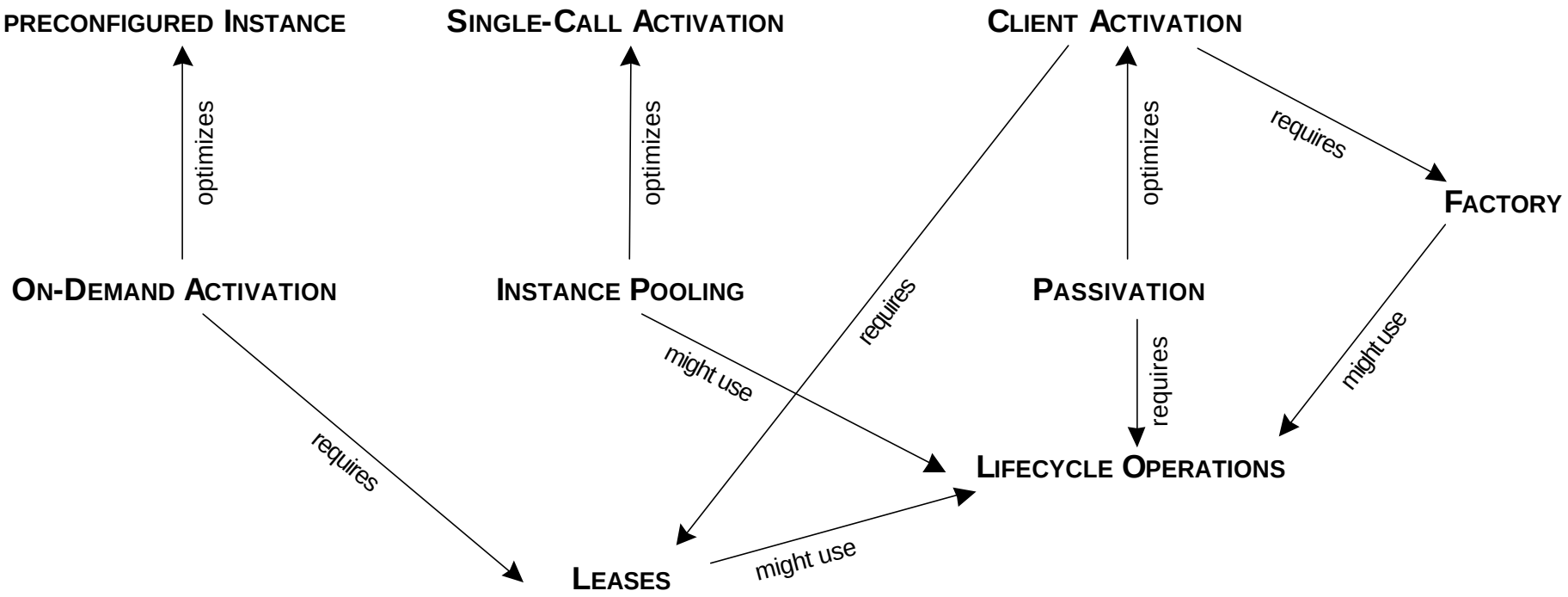


Basics - .NET

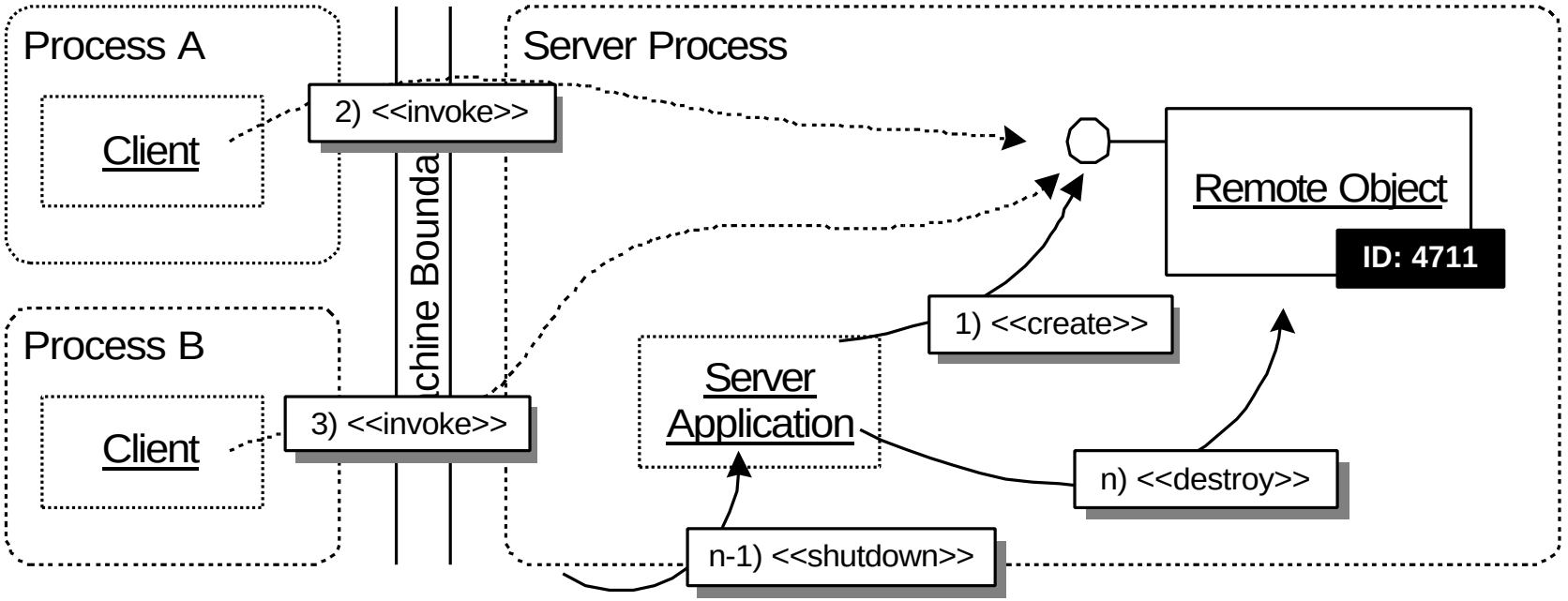
- **Naming** is based on the host name and an object name that is unique with respect to its host.
  - The hostname identifier is dependent on the undelying transport protocol
  - The object name is basically a simple string, possible structured using /
  - Client need to know the server and the protocol to look up an object
- There is no single **Framework Singleton**, however there is a set of helper classes (such as *RemotingConfiguration*, *ChannelServices*) which the client can use to configure and initialize the framework.
- **Server Applications** are .NET programs that initialize and configure the remoting infrastructure, instantiate remote objects and publish them in Naming.

Contents

Remote Object Lifecycle Management

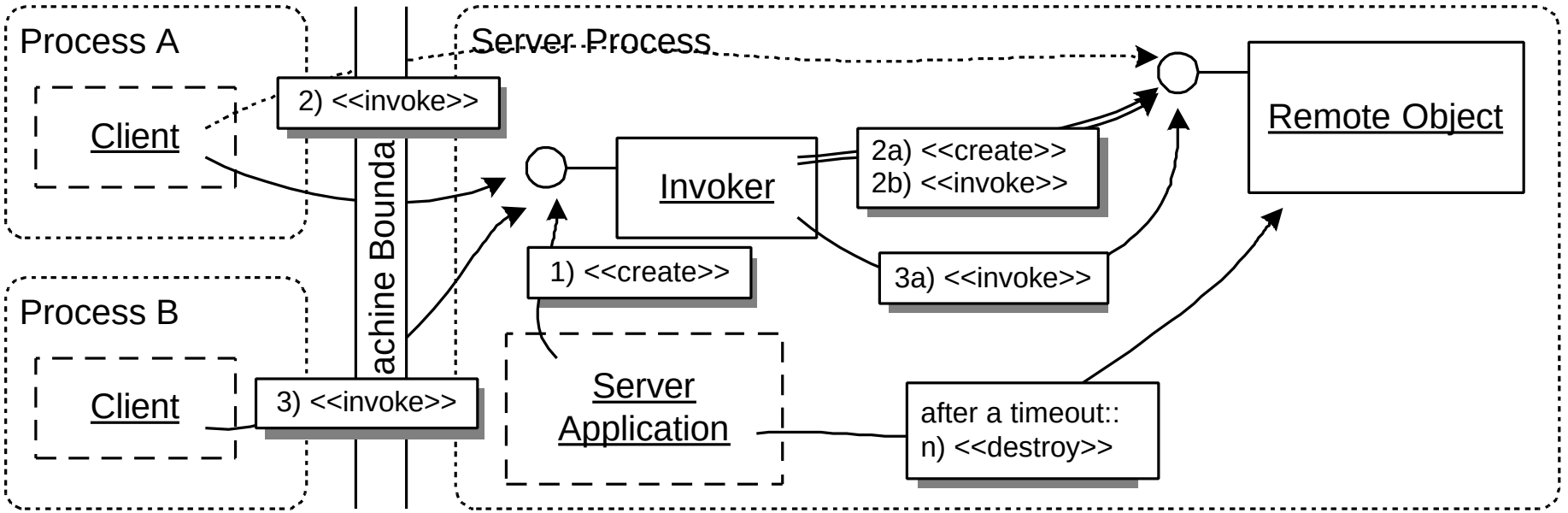


Pre-Configured Instances



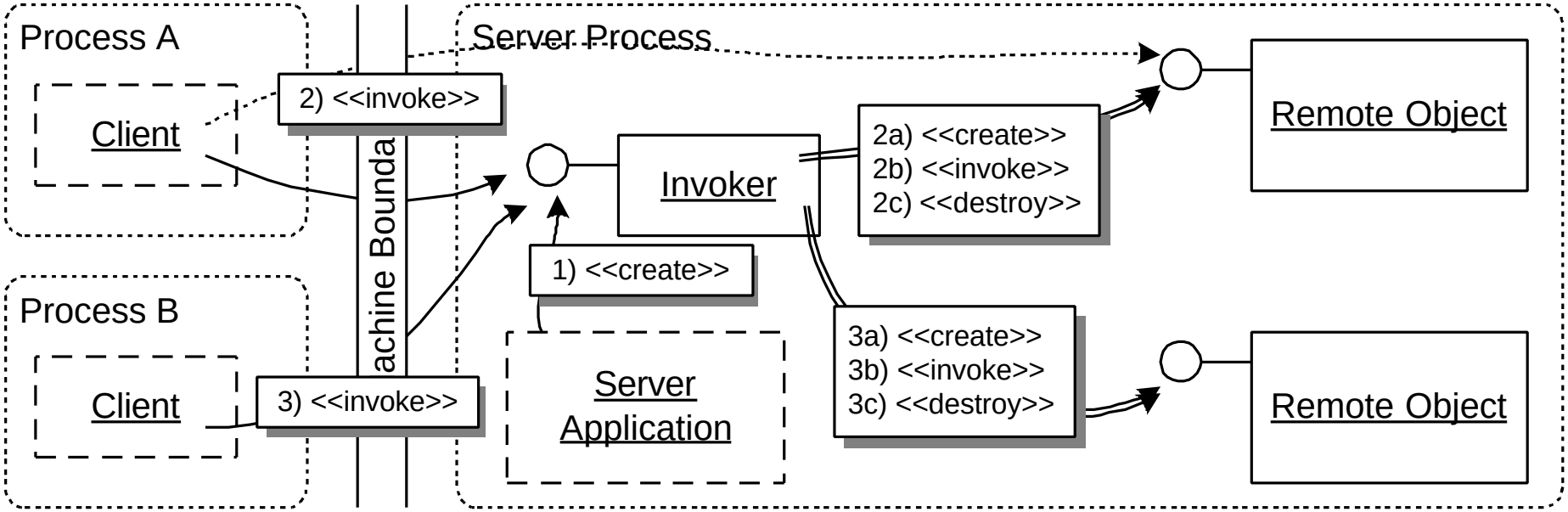
Provide PRE-CONFIGURED INSTANCES that are well-known and stable. These instances are manually created in the SERVER APPLICATION during startup, and they live until the SERVER APPLICATION is shut down. All invocations targeting the service of a specific PRE-CONFIGURED INSTANCE are actually handled by the same object.

On-Demand Activation



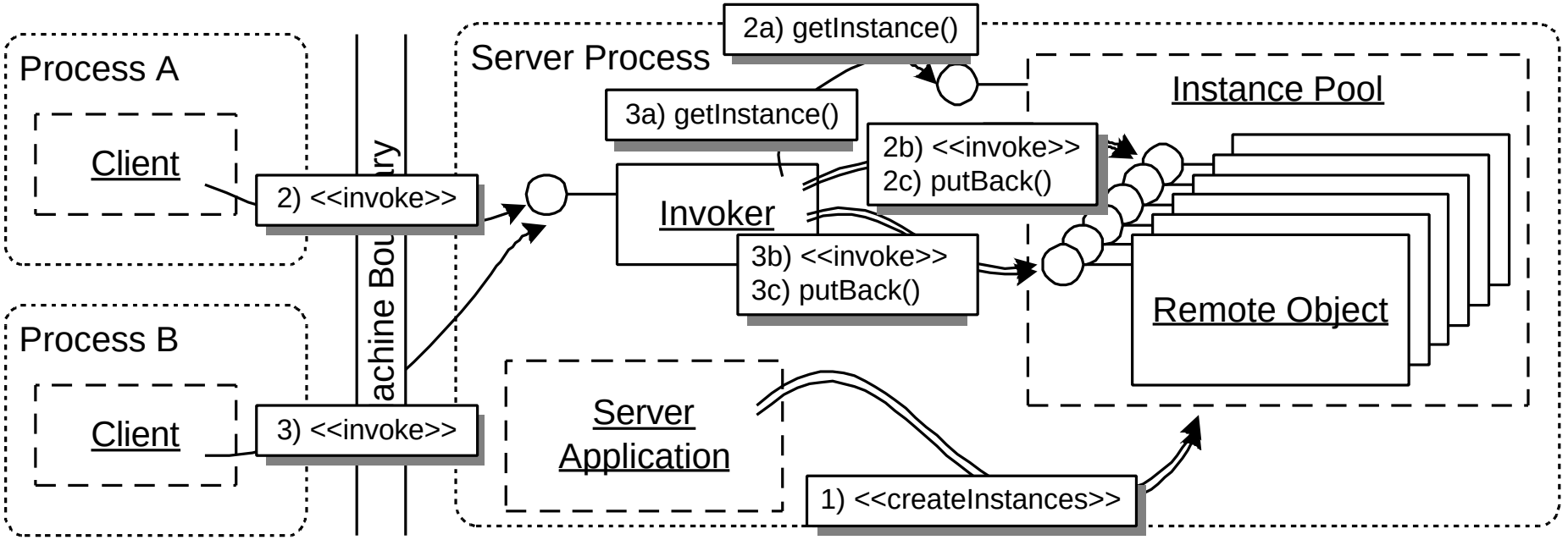
Provide ON-DEMAND ACTIVATION for REMOTE OBJECTS. Logically, the REMOTE OBJECTS are accessible all the time, but in reality they are only activated when a request for the object reaches the SERVER APPLICATION. Make sure that the objects are deactivated after a certain amount of time or based on some other criteria, for example using LEASES.

Single Call Activation



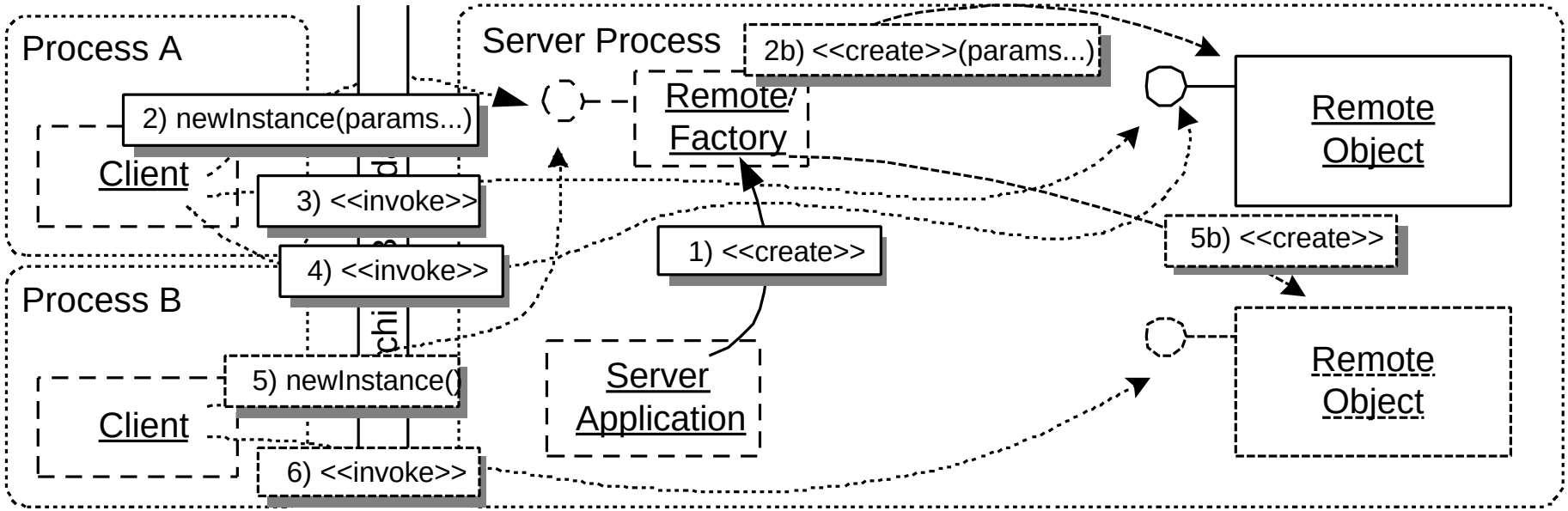
For each incoming invocation, create a new, clean instance of the REMOTE OBJECT. Let it handle the request, perhaps accessing shared resources, and then destroy the instance right away.

Instance Pooling



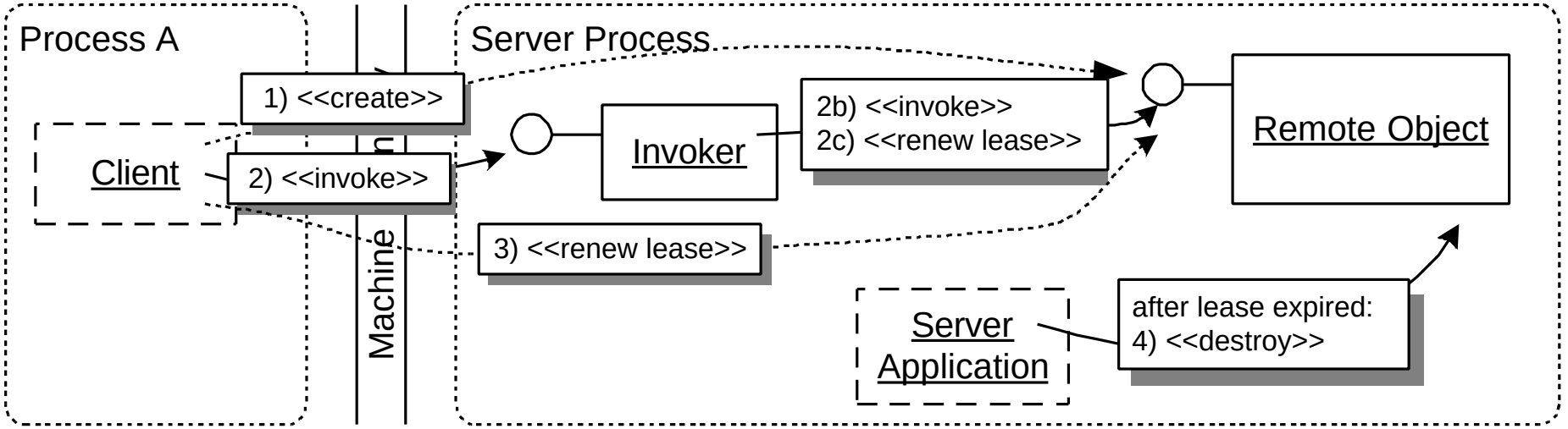
Do not instantiate new REMOTE OBJECTS for each invocation. Instead take an "idle" pre-allocated instance from a pool. After servicing the request, the instance is put back into the pool. The pool is filled with a pre-configured number of identical instances during startup of the SERVER APPLICATION, perhaps growing and shrinking at runtime.

Client Activation



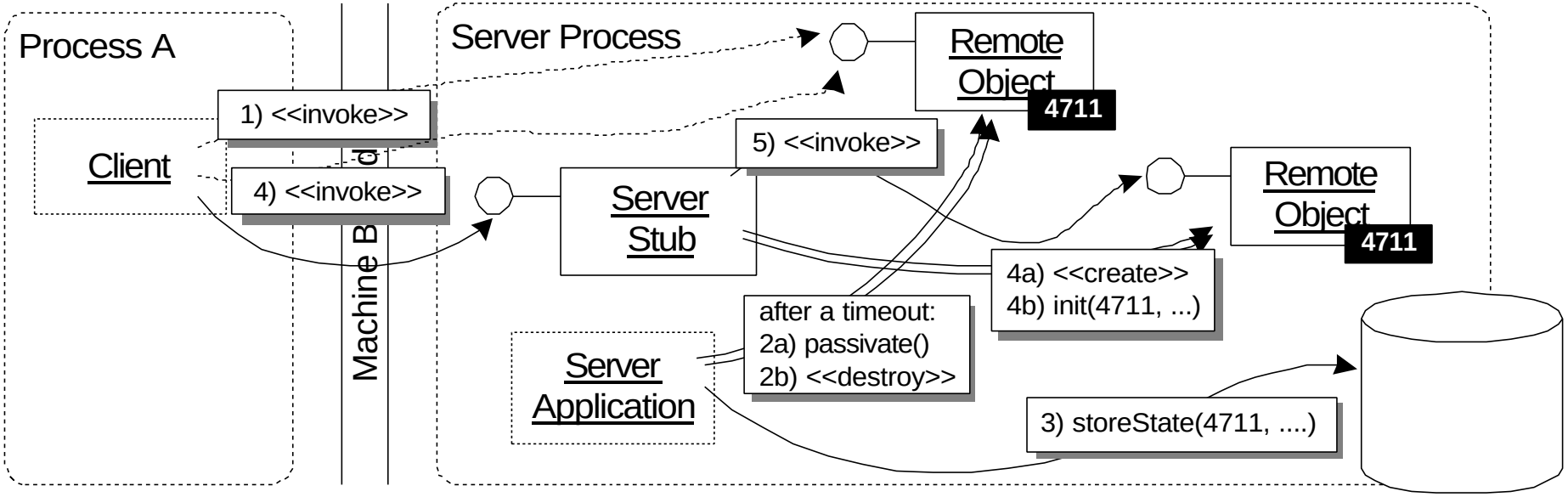
Activate a REMOTE OBJECT on an explicit client request. The instantiated object has state that can be changed by the client. Make sure that the instance is removed when it is no longer needed, for example by using LEASES. To allow the client to create instances explicitly, the client needs remote access to a REMOTE FACTORY object.

Leases



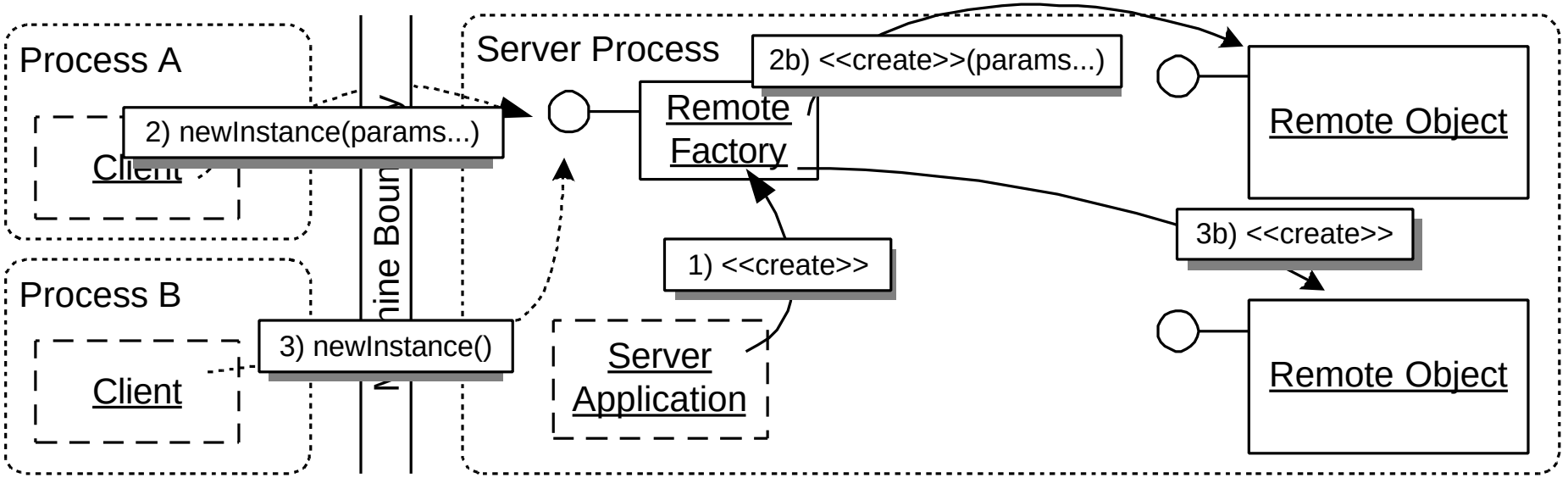
Associate a LEASE with each REMOTE OBJECT. Whenever the LEASE expires, the SERVER APPLICATION is free to discard the corresponding REMOTE OBJECT. The clients needs to actively make sure that the LEASE does not expire as long as it really needs the object; thus, it has to renew the LEASE from time to time.

Passivation



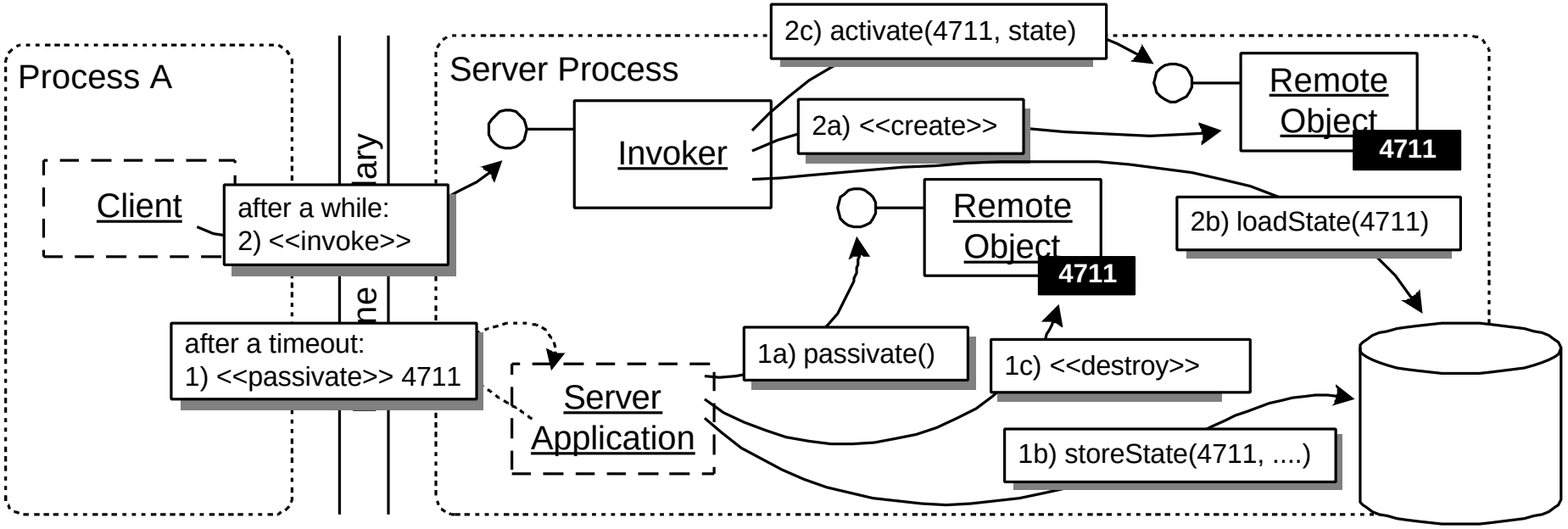
Whenever the SERVER APPLICATION needs to free resources, but cannot destroy enough active instances, passivate those CLIENT-ACTIVATED objects that have not been accessed for a while. Store their state persistently, and when a new request arrives for such an instance, resurrect the instance based on the previously stored persistent state. Use LIFECYCLE OPERATIONS to notify the instances of the respective events.

Remote Factory



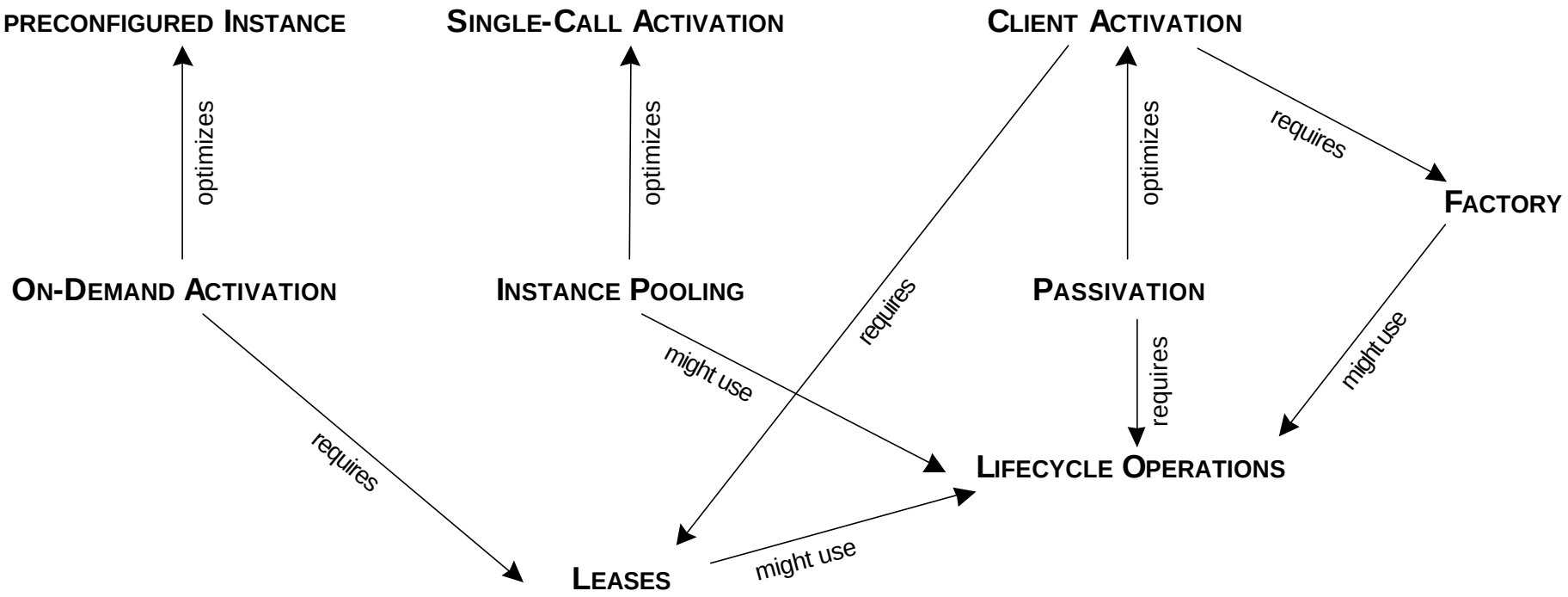
Provide clients with a REMOTE FACTORY object that provides operations to allow clients to "request" the respective type of CLIENT-ACTIVATED objects. Factories are usually well-known REMOTE OBJECTS, thus they are usually PRE-CONFIGURED INSTANCES and registered in NAMING.

Lifecycle Operations



Require that a REMOTE OBJECT provides specific lifecycle operations that allow the SERVER APPLICATION to inform instances about state changes in their lifecycle. How these operations are called, what the object should do in them, and when they are called depends on the lifecycle actually used, and thus primarily on the activation strategy.

Remote Object Lifecycle Management



Lifecycle Management - CORBA

- In CORBA there is a distinction between a logical object (identified by ist Object ID and reference) and the programming language-level object that actually handles requests for a logical object (the so-called servant).
- In CORBA, the lifecycle of instances depends on various policies that can be set for each **Object Group** (represented by a hierarchy of POAs, Portable Object Adapters).
- Depending on the policies, the associations between objects and servants changes over time.

Lifecycle Management - CORBA

- POA policies include:
  - **Lifespan**: Determines how long an instance lives
    - Transient
    - Persistent
  - **Request Processing:** Determines which servants handle requests for an object (
    - USE_ACTIVE_OBJECT_MAP_ONLY
    - USE_DEFAULT_SERVANT
    - USE_SERVANT_MANAGER (kind of user-defined policies)
  - **ObjectID to Servant Association**: How is the relationship between servants and objects managed once a request is handled
    - RETAIN
    - NON_RETAIN
  - **Implicit Activation:** Determines whether servants should be activated automatically
    - IMPLICIT ACTIVATION
    - NO_IMPLICIT ACTIVATION

Lifecycle Management - CORBA

- POA policies continued:
  - **Thread:** Determines how many threads can be „in" a servant at the same time
    - ORB_CTRL_MODEL: several threads can be in a servant, requires thread-safe servants
    - SINGLE_THREAD_MODEL: all requests for objects in that POA are handled sequentially (to control threading, there can be several „parallel" POAs!)
- Note that there can be several POAs in an application, each managing different object (instances).
- The different (server-determined) lifecycles mentioned in the patterns can be emulated by setting the appropriate policies and sometimes implementing a custom servant manager.
  - **Preconfigured Instances**
  - **On-Demand Activation**
  - **Single Call Activation**
  - **Instance Pooling**

Lifecycle Management - CORBA

- **Client Activation** must be programmed manually by providing a server-activated **Factory** object that instantiates other objects on the server and returns their references.
- By default, CORBA does not support any form of distributed reference counting or **leasing**. However, a custom servant manager implementation can be used to achieve this behaviour.
- **Passivation** is also not supported by CORBA by default, but again, a custom servant manager helps.
- If a specific implementation of a custom servant manager requires **Lifecycle Operations**, they have to be provided. By default, none are required.
- Note that CCM provides more support here!!

Lifecycle Management - .NET

- .NET distinguishes server- and client-activated objects.
- For server-activated objects, .NET provides
  - **On-Demand Activation** with singleton semantics (*WellKnownObjectMode.Singleton*) or
  - **Single Call Activation** (*WellKnownObjectMode.SingleCall).*
  - **Preconfigured Instances** are also possible using the *RemotingServices.Marshal()*-operation.
- .NET also provides **Client-Activation**. There are three ways for clients to instantiate a client-activated object on the server:
  - The client's CLR can be configured to allow client to transparently use the *new* operator.
  - Clients can use a special generic factory
  - Developers can hand-craft a server-activated **Factory** that activates other objects on request.

Lifecycle Management - .NET

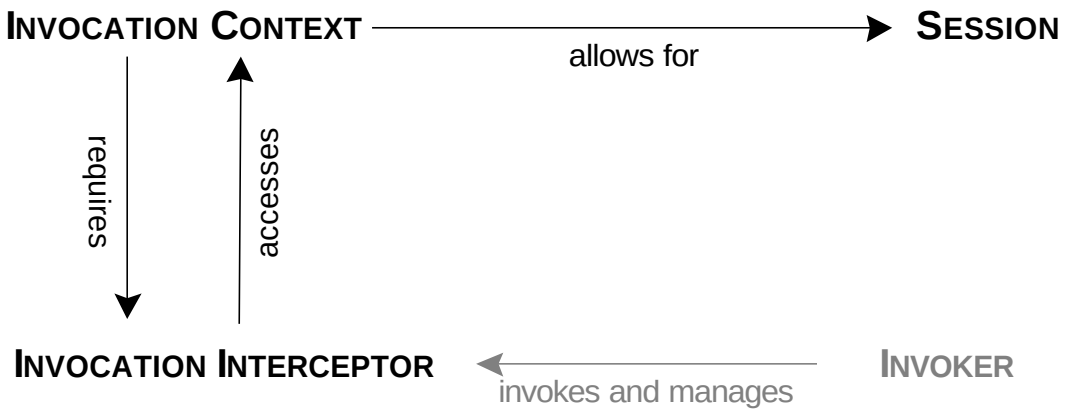
- **Instance Pooling** is not supported on the .NET remoting level.
- NET supports **leases**. Each instance has TTL attribute internally which is checked regularly by the LeaseManager.
- There is an initial TTL and a increment to which the TTL is set whenever an invocation for the instance arrives.
- When the TTL is zero, the *LeaseManager* contacts a (possibly remote) sponsor to ask for a further increase of the lease.
- There is a generic **Remote Factory** for client-activated objects, however, a hand-crafted one has several advantages.

Lifecycle Management - .NET

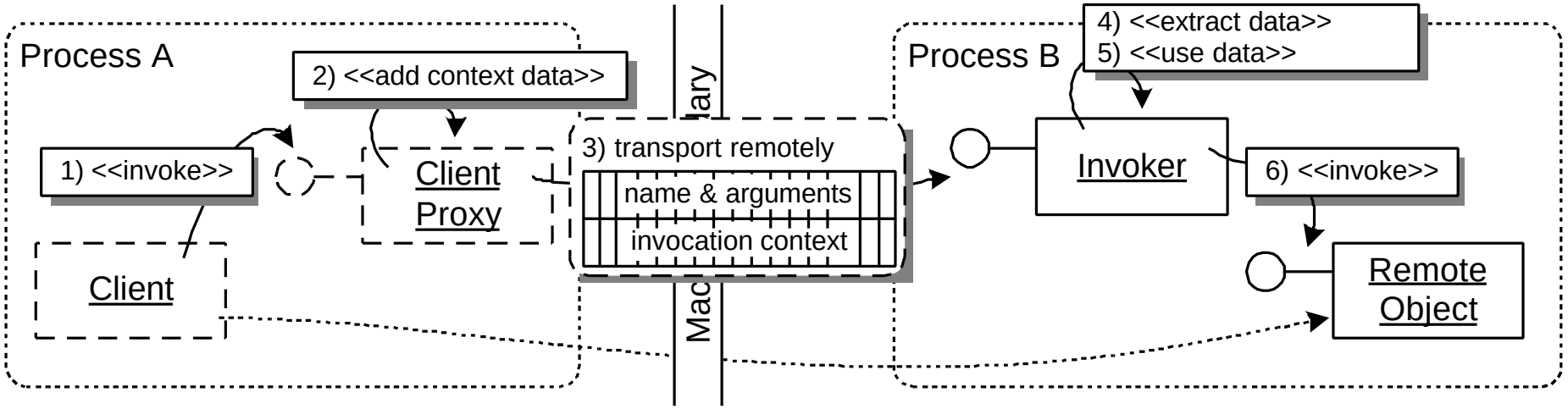
- Because of the very simple lifecycles, there are no **Lifecycle Operations** necessary.
  - In contrast to CORBA it is also not possible to customize this feature (at least, there is no *documented* feature to do so).
- For more advanced lifecycle optimizations (incl. Pooling, e.g.) objects have to be deployed to COM+. But that's another story...

Contents

Additional Services

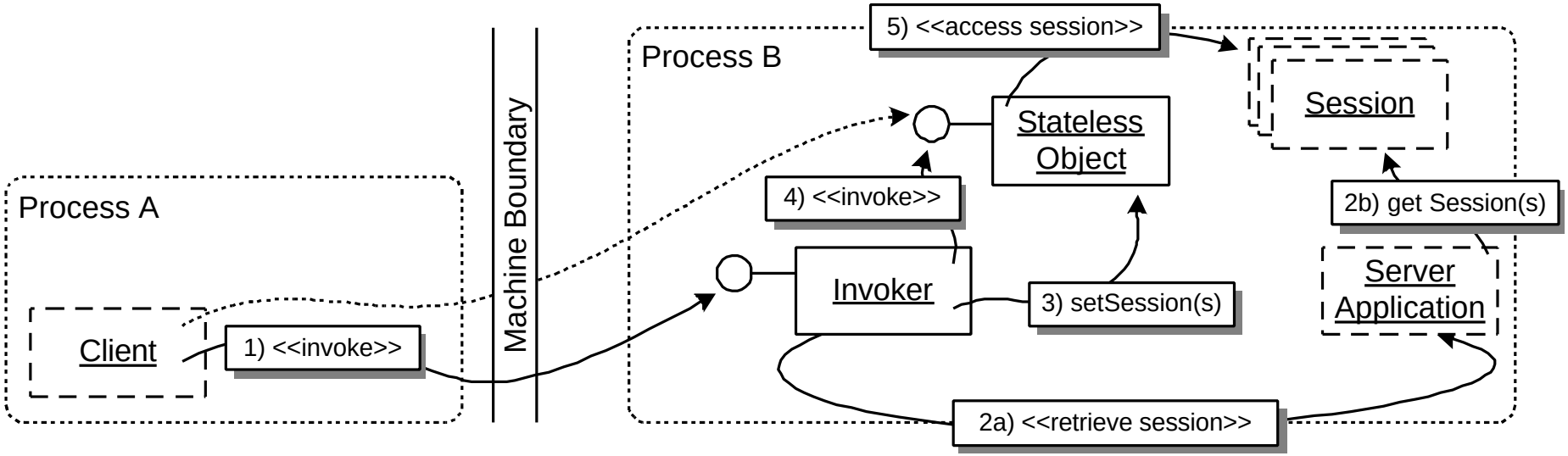


Invocation Context



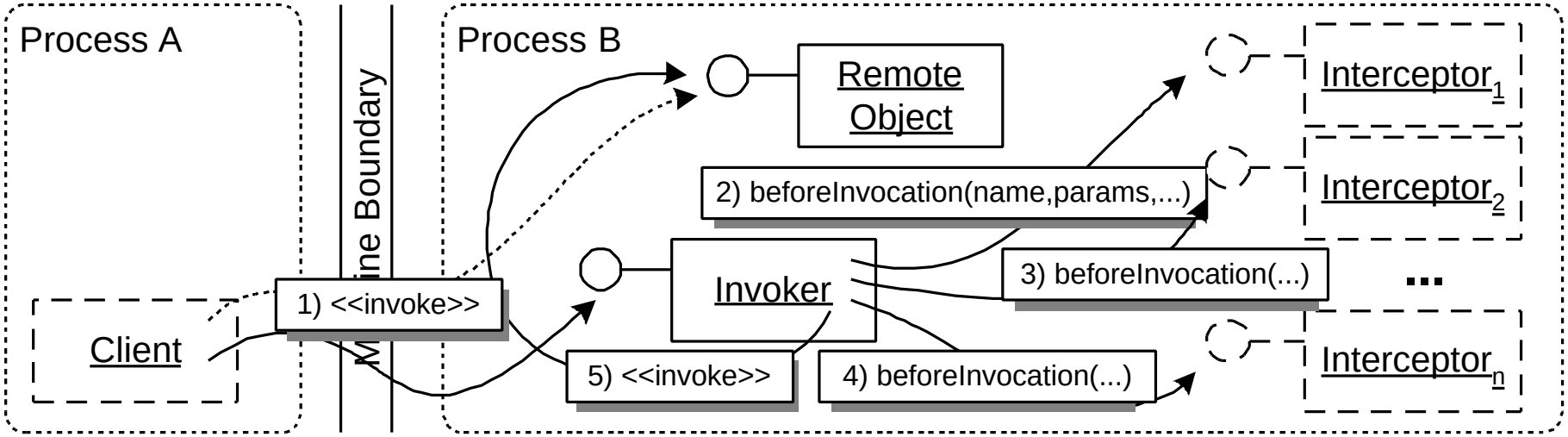
Provide an INVOCATION CONTEXT, an extensible data structure which is passed along each method call. The CLIENT PROXY and the INVOKER create or use this data structure. Depending on the services provided by the SERVER APPLICATION, different contextual data is transported in the invocation context.

Session



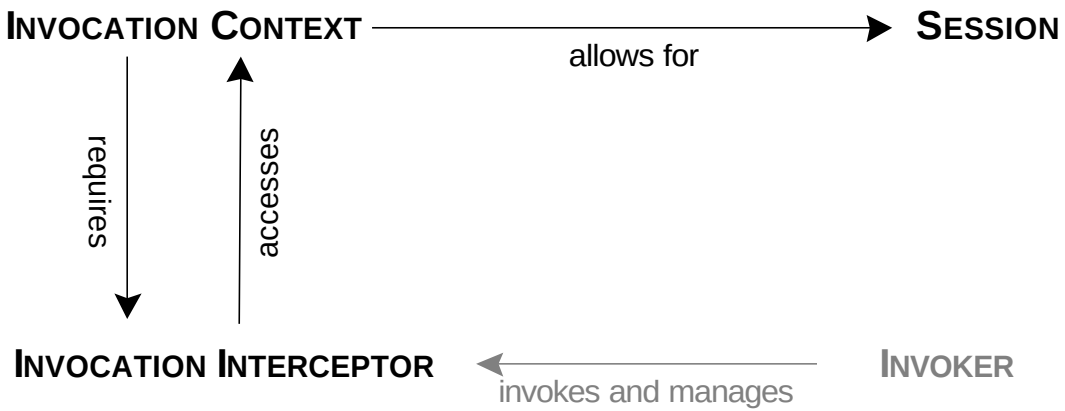
Provide client SESSIONS in the SERVER APPLICATION. Such a SESSION can contain arbitrary state information on behalf of a client. A REMOTE OBJECT should be given access to the session that belongs to the calling client, transparently. The client, as well as the REMOTE OBJECT, should not be involved with SESSION management.

Invocation Interceptors



Provide a hook in the INVOKER to plug in interceptors. An interceptor has operations that are called before and after a method invocation on a REMOTE OBJECT, and it thus has the chance to add whatever functionality it likes. The interceptor is provided with all the necessary data on the invocation just made by the client, such as method, parameters, target instance, and INVOCATION CONTEXT.

Additional Services



Additional Services - CORBA

- CORBA's standard IIOP transport protocol provides for transporting **Invocation Contexts**. For example, they are used by the security and transaction services.
- Inserting data into the context is a rather low-level work that is accomplished by using **Invocation Interceptors**.
- The CORBA standard provides standardized interfaces for interceptors (so-called portable interceptors). They can be used on the server as well as on the client.
- **Sessions** are not supported natively. However, transporting session IDs can be done using the invocation contexts together with interceptors, and a custom servant manager to provide servants with the current session object.

Additional Services - .NET

- .NET Remoting also provides **Invocation Contexts**, they are called *CallContext* here.
- Clients and Object implementations can use static operations on the *CallContext* class to insert/retrieve objects
  - These objects have to be serializable and implement the *ILogicalThreadAffinitive* (marker) interface.
- When remote objects are deployed in the webserver (essentially as a webservice, then) then automatic **Session Management** is available – using static operations on the Session class.
- Note that it is very easy to create your own session management using *CallContexts*.
  - However, there is no automatic assignment of the session to the remote object – the implemetation has to do a manual lookup.

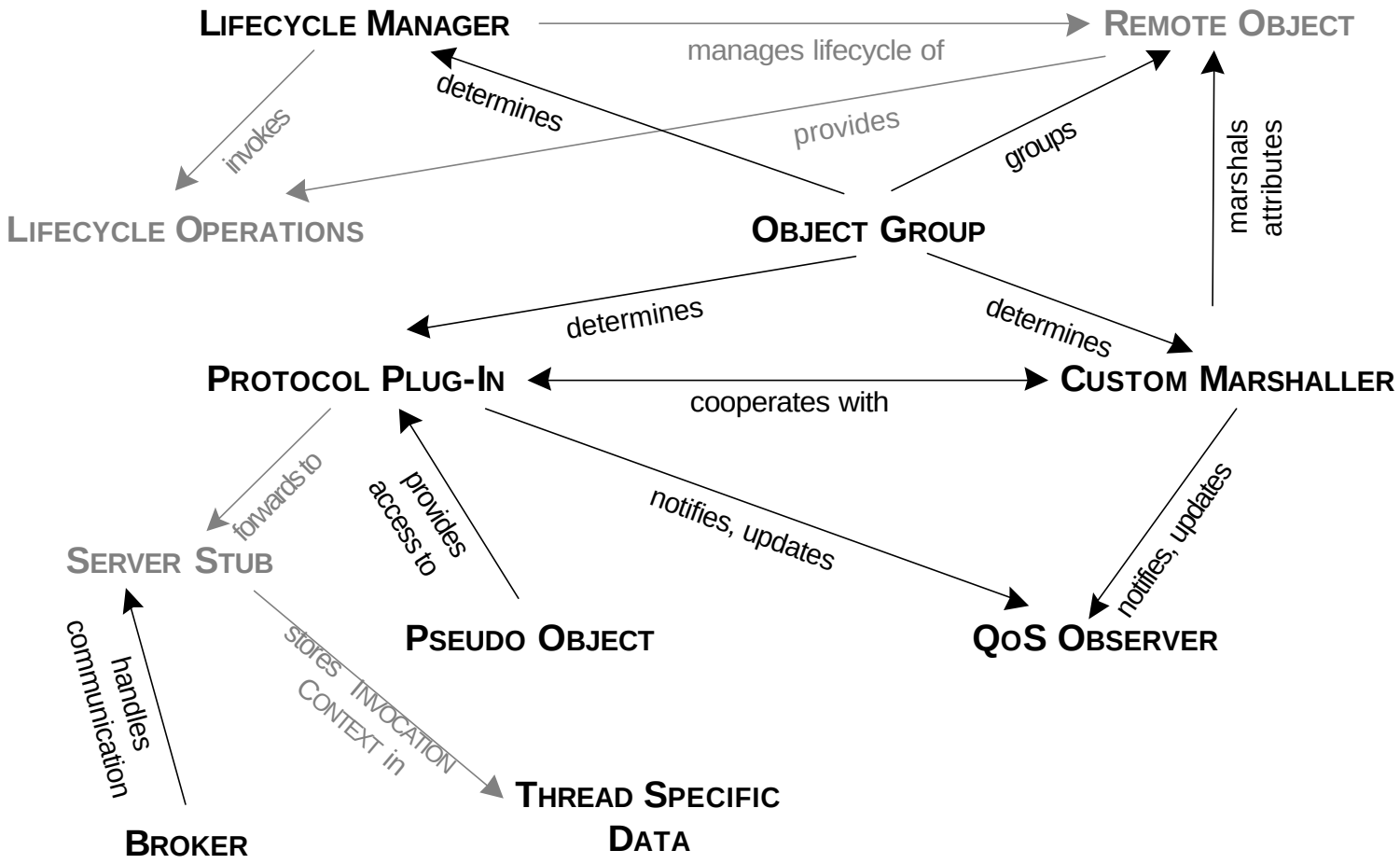
Additional Services - .NET

- The concept of **Invocation Interceptors** is available in .NET, although they are not technically interceptors.
  - Invocation messages in .NET are passed through a stack of so-called *sinks* at the server and at the client (this resembles TCP/IPs layered architecture).
  - Each sink has a specific task to fulfil.
  - Using specific factories, it is possible to insert custom sinks at different points in the server's or client's invocation and message processing chain.
- Using custom sinks and the *CallContext* class together, provides a simple way to transparently transport and provide additional data with method calls, such as principals, keys or transaction IDs.

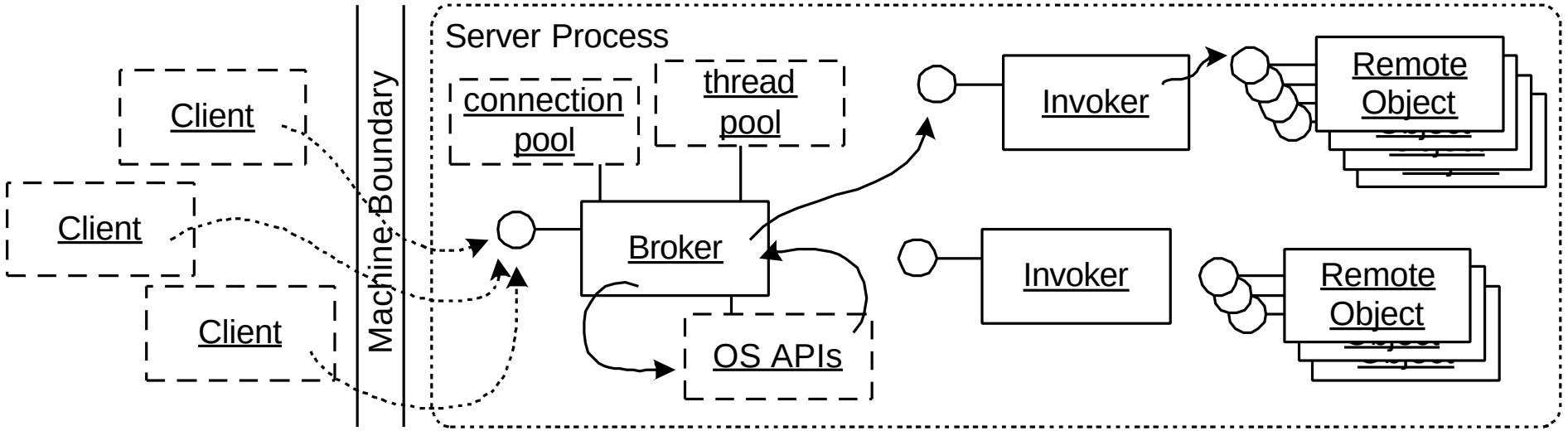
Contents

- Introduction
- Patterns and Technology Projections
  - Basic Remoting Patterns
    - CORBA Projection
    - .NET Remoting Projection
  - Lifecycle Management
    - CORBA Projection
    - .NET Remoting Projection
  - Providing Additional Services
    - CORBA Projection
    - .NET Remoting Projection
  - Building High-Performance Servers
    - CORBA Projection
    - .NET Remoting Projection
  - Asynchronous Operations
    - CORBA Projection
    - .NET Remoting Projection

High-Performance Servers

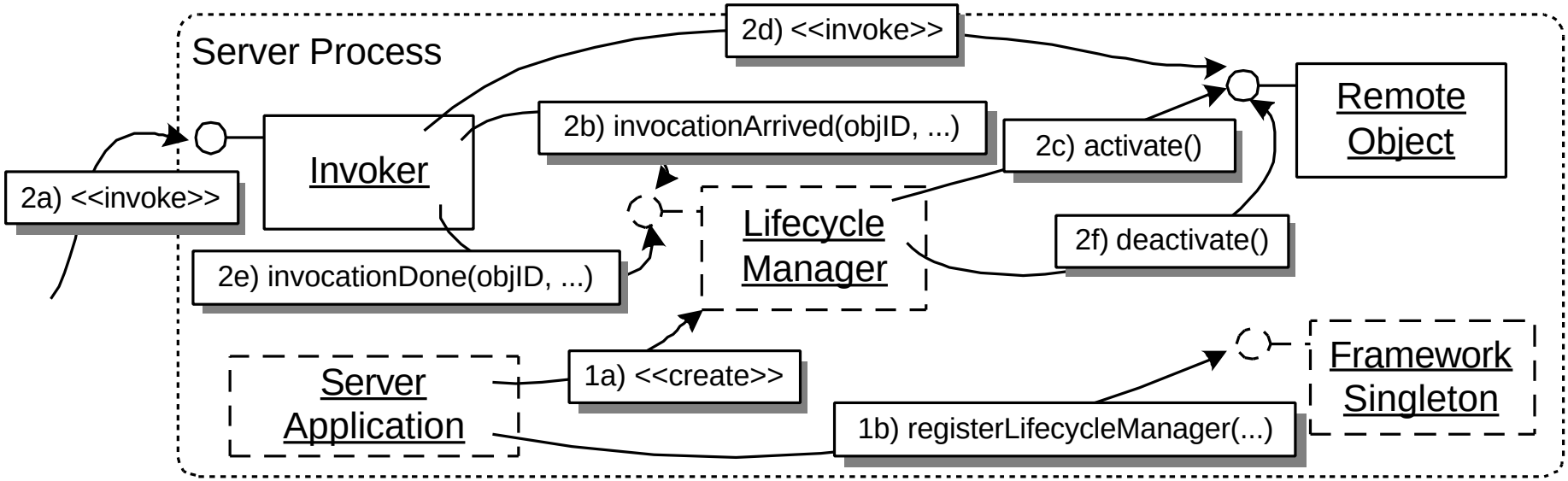


Broker



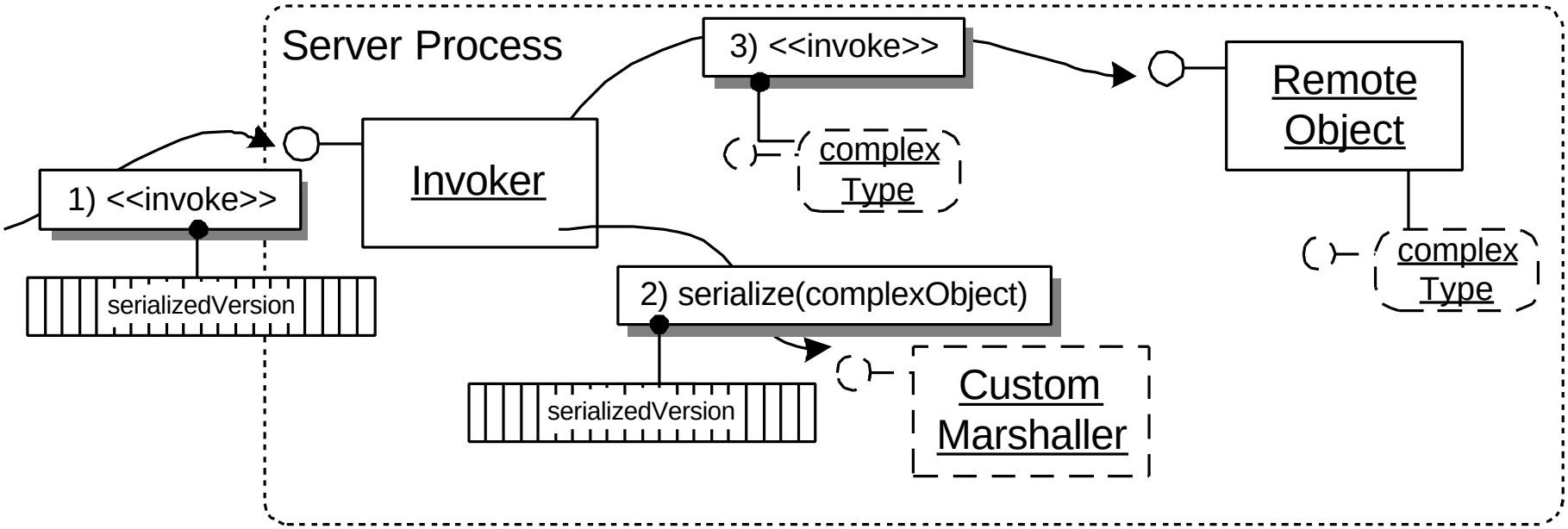
Provide a central network communication component, called a BROKER. A BROKER handles connections, threading and event management, typically using efficient, optimized mechanisms of the underlying operating system. Once a request is received from the network, it is forwarded to the respective INVOKER for further processing.

Lifecycle Manager



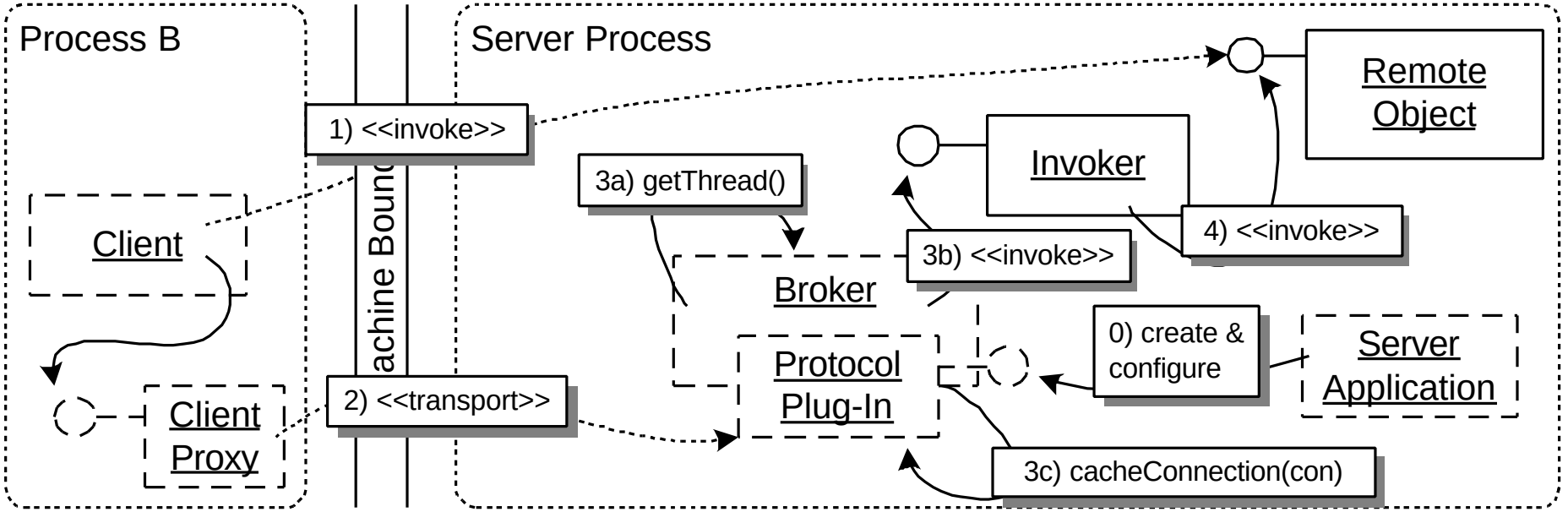
Provide an API in your FRAMEWORK SINGLETON for developers to plug-in custom developed LIFECYCLE MANAGERS. The instantiation and eviction patterns, described earlier, can be provided in the form of default implementations for the LIFECYCLE MANAGER. Make sure the LIFECYCLE MANAGER is called at the appropriate times by the BROKER or the SERVER STUB. That is, let it be based on (extensible) event raised as LIFECYCLE OPERATIONS.

Custom Marshaller



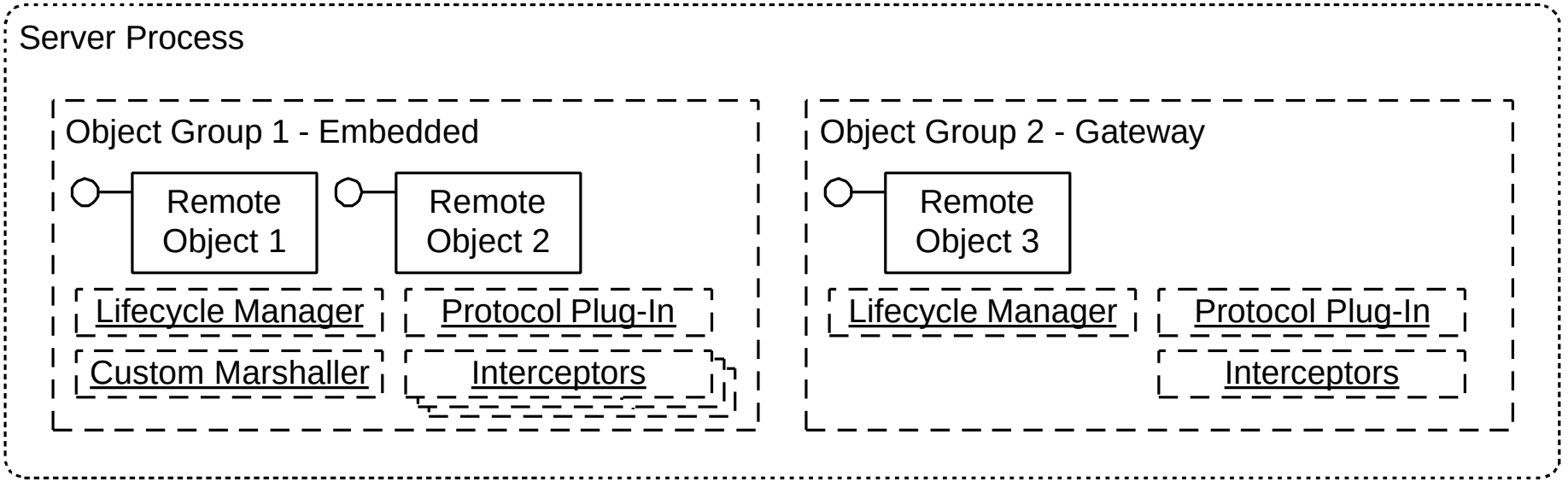
Make the marshaller in your SERVER APPLICATION and its respective conuterpart in the client extensible and/or exchangeable. Provide well-defined interfaces to allow developers to implement their own CUSTOM MASHALLERS. Make sure the INVOKER or BROKER calls the respective marshaller at the appropriate times.

Protocol Plug-In



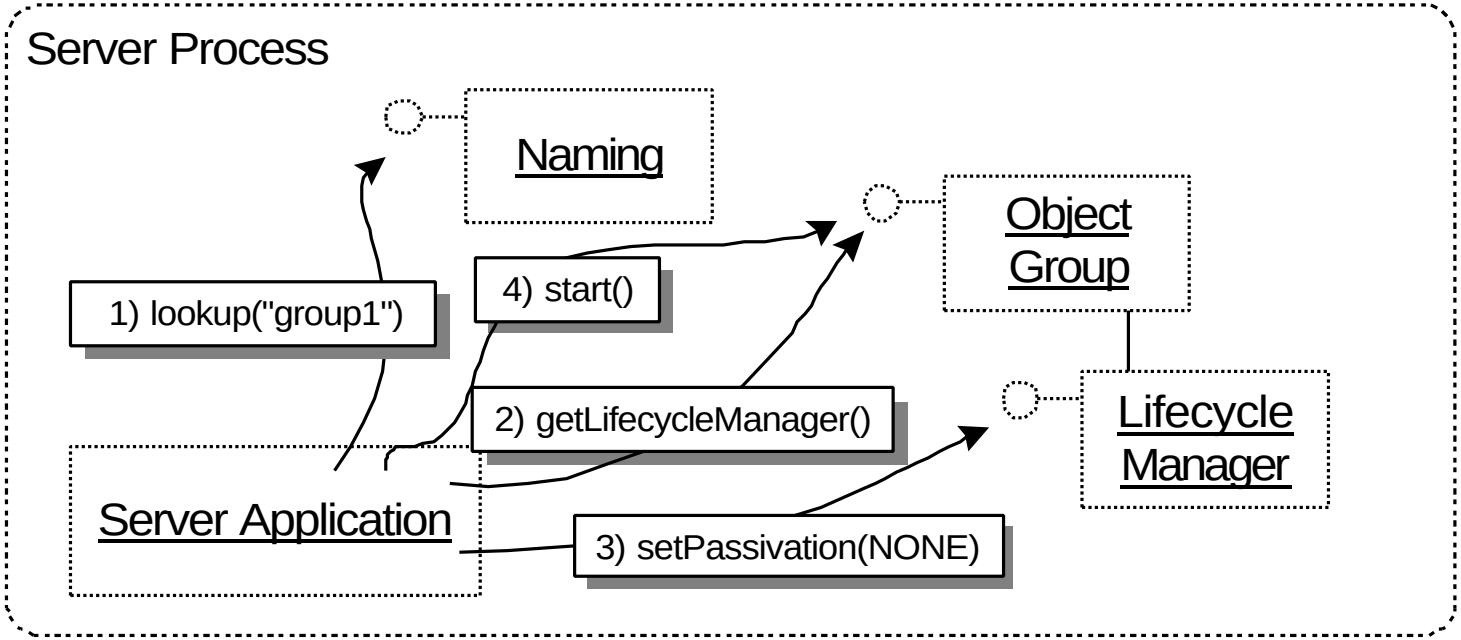
Provide an API to plug in different protocol adaptors into the BROKER. They are responsible for handling the low-level network issues in cooperation with the operating system. Expose the PROTOCOL PLUG-INS to the application developer in order to allow for tight control and specific optimizing configurations.

Object Group



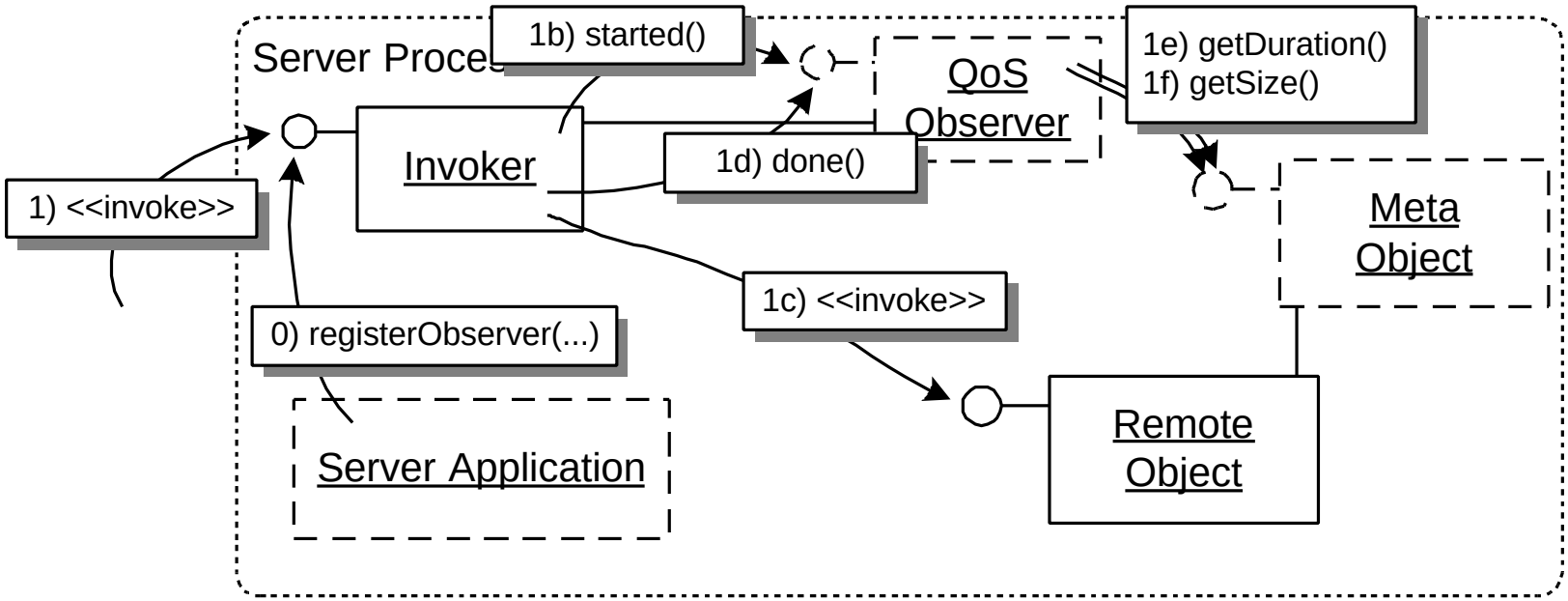
Provide OBJECT GROUPS which group all REMOTE OBJECTS that have a set of common properties, say, the same QoS proerties. A SERVER APPLICATION can have several OBJECT GROUPS at the same time, and, as a variant, each object may be part of different OBJECT GROUPS. All non-individual configuration, such as providing LIFECYCLE MANAGER, PROTOCOL PLUG-IN, etc., is done on OBJECT GROUPS level.

Pseudo Object



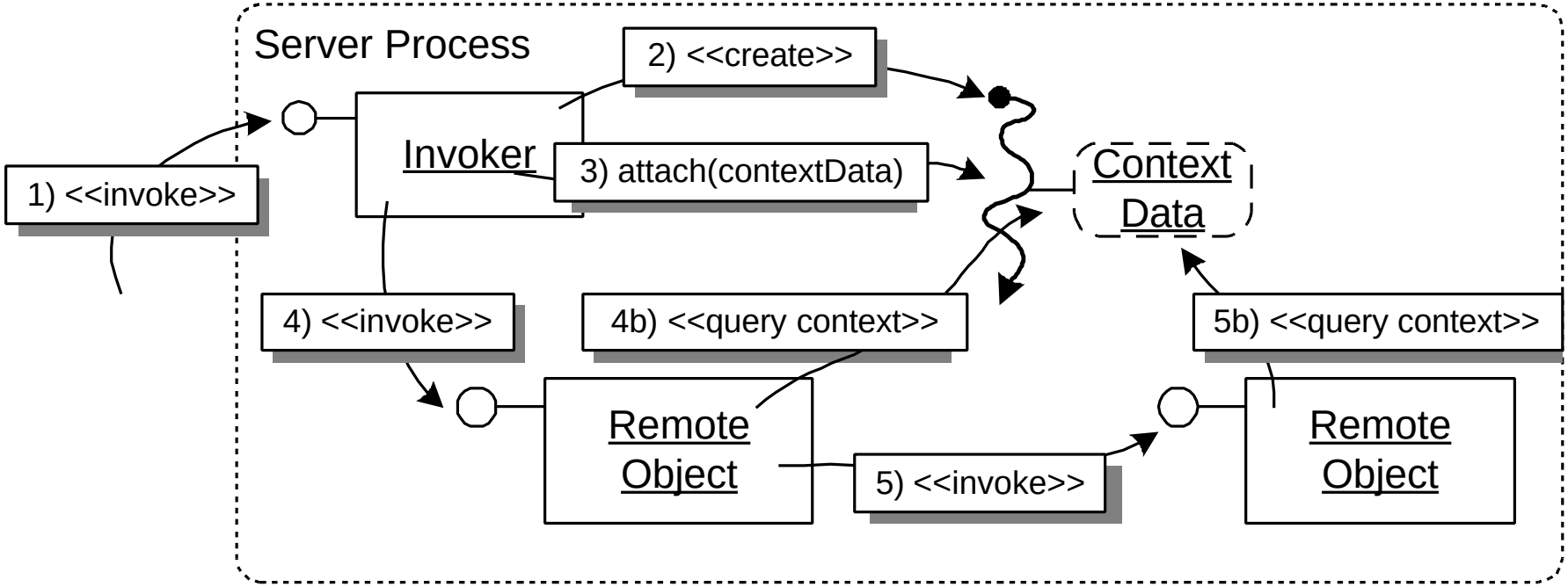
Provide PSEUDO OBJECTS that have a behavior and interface of REMOTE OBJECTS provided by the SERVER APPLICATION. Make them available through NAMING, just as any other REMOTE OBJECT. By not specifying a GLOBAL OBJECT REFERENCE for them, you can make sure that they are not accessible externally.

QoS Observer



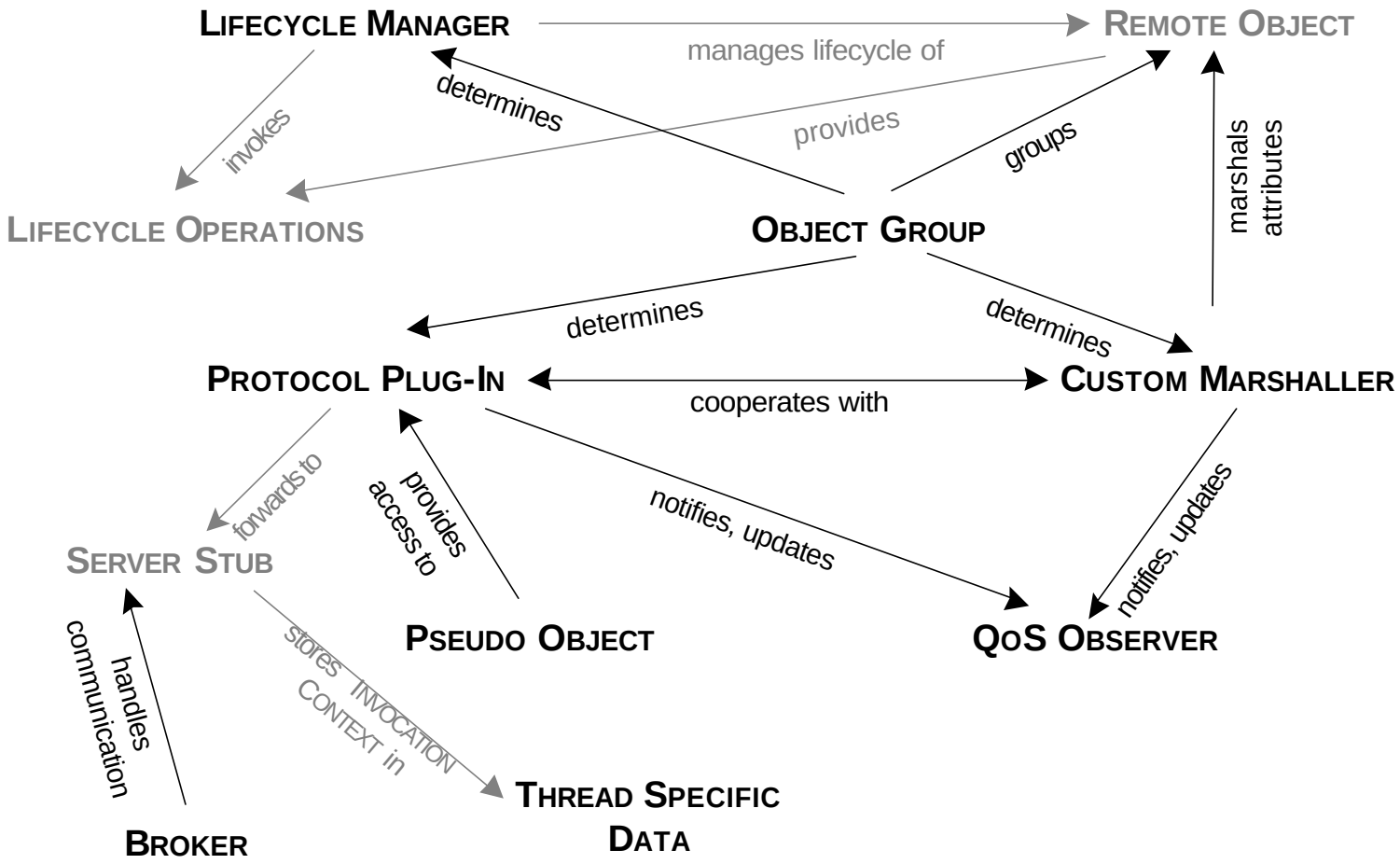
Provide hooks for application developers to be selectively notified of important events that are related to quality of service characteristics. In addition, use meta objects (instances of PSEUDO OBJECTS) associated with the REMOTE OBJECTS, INVOKER, and BROKER, etc. that provide information about timing and bandwidth.

Thread-Specific Data



Associate context data like INVOCATION CONTEXT with the thread in which the request is served. This allows REMOTE OBJECTS, their INVOKERS, and any other entity in the call chain to access the context data, e.g. in the INVOCATION CONTEXT, without any additional overhead or any in-between code.

High-Performance Servers



High-Performance Servers - CORBA

- The **Broker** is, as the name CORBA implies, the central backbone of CORBA applications.
- Implementations, and thus, performance, scalability and efficiency vary widely among CORBA implementations.
- There are several high-performance ORBs for different scenarios such as embedded, enterprise or desktop.
- Typical implementations for Brokers are based on the Reactor, Proactor, Half-Sync/Half-Async und Leader-Followers patterns (see POSA2).
- RT CORBA implementation lets you customize Broker behavior with regard to priorities, threads, connections and memory usage.
- **Thread-Local Data** is used in ORB implementations to enhance local (non-remote) performance and to associate out-of-band data with servants.

High-Performance Servers - CORBA

- The **Lifecycle Manager** is basically the POA in CORBA.
  - POAs can be hierarchically arranged.
  - The POA can be configured with a large set of policies to customized its behavior
  - A POA manages a limited set of objects; naturally those that have similar policies.
  - It is possible to configure a POA with a custom servant manager. This allows customized management of objects‘ lifecycles.
- **Custom Marshallers** cannot be easily implemented in CORBA. However,
  - Interceptors can be used to modify messages
  - CORBA‘s underlying transport protocol can be exchanged, including the marshalling parts.

High-Performance Servers - CORBA

- **Protocol Plug-Ins** are not directly supported (i.e. the interfaces for plugging in different protocols are not standardized).
  - However, it is possible to exchange the communication protocol used by CORBA (GIOP, IIOP, native).
  - Some protocol plug-ins (especially in RT CORBA) allow low-level control and customization of connection management
- Providing different **Object Groups** with different lifecycle parameters is provided in CORBA by using several (differently configured) POAs in a single application.
- CORBA also provides (some) **Pseudo-Objects**. They are called locality-constraint objects.
- Standard-CORBA does not provide **QoS-Observers**, however, there are RTCORBA implementations that provide this feature (such as QuO).

High-Performance Servers - .NET

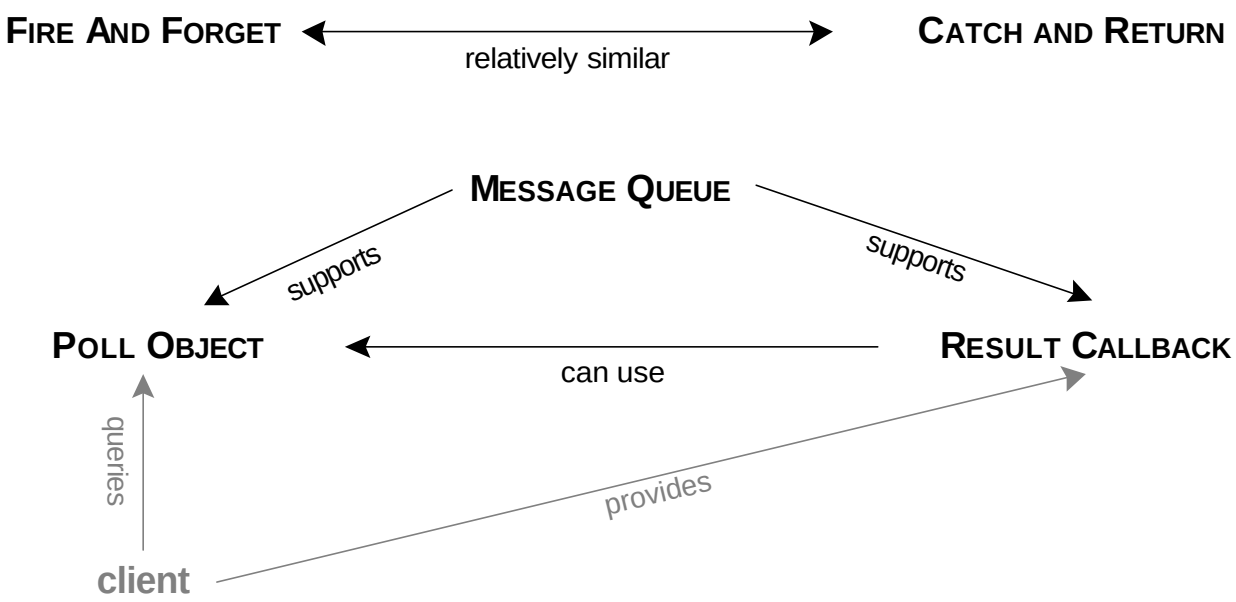
- In .NET the **Broker** is basically integrated into the CLR – the CLR is the **Broker**. There are not many ways how its behavior can be controlled or customized.
- In contrast to CORBA, there is only one implementation (the CLR) and not many different implementations from different vendors.
- Because lifecycles are so trivial, there is no concept of a **lifecycle manager** (except for the lease concept).
- **Marshalling** can be customized by replacing the formatter associated with a specific channel. There are two default formatters:
  - Binary formatter
  - XML formatter
- Developers can create their own formatters.

High-Performance Servers - .NET

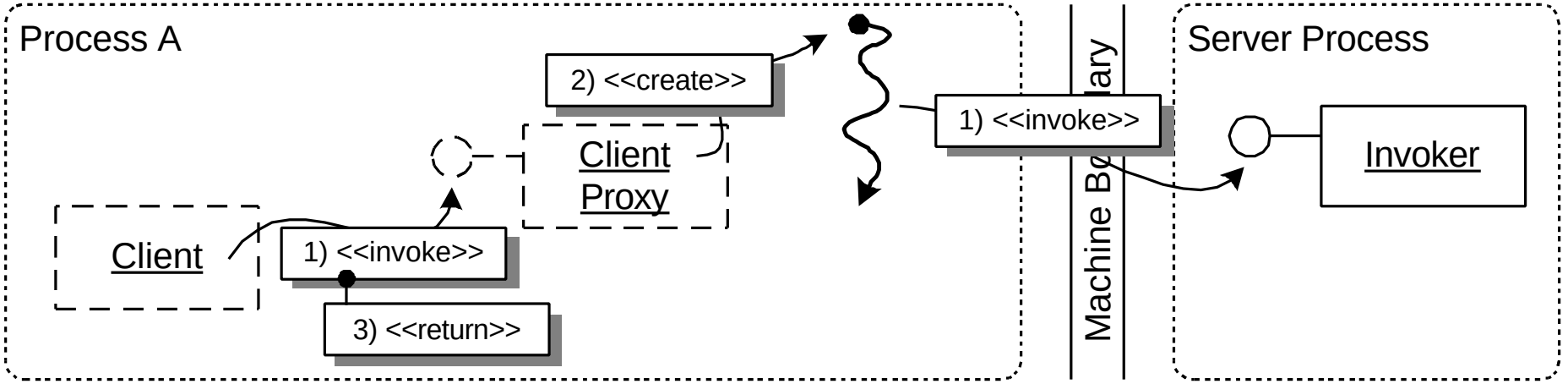
- Also, the communication protocol can be exchanged, true **Protocol Plug-Ins** are available. Again, there are two defaults:
  - TCP/IP, and
  - HTTP
- Developers can create their own protocols.
- The concept of **Object Groups** is not available. The same is true for **Pseudo Objects.**
- **QoS Observers** are not supported, but custom sinks can be used in some cases.

Contents

Asynchronous APIs

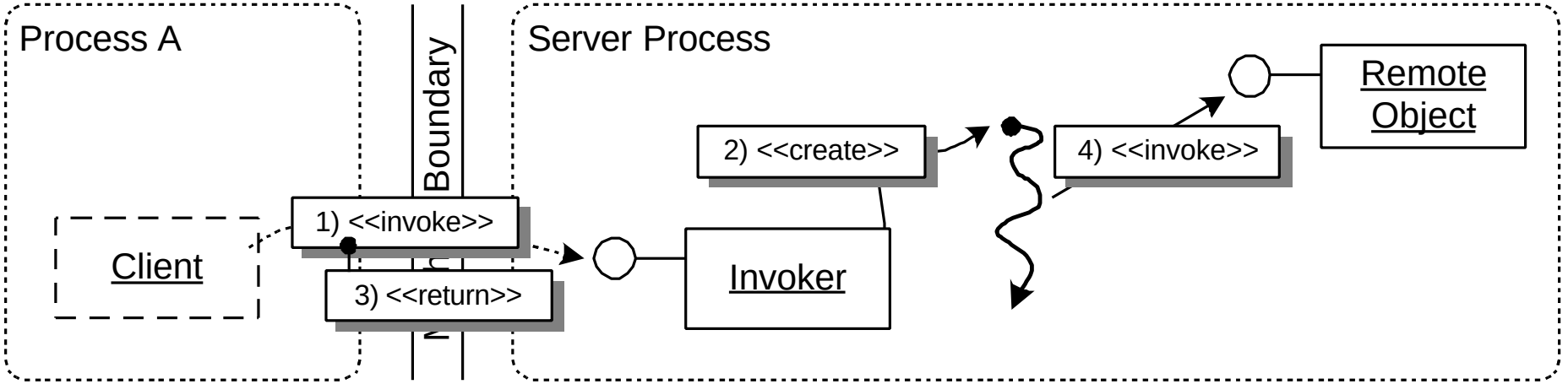


Fire and Forget



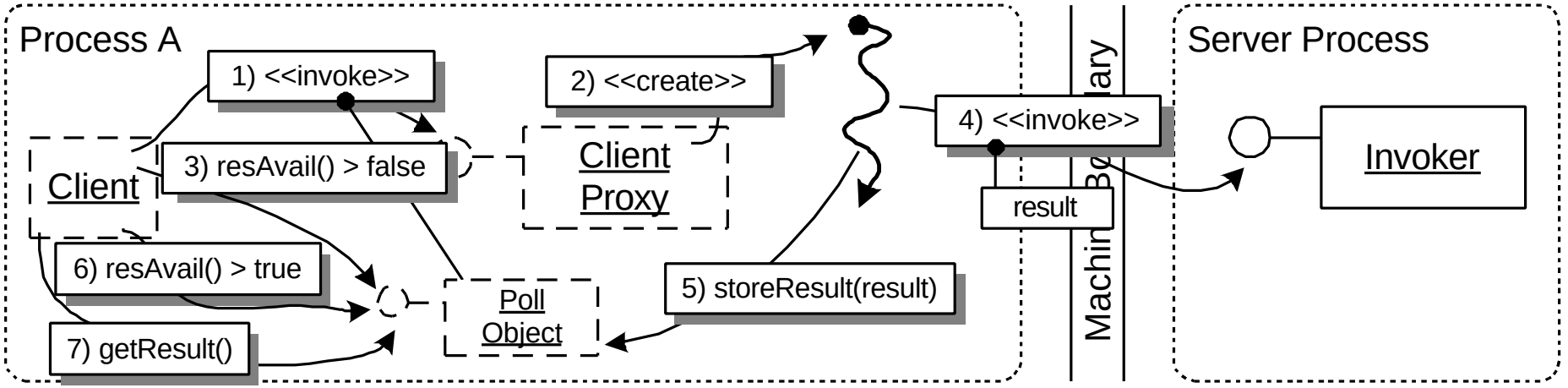
Provide FIRE AND FORGET operations. When called, the CLIENT PROXY forwards the message over the network in a new thread, returning control to the caller immediately. This behaviour is transparent for the client because only the (usually generated) CLIENT PROXY needs to change its implementation.

Catch and Return



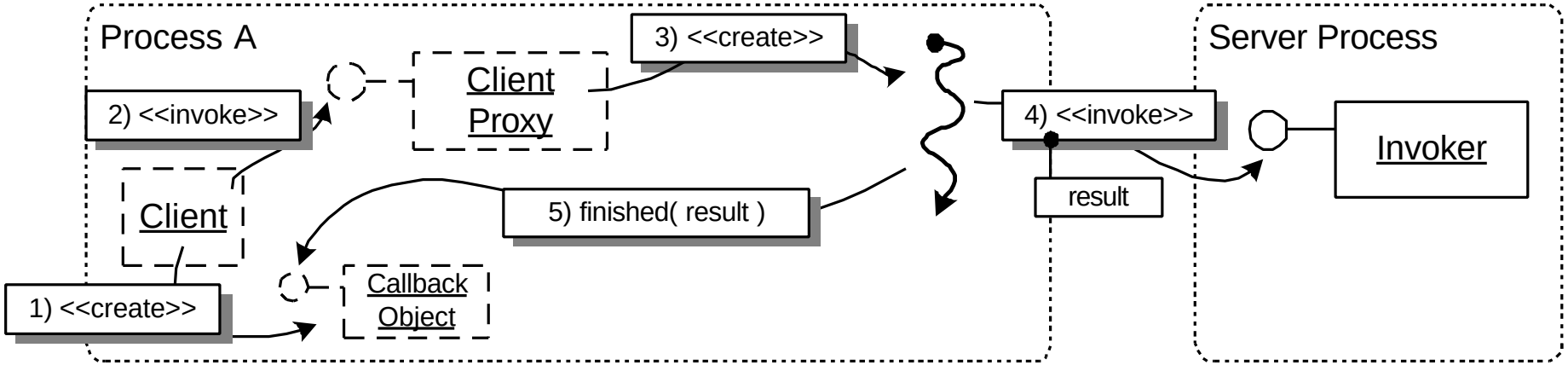
Provide CATCH AND RETURN semantics for remote operations. The client calls an operation. It is then synchronously dispatched to the SERVER APPLICATION which creates a new thread to invoke the operation on the REMOTE OBJECT. After creating the new thread, the INVOKER returns control to the remote client directly.

Poll Object



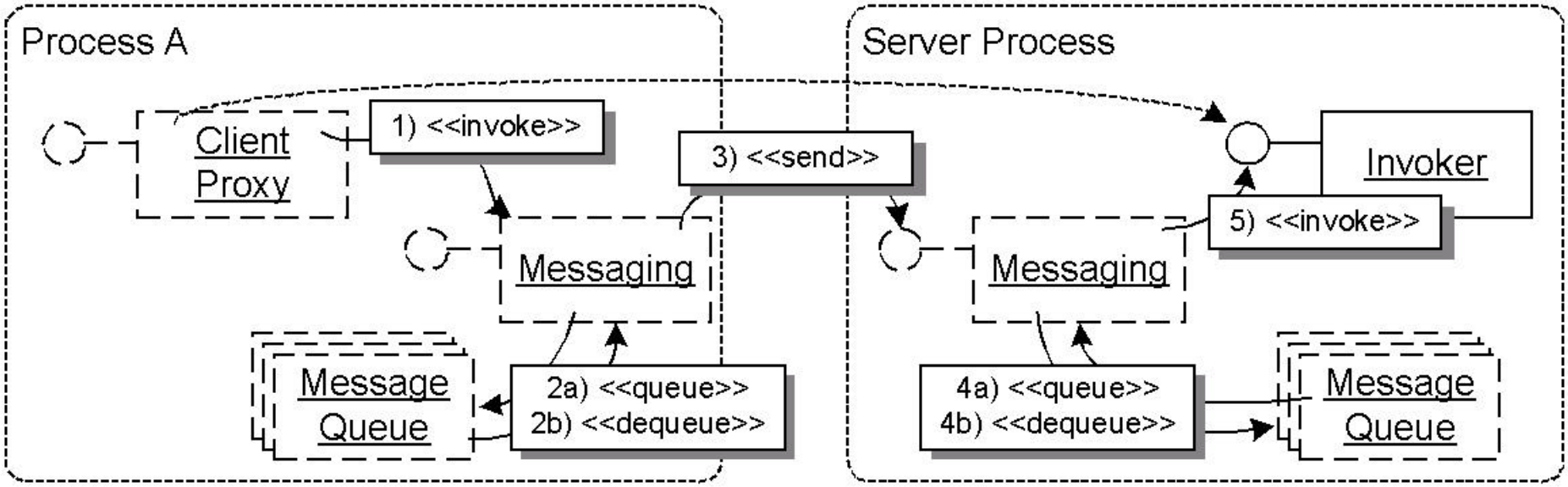
Provide a way to call the remote operation that returns a POLL OBJECT immediately. The client can use this POLL OBJECT to query the result, or block waiting until the result becomes available. It is called POLL OBJECT because it offers a "result available" method for the client, so that the client can poll the object from time to time to see whether the result is already there. In the meantime, the client can go on with processing.

Result Callback



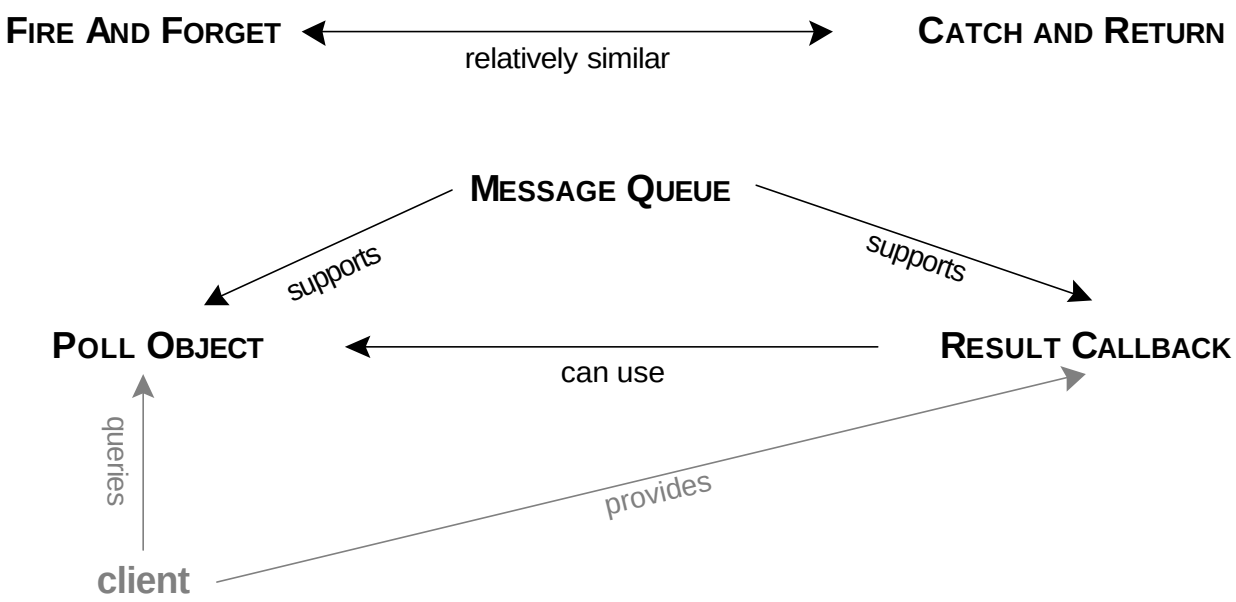
Provide a callback-based interface for remote invocations. Upon invocation, the client passes a callback object to the remoting framework and the invocation returns directly. Once the result is available, the framework calls an predefined operation on the callback object, passing it the result of the invocation, and the callback object knows how to handle the result. Note that the callback object and the client may be identical instances; that is, the callback-based interface is added then to the client itself.

Message Queue



Build a MESSAGE QUEUE on client and server side. A messaging system handles the MESSAGE QUEUES. It allows for adding messages to the queue and it tries to sent the messages periodically, so that temporal failures of network or server process can be tolerated. Each remote invocation, sent via the MESSAGE QUEUE, is packaged as a message; that is, header information for the messaging system are added (and removed on reception).

Asynchronous APIs



Asynchronous APIs - CORBA

- In the beginning, CORBA was basically only synchronous; over time, asynchronous features have been added.
- **Fire and Forget** semantics can be achieved in CORBA by declaring an operation in an IDL interface to be *oneway*.
  - The client ORB processes the invocation asynchronously.
- **Catch and Return** semantics are not supported by CORBA natively. The functionality has to be implemented by the object implementation or by modifying the generated skeleton code.

Asynchronous APIs - CORBA

- With CORBA 3, the AMI (Asynchronous Messaging) specification allows for more advanced asynchronous features:
  - **Poll Objects** and **Result Callbacks** are supported.
  - Message transport can be over different messaging middlewares – routers, persistent queues, etc. included.
  - Asynchronucity is purely client-centric: servers and object implementations do not need to care and technically still support only synchronous calls.
  - Technically, the client developer uses the IDL interfaces and generates a different set of interfaces and stubs from them which provide asynchronoucity.
    - The generation process is based primarily on naming conventions and a set of interfaces
- As mentioned, AMI messages can be transported over an arbitrary set of **Message Queues** and routers with different quality of service guarantees.

Asynchronous APIs - .NET

- .NET Remoting provides a rather powerful and convenient API for asynchronous invocations.
- However, it's not easily possible to change the underlying transport – it's basically still the normal RPC.
  - Only in DCOM/COM+ is it possible to provide real messaging using COM+ Queued Components and MSMQ.
- As in CORBA, an operation can be declared oneway to sport **Fire and Forget** semantics.
- **Catch and Return** has to be implemented manually, too.
- All other kinds of asynchronous operations in .NET are based on delegates, a kind of strongly typed, OO function pointer.

Asynchronous APIs - .NET

- **Poll Objects** are supported in .NET:
  - The invocation on the delegate returns a future object.
  - You can poll on this future object, or wait (block) until the result is available.
- Alternatively, you can use **Result Callbacks.**
  - Upon invocation on the delegate, you can register a callback which is called by the runtime when the result becomes available.
- Again, the concept of **Message Queues** is not there in .NET Remoting.

Contents

The End.

- **Thanks**
  - Questions?
  - Comments?
  - Criticism?

völter.de
ingenieurbüro für softwaretechnologie

Markus Völter
voelter@acm.org
www.voelter.de