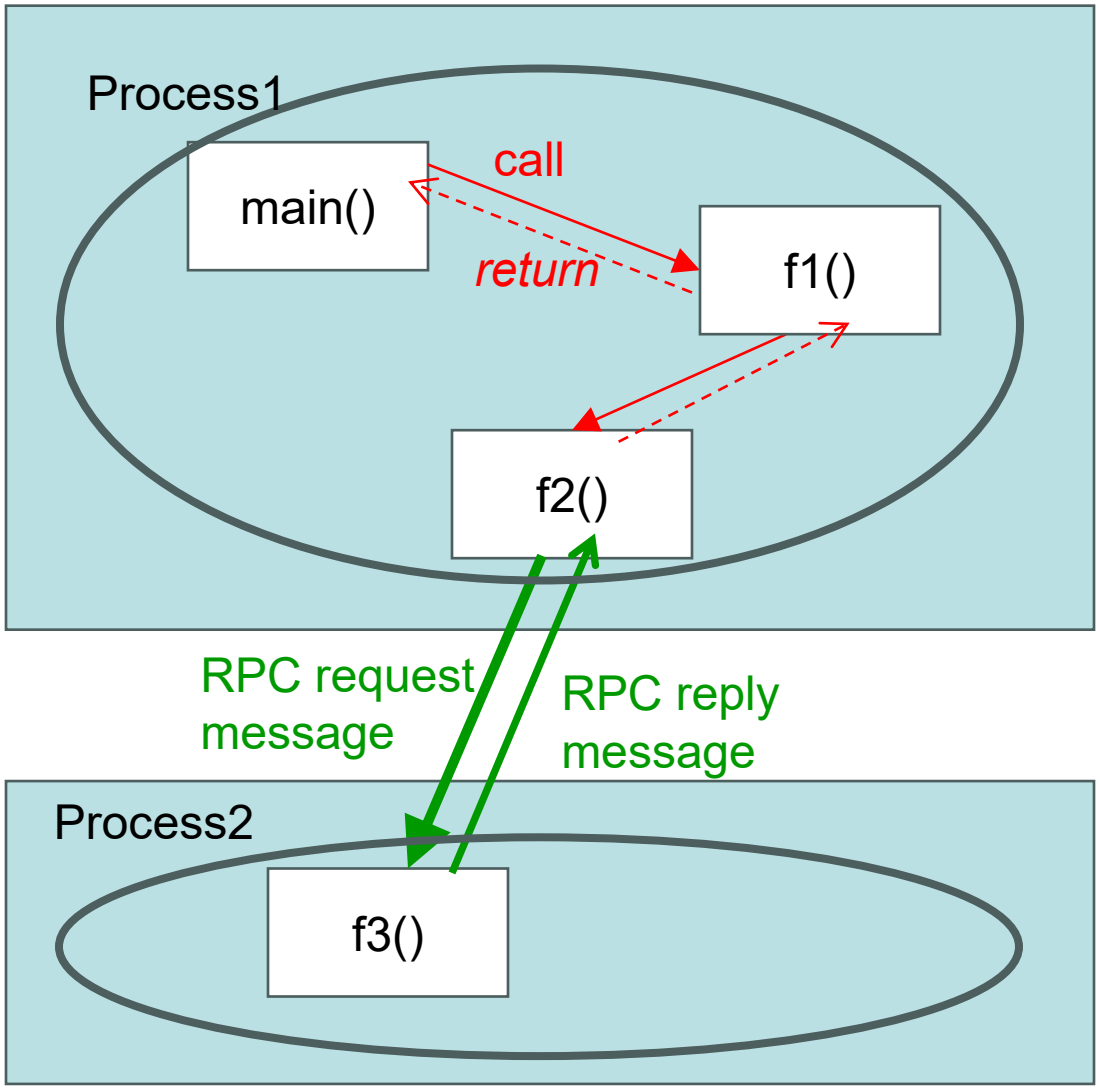
Remote Procedure Call
Remote Method Invocation
The Remote Proxy Pattern

RPC – The principles

Motivation

- RPC = Remote Procedure Call
- Concept was introduced by Birrell and Nelson in 1984
- Goal: Make distributed computing look like centralized(local) computing
- Introduces some abstractions to allow functions in one process to call functions in other processes
- Transforms client-server interaction: allow remote services to be called as procedures
  - Transparency with regard to location, implementation, language
- Implemented and used in most distributed systems, including cloud computing systems
- Counterpart in Object-based settings is called RMI (Remote Method Invocation)
-

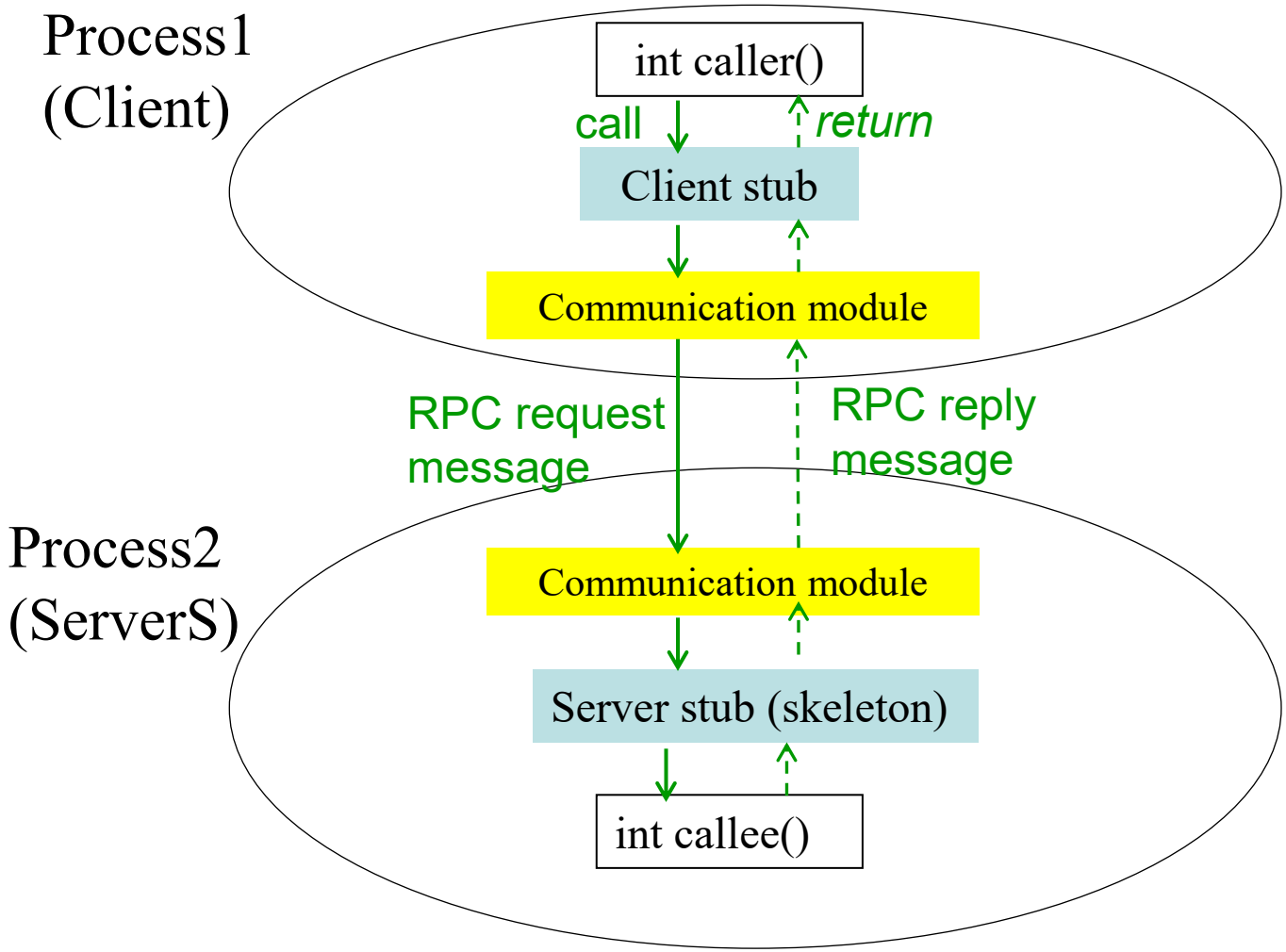
LPC vs RPC


Process1
main()
call
return
f1()
f2()
RPC request message
RPC reply message
Process2
f3()

LPC vs RPC

- **LPC (Local Procedure Call):** Call from one function to another function within the same process
  - Uses stack to pass arguments and return values
  - Accesses objects via pointers (C) or by reference (Java)
- **RPC (Remote Procedure Call):** Call from one function to another function, where caller and callee function reside in different processes
  - Function call crosses a process boundary
  - Accesses procedures via global references
    - Can't use pointers across processes since there is no shared memory space
    - Procedure address = IP + port + procedure number
- Similarly, RMI (Remote Method Invocation) in Object-based settings

RPC Architecture



RPC Architecture – The Components

- **Caller():** the client
- **Callee():** the server
- **Client stub**: has same function signature as callee()
  - In OO setting, has same interface as the remote object (Remote Proxy pattern)
- **Communication Module**: Forwards requests and replies to appropriate hosts
- **Server stub**: calls callee(), allows it to return a value

Marshalling

- Different architectures use different ways of representing data
  - Big endian: IBM z; Little endian: Intel
- Caller (and callee) process uses its own *platform-dependent* way of storing data
- Middleware has a common data representation (CDR)
  - *Platform-independent*
- Marshalling: Caller process converts arguments into CDR format
- Unmarshalling: Callee process extracts arguments from message into its own platform-dependent format
- Return values are marshalled on callee process and unmarshalled at caller process

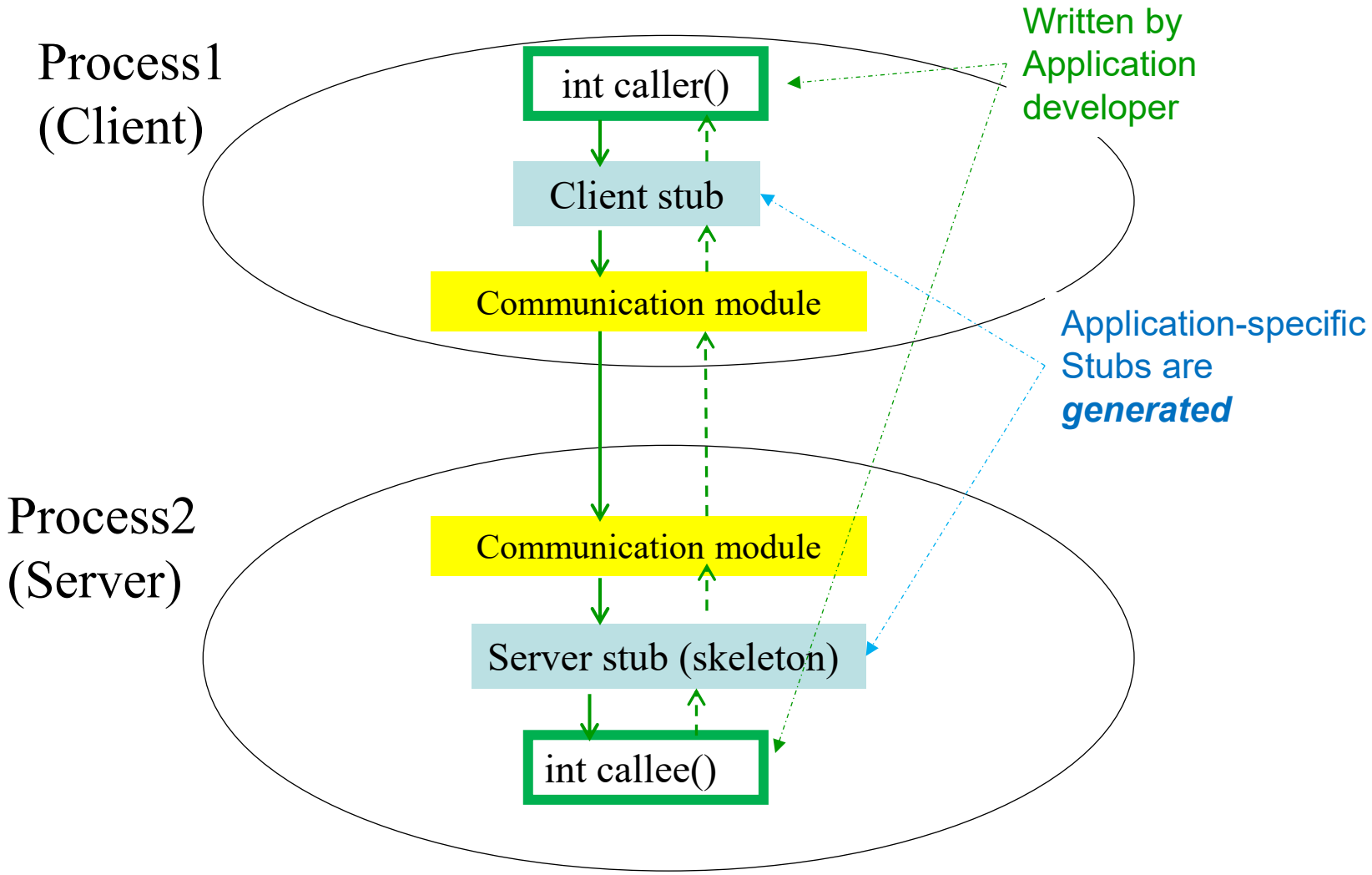
Steps of a RPC

1. **Caller** calls client stub (local call)
2. **Client stub** marshalls parameters, builds message, calls local communication module
3. **Client's communication module** sends message to remote communication module
4. **Server's communication module** gives message to server stub
5. **Server stub** unmarshalls (unpacks) parameters, calls callee (server)
6. **Callee** (server) does work, returns result to the server stub
7. **Server stub** marshalls result in message, calls local communication module
8. **Server-side communication module** sends message to client's communication module
9. **Client's communication module** gives message to client stub
10. **Client Stub** unmarshalls (unpacks) result, returns to caller

Responsabilities of stubs

- The stubs are responsible for managing all details of the remote communication between client and server
- Responsabilities of stubs are standard:
  - Client stub:
    - Packs arguments of caller() function into a message (marshalling)
    - Sends message using communication module
    - Waits to receive response message
    - Unpacks from received message (unmarshalling)
    - Returns result to caller
  - Server stub (skeleton):
    - Receives message from Client stub
    - Unpacks the arguments (unmarshalling)
    - Calls the actual callee function with the given arguments and gets result
    - Packs the result into a message (marshalling)
    - Sends message to client stub
- The stubs perform standard activities but they are still **application specific** because they must know the exact number and types of arguments for the callee function

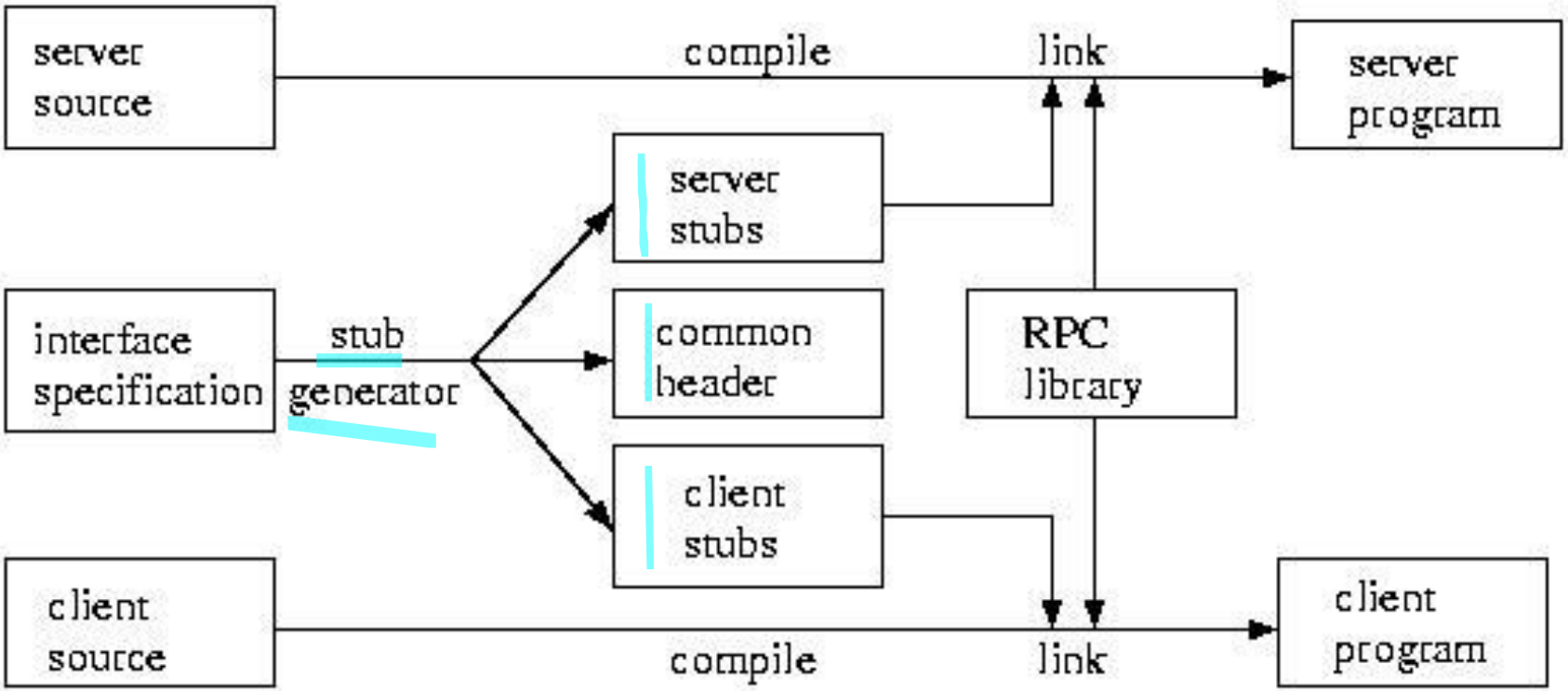
Code: written by developer and generated



Generating Stubs

- A server defines the server's interface using an interface definition language(IDL). The IDL specifies the names, parameters, and types for all client-callable server procedures
    - IDL = neutral language, provides interoperability between different implementation languages for client and server
- Stub functions for a given interface can be generated in two ways:
- **static stub generation:**
  - traditional RPC systems
  - an IDL compiler reads an interface definition and generates code for the 2 stubs stubs (client-side and server-side).
  - The server programmer implements the server procedures and links them with the server-side stubs
  - The client programmer implements the client program and links it with the client-side stubs
- **dynamic stub generation:** the language runtime creates proxy objects at runtime, relies on **reflection**

Code written by developer vs generated



From https://cseweb.ucsd.edu/classes/sp16/cse291-e/applications/ln/lecture3.html

RPC Principles - Example

- Code written by programmer:

Server Interface:

```
int add(int x, int y);
```

Server Procedure:

```
int add(int x, int y) {
    return x+y;
}
```

Will be called by generated server-side stub

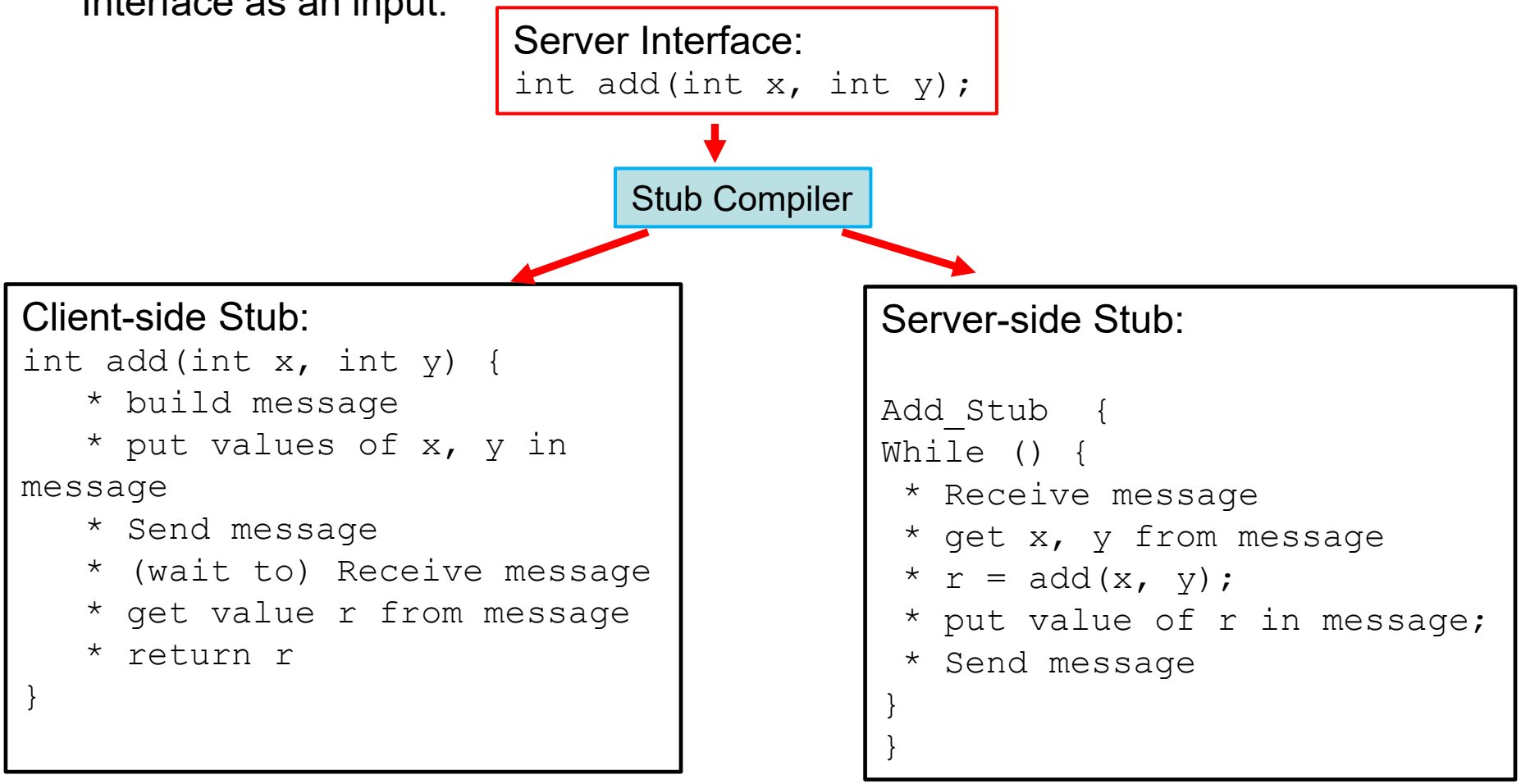
Client Program:

```
…
r = add(3, 6);
…
```

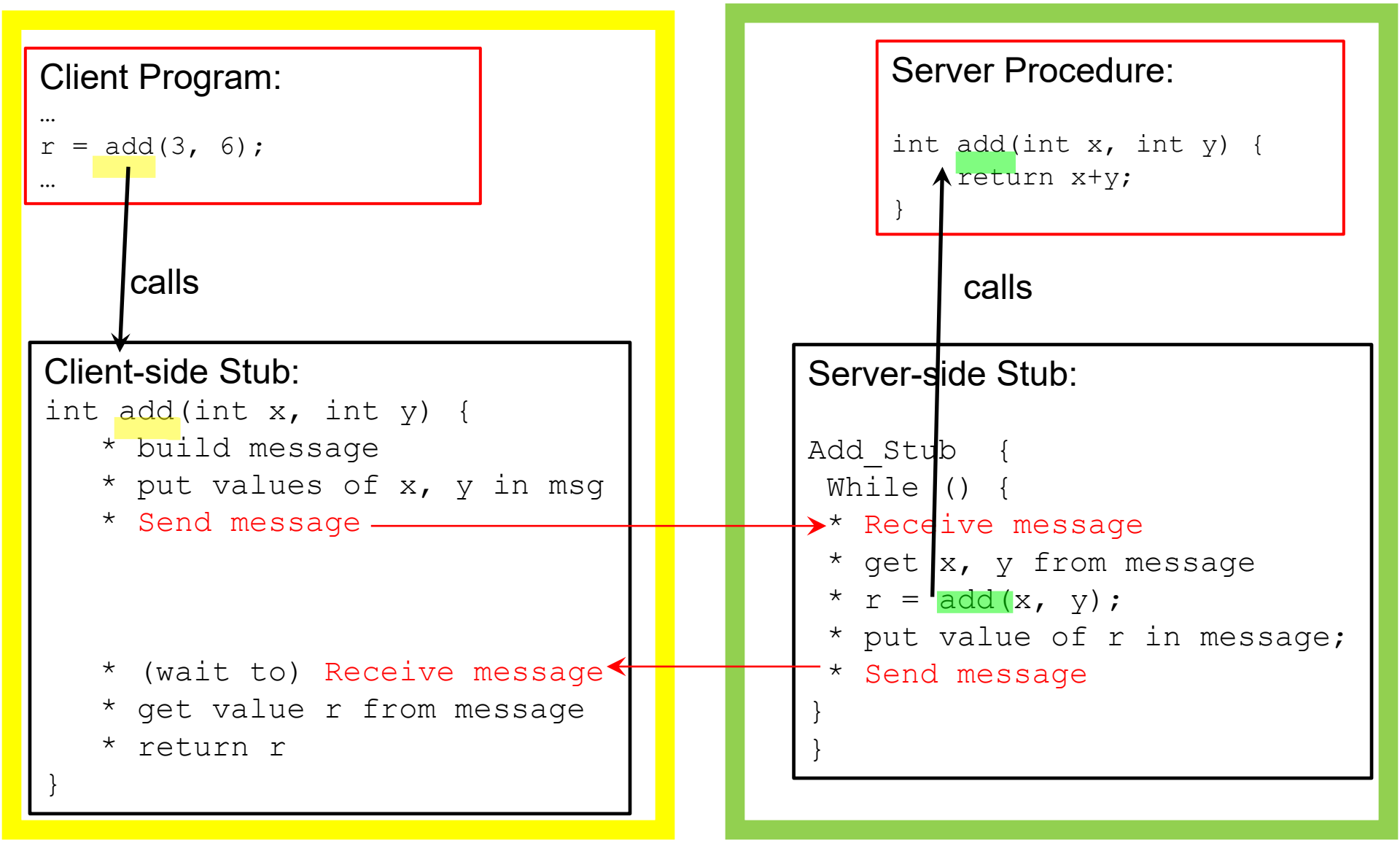
Is linked with add function from generated client-side stub

RPC Principles - Example

- Stub code is usually **generated** by a Stub-compiler taking the Server Interface as an input:

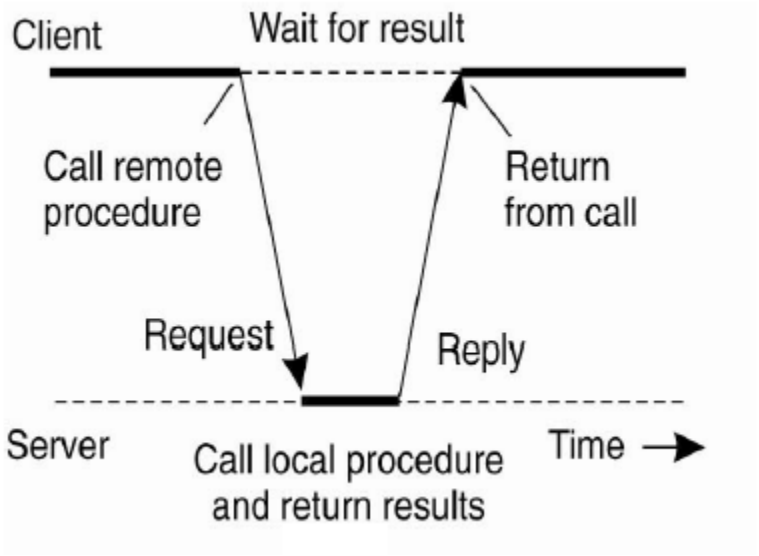


Linking Client and Server Programs

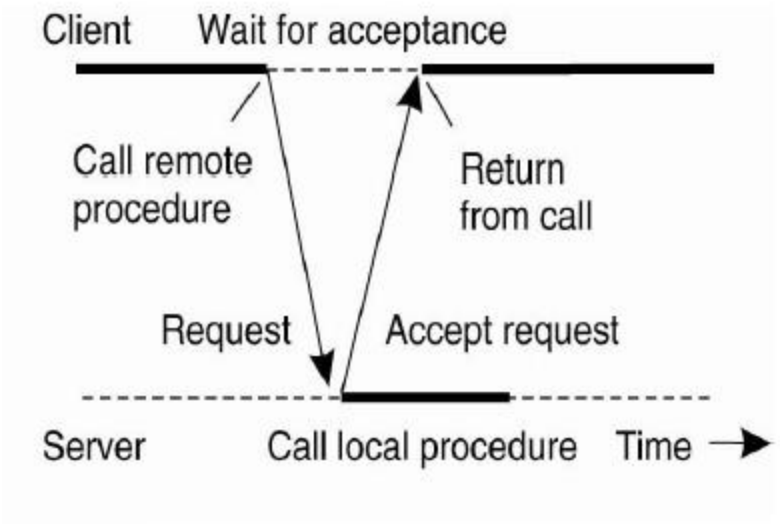


Synchronous vs Asynchronous

- Synchronous RPC: the natural way for procedure calls:
  - RPC call blocks client that waits for server reply



- Asynchronous RPC: few RPC middleware support it
  - Client continues without waiting for server reply
  - Server can ACK as soon as request is received and execute procedure later



RPC issues: Failures

- Different communication failures between sender (client) and receiver (server): what can go wrong?
  1. Request or reply message is lost or delayed, connection is reset - Network is not realiable
  2. Server crashes
     a) before performing service
     b) after performing service
     - client cannot distinguish between 2.a) and 2.b)
  3. Client crashes after sending request

Semantics of remote calls

- What is the semantics of communication in the presence of failures:
  - Maybe
  - At-least-once
  - At-most-once
  - Exactly-once
- What guarantees the middleware about the execution ?
  - More guarantees -> more complexity
  - Exactly once semantics is too costly: most RPC systems implement weaker semantics
  - At-least-once semantics: if client receives reply from server, then remote call has been executed at least once by server
  - At-most-once semantics: if client receives reply from server, then remote call/method has been executed at most once by server

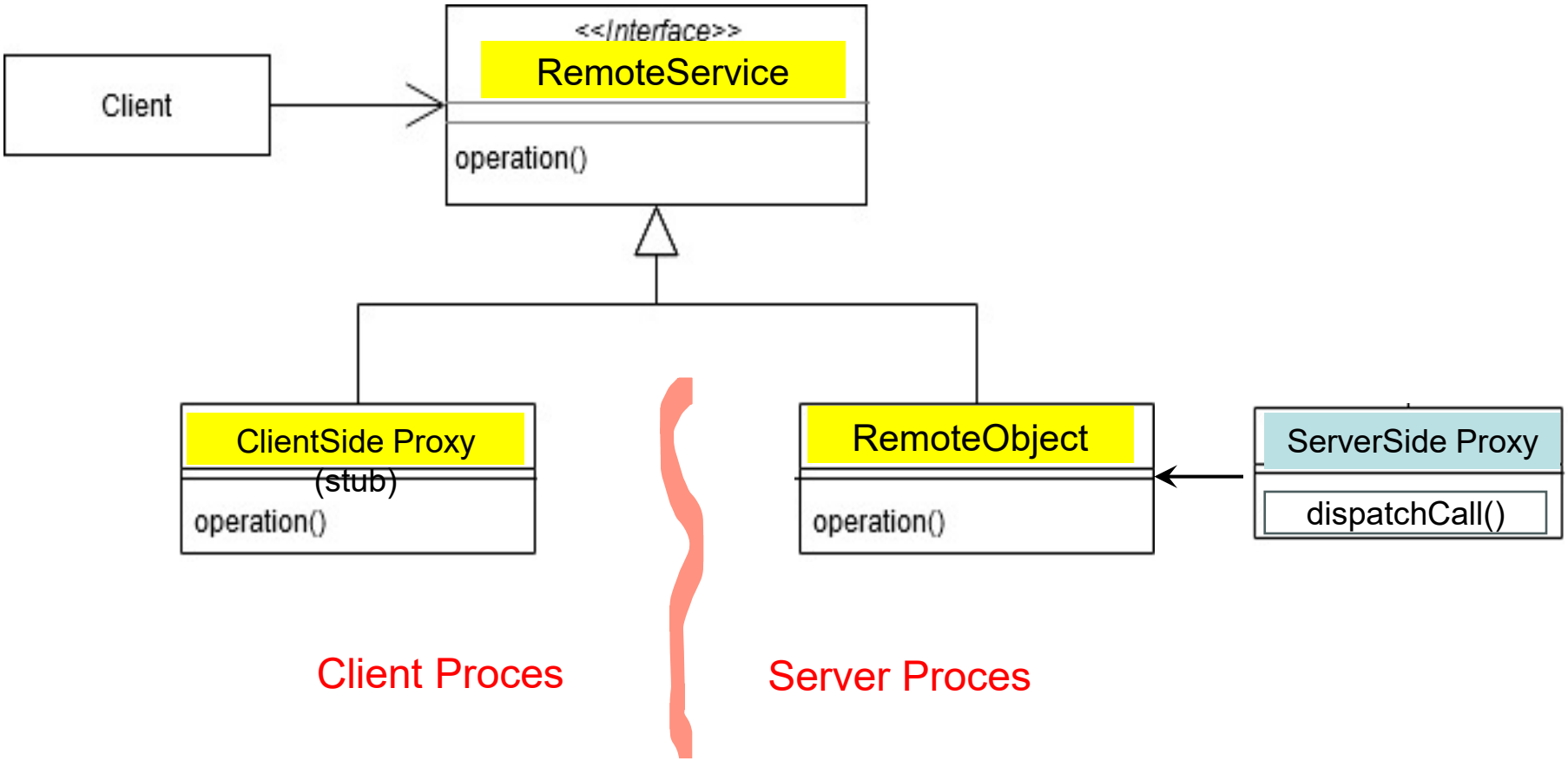
Server Binding

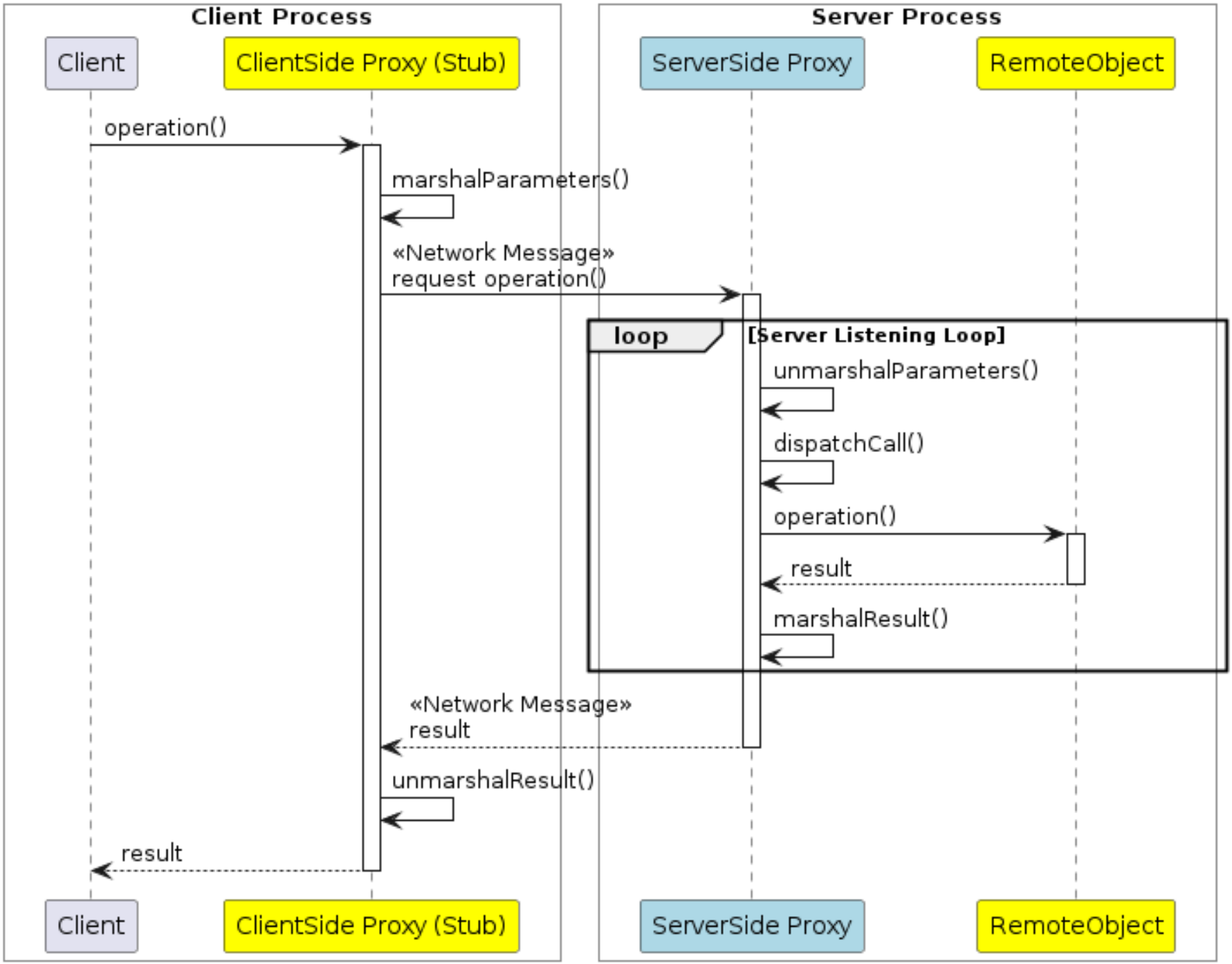
- Binding: how to locate the server endpoint, including the proper process (port or transport address) on it
- Can be static or dynamic
- Static binding
  - Binding is known at design time: server address and other info (e.g., port) are hard-coded
  - Easy and no overhead, but lacks transparency and flexibility
- Dynamic binding
  - At run-time
  - Increased overhead, but gains transparency and flexibility. E.g., we can redirect requests in case of server replication
  - Try to limit overhead
  - Uses a Binder (Naming service, registry service): a service that maintains a table containing mappings from textual names to remote servers. (sort of like DNS, but for the specific middleware)

Remote Method Invocation (RMI)

- RPCs applied to objects (instances of a class)
  - Class: object-oriented abstraction; module with data and operations
  - Separation between interface and implementation
  - Interface is known by several machines, implementation on another machine
- RMIs usually support system-wide object references
  - Parameters can be object references
- Stubs (proxies) implement the Remote Proxy pattern: the client-side stub is a Proxy (implements the same interface as the remote object it represents)

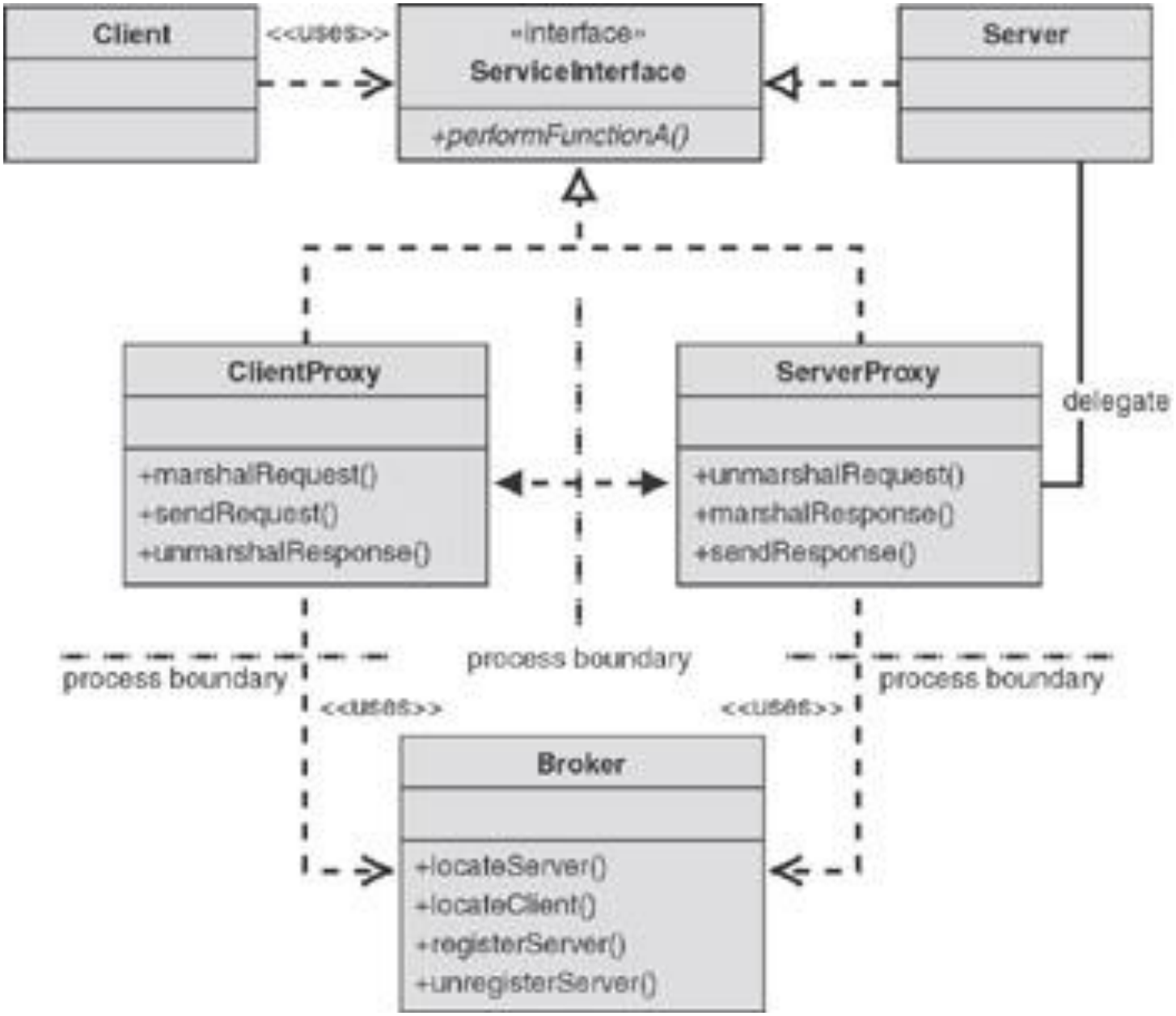
The Remote Proxy Pattern




Client Process
Server Process
Client
ClientSide Proxy (Stub)
ServerSide Proxy
RemoteObject
operation()
marshalParameters()
«Network Message»
request operation()
loop
[Server Listening Loop]
unmarshalParameters()
dispatchCall()
operation()
result
marshalResult()
«Network Message»
result
unmarshalResult()
result
Client
ClientSide Proxy (Stub)
ServerSide Proxy
RemoteObject

The Object Request Broker Pattern

- The Object Request Broker: architectural pattern where a **middleware component (broker)** sits between clients and servers and is responsible for **decoupling clients from servers** (location transparency + flexibility)
  - Service **registration**
  - Service **lookup (binding / naming)**
  - Sometimes it can also do:
    - Message routing / forwarding
    - Load balancing
- Object Request Broker = Remote Proxy (RMI) + Dispatcher
  - RMI hides communication details
  - Broker pattern hides location + connection details
  - RMI without a broker:
    - Client must know **IP + port + protocol**
  - RMI with a broker (dynamic binding):
    - Client just uses a **logical name** and the Broker (Dispatcher) resolves and routes the call



https://flylib.com/books/en/2.359.1.57/1/

Broker Pattern

- The term Broker Pattern is used for 2 different things:
  - Object Request Broker
    - a middleware that allows a client to **invoke methods on remote objects** as if they were local (Combines RPC/RMI with a Dispatcher/Dynamic Binding)
  - Message Broker
    - middleware that enables **asynchronous communication** by sending messages between systems
    - Message Broker stores / queues messages, routes messages to the correct consumer
  - Common principle: Broker is a **middleman** between clients and servers / between participants

Examples of RPC/RMI Technologies

- Procedural: expose **procedures/functions** over a network
  - Sun RPC (1984)
  - Microsoft RPC

- Distributed objects: allow methods of **remote objects** to be invoked as if they were local objects, hiding network details
  - CORBA (1991)
  - Java RMI (1997)
  - WCF (Windows Communication Foundation) (2006)
  - RPyC (2005)

- Service oriented: expose **service interfaces** with explicit contracts and messaging
  - gRPC (2015)