AARHUS UNIVERSITET

Microservices and DevOps

DevOps and Container Technology

Broker Pattern

Henrik Bærbak Christensen

Distribution

Definition: Distributed System

A distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages. (Coulouris, Dollimore, Kindberg, and Blair 2012)

- We stay in the realm of *client-server architectures*
  - One logical server, reactive – just responding to requests
  - Multiple clients, proactive – requesting behavior by server
    - Clients do not know other clients

- We *disregard security for now!*
  - Because it is important but cumbersome…

AARHUS UNIVERSITET

Broker's intent

- To allow OO paradigm for programming

> **Definition: Object-orientation (Responsibility)**
>
> An object-oriented program is structured as a community of interacting agents called objects. Each object has a role to play. Each object provides a service or performs an action that is used by other members of the community.

- … where an object *may reside on another computer*
- However - networks *only support two asynch functions!*

```
void send(Object serverAddress, byte[] message);
byte[] receive();
```

Compare with REST

AARHUS UNIVERSITET

Issues (at least!)

- Send/receive is a too low level a programming model
- Send() does not wait for a reply from server (Asynch)
- Reference to object on *my* machine does not make sense on *remote* computer (memory address)
- Networks does not transfer objects, just bits
- Networks are slow
- Networks and Remote computers may fail
- Networks are insecure, others may listen

AARHUS UNIVERSITET

Elements Overview

- Solutions are
  - Request/Reply Protocol
    - Simulate synchronous call (solves (partly) concurrency issue)
  - Marshalling + Demarshalling
    - Packing objects into bits and back (solves data issue)
  - Proxy Pattern (and Broker pattern)
    - Simulate method call on client (solves programming model issue)
  - Naming Services
    - Use a registry/name service (solves remote location issue)

AARHUS UNIVERSITET

Broker

AARHUS UNIVERSITET

Motivation

- C'mon Henrik? *Broker is so yesterday!*
  - Everything is REST, GraphQL, no one do .NET remoting!
- Right, I agree, but…
  - **It is a bit sad.** I think vendors stole the pattern and turned into a *big ball of mud: SOAP, WSDL, CORBA IDL, Java Security model, [WebService(Namespace=http://kissmycensored.org/)]*
  - **From an architectural point of view, Broker is a clean pattern, well defined roles with architectural integrity**
  - **Broker allows tweaking all the architectural qualities (performance, availability, security, …) in a clean fashion**
- In contrast, REST is pretty *big ball of mud!*

Therefore We will spend energy on it!

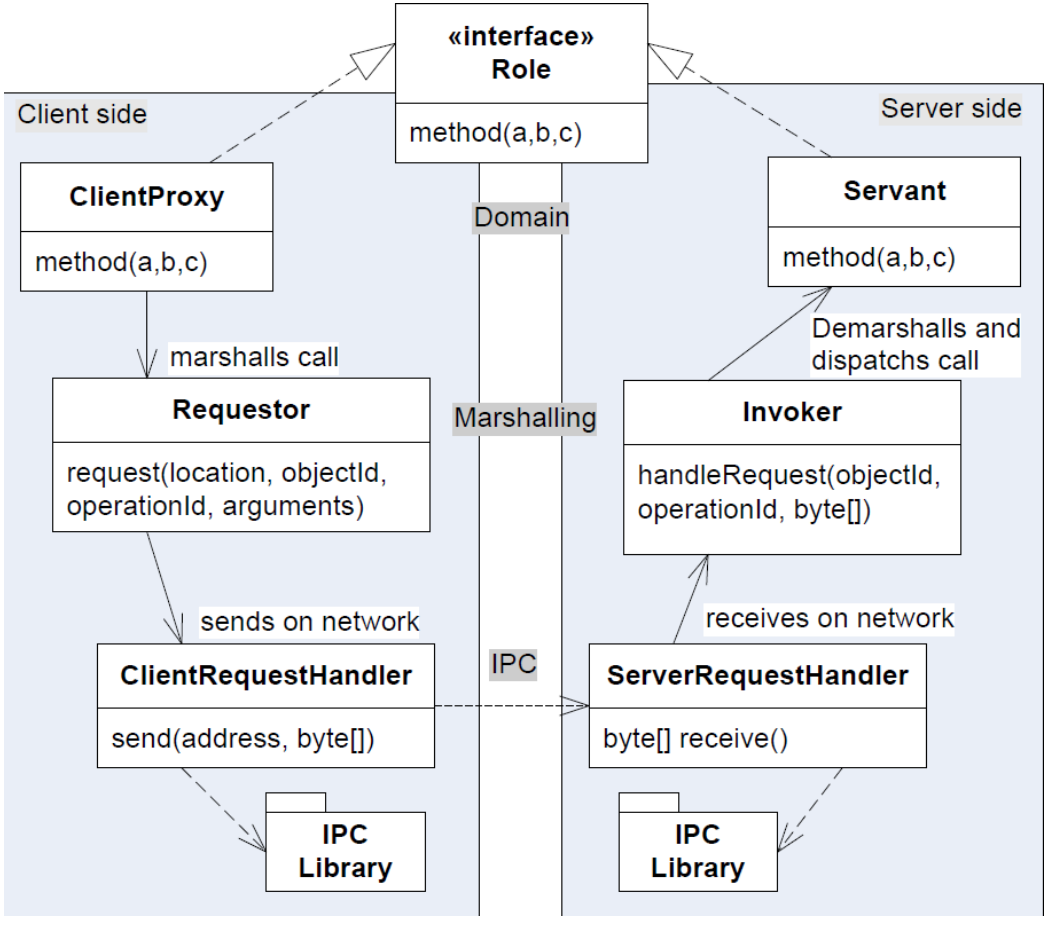
AARHUS UNIVERSITET

Motivation

- **But we will use both in this course**
  - Primary client-server connection is **Broker based**
    - We will use the FRDS.Broker library which
      - Is a lightweight Broker pattern (no fancy tool support)
      - Therefore requires a little bit of handcoding
    - So we can get detailed control the architectural QAs…
    - And I also brainwash you all to acknowledge the beauty of Broker ☺
  - Secondary server-service connections are **HTTP/REST based**
    - Use what-ever library you like
    - So you also code ‘what is widely used out there’
    - And hopefully see some of its deficiencies ☺

Broker

- Broker
  - Complex pattern
  - Three levels
    - Domain level
    - Marshalling level
    - IPC level
  - Two mirrored ‘sides’
    - Client side
    - Server side
- Let’s have a look at each…



AARHUS UNIVERSITET

The 'Side' Perspective

- Client side

ClientProxy

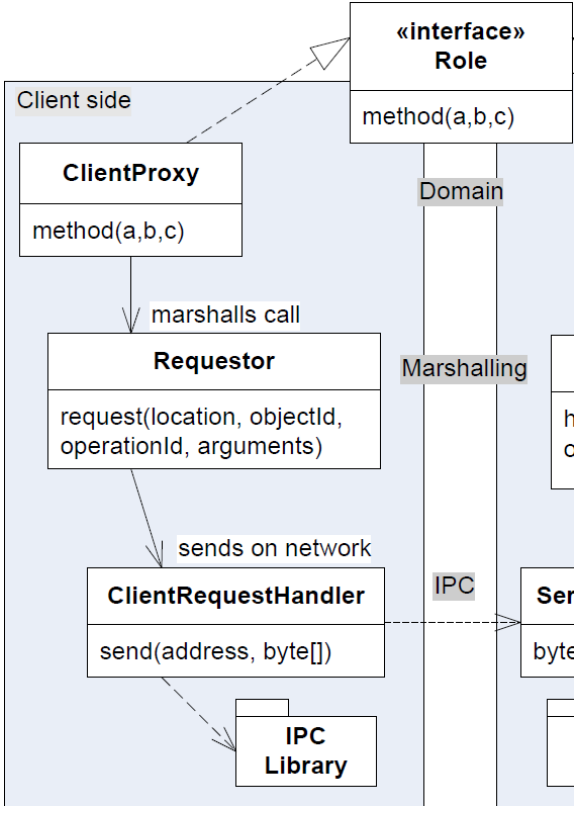
- *Proxy* for the remote servant object, implements the same interface as the servant.
- Translates every method invocation into invocations of the associated *requestors*'s `request()` method.

Requestor

- Performs marshalling of method name and arguments into a **request object**.
- Invokes the **client request handler**'s `send()` method.
- Demarshalls returned **reply object** into return value(s).
- Creates client side exceptions in case of failures detected at the server side.

ClientRequestHandler

- Performs all *inter process communication* on behalf of the client side, interacting with the server side's **server request handler**.



The 'Side' Perspective

- Server side

Servant

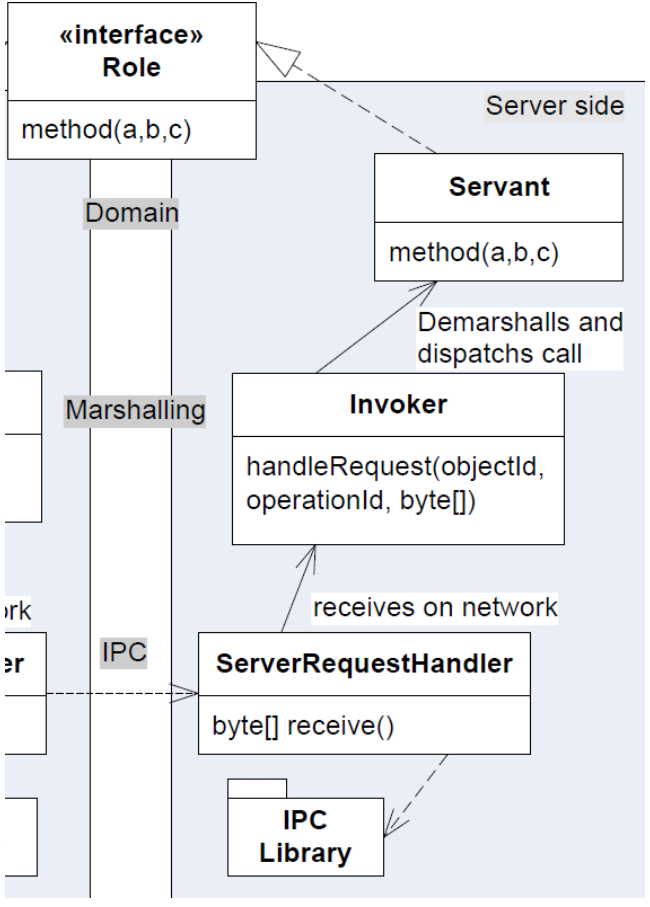
- Domain object with the domain implementation on the server side.

Invoker

- Performs demarshalling of incoming **request objects**.
- Determines servant object, method, and arguments and calls the given method in the identified **servant** object.
- Performs marshalling of the return value from the **servant** object into a **reply object**.

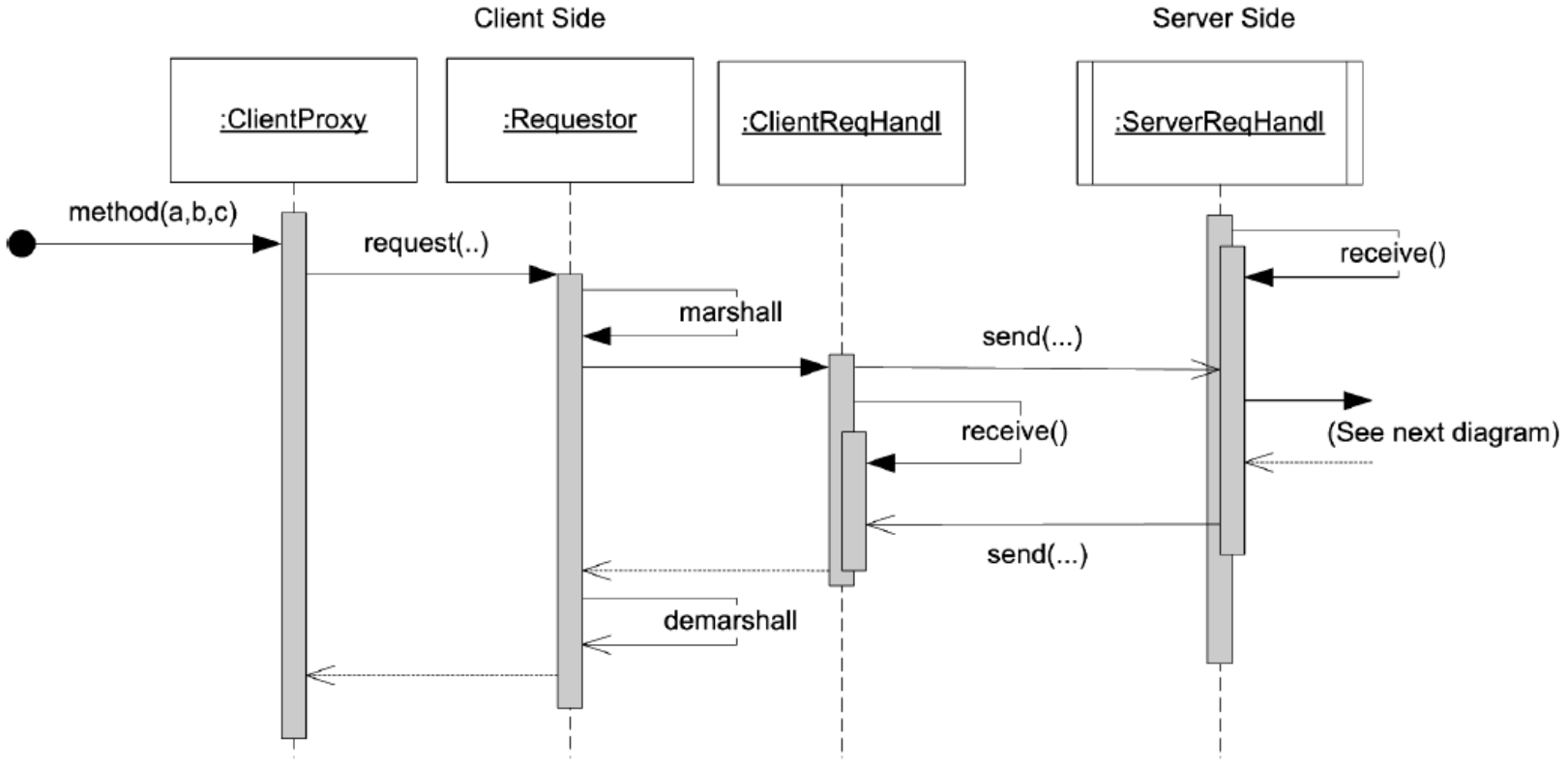
ServerRequestHandler

- Performs all *inter process communication* on behalf of the server side, interacting with the client side's **client request handler**.
- Contains the event loop thread that awaits incoming requests from the network.
- Upon receiving a message, calls the **invoker** with the received **request object**. Sends the **reply object** back to the client side.



AARHUS UNIVERSITET

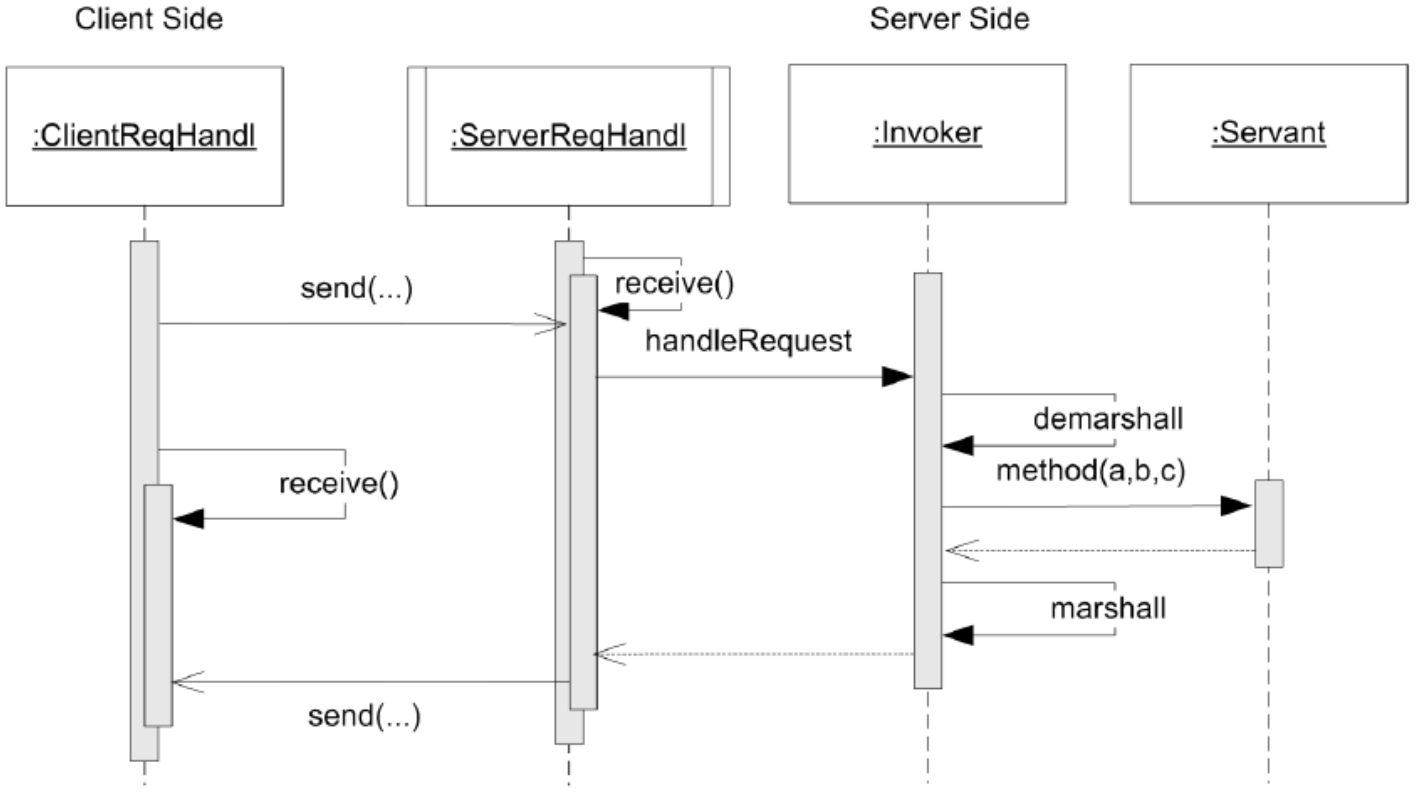
Dynamics (Client)



Broker client side dynamics.

AARHUS UNIVERSITET

Dynamics (Server)

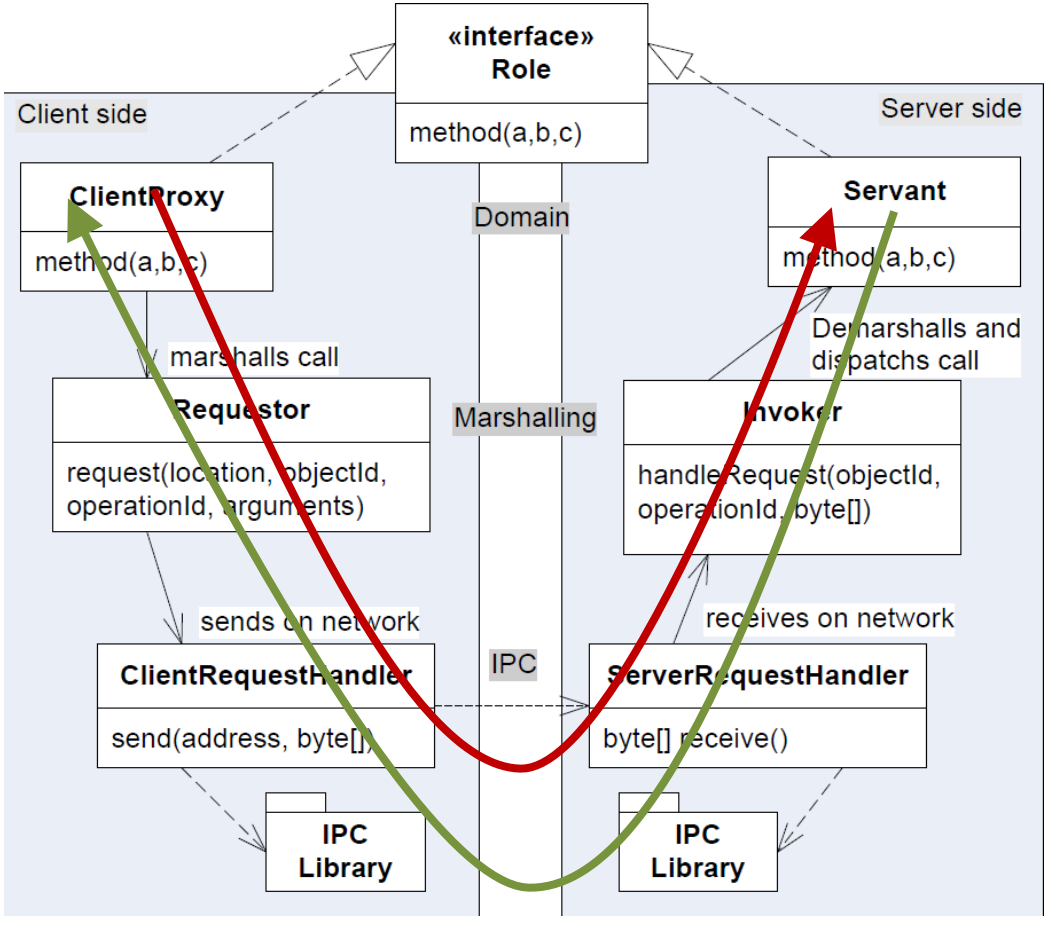


Broker server side dynamics.

AARHUS UNIVERSITET

The Flow

- Method call flows through well defined roles, each with its specific responsibility
  - Translating from high level OO method to binary network and back again…



AARHUS UNIVERSITET

Domain Level

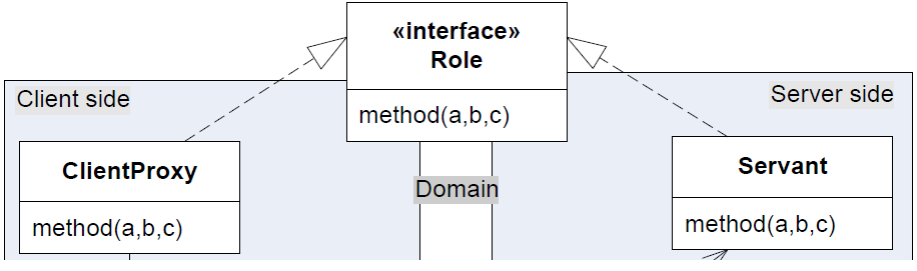
- Domain level represents the actual **Role**

Servant

- Domain object with implementation on the server side

ClientProxy

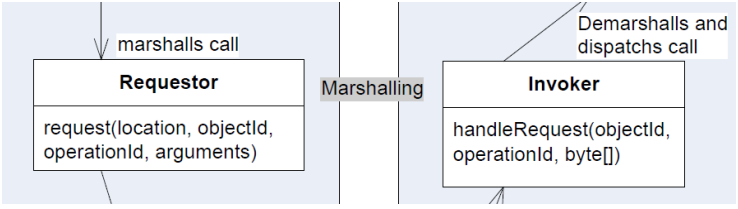
- PROXY for remote object
- Translate every method invocation into invocations of the associated **requestor**'s request method.



AARHUS UNIVERSITET

Marshalling Level

- Encapsulate translation to/from bits and objects



RequestObject

- Encapsulate all information about a method invocation, including object identity, server location, and parameters, in a marshalling format

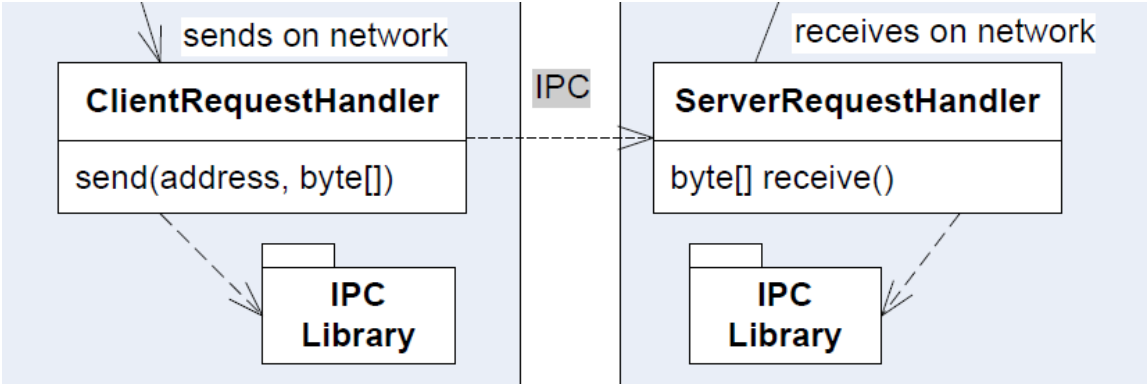
ReplyObject

- Encapsulate all information about return values, including exceptions thrown and error descriptions, from a method invocation in a marshalling format

AARHUS UNIVERSITET

IPC Level

- Interprocess Communication
  - Encapsulate low-level OS/Network communication



AARHUS UNIVERSITET

FRS: Relating to ③ ① ②

- Broker pattern and ③ ① ② ?
  - Yes, yes, and yes
- ③ *Encapsulate what varies*
  - *We would like to vary marshalling format: Requestor+Invoker*
  - *We would like to vary IPC method: RequestHandler*
- ② *Object composition*
  - *We delegate to the requestor. We delegate to the RequestHandl.*
- However, the roles *bleed* into each other somewhat
  - Often RequestHandlers need additional marshalling/demarshalling
  - REST and HTTP solves multiple aspects => natural ‘bleeding’

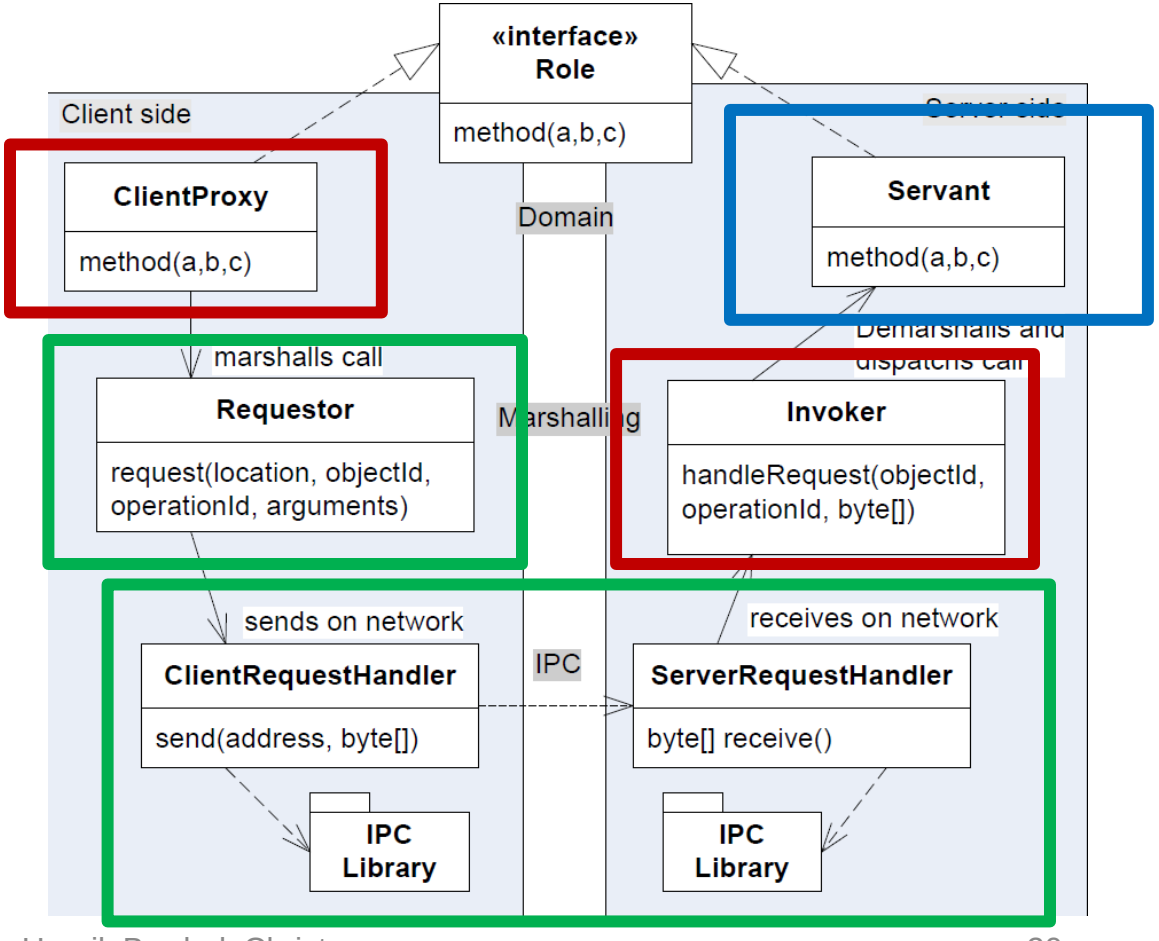
AARHUS UNIVERSITET

MSDO Fast Track Version

FRDS.Broker in 8 slides

FRDS.Broker

- FRDS.Broker
  - Default Impl for roles that are general
- Servant is *given*
- Remains:
  - ClientProxy
  - Invoker
- Basically template code!



So: Let the *player move* east…

AARHUS UNIVERSITET

```
== Welcome to SkyCave, player Mikkel ==
Entering command loop, type "q" to quit, "h" for help.
> e
You moved EAST
You are inside a building, a well house for a large spring.

>
```

```
private void handleSingleCharCommand(char primaryCommand) {
  switch (primaryCommand) {
  // look
  case 'l': {
    systemOut.println(player.getLongRoomDescription());
    break;
  }
  // The movement commands
  case 'n': {
    tryToMove(getDirectionFromChar(primaryCommand));
    break;
  }
  case 's': {
    tryToMove(getDirectionFromChar(primaryCommand));
    break;
  }
  case 'e': {
    tryToMove(getDirectionFromChar(primaryCommand));
    break;
  }
```

```
private void tryToMove(Direction direction) {
  if ( player.move(direction) ) {
    systemOut.println("You moved "+direction);
    systemOut.println(player.getShortRoomDescription())
  } else {
    systemOut.println("There is no exit going " + direction);
  }
}
```

Broker: Let a ‘player.move(d)’ method on the *client side* be ”transparently” translated into a ‘player.move(d)’ on the *server side*…

AARHUS UNIVERSITET

ClientProxy

- Tell requestor to marshall (and beyond), using
  - ‘this’ object’s ID
    - Actually two parts required to identify object, will return to this later…
  - The string constant defined for the move method
  - Return type is UpdateResult.class
  - Provide *all* parameters of the method call

```
@Override
public UpdateResult move(Direction direction) {
  UpdateResult isAllowed = requestor.sendRequestAndAwaitReply(getMangledID(),
          MarshallingKeys.MOVE_METHOD_KEY, UpdateResult.class, direction);
  return isAllowed;
}
```

```
public String getMangledID() {
  return Marshalling.manglePlayerIDAndAccessToken(playerID, accessToken);
}
```

AARHUS UNIVERSITET

PlayerInvoker

```java
@Override
public String handleRequest(String request) {
  ReplyObject reply = null;

  // Do the demarshalling
  RequestObject requestObject =
          gson.fromJson(request, RequestObject.class);

  // Cache the name of called method
  String operationName = requestObject.getOperationName();

  // objectId is a mangling of both the player's playerId and accessToken,
  // so we have do demangle it to get the two parts
  String[] parts = Marshalling.demanglePlayerIDAndAccessToken(requestObject.getObjectId());
  String playerId = parts[0];
  String accessToken = parts[1];

  // Payload is delivered as JsonArray from the server request handler.
  JsonArray array = JsonParser.parseString(requestObject.getPayload()).getAsJsonArray();

  try {
    // Fetch the player object from the name service
    Player player = objectManager.getPlayerNameService().get(playerId);
```

Actual ‘upcall’

```java
// === MOVE
else if (operationName.equals(MarshallingKeys.MOVE_METHOD_KEY)) {
  // move(direction)
  String directionAsString = gson.fromJson(array.get(0), String.class);
  Direction direction = Direction.valueOf(directionAsString);
  UpdateResult isValid = player.move(direction);

  reply = new ReplyObject(HttpServletResponse.SC_OK,
          gson.toJson(isValid));
}
```

AARHUS UNIVERSITET

Exercise Algorithms ☺

- Iteration 0 contains two Broker exercises.
  - Implement the missing PlayerClientProxy methods
  - Implement the extra *if(opName.equals("method")){}* upcalls

- Test it using JUnit
  - Huh – how? It is remote calls?

- Broker *roles* to the rescue…

AARHUS UNIVERSITET

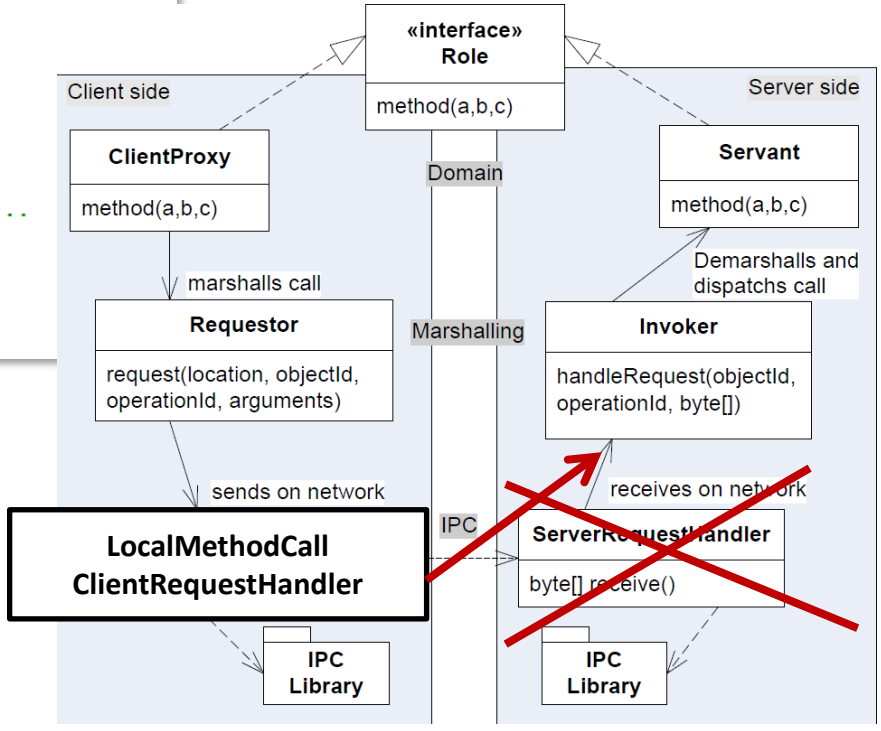
”Fake Object” IPC

```
public class LocalMethodCallClientRequestHandler implements ClientRequestHandler {

  private Invoker invoker;
  private ReplyObject lastReply;

  public LocalMethodCallClientRequestHandler(Invoker invoker) {
    this.invoker = invoker;
  }

  @Override
  public ReplyObject sendToServer(RequestObject requestObject) {
    // The send to the server can be mimicked by a direct method call...
    lastReply = invoker.handleRequest(requestObject.getObjectId(),
        requestObject.getOperationName(),
        requestObject.getPayload());
    return lastReply;
  }
```



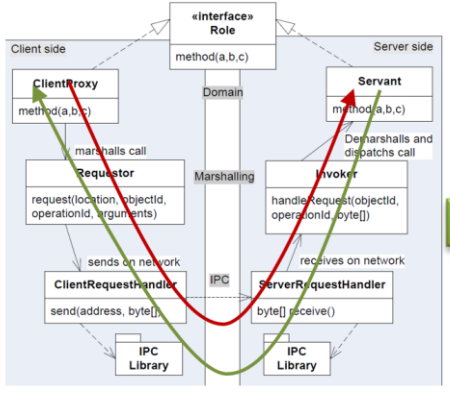
No need to start server.
No concurrency.

All aspects (except IPC) can be TDD’ed
- [Compare with REST!]

AARHUS UNIVERSITET

Establishing the Chain

- Broker roles are a chain that needs to be established
  - Pass ‘servant’ to invoker, pass to request handler, pass to requester, pass to client proxy…
  - (A bit hidden in SkyCave code due to reusing test fixture in multiple places, and because Player is instantiated by a login into the cave…)




```
public static Cave createCaveProxyForTesting() {
  // Create the server tier
  ObjectManager objMgr = CommonCaveTests.createTestDoubledConfiguredCave();

  Invoker invoker = objMgr.getInvoker();

  ClientRequestHandler crh = new LocalMethodCallClientRequestHandler(invoker);
  Requestor requestor = new StandardJSONRequestor(crh);

  // Create the cave proxy
  return new CaveProxy(requestor);
}
```

AARHUS UNIVERSITET

A Cleaner Version

- From the TeleMed case in the FRDS book…

```
@Before
public void setup() {
  teleObs1 = HelperMethods.createObservation120over70forNancy();
  // Create server side implementations
  xds = new FakeObjectXDSDatabase();
  TeleMed teleMedServant = new TeleMedServant(xds);

  // Server side broker implementations
  Invoker invoker = new StandardJSONInvoker(teleMedServant);

  // Create client side broker implementations
  ClientRequestHandler clientRequestHandler = new LocalMethodCallClientRequestHandler(invoker);
  Requestor requestor = new StandardJSONRequestor(clientRequestHandler);

  // Finally, create the client proxy for the TeleMed
  teleMed = new TeleMedProxy(requestor);
}
```

AARHUS UNIVERSITET

Summary