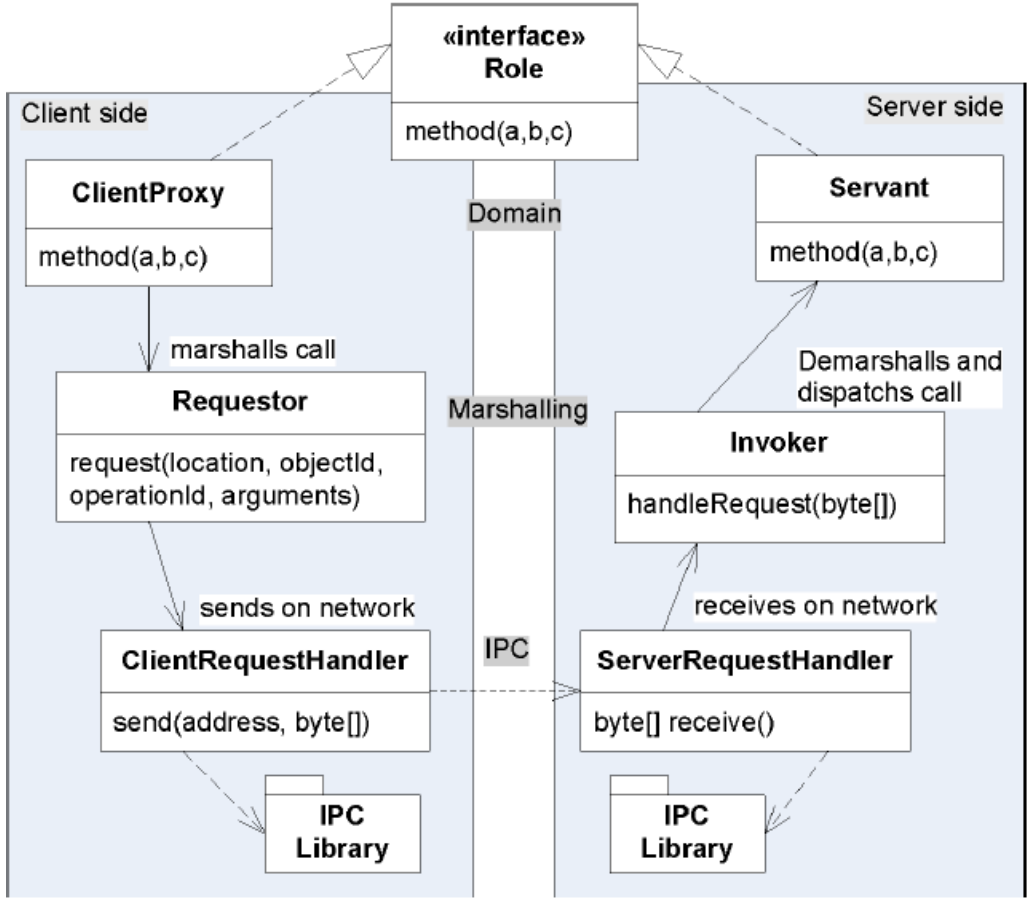
AARHUS UNIVERSITET

Software Engineering and Architecture

Broker Pattern
Architectural Pattern for
Remote Method Invocation

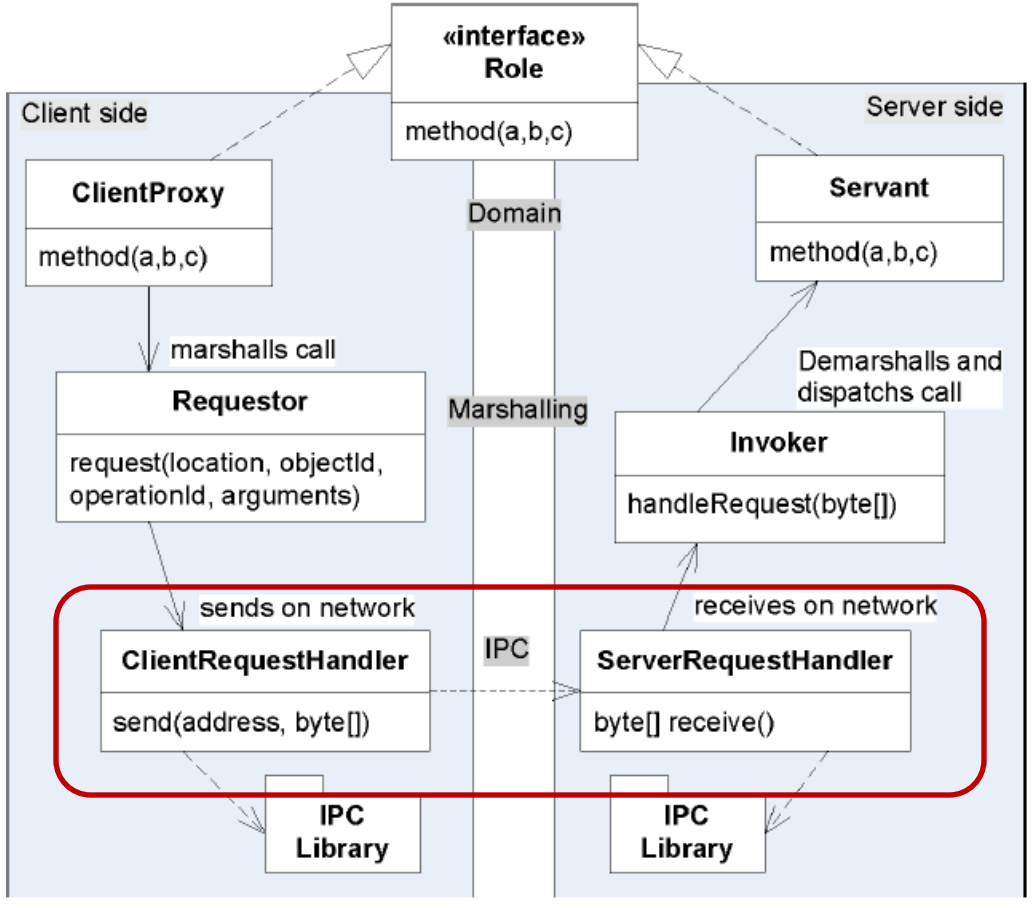
Broker

- Broker "bundles" the four elements:
- Solutions are
  - Request/Reply protocol
  - Marshalling
  - Proxy Pattern
  - Naming Systems



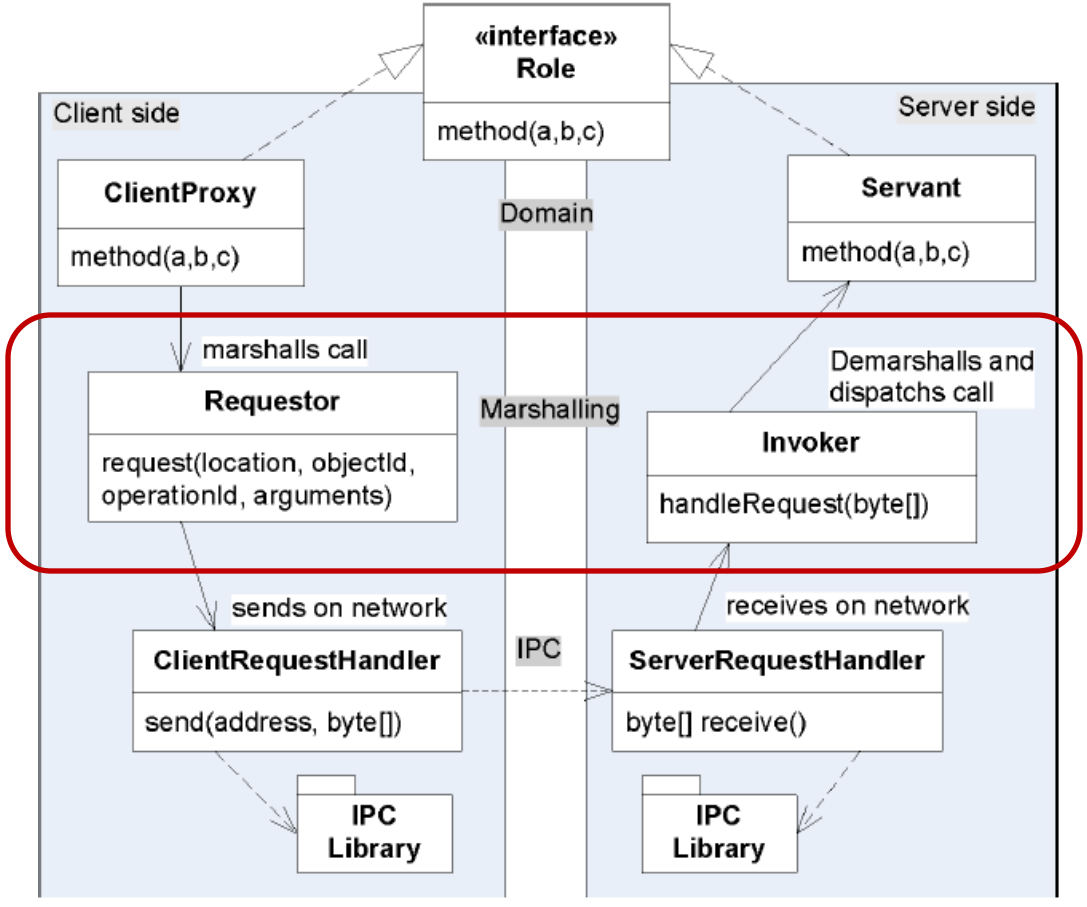
Broker

- Broker bundles the four elements:
- Solutions are
  - Request/Reply protocol
  - Marshalling
  - Proxy Pattern
  - Naming Systems



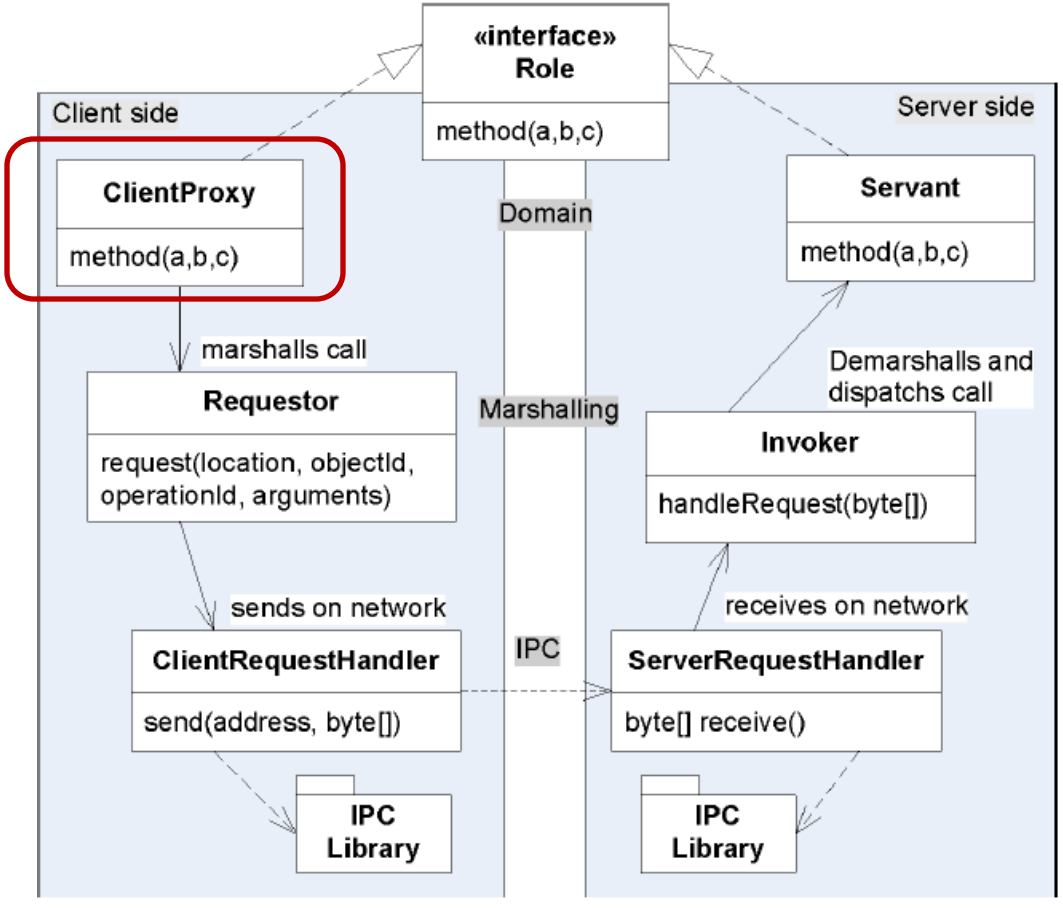
Broker

- Broker bundles the four elements:
- Solutions are
  - Request/Reply protocol
  - Marshalling
  - Proxy Pattern
  - Naming Systems



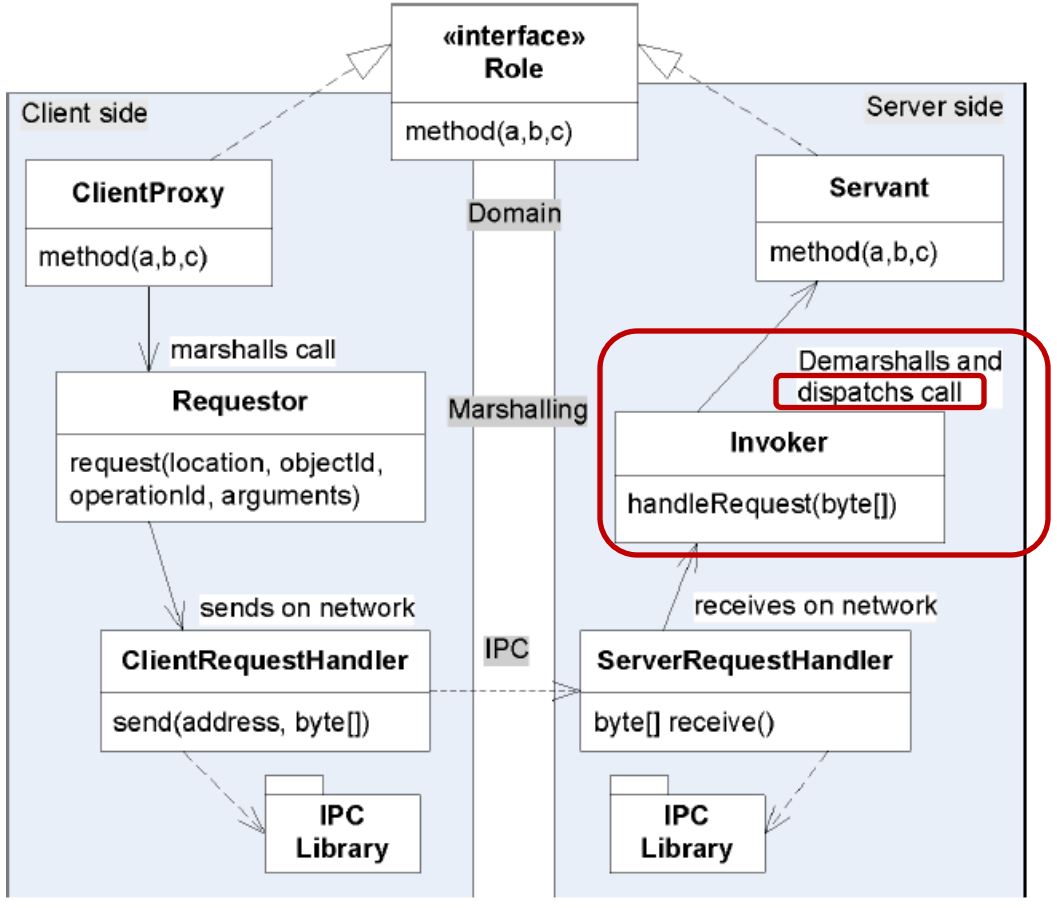
Broker

- Broker bundles the four elements:
- Solutions are
  - Request/Reply protocol
  - Marshalling
  - Proxy Pattern
  - Naming Systems



Broker

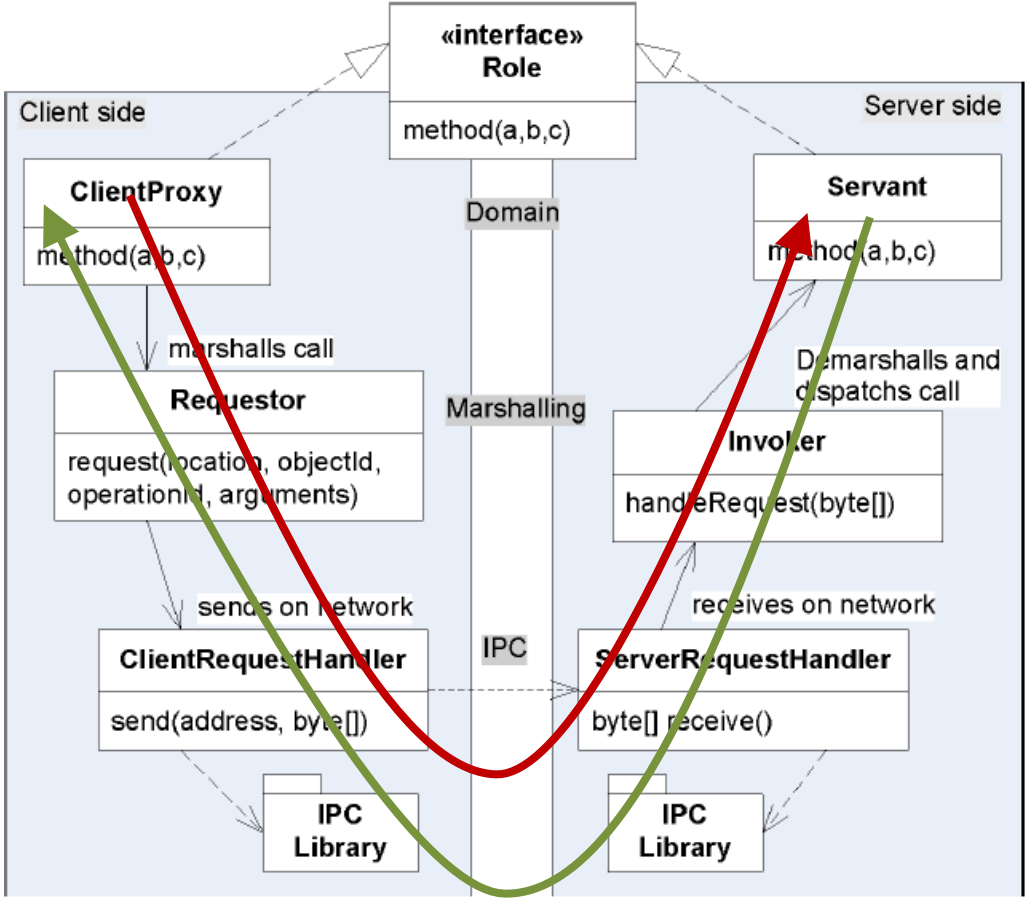
- Broker bundles the four elements:
- Solutions are
  - Request/Reply protocol
  - Marshalling
  - Proxy Pattern
  - Naming Systems



AARHUS UNIVERSITET

A Picture of the 'Flow'

- The method call *flows* from the client's ClientProxy, through intermediaries until it ends in the Servant
  - Each intermediate responsible for a transformation
    - domain-to-network and vice-versa
- … and back again…
  - "Chained calls"



AARHUS UNIVERSITET

The 'Side' Perspective

- Client side

ClientProxy

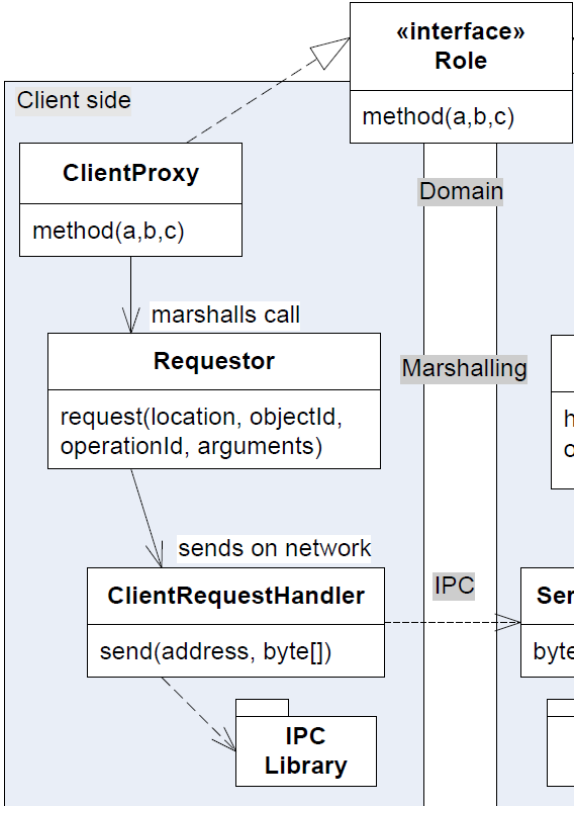
- *Proxy* for the remote servant object, implements the same interface as the servant.
- Translates every method invocation into invocations of the associated *requestors*'s `request()` method.

Requestor

- Performs marshalling of object identity, method name and arguments into a byte array.
- Invokes the **ClientRequestHandler**'s `send()` method.
- Demarshalls returned byte array into return value(s).
- Creates client side exceptions in case of failures detected at the server side or during network transmission.

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

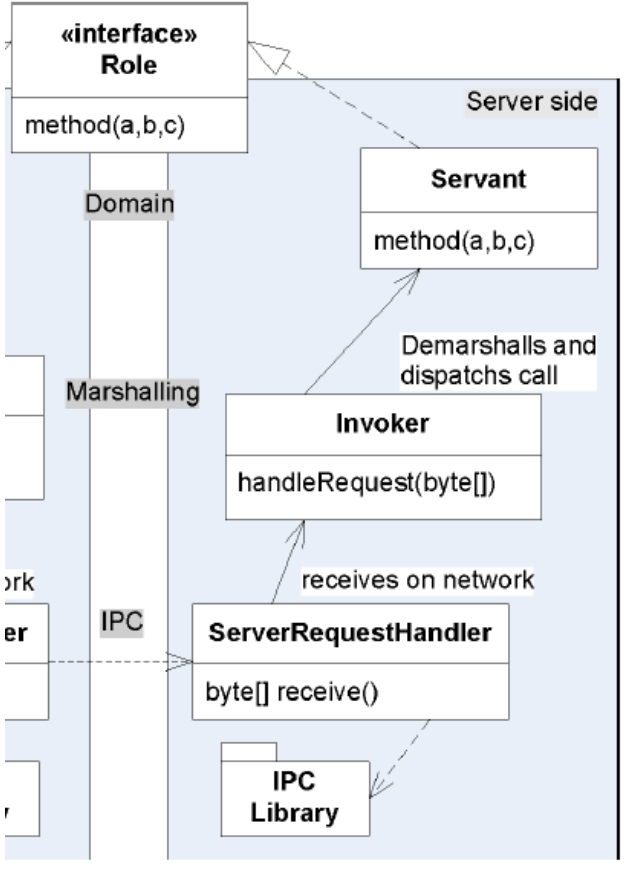
- Domain object with the domain implementation on the server side.

Invoker

- Performs demarshalling of incoming byte array.
- 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 byte array, or in case of server side exceptions or other failure conditions, return error replies allowing the **Requestor** to throw appropriate exceptions.

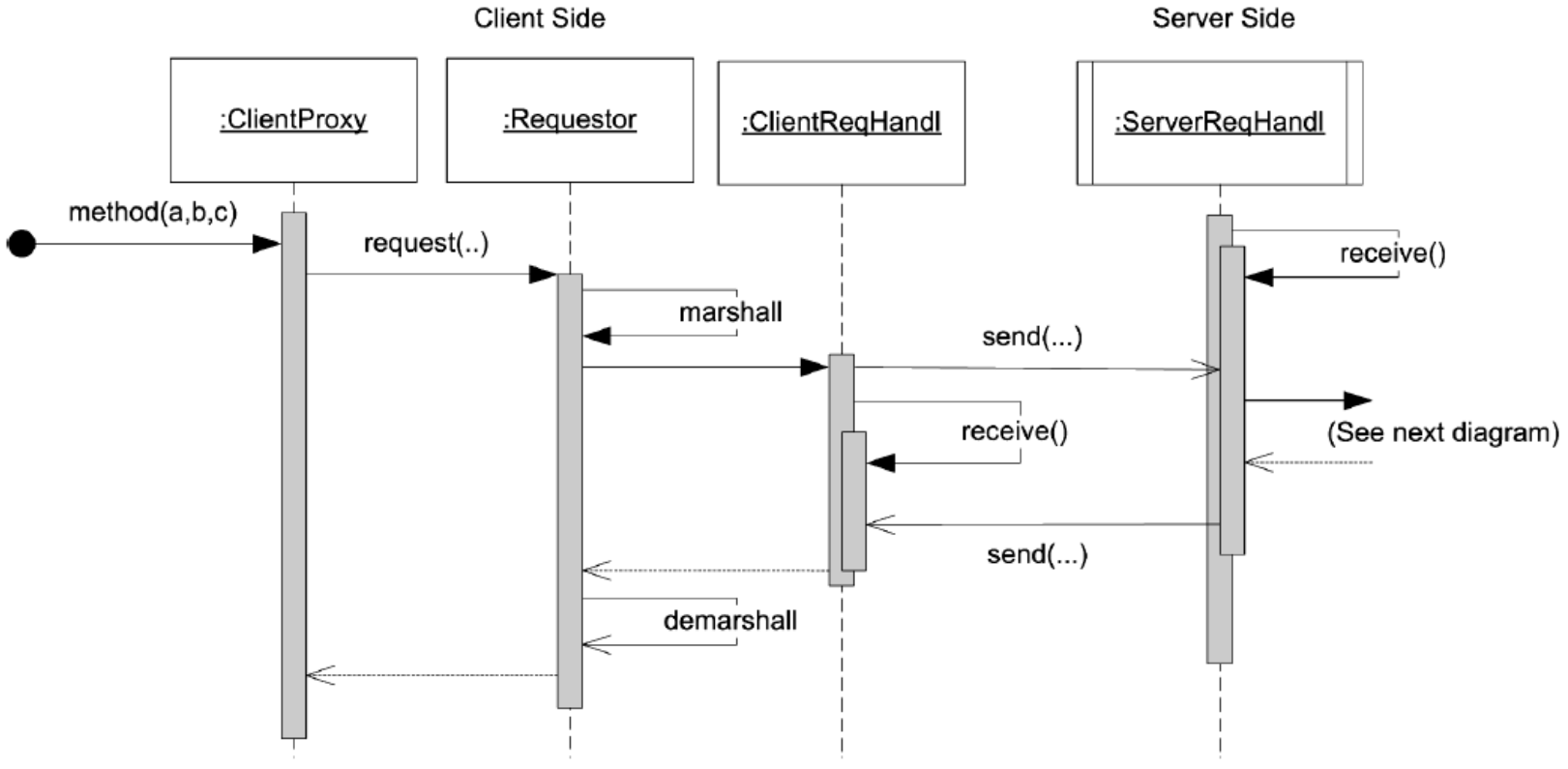
ServerRequestHandler

- Performs all *inter process communication* on behalf of the server side, interacting with the client side's **ClientRequestHandler**.
- Contains the event loop thread that awaits incoming requests from the network.
- Upon receiving a message, calls the **Invoker**'s `handleRequest` method with the received byte array. Sends the reply back to the client side.



AARHUS UNIVERSITET

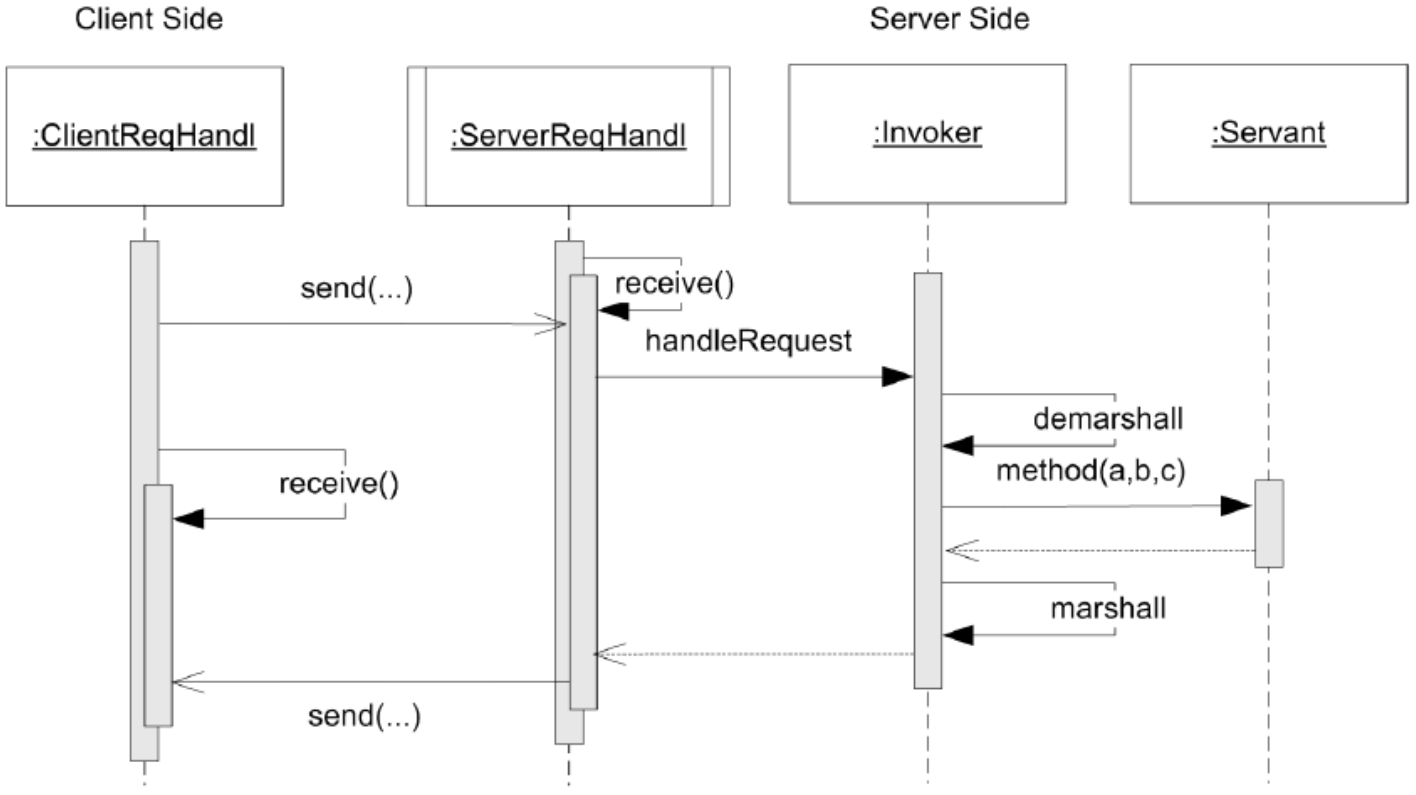
Dynamics (Client)



Broker client side dynamics.

AARHUS UNIVERSITET

Dynamics (Server)



Broker server side dynamics.

Domain Level

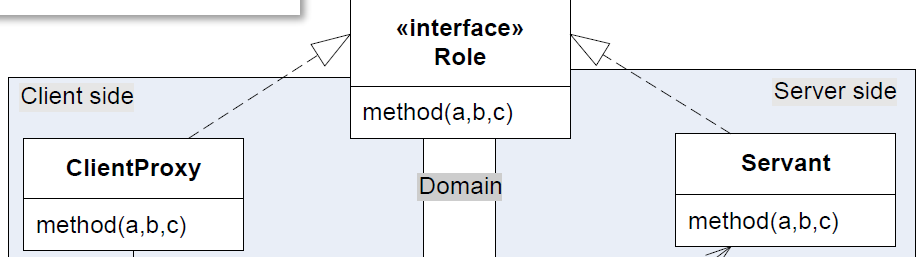
- Domain level represents the actual **Role**

Servant

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

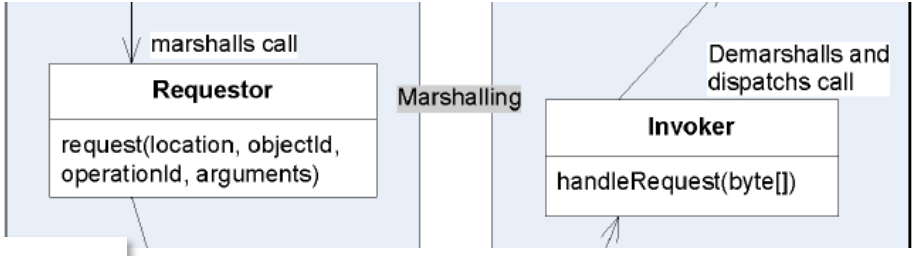
ClientProxy

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



Marshalling Level

- Encapsulate translation to/from bits and objects



Requestor

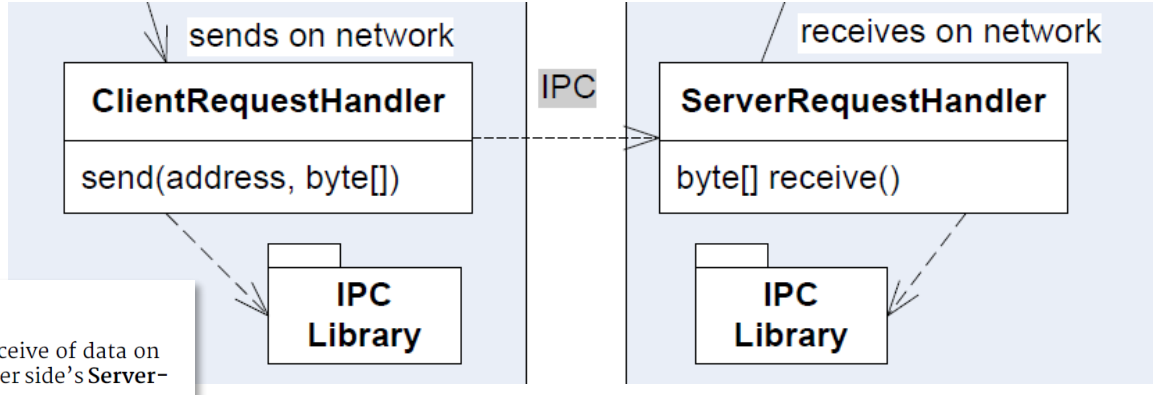
- Performs marshalling of object identity, method name and arguments into a byte array.
- Invokes the **ClientRequestHandler**'s `send()` method.
- Demarshalls returned byte array into return value(s).
- Creates client side exceptions in case of failures detected at the server side or during network transmission.

Invoker

- Performs demarshalling of incoming byte array.
- 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 byte array, or in case of server side exceptions or other failure conditions, return error replies allowing the **Requestor** to throw appropriate exceptions.

IPC Level

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



ClientRequestHandler

- Performs all *inter process communication* send/receive of data on behalf of the client side, interacting with the server side's **Server-RequestHandler**.

ServerRequestHandler

- Performs all *inter process communication* on behalf of the server side, interacting with the client side's **ClientRequestHandler**.
- Contains the event loop thread that awaits incoming requests from the network.
- Upon receiving a message, calls the **Invoker**'s `handleRequest` method with the received byte array. Sends the reply back to the client side.

AARHUS UNIVERSITET

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: xRequestHandler*

- ② *Object composition*
  - *We delegate to the requestor. We delegate to the RequestHandl.*

In Practice

How Does It Look Then…

AARHUS UNIVERSITET

The TeleMed Interface

- Will only look at the two methods to
  - Upload
    - *processAndStore*
  - Download
    - *getObservationsFor*

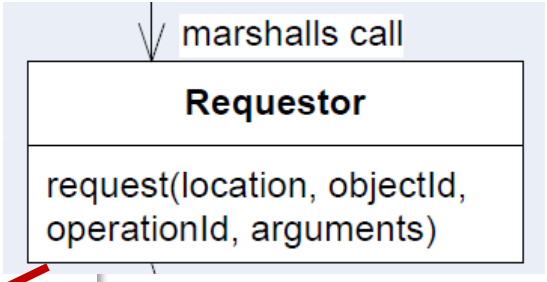
```
public interface TeleMed {

  /**
   * Process a tele observation into the HL7 format and store it
   * in the XDS database tier.
   *
   * @param teleObs
   *          the tele observation to process and store
   * @return the id of the stored observation
   * @throws IPCException in case of any IPC problems
   */
  String processAndStore(TeleObservation teleObs);

  /**
   * Retrieve all observations for the given time interval for the
   * given patient. If no observations exists return a 0 sized
   * list.
   *
   * @param patientId
   *          the ID of the patient to retrieve observations for
   * @param interval
   *          define the time interval that measurements are
   *          wanted for
   * @return list of all observations
   * @throws IPCException in case of any IPC problems
   */
  List<TeleObservation> getObservationsFor(String patientId,
    TimeInterval interval);
```

AARHUS UNIVERSITET

TeleMed Proxy

- ClientProxy = Proxy calls

```
public class TeleMedProxy implements TeleMed, ClientProxy {
  public static final String TELEMED_OBJECTID = "singleton";

  private final Requestor requestor;
  public TeleMedProxy(Requestor requestor) {
    this.requestor = requestor;
  }

  @Override
  public String processAndStore(TeleObservation teleObs) {
    String uid =
      requestor.sendRequestAndAwaitReply(TELEMED_OBJECTID, OperationNames.PROCESS_AND_STORE_OPERATION,
      String.class, teleObs);
    return uid;
  }

  @Override
  public List<TeleObservation> getObservationsFor(String patientId, TimeInterval interval) {
    Type collectionType =
      new TypeToken<List<TeleObservation>>(){}.getType();
    List<TeleObservation> returnedList;
    try {
      returnedList = requestor.sendRequestAndAwaitReply(TELEMED_OBJECTID,
              OperationNames.GET_OBSERVATIONS_FOR_OPERATION,
              collectionType, patientId, interval);
```



Note: *There is only a single TeleMed servant object. Thus the objectId is 'not applicable'*

Note: *location = server, is provided as a global parameter, and not part of parameter list…*

AARHUS UNIVERSITET

Identity of Methods

- Remember: *We can only send byte arrays aka. Strings*
- Need to Marshall method names as well.

```
public class OperationNames {
  // Method names are prefixed with the type of the method receiver ('telemed') which
  // can be used in when serveral different types of objects are present at the server side
  // and is also helpful in case of failure on the server side where log files can be
  // inspected.
  public static final String PROCESS_AND_STORE_OPERATION = "telemed-process-and-store";
  public static final String GET_OBSERVATIONS_FOR_OPERATION = "telemed-get-observation-for";
  public static final String CORRECT_OPERATION = "telemed-correct";
  public static final String GET_OBSERVATION_OPERATION = "telemed-get-observation";
  public static final String DELETE_OPERATION = "telemed-delete";
```

- ”Mangling” = Concatenate class name and method name

Requestor

General Implementation!
Use JSON and the GSON library

Generic return type is pretty helpful…

And Object… = arrays of mixed types are really nasty that required some googling to find out how.

This is code provided by the FRDS.Broker library!

```
@Override
public <T> T sendRequestAndAwaitReply(String objectId, String operationName,
                                      Type typeOfReturnValue, Object... arguments) {
  // Perform marshalling, first arguments, next full request
  String marshalledArgumentList = gson.toJson(arguments);
  RequestObject request = new RequestObject(objectId, operationName, marshalledArgumentList);
  String marshalledRequest = gson.toJson(request);
  // Ask CRH to do the network call
  String marshalledReply = clientRequestHandler.sendToServerAndAwaitReply(marshalledRequest);
  // Demarshall the reply
  ReplyObject reply = gson.fromJson(marshalledReply, ReplyObject.class);
  // First, verify that the request succeeded
  if (!reply.isSuccess()) {
    throw new IPCException(reply.getStatusCode(),
        "Failure during client requesting operation '"
                + operationName
                + "'. ErrorMessage is: "
                + reply.errorDescription());
  }
  // No errors - so get the payload of the reply
  String payload = reply.getPayload();
  // and demarshall the returned value
  T returnValue = null;
  if (typeOfReturnValue != null)
    returnValue = gson.fromJson(payload, typeOfReturnValue);
  return returnValue;
}
```

```
public class RequestObject {
  private final String operationName;
  private final String payload;
  private final String objectId;
```

AARHUS UNIVERSITET

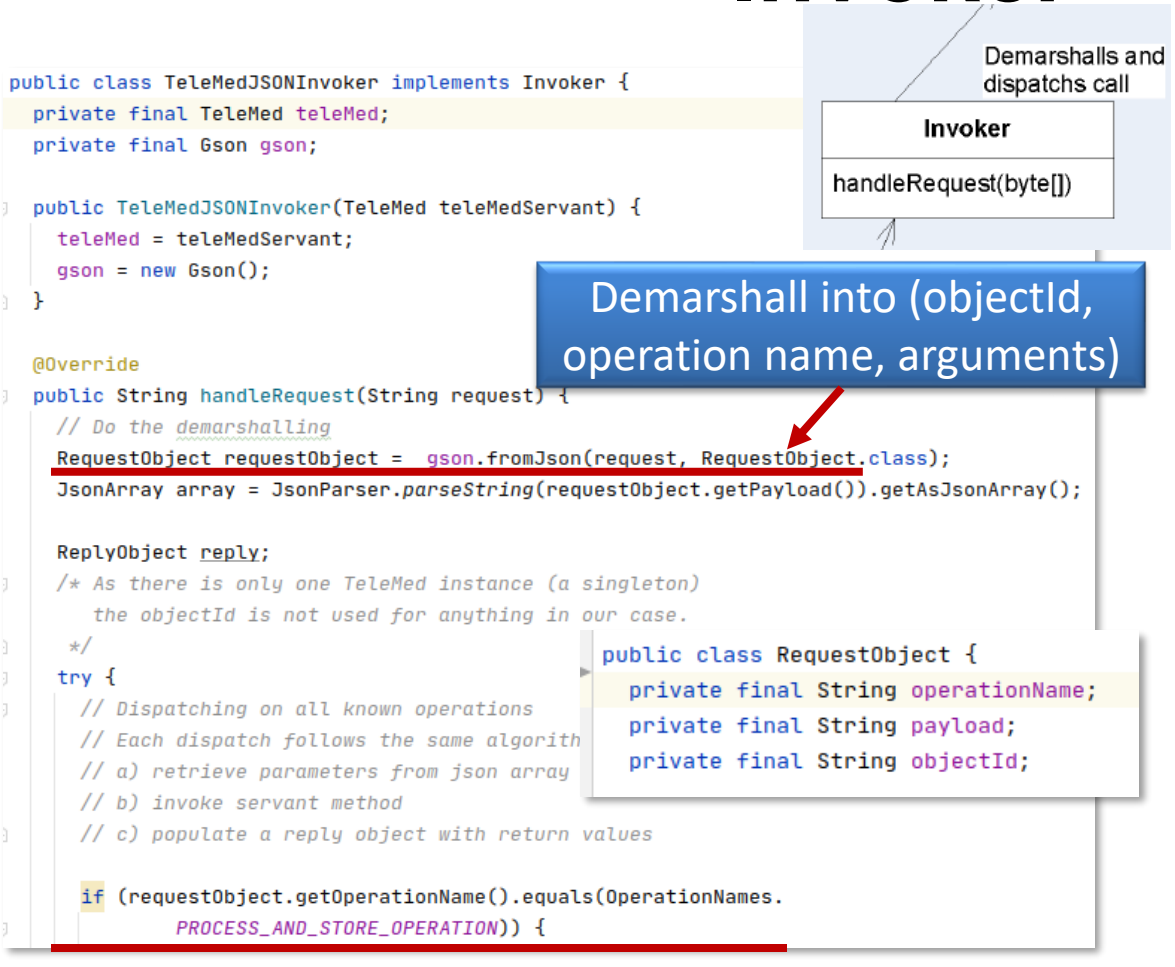
Request Handlers

- Let us skip them for the moment…
- Basically, they are responsible for the request/reply protocol
- Broker Library code base come with two variants:
  - Socket: Raw Java TCP/IP network implementations
  - HTTP: Use as a raw transport (URI Tunneling)

Invoker

Basically, you need a *large switch* on each method name to do the 'upcall', and extract the relevant parameters for the method

For multi-object system, you need something more complex. Stay tuned – we will look at it next week...



AARHUS UNIVERSITET

Invoker

- Once the method is determined, parameter list can be demarshalled, and the *upcall* made…

```
if (requestObject.getOperationName().equals(OperationNames.
        PROCESS_AND_STORE_OPERATION)) {
  // Parameter convention: [0] = TeleObservation
  TeleObservation ts = gson.fromJson(array.get(0),
          TeleObservation.class);

  String uid = teleMed.processAndStore(ts);
  reply = new ReplyObject(HttpServletResponse.SC_CREATED,
          gson.toJson(uid));
```

```
} else if (requestObject.getOperationName().equals(OperationNames.
        GET_OBSERVATIONS_FOR_OPERATION)) {
  // Parameter convention: [0] = patientId
  String patientId = gson.fromJson(array.get(0), String.class);
  // Parameter convention: [1] = time interval
  TimeInterval interval = gson.fromJson(array.get(1),
          TimeInterval.class);

  List<TeleObservation> tol =
          teleMed.getObservationsFor(patientId, interval);
  int statusCode =
```

```
// And marshall the reply
return gson.toJson(reply);
```

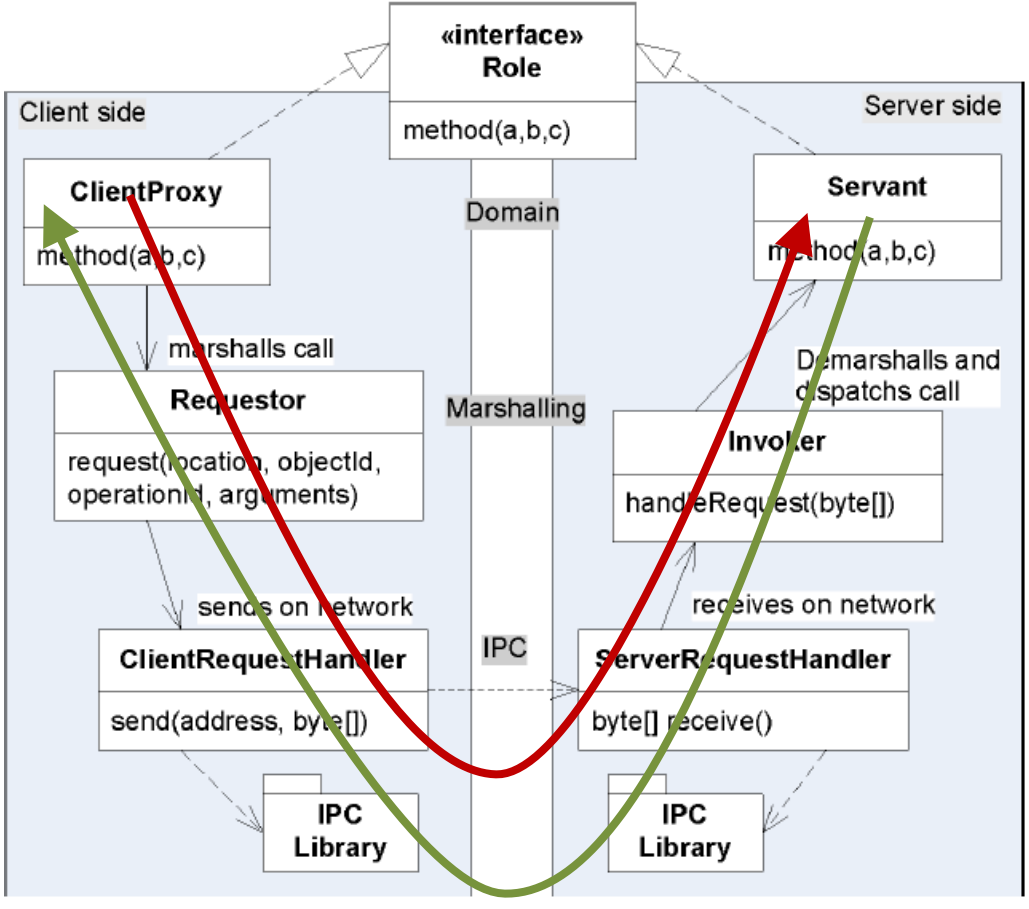
TeleMed Servant

- Servant = Domain implementation
- Not really relevant for Broker, but for the system ☺

```
public class TeleMedServant implements TeleMed, Servant {

  private XDSBackend xds;

  public TeleMedServant(XDSBackend xds) {
    this.xds = xds;
  }

  @Override
  public String processAndStore(TeleObservation teleObs) {
    // Generate the XML document representing the
    // observation in HL7 (HealthLevel7) format.
    HL7Builder builder = new HL7Builder();
    Director.construct(teleObs, builder);
    Document hl7Document = builder.getResult();

    // Generate the metadata for the observation
    MetadataBuilder metaDataBuilder = new MetadataBuilder();
    Director.construct(teleObs, metaDataBuilder);
    MetaData metadata = metaDataBuilder.getResult();

    // Finally store the document in the XDS storage system
    String uniqueId = null;
    uniqueId = xds.provideAndRegisterDocument(metadata, hl7Document);

    return uniqueId;
  }
```

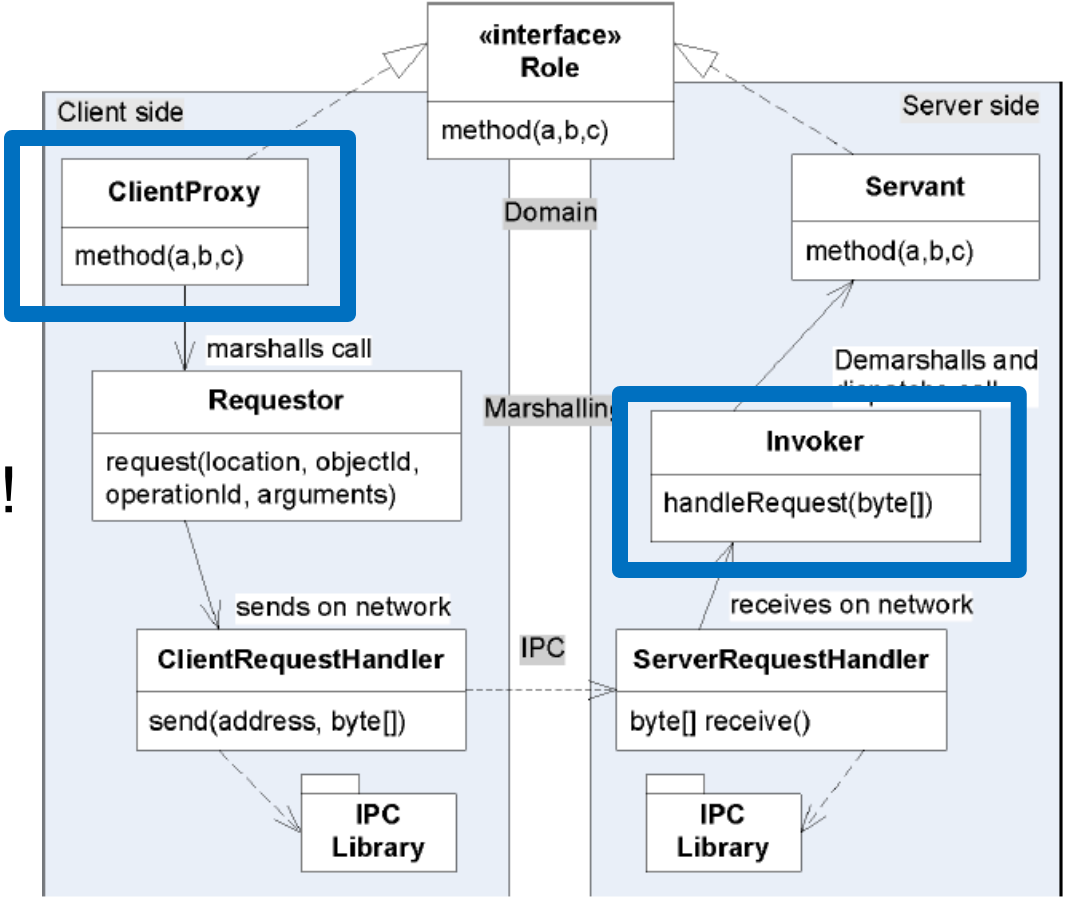
Summary

- The flow



Summary

- It is a Framework !
- Only roles
  - ClientProxy
  - Invoker
- … are TeleMed specific!
- (… and HotStone specific!)



Limitations

- No Name Service / Registry required for TeleMed
  - Parameterized which machine the servant object resides on
    - Use DNS as kind of registry, defaults to 'localhost'
  - More RPC than RMI
    - Remote Procedure Call on 'single type object', not on multiple objects
- Only Value types can be passed, not Reference types
  - No object references ever pass **from client to server!**
- Asymmetric
  - Client-server protocol, no 'callback' **from** server possible
  - I.e. The Observer pattern **can not** be implemented

We treat String as pass-by-value

AARHUS UNIVERSITET

Why No Call Backs to Clients?

- Because *server calling clients is **BAD ☺!***

- No no no. Nothing is every ‘good’ or ‘bad’ in science ☺

- We will return to why ‘servers should not call clients’ in next week…

AARHUS UNIVERSITET

Deployment

- In the code bases distributed, the client and server side classes are pooled into one big source tree
  - src/

Simply easier in our teaching

- In *real* deployments you need to split'em
  - Server: Server side specific classes
  - Core: Core domain *interfaces* and *PODOs*
  - Client: Client side specific classes

- The client side deployment (Core + Client)
- The server side deployment (Core + Server)

AARHUS UNIVERSITET

The Process?

How did I get there?

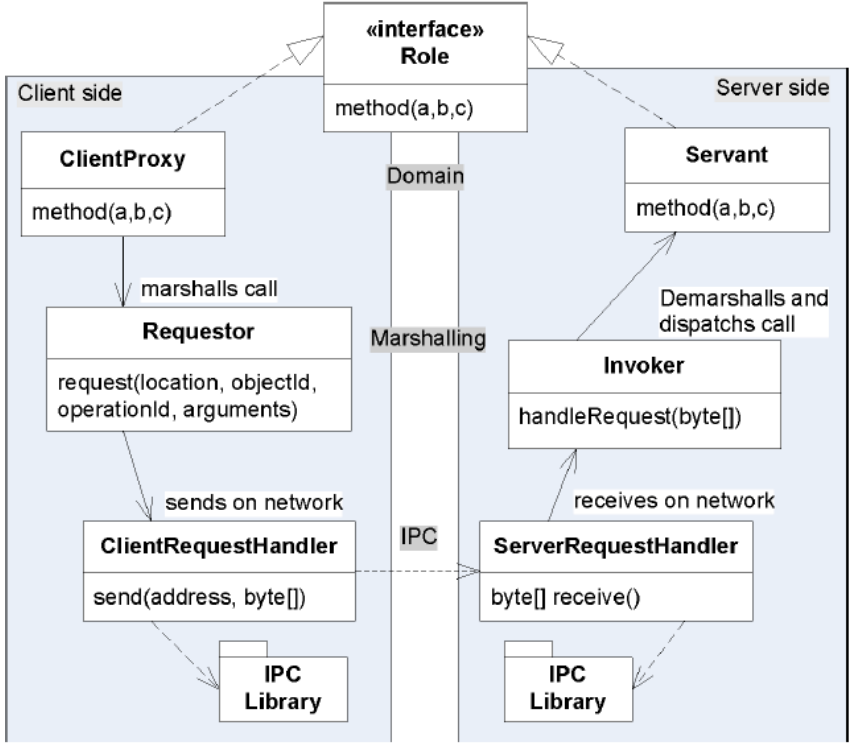
Developing it

- All well – you see the *final picture* but how was it painted?

- Challenge: **TDD of a distributed system?**
  - I cannot (easily) automate that a server needs to be running on some remote machine, can I?
    - (Well we can, but that is another course…)

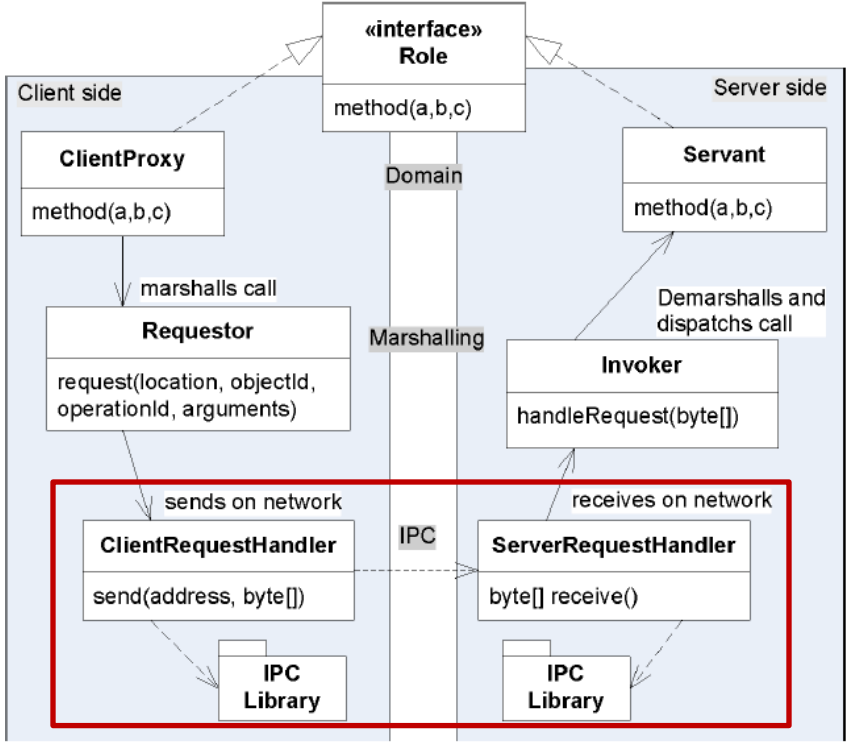
Exercise

- Which level hinders TDD???
  - Or rather *automated testing*
- And **you know** how to deal with it, right?!?
- **What is the answer???**



Principle 1+2+Doubles

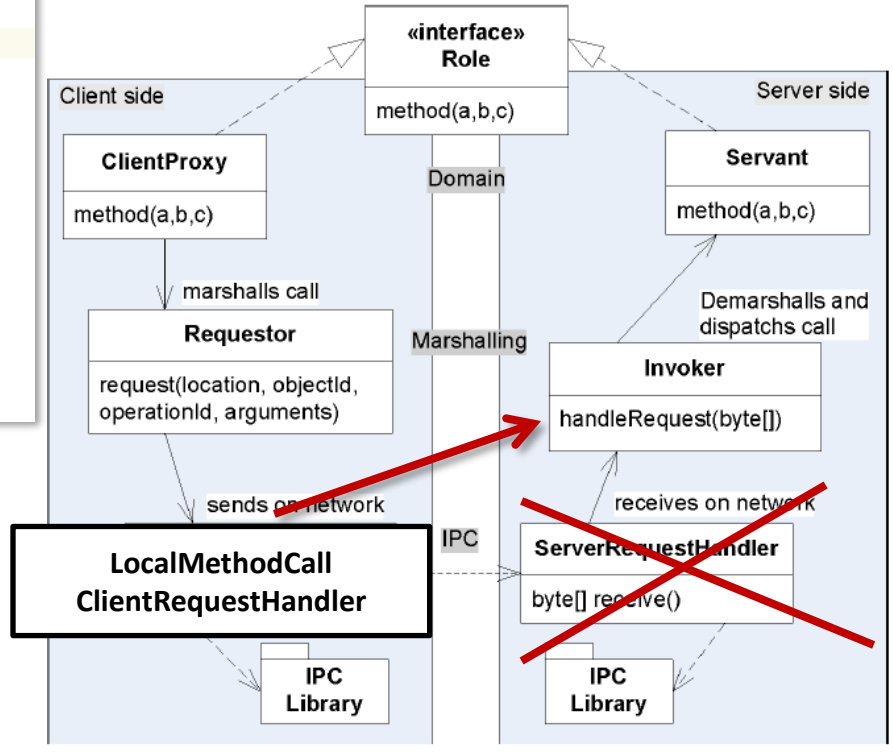
- *It is the IPC Level that hinders TDD*
- *But*
  - *Programmed to an interface*
  - *Object compose a Test Double into place instead!!!*
    - *A Fake Object IPC*



AARHUS UNIVERSITET

Faking the IPC

```
public class LocalMethodCallClientRequestHandler implements ClientRequestHandler {

  private final Invoker invoker;
  private String lastRequest; private String lastReply;

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

  @Override
  public String sendToServerAndAwaitReply(String request) {
    lastRequest = request;
    String reply = invoker.handleRequest(request);
    lastReply = reply;
    return reply;
  }
```



No need to start server.
No concurrency.

All aspects (except IPC) can be TDD'ed

AARHUS UNIVERSITET

@BeforeEach

- Binding the Broker / Coupling the delegates together

```
@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);
}
```

The *only* test double!
The rest are production code!

- That is
  - Link proxy to requestor, requestor to CRH double, CRH to invoker, and the Invoker to the servant object

AARHUS UNIVERSITET

Fun Fact

- Nancy?
  - A fictive person which exists in all Danish medical systems

2512489996	25-12-1948	K	Berggren	Nancy Ann Test	Testpark Allé 48	3400	219	84	Sønnerne Max og Ruddi. Døtrene Kirsten og Britta.

- She even has a face book profile ☺

https://da-dk.facebook.com › nancy.a.berggren

Nancy Ann Berggren | Facebook

Nancy Ann **Berggren** er på Facebook. Bliv medlem af Facebook, og få kontakt med **Nancy** Ann **Berggren** og andre, du måske kender. ... **MedCom**, profile picture.

AARHUS UNIVERSITET

Make a Test Case

- Call client proxy, assert something stored in XDS

```
@Test
public void shouldStoreFromClient() {
  // Nancy uploads a single observation
  teleMed.processAndStore(teleObs1);

  // And the proper HL7 document is stored in the backend XDS
  Document stored = xds.getLastStoredObservation();
  HelperMethods.assertThatDocumentRepresentsObservation120over70forNancy(stored);
}
```

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

'xds' is both Spy and FakeObject

AARHUS UNIVERSITET

The IPC Level

Talking network'ish

Choosing IPC

- The most fundamental level
  - Sockets

- More modern approach
  - URI Tunneling using HTTP web servers

Rule #1

- Find stuff on the internet ☺
  - Jakob Jenkov has fine tutorials on socket server programming
    - Single thread
    - Multi thread
    - Thread pooled

- Question of concurrency
  - Single thread — Only one call at the time
  - Multi thread — Unlimited => Memory exhausted!
  - Thread pool — N threads = Best of both worlds

AARHUS UNIVERSITET

Client Request Handler

- Socket
  - “modificed EchoClient"
- The old HTTP protocol
  - Create socket
  - Send request
  - Read reply
  - Close socket
- Inefficient but reliable

```
@Override
public String sendToServerAndAwaitReply(String request) {
  Socket clientSocket = null;

  // Create the socket connection to the host
  PrintWriter out;
  BufferedReader in;
  try {
    clientSocket = new Socket(hostname, port);
    out = new PrintWriter(clientSocket.getOutputStream(), true);
    in = new BufferedReader(new InputStreamReader(
           clientSocket.getInputStream()));
  } catch (IOException e ) {
    throw new IPCException("Socket creation problems", e);
  }

  // Send it to the server (= write it to the socket stream)
  out.println(request);

  // Block until a reply is received
  String reply;
  try {
    reply = in.readLine();

  } catch (IOException e) {
    throw new IPCException("Socket read problems", e);
  } finally {
    try {
      clientSocket.close();
    } catch (IOException e) {
      throw new IPCException("Socket close problems (1)", e);
    }
  }
  // ... and close the connection
  try {
    clientSocket.close();
  } catch (IOException e) {
    throw new IPCException("Socket close problems (2)", e);
  }

  return reply;
}
```

AARHUS UNIVERSITET

Server Request Handler

- Jenkov single thread
  - Accept incoming socket
  - Read request
  - Call invoker
  - Send reply
  - Close socket

```
@Override
public void run() {
  openServerSocket();

  System.out.println("*** Server socket established ***");

  isStopped = false;
  while (!isStopped) {

    System.out.println("--> Accepting...");
    Socket clientSocket;
    try {
      clientSocket = serverSocket.accept();
    } catch(IOException e) {
      if(isStopped) {
        System.out.println("Server Stopped.") ;
        return;
      }
      throw new RuntimeException(
          "Error accepting client connection", e);
    }

    try {
      readMessageAndDispatch(clientSocket);
    } catch (IOException e) {
      System.out.println("ERROR: IOException encountered: "
              + e.getMessage());
    }
  }
  System.out.println("Server Stopped.");
}
```

```
private void readMessageAndDispatch(Socket clientSocket)
        throws IOException {
  PrintWriter out =
          new PrintWriter(clientSocket.getOutputStream(), true);
  BufferedReader in = new BufferedReader(new InputStreamReader(
      clientSocket.getInputStream()));

  String inputLine;
  String marshalledReply = null;

  inputLine = in.readLine();
  System.out.println("--> Received " + inputLine);
  if (inputLine == null) {
    System.err.println(
            "Server read a null string from the socket???");
  } else {
    marshalledReply = invoker.handleRequest(inputLine);

    System.out.println("--< replied: " + marshalledReply);
  }
  out.println(marshalledReply);

  System.out.println("Closing socket...");
  in.close();
  out.close();
}
```

AARHUS UNIVERSITET

Summary