5. Broker Part Two

5.1 Learning Objectives

In this chapter, I will dig deeper into the Broker pattern and show how to handle distributed systems that have more than a single type that needs to make remote method calls. Supporting many different types each with many methods each will force the server side invoker to handle a lot of different method up calls. Thus a secondary learning objective is to show a simple way to avoid the invoker code to become incohesive.

5.2 Limitations in the TeleMed Case

While the TeleMed case is a realistic one, it is also quite simple with regards to design and implementation. It does not showcase techniques for handling some of the more complex challenges when using the *Broker* architecture. It is simple because:

- Only one type, and indeed only one object, is handled in our TeleMed case: The single `TeleMed` instance. While it captures systems like web based shopping baskets, we also have to look at systems that deal with many different types (classes) and many instances of each type.
- The central object identity used in TeleMed is the person's social security number and thus given by the medical domain. However, in many computing systems, this is not the case - the server *creates* many objects of a given type, and the client must then be able to identify exactly which object it needs to call a method on.

So, I will present a new case below, which allows me to discuss designs and implementations to deal with multi type, multi object, systems. Beware that I have on purpose *over-engineered* the architecture a bit in order to showcase more types and objects for the sake of demonstration.

5.3 Game Lobby Stories

The context is **GameLobby** - a system to match two players, in different locations, that want to play a computer game together over the internet.

To do that, one player creates a game on a server computer, and invites the second player to join it. Once the second player has joined, the remote game becomes available for them to play.

Rephrasing this as user stories

Story 1: Creating a remote game. Player *Pedersen* has talked with his friend *Findus* about playing a computer game together; they both sit in their respective homes, so it must be a remote game, played over the internet. They agree that Pedersen should create the game, and Findus should then join it. Pedersen opens a web browser and opens the game's *game lobby page*. On this lobby page, he hits the button to *create game*. The web page then states that the game has been created, and displays the game's *join token*, which is simply a unique string, like "game-17453". It also displays a *play game* button but it is inactive to indicate that no other player has joined the game yet. Pedersen then tells Findus the game's join token. Next, he awaits that Findus joins the game.

Story 2: Joining an existing game. Meanwhile *Findus* has entered the same game lobby page. Once he is told the join token, "game-17453", from Pedersen, he hits the *join game* button, and enters the join token string. The web page displays that the game has been created, and he hits the *play game* button, that brings him to the actual game.

Story 3: Playing the game. Pedersen has waited for Findus to join the game. Now that he has, the *play game* button becomes active, and he can hit it to start playing the game with Findus.

This is a more complex scenario, as there are several roles involved, like the game lobby, the game to be played, and also the intermediate state of a 'game that has been created but is not playable yet.'

A Role Based Design

I design the game lobby domain using three basic roles: I need an object to represent the **GameLobby**, which is responsible for allowing players to create and join games. One player must create a game, while another may join it. When a user create a game, he/she will be given a **FutureGame** object, that is a role that represents "the game to be" in the near future. The FutureGame has a method `isAvailable()` that is false until the second player has joined the game. Once he/she has done that, the FutureGame can be queried using `getGame()` to get an object of the final role **Game**. Game of course represents the actual game to be played with game specific methods for movement, game state accessors, or whatever is relevant.

A *Future* is a well-known software engineering concept which represents the answer to a request which may take a long time to compute - instead of waiting for the answer, the client receives a *Future* immediately. The future can then be polled to see if the answer has become available, and once it is, the answer can be retrieved.

The purpose of the Future is to avoid that the client blocks for a long time while waiting for the answer to be computed.

GameLobby

- Singleton object, representing the entry point for creating and joining games.

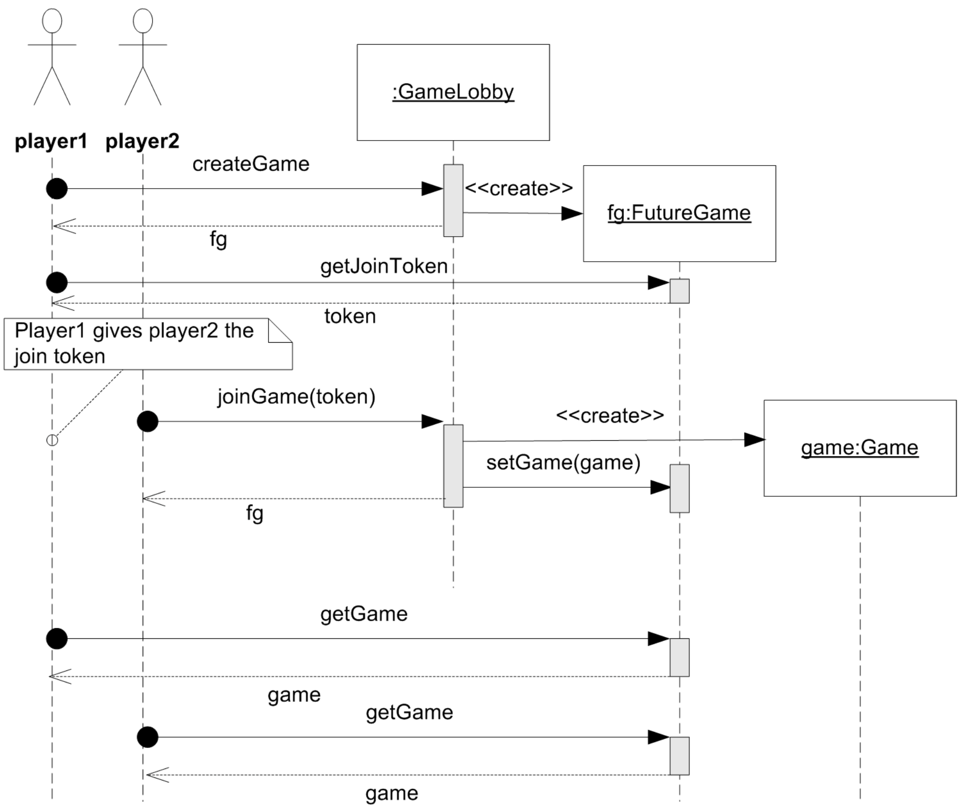
FutureGame

- A Future, allowing the state of the game (available or not) to be queried, and once both players have joined, return the game object itself.
- Provides an accessor method `getJoinToken()` to retrieve the join token that the second user must provide.

Game

- The actual game domain role.

So a typical execution of the stories of the scenario with Pedersen and Findus can be visualized using this UML sequence diagram:



Game lobby dynamics.

I have encoded this sequence diagram as a JUnit test case below. Note that the test case act both as Pedersen and Findus (player1 and player2).

(Fragment from *gamelobby* package, in `TestScenario.java`):

```java
// Lobby object is made in the setup/before method
FutureGame player1Future = lobbyProxy.createGame("Pedersen", 0);
assertThat(player1Future, is(not(nullValue())));

// Get the token for my fellow players to enter when wanting
// to join my game. The token may appear on a web site
// next to player 1's name so player 2 can see it; or
// some other mechanism must be employed by the two players
// for player 2 to get hold of the token.
String joinToken = player1Future.getJoinToken();
assertThat(joinToken, is(not(nullValue())));

// As a second player has not yet joined, the game
// is not yet created
assertThat(player1Future.isAvailable(), is(false));

// Second player - wants to join the game using the token
FutureGame player2Future = lobby.joinGame("Findus", joinToken);
assertThat(player2Future, is(not(nullValue())));
```

```

// Now, as it is a two player game, both players see
// that the game has become available.
assertThat(player1Future.isAvailable(), is(true));
assertThat(player2Future.isAvailable(), is(true));

// And they can make state changes and read game state to the game
Game gameForPlayer1 = player1Future.getGame();
assertThat(gameForPlayer1.getPlayerName(0), is("Pedersen"));
assertThat(gameForPlayer1.getPlayerName(1), is("Findus"));
assertThat(gameForPlayer1.getPlayerInTurn(), is("Pedersen"));

// Our second player sees the same game state
Game gameForPlayer2 = player2Future.getGame();
assertThat(gameForPlayer2.getPlayerName(0), is("Pedersen"));
assertThat(gameForPlayer2.getPlayerName(1), is("Findus"));
assertThat(gameForPlayer2.getPlayerInTurn(), is("Pedersen"));

// Make a state change, player one makes a move
gameForPlayer1.move();

// And verify turn is now the opposite player
assertThat(gameForPlayer1.getPlayerInTurn(), is("Findus"));
assertThat(gameForPlayer2.getPlayerInTurn(), is("Findus"));
```

Challenges

What are the challenges in the above design which is not already covered by the previous discussion of Broker?

Well, let us consider what our Broker *does provide solutions* for already.

- All three roles (lobby, future, game) must be defined as Java interfaces, and we must implement Servant and Proxy code.
- The Invoker on the server side must switch on quite a few operation names. As outlined earlier, a good operation name includes both the type and the method name, so, say, `gamelobby_create_game` may represent the `createGame()` method for the `GameLobby` interface. Of course, our invoker code easily becomes pretty long and tedious to write, but it is doable.
- Marshalling and demarshalling does not pose any problems we have not already dealt with.
- The same goes with IPC, nothing new under the sun here.

But, the main challenge is that the server side creates new objects all the time, like the outcome of `createGame()` that returns a FutureGame instance.

Our Broker cannot “pass-by-reference” and the FutureGame is of course an object reference, so our current Broker pattern is seemingly stuck. This is *issue 1*.

Another, and unrelated, issue is the cluttering in the Invoker - consider having 20 different types with 20 methods each, this would require a switch on operation name having 400 different cases. This is hardly manageable. This is *issue 2*.

5.4 Walkthrough of a Solution

Instead of just providing solutions to the two issues, I will outline aspects of the development process to make this work, and then provide an abstract process description in the conclusion. You can find a detailed development diary in the *diary.md* file in the *gamelobby* package in the source code.

Issue 1: Server Created Objects

Let us focus first on *issue 1: server created objects*, and let us start by looking at what happens in the client code. I will take my starting point at the following code, which exposes the issue:

```
// Client side code
FutureGame player1Future = lobbyProxy.createGame("Pedersen", 0);
```

Thus, the `createGame()` method in the **ClientProxy** of *GameLobby* must invoke the servant’s identical method, and return a FutureGame instance to the caller. As the returned FutureGame instance is on the server side, the actual return type on the client must of course be a proxy, *FutureGameProxy*, which is associated with that particular instance of *FutureGameServant*.

Take a moment to consider how the two objects are associated before you read on.

How are proxies associated correctly with their servant objects? **Through the ‘objectId’.** So the key insight is that while the server cannot pass-by-reference a new FutureGame instance to the client, it can pass the objectId, as they are simple values, typically strings. Once the client proxy receives the objectId, it can then instantiate a FutureGameProxy and tell it which objectId it should use.

So the GameLobbyProxy’s implementation becomes

```java
// Client side GameLobbyProxy code.
@Override
public FutureGame createGame(String playerName, int playerLevel) {
  String id =
    requestor.sendRequestAndAwaitReply(GAMELOBBY_OBJECTID,
            MarshallingConstant.GAMELOBBY_CREATE_GAME_METHOD,
            String.class, playerName, playerLevel);
  FutureGame proxy = new FutureGameProxy(id, requestor);
  return proxy;
}
```

That is, the call chain will end up on the server's Invoker implementation which then also must do something a little different than normal, because the up-call to the GameLobbyServant's `createGame()` method will return a FutureGame instance, and not a objectId string. So this is the next issue I have to tackle.

The first part of the Invoker's code to handle the `createGame()` method looks normal: demarshall the parameters and do the up-call to the servant ('lobby' in the code below):

```java
// Server side Invoker code.
if (operationName.equals(MarshallingConstant.GAMELOBBY_CREATE_GAME_METHOD)) {
  String playerName = gson.fromJson(array.get(0), String.class);
  int level = gson.fromJson(array.get(1), Integer.class);
  FutureGame futureGame = lobby.createGame(playerName, level);
```

Now I have the FutureGame instance, but what I need to return is the objectId, and I have none. So, the question is how to create that? Actually, I can do this in a number of ways. I will discuss alternatives later in the discussion section, but for now, I will solve it by adding a responsibility to the FutureGameServant, namely that it creates and maintains a unique ID; and by adding a responsiblity to the FutureGame interface, namely to have an accessor to this objectId: a `getId()` method:

```java
public interface FutureGame {
  [...]
  /** Get the unique id of this game.
   *
   * @return id of this game instance.
   */
  String getId();
}
```

Then I make the servant constructor assign an objectId:

```java
// Server side Servant code.
public class FutureGameServant implements FutureGame, Servant {
  private String id;

  public FutureGameServant(String playerName, int playerLevel) {
    // Create the object ID to bind server and client side
    // Servant-ClientProxy objects together
    id = UUID.randomUUID().toString();
    [...]
  }

  @Override
  public String getId() {
    return id;
  }
  [...]
}
```

I here use the Java library UUID which can create universally unique ID's. The `getId()` method of course just returns the 'id' generated once and for all in the constructor. As always, the `[...]` represents the "rest of the code."

Why add the `getId()` to the interface, thus forcing both proxy and servant to implement it? Well, the proxy also need to know the objectId of its associated servant object, so it makes sense to expose it through an accessor method on both the server and client side code.

So, now any newly created FutureGameServant object will generate a unique id, and I can complete the Invoker code by marshalling this and return it back to the client proxy, as is shown in line 7-8:

```java
 1  // Server side Invoker code.
 2  if (operationName.equals(MarshallingConstant.GAMELOBBY_CREATE_GAME_METHOD)) {
 3    String playerName = gson.fromJson(array.get(0), String.class);
 4    int level = gson.fromJson(array.get(1), Integer.class);
 5    FutureGame futureGame = lobby.createGame(playerName, level);
 6    String id = futureGame.getId();
 7
 8    reply = new ReplyObject(HttpServletResponse.SC_CREATED,
 9            gson.toJson(id));
10  } [...]
```

One very important thing is missing though, namely how the returned objectId is used by the proxy. So, consider the next line of our test case, the in which the join token is fetched:

```
1  // Testing code.
2  FutureGame player1Future = lobbyProxy.createGame("Pedersen", 0);
3  String joinToken = player1Future.getJoinToken();
```

The FutureGame proxy code is the normal template implementation:

```
1  // Client side ClientProxy code.
2  @Override
3  public String getJoinToken() {
4    String token = requestor.sendRequestAndAwaitReply(getId(),
5            MarshallingConstant.FUTUREGAME_GET_JOIN_TOKEN_METHOD,
6            String.class);
7    return token;
8  }
```

However, the Invoker will receive the operationName and the objectId, but

```
1  // Server side Invoker code.
2  if (operationName.equals(MarshallingConstant.FUTUREGAME_GET_JOIN_TOKEN_METHOD)) {
3    FutureGame futureGame = ???
4    String token = futureGame.getJoinToken();
5    reply = new ReplyObject(HttpServletResponse.SC_OK, gson.toJson(token));
6  }
```

The culprit is line 3. How does the Invoker know which FutureGame object to do the up-call on?

The answer is that we need a *name service*. Remember name services are dictonaries that map remote object identities to the real remote objects. Here, I go for a simple name service implementation, and add another line, line 7, to the GameLobby's invoker code for the `createGame()` method:

```
1  // Server side Invoker code.
2  if (operationName.equals(MarshallingConstant.GAMELOBBY_CREATE_GAME_METHOD)) {
3    String playerName = gson.fromJson(array.get(0), String.class);
4    int level = gson.fromJson(array.get(1), Integer.class);
5    FutureGame futureGame = lobby.createGame(playerName, level);
6    String id = futureGame.getId();
7    nameService.putFutureGame(id, futureGame);
8
9    reply = new ReplyObject(HttpServletResponse.SC_CREATED,
10           gson.toJson(id));
11 } [...]
```

'nameService' is a new abstraction that I introduce which is essentially a Map data structure with `put()` and `get()` methods that maps unique id (objectId) to a servant object.

```java
public interface NameService {
  void putFutureGame(String objectId, FutureGame futureGame);
  FutureGame getFutureGame(String objectId);

  [...]
}
```

Every time, the server side creates an object, it must remember to add the newly created object reference into the name service under its unique object id.

Now, my FutureGame invoker code for the `getJoinToken()` method can simply look up the proper instance, based upon the object id, and next do the up-call:

```java
// Server side Invoker code.
if (operationName.equals(MarshallingConstant.FUTUREGAME_GET_JOIN_TOKEN_METHOD)) {
  FutureGame futureGame = nameService.getFutureGame(objectId);
  String token = futureGame.getJoinToken();
  reply = new ReplyObject(HttpServletResponse.SC_OK, gson.toJson(token));

} else if ( operationName.equals( [...]
```

The same process is of course repeated when creating a Game object: Add an accessor method `getId()` to the Game interface, let the GameServant constructor generate a unique ID and assign it to an instance variable; and return the objectId back to the ClientProxy which creates a GameProxy with the given objectId.

Object reference accessor methods

The discussion above pointed out the process of passing object references for objects created by the server.

The same process also applies for accessor methods that return object references.

Our GameLobby case study already has such a method in FutureGame:

```java
public interface FutureGame {
  [... ]

  Game getGame();
}
```

The magic lives in the Invoker code. The insight is that the returned game instance already has an objectId, so it is just that, which must be returned to the client side proxy:

```java
// Server side Invoker code.
[...]
} else if (operationName.equals(
    MarshallingConstant.FUTUREGAME_GET_GAME_METHOD)) {
  FutureGame futureGame = nameService.getFutureGame(objectId);
  Game game = futureGame.getGame();
  String id = game.getId();
  reply = new ReplyObject(HttpServletResponse.SC_OK, gson.toJson(id));
}
```

Here, the Invoker fetches the future game servant, makes the upcall, and then just marshalls the returned object's (a game instance) objectId back to the client side.

Discussion

The presented solution now allows our **Broker** to handle both pass-by-value and pass-by-reference, however the latter only when method references are references to *objects on the server*. As outlined earlier, a local object residing on the client side can never be pass-by-reference to the server side.

Argue why we cannot pass a client object reference to the server.

If you have a method in which a parameter is a server side object, ala this one:

```java
Game game = futureGame.getGame();
lobbyProxy.tellIWantToLeave(game);
```

Then your proxy code of course shall just send the objectId to the server. This will allow the server side invoker to lookup the proper server object, and pass that to the equivalent `tellIWantToLeave()` method of the servant object.

Note also that in a remote system you have to decide for each class if they are pass-by-value or pass-by-reference. Going back to the TeleMed case - there a TeleObservation instance was actually part of the method signature of `processAndStore()` method. However, as it was record type class, only storing dumb values, it made perfectly sense to treat it as a pass-by-value object. The same goes for the String class.

I have arguably added a responsibility to my domain roles, **Game** and **FutureGame**, by adding the `getId()` method to their interfaces, and thus added a responsiblility to create and handle (remote) object identities. One may argue, that remote aspects thereby sneak into the domain abstractions which is an aspect that does not belong there. However, the premise for this argument is basically that domain object should not be aware that they are working in a distributed environment, and from a software architectural viewpoint, this is simply an incorrect premise. Remote access to an object has profound implications to the architecture of a system, as quality attributes like performance, security and availability have to be analyzed and handled, as argued in Chapter Basic Concepts. The 'getId()' is thus just one small hint of that.

The responsibility to assign the unique ID must be given to some role in the system. I gave it to the domain role itself in the discussion above. However, there are other options.

- Often the domain itself has a notion of objectId, typically through some 'catalog', 'invoicing', 'inventory', or 'orderService' role. For instance, invoices often have a numerical sequence number assigned when created, which can serve as the 'objectId'. Or objects are stored and handled in a database, and as the database will maintain a unique id for any tuple or document (often a primary key), it is obvious just to reuse that, or derive an id from it.
- As the Invoker role is the one that calls any `create()` method, one may let the Invoker handle unique objectId creation and maintenance itself, and thus remove this responsibility completely from the servants.

Rewrite the *GameLobby* system in such a way, that the assignment and maintenance of unique object identities for created servant objects is the responsibility of the **Invoker**. That is, avoid introducing the `getId()` methods in the **FutureGame** and **Game** interfaces.

The **Name Service** implementation in my GameLobby systems is simple and have some limitations. I use an in-memory Map based data structure and thus it will not survive a server restart. This is of course not feasible for a large business system. Another implication is that our system relies on a single server, a *single point of failure*. "If that single server fails, the system fails." This is of course also not feasible for a large system. Tried and tested solutions exist for both issues, in the form of *cache servers* and *load balancing*, so these are not intrinsic liabilities for the **Broker** pattern. However, they are outside the scope of this book.

A final point worth discussing is what to return to the client to represent the servant object created on the server side. I used simple string typed object ids, but if the role contains many fixed valued attributes, you should consider using a **Data Transfer Object (DTO)** as alternative. For instance, our **Game** have accessors to get the names of the two players through the `getPlayerName(int index)` method, but as the names of the players are highly unlikely to change, it makes sense to transfer them as part of the `getGame()` payload. So one suggestion for a DTO is

```java
public class GameDTO {
  public String objectId;
  public String player0Name;
  public String player1Name;
}
```

Thus, the **Invoker**'s code, that currently only transmits the object id:

```java
} else if (operationName.equals(MarshallingConstant.FUTUREGAME_GET_GAME_METHOD)) {
  FutureGame futureGame = nameService.getFutureGame(objectId);
  Game game = futureGame.getGame();
  String id = game.getId();
  reply = new ReplyObject(HttpServletResponse.SC_OK, gson.toJson(id));
}
```

is then rewritten into returning a DTO instead:

```java
} else if (operationName.equals(MarshallingConstant.FUTUREGAME_GET_GAME_METHOD)) {
  FutureGame futureGame = nameService.getFutureGame(objectId);
  Game game = futureGame.getGame();
  GameDTO dto = new GameDTO();
  dto.objectId = game.getId();
  dto.player0Name = game.getPlayerName(0);
  dto.player1Name = game.getPlayerName(1);
  reply = new ReplyObject(HttpServletResponse.SC_OK, gson.toJson(dto));
}
```

This way, the receiving Requestor can create a more complete **GameProxy** that already stores the player names, and the proxy's `getPlayerName()` method can then *avoid* the expensive server call, and simply return the local values. This caching-in-the-proxy trick is essential to improve performance in a distributed system, and another benefit of partially coding the *Broker* pattern yourself instead of relying on e.g. Java RMI.

A DTO is actually similar to a *resource* in REST terminology which I will dicuss in the REST Chapter.

Issue 2: Invoker Cluttering

The three types in our game lobby system have eight methods in total that need a remote implementation using our Broker. While having a switch in the invoker with eight branches it not much, it still highlights the issue: as more and more types must be supported with more and more methods, the invoker's `handleRequest()` method will just become longer and longer, lowering analyzability and maintainability. Basically, cohesion is low because the up calls for all types are merged together into a single invoker implementation.

As an example, the HotCiv project from *Flexible, Reliable Software*, has several types: **Game**, **Unit**, **City**, and **Tile**, that all have quite a long list of methods, and thus you quickly end up adding comments as kind of section marker in code that goes on page after page, like:

```
// === GAME
if (operationName.equals(MarshallingConstants.GAME_GET_PLAYER_IN_TURN)) {
  ...
} else if (operationName.equals(MarshallingConstants.GAME_END_OF_TURN)) {

  [lots of more if clauses removed]

  // === UNIT
} else if (operationName.equals(MarshallingConstants.UNIT_GET_OWNER)) {

  [lots of more if clauses removed]

  // === CITY
} else if (operationName.equals(MarshallingConstants.CITY_GET_OWNER)) {
  ...
```

In general, comments have their purpose, but these kind of section markers highlight low analyzability of the code - so let us refactor our code into smaller, more cohesive units.

As the invoker code became cluttered because it handles all methods for all types, the solution is straight forward: Split it into a set of smaller and more cohesive roles - one for each type. Thus I split it into **GameInvoker**, a **FutureGameInvoker**, and a **GameLobbyInvoker**, each of which only handles the switch on `operationName` and associated servant up call for that particular type. I still need a single entry point, though, the one `handleRequest()` that is called from the **ServerRequestHandler**, which then must determine which of the three types the method call is for. This is a kind of root in the system, so I have called this class **GameLobbyRootInvoker**.

Thus the root `handleRequest` will look something like

```java
@Override
public String handleRequest(String request) {
  RequestObject requestObject = gson.fromJson(request, RequestObject.class);
  String operationName = requestObject.getOperationName();

  String reply;

  // Identify the invoker to use
  Invoker subInvoker = [find invoker for Game or FutureGame or Lobby]

  // And do the upcall on the subInvoker
  try {
    reply = subInvoker.handleRequest(request);

  } catch (UnknownServantException e) {
    reply = gson.toJson(
            new ReplyObject(
                    HttpServletResponse.SC_NOT_FOUND,
                    e.getMessage()));
  }

  return reply;
}
```

This is a simple compositional design: the root invoker decides which of the three "sub invoker" to delegate the call to, each of which themselves implement the **Invoker** interface. It is actually a *State* pattern: based upon the the incoming call the invoker changes state to a "handle Game calls", "handle FutureGame calls", etc., and the request is delegated to the relevant state object. The **GameLobbyInvoker** sub invoker handles all methods for **GameLobby** etc., making each of these classes smaller and cohesive.

The final piece is the algorithm is to select which of the invokers to use. A simple approach that I have selected is to make the operation name string constants follow a fixed template, as shown in the *MarshallingConstant* class

```java
public class MarshallingConstant {

  public static final char SEPARATOR = '_';

  // Type prefixes
  public static final String GAME_LOBBY_PREFIX = "gamelobby";
  public static final String FUTUREGAME_PREFIX = "futuregame";
  public static final String GAME_PREFIX = "game";

  // Method ids for marshalling
  public static final String GAMELOBBY_CREATE_GAME_METHOD =
    GAME_LOBBY_PREFIX + SEPARATOR + "create-game-method";
  public static final String GAMELOBBY_JOIN_GAME_METHOD =
    GAME_LOBBY_PREFIX + SEPARATOR + "join-game-method";

  public static final String FUTUREGAME_GET_JOIN_TOKEN_METHOD =
    FUTUREGAME_PREFIX + SEPARATOR + "get-join-token-method";
  public static final String FUTUREGAME_IS_AVAILABLE_METHOD =
    FUTUREGAME_PREFIX + SEPARATOR + "is-available-method";
  [...]
```

Each method name string begins with the type name, followed by the method name. Then the sub invoker can be found by a simple lookup upon the prefix string:

```
@Override
public String handleRequest(String request) {
  RequestObject requestObject = gson.fromJson(request, RequestObject.class);
  String operationName = requestObject.getOperationName();

  String reply;

  // Identify the invoker to use
  String type = operationName.substring(0,
    operationName.indexOf(MarshallingConstant.SEPARATOR));
  Invoker subInvoker = invokerMap.get(type);
  [...]
```

And the three invokers are then stored in a Map data structure, that maps the prefix string to the actual invoker.

```
public GameLobbyRootInvoker(GameLobby lobby) {
  this.lobby = lobby;
  gson = new Gson();

  nameService = new InMemoryNameService();
  invokerMap = new HashMap<>();

  // Create an invoker for each handled type/class
  // and put them in a map, binding them to the
  // operationName prefixes
  Invoker gameLobbyInvoker = new GameLobbyInvoker(lobby,
    nameService, gson);
  invokerMap.put(MarshallingConstant.GAME_LOBBY_PREFIX,
    gameLobbyInvoker);

  Invoker futureGameInvoker = new FutureGameInvoker(nameService, gson);
  invokerMap.put(MarshallingConstant.FUTUREGAME_PREFIX,
    futureGameInvoker);

  Invoker gameInvoker = new GameInvoker(nameService, gson);
  invokerMap.put(MarshallingConstant.GAME_PREFIX,
    gameInvoker);
}
```

5.5 Summary of Key Concepts

Server Created Objects

The discussion above allows me to express the process handling server created object more abstractly.

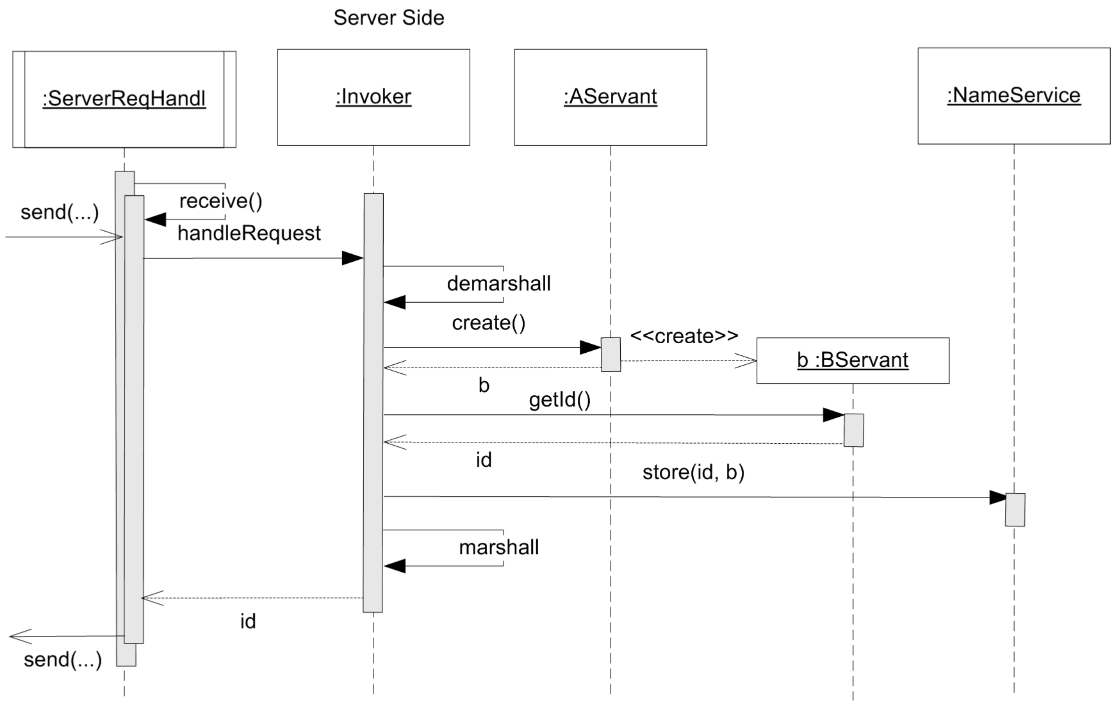
Transferring Server Created Objects

Consider a remote method `ClassB create()` in `ClassA`, that is, a method that creates new instances of ClassB.

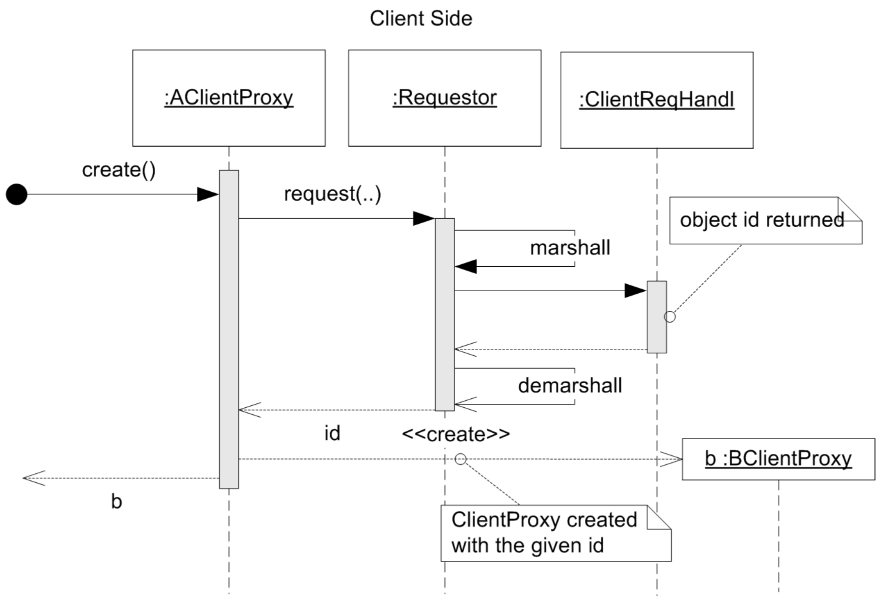
To transfer a reference to an object created on the server side, you must follow this template

- Make the Class B Servant object generate a unique ID upon creation (typically in the constructor using `id = UUID.randomUUID().toString();`, or by the domain/database providing one), and provide an accessor method for it, like `getId()`. Often, it does make sense to include the `getId()` method in the interface, as the ClientProxy object also needs the ID when calling the Requestor.
- Once a servant object is created, it must be stored in a name service using the unique id as key.
- In the Invoker implementation of `ClassA.create()`, use a String as return type marshalling format, and just transfer the unique object id back to the client.
- On the client side, in the ClassAProxy, create a instance of the ClassB-ClientProxy, and store the transferred unique id in the proxy object, and return that to the caller.
- Client code can now communicate with the Class B servant object using the returned client proxy object.
- When the server's Invoker receives a method call on some created object, it must use the provided `objectId` to fetch the servant object from the name service, and call the appropriate method on it.

The UML sequence diagrams below show the server side and the the client side of this algorithm respectively.



Transferring Server Created Objects - Server side



Transferring Server Created Objects - Client side

In the case of just returning object references, a simplified process applies:

Transferring Server Objects

Consider a remote method `ClassB getB()` in `ClassA`, that is, a method that return references to instances of ClassB.

To transfer a reference to an object created on the server side, you must follow this template

- In the Invoker implementation of `ClassA.getB()`, retrieve the objectId of the ClassB instance, and use a String as return type marshalling format, and just transfer the unique object id back to the client.

The processes also introduced yet another responsibility of the invoker, and I therefore have to enhance the role description of **Invoker** and **Requestor**.

Invoker

- Performs demarshalling of incoming byte array.
- Determines servant object, methods, and arguments (some of which may be object IDs in which case the servant reference must be fetched from the **Name Service**), 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.
- When servants create new objects, store their IDs in the **Name Service** and return their ID instead.

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). If returned value is an object ID of a server created object, create a **ClientProxy** with the given object ID and return that.
- Creates client side exceptions in case of failures detected at the server side or during network transmission.

And finally we add the

Name Service

- A server side storage that maps objectId's to objects
- Allows adding, fetching, and deleting entries in the storage

Thus, the Broker UML diagram in the Broker Part One Chapter have to include a **Name Service** role and an association between the **Invoker** and the **Name Service** role.

Update the Broker UML diagram to include the **Name Service** role.

Multi Type Dispatching

The discussion above allows me to express the process handling multi type invokers more abstractly.

Multi Type Dispatching

Consider an **Invoker** that must handle method dispatching for a large set of roles. To avoid a *blob* or *god class* **Invoker** implementation, you can follow this template:

- Ensure your *operationId* follows a mangling scheme that allows extracting the role name. A typical way is to construct a String type *operationId* that concatenates the type name and the method name, with a unique seperator in between. Example: "FutureGame_getToken".
- Construct **SubInvoker**s for each servant role. A **SubInvoker** is role specific and only handles dispatching of methods for that particular role. The **SubInvoker** implements the **Invoker** interface.
- Develop a **RootInvoker** which constructs a (key, value) map that maps from role names (key) to sub invoker reference (value). Example: if you look up key "FutureGame" you will get the sub invoker specific to the **FutureGameServant**'s methods
- Associate the **RootInvoker** with the **ServerRequestHandler**. In it's *handleRequest()* calls, it demangles the incoming *operationId* to get the role name, and uses it to look up the associated **SubInvoker**, and finally delegates to its *handleRequest()* method.

5.6 Review Questions

Outline the issues involved when there are server side methods that create objects to be returned to the client. Why is it that you just can't marshall the newly created object itself?

Outline the process involved in establishing a relation between a **ClientProxy** on the client side with the correct **Servant** object on the server side, the *Transferring Server Created Objects* process.

Outline the responsibilities of the **Name Service** in this process.

Outline the issue that arise in the **Invoker** code when dealing with many servant types each with potentially many methods. Explain the solution, and how it improves maintainability of the code.