

## Vernetzte Systeme Exercise 12

Ausgabe: 17. Juni 2005  
 Abgabe: 24. Juni 2005

### 1. Task: Implementation of Message Passing

Based on the implementation of persist message queue server and clients done in Exercise 10, this exercise is to use a message passing mechanism to generate remote function operations instead of using pure RPC/RMI mechanism.

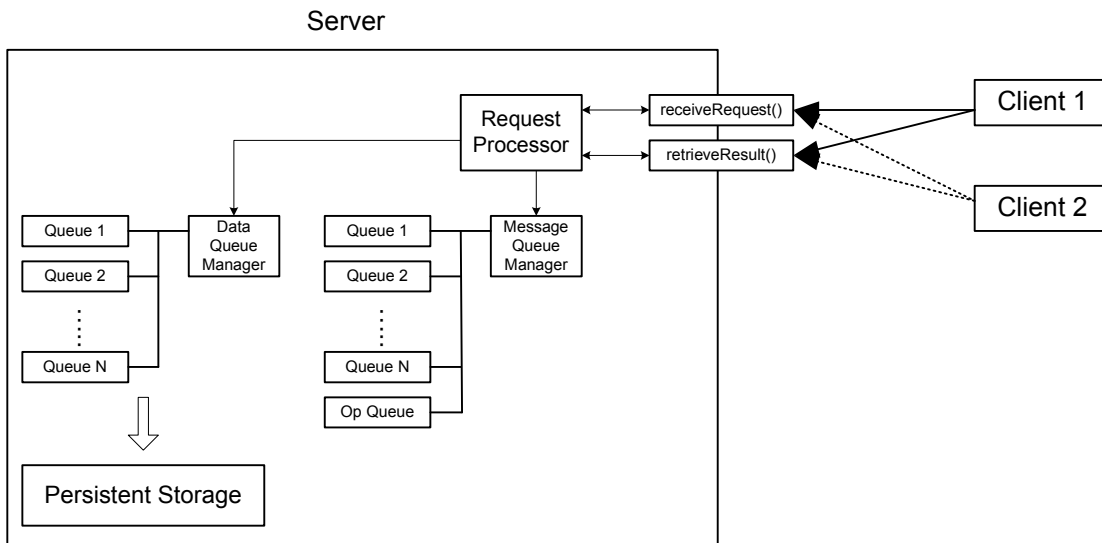


Figure 1: Message Server

- The server `DataMessagePoolServer` implements two remote methods defined in the interface class `DataMessagePool`:

```
package server;

import java.rmi.Remote;
import java.rmi.RemoteException;

import server.messaging.AckMessage;
```

```

import server.messaging.RequestMessage;
import server.messaging.ResultMessage;

public interface DataMessagePool extends Remote{
    public AckMessage receiveRequest(RequestMessage msg) throws
RemoteException;
    public ResultMessage retrieveResult(AckMessage msg) throws
RemoteException;
}

```

List 1. Interface `PersistMessagePool`

- The client sends a request by calling `receiveRequest()` to instruct the server to perform certain operations. Once the request is processed, the server generates a `RequestMessage` containing the output and the result of the operation, and puts it into a specific `ResultMessageQueue`. The server then replies the client with an `AckMessage` which contains the information about the `ResultMessageQueue` where the `ResultMessage` is stored and the identifier of the message. Finally, the client calls `retrieveResult` to retrieve the `ResultMessage` from the server.
- The server contains two `QueueManager`: `PersistDataMessageQueueManager` and `ResultMessageQueueManager`. `PersistDataMessageQueueManager` and its structure are the same as those in Ex10. `ResultMessageQueueManager` has multiple `ResultMessageQueue`: one special `opQueue` to store `ResultMessage` related to the queue management; and the same number of `ResultMessageQueue` which matches one-by-one to the `PersistDataMessageQueue` managed by the `PersistDataMessageQueueManager`.
- `receiveRequest()` method accepts a `RequestMessage` from the client. The `RequestMessage` requests one of the 4 operations (`createQueue`, `deleteQueue`, `putMessage`, `getMessage`) which the server supports.
  - `createQueue`, `deleteQueue`: the server creates/deletes a queue according to its name. The `ResultMessage` generated for the operation is stored in the special `ResultMessageQueue` – `opQueue`.
  - `putMessage`, `getMessage`: the server puts/gets a message to/from a specific `PersistDataMessageQueue`. The `ResultMessage` generated for the operation is stored in the `ResultMessageQueue` corresponding to the `PersistDataMessageQueue` involved in the operation.

The server creates an `AckMessage` which contains the ID of the `ResultMessage` and the name of the `ResultMessageQueue` where the message is stored, and return the `AckMessage` to the client.
- `retrieveResult()` method accepts an `AckMessage` from the client. According to the content of the `AckMessage`, the server retrieves the `ResultMessage` from the corresponding `ResultMessageQueue`, and returns it to the client.

- There are three types of messages exchanged between the server and the clients.
  - `RequestMessage` supports 4 types: `getRequest`, `putRequest`, `createQueueRequest` and `deleteQueueRequest`, which match to the 4 operations the server supports.
  - `AckMessage` contains 2 fields: `String queueName` and `int msgNumber`.
  - `ResultMessage` contains information about the result and output of the operation. If the operation fails, an `OperationException` (an inner class of `ResultMessage`) is attached to the `ResultMessage`. The `OperationException` shows the exception or error happened during the operation. If the operation succeeds, no `OperationException` is attached.
  - `OperationException` is used to replace the `RemoteException` which is directly attached to the RMI method call. It has 5 types:
    - `EXCEPTION_TYPE_MESSAGE_NULL`: It happens when trying to put an empty message to the queue in the server.
    - `EXCEPTION_TYPE_QUEUE_NOT_FOUND`: It happens when trying to delete a queue, put or get messages from or to a queue, and the queue does not exist.
    - `EXCEPTION_TYPE_QUEUE_FULL`: It happens when trying to put a message into a full queue.
    - `EXCEPTION_TYPE_QUEUE_EMPTY`: It happens when trying to get a message from an empty queue.
    - `EXCEPTION_TYPE_QUEUE_DUPLICATION`: It happens when trying to create a queue which is already created.
    - `EXCEPTION_TYPE_UNKNOWN`: For all other unknown exceptions.
- Two clients `MessageGetClient` and `MessagePutClient` are to be implemented. Similar to the clients in Ex10, these two clients are to get/put messages from/to the server.
  - `MessageGetClient` generates a `GetRequestMessage` with the name of the queue from which the data message is to be retrieved, and sends it to the server. Then it gets the `ResultMessage` in which the content of the data message is stored. The client retrieves messages periodically (1 message per 2 second); In case the specified queue cannot be found or the queue is empty, the client will wait and retry.
  - `MessagePutClient` generates a `PutRequestMessage` with the content of the data message and the name of the queue. The client generates messages periodically (1 message per second). In case the queue is full, the client will wait and retry.
- For this exercise, the program should be single threaded.