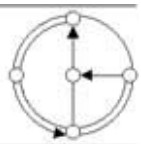
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



SS 2006 Prof. Gustavo Alonso / Angela Nicoară / Patrick Stüedi

Computer Networks
Exercise 9

Start date: June 16, 2006
Due date: June 23, 2006

1. Task: Implementation of Message Passing

Based on the implementation of the persistent message queue server and clients done in Exercise 5, the objective of this assignment is to use a message passing mechanism to generate remote function operations instead of using the pure RPC/RMI mechanism.

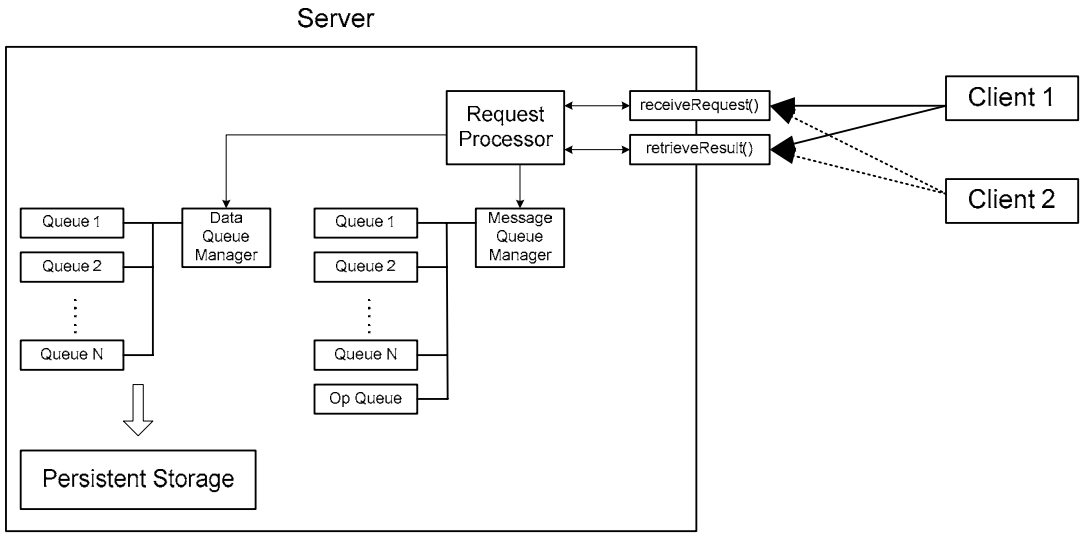


Figure 1: The Message Server

- 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;
}
```

Figure 2. The remote interface **DataMessagePool.java**

- 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** that contains the output and the result of the operation, and puts it into a specific **ResultMessageQueue**. The server then replies to 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 the Exercise 5. **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**.

- the **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 returns the **AckMessage** to the client.

- the `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 Exercise 5, 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.

2. Multiple Choice: 2 Phase Commit (2PC) - 3 Phase Commit (3PC)

Please answer the following questions. For each question, mark the correct answer. There is exactly one correct answer per question.

a) One of the rules of 2PC states “Commit can only be decided if everybody votes YES”. Assume we change that rule to “if everybody votes YES, then the decision must be to COMMIT”. This will result in:

- ☐ the coordinator having to block
- ☐ the participants having to block
- ☐ a smaller probability of blocking
- ☐ less messages being exchanged

b) In linear 2PC, is there any process that is never in an uncertainty period?

- ☐ The one at the beginning
- ☐ The one at the end
- ☐ The one at the beginning and the one at the end
- ☐ None

c) How many messages are exchanged in 2PC for N processes if there are no failures

- ☐ 3N
- ☐ 2N + 1
- ☐ 3N + 1
- ☐ 2N

d) How many messages are exchanged in linear 2PC for N processes if there are no failures

- ☐ 3N
- ☐ 2N +1
- ☐ 3N +1
- ☐ 2N

e) How many rounds of one-way communication are needed for 2PC for N processes if there are no failures (a round implies a set of messages from the same node or sent to the same node)

- ☐ 2
- ☐ 3
- ☐ 2N
- ☐ 3N

f) How many rounds of communication are needed for linear 2PC for N processes if there are no failures (a round implies a set of messages from the same node or sent to the same node)

- □ 2
- □ 3
- □ 2N
- □ 3N

g) How many messages are exchanged in 3PC for N processes if there are no failures

- □ 3N
- □ 2N
- □ 5N
- □ 3N + N/2

h) How many rounds of communication are needed in 3PC for N processes (a round implies a set of messages from the same node or sent to the same node)

- □ 5N
- □ 3
- □ 3N
- □ 5