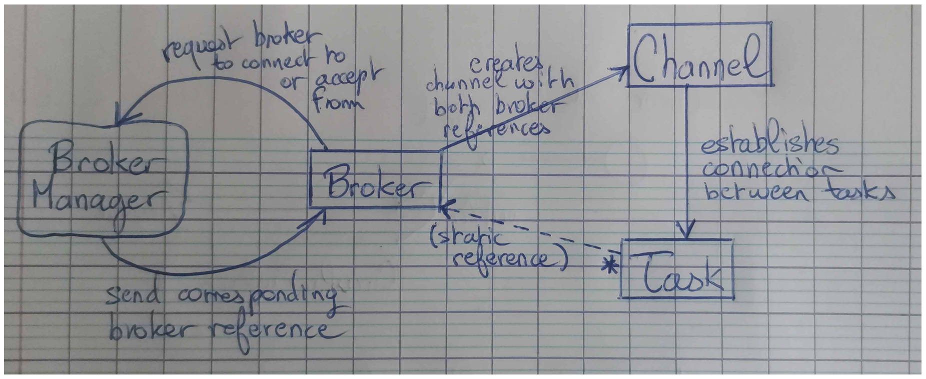
Broker API Design

Overview

Each broker must know the existence of the others it aims to connect to whenever. In order to allow brokers to know each others, an instance of ´´BrokerManager´´ is created in the main thread. This instance stores broker's references about their names and port connection and send them to each broker that requests other broker's reference to establish a new connection:



request broker to connect to or accept from
Broker Manager
Broker
creates channel with both broker references
Channel
establishes connection between tasks
(static reference)
*
Task
send corresponding broker reference

Figure 1: Overview

Why the getBroker() method is static ?

While optimizing the number of calls and calculations in the API's methods, it was necessary to permit any thread or any task to access its broker without needing to call the `BrokerManager`'s instance. This method was implemented in such way:

```
static Broker getBroker() {
    // ...
    return (Task)(Thread.thisThread()).broker;
}
```

RendezVous (RdV) between two tasks

An instance of `RdV` is created every broker's request a connection or an accept with an other broker. Every broker stores the broker requests it sent or received earlier that still waits for a right accepting broker request, in a `RdVMap`'s instance. This instance handles with connect/accept requests and notifies the broker when a connection is ready to be established between two tasks via a `Channel` instance (see below how channels work):

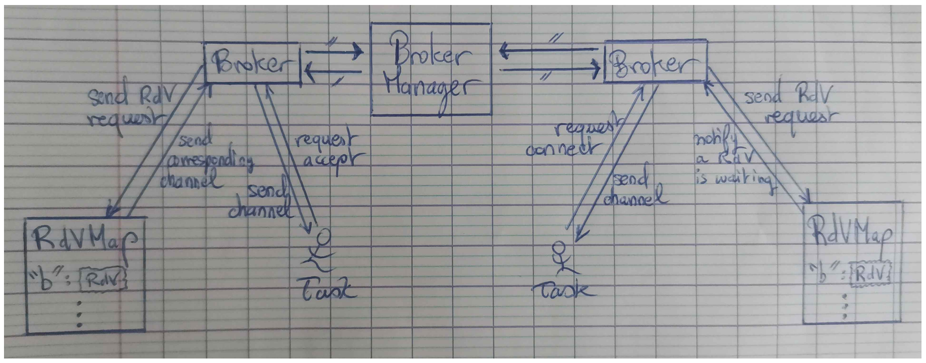


Figure 2: RendezVous

Channels and Circular Buffers

Once a broker has been notified by its own `RdVMap` that a connection must be established with another broker, a `Channel`'s instance is created to permit this connection. This instance is independant from the brokers and the broker manager. It is known by both threads that are communicating each others and it allows a double-way connection:

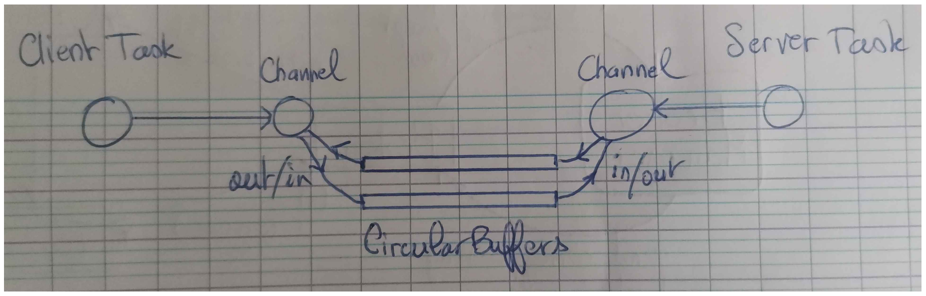


Figure 3: Channel

A `Channel` contains two circular buffers to store exchanged data between task **A** and task **B**. For task **A**, one of the two circular buffers is considered to be the input stream and the other circular buffer is considered to be the output stream. For task **B**, this behavior is reversed so that when task **B** reads bytes sent by task **A**, it occurs in a different stream than the bytes sent by task **B** in order to keep reliability of data exchanging.

Disconnecting

When a thread requests a channel's disconnection, it waits until every reading thread has terminated. When a channel is disconnected, it is impossible to read or write bytes from it:

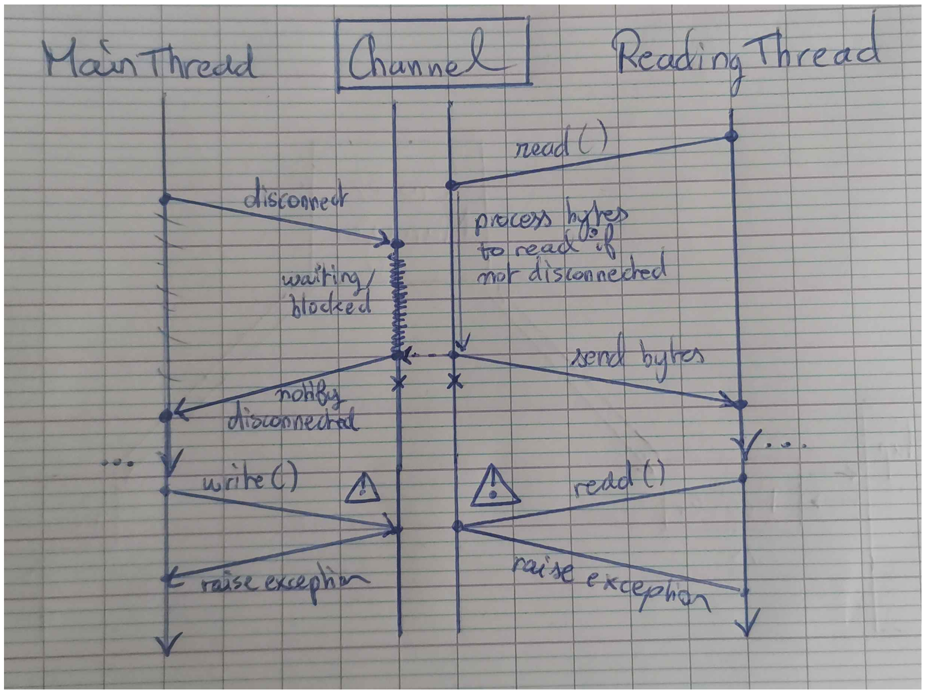


Figure 4: Disconnect

We consider that writing threads can be cut without invoking any misbehavior. However, we must ensure that every reading thread can terminate with success before disconnecting channels.