# Rotalumis Socket class explanation

This document describe the implementation details of the Rotalumis socket class.  
The Rotalumis socket class is implemented as the class PDO\_Socket inside $Rotalumis\core\pdo.h and pdo.cxx. Some extra additions are made to make Rotalumis aware of the new Socket class. These can be found in the document “How To add a native class to Rotalumis.docx” under the Documents folder of the Poosl SVN. The API of the socket class is available in the document “API Specification for Basic Classes.docx” in the same directory under the Poosl SVN.  
The PDO\_Socket class uses the Boost tcp socket as its underlying socket. To store the data that is received on this socket the implementation uses a Boost circular buffer. The interfaces for these classes can be found here: <http://www.boost.org/doc/libs/1_54_0/doc/html/boost_asio/reference/ip__tcp/socket.html>  
<http://www.boost.org/doc/libs/1_54_0/libs/circular_buffer/doc/circular_buffer.html>

## State machine

The best way to explain the behavior of the socket class is to explain the state machine behind the socket class. So let’s first have a look at the following state machine.



As you can see above there are four states that the socket can be in. These states can be changed by any of the state changes defined by the arrows between them. The four different colors separate the four main scenarios that can be executed.

* Black -> The socket will be used as a client to connect to an external socket.
* Green -> The socket will be used as a server and will wait any incoming client connections.
* Red -> These state changes can occur in both server and client socket and are related to writing data on the socket.
* Blue -> These state changes can occur in both server and client socket and are related to reading data on the socket.

The state “connected” and the state change “close” are part of both the black and the green scenarios.

Not all state changes have been described in the state machine and some state changes have been simplified or grouped to avoid clutter in the figure. We list the changed transitions and the motivation why they have been left out / simplified.

* The state change “write\*()” is a grouping for the Poosl API calls;
  + write()
  + writeline()
  + writeString()

These state changes are grouped because they all have the same events and actions for the state machine

* The state change “read\*()” is a grouping for the Poosl API calls;
  + read()
  + readUntil()
  + readWord()
  + readLine()
  + readString()

These state changes are grouped because they all have the same events and actions for the state machine

* All “has\*” Poosl API calls have been left out because they cannot cause any state change.

All “has\*”, “read\*” and “write\*” Poosl API calls can be called from the any state but they will only have an action in the “Connected” state.

The text on the arrows describes when this state change will happen and what the following action will be. All state changes are described by the following syntax:

**Event [guard] / action**

The event part of the state change is a trigger that will either come from a direct call to one of the socket API methods or by internal socket calls. The difference between the two is that all direct calls on the Poosl socket API are *italic* and are followed by ‘()’.

When an action can cause a state change to more than one other state the guard will describe the condition that will lead to this state.

The last part is the next action (if there is one) that is executed.

One thing that cannot be seen in the state machine is the dependency between the internal action of one state change with the event of another. For instance the direct call to “acceptFrom()” will result in a state change to “Listening” and start the action “async\_accept”. What cannot be deducted from the state machine is that “async\_accept” will (in time) trigger the “handle\_accept” event. These cases are described in the table below along with the triggers that cause the event.

|  |  |  |
| --- | --- | --- |
| Action | Trigger | Event |
| async\_accept | Timeout | handle\_accept with error=true |
| async\_accept | Client connected to socket | handle\_accept with error=false |
| async\_connect | Timeout | handle\_connect with error=true |
| async\_connect | Refused by client | handle\_connect with error=true |
| async\_connect | Connected to server socket | handle\_connect with error=false |
| async\_read\_some | Received data on socket | handle\_read with error=false |
| async\_read\_some | Socket has lost connection | handle\_read with error=true |
| async\_write | Write was successful | handle\_write with error=false |
| async\_write | Socket has lost connection | handle\_write with error=true |

## Sequence diagrams

The state machine figure shows that there are four main scenarios the socket can perform. To get more detail about these four scenarios, we present a number of sequence diagrams. These sequence diagrams only show the most common scenarios and are used for explanation only.

In all diagrams there are lifelines for the following three objects:

* Rotalumis Engine -> this is the part running the model and able to make Poosl socket API calls
* PDO\_Socket -> this is the internal socket class that Rotalumis uses to handle Poosl sockets.
* Io\_service -> this is a thread running from the boost library to support all async socket calls.

### Client diagram without error



### Client diagram with error



### Server diagram without error



### Server diagram with error



### Write diagram with and without error

### Read diagram with and without error



### Read diagram while buffer is full

