08: Sockets

Exercise

Goal

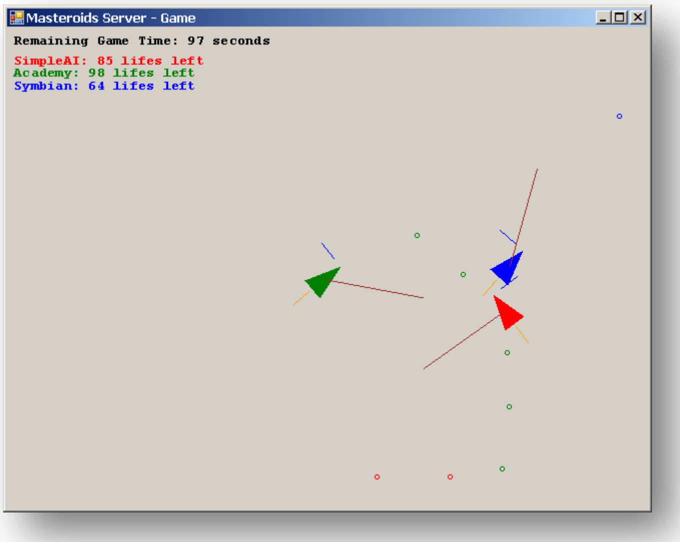
In this exercise, you have to implement the sockets communication part for a multiplayer game client.

Introduction

In this game, the players have to control small spaceships. They float around in space with no gravity or air resistance. Several clients are connected to a central server at the same time and try to shoot others, while evading bullets that are flying around in space.

The “Masteroids”-server is started on a desktop PC (the Microsoft .net framework is required!). Its task is to visualize the game; this visualization should be projected in the lab, so that every client can see it.

One or more Symbian OS / S60 clients can connect to the server. Each client is allowed to control an individual player. It’s possible to rotate the space ship, to accelerate and to shoot. Braking is not directly possible and requires turning by 180° and accelerating again. The physics can be configured by the server before the game is started.



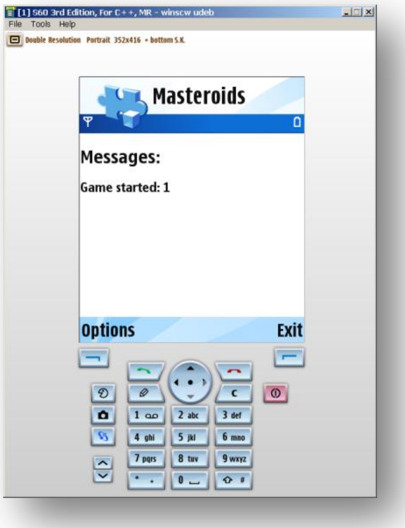
```
Remaining Game Time: 97 seconds

SimpleAI: 85 lifes left
Academy: 98 lifes left
Symbian: 64 lifes left
```

This exercise requires implementing the socket communication part of the client. It’s also possible to extend the task in order to write a custom game AI or to implement a more sophisticated user interface for the Symbian OS client application.

Structure of this Exercise

The user interface of the client application is quite basic – it features a menu to connect to the game server and to send the login information. A text label displays the current status, error messages or information received by the server.



[1] S60 3rd Edition, For C++, MR - winscw udeb
File
Tools
Help
Double Resolution Portrait 352x416 + bottom S.K.
Masteroids
Options
Exit

Messages:

Game started: 1

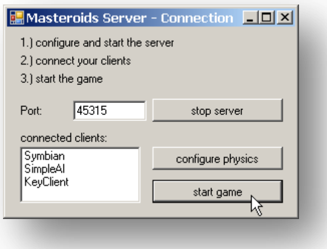
The main task of the Symbian OS client application is to connect to the server through a TCP/IP socket connection. After a successful connection + login, the client can send movement information for its player.

The game protocol used for communication between server and client is described in a chapter at the end of this document, but is not important for this exercise. The required edits are only related to implementing the socket connection, as well as reading from and writing to the socket.

The game framework as well as the existing UI have been prewritten. Import the provided start project into Carbide.c++ and proceed working on the edits. If the import process worked out, the application should run without errors, even before any edits have been performed.

Using the Game Server

Execute the game server application and (optionally) configure the physics of the world. Note that by default a maximum of 10 players is allowed. If projecting the game, set the size of the game field to fill as much of the available resolution as possible.

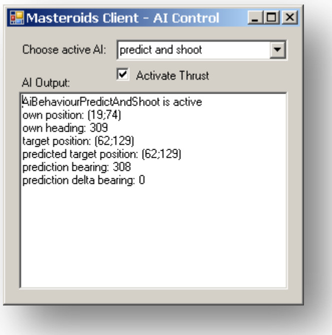


1.) configure and start the server
2.) connect your clients
3.) start the game

Port: 45315

connected clients:
Symbian
SimpleAI
KeyClient

Start the game server to allow clients to connect to the server. When a client is connected to the server but has not sent the login information yet, it is displayed as “-- unknown --”. You can also connect the sample PC implementations of a simple key client and a simple AI client to the server. This is useful to test the game before implementing the Symbian OS client.

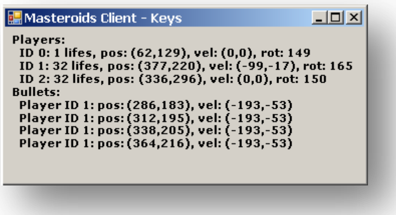


Choose active AI: predict and shoot

☑ Activate Thrust

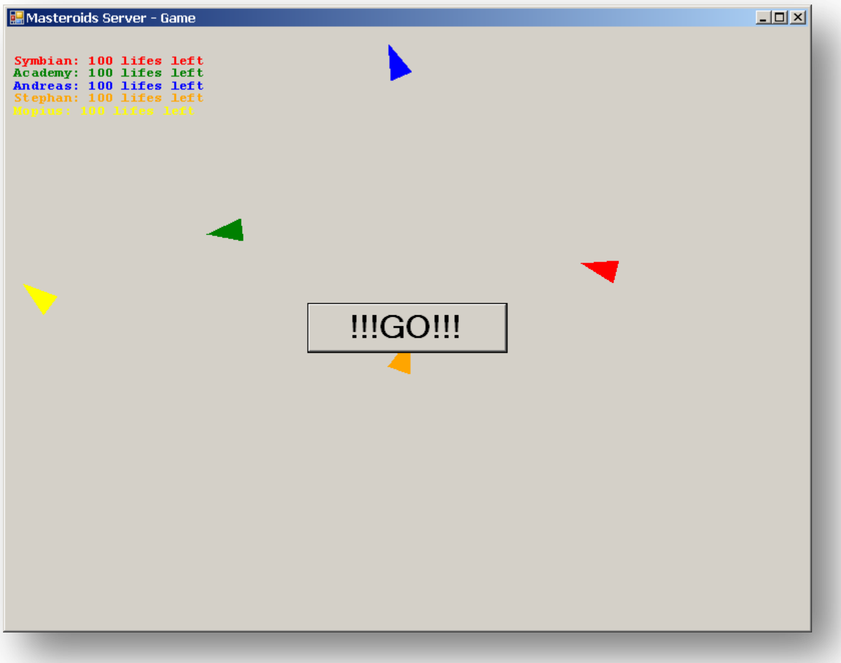
AI Output:

```
AiBehaviourPredictAndShoot is active
own position: (19;74)
own heading: 309
target position: (62;129)
predicted target position: (62;129)
prediction bearing: 308
prediction delta bearing: 0
```



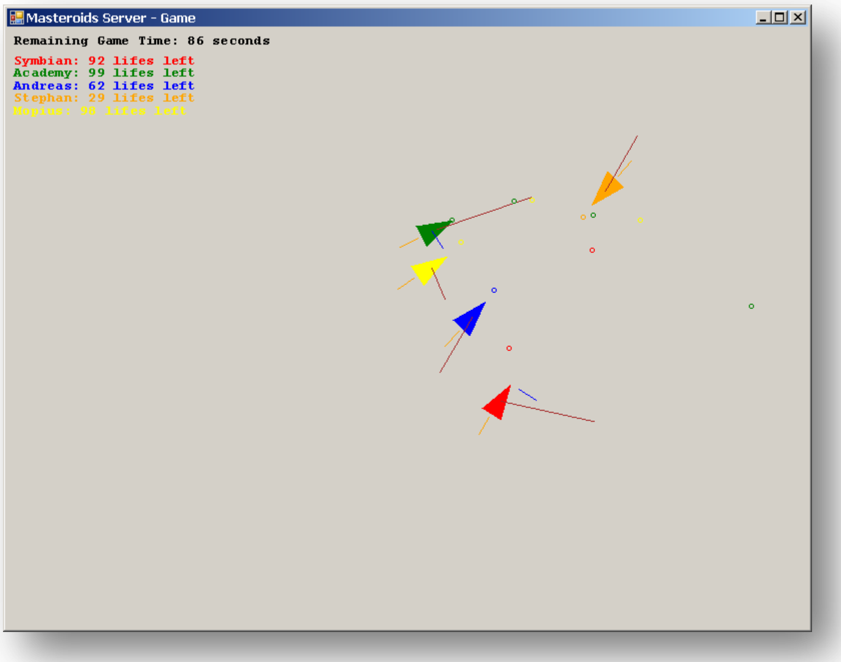
```
Players:
  ID 0: 1 lifes, pos: (62,129), vel: (0,0), rot: 149
  ID 1: 32 lifes, pos: (377,220), vel: (-99,-17), rot: 165
  ID 2: 32 lifes, pos: (336,296), vel: (0,0), rot: 150
Bullets:
  Player ID 1: pos: (286,183), vel: (-193,-53)
  Player ID 1: pos: (312,195), vel: (-193,-53)
  Player ID 1: pos: (338,205), vel: (-193,-53)
  Player ID 1: pos: (364,216), vel: (-193,-53)
```

Once one or more players are connected, you can start the game at any time. After you got to the game screen, you can start the game by clicking on the “GO”-button.



```
Symbian: 100 lifes left
Academy: 100 lifes left
Andreas: 100 lifes left
Stephan: 100 lifes left
Mopius: 100 lifes left
```

Only while the game is active, players are allowed to send control messages. Whenever a client does not behave according to the protocol, an information message is displayed by the server.



```
Remaining Game Time: 86 seconds

Symbian: 92 lifes left
Academy: 99 lifes left
Andreas: 62 lifes left
Stephan: 29 lifes left
Mopius: 98 lifes left
```

The game will run until the time is over or one of the players has won the game.

To test the application, the game server can be run on the same PC as the S60 emulator. In this case, connect the emulator to the IP `127.0.0.1`. The default port (`45315`) should usually work fine. To run the game in a lab, start the game server on the PC that is connected to the projector. Hand out the IP of this PC and make sure that the firewall and the router don't block the port `45315`.

Your Task: The Sockets Engine

The engine that encapsulates the socket connection is the main focus of this exercise. It is split up into three active objects:

- **SocketsEngine.cpp**
    - Responsible for opening a connection to the sockets server
    - Connecting a socket to the remote game server
    - Manages the reader / writer classes for transferring data through the socket connection
- **SocketsWriter.cpp**
    - Sends messages through the socket to the game server
- **SocketsReader.cpp**
    - Waits for messages from the game server and informs the listener whenever a message was received.

Whenever a message has been received or in case a problem occurs, the sockets engine classes inform a listener, which implements the (custom) `MSocketsNotifier` interface. In this example, this is the `CMasteroidsEngine`-class.

General Structure

This section contains a short textual description of the general application structure. This exercise doesn't require modifying or extending other parts than the sockets engine, but it may still be helpful to know more about the pre-written framework.

The user interface (managed by the `CSocketsClientContainerView` and `CSocketsClientContainer`, according to the S60 view architecture) consists of two label controls. One of them is used by the application to display information / error messages.

Through the options-menu, the game engine (`CMasteroidsEngine`) can be instructed to connect to the server and to log in. The IP of the server and the username are hardcoded in the menu selection methods (in the `CSocketsClientContainerView`-class).

During the game, the left / right / up (accelerate) key events and the centre joystick (fire) are processed by the game engine. Messages are prepared according to the game protocol and are sent to the game server (obviously through the sockets engine).

The game processes several of the messages sent by the game server. Some message types are ignored, as they would only be required when implementing an AI or a client-side visualization. Whenever a message is received by the reader-class, a call-back method of the game engine (defined through the `MSocketsNotifier`-interface) is executed.

When the game / sockets engines want to display information- or error-messages to the user, they sent it through the `MMessageDisplay`-interface, which is implemented by the container that owns the label control.

Exercise

An overview of the edits that have to be done to complete this exercise:

Sockets Engine

Edits 1 – 4: SocketsEngine.h
Edits 5 – 18: SocketsEngine.cpp

The main goal of the edits in the sockets engine class is to establish the connection to the socket server and to connect the socket to the game server. As the sockets engine also manages the two active objects that are responsible for writing to / reading from the socket connection, it has to create instances of those two classes.

Sockets Writer

Edit 19: SocketsWriter.h
Edits 20 – 27: SocketsWriter.cpp

The sockets writer just has to send messages (compiled by the game engine) over the socket connection (opened by the sockets engine). To make sure that the message still exists when the Symbian OS socket server handles the asynchronous sending process, the sockets writer has to create a local copy of the message.

Sockets Reader

Edits 28 – 29: SocketsReader.h

Edits 30 – 36: SocketsReader.cpp

The sockets reader has to wait until data is received from the socket. Whenever this is the case, the `RunL()` method of the active object is executed. If a message was received, the sockets reader has to inform the game engine.

View Class and General Edits

Edits 37 – 39 + two TODOs: SocketsClientContainerView.cpp

The edits in this class are necessary to call the connect method of the sockets engine. An object that stores the URL as well as the port has to be created. Two literals marked with TODO contain the address of the server and the username of this client, which should be adapted.

Possible Extensions

The tasks described below are optional. They extend the original application with new features and require working independently, without instructions of the steps needed to complete them. All of the possible extensions (except DNS) are not covered in the Symbian Academy course materials.

DNS

As the game server is intended to be started in the local network, the socket engine of this exercise directly connects to an IP address. Add a feature that allows dynamic host name resolution (DNS), in order to allow specifying a host name instead of only an IP.

User Interface

Extend the application to offer dialogs for entering the player name and the target IP address / port. Implement a more sophisticated design to display the current status of the client and the game. Also, extend the user interface to allow disconnecting from the server without closing the application.

Visualization

Implement a (scaled down) visualization of the game field on the client.

Note: This task is not directly related to Symbian OS training.

Artificial Intelligence

Implement an AI client for the game using the protocol information below. The server sends out information about the players and the bullets in regular intervals. The data is sufficient to implement an own AI algorithm.

Note: This task is not directly related to Symbian OS training.

Protocol and Details

The provided Symbian OS client ignores the game status messages sent by the server. If you want to visualize the game on the client or implement your own AI, it is necessary to interpret the rest of the messages from the server. This chapter presents an overview of the protocol used by the game.

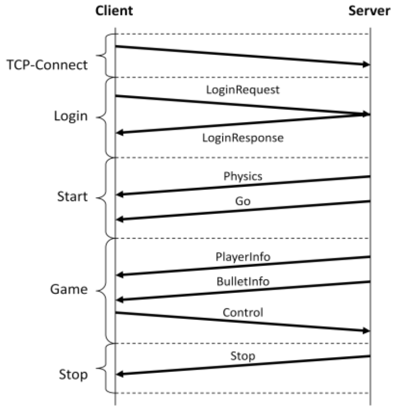
Protocol Flow Diagram

First, the client has to log in and gets a response indicating if the request was successful.

When the server starts the game, the physics of this round are sent to the clients, followed by a “go”-message.

During the game phase (only between “go” and “stop”), the server regularly sends player and bullet information messages. The clients can send their control information to move their space ships.

When the round is over, the server sends a “stop” message to the clients.



Protocol Specification

The protocol is sent over TCP, values have to be coded in little-endian.

General definition

Each message consists of the following parts:

- MID = Message ID
- Size = Length of the following payload
- Payload = Data, depending on the message (as specified by the MID)

2 Byte UINT	2 Byte UINT	0 – N Bytes
MID	Size	Payload

Messages

The following messages are defined in the protocol:

Login Request Message (Client → Server)

MID	Length	N Bytes
1	N	ASCII string – nickname

Login Response Message (Server → Client)

MID	Length	1 Byte
2	1	Login success (1 = OK, 0 = NO)

Physics Message (Server → Client)

MID	Length	24 Bytes
3	24	Physics values

- 2 Byte UINT: playfield size X [Pixel]
- 2 Byte UINT: playfield size Y [Pixel]
- 2 Byte UINT: number of players in game
- 2 Byte UINT: maximum number of lifes per player
- 2 Byte UINT: maximum ship velocity [pixels / second]
- 2 Byte UINT: ship‘s thrust [pixels / second^2]
- 2 Byte UINT: ship‘s angular velocity [degrees / second]
- 2 Byte UINT: ship‘s weapon cooldown [ms]
- 2 Byte UINT: Bullet velocity; [pixels / second]
- 2 Byte UINT: maximum number of bullets on screen [bullets / player]
- 2 Byte UINT: bullet radius [pixel]
- 2 Byte UINT: radius of ship‘s bounding sphere [pixel]

Go / Stop Message (Server → Client)

MID	Length	1 Byte	2 Bytes
4	3	Go-flag (1 = start, 0 = stop)	Assigned player ID

Control Message (Client → Server)

MID	Length	1 Byte	1 Byte
5	2	Action	Parameter

- **Action = 0: Rotation**
  - Parameter = 0: No Rotation
  - Parameter = 1: Rotate Left
  - Parameter = 255: Rotate Right
- **Action = 1: Thrust**
  - Parameter = 0: Thrust off
  - Parameter = 1: Thrust on
- **Action = 2: Fire**
  - Parameter unused

Players Info Message (Server → Client)

MID	Length	2 Byte UINT	14 Bytes	14 Bytes	...
6	2+14*P	Number of players (P)	Player info	Player info	...

- 2 Byte UINT: Player ID
- 2 Byte UINT: Remaining Lifes
- 2 Byte UINT: Position X [pixels]
- 2 Byte UINT: Position Y [pixels]
- 2 Byte SINT: Velocity X [pixels / second]
- 2 Byte SINT: Velocity Y [pixels / second]
- 2 Byte UINT: Rotation [degrees]

Bullets Info Message (Server → Client)

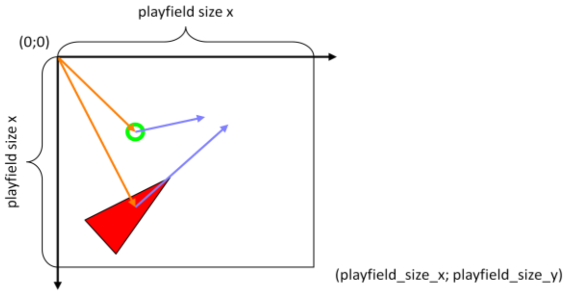
MID	Length	2 Byte UINT	10 Bytes	10 Bytes	...
7	2+10*B	Number of bullets (B)	Bullet info	Bullet info	...

- 2 Byte UINT: Player ID
- 2 Byte UINT: Position X [pixels]
- 2 Byte UINT: Position Y [pixels]
- 2 Byte SINT: Velocity X [pixels / second]
- 2 Byte SINT: Velocity Y [pixels / second]

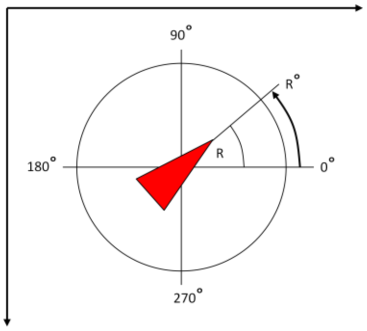
Coordinate System

The playfield is specified by a top down coordinate system. The total size can be configured by the server (→ physics settings).

The following image visualizes the positions and velocity vectors of players and bullets.



Rotation of the players is measured in a top down coordinate system:

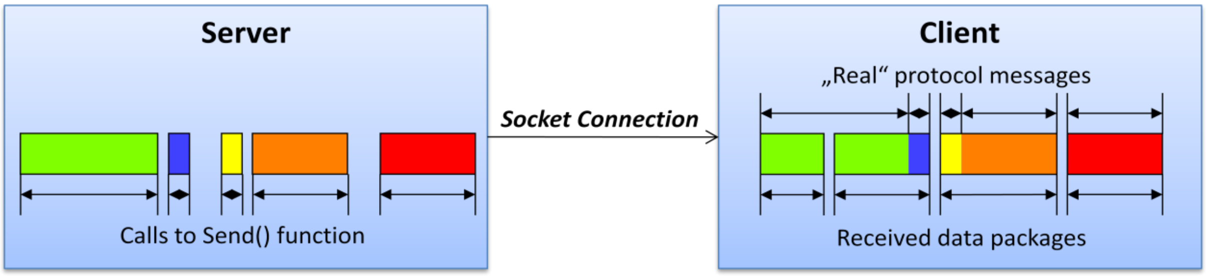


Accumulation and Fragmentation

It isn't guaranteed that messages are retrieved in one packet and that each of the received packages only contains one message. This also applies to the Symbian OS `RecvOneOrMore()`-call – the data that is read can be received in one of the following ways:

🟥	Usually, messages are received in exactly the same structure as they are sent.
🟨🟧	Sometimes, several messages can be accumulated in one package (e.g. when sent shortly after each other)
🟩 🟩	Messages can be split up and read through multiple calls to the read-method (e.g. when the read-buffer is too small)
🟩 🟩🟦	Combination of the possibilities above.

The following diagram visualizes the data transfer between server and client:



To correctly parse the protocol, the client should buffer the messages and parse them after they have been received. It should be prepared to have more than one message in the buffer and should not have problems if a message is not yet fully received. The game engine implementation provided with the sample Symbian OS client does already implement this behaviour.