Open Robot Control and Simulation Library

– ORCS –

ORCS Standard Protocol (OSP)

Interfacing Agents of the ORCS Simulator via UDP

Version 1.1
(September 2008)

Christian Rempis
Verena Thomas

Contents

1 ORCS Standard Protocol

The communication protocol between ORCS and external applications (e.g. an evolution environment) allows for the investigation, manipulation, observation and control of the simulation process and the simulation scenarios. This protocol is called the ORCS Standard Protocol (OSP).

The OSP uses the UDP protocol for the communication between clients and servers. Using UDP reduces the delay between messages compared to TCP, but because of its unreliability it is recommended to be used only at the loop-back adapter of the computer. Thus the ORCS simulator and the OSP clients have to run on the same computer to use the protocol in a reliable way. A fast communication is very important, as communication takes place between each simulation step when the OSP is used to control agents.

The communication between the ORCS simulator and OSP clients supports the following commands and functions:

- Request an overview on all available controllable agents and their control state.
- Request detail information about a specific controllable agent.
- Register for and deregister from a controllable agent.
- OSP clients can control an arbitrary number of controllable agents in an ORCS simulation.
- An arbitrary number of OSP clients can connect to the OCRS simulator to control the simulation or simulated agents.
- OSP clients can trigger the execution of simulation steps or reset the simulation in a synchronized way.
- Any OSP client can reset the entire simulator communication to restart the communication if a serious problem with one of the OSP clients occurs.
- OSP clients can search for variables in the global variable repository of the ORCS simulator and register as remote observers to these variables.
- OSP clients can also directly read or change variables remotely.
- OSP clients can search and register for ORCS events to be notifies when events occurred.

2 Basic Structure of the Client-Server System

The ORCS simulator can be started with an OSP server running as the simulation controller. This means that the OSP server controls the simulation step execution, initialization, reset, parameterization and synchronous task execution. So it is running the main thread of execution of the simulator. The OSP server is bound to a specific port (default is 45454) and waits for clients. OSP clients can send an initial datagram to the server to establish a connection. On such a request the OSP server starts a new client handler in an own thread for each OSP client connecting to the simulator. Such a client handler is responsible for the entire communication with the OSP client. OSP clients can terminate a connection by sending a termination command. Furthermore any client can reset the entire simulation communication, which results in a termination of *all* connected OSP clients. This is useful if a client detects a problem in the simulator communication (e.g. a lock due to missing datagram packets), so it can restart the communication without the need to restart the entire simulator.

3 Data Format

Messages of the OSP protocol use a single byte as header to identify the command. The remaining part of a datagram packet is dependent on the command type. These commands are explained later in section 4.

As part of the messages data is submitted. The protocol makes use of integers, floats, strings and single bytes.

Integers are sent as 4 byte numbers in little-endian format. Thus only 32 bit integers are supported. If the OSP client internally works with larger integers, then the remaining bytes have to be truncated to the 32 most significant bits.

```
(char) (value >> 0)
(char) (value >> 8)
(char) (value >> 16)
(char) (value >> 24)
```

Floats are sent as 4 byte floating-point values according to the IEEE 754 floating-point "single format" bit layout. In Java this can be achieved by using `Float.floatToIntBits()`, in C++ by interpreting the float bytes as int bytes. The transformed integer representing the float bits can then be sent as an ordinary integer.

Doubles are sent as 4 byte Floats. All remaining bytes are discarded. Therefore the maximum precision of a double number using the protocol is 32 bit.

Strings are sent as a sequence of bytes, terminated by a `'\0'`. This termination is important as the length of a string is not specified separately.

Bytes are simply sent without modifications.

4 Protocol Command Reference

4.1 Connecting to the ORCS OCP Server

An OSP client has to send an initial connection request datagram to the OSP server of the simulator. The message contains the main protocol number and a minor revision number. These numbers are required to decide, which client handler has to be created to support the communication with the client. This allows a flexible extension of the protocol by sub-protocols or entirely different protocols through the same OSP server. To keep compatible with older protocols the server simply creates an older version of the client handler, so most applications stay compatible, while new applications can already use the newer versions.

The currently supported protocol number for the OSP protocol is 1.1 (Major 1 / Minor 1).

Connection Request:

```
Client <: [UDP_INIT_COMMUNICATION (byte),
           UDP_PROTOCOL_MAJOR (int),
           UDP_PROTOCOL_MINOR (int)]
Server >: [UDP_INIT_COMMUNICATION_ACK (byte),
           UDP_PROTOCOL_MAJOR (int),
           UDP_PROTOCOL_MINOR (int)]
```

The UDP datagram sent by the server also contains the host address and the port of the UDP socket of the new client handler, which is responsible for the rest of the communication with the client. All further commands therefore have to be sent to this socket address.

4.2 Restarting the Simulator Communication

In case that the simulator communication gets stuck because of a misbehaving client or if an already running simulator should be reused for an entirely new scenario, the communication can be restarted. This cancels all connections to the simulator and therefore all registrations for agents, values or events are also canceled. These can then be reassigned to other clients.

To restart the communication, any client can simply send the communication reset command.

Restart the Simulator Communication:

```
Client <: [UDP_RESET_COMMUNICATION (byte)]

foreach <client>
Server >: [UDP_RESET_COMMUNICATION_ACK (byte)]
```

The reset confirmation is sent to all connected clients so that they get notified and can reconnect if desired. Clients are not expected to confirm the reset message.

4.3 Controlling Agents

The ORCS simulator allows the definition of an arbitrary number of agents, that can be controlled by external of internal clients, e.g. by neural networks evaluation tools. Agents are represented by so called `SimObjectGroups`. These are collections of `SimObjects` logically belonging together. A `SimObject` can be any part of the physical simulation, like a sensor, an actuator, a body part or an environment object. `SimObjects` can provide so called `InterfaceValues`, which represent motors or sensors as named, normalized double variables. These `InterfaceValues` can be used to interact with their `SimObjects`. A `SimObject` can provide three types of `InterfaceValues`:

- **Input Values**: These `InterfaceValues` can be set by an OSP client and can be used to control an agent, e.g. by setting a motor torque.
- **Output Values**: Sensors provided by the `SimObject` are exported to the controller client via output values. An example is the measured angle between two objects.
- **Info Values**: Such `InterfaceValues` are exported like output values, but they are not interpreted as part of the controller (e.g. as input neuron in a neural network). Instead info values can be used by other parts of the client application, e.g. by fitness functions, network filters or statistic functions.

All provided `InterfaceValues` of all `SimObjects` of a `SimObjectGroup` are automatically collected and become part of the communication interface of an agent.

In addition, each `SimObjectGroup` can be identified by a human readable identifier, usually a simple name.

To control an agent an OSP client first has to find the desired agent it wants to control. Therefore it can request an overview over all available agents from the ORCS simulator.

Overview on Available Agents:

```
Client <: [UDP_GET_AGENT_OVERVIEW (byte)]
Server >: [UDP_AGENT_OVERVIEW (byte),
           <numberOfAgents> (int)]

foreach <numberOfAgents> do
Server >: [UDP_AGENT_OVERVIEW_NEXT (byte),
           <datagramIndex> (int),
           <groupId> (int),
           <groupType> (string),
           <available> (byte),
           <groupName> (string)]


Client <: [UDP_AGENT_OVERVIEW_ACK (byte)]
```

With the information obtained by the overview command, an agent can get specific information about an agent, especially about the number of `InterfaceValue`s and their type and name. This information can be used to check whether the client is compatible with a certain agent and its interface. The `<available>` flag indicates whether the agent is already controlled by an OSP client.

Information on Agent:

```
Client <: [UDP_GET_AGENT_INFO (byte),
           <idOfAgent> (int)]
Server >: [UDP_AGENT_INFO (byte),
           <idOfAgent> (int),
           <numberOfInputs> (int),
           <numberOfOutputs (int),
           <numberOfInfos> (int),
           <agentName> (string)]

foreach <numberOfInputs> <numberOfOutputs> <numberOfInfos> do
Server >: [UDP_AGENT_INFO_NEXT (byte),
           <index> (int),
           <minValue> (float),
           <maxValue> (float),
           <valueName> (string)]

Client <: [UDP_AGENT_INFO_ACK (byte)]
```

An OSP client now can decide to control an agent, if its interface is compatible and if the agent is not already controlled by another client. The control state of an agent is indicated by the `<available>` flag in the agent overview reply. An agent can register for a free agent the following way:[1]

Control an Agent:

```
Client <: [UDP_REGISTER_FOR_AGENT (byte),
           <idOfAgent> (int)]
Server >: [UDP_REGISTER_FOR_AGENT_ACK (byte),
           <idOfAgent> (int),
           <status> (byte)]
```

The `<status>` flag is 1 at success and 0 if the agent could not be found or if the agent is already controlled by another client. Each client can register for as many agents as desired. Each registration has to be done separately.

[1]Before a Client can access information about or (de-)register an agent, the Overview Command (page 5) has to be sent. Otherwise the client handler does not know any agent by its id.

A client can also deregister from an agent to allow other clients to register and control this agent.

Deregister Agent Control:

```
Client <: [UDP_DEREGISTER_FROM_AGENT (byte),
           <idOfAgent> (int)]
Server >: [UDP_DEREGISTER_FROM_AGENT_ACK (byte),
           <idOfAgent> (int),
           <status> (byte)]
```

The `<status>` flag is 1 if the deregistration was successful, 0 if the deregistration failed and 2 if the agent with the id could not be found. In case the agent was not controlled by the OSP client, also 1 is returned.

4.4 Monitoring Events

Clients can register for regular `Events` of the simulator. Such `Events` are used internally to trigger most of the simulator phases (like simulation steps, the randomization phase, initialization, etc.). However the same type of events can also be used to signal other events during simulation, such as selected object collisions or event based sensors. Clients that are registered for events get a notification after each simulation step about all the registered events that did occur during the step. Such information is useful for fitness functions or to terminate an evaluation when the agent violates a critical requirement.

To register for an `Event`, the client has to know its exact global name.

Register for an Event:

```
Client <: [UDP_REGISTER_FOR_EVENT (byte),
           <eventName> (string)]
Server >: [UDP_REGISTER_FOR_EVENT_ACK (byte),
           <localEventId> (int)]
```

The `<localEventId>` uniquely identifies a specific `Event`. This id is only unique for this client, so the id can not be used for comparison with event ids of other clients. If an `Event` could not be found, then the `<localEventId>` is 0. For all further work with the `Event` only this local id is used. For instance after a simulation step all the ids of all registered `Events` that occurred during the step are communicated to the client.

Deregister from an Event:

```
Client <: [UDP_DEREGISTER_FROM_EVENT (byte), <localEventId>]
Server >: [UDP_DEREGISTER_FROM_EVENT_ACK (byte), <localEventId>]
```

4.5 Observing Variables of the Variable Repository

The OSP protocol also allows to read, write and observe global variables of the global variable repository of the simulator. Such variables (class `Value`) contain parameters of the simulation, like update frequency, physical accuracy or the simulation step size. Furthermore all parameters of the physical simulation model can be accessed via the variable repository, such as the dimension, orientation and location of objects. Such variables can be modified by clients to adjust the simulator by remote. Parameters of the physical simulation model can be helpful for instance during the evaluation of fitness functions. Even modifications of the physical model during the evolution is possible, e.g. when co-evolving the robot morphology and the controllers.

To access a variable from the repository, the client has to know the exact name of this value. As there may be many variables and the variable names are organized with path names to maintain namespaces, there is a command to request all variable names matching a given regular expression. This way the client can get for instance all Variable names containing "StepSize" or ending with "Sensor".

Request Details on Available Variables Matching a Regular Expression:

```
Client <: [UDP_GET_VALUE_IDS (byte),
          <regularExpression> (string)]
Server >: [UDP_VALUE_IDS (byte),
          <numberOfFoundValues> (int)]

foreach <numberOfValuesFound>
Server >: [UDP_VALUE_INFO (byte),
           <index> (int),
           <localValueId> (int),
           <valueType> (string),
           <fullValueName> (string)]
```

The `<index>` is used to verify the number of received datagrams. The `<localValueId>` is (like the `localEventId`) a unique identifier only in the context of the client. Any further usage of `Values` is done via their ID, not using their names. The `<valueType>` is a string describing the variable type, e.g. `String`, `Integer`, `NormalizedDouble` or `Point3D`. This information is required to verify that the found variable has a compatible data type to the one required at the client. The `<fullValueName>` is a string with the full name of the `Value`.

Knowing a `Value`'s `<localValueId>` a client can read its content.

Read a Variable from the Repository:

```
Client <: [UDP_GET_VALUE (byte),
           <localValueId> (int)]
Server >: [UDP_VALUE (byte),
           <localValueId> (int),
           <valueContent> (string)]
```

The `<valueContent>` always is a string, which is created by the `Value` object in a type dependent manner. For further information about the exact string format the ORCS API documentation of the various `Value` subclasses can be consulted.

A Value can also be set by remote.

Set a Variable from the Repository:

```
Client <: [UDP_SET_VALUE (byte),
           <localValueId> (int),
           <valueContent> (string)]
Server >: [UDP_SET_VALUE_ACK (byte),
           <localValueId> (int),
           <status> (byte)]
```

The `<status>` flag indicates a successful operation with a 1. If there was a problem to parse the string into the correct data format, then `<status>` is 0. If the specified `Value` could not be found, `<status>` is 2;

In some cases it makes sense to register for a `Value`, which results in an automatic delivery of the `Value`'s content after each simulation step (see section 4.7). This function should be rarely used as the `Value` content usually is sent as a string, which is comparably expensive to send (in terms of performance).

Register for the Observation of a Variable:

```
Client <: [UDP_REGISTER_FOR_VALUE (byte),
           <numberOfValues> (int),
           <localValueId1> (int),
           <localValueId2> (int),
           ...,
           <localValueId_n> (int)]
```

```
Server >: [UDP_REGISTER_FOR_VALUE_ACK (byte),
           <localValueId1> (int),
           <localValueId2> (int),
           ...,
           <localValueId_n> (int)]
```

The `<localValueId>`s in the server reply are the same as in the client request for all `Values` that could be found. All other IDs are set to 0 to indicate that the `Values` could not be found in the simulator and therefore registration failed. Previously registered `Values` are not affected by the registration command, so that they are still registered independently of whether they have been mentioned in the last registration command or not.

The deregistration of Values is analog:

Deregister from a Variable in the Repository:

```
Client <: [UDP_DEREGISTER_FROM_VALUE (byte),
           <numberOfValues> (int),
           <localValueId1> (int),
           <localValueId2> (int),
           ...,
           <localValueId_n> (int)]

Server >: [UDP_DEREGISTER_FROM_VALUE_ACK (byte),
           <numberOfValues> (int),
           <localValueId1> (int),
           <localValueId2> (int),
           ...,
           <localValueId_n> (int)]
```

Here the `<localValueId>`s are also set to 0 if deregistration failed, e.g. because the Value could not be found.

4.6 Simulation Reset

The simulation has to be reset before each simulation try. During the reset all physical objects are moved to their starting positions and initial inner states, so that each simulation starts with the same initial conditions. An exception are randomized parameters, which can be used to vary certain initial conditions. But even these randomized parameters depend on a deterministic randomization seed, so that the initial conditions of the physical simulation for a specific randomization seed always is the same.

To trigger a simulation reset, all clients controlling at least one agent have to send a simulation request for each agent they control. Hereby they submit the desired randomization seed. If clients send different randomization seeds, then the last valid seed is used. A randomization seed of 0 is interpreted as "don't care".

Trigger Simulation Reset:

```
Client <: [UDP_RESET_SIMULATION (byte),
           <randomizationSeed> (int)]
Server >: [UDP_RESET_SIMULATION_ACK (byte)]
```

When all agent controlling clients have sent a reset request for each controlled agent, the server executes a simulation reset and notifies all clients about its completion.

Reset Notification:

```
Server >: [UDP_RESET_SIMULATION_COMPLETED_ACK (byte)]
Client <: [UDP_RESET_SIMULATION_COMPLETED (byte)]
```

4.7 Simulation Step Execution

After a simulation reset clients can start to execute simulation steps. To enforce a synchronous execution, a client has to request the execution of a new simulation step for each controlled agent (as for the simulation reset). The execution of a simulation step is separated in two phases, the trigger phase and the completion phase. In the trigger phase, each client sends a next step request for each controlled agent to the server, containing the actuator settings required to control the agent.

Request the Next Simulation Step:

```
Client <: [UDP_NEXT_SIMULATION_STEP (byte),
           <agentId> (int),
           <numberOfInputs> (int),
           <input1> (float),
           <input2> (float),
           ...,
           <input_n> (float)]
Server >: [UDP_NEXT_SIMULATION_STEP_ACK (byte),
           <agentId> (int)]
```

The `<agentId>` identifies the agent whose input data are sent. This is required to distinguish between different agents if more than one agent is controlled by the same OSP client. The following `<inputs>` contain the current input settings of the actuators, in the same order as given in the *Agent Information* message (section 4.3).

After all agent controlling clients sent a next step request for each controlled agent, the server executes a simulation step. After its completion (completion phase) the server sends a notification message to each client for each controlled agent. Such a notification message contains the sensor (output) and info data of the agent, a list of `Events` that occurred during the simulation step - if the client registered for these `Events` - and all `Values` that are currently observed by the client.

Simulation Step Completed Notification:

```
Server >: [UDP_NEXT_SIMULATION_STEP_COMPLETED (byte),
           <agentId> (int),
           <numberOfOutputs> (int),
           <numberOfInfos> (int),
           <numberOfOccuredEvents> (int),
           <numberOfObservedValues> (int),
           <output1> (float),
           ...,
           <output_n> (float),
           <info1> (float),
```

```
            ...,
            <info_n> (float),
            <occuredEventId1> (int),
            ...,
            <occuredEventId_n> (int)]

foreach <numberOfObservedValues>
Server >: [UDP_VALUE (byte),
            <localValueId> (int),
            <valueContent> (string)]

Client <: [UDP_NEXT_SIMULATION_STEP_COMPLETED_ACK (byte),
            <agentId> (int)]
```

The order of the `<outputs>` and `<infos>` is the same as in the *Agent Information* message (section 4.3). The `<occuredEventId>`s contain only the local IDs of `Events` that occurred during the simulation step. Other registered `Events` are not shown.

In case a client controls more than one agent, the observed `Values` and registered `Events` are sent only in the first of the notification messages to reduce traffic. In all other notifications of the current simulation step `<numberOfOccuredEvents>` and `<numberOfObservedValues>` are set to 0.

5 Appendix: Protocol Command and Reply Identifiers

All identifiers are unsigned chars (this is important when using JAVA because bytes are ranging from -128 to +128 instead of from 0 to 255 there).

Simulation Step Completed Notification:

```
UDP_INIT_COMMUNICATION = 5;
UDP_INIT_COMMUNICATION_ACK = 6;
UDP_END_COMMUNICATION = 7;
UDP_END_COMMUNICATION_ACK = 8;
UDP_RESET_COMMUNICATION = 10;
UDP_RESET_COMMUNICATION_ACK = 11;


UDP_REGISTER_FOR_EVENT = 20;
UDP_REGISTER_FOR_EVENT_ACK = 21;
UDP_DEREGISTER_FROM_EVENT = 30;
UDP_DEREGISTER_FROM_EVENT_ACK = 31;


UDP_GET_VALUE_IDS = 40;
UDP_VALUE_IDS = 41;
UDP_VALUE_INFO_ACK = 42;
UDP_VALUE_INFO = 43;
UDP_GET_VALUE = 50;
UDP_VALUE = 51;
UDP_SET_VALUE = 52;
UDP_SET_VALUE_ACK = 53;
UDP_REGISTER_FOR_VALUE = 54;
UDP_REGISTER_FOR_VALUE_ACK = 55;
UDP_DEREGISTER_FROM_VALUE = 56;
UDP_DEREGISTER_FROM_VALUE_ACK = 57;


UDP_RESET_SIMULATION = 70;
UDP_RESET_SIMULATION_ACK = 71;
UDP_RESET_SIMULATION_COMPLETED = 75;
UDP_RESET_SIMULATION_COMPLETED_ACK = 76;
UDP_NEXT_SIMULATION_STEP = 80;
UDP_NEXT_SIMULATION_STEP_ACK = 81;
UDP_NEXT_SIMULATION_STEP_COMPLETED = 85;
UDP_NEXT_SIMULATION_STEP_COMPLETED_ACK = 86;


UDP_GET_AGENT_OVERVIEW = 90;
```

```
UDP_AGENT_OVERVIEW = 91;
UDP_AGENT_OVERVIEW_NEXT = 92;
UDP_AGENT_OVERVIEW_ACK = 93;
UDP_GET_AGENT_INFO = 95;
UDP_AGENT_INFO = 96;
UDP_AGENT_INFO_ACK = 97;
UDP_AGENT_INFO_NEXT = 98;
UDP_REGISTER_FOR_AGENT = 100;
UDP_REGISTER_FOR_AGENT_ACK = 101;
UDP_DEREGISTER_FROM_AGENT = 105;
UDP_DEREGISTER_FROM_AGENT_ACK = 106;


UDP_DATAGRAM_END = 120;
```