

Evolving Robots In A Simulated World: Specification Of A Communication Protocol

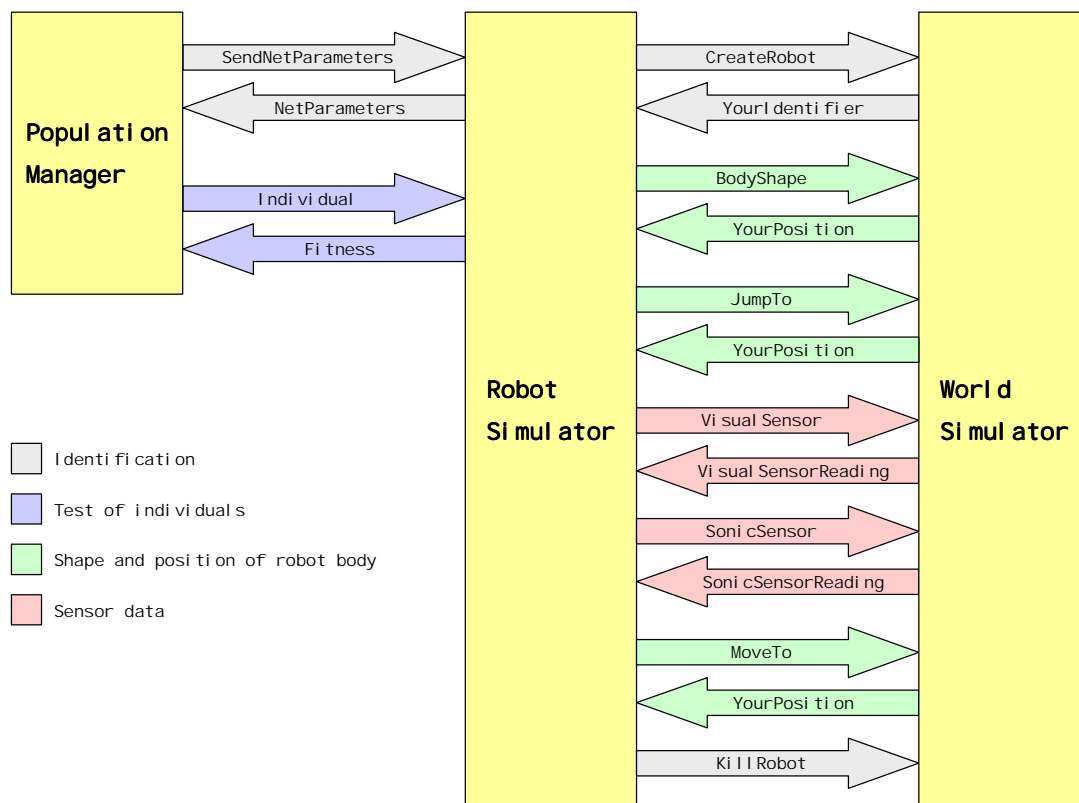
1 Overview

Generally we use three pieces of software (programs) to evolve robots:

- Population Manager: holds a generation of genotypes and moves them on into the next generations using genetic algorithms.
- Robot Simulator: maps the genotype onto a phenotype, simulates the robots sensor-motor-actions for some timespan and reports a fitness value back.
- World Simulator: tells the Robot Simulator the position of its virtual body and the distance to other objects that reside in the simulated world.

Robot Simulator and World Simulator may be combined in a single program, but this has no consequence on the protocol and is therefore ignored for the rest of this document.

The Population Manager and the two simulators talk to each other using a communication protocol. This protocol is described in detail in this document. All possible messages, that can be used within the protocol, are shown in the following chart.



We have 15 messages in total, one of which is used three times ("YourPosition"). They are used in pairs: the client (always to the left in the chart) asks the server (to the right) for an answer – only "KillRobot" is a one-way message.

The Population Manager is a client to the Robot Simulator, the Robot Simulator is a server to the Population Manager and client to the World Simulator at the same time. Finally, the World Simulator is a server to the Robot Simulator.

2 Protocol Specification

All programs may be written and compiled using different programming languages and executed on a single computer (probably running Windows2000) or distributed over several computers, which are interconnected through the same network (even the Internet). Therefore we have to use a platform independent protocol.

2.1 Connection

The programs connect to each other using TCP/IP sockets. As a consequence each running instance of a program, that acts as a server should have a unique combination of IP address and port.

The IP address is fully determined by the computer the program is running on, and should be displayed on the user interface. The port to be used has a default value, but may be edited by the user.

The default port values are as follows:

- Robot Simulator: 7720 (up to 7769 for different Robot Simulators)
- World Simulator: 7770 (up to 7799 for different World Simulators)

The Population Manager doesn't need a port as it only functions as a client.

When a program connects to another one – say Population Manager to Robot Simulator, then the client (here: Population Manager) should try to open a TCP/IP connection to the same IP address, using the default port of the target program (here: 7720 for the Robot Simulator). In other words, it is always assumed, that all programs run on the same computer and use their default port values. Surely target IP address and port have to be fully changeable via the user interface.

An extended version of the World Simulator may offer two or more distinct server ports, enabling more than one Robot Simulator to connect to the same virtual world. Using this scenario one can coevolve two kinds of robot at the same time.

2.2 Communication Syntax

Once a connection is established between two programs (client and server), they communicate using 15 simple messages. Since readability has a higher priority than speed of communication, all messages consist of a single line of words, separated by spaces and terminated with carriage return and line feed, as shown below:

```
MessageName Value1 Value2 ... Valuen CrLf
```

where

MessageName	is a string of characters, only using A..Z and a..z,
Value _i	are numerical values, only using 0..9, +, -, E, e, and ".", (valid examples are: 0, +1, -0.3, 12.5E-2, -711e3)
CrLf	is carriage return and line feed.

Normally the client sends a single message to the server and gets a single message back.

Note: For the sake of readability, the CrLf is discarded in the following message descriptions, but surely every line needs to be terminated by a CrLf in real life.

3 Description of Messages

The following two sections describe all messages in full detail, separated into the messages that are interchanged between Population Manager and Robot Simulator on one side and between Robot Simulator and World Simulator on the other side. Only the Robot Simulator needs to understand all messages.

The colors of the message descriptions correspond to the categories shown in the chart on the first page of this document.

3.1 Population Manager/Robot Simulator

The first message in every evolutionary run is sent from the Population Manager to the Robot Simulator (after the Start-Button has been pressed) in order to identify the possible structures of artificial networks, that the Robot Simulator can handle.

SendNetParameters

Note that this message has no arguments. The Robot Simulator answers with a whole bunch of parameters as shown below.

NetParameters Cci Cgp Lgp Hgp Cin Vin Lin Hin Con Von Lon Hon
CIhn CAhn Vhn Lhn Hhn Vw Lw Hw Fio Fre

Remember that all values are separated by single spaces. Surely the message consists of a single line being too large to be printed as a whole in this document. The parameters have the following meanings:

Cci	Maximum count of concurrent individuals
Cgp	Count of global parameters g_n , $Cgp \geq 0$, ($1 \leq n \leq Cgp$ for $Cgp > 0$)
Lgp, Hgp	Lowest and highest value: $Lgp \leq g_n \leq Hgp$
Cin	Count of input neurons
Vin	Count of parameter values per input neuron
Lin, Hin	Lowest and highest value for parameters of input neurons
Con...Hon	Same as Cin...Hin, but for output neurons
CIhn, CAhn	Minimum and maximum count of hidden neurons
Vhn...Hhn	Same as Vin...Hin, but for hidden neurons
Vw	Count of param. values for a neuron-neuron interconnection (weight)
Lw...Hw	Same as Lin...Hin, but for weights
Fio	0/1-flag, if 0: all weights from input to output neurons are nulled
Fre	0/1-flag, if 0: all recurrent connections are nulled

Thus, a simple recurrent network with two input and two output neurons, no hidden neurons, no global parameters and a single value between -10 and 10 per weight would need the following message:

NetParameters 1 0 0 0 2 0 0 0 2 0 0 0 0 0 0 0 0 1 -10 10 1 1

For a better readability the corresponding values for global parameters, input/output/hidden neurons, and weights are shaded grey.

Having this structural description, the Population Manager is able to initialize the first generation of individual genotypes using random values. Afterwards, up to C_{ci} (see above) individual genotypes are sent to the Robot Simulator, awaiting the corresponding fitness values in return. As soon as a fitness value is received, the Population Manager sends out the next genotype, keeping the value of concurrent tested individuals up at C_{ci} as long as all genotypes of the current generation are tested. Then the Population Manager generates the next generation and the game starts all over.

The individual genotypes are sent as follows:

Individual Gen Ind Chn $g_1 \dots g_{C_{gp}}$ $i_1 \dots i_{N_i}$ $o_1 \dots o_{N_o}$ $h_1 \dots h_{N_h}$ $w_1 \dots w_{N_w}$

With these values:

Gen	Number of current generation
Ind	Number of this individual
Chn	Count of hidden neurons, $C_{Ihn} \leq Chn \leq C_{Ahn}$ (see NetParameters)
g_n	Global parameter values, $1 \leq n \leq C_{gp}$
i_n	Input neuron parameter values, $1 \leq n \leq N_i$, $N_i = C_{in} \cdot V_{in}$
o_n	Output neuron parameter values, $1 \leq n \leq N_o$, $N_o = C_{on} \cdot V_{on}$
h_n	hidden neuron parameter values, $1 \leq n \leq N_h$, $N_h = Chn \cdot V_{hn}$
w_n	weight values, $1 \leq n \leq N_w$, $N_w = (C_{in} + Chn + C_{on}) \cdot (Chn + C_{on}) \cdot V_w$

If V_{in} , V_{on} , V_{hn} or V_w are greater than one, then first all values corresponding to one neuron or weight are transmitted, afterwards the values of the next neuron or weight and so on. The weights are transmitted line by line of the weight matrix W , where:

$$y_t = F(W \cdot x_{t-1})$$

y_t $(C_{on} + Chn) \times 1$ -vector of output and hidden neurons' output at time t , starting with output neurons, then hidden neurons

x_{t-1} $(C_{in} + C_{on} + Chn) \times 1$ -vector of input, output, and hidden neurons at time $t-1$, starting with input neurons, then output neurons, finally hidden neurons

$F(\cdot)$ component-wise transfer function (for example: \tanh)

Note: It is not necessary for the Population Manager to use the structural information about the weight matrix. One can seamlessly transmit Nw values in the correct range. Nevertheless it may be helpful to use this information whilst creating genetic algorithms, that can handle genotypes of different lengths in the same population.

So the very first Individual message of the above example (recurrent network with two input and two output neurons) could read like this:

```
Individual 1 1 0 0.3 -1.5 2.7 -0.4 2.1 0.1 -3.6 -0.9
```

Again, for a better readability, the eight values for the weights are shaded grey.

After the Robot Simulator has translated the individuals genotype into a phenotype, and tested it together with the World Simulator for a while, it sends back the fitness value:

```
Fitness Gen Ind Fit
```

Where:

Gen	Number of current generation, as received with message Individual
Ind	Number of this individual, as received with message Individual
Fit	Fitness value of this individual

The Individual/Fitness message pair is exchanged until the user stops the evolutionary run on the Population Manager.

3.2 Robot Simulator/World Simulator

In order to calculate a fitness value, the Robot Simulator has to place a model of some physical robot into a surrounding world and evaluate the robot-environment interaction. Maybe there coexist several robots in the simulated world, so each robot needs an unique identifier. Therefore the first pair of messages between the Robot Simulator and the World Simulator just creates this identifier for a single instance of the robot:

```
CreateRobot
```

The World Simulator answers:

```
YourIdentifier ID
```

Where:

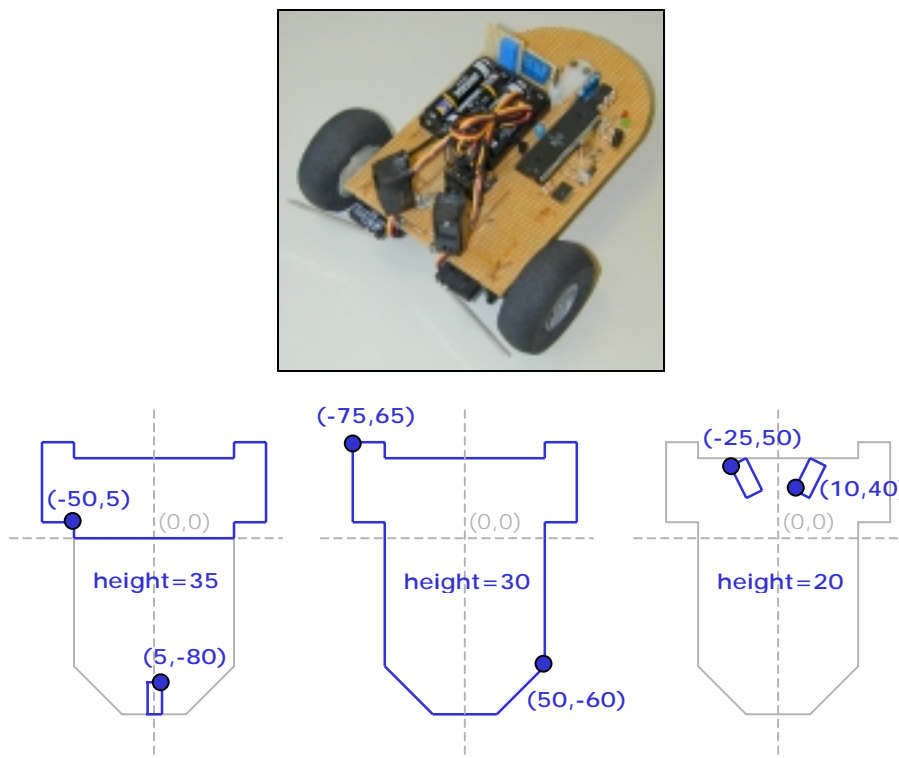
ID	Unique identifier of the created robot instance ($1 \leq ID \leq 2^{16}-1$)
----	---

At the end of the simulation run of this robot, the Robot Simulator has to signal the end of the robot's lifespan to the World Simulator, again using this same ID:

KillRobot ID

The World Simulator will not acknowledge this by another message. Despite of recognizing the Kill message, the World Simulator may additionally have implemented a time-out mechanism, that will remove the robot from the world, if there haven't been any messages with the corresponding ID for a long time.

After the Robot Simulator got the ID, the robots physical shape has to be told to the World Simulator. The robot is represented by one or more layers of (generally) different shapes and height, starting from bottom, ending with the topmost layer. The layers are aligned by the centre of gravity, which defines the point (0,0). All measures are in millimeters. Here is an example (Fahrende Platine):



Each layer consists of one or more closed polygons of the same height. The Robot Simulator issues the following message:

BodyShape ID L H₁ P₁ n₁₁ x y ... x y n₁₂ ... H₂ P₂ ...

Where :

- ID Identifier of robot instance
- L Number of layers
- H_i Height of layer I, 1 ≤ I ≤ L

P_l	Number of polygons per layer l , $1 \leq l \leq L$
n_{lp}	Number of points per polygon p in layer P_l , $1 \leq p \leq P_l$
$x \ y$	Point (x,y) of polygon p in layer l

Thus, in our example we would expect this message (the coordinates of the points are left out for a better readability):

```
BodyShape 1 3 35 2 12 ...xy... 4 ...xy... 30 1 14 ...xy... 20 2 4 ...xy... 4 ...xy...
```

The closed polygons are shaded grey, starting with the number of points per polygon, for example twelve points for the first polygon in the first layer (bottom layer). A very simple robot with the shape of a rectangle reads as follows (ID = 17 in this example):

```
BodyShape 17 1 20 1 4 -30 40 30 40 30 -40 -30 -40
```

The corresponding robot would be 6 x 8 square centimeters and have a height of two centimeters. The supposed normal direction of travel is along the positive y-axis.

Note: Normally the BodyShape message is sent only once per robot instance, but the World Simulator should be prepared to receive further BodyShape messages during the robot's lifetime. This makes sense, if the robot has a pivoting head or gripper.

Having received the BodyShape message, the World Simulator places the robot at a random position within the simulated world, making sure, that there is no collision between robot and environment. If the World Simulator already got a BodyShape message before, then there exists already a valid position for the robot. The World Simulator then tries to leave the robot where it is and corrects the position in the most minimal way if necessary.

After positioning or repositioning, the World Simulator sends the following message back to the Robot Simulator:

```
YourPosition ID X Y Z Dir TiltS TiltC Stress Death
```

Where:

ID	Identifier of robot instance
X Y Z	Actual position of robot in millimeters (Z is height above ground zero)
Dir	Direction of travel (positive y-axis of robot's body shape)
TiltS	Tilt straightforward, $-\pi/2 < \text{TiltS} < \pi/2$ (positive means uphill)
TiltC	Tilt across left-right, $-\pi/2 < \text{TiltC} < \pi/2$ (positive means left side up)
Stress	0 normally, 1 if robot fully blocked, inbetween: see MoveTo message
Death	0/1-Flag, 0 normally, 1 if robot's bottom shape is on deathly ground

The valid operating range for the robot's position is as follows:

$0 < X, Y < 2000$ so we have an arena of 2 x 2 square meters
 $-2000 < Z < 2000$ with $Z=0$ being ground zero.

Sometimes it is necessary to start all robots of the same generation at the same point. This can easily be done using the JumpTo message:

`JumpTo ID X Y Dir`

Where:

ID	Identifier of robot instance
X Y	Desired absolute position of robot in millimeters
Dir	Desired orientation of robot (positive y-axis of robot's body shape)

Again, the World Simulator answers with a YourPosition message (see above). If the target position has enough space for the robot, then X, Y, and Dir of the YourPosition message define the target position, with Stress=0. Otherwise the returned values are the old coordinates with Stress=1.

Now, that the Robot Simulator knows the absolute position, direction, and tilt of the robot, the environment can be sensed by multiple sensor readings. As the positions of the robot's sensors are unknown to the World Simulator, the Robot Simulator needs to calculate the absolute positions and directions of all sensors and ask the World Simulator for the corresponding sensor readings. There are two reasons for using absolute positions and letting the Robot Simulator maintain the bookkeeping of sensor positions: in this way it is possible for the robot to switch between local sensors that are mounted to the robots body and global sensors, that may be mounted to the world itself. In addition it is easy for the robot, to simulate sensors mounted onto a pivoting head. The robot may even stay in place and just move sensors, which may be useful for active vision.

The first pair of messages deals with visual sensors:

`VisualSensor ID X Y Z Dir Tilt AngH AngV`

Where:

ID	Identifier of robot instance
X Y Z	Sensor position (absolute, in millimeters)
Dir	Direction of sensor beam (absolute, not relative to robot body)
Tilt	Tilt up/down, $-\pi/2 < \text{Tilt} < \pi/2$ (absolute, positive means tilt up)
AngH	Horizontal opening angle of sensor beam, $0 \leq \text{AngH} \leq \pi/2$
AngV	Vertical opening angle of sensor beam, $0 \leq \text{AngV} \leq \pi/2$

The World Simulator returns distance and color of the nearest object within the sensor beam, using the following message:

VisualSensorReading ID Dist $C_R C_G C_B$

Where:

ID Identifier of robot instance
Dist Distance to nearest object in millimeters
 $C_R C_G C_B$ Red, green, and blue components of sensed color, $0 \leq C_R, C_G, C_B \leq 255$

For distance measurements using sonic waves (for example ultrasonic transducers), there exists a second pair of messages which are nearly identical to the visual sensor messages, except that the color components are missing:

SonicSensor ID X Y Z Dir Tilt AngH AngV

Where:

ID Identifier of robot instance
X Y Z Sensor position (absolute, in millimeters)
Dir Direction of sensor beam (absolute, not relative to robot body)
Tilt Tilt up/down, $-\pi/2 < \text{Tilt} < \pi/2$ (absolute, positive means tilt up)
AngH Horizontal opening angle of sensor beam, $0 \leq \text{AngH} \leq \pi/2$
AngV Vertical opening angle of sensor beam, $0 \leq \text{AngV} \leq \pi/2$

Again, the World Simulator returns the distance of the nearest object within the sensor beam, using the following message:

SonicSensorReading ID Dist

Where:

ID Identifier of robot instance
Dist Distance to nearest object in millimeters

Note: Using the appropriate combination of sensors, it is possible to distinguish between materials like glass, cotton, and ingrain wallpaper.

Having sensed the environment, the Robot Simulator updates its internal state (for example: recurrent neural net) and finally tries to move to a new position.

MoveTo ID X Y Dir

Where:

ID	Identifier of robot instance
X Y	Desired absolute position of robot in millimeters
Dir	Desired orientation of robot (positive y-axis of robot's body shape)

The World Simulator answers with a YourPosition message (see above for detailed explanation).

If there haven't been any obstacles, the robot fully reaches it's desired position with Stress=0. If there is an obstacle halfway to the desired position, then the World Simulator will position the robot in front of the obstacle, giving back the actually reached position with Stress=0.5 as an indicator, that the robot drove against the obstacle half of the time.

Using this value, the Robot Simulator is able to simulate tactile sensors: the robot bumped against an obstacle, if Stress>0.

Even the amount of power consumption can be estimated, using the difference of the Z-coordinates before and after the move, together with the Stress value and the distance travelled.

Finally the next simulation loop starts all over, using the last messages again and again.