Network Protocol

From Simspark

Contents

Message Formats

Messages from and to the server use S-expressions (short for symbolic expressions) as their basic data structure. The basic idea of S-expressions is very simple: they are either strings, or lists of simpler S-expressions. They are probably best known for their use in the Lisp family of programming languages where they are used for both code and data.

An advantage of using S-expressions over other data formats is that it provides an easy to parse and compact syntax that is to some extent still readable by humans for debug purposes. It is further easy to add new sensors to the messages as the parser on the client side can easily ignore unknown parts.

Messages exchanged between client and server use the default ASCII character set, i.e. one character is encoded in a single byte. Further each individual message is prefixed with the length of the payload message. The length prefix is a 32 bit unsigned integer in

network order, i.e. big endian notation with the most significant bits transferred first.

- S-expressions on Wikipedia (http://en.wikipedia.org/wiki/S-expression)

Server/Agent Communication

The server exposes a network interface to all agents, on TCP port 3100 by default.

When an agent connects to the server the agent must first send a CreateEffector message followed by a InitEffector message.

Once established, the server sends groups of messages to the agent that contain the output of the agent's perceptors, including any hinge positions of the model, any heard messages, seen objects, etc. The exact messages sent depend upon the model created for the agent. Details of effector messages are given on the perceptors page.

In response to these perceptor messages, the agent may influence the simulation by sending effector messages. These perform tasks such as moving hinges in the model. Details of effector messages are given on the effectors page.

Server/Monitor Communication

The server exposes a network interface to any monitors, on TCP port 3200 by default.

This interface allows external processes to be periodically notified of the simulation's state for purposes of visualisation, logging, etc. In addition, the connected process may send various commands to the server for purposes of training or machine learning. These commands are not available to agents directly, nor are they allowed during competitive soccer simulation.

When a monitor connects to the Simspark server using the monitor protocol, the information arrives in the following order:

1. The server sends an environment information message followed by the full scene graph.
2. The server sends a game state message followed by the full scene graph.
3. The server keeps sending partial game state messages followed by a full or partial scene graph, depending on what has been updated lately. The rate at which the server sends messages by the monitor port is defined in the file `spark.rb` which is located in the installation directory.

If you want to see the full output, view the Sample Monitor Messages.

Note that much of the information in this section comes from a paper (http://jeap-res.ams.eng.osaka-u.ac.jp/~joschka/simspark/monitorprotocol.pdf) written by Carlos Bustamante with Markus Rollmann and Joschka Boedecker in 2008. Additional information was obtained by observation of the source code in `trainercommandparser.cpp`.

Environment Information Message

The environment information message has the following structure:

```
((<EnvironmentInformation>)(<SceneGraphHeader>)(<SceneGraph>))
```

Here is an example of the EnvironmentInformation part, as provided by rcssserver3d 0.6.3:

```
((FieldLength 18)(FieldWidth 12)(FieldHeight 40)
 (GoalWidth 2.1)(GoalDepth 0.6)(GoalHeight 0.8)
 (FreeKickDistance 1.3)(WaitBeforeKickOff 2)
 (AgentRadius 0.4)(BallRadius 0.042)(BallMass 0.026)
 (RuleGoalPauseTime 3)(RuleKickInPauseTime 1)(RuleHalfTime 300)
 (play_modes BeforeKickOff KickOff_Left KickOff_Right PlayOn
             KickIn_Left KickIn_Right corner_kick_left corner_kick_right
             goal_kick_left goal_kick_right offside_left offside_right
             GameOver Goal_Left Goal_Right free_kick_left free_kick_right)
)
```

Each subexpression of the init expression is a name value pair that gives one parameter that the current instance of the simulation uses. The meaning of the different parameters:

- FieldLength, FieldWidth, FieldHeight dimensions of the soccer field in meters
- GoalWidth, GoalDepth, GoalHeight dimensions of the goals in meters
- FreeKickDistance gives the distance in meters that agents of the opposite have to adhere when a player carries out a free kick.
- WaitBeforeKickOff gives the time in seconds the server waits before automatically starting the game
- AgentRadius the radius of each agent in metres
- BallRadius the radius if the ball in metres
- BallMass the mass of the ball in kg
- RuleGoalPauseTime the time in seconds that the server waits after a goal is scored before switching to kick off playmode
- RuleKickInPauseTime the time in seconds that the server waits after the ball left the field before switching to the kick in playmode
- RuleHalfTime the length of one half time in seconds
- play_modes lists the the different play_modes of the soccer simulation. Later on, play_modes are referenced by a zero based index into this list.

GameState Message

The environment information message has the following structure:

```
[GameState] [SceneGraphHeader] [SceneGraph]
```

Here is an example of the GameState part:

```
((time 0)(half 1)(score_left 0)(score_right 0)(play_mode 0))
```

A partial GameState would look like this:

```
((time 0))
```

The play mode is currently an integer which represents a certain type defined in plugin/soccer/soccertypes.h in the TPlayMode enum.

Scene Graph Header

The scene graph header contains information about the completeness of the scene graph and the version number, and has the following form:

```
(Name Version Subversion)
```

- Name Can take two values:
  - RSG Stands for Ruby Scene Graph, and indicates that the scene graph is a full description of the environment.
  - RDS Stands for Ruby Diff Scene, and indicates that the scene graph is a partial description of the environment, i.e. it contains a bunch of empty nodes representing the structure of the scene graph, and some updated information about the nodes who have changed lately.
- Version The main version number of the scene graph.
- Subversion The subversion number.

Note that across different server versions the message format has changed although the version number used here has not (as of server 0.6.3)

For example:

- `(RSG 0 1)` indicates that the scene graph is full with version 0.1
- `(RDS 0 1)` indicates that the scene graph is partial with version 0.1

There are two cases in which the full state message is generated:

1. When a new client connects (`SparkMonitor::GetMonitorHeaderInfo`)
2. When one or more nodes are added or removed (e.g. on scene import, on agent connect/disconnect, etc.)

Here is an example of a scene graph header, as returned by rcssserver3d version 0.6.3:

```
(RSG 0 1)
```

Scene Graph Contents

The scene graph is a structure that arranges the logical and often spatial representation of a graphical scene. In Simspark, the scene graph is a tree with a root node defined to be at the origin < 0; 0; 0 > without rotation. Each node has one or more children. The position and rotation of each child is the multiplication of the transformation matrices from the root node down to the child node.

The monitor does not care about the specific objects (whether it's a goal or a ball). It knows about Meshes, i.e. objects that are loaded from files or built into the simulator. Currently, Simspark uses files from the Blender 3d modeler (http://www.blender.org/) for which there is an importer (see `plugin/objimporter`) and some built-in objects (sphere, box, and cylinder) that are constructed in the `stdmeshimporter`. The monitor protocol implementation is defined in `plugin/sparkmonitor`.

The current monitor protocol is a shortened version of the graph RubySceneGraph scene description language (http://en.wikipedia.org/wiki/Scene) . It consists of several abbreviations instead of the full node names. This was done in order to save bandwidth.

In the following sections, each node type is briefly described. Every non-leaf node is defined with nd which is an abbreviation for node. For more information about abbreviations, see abbreviations.

It is important to mention that when the message represents an update to the scene graph, the scene graph omits node types and node information. In other words, the structure remains unchanged, but the message will be full of empty nd nodes, except for those nodes which are being updated.

Base Node

Every node type which is not a transform node, a static mesh node or a light node, is considered a base node (including the root node). Base nodes do not have a description in the message, they are simply used for preserving the structure of the scene graph. They can contain any number of child nodes. A base node has the following form:

```
(nd BN <contents>)
```

Transform Node

A transform node represents a 4x4 homogeneous transformation matrix which is commonly used in robotics and computational geometry for representing geometric transformations. A matrix transformation matrix (http://en.wikipedia.org/wiki/Transformation) defines how to map points from one coordinate space into another coordinate space, by defining a translation, a rotation and a scaling factor. The scaling factor is considered 1 for most applications.

The typical form of a homogeneous transformation matrix is as follows:

```
[nx ox ax Px]
[ny oy ay Px]
[nz oz az Pz]
[ 0  0  0  1]
```

where the unit vectors n; o; a stand for normal, orientation and approach, respectively, and refer to the three vectors that represent a reference frame in the three dimensional space.

A transform node in Simspark representing the aforementioned transformation matrix is as follows:

```
(nd TRF (SLT nx ny nz 0 ox oy oz 0 ax ay az 0 Px Py Pz 1 ))
```

Where SLT represents a SetLocalTransform leaf node, which is a ruby function for setting the local transformation of a given node in the scene graph.

Geometry Nodes

The following two node types describe object shapes. They specify its scale and its material (texture). They have no child nodes (they are leaves).

StaticMesh

Defines a mesh to be loaded from a .obj file:

```
(nd StaticMesh (load <model>) (sSc <x> <y> <z>)
  (setVisible 1)
  (setTransparent)
  (resetMaterials <material-list>)
)
```

- model is the path to the .obj file
- sSc defines the scale of the object
- setVisible (optional) carries a bit flag that indicates whether the object is visible or not
- setTransparent (optional) TODO what is the difference between this and not visible?
- resetMaterials defines the list of materials used in the associated .obj file

Examples:

```
(nd StaticMesh (load models/rlowerarm.obj) (sSc 0.05 0.05 0.05)(resetMaterials matLeft naowhite))
(nd StaticMesh (load models/naohead.obj) (sSc 0.1 0.1 0.1)(resetMaterials matLeft naoblack naogrey naowhite))
```

SMN

Defines a predefined mesh type:

```
(nd SMN (load <type> <params>) (sSc <x> <y> <z>)
  (setVisible 1)
  (setTransparent)
  (sMat <material-name>)
)
```

- type must be one of the following:
  - StdUnitBox
  - StdUnitCylinder where <params> holds two values: length and radius
  - StdUnitSphere
  - StdCapsule with <params (not used in rcssserver3d 0.6.3, but available for other simspark simulations) TODO what are the params?

- sSc defines the scale of the object
- setVisible (optional) carries a bit flag that indicates whether the object is visible or not
- setTransparent (optional) TODO what is the difference between this and not visible?
- sMat specifies the name of the material to be applied to the object, e.g. matWhite, matYellow, etc.

Examples:

```
(nd SMN (load StdUnitBox) (sSc 1 31 1) (sMat matGrey))
(nd SMN (load StdUnitCylinder 0.015 0.08) (sSc 1 1 1) (sMat matDarkGrey))
```

Light Node

A light node describes the way in which lights affect the object. Specifically, the diffuse, ambient and specular lights are defined. It has the following form:

```
(nd Light (setDiffuse x y z w) (setAmbient x y z w) (setSpecular x y z w))
```

where < x; y; z > is a vector defining the light direction and w is a scaling factor. Each of the leaf nodes is described below:

- setDiffuse Diffuse reflection (http://en.wikipedia.org/wiki/Diffuse_reflection) is the reflection of light from an uneven or granular surface such that an incident ray is seemingly reflected at a number of angles. It is the complement to specular reflection.
- setAmbient Ambient light (http://en.wikipedia.org/wiki/Ambient_light) (also available light or existing light) refers to the illumination surrounding a subject or scene.
- setSpecular Specular reflection (http://en.wikipedia.org/wiki/Specular_reflection) is the perfect, mirror-like reflection of light (or sometimes other kinds of wave) from a surface, in which light from a single incoming direction (a ray) is reflected into a single outgoing direction.

Monitor Protocol Abbreviations

The monitor protocol uses abbreviations of terms for describing the environment, with the objective of saving bandwidth when sending messages to the connected clients. Here is a list of the meanings of the most important abbreviations:

- nd Node
- BN Base Node
- TRF Transform Node
- SLT SetLocalTransform Leaf Node
- SMN StaticMesh Node
- sSc SetScale Leaf Node
- sMat SetMaterial Leaf Node
- RSG RubySceneGraph Node : Full State
- RDS RubySceneGraph Node : Update

Command Messages from Coach/Trainer

A monitor may be used by a referee during a competition to enforce rules of the game that are not currently catered for by the simulation. Additionally, machine learning processes may send commands to influence the environment in special ways to improve the speed or quality of learning exercises.

When debugging an application that sends these messages to the server, check the server console or log file for error messages. Errors are not returned to the caller.

Note that sending a command may have no effect in certain play modes. For example, moving the ball away from the center position in `BeforeKickOff` mode is immediately counteracted by the simulator.

Moving an Agent

You can either move an agent, or move and rotate it. Additionally, you can set the battery level and temperature.

```
(agent (unum <num>) (team <team>) (pos <x> <y> <z>)
                                  (move <x> <y> <z> <rot>)
```

```
                                (battery <batterylevel>)
                                (temperature <temperature>)
)
```

- <team> is one of Left, Right.
- unum and team must both be specified, but the remainder are optional. Zero or more operations may be applied to the specified agent via this command.
- x, y, z are absolute values in field coordinates. That is, (0,0) is the centre of the field.

- TODO what units are <rot> in? again, relative or absolute?

Positioning the Ball

The ball may be repositioned and optionally given an explicit velocity as well.

```
(ball (pos <x> <y> <z>)
      (vel <x> <y> <z>)
)
```

pos and vel are both optional, though if neither are included, then no change is made. Note that setting the velocity of the ball will also reset any angular velocity of the ball to zero.

Setting the Play Mode

The play mode may be manually set to an explicit value.

```
(playMode <playmode>)
```

Where <playmode> is one of the predefined, case sensitive, play mode values. Possible play modes are given as strings in the play_modes expression of the init expression the monitor receives when it connects.

TODO link to these values as they're elsewhere on this wiki

Drop the Ball

Drops the ball at its current position and move all players away by the free kick radius. If the ball is off the field, it is brought back within bounds.

```
(dropBall)
```

This operation has no arguments.

Kick Off

Assigns a soccer kickoff to the specified team.

```
(kickOff <team>)
```

Where <team> is one of Left, Right, None. If None is passed, a coin is tossed to select a team.

Select Agent

```
(select (unum <num>) (team <team>))
```

Where <team> is one of Left, Right, None.

If unum and/or <team> are not specified, then the next agent is selected.

Only one agent may be selected at a time. TODO confirm this.

Selection can be used with the kill and reposition commands.

Kill Agent

Removes the specified agent from the simulation.

```
(kill (unum <num>) (team <team>))
```

Where <team> is one of Left, Right, None.

If unum and/or <team> are not specified, then the selected agent is killed.

Repositioning an Agent

Repositions the specified agent to a calculated location. TODO explain how this new location is calculated, and under what circumstances in a game this would be used.

```
(repos (unum <num>) (team <team>))
```

Where <team> is one of Left, Right, None.

If unum and/or <team> are not specified, then the selected agent is repositioned.

Ack

```
(getAck <cookie string>):
```

Experimental feature, currently disabled. Requests an ack reply from the server. The server will send the answer as soon as the command is carried out. This is used to synchronize a trainer implementation with the server. The getAck expression is appended behind one of the above commands.

For example:

```
((ball (pos 0 0 50))(getAck moved_ball_in_the_air))
```

The server returns:

```
(ack <cookie>)
```

Setting the Time

Sets the game time.

```
(time <time>)
```

Setting the Score

Sets the score of the game.

```
(score (left <score>) (right <score>))
```

Request Full State Message

Requests a full scene update message from the server.

```
(reqfullstate)
```