Scalatron Protocol

About Scalatron

Scalatron is an educational resource for groups of programmers that want to learn more about the Scala programming language or want to hone their Scala programming skills. It is based on Scalatron BotWar, a competitive multi-player programming game in which coders pit bot programs (written in Scala) against each other.

The documentation, tutorial and source code are intended as a community resource. Feel free to use, copy, and improve them!

Summary

The behavior of each bot is determined by an associated computer program fragment that implements a control function mapping inputs (what the bot sees) to outputs (how the bot responds).

This document describes how the game server interacts with plug-ins and lists all the commands that the game server or a plug-in may send or receive.

Table of Contents

Plug-in Mechanics

Plug-in Locations

The Scalatron server is configured to use a specific shared directory as its plug-in base directory. The server will load modified plug-ins from user directories below that directory each time a new game round in the tournament loop starts. On the server locally, this directory may for example be located at `/Users/Scalatron/Scalatron/bots`. For more details, see the Server Setup guide.

Below the plug-in base directory the server expects a collection of sub-directories, with one sub-directory per participant, bearing the participant's name. Example:

```
/Users/Scalatron/Scalatron/bots/         -- the plug-in base directory
/Users/Scalatron/Scalatron/bots/Daniel   -- plug-in directory of user Daniel
/Users/Scalatron/Scalatron/bots/Frank    -- plug-in directory of user Frank
```

The server will traverse all these sub-directories, attempt to load a plug-in from each one and then try to extract a control function factory from the plug-in. The server recognizes plug-ins by their file name; every plug-in must have the name `ScalatronBot.jar`.

So the overally layout on disk for an example setup with two plug-ins for users called Daniel and Frank would be as follows (on MacOS X):

```
/Users                       -- parent directory of user directories
  /Scalatron                 -- user directory of user 'Scalatron'
    /Scalatron               -- installation directory (from ZIP)
      /bots                  -- plug-in base directory
        /Daniel              -- plug-in directory of user Daniel
          ScalatronBot.jar   -- plug-in .jar file w/ compiled bot
        /Frank               -- plug-in directory of user Frank
          ScalatronBot.jar   -- plug-in .jar file w/ compiled bot
```

Publishing Plug-ins

The server will automatically pick up any plug-in located in the appropriate location, i.e. in a sub-directory below the plug-in base directory. To "publish" a plug-in it is therefore sufficient to place your `ScalatronBot.jar` file into the appropriate directory.

In practice, there are two primary ways to "publish" a bot plug-in .jar file into the plug-in directory:

- The most straight-forward approach, used in the "serious" bot development path, is to simply copy the `.jar` file into the appropriate plug-in directory after building it on a client machine. Naturally, the plug-in directory must be shared and accessible over the network.
- A more indirect approach, used in the "casual" bot development path, is via the browser interface served by the Scalatron server. After typing or pasting code into the browser-based editor, a participant can push a button to have the code uploaded to the server, compiled there, zipped into a .jar file and then copied into the appropriate plug-in directory. No network share is required.

As outlined in more detail in the Server Setup guide, the network share will have to be mounted (made accessible) locally on the computer of every participant that wishes to follow the "serious" path (build and test locally, then copy files into the server's plug-in directory). On each local machine, the directory structure outlined above will appear below the mount point of the network share.

If, for example, the plug-in base directory on the server that was shared was `/Users/Scalatron/Scalatron/bots` and the local mount point (e.g. on MacOS X) was `/Volumes/Scalatron/bots/`, then the plug-in directory structure would locally appear as follows:

```
/Volumes/Scalatron/bots/          -- the plug-in base directory
/Volumes/Scalatron/bots/Daniel   -- plug-in directory of user Daniel
/Volumes/Scalatron/bots/Frank    -- plug-in directory of user Frank
```

If you are following the "serious" development path and wish to copy your plug-in into the appropriate plug-in directory on the server manually, ask your server administrator for access to that directory if you do not have it already.

Providing a Control Function Factory

Overview

The Scalatron server interacts with bots by calling their control function. The control function is a simple Scala function that expects a `String` as its input and returns a new `String` as its output. The server allows control functions to hold persistent state (although this is not necessarily recommended, see the Tutorial for details), so each game round in the tournament loop uses a new control function instance.

To obtain a control function instance, the server requires a control function factory, which is a Scala class that exposes a single function that returns a control function instance. The server gains access to the control function factory by loading the plug-in `.jar` file and looking for a class with a special name.

Details

Each plug-in `.jar` file must contain a class with the name `ControlFunctionFactory` with a single function with the name `create` that takes no parameters and returns a function that receives and returns a `String`:

```
class ControlFunctionFactory {
    def create : (String => String) = { ... }
}
```

This class is by default expected to reside in a package called `scalatron.botwar.botPlugin`, i.e. be present in the compiled `.jar` file with the fully qualified name of `scalatron.botwar.botPlugin.ControlFunctionFactory`. This harmonizes the symbols of the plug-ins with those of the server.

However, for simplicity's sake, the package name can also be omitted if the class is only ever loaded from a plug-in `.jar` file. The bot programming tutorial uses this option to omit the package specification and make the example bots a bit simpler. The bots coded according to the tutorial are therefore present in the compiled `.jar` file with a fully qualified name of just `ControlFunctionFactory`. This works because each plug-in is loaded using a dedicated `URLClassLoader` instance that resolves types within each .jar first.

The Scalatron plug-in loader, after "mounting" a plug-in `.jar`, first tries to instantiate the control function factory via the recommended fully qualified name. If no class is found for the fully qualified name, the plug-in loader next attempts to instantiate the control function factory using an empty package name. This is why the example bots used in the tutorial, which omit a package specification, work.

At the start of each game, the server will invoke the `ControlFunctionFactory.create()` function once. It should create an instance of a control function instance and return it to the server:

```
class ControlFunctionFactory {
    def create = (input: String) => "Move(direction=1:1)"
}
```

For more details, see the Scalatron Tutorial.

The goal of this somewhat unusual setup was to make the bots as simple as possible to create. Beyond the naming convention for the factory class, there is no coupling between the types of the server and the plug-in at all and very few things that can go wrong in setting up a new player.

Control Function Protocol

As outlined above, the Scalatron server uses invocations of a plug-in-supplied control function to determine the behavior of players' bots. The control function is retrieved by the server by invoking a creation method on a factory class. The server uses the same control function instance for the duration of each game round and requests a new instance from the (potentially reloaded) plugin at the start of every new game round. In general the returned function will therefore be a method of a class instance which can hold the state of the bot(s) controlled by it.

The control function has the signature `String => String`, i.e. it accepts a `String` and returns a `String`. Here is an example implementation of a complete bot:

```
class ControlFunctionFactory {
    def create = (input: String) => "Move(direction=1:1)"
}
```

Every input string contains one and every output string zero or more parameterized commands. The string has the following format:

```
"Opcode(key=value,key=value,...)|Opcode(...)|..."
```

No spaces will be present between key name, equals sign and value. No pipe characters are permitted anywhere in the string except between commands. The order of the key-value pairs is not fixed.

Example communication between server and plug-in:

Server input:

```
"React(entity=Master,time=100,view=__W_W_W__,energy=100)"
```

Plugin reply:

```
"Move(direction=1:0)|Spawn(direction=1:1,energy=100)|Say(text=Hello)"
```

Bot State

The server maintains state information for all entities, including bots and mini-bots. The state includes a variety of properties which are managed by the server and are read-only, including the bot's position, energy and id.

In addition to these read-only properties, each bot can carry an arbitrary set of user-defined, properties which are writable. These properties can be used to maintain custom state information for each bot or mini-bot without holding mutable state - or, in fact, any state - within the control function.

The custom state properties of each bot are name/value paris stored as strings. Due to the command syntax, certain restrictions apply to the content of these strings. Specifically, they may not contain any of the following characters:

- comma (`,`)
- opening parenthesis (`(`)
- closing parenthesis (`)`)
- equals sign (`=`)
- pipe (`|`)

Certain property names are also reserved by the server because they correspond to built-in, read-only properties and the server needs their names to communicate with the plug-ins. The following property names are reserved and must not be used for custom properties:

- `"name"`
- `"generation"`
- `"energy"`
- `"time"`
- `"view"`
- `"direction"`
- `"master"`
- `"collision"`

Custom state properties cannot have empty strings as their values. Setting a custom state property to an empty string deletes it from the state.

Opcodes of Server-to-Plugin Commands

These are the opcodes valid for commands sent by the server to a plug-in's control function. Only one opcode will be present per control function invocation.

Welcome(name=String,apocalypse=int,round=int)

"Welcome" is the first command sent by the server to a plug-in before any other invocations of the control function.

Parameters:

- `name`: the player name associated with the plug-in. The player name is set based on the name of the directory containing the plug-in.
- `apocalypse`: the number of steps that will be performed in the upcoming game round. This allows bots to plan ahead and to e.g. schedule the recall of harvesting drones. Keep in mind, however, that the control function of master bots is only invoked with `React` every second simulation step! (see the Game Rules for details).
- `round`: the index of the round for which the control function was instantiated. A game server continually runs rounds of the game, and the round index is incremented each time.

React(generation=int,name=string,time=int,view=string,energy=int,master=int:int,collision=int:int)

"React" is invoked by the server once for each entity for each step in which the entity is allowed to move (mini-bots every cycle, bots every second cycle - see the Game Rules for details). The plug-in must return a response for the entity with the given entity name that is appropriate for the given view of the world.

Parameters:

- `generation`: the generation of this bot. The master bot is the only bot of generation 0 (zero); the mini-bots it spawned are of generation 1 (one); the mini-bots spawned by these are generation 2 (two), etc. Use this parameter to distinguish between mini-bots (slaves) and your master bot.
- `name`: the name of the entity. For master bots, this is the name of the player (which in turn is the name of the plug-in directory the bot was loaded from). For mini-bots, this is either the name provided to `Spawn()` or a default name that was auto-generated by the server when the mini-bot was spawned.
- `time`: a non-negative, monotonically increasing integer that represents the simulation time (basically a simulation step counter).
- `view`: the view that the player has of the playing field. The view is a square region containing N*N cells, where N is the width and height of the region. Each cell is represented as a single ASCII character. The meaning of the characters is defined in the table "View/Cell Encoding".
- `energy`: the entity's current energy level
- `master`: for mini-bots only: relative position of the master, in cells, in the format "x:y", e.g. "-1:1". To return to the master, the mini-bot can use this as the move direction (with some obstacle avoidance, of course).

- `collision`:
  if the entity failed to execute a move requested in the previous cycle because a collision with another entity occurred, this parameter is set to the direction of the failed move, e.g. "1:-1" if a move right and up could not be executed. If no collision occurred, this property is not defined.

In addition to these system-generated parameters, the server passes in all state parameters of the entity that were set by the player via `Spawn()` or `Set()` (see below). If, for example, a mini-bot was spawned with `Spawn(...,role=missile)`, the `React` invocation will contain a parameter called `role` with the value `missile`.

The control function is expected to return a valid response, which may consist of zero or more commands separated by a pipe ('|') character. The available commands are listed in the section below, Opcodes of Plugin-to-Server Commands.

Goodbye(energy=int)

"Goodbye" is the last command sent by the server to a plug-in after all other invocations. The plug-in should use this opportunity to close any open files (such as those used for debug logging) and to relinquish control of any other resources it may hold.

Parameters:

- `energy`: the bot's final energy level

Opcodes of Plugin-to-Server Commands

These are the opcodes valid for commands returned by a plug-in to a server. Multiple such commands can be combined into a multi-command by separating them with a pipe ("|") character. However, the server will process only one command instance per opcode; this means you should not return a string containing multiple instances of e.g. `Move`, `Set` or `Log` - only one of them would end up being used.

Simulation-Affecting Commands

These commands, when executed by the server, alter the state of the game and affect the course of the simulation.

Move(direction=int:int)

Moves the bot one cell in a given direction, if possible. The direction value is a pair of signed integers in the range "-1", "0" or "1".

Parameters:

- `direction`: desired displacement for the move, e.g. "1:1" or "0:-1"

Example:

- "`Move(direction=-1:1)`" moves the entity left and down.

Energy Cost/Permissions:

- for master bot: 0 EU (free)
- for mini-bot: 0 EU (free)

Spawn(direction=int:int,name=string,energy=int)

Spawns a mini-bot from the position of the current entity at the given cell position, expressed relative to the current position.

Parameters:

- `direction`: desired displacement for the spawned mini-bot, e.g. "-1:1"
- `energy`: energy budget to transfer to the spawned mini-bot (minimum: 100 EU)
- `name`: arbitrary string, except the following characters are not permitted: '|', ',', '=', '('

Defaults:

- `name` = "Slave_nn" an auto-generated unique slave name
- `energy` = 100 the minimum permissible energy

Additional Parameters:

- In addition to the parameters listed above, the command can contain arbitrary additional parameter key/value pairs. These will be set as the initial state parameters of the entity and will be passed along to all subsequent control function invocations with `React`. This allows a master bot to "program" a mini-bot with arbitrary starting parameters.
- The usual restrictions for strings apply (no comma, equals sign, parentheses or pipe characters).
- The following property names are reserved and must not be used for custom properties: `"generation"`, `"name"`, `"energy"`, `"time"`, `"view"`, `"direction"`, `"master"`, `"collision"`.
- Properties whose values are empty strings are ignored.

Example:

- `"Spawn(direction=-1:1,energy=100)"` spawns a new mini-bot one cell to the left and one cell down, with an initial energy of 100 EU.

Energy Cost/Permissions:

- for master bot: as allocated via `energy` parameter
- for mini-bot: as allocated via `energy` parameter

Set(key=value,...)

Sets one or more state parameters with the given names to the given values. The state parameters of the entity will be passed along to all subsequent control function invocations with `React`. This allows an entity to store state information on the server, making its implementation immutable and delegating state maintenance to the server.

- The usual restrictions for strings apply (no comma, equals sign, parentheses or pipe characters).
- The following property names are reserved and must not be used for custom properties: `"generation"`, `"name"`, `"energy"`, `"time"`, `"view"`, `"direction"`, `"master"`, `"collision"`.
- Properties whose values are empty strings are deleted from the state properties.

Energy Cost/Permissions:

- for master bot: permitted, no energy consumed
- for mini-bot: permitted, no energy consumed

Explode(size=int)

Detonates the mini-bot, dissipating its energy over some blast radius and damaging nearby entities. The mini-bot disappears. Parameters:

- `size`: an integer value 2 < x < 10 indicating the desired blast radius

Energy Cost/Permissions:

- for master bot: cannot explode itself
- for mini-bot: entire stored energy

Simulation-Neutral Commands

These commands, when executed by the server, do not alter the state of the game and do not affect the course of the simulation. They are primarily intended for debugging purposes.

Say(text=string)

Displays a little text bubble that remains at the position where it was created. Use this to drop textual "breadcrumbs" associated with events. You can also use this as a debugging tool. Don't go overboard with this, it'll eventually slow down the gameplay.

Parameters:

- `text`: the message to display; maximum length: 10 chars; can be an arbitrary string, except the following characters are not permitted: '|', ',', '=', '('

Energy Cost/Permissions:

- for master bot: permitted, no energy consumed
- for mini-bot: permitted, no energy consumed

Status(text=string)

Shortcut for setting the state property "status", which displays a little text bubble near the entity which moves around with the entity. Use this to tell spectators about what your bot thinks. You can also use this as a debugging tool. If you return the opcode `Status`, do not also set the `status` property via `Set`, since no particular order of execution is guaranteed.

Parameters:

- `text`: the message to display; maximum length: 20 chars; can be an arbitrary string, except the following characters are not permitted: '|', ',', '=', '('

Energy Cost/Permissions:

- for master bot: permitted, no energy consumed
- for mini-bot: permitted, no energy consumed

MarkCell(position=int:int,color=string)

Displays a cell as marked. You can use this as a debugging tool.

Parameters:

- `position` desired displacement relative to the current bot, e.g. '-2:4' (defaults to '0:0')

- `color` color to use for marking the cell, using HTML color notation, e.g. `#ff8800` (default: `#8888ff`)

Energy Cost/Permissions:

- for master bot: permitted, no energy consumed
- for mini-bot: permitted, no energy consumed

DrawLine(from=int:int,to=int:int,color=string)

Draws a line. You can use this as a debugging tool.

Parameters:

- `from` starting cell of the line to draw, e.g. '-2:4' (defaults to '0:0')
- `to` destination cell of the line to draw, e.g. '3:-2' (defaults to '0:0')
- `color` color to use for marking the cell, using HTML color notation, e.g. `#ff8800` (default: `#8888ff`)

Energy Cost/Permissions:

- for master bot: permitted, no energy consumed
- for mini-bot: permitted, no energy consumed

Log(text=string)

Shortcut for setting the state property 'debug', which by convention contains an optional (multi-line) string with debug information related to the entity that issues this opcode. This text string can be displayed in the browser-based debug window to track what a bot or mini-bot is doing. The debug information is erased each time before the control function is called, so there is no need to set it to an empty string.

Parameters:

- `text`: the debug message to store. The usual restrictions for string values apply (no commas, parentheses, equals signs or pipe characters). Newline characters are permitted, however.

Energy Cost/Permissions:

- for master bot: permitted, no energy consumed
- for mini-bot: permitted, no energy consumed

View/Cell Encoding

The server encodes the view of each bot into an ASCII string, with one ASCII character per cell in the view (see the Scalatron Game Rules for details). Here are the ASCII codes used by the server and what they mean. As a general rule, you can assume that lowercase letters stand for something bad, while uppercase letters stand for something good.

- `?` cell whose content is occluded by a wall
- `_` empty cell (underscore)
- `W` wall
- `M` Bot (=master; yours, always in the center unless seen by a slave)
- `m` Bot (=master; enemy, not you)
- `S` Mini-bot (=slave, yours)
- `s` Mini-bot (=slave; enemy's, not yours)
- `P` Zugar (=good plant, food)
- `p` Toxifera (=bad plant, poisonous)
- `B` Fluppet (=good beast, food)
- `b` Snorg (=bad beast, predator)