Multi-Agent Programming Contest Protocol Description (2013 Edition)

http://www.multiagentcontest.org/2013/

Jürgen Dix   Michael Köster   Federico Schlesinger

June 6, 2013

New in 2013: Differences between the last year and 2013 are now marked with boxes.

Contents

1 General Agent-2-Server Communication Principles

The agents from each participating team will be executed locally (on the participant's hardware) while the simulated environment, in which all agents from competing teams perform actions, runs on the remote contest simulation server.

Agents communicate with the contest server using standard TCP/IP stack with socket session interface. The Internet coordinates (IP address and port) of the contest server (and a dedicated test server) will be announced later via the official contest mailing list.

Agents communicate with the server by exchanging XML messages. Messages are well-formed XML documents, described later in this document. We recommend using standard XML parsers available for many programming languages for generation and processing of these XML messages. Note that ill-formed messages, that is messages that do not comply to the message-syntax outlined here, are ignored.

2 Communication Protocol Overview

Logically, the tournament consists of a number of matches. A match is a sequel of simulations during which several teams of agents compete in several different settings of the environment. However, from agent's point of view, *the tournament consists of a number of simulations in different environment settings and against different opponents.*

The tournament is divided into three phases:

1. the initial phase,
2. the simulation phase, and
3. the final phase.

During the initial phase, agents connect to the simulation server and identify themselves by username and password (`AUTH-REQUEST` message). Credentials for each agent will be distributed in advance via e-mail. As a response, agents receive the result of their authentication request (`AUTH-RESPONSE` message) which can either succeed, or fail. After successful authentication, agents should wait until the first simulation of the tournament starts.

Fig. 1 shows a picture of the initial phase (UML-like notation).

At the beginning of each simulation, agents of the two participating teams are notified (`SIM-START` message) and receive simulation specific information.

In each simulation step each agent receives a perception about its environment (`REQUEST-ACTION` message) and should respond by performing an action (`ACTION` message).

The agent has to deliver its response within the given deadline. The action message has to contain the identifier of the action, the agent wants to perform, and action parameters, if required.

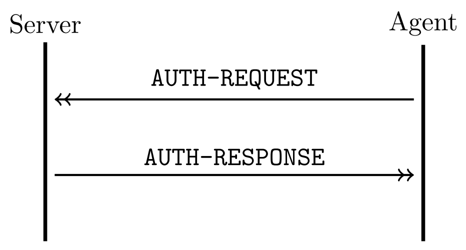

Figure 1: The initial phase.

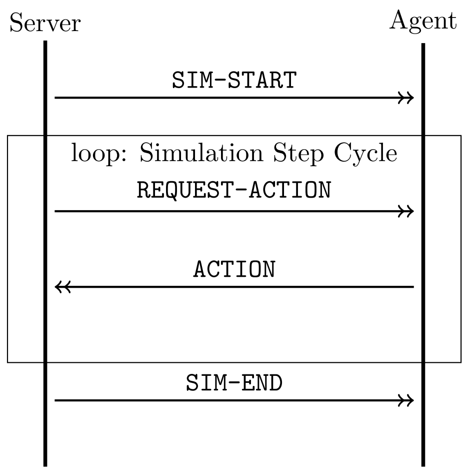

Figure 2: The simulation-phase.

Fig. 2 shows a picture of the simulation phase.

When the simulation is finished, participating agents receive a notification about its end (`SIM-END` message) which includes the outcome of the simulation.

All agents which currently do not participate in a simulation should wait until the simulation server notifies them about either 1) the start of a simulation, they are going to participate in, or 2) the end of the tournament.

At the end of the tournament, all agents receive a notification (`BYE` message). Subsequently the simulation server will terminate the connections to the agents.

Fig. 3 shows a picture of the final phase.

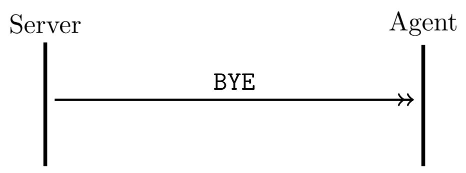

Figure 3: The final phase.

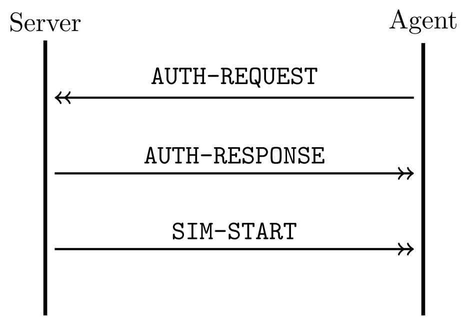

Figure 4: Reconnecting.

2.1 Reconnection

When an agent loses connection to the simulation server, the tournament proceeds without disruption, only all the actions of the disconnected agent are considered to be empty (*skip*). Agents themselves are responsible for maintaining the connection to the simulation server and in a case of connection disruption, they are allowed to reconnect.

Agents reconnect by performing the same sequence of steps as at the beginning of the tournament. After establishing the connection to the simulation server, it sends `AUTH-REQUEST` message and receives `AUTH-RESPONSE`. After successful authentication, the server sends `SIM-START` message to an agent. If an agent participates in a currently running simulation, the `SIM-START` message will be delivered immediately after `AUTH-RESPONSE`. Otherwise an agent will wait until a next simulation in which it participates starts. In the next subsequent step when the agent is picked to perform an action, it receives the standard `REQUEST-ACTION` message containing the perception of the agent at the current simulation step and simulation proceeds in a normal mode.

Fig. 4 shows a picture of the reconnection.

2.2 XML Messages Description

2.2.1 XML message structure

XML messages exchanged between server and agents are zero terminated UTF-8 strings. Each XML message exchanged between the simulation server and agent consists of three parts:

- Standard XML header: Contains the standard XML document header

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  ```

- Message envelope: The root element of all XML messages is `<message>`. It has attributes the timestamp and a message type identifier.
- Message separator: Note that because each message is a UTF-8-zero-terminated string, messages are separated by nullbyte.

The timestamp is a numeric string containing the status of the simulation server's global timer at the time of message creation. The unit of the global timer is milliseconds and it is the result of standard system call "time" on the simulation server (measuring number of milliseconds from January 1st, 1970 UTC). The message type identifier is one of the following values: `auth-request`, `auth-response`, `sim-start`, `sim-end`, `bye`, `request-action`, `action`.

Messages sent from the server to an agent contain all attributes of the root element. However, the timestamp attribute can be omitted in messages sent from an agent to the server. In the case it is included, server silently ignores it.

Example of a server-2-agent message:

```
<message timestamp="10001980000000" type="request-action">
  <!-- optional data -->
</message>
```

Example of an agent-2-server message:

```
<message type="auth-request">
  <!-- optional data -->
</message>
```

Depending on the message type, the root element `<message>` can contain simulation specific data.

2.2.2 `AUTH-REQUEST` (agent-2-server)

When an agent connects to the server, it has to authenticate itself using the username and password provided in advance by the contest organizers. This way we prevent the unauthorized use of connections belonging to a contest participant. `AUTH-REQUEST` is the very first message an agent sends to the contest server.

The message envelope contains one element `<authentication>` without subelements. It has two attributes `username` and `password`.

Example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<message type="auth-request">
  <authentication password="1" username="a1"/>
</message>
```

2.2.3 `AUTH-RESPONSE` (server-2-agent)

Upon receiving `AUTH-REQUEST` message, the server verifies the provided credentials and responds by a message `AUTH-RESPONSE` indicating success, or failure of authentication. It has one attribute `timestamp` that represents the time when the message was sent.

The envelope contains one `<authentication>` element without subelements. It has one attribute `result` of type string and its value can be either `"ok"`, or `"fail"`. Example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<message timestamp="1297263037617" type="auth-response">
  <authentication result="ok"/>
</message>
```

2.2.4 `SIM-START` (server-2-agent)

The simulation starts by notifying the corresponding agents about the details of the starting simulation. This notification is done by sending the `SIM-START` message.

The data about the starting simulation are contained in one `<simulation>` element with the following attributes:

- the number of edges,
- the number of vertices,
- the respective agent's role,
- the id of the simulation, and
- the number of steps the simulation will last.

One step involves all agents acting at once. Therefore if a simulation has n steps, it means that each agent will receive n `REQUEST-ACTION` messages during the simulation (assuming a stable connection to the server).

Example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<message timestamp="1297263004607" type="sim-start">
  <simulation edges="47" id="0" steps="500" vertices="20"
              role="Explorer"/>
</message>
```

2.2.5 SIM-END (server-2-agent)

Each simulation lasts a certain number of steps. At the end of each simulation the server notifies agents about its end and its result.

The `<sim-result>`-tag has two attributes. `ranking` is the ranking of the team and `score` is the final score.

Example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<message timestamp="1297269179279" type="sim-end">
  <sim-result ranking="2" score="9"/>
</message>
```

2.2.6 BYE (server-2-agent)

At the end of the tournament the server notifies each agent that the last simulation has finished and subsequently terminates the connections. There is no data within the message envelope of this message.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<message timestamp="1204978760555" type="bye"/>
```

2.2.7 REQUEST-ACTION (server-2-agent)

In each simulation step the server asks the agents to perform an action and sends them the corresponding perceptions.

This message, due to its complexity, is best explained using an example. Note, however, that the following message is an artificial one, which has never been sent by the server:

Now, it is not necessary to elaborate on the nesting of the tags, which is obvious from the example. We will only focus on the relevant tags.

- `<perception>` has two attributes
  - `deadline` denotes the latest moment in time when the server will accept an action, and
  - `id` represents the action-id, that is the id, that is supposed to be added to the action-message.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<message timestamp="1297263230578" type="request-action">
  <perception deadline="1297263232578" id="201">
    <simulation step="200"/>
    <self energy="19" health="9" lastAction="skip"
    lastActionParam=""
    lastActionResult="successful" maxEnergy="19"
    maxEnergyDisabled="9" maxHealth="9" position="vertex4"
    strength="5" visRange="5" zoneScore="27"/>
    <team lastStepScore="27" money="1" score="4270"
    zonesScore="26">
      <achievements>
        <achievement name="area20"/>
        ...
      </achievements>
    </team>
    <visibleVertices>
      <visibleVertex name="vertex19" team="none"/>
      ...
    </visibleVertices>
    <visibleEdges>
      <visibleEdge node1="vertex0" node2="vertex11"/>
      ...
    </visibleEdges>
    <visibleEntities>
      <visibleEntity name="b5" team="B" node="vertex0"
                     status="normal"/>
      ...
    </visibleEntities>
    <probedVertices>
      <probedVertex name="vertex18" value="4"/>
    </probedVertices>
    <surveyedEdges>
      <surveyedEdge node1="vertex3" node2="vertex7" weight="2"/>
      ...
    </surveyedEdges>
    <inspectedEntities>
      <inspectedEntity energy="8" health="9" maxEnergy="8"
      maxHealth="9" name="b5" node="vertex10" role="role2"
      strength="6" team="B" visRange="2"/>
      ...
    </inspectedEntities>
  </perception>
</message>
```

- `<simulation>` has a `step`-attribute, that denotes the current step of the simulation.
- `<self>` represents the state of the vehicle, with the attributes
  - `energy`, which is the current energy,
  - `health`, which is the current health,
  - `lastAction`, which is the last action that has been performed,
  - `lastActionParam`, which is the parameter of last action that has been performed,
  - `lastActionResult`, which is the outcome of the last action, with precise semantics,
  - `maxEnergy`, which is the maximum energy,
  - `maxEnergyDisabled`, which is the maximum energy, when the vehicle is disabled,
  - `maxHealth`, which is the maximum health,
  - `position`, which is the vehicle's current position,
  - `strength`, which is the strength,
  - `visRange`, which is the visibility range, and
  - `zoneScore`, which is the value of the zone that the vehicle is part of.
- `team` represents the state of the vehicles team, with the attributes
  - `lastStepScore`, which is the score of the team in the last step,
  - `money`, which is the current amount of money the team has,
  - `score`, which is the overall score of the team, and
  - `zonesScore`, which is the sum of the values of all zones occupied by the team.

  Note, at this point, that `lastStepScore` is the sum of `money` and `zonesScore` from the last step. Note also that `score` is the sum of all `lastStepScore`s
- `<achievement>` is an achievement, whose name is indicated by the `name`-attribute.
- `<visibleVertex>` represents a visible vertex, whose name is indicated by the `name`-attribute, which denotes its identifier, and by the `team`-attribute, representing the team occupying the vertex.
- `<visibleEdge>` denotes a visible edge, its vertices are represented by the attributes `node1` and `node2`.
- `<visibleEntity>` represents a visible entity, denoted by the `name`-attribute. The `status` of the agent can be either `normal` or `disabled`.

- `<probedVertex>` is a probed vertex, the `name`-attribute is the vertex's name and the `value` is the vertex's value.
- `<surveyedEdge>` is a surveyed edge, `node1` and `node2` denote the adjacent vertices, and `weight` represents the weight.
- `<inspectedEntity>` represents an inspected vehicle, the attributes are
  - `energy`, which is the current energy of the vehicle,
  - `health`, which is the current health of the vehicle,
  - `maxEnergy`, which is the maximum energy of the vehicle,
  - `maxHealth`, which is the maximum health,
  - `name`, which is the vehicle's name
  - `node`, which is the name of the vertex the vehicle is standing on,
  - `role`, which is the vehicles role,
  - `strength`, which is the vehicle's strength,
  - `team`, which is the vehicle's team, and
  - `visRange`, which is the vehicle's visibility range.

2.2.8 `ACTION` (agent-2-server)

The agent should respond to the `REQUEST-ACTION` message by an action it chooses to perform.

The envelope of the `ACTION` message contains one element `<action>` with the attributes `type` and `id`. The attribute `type` indicates an action the agent wants to perform. It contains a string value which can be one of the following strings:

- `"goto"` with an obligatory attribute `param`, moves the entity to another vertex, whereas the attribute denotes the vertex,
- `"attack"` with an obligatory attribute `param`, attacks another entity, whereas the attribute denotes the entity-to-be-attacked,
- `"parry"` parries any attack,
- `"probe"` with an optional attribute `param`, probes the a vertex, whereas the attribute denotes the vertex-to-be-probed. If no attribute is given, current vertex is probed.

New in 2013: Added parameter for remote probing.

- `"survey"` surveys some visible edges,
- `"inspect"` with an optional attribute `param`, probes the a vertex, whereas the attribute denotes the entity-to-be-inspected. If no attribute is given, entities in the same vertex are inspected.

New in 2013: Added parameter for remote inspecting.

- `"repair"` with an obligatory attribute `param`, repairs another entity, whereas the attribute denotes the entity-to-be-repaired,
- `"buy"` with an obligatory attribute `param`, buys an item, whereas the attribute denotes the item-to-be-bought,
- `"recharge"` recharges, and
- `"skip"` does nothing.

Note, however, that the scenario description contains the precise semantics of the actions.

Here is an example of a `goto`-action:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="action">
  <action type="goto" param="vertex1">
</message>
```

Here is an example of a `attack`-action:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="action">
  <action type="attack" param="a2"/>
</message>
```

New in 2013: Added optional parameter.

Here is an example of a `probe`-action:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="action">
  <action type="probe" param="vertex1"/>
</message>
```

Here is an example of a `survey`-action:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="action">
  <action type="survey"/>
</message>
```

New in 2013: Added optional parameter.

Here is an example of a `inspect`-action:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="action">
  <action type="inspect" param="a2"/>
</message>
```

Here is an example of a `parry`-action:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="action">
  <action type="parry"/>
</message>
```

Here is an example of a `recharge`-action:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="action">
  <action type="recharge"/>
</message>
```

Here is an example of a `repair`-action:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="action">
  <action type="repair" param="b2"/>
</message>
```

Here is an example of a `buy`-action:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="action">
  <action type="buy" param="battery"/>
</message>
```

The attribute `id` is a string which should contain the `REQUEST-ACTION` message identifier. The agents must plainly copy the value of `id` attribute in `REQUEST-ACTION` message to the `id` attribute of `ACTION` message, otherwise the action message will be discarded.

Note that the corresponding `ACTION` message has to be delivered to the time indicated by the value of attribute `deadline` of the `REQUEST-ACTION` message. Agents should therefore send the `ACTION` message in advance before the indicated deadline is reached so that the server will receive it in time.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="action">
  <action id="70" type="skip"/>
</message>
```

2.2.9 Action Results

New in 2013: This section was added.

A part of the `REQUEST-ACTION` message is the result of the previously performed action. Usually three attributes will be provided: `lastAction` is the action name sent by the agent, `lastActionParam` is the action parameter sent by the agent, and `lastActionResult` is the outcome of the action.

For the semantics of each possible value of `lastActionResult`, please refer to the scenario description. All the possible values are listed here:

- `successful`
- `failed_resources`
- `failed_attacked`
- `failed_parried`
- `failed_unreachable`
- `failed_out_of_range`
- `failed_in_range`
- `failed_wrong_param`
- `failed_role`
- `failed_status`
- `failed_limit`
- `failed_random`
- `failed`