Multi-Agent Programming Contest Protocol Description (2015 Edition)

http://www.multiagentcontest.org/

Tobias Ahlbrecht Jürgen Dix Federico Schlesinger

May 18, 2015

New in 2015: Since we implemented a brand new scenario for 2015, specific contents of all messages have changed. The general communication protocol, however, remains the same.

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.

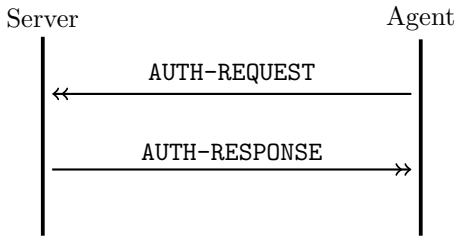

Figure 1: The initial phase.

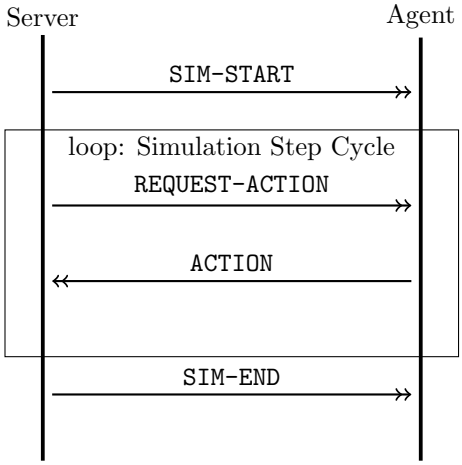

Figure 2: The simulation-phase.

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.

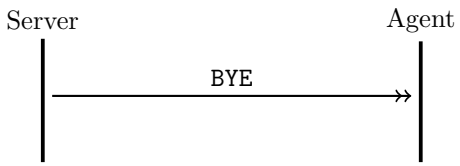

Figure 3: The final phase.

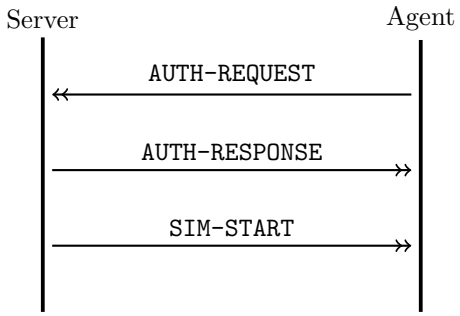

Figure 4: Reconnecting.

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

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 id="0" steps="500" role="Drone" team="A">
```

```
   <products>
    <product name="product1" volume="50" assembled="true">
     <consumed>
      <item name"it1" amount="5"/>
      ...
     </consumed>
     <tools>
      <item name="it2" amount="1"/>
      ...
     </tools>
    </product>
   </products>
  </simulation>
</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:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<message timestamp="1297263230578" type="request-action">
<perception deadline="1297263232578" id="201">
 <simulation step="200"/>
 <self
  charge="19"
  load="9"
  lastAction="skip"
  lastActionParam=""
  lastActionResult="successful"
  batteryCapacity="19"
  loadCapacity="9"
  lat="44"
  lon="44"
  inFacility="none"
  fPosition="-1"
  routeLength="2">
  <items>
   <item name="it1" amount="2"/>
   ...
  </items>
  <route>
   <n i="0" lat="10" lon="11"/>
   <n i="1" lat="10" lon="12"/>
   ...
  </route>
 </self>

<team money="1">
 <jobs-taken>
  <job id="job_id1"/>
  ...
 </jobs-taken>
 <jobs-posted>
  <job id="job_id2"/>
  ...
 </jobs-posted>
</team>

<entities>
 <entity name="b5" team="B" lat="44" lon="44" role="Truck"/>
 ...
</entities>

<facilities>
```

```
 <chargingStation name="charge1" lat="22" lon="44"
  rate="20" price="10" slots="3">
  <info qSize="3"/>
 </chargingStation>

 <dumpLocation name="dump1" lat="33" lon="77"
  price="10"/>

 <shop name="shop1" lat="99" lon="44">
  <item name="it1">
   <info cost="10" amount="5" restock="35"/>
  </item>
  ...
 </shop>

 <storage name="stor1" lat="22" lon="55"
  price="4" totalCapacity="50" usedCapacity="20">
  <item name="it1" stored="3" delivered="2"/>
  ...
 </storage>

 <workshop name="worksh1" lat="11" lon="33"
  price="9"/>

</facilities>

<jobs>
 <auctionJob id="job_id1" storage="stor1" begin="35" end="85"
  fine="100" maxBid="200">
  <items>
   <item name="it1" amount="10">
   ...
  </items>
 </auctionJob>
 <pricedJob id="job_id2" storage="stor2" begin="50" end="120"
  reward="150">
  <items>
   <item name="it2" amount="35" delivered="10">
   ...
  </items>
 </pricedJob>
</jobs>

</perception>
</message>
```

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.
- `<simulation>` has a `step`-attribute, that denotes the current step of the simulation.
- `<self>` represents the state of the vehicle, with the attributes
  - `charge`, which is the current charge,
  - `load`, which is the current load,
  - `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,
  - `batteryCapacity`, which is the maximum charge,
  - `loadCapacity`, which is the maximum load,
  - `lat`, which is the agent's latitude,
  - `lon`, which is the agent's longitude,
  - `inFacility`, which is the facility where the agent currently is located (or "none"),
  - `fPosition`, which is the position in the above facility (or -1),
  - `routeLength`, which is the length of the route, if the agent is following one,

  also the `<items>`-tag, which describes the items an agent carries by `<item>`-tags with the attributes
  - `name`, which is the item's name, and
  - `amount`, which is the amount carried,

  and the `route`-tag which describes the route the agent follows by a number of node-tags `<n>` with the attributes:
  - `i`, which is the index in the ordererd list of waypoints,
  - `lat`, which is the latitude of the waypoint, and
  - `lon`, which is the waypoint's longitude,

- `<team>` represents the state of the vehicles's team, with the attribute
  - `money`, which is the current amount of money the team has,

  and the subtags
  - `<jobs-taken>`, and
  - `<jobs-posted>`,

  which can include multiple `<job>`-tags with the attribute `id`,
- `<entities>` includes a tag for each agent in the simulation with self-explanatory attributes,
- `<facilities>` represents all facilities in the simulation (for readability, we will omit trivial attributes in the following):
  - `chargingStation`, representing a charging station, where
    - `rate` means the charging rate,
    - `slots` denotes the number of charging outlets, and
    - the `<info>`-tag contains information only visible if the agent is nearby, like the number of vehicles waiting to charge,
  - `dumpLocation` represents a dump facility,
  - `shop` represents a shop, listing all available items and showing the prices and quantities if the agent is nearby,
  - `storage` representing a storage facility with all stored and delivered products,
  - `workshop` representing a workshop facility.
- `<jobs>` containing all jobs currently active or to be auctioned, i.e.
  - `<auctionJob>`, and
  - `<pricedJob>`,

  again with all self-explanatory attributes.

Finally, to get a better understanding of what all this information means, you can read the scenario description.

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 optional attribute `param`, moves the entity to another location or facility.
- `"buy"` with an obligatory attribute `param`, buys a number of instances of an item from the shop at the current location.
- `"give"` with an obligatory attribute `param`, gives a number of instances of an item to a teammate.
- `"receive"` receives items from teammates,
- `"store"` with an obligatory attribute `param`, stores a number of instances of an item from the storage facility at the current location.
- `"retrieve"` with an obligatory attribute `param`, retrieves a number of instances of an item (previously stored) from the storage facility at the current location.
- `"retrieve_delivered"` with an obligatory attribute `param`, retrieves a number of instances of an item (previously delivered as part of a job's completion) from the storage facility at the current location.
- `"dump"` with an obligatory attribute `param`, dumps a number of instances of an item from the dump facility at the current location.
- `"assemble"` with an obligatory attribute `param`, creates a single instance of an item.
- `"assist_assemble"` with an obligatory attribute `param`, creates a single instance of an item.
- `"deliver_job"` with an obligatory attribute `param`, delivers at the storage facility at the current location, all relevant items that the agent is carrying, towards the completion of the job.
- `"charge"` increases the charge of the agent according to the charging station at the current location.
- `"bid_for_job"` with an obligatory attribute `param`, places a bid for an action job currently in the auction period.
- `"post_job"` with an obligatory attribute `param`, creates a new job posting.
- `"continue"` if there is an ongoing action (`goto` or `charge`), it is continued, otherwise does nothing.
- `"skip"` same as continue.
- `"abort"` does nothing, stops ongoing actions.

Note, however, that the scenario description contains the precise semantics of the actions, as well as the parameters expected for each action.

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

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="action">
  <action type="goto" param="at=51.805 lon=10.3355">
</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

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_location`
- `failed_unknown_item`
- `failed_unknown_agent`
- `failed_unknown_job`
- `failed_unknown_facility`
- `failed_item_amount`
- `failed_capacity`

- failed_wrong_facility
- failed_tools
- failed_item_type
- failed_job_status
- failed_job_type
- failed_counterpat
- failed_wrong_param
- failed_unknown_error
- failed
- successful_partial
- useless