$\mathcal{A}$-globe Manual

v1.1

Contents

1 Introduction

$\mathcal{A}$-**globe** is an agent platform designed for testing experimental scenarios featuring agents' position and communication inaccessibility, but it can be also used without these extended functions. The platform provides functions for residing agents, such as communication infrastructure, store, directory services, migration function, deploy service, etc. Communication in $\mathcal{A}$-**globe** is very fast and the platform is relatively lightweight.

See the web page of the $\mathcal{A}$-**globe** project:

`http://agents.felk.cvut.cz/projects/#aglobe`

2 Licence

$\mathcal{A}$-**globe** is licensed under Common Public Licence.

`http://www.opensource.org/licenses/cpl1.0.php`

3 Terminology

- **Agents** (classes derived from `aglobe.container.agent.Agent` or `aglobe.container.agent.CMAgent`) are autonomous entities.
  In $\mathcal{A}$-**globe**, each agent runs in its own thread (class `aglobe.platform.thread.AglobeThread`). Agents communicate with other agents using messages (class `aglobe.ontology.Message`). It is expected that a programmer of a multi-agent system will base their agents on these abstract classes (i.e. `Agent` or `CMAgent`).

- **Container** (class `aglobe.container.AgentContainer`) is a place where agents "live". When the platform is running, there must exist one server (master) container and some (or none) client containers. The *server container* is a place where "environmental" agents run that typically take care of the simulation infrastructure (environment simulator, entity manager, visibility manager, configuration manager etc.). The *client container* is a place where "scenario" agents run that are responsible for the execution of the simulation scenario itself. In $\mathcal{A}$-**globe**, it is possible to distribute the system over multiple computers, running each of the containers on a different PC. When more agent containers run on a single computer, only one Java Virtual Machine (JVM) is executed which results in efficient performance.

Note: When the platform is started without any parameters determining container type (neither "-master" nor "-client"), the container is started in a special "stand alone" mode. Such container is executed with no GIS service present (see section 5.1). This unusual container type can be used for simple simulations with a single container.

The level of decentralization depends on programmer's will. With **$\mathcal{A}$-globe** , it is possible to create multi-agent systems that can range from fully centralized to fully decentralized ones.

- **Accessibility** – agents on the same agent container are always mutually accessible (they can send messages (class `aglobe.ontology.Message`) to one another). As a result, messages sent between agent living on the same container are always successfully delivered. On the other hand, accessibility (the ability to transmit messages) between different client containers is controlled by a dedicated agent (e.g. class `aglobex.simulation.visibilitycrash.VisibilityCrashAgent`) located on the server container. This feature can be used for simulation of fully or partially inaccessible environments.

  Accessibility between the server container and any client container is not restricted, the server container is always accessible to any client container, and vice versa.

- **Services** (classes derived from `aglobe.container.service.Service` or `aglobe.container.service.CMService`) – each container can provide system services to be used by the agents living on the particular container. By using these services, agents are able to locate other agents (class `DirectoryService`), send and receive system messages, so called *topics* (class `GISClientService`), migrate (`AgentMoverService` and `DeployService`) etc. Usually, programmers don't need to extend these classes as they typically don't need to modify their default behavior.

- **Topic** is a special kind of communication. In contrary to messages whose delivery can be controlled by setting accessibility between containers, topics are system messages that will *always* be delivered, regardless of the container accessibility. Topics are meant to be used for distribution of system information (time stamps, data from sensors, entity position updates etc.)

- **Advertising** – an agent can offer some of its skills to other agents. It can advertise its abilities by using the directory service (class `aglobe.container.sysservice.directory.DirectoryService`) that basically serve as the ”yellow pages”, providing names and addresses of the publishers.

- **Agent migration** – any agent is able to migrate from one container to another. When an agent starts migrating, its execution is interrupted, its code and state is serialized and it is sent to the remote container. On the remote container, the agent is deserialized, its state is restored and its execution is resumed. If the migration fails for any reason, the agent stays on the original container. The migration process is transparent and is handled automatically by container services.

- **Container stores** – each container maintains XML files where startup configuration and parameters of each agent are stored. These files can either be generated automatically during execution or the system or can be created manually by the user.

  A service or an agent can also use the store as a place where it can save data for future usage. These data persist in the store even after the platform is shut down and they are loaded when the platform is started next time. When an agent migrates, its store migrates with it as well.

  To specify the directory in which the store of the platform is located (or will be created from scratch), the platform parameter `"-store` *`directory`*`"` can be used.

  Repetitive experiments usually require to be started with the same initial configuration of the system. In order to ensure that the stores will not be modified during the simulation, the platform startup parameter `"-noStoreChange"` can be used.

4 The Basic Concepts

4.1 Agent Initialization

The following example demonstrates how to create a simple agent in $\mathcal{A}$-**globe**.

The agent prints a text string "Hello World!" to the output console.

```
import aglobe.container.agent.*;
import aglobe.ontology.*;

public class HelloWorldAgent extends Agent {

   public void init(AgentInfo ai, int initState) {
      System.out.println("Hello World!");
   }

   protected void handleIncomingMessage(Message m) {
   }

}
```

As we can see, the `HelloWorldAgent` is derived from the abstract class `Agent`.

An agent typically provides an implementation of the method

```
public void init(AgentInfo ai, int initState)
```

This is the place where the initialization of the agent should take place (the use of constructors for this purpose is not recommended).

Any agent derived from the abstract class `Agent` must provide implementation of the method

```
protected void handleIncomingMessage(Message m)
```

This method is called when the agent receives an incoming message. In case of our example, the method is empty, because we don't expect to receive any messages.

To ensure proper functioning, the file `HelloWorldAgent.class` must be located in the `CLASSPATH` of the JMV where the agent will be loaded.

4.2 Scheduling an Event

To schedule an execution of an event in agent's body, use the agent's method `scheduleEvent()`.

```
scheduleEvent(new Runnable() {
   public void run() {
      System.out.println("Hello world");
   }
}, 10000);
```

The method `scheduleEvent()` creates an instance of the class `TimerTask`. This instance is inserted to the agent's task queue (`plan`) by the thread that runs separately from the container thread. In this example, the event is scheduled to be inserted to the queue after 10000 milliseconds. When the event is picked out of the queue, it is executed in the agent's thread. This approach ensures that the event will be executed synchronously and therefore there's no need for the programmer to take care of the synchronization issues at all.

For periodical scheduling of a repetitive event, use the alternate version of the same method: `public void scheduleEvent(Runnable event, long delay, long period)`.

4.3 Agent Termination

There are several different ways how an agent can terminate itself:

- The agent calls its own method `kill()`. After that, the agent is removed from the system *and* from the container store (configuration files). As a result, the agent *will not* be automatically loaded on the container during the next startup.

- The agent calls its own method `stop()`. After that, the agent is removed from the system *but not* from the container store (configuration files). As a result, the agent *will be* automatically loaded on the container during the next startup.
- The agent can send the quit command to the command service on the local container and this will cause the container (including all the agents located there) to terminate. These agents *will be* automatically loaded on the container during the next startup.
- The agent is terminated if it throws an catch exception.

4.3.1 Finish method

Before the agent is actually disposed of, its user-defined method `finish()` is called. This method can be regarded as the agent's destructor.

4.4 Agent's GUI

Showing and hiding of the graphic user interface (GUI) of an agent can be controlled in the GUI of the container the agent in located on. To enable this functionality, the agent must contain a public variable `gui` that is derived from the class `java.awt.Component`. The name of the variable is mandatory because the method `setVisible(boolean b)` of the GUI is called via Java reflection technique.

5 Container Services

Several services run on each container. They can be used by the agents living on that particular container. However, agents don't use the services directly. Instead, agents use service "shells" (proxies). The main reason for existence of shells is that an agent can migrate between containers and therefore it should not be linked to the services of a particular container directly. Instead, it uses only a generic interface (the shell) that shields the agent from a specific instance of a service. During migration, the shells take care of switching between container services automatically.

The shell automatically deregisters the agent from all records that agent already registered on the leaving container and registers the agent for this records on the destination container.

Also, shells simplify the access to the services by hiding the underlying messaging layer: the agent doesn't have to send messages in order to communicate with the services – instead, it only calls methods of the corresponding shells.

5.1 GIS Service

The GIS (Geographical Information System) service is used for distribution of topics. A topic is a system message. Delivery of the topic is always reliable, it doesn't depend on container accessibility (in contrast to the message delivery).

The GIS service has two variants – one for the server container (class `aglobe.service.gis.server.GISServerService`) and one for client container (class `aglobe.service.gis.client.GISClientService`).

When an agent located on a server container wants to receive some topics, it must implement the `aglobe.service.gis.server.GISTopicServerListener` interface. Similarly, an agent located on a client container must implement the `aglobe.service.gis.client.GISTopicListener` interface.

The system allows to send topics between agents located on the same container and between agents located on a server and a client container. Communication through topics is not possible between agents located on different client containers.

A topic consists of:

- String topic – the name of the topic (e.g. "TOPIC_TIMESTAMP_UPDATE"). An agent may subscribe for receiving a particular type of topics identified by this string.
- Object content – it is an arbitrary serializable object. This object contains the actual data being sent by the topic. Each topic of the topic receives its own unique copy (instance) of this object. Object content can be null.
- String reason – this is an optional string field of the topic that provides additional information about the topic. String reason can be null.

The following example demonstrates the subscription of an agent for topic "SOME_TOPIC" on a client container:

```
// get the service shell first
GISClientService.Shell gisShell =
   (GISClientService.Shell) getContainer().getServiceManager().
      getService(this, GISClientService.SERVICENAME);
// now, the subscription
gisShell.subscribeTopic("SOME_TOPIC", this);
```

After successful subscription, the agent will start receiving the topics "SOME_TOPIC". Whenever a new topic is received, the agent's method handleTopic() will be executed. This method is part of the GISTopicListener interface that the agent has to implement.

In case the agent runs on a server container, it must implement the GISTopicServerListener interface in order to subscribe for and receive topics.

If an agent subscribes for more topics (e.g. "SOME_TOPIC1" and "SOME_TOPIC2"), it must determine the type of the received topic as shown in the following code:

```
public void handleTopic(String topic, Object content, String reason)
{
   if (topic.equals("SOME_TOPIC1")) {
      // ...
```

```
    } else
    if (topic.equals("SOME_TOPIC2")) {
        // ...
    }
}
```

There is yet another technique of filtering topics: it is also possible to register a separate topic listener for each topic.

```
GISTopicListener gisTopicListener1 = new GISTopicListener() {
    public void handleTopic(String topic, Object content, String reason) {
        // handling topics SOME_TOPIC1
    }
    public void addEvent(Runnable r) {
        SomeAgent.this.addEvent(r);
    }
};

GISTopicListener gisTopicListener2 = new GISTopicListener() {
    public void handleTopic(String topic, Object content, String reason) {
        // handling topics SOME_TOPIC2
    }
    public void addEvent(Runnable r) {
        SomeAgent.this.addEvent(r);
    }
};

gisShell.subscribeTopic("SOME_TOPIC1", gisTopicListener1);
gisShell.subscribeTopic("SOME_TOPIC2", gisTopicListener2);
```

Code `SomeAgent.this.addEvent(r);` ensures that `handleTopic` will be executed in the agent's thread.

An agent submits a topic using variants of the method submitTopic() of the GIS shell.

Routing of topics depends on the type of the container (server or client) where the topic was submitted:

The submitting agent is located on a *client container*.

- `gisShell.submitTopicToLocal("SOME_TOPIC", content, reason);` the topic will be sent to all agents subscribed for the topic `"SOME_TOPIC"` that are located on the *same container* as the submitting agent
- `gisShell.submitTopicToServer("SOME_TOPIC", content, reason);` the topic will be sent to all agents subscribed for the topic `"SOME_TOPIC"` that are located on the *server container*
- `gisShell.submitTopic("SOME_TOPIC", content, reason);` the topic will be sent to all agents subscribed for the topic `"SOME_TOPIC"` that are located on the *same container* as the submitting agent or on the *server container*

The submitting agent is located on a *server container.*

- `gisServerShell.sendTopicToLocal("SOME_TOPIC", content, reason);` the topic will be sent to all agents subscribed for the topic `"SOME_TOPIC"` that are located on the *same container* as the submitting agent (i.e. the *server container*)
- `gisServerShell.sendTopic(destContainerName, "SOME_TOPIC", content, reason);` the topic will be sent to all agents subscribed for the topic `"SOME_TOPIC"` that are located on the *container of the given name* (e.i. `destContainerName`)
- `gisServerShell.broadcastTopic("SOME_TOPIC", content, reason);` the topic will be broadcast to all agents subscribed for the topic `"SOME_TOPIC"` that are located on *any container* (i.e. *either server or client)*

Note that *topics* are transmitted either between a server and a client container or locally within a single container. On the contrary, communication between different client containers is only possible via sending *messages* (`class aglobe.ontology.Message`).

5.1.1 Unsubscribing from a topic

An agent can unsubscribe from receiving a topic it subscribed for earlier:

```
gisShell.unsubscribe("SOME_TOPIC");
```

If the agent uses `getService(this, DirectoryService.SERVICENAME);` it means that unsubscription from the service will be proceeded automatically and it is not needed to do the unsubscription manually in the `finish` method. Instead of `this` is possible to use ”null” and manual unsubscription should be done in the finish method.

5.2 Directory Service

Each agent can advertise its abilities or services (generally records) in the Directory Service (class `aglobe.container.sysservice.directory.DirectoryService`) that serves as the ”yellow pages” for agents.

Directory Service is a *client container service* responsible for distributing information about agents advertisements over all client containers.

An agent can use the Directory Service in order to obtain information about other agents providing particular services (skills or abilities), as advertised in the Directory Service (the ”yellow pages”).

The following code demonstrates how an agent can advertise its record(s) in the Directory Service.

```
DirectoryService.Shell directoryShell =
   (DirectoryService.Shell) getContainer().getServiceManager().
      getService(this,DirectoryService.SERVICENAME);
String recordName = "SOME_RECORD";
Collection<String> list = new ArrayList<String>();
list.add(recordName);
try {
   directoryShell.register(this, list);
}
catch (DirectoryException ex) {
   ex.printStackTrace();
}
```

An agent can advertise several records at once and for this purpose, the method `directoryShell.register()` accepts a *list* of records. In case of our example, this list contains a single record only (`"SOME_RECORD"`).

Agents can subscribe for receiving a notification whenever an agent (or agents) start or stop advertising a record of a particular name.

```
directoryShell.subscribe(this, matchingFilter);
```

The subscription involves a string (`matchingFilter`) that is interpreted as a regular expression. Therefore, one subscription may cause an agent to subscribe for several records simultaneously.

Handling the information received from the Directory Service is implemented by the `aglobe.container.sysservice.directory.DirectoryListener` interface which includes the following methods:

```
// called when the agent(s) registered (advertised) their record(s)
// in the Directory Service
public void handleNewRegister(
                final String containerName,
                final DirectoryRecord[] records,
                String matchingFilter);

// called when the agent(s) deregistered their record(s)
// in the Directory Service
public void handleDeregister(
                final String containerName,
                final DirectoryRecord[] records,
                String matchingFilter);

// called when the registered agent(s) became visible (accessible)
public void handleVisible(
                final String containerName,
                final DirectoryRecord[] records,
                String matchingFilter);

// called when the registered agent(s) became invisible (inaccessible)
public void handleInvisible(
                final String containerName,
                final DirectoryRecord[] records,
                String matchingFilter);
```

The `containerName` is the name of the container where the advertising agent is located. The `records` array contains names and addresses of all the advertisers whose record names match the `matchingFilter` (defined earlier when calling the method `directoryShell.subscribe()`).

5.2.1 Unsubscribing from an agent service

Deregistration from the Directory Service is done analogously:

```
DirectoryService.Shell directoryShell =
   (DirectoryService.Shell) getContainer().getServiceManager().
```

```
        getService(this, DirectoryService.SERVICENAME);

String recordName = "SOME_RECORD";
Collection<String> list = new ArrayList<String>();
list.add(recordName);

directoryShell.unsubscribe(list);
```

The GIS service shell can be retrieved in two ways:

1. `getService(this, GISClientService.SERVICENAME);`
2. `getService(null, GISClientService.SERVICENAME);`

In case (1), the unsubscription from the service will be done automatically upon the removal of the agent, while in case (2), the same unsubscription must be done manually by the programmer (typically in the method `finish()`).

5.3 Command Service

This container service (class `aglobe.container.AgentContainer.CommandService`) runs on both types of containers (server and client) and provides the following system services:

- show the container GUI
- hide the container GUI
- shutdown the container

To (locally) shutdown a container on which the agent is currently running (and thus also killing all agents running on this container), one can use the following code:

```
CommandService.Shell commandShell =
    (CommandService.Shell) getContainer().getServiceManager().
        getService(this, AgentContainer.COMMANDSERVICE);
```

```
Command quitCommand = new Command();
quitCommand.setName(CommandService.QUIT);
commandShell.execute(quitCommand);
```

An important feature of the Command Service is the ability to execute the commands *remotely*. Let us present two different ways to shutdown a remote container:

- using a topic:
  ```
  Command quitCommand = new Command();
  quitCommand.setName(AgentContainer.CommandService.QUIT);
  gisServerShell.sendTopic(remoteContainerName,
        AgentContainer.CommandService.TOPIC_COMMAND, quitCommand);
  ```

- using a message:
  ```
  Message m = new Message(
           MessageConstants.REQUEST,
           getAddress(), remoteAgentAddress.
           deriveServiceAddress(AgentContainer.COMMANDSERVICE));
  Command quitCommand = new Command();
  quitCommand.setName(AgentContainer.CommandService.QUIT);
  m.setContent(quitCommand);
  try {
     sendMessage(m);
  }
  catch (InvisibleContainerException ex) {
     ex.printStackTrace();
  }
  ```

6 Messages

A message is implemented in the class `aglobe.ontology.Message`.

6.1 Addressing

An address is implemented in the class `aglobe.container.transport.Address`.

To construct an instance of the class `Address` from a string, use the static method `Address.getAddress(String str)`.

The following formats are acceptable:

- address of an agent or a service:
  `aglobe://`*<ip>*`:`*<port>*`/`*<container_name>*`/agent/`*<agent_name>*
  `aglobe://`*<ip>*`:`*<port>*`/`*<container_name>*`/service/`*<service_name>*
- address of a container
  `aglobe://`*<ip>*`:`*<port>*`/`*<container_name>*`/`
- address of a platform (JVM)
  `aglobe://`*<ip>*`:`*<port>*`/`

The class `Address` also provides many other methods, e.g.:

- `String getHost()` – resolves the IP address
- `String getPort()` – resolves the port number
- `String getContainerName()` – resolves the name of the container
- `String getName()` – resolves the name of the agent or the service
- `boolean isAgent()` – resolves whether the address represents an agent
- `boolean isService()` – resolves whether the address represents a service

6.2 Sending Messages

If we know the address of another agent or service, we can try to send a message to it:

```
Message m = new Message(
   performative, // performative string
   this.getAddress(), // sender
   targetAddress); // receiver
m.setContent("Hi there!");
try {
   sendMessage(m);
}
catch (InvisibleContainerException ex) {
   ex.printStackTrace();
}
```

The most common performative strings (based on FIPA protocols) are defined in the class `aglobe.ontology.MessageConstants`.

In case the target container is not accessible, the `InvisibleContainerException` is thrown.

If the content of the message is a serializable type or it is `javax.xml.bind.MarshallableRootElement`, the receiver gets the clone of content instance. If not, the receiver gets a String instead of instance of the sent object. This string is created by method `toString()` of content object.

6.3 Receiving Messages

As we mentioned earlier, each agent (and each service) derived from the abstract class `Agent` (or `Service`) must implement the method

```
public void handleIncomingMessage(Message m);
```

that will handle all incoming messages.

Message handling in classes derived from `CMAgent` or `CMService` is done differently, see chapter 7.

7 Conversation Manager Agent (CMAgent)

Conversation Manager Agent (class `aglobe.container.agent.CMAgent`) extends the class `aglobe.container.agent.Agent`.

`CMAgent` is designed to take care of handling complex conversation protocols automatically.

Each communication session is assigned with its own `conversationID` and the `CMAgent` creates an instance of the class `aglobe.container.task.Task` that manages the conversation flow for this particular communication session.

The `conversationID` determines to which communication session each incoming message belongs and based on this ID, the `CMAgent` passes each incoming message to the corresponding conversation task that processes it.

In contrary to agents derived from the `Agent` class, agents derived from the class `CMAgent` *do not* implement the method `handleIncomingMessage(Message m)`. All messages that do not match any of the existing `conversationID`s are processed by a special `Task` called an "idle task" (the "default" task) that must be set after the agent is created by using the `CMAgent`'s method `setIdleTask(Task t)`.

```
public class MyCMAgent extends CMAgent {

   public void init(AgentInfo ai, int initState) {
      this.setIdleTask(new MyIdleTask(this));
   }

   private class MyIdleTask extends Task {

      public MyIdleTask(MyCMAgent owner) {
         super(owner);
      }

      protected void handleIncomingMessage(Message m) {
         // handle the incoming messages,
         // create a dedicated reply task
         //    for each conversation session
         // etc.
      }
```

```
    }
}
```

FIPA communication protocols are implemented by corresponding pairs of *initiator* and *participant* tasks (e.g. derived from `aglobex.protocol.SubscribeInitiatorTask` and `aglobex.protocol.SubscribeParticipantTask`) – for more details and protocols see `http://www.fipa.org`.

When a conversation starts, it is important to send the opening message from the initiator task (e.g. derived from `SubscribeInitiatorTask`) so that the `conversationID` of the message is generated and set correctly.

At this point, the newly generated `conversationID` is not known to the receiver and therefore this message is handled in the method `handleIncomingMessage()` of the receiver's idle task. In this method, a new participant task (e.g. derived from `SubscribeParticipantTask`) should be created that will handle all subsequent messages belonging to this conversation session (described by the `conversationID`).

Tasks create reply messages using the method `message.getReply()` that creates a new instance of the message keeping the same `conversationID` and swapping the sender's and the receiver's address.

To send the message, use methods provided by the participant task (e.g. the method `void informResult(Object result, String reason)` in case the of `SubscribeParticipantTask`).

For more example implementations of communication protocols see the package `aglobex.protocol` which provides classes for handling several basics protocols. In following subsections we will desrcibe how to use these protocols.

7.1 Query-ref protocol

Inititator asks other agent using `query-ref` can be implemented using `aglobex.protocol.queryref.QueryRefInitiatorTask`. Example of code inside the initiator, where the query itself is described by object `query`:

```
   Task task = new QueryRefInitiatorTask(this, participantAddress,
query, false) {
       protected void informResult(Object result) {
           ... process the answer ...;
       }
   };
   task.start();
```

Received answer is processed in the `informResult` method, that has to be defined because it is abstract in the `QueryRefInitiatorTask` class.

Participant that can reply for `query-ref` can be implemented using `aglobex.-protocol.queryref.QueryRefParticipantTask`. Example of code inside the participant reacting on incoming message `message`:

```
    Task task = new QueryRefParticipantTask(this, message, false)
{
            Object answer = ... create the answer ...;
            informResult(answer);      // send the answer
        }
    };
    task.start();
```

The query is processed in the `processQuery` method, that has to be defined because it is abstract in the `QueryRefParticipantTask` class. It should call `informResult(...)` method that sends prepared answer to the initiator (asking agent).

`Query-if` and `request` protocols works in a similar way.

7.2 Subscribe protocol

Inititator subscribe other agent to receive updates on all the updates of its knowledge. For the implementation we can use `aglobex.protocol.-subscribe.SubscribeInitiatorTask`. Example of code inside the initiator, where the subscription is described by object `content`:

```
   Task task = new SubscribeInitiatorTask(this, participantAddress,
subscription, false) {
      protected void subscribeInformResult(Object object) {
         ...  process the answer ...;
      }
   };
   task.start();
```

Received answer is processed in the `subscribeInformResult` method that can be called several times. It has to be defined because it is abstract in the `SubscribeInitiatorTask` class.

Participant that can handle `subscribe` request can be implemented using `aglobex.protocol.subscribe.SubscribeParticipantTask`. Example of code inside the participant reacting on incoming message `message`:

```
   List<SubscribeParticipantTask> subscribers
      = new ArrayList<SubscribeParticipantTask>();
   ...
   Task task = new SubscribeParticipantTask(this, message, false)
{
      protected void processSubscribe(Message subscribeMessage)
{
         subscribers.add(this);
         informResult(actualValue);
      }
   };
   task.start();
   ...
   // on knowledge change, inform all subscribers
   for (SubscribeParticipantTask task :  subscribers) {
      task.informResult(actualValue);
```

The subscribe request is processed in the `processSubscribe` method, typically it only add the subscriber to the list of subscribers and sends actual value of subscribed knwoledge. When knowledge chagnes `informResult(...)`

method should be called with actual value to inform initiator (subscribing agent).

7.3 Contract-Net Protocol

Inititator sends call for proposals (CFP) to other agents that send him their proposals in reply. Initiator then selects good proposals and sends accept or refuse to all the participants.For the implementation we can use `aglobex.protocol.cnp.CNPInitiatorTask`. Example of code inside the initiator, where the recieved proposals are stored in `LinkedHashMap<Address,Message> receivedOffers`:

```
   Task task = new CNPInitiatorTask(this, participants, cfpContent,
false) {
      protected List<Address> evaluateReplies() {
         ...  process the proposals ...;
         return listOfChosenProposals;
subset of receivedOffers.keySet();       }
   };
   task.start();
```

`Accept` is send to the participants whose addresses are returned by `evaluateReplies` method, `refuse` is send to the rest.

Participant that can handle `cfp` can be implemented using `aglobex.protocol.cnp.CNPParticipantTask`. Example of code inside the participant reacting on incoming message `message`:

```
   Task task = new CNPParticipantTask(this, message, false) {
      protected void prepareProposal() {
         if (goodCfp(message)) {
            Object proposal = createProposal(message); {
            sendProposal(proposal); {
         } else {
            sendRefuse(proposal); {
         }
      }
   ...
      protected void proposalAccepted(Message message) {
         ...  do what you promised ...        }
   ...
      protected void proposalRefused(Message message) {
         ...  free blocked resources if necessary ...        }
   };
   task.start();
```

The subscribe request is processed in the `processSubscribe` method, typically it only add the subscriber to the list of subscribers and sends actual value of subscribed knwoledge. When knowledge chagnes `informResult(...)` method should be called with actual value to inform initiator (subscribing agent).

8 Simulation (Simulation)

8.1 Example setup of a scenario using A-globe simulation framework

This is a minimal configuration for usage of A-globe simulation framework. There are deployed two agents, each on a different container C1 and C2. Two configuration file are needed (*main.xml* and *scenario.xml*). *scenario.xml* is referenced by *main.xml*. This example is motivated by a test of *AliveProtocol*.

Before start of scenario in Entity Manager, you must start FullVisibilityAgent, EntityManagerAgent, EntitySimulatorAgent and ConfiguratorAgent: `store/AliveProtocolTest/prefs/agents/autorun.xml`:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<AgentList>
    <AgentInfo>
        <Name>EntityManager</Name>
        <ReadableName>EntityManager</ReadableName>
        <Type></Type>
        <MainClass>aglobex.simulation.entitymanager.EntityManagerAgent</MainClas
        <Libraries/>
    </AgentInfo>
    <AgentInfo>
        <Name>VisibilityAgent</Name>
        <ReadableName>VisibilityAgent</ReadableName>
        <Type></Type>
        <MainClass>aglobe.agent.visibility.FullVisibilityAgent</MainClass>
        <Libraries/>
    </AgentInfo>
    <AgentInfo>
        <Name>EntitySimulator</Name>
        <ReadableName>EntitySimulator</ReadableName>
        <Type></Type>
        <MainClass>aglobex.simulation.entitysimulator.EntitySimulatorAgent</Main
        <Libraries/>
    </AgentInfo>
    <AgentInfo>
        <Name>Configurator</Name>
        <ReadableName>Configurator</ReadableName>
        <Type></Type>
        <MainClass>aglobex.simulation.configurator.ConfiguratorAgent</MainClass>
        <Libraries/>
    </AgentInfo>
</AgentList>
```

aglobex_simulation/xml/AliveProtocolTest/main.xml:

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

<Configuration
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation=
    "../../src/ontology_src/Configuration.xsd"
SimulationCyclePeriod="100" MaxEntities="10" AllowReplay="true"
AllowFastSimulation="true">
    <SimulationScenario MainScenario="scenario.xml">
    </SimulationScenario>
    <EntityType Name="H"
        SimulationClass=
            "aglobex.simulation.entitysimulator.VoidBehaviour"
        VisioTypeName="AliveTestAgent"
        CrashRange="1">
  </EntityType>
</Configuration>
```

`aglobex_simulation/xml/AliveProtocolTest/scenario.xml`:

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

<Scenario xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation=
    "../../src/ontology_src/Configuration.xsd">
    <EntityStartupTemplate Name="Plane">
        <Agent Name="%CONTAINER_NAME%A"
            MainClass="aglobex.simulation.test.
                        AliveProtocolTestAgent"/>
    </EntityStartupTemplate>
    <Entity ContainerName="C1"
            EntityTypeName="H"
            UseEntityStartupTemplateName="Plane">
    </Entity>
    <Entity ContainerName="C2"
            EntityTypeName="H"
            UseEntityStartupTemplateName="Plane">
    </Entity>
</Scenario>
```

Program arguments:

```
    -gui
    -name AliveProtocolTest
    -store ../store
    -master
    -p config=../xml/AliveProtocolTest/main.xml
```

Tip: After start of simulation, the entity manager agent waits until all agents are logged in. Topic $ClientServerTopicConstants.TOPIC_ENTITY_CONNECTION$ is used for logging entity agent to the entity manager agent:

```
Example:
```

```
Shell gisShell= (GISClientService.Shell)getContainer()
    .getServiceManager().getService(this,GISClientService.SERVICENAME);
gisShell.subscribeTopic(
    ClientServerTopicConstants.TOPIC_ENTITY_CONNECTION,this);
```