Universidade de São Paulo
Escola Politécnica
Laboratório de Técnicas Inteligentes

Saci Programming Guide

(version 0.9)

Jomi Fred Hübner
e-mail:jomi@pcs.usp.br

Jaime Simão Sichman
e-mail:jaime@pcs.usp.br

São Paulo, July 2003

Abstract

Saci is a tool that turns programming communication among distributed agents easier. Two kinds of features are provided by Saci: an API for composing, sending, and receiving KQML messages; and a collection of tools that simplify some inherent difficulties to distribution (agent name service, yellow pages service, remote launching, communication debug, etc.). This manual explains the Saci's fundamentals, describe how to program agents using Saci's API, and how to use Saci's tools.

Acknowledgments

We would like to thank Rafael Heitor Bordini for his motivation and his contributions in the development of Saci. We also would like to thank Irene Durval Fichman for her valuable revision of this manual text.

Contents

Chapter 1

Introduction

The implementation of a Multi-Agent System (MAS) usually requires communication between agents in a distributed environment [9]. However, sometimes this is not a simple task. Developers need some knowledge about network protocols, TCP/IP ports, distributed programming (RMI, CORBA, DCOM), and some other technologies. In order to help the MAS designer, there are Multi-Agent System Development Environment (MDE) that provide tool boxes for the development of software applications using a MAS approach. Four tool boxes are proposed by Demazeau [3] (Figure 1.1):

Agent software skeletons ranging from simple reactive to cognitive agents.

Environment software skeletons containing domain topological models.

Interaction software skeletons for language and protocols that agents can use.

Organisation software skeletons for optimising the functioning of all the agents as a whole, ranging from unstructured to more structured organisations.

Thus, to develop a MAS the designer must choose agents, environment, interaction, and organisation models suitable for his application and available in these tool boxes. Saci (Simple Agent Communication Infrastructure) is a tool that can be considered an interaction tool box.

Saci enables distributed agents to communicate in an easy way. It is a set of Java classes and facilities that can be used in order to help the development of distributed agent societies. It provides the following main features:

- Agents are grouped in societies.
- Agents communicate using Knowledge Query and Manipulation Language (KQML) messages [7]. Saci has procedures for composing, sending, and receiving KQML messages.
- Agents are identified by name. They can send messages to other agents just using the receiver's name. Its location in the network is made transparent to the user by a facilitator agent (this facility is also know as "white pages service" or "agent name service").

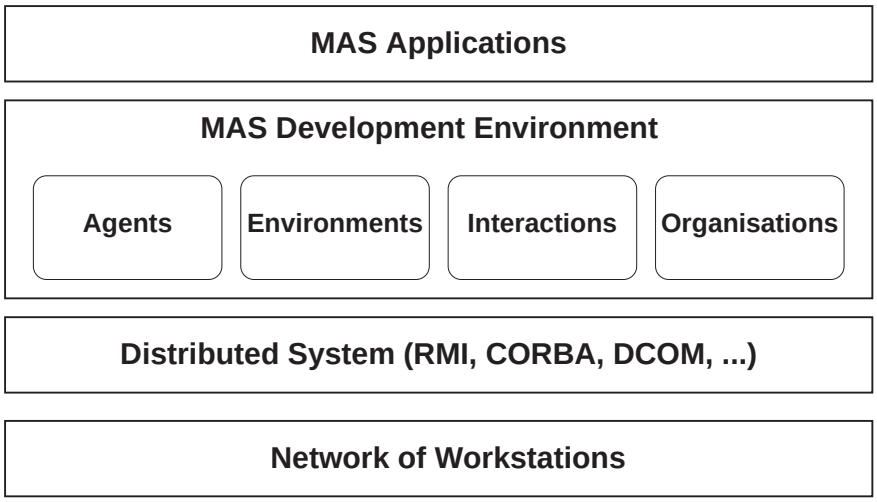

Figure 1.1: MAS Development tool boxes

- Agents may know others by a "yellow pages service". Agents can register their services with a facilitator and query this facilitator to find out what services are offered by which agents.
- Agents can be implemented as applets and run on web browsers.
- Agents can be launched remotely.
- Agents can be monitored. Social events (entrance, leaving, sending, or receiving messages) can be logged for further analysis.

This document sets out how to program agents using Saci's facilities. It does not aim to show how to install and configure Saci, this sort of information is available on Saci's home-page (`http://www.lti.pcs.usp.br/saci`).

Chapter 2 describes some MAS tools used for agent communication that have inspired Saci's development. In Chapter 3, Saci's infrastructure and functioning are specified. Chapter 4 describes how to program an agent using Saci's Application Program Interface (API), how to use Saci's facilitator services, and how to implement agents as applets. Finally, Chapter 5 shows how to use the facilities that come together with Saci.

Chapter 2

Agent Communication

2.1 KQML

Agents developed with Saci's infrastructure communicate among themselves using messages written in KQML. KQML is a language and a protocol specification that supports high level communication among agents [7, 8][1]. As an Agent Communication Language (ACL), KQML provides agents with the means of exchanging information and knowledge.

KQML is widely adopted because it has some effective features:

- Any language can be used for the content of the message (KIF, SQL, Prolog, etc.).
- The information used to understand the content of a message is included in the communication itself.
- When agents exchange messages at KQML level, they could ignore the transport mechanism (TCP/IP, RMI, IIOP, etc.), i.e., how the message will leave the sender agent and arrive at the receiver.
- KQML message format is simple, it is easy to parse and readable by humans.

The KQML message format is based on a LISP syntax, however, the arguments are identified by keywords preceded by a colon:

(*performative*			
	:language	*word*	} message layer
	:ontology	*word*	
	:sender	*word*	} communication layer
	:receiver	*word*	
	:reply-with	*word*	
	:content	*expression*	} content layer
	...)		

[1] A good introduction to KQML can be found on [2](in portuguese)

The message layer includes information that helps the receiver understand the content of the message. The *performative* field identifies the sender's intention with the message (inform, query, ask, ...), the value of the :language keyword is the language in which the message is expressed, and the value of :ontology is the vocabulary used for the "words" in the message. The communication layer describes the lower-level communication parameters, such as the identity of the sender and receiver, and a unique identifier for the message (:reply-with keyword). For example, with the message:

```
(ask-one
    :ontology     bovespa
    :language     SQL
    :receiver     stock-server
    :sender       ag1
    :reply-with   q1
    :content      "select price from stocktable where ent = Conectiva")
```

agent ag1 is asking agent stock-server something, the query is written using the SQL language under bovespa ontology. Since it is a performative query[2], the agent stock-server must answer it: [3]

```
(tell
    :language       prolog
    :ontology       bovespa
    :receiver       ag1
    :sender         stock-server
    :in-reply-to    q1
    :reply-with     a2
    :content        "[price(10.0)]")
```

This example shows just one possible use of the ask-one performative, but KQML specifies the use of many others. These performatives may be organised into seven basic categories:

- Query (evaluate, ask-one, ask-all, ...)
- Multiresponse query (stream-in, stream-all, ...)
- Response (reply, sorry, ...)
- Information (tell, achieve, cancel, untell, ...)
- Generator (standby, ready, next, ...)
- Capability (advertise, subscribe, ...)
- Networking (register, forward, broadcast, ...)

[2]This is a KQML protocol specification: every message with the ask performative must be answered. So, agents that implement the ask performative must follow this rule.

[3]The reply-with and in-reply-to fields may maintain, if the agent designer wishes, the *conversation*'s history among the agents.

Although KQML has a predefined set of reserved performatives and keywords, it is neither a minimal required set nor a closed one. However, agents that choose to implement one of the reserved performatives must implement it in the standard way.

Chapter 3

Saci's Specification

3.1 Saci society's model

Almost all agent's definitions agree that an agent exists within an environment and interact with it [4, 5, 1, 6, 10]. Moreover, we see a MAS as a system where agents are grouped in societies and can communicate with each other using agents' identification and a common language. The agent's identification is a name that uniquely identifies an agent as unique inside its society.

Saci provides a way for an agent to know others and communicate with them. In a certain way, we could say that some of the *social* features of a MAS is provided by the tool. In order to enable agents to know who are the others in their society, the agents' identifications are available to all agents. However, sometimes an agent needs to know more than just the other agents' identification to co-operate, it has to know their problem solving skills.

Formally, a society's structure state is defined as a tuple

$$Soc = \langle \mathcal{A}, \mathcal{S}, l, \delta \rangle$$

such that

$\mathcal{A} = \{\alpha \mid \alpha \text{ is an agent's identification within the society}\}$,
$\mathcal{S} = \{\sigma \mid \sigma \text{ is an available skill in the society }\}$,
l is a society's language, and
$\delta : \mathcal{A} \nrightarrow \mathbb{P}(\mathcal{S})$ is a partial function that maps agents to skills, such that
$\delta(\alpha) = \{\sigma \mid \sigma \text{ is an } \alpha\text{'s skill}\}$.

For instance:

lti = ⟨{Jomi, Jaime, Julio, Jose},
{Java, C, Prolog, Teach},
Portuguese,
{Jomi ↦ {Java}, Jaime ↦ {C,Teach}, Julio ↦ {Java}}⟩

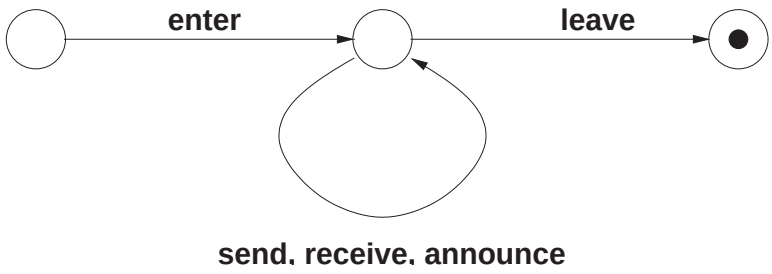

Figure 3.1: Agent Live Cycle

This structure state may change over time by some social event, like agent entrance, skill announcement, etc:

$Soc_i \Rightarrow Soc_{i+1}$ | some social event happened at moment i.

In Saci, the internal agent functioning does not matter: it could be reactive, cognitive, or whatever. There is only one restriction on the agent behaviour, they must have the following life cycle (cf. Figure 3.1):

Enter into a society: the agent gets a social identification. The entrance of agent α at moment i will change the society's structure as follows:

$$\langle \mathcal{A}, \mathcal{S}, l, \delta \rangle_i \Rightarrow \langle \mathcal{A}', \mathcal{S}, l, \delta \rangle_{i+1} \mid \mathcal{A}' = \mathcal{A} \cup \{\alpha\}$$

Announce skills: optionally, one agent may announce a skill to the society. If agent α announce a skill σ, the society's structure will change as follow:

$$\langle \mathcal{A}, \mathcal{S}, l, \delta \rangle_i \Rightarrow \langle \mathcal{A}, \mathcal{S}', l, \delta' \rangle_{i+1} \mid \mathcal{S}' = \mathcal{S} \cup \{\sigma\}$$
$$\delta'(x) = \begin{cases} \delta(x) & \text{if } x \neq \alpha \\ \delta(x) \cup \{\sigma\} & \text{otherwise} \end{cases}$$

Send/receive messages to/from agents in the same society; and

Leave the society: agent α looses its identity inside the society.

$$\langle \mathcal{A}, \mathcal{S}, l, \delta \rangle_i \Rightarrow \langle \mathcal{A}', \mathcal{S}', l, \delta' \rangle_{i+1} \mid \mathcal{A}' = \mathcal{A} - \{\alpha\}$$
$$\delta'(x) = \begin{cases} \delta(x) & \text{if } x \in \mathcal{A}' \\ \{\} & \text{otherwise} \end{cases}$$
$$\mathcal{S}' = \{\sigma \mid \sigma \in \delta'(x)\}$$

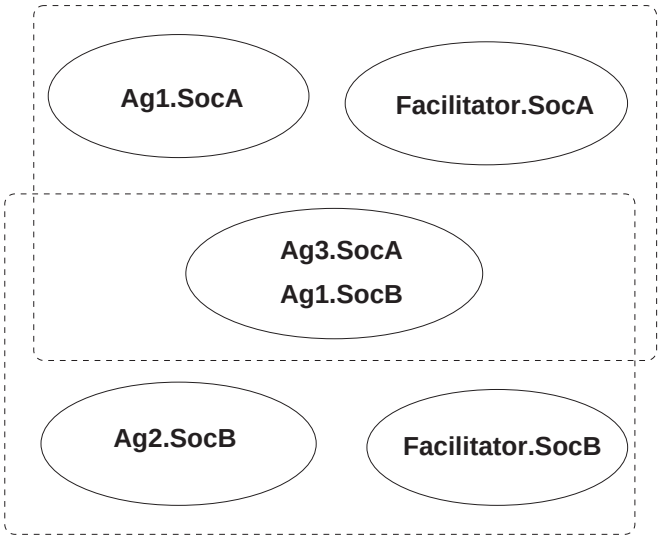

Figure 3.2: Saci environment sample

3.2 Saci's architecture

3.2.1 Entering and leaving societies

As suggested by the KQML architecture, every Saci society has a facilitator agent that maintains its structure: the identity, the location[1], and the services provided by the society's agents. Figure 3.2 shows an example of a Saci environment where there are two societies (*SocA* and *SocB*) with two agents and a facilitator in each one. When an agent wants to enter into a society it has to contact the society's facilitator and register its name. The facilitator will bind the agent's name with its location and give it a unique identification. Notice that when leaving the society, an agent has to notify the societies' facilitator to which it belongs.

3.2.2 Sending and receiving messages

A MBox component serves as an interface between the agent and the society. Its main aim is to deliver a message to the receiver and to turn transport network mechanisms and remote locations *transparent* to the agent developer. Thus, in order to communicate, Saci agents use MBox's methods that encapsulate interaction tasks (sending messages, receiving messages, advertising, etc.). Of course, an agent may have more than one or two MBox objects, each of them corresponding to a single society to which it belongs.

In the case of Figure 3.3, agent $Ag2$ of society *SocA* wants to communicate with agent $Ag1$ which belongs to the same society *SocA*, $Ag2$ also knows $Ag1$'s name (these are two

[1]Some architectural components were introduced in the society model presented at Section 3.1, like the agent's location in the network.

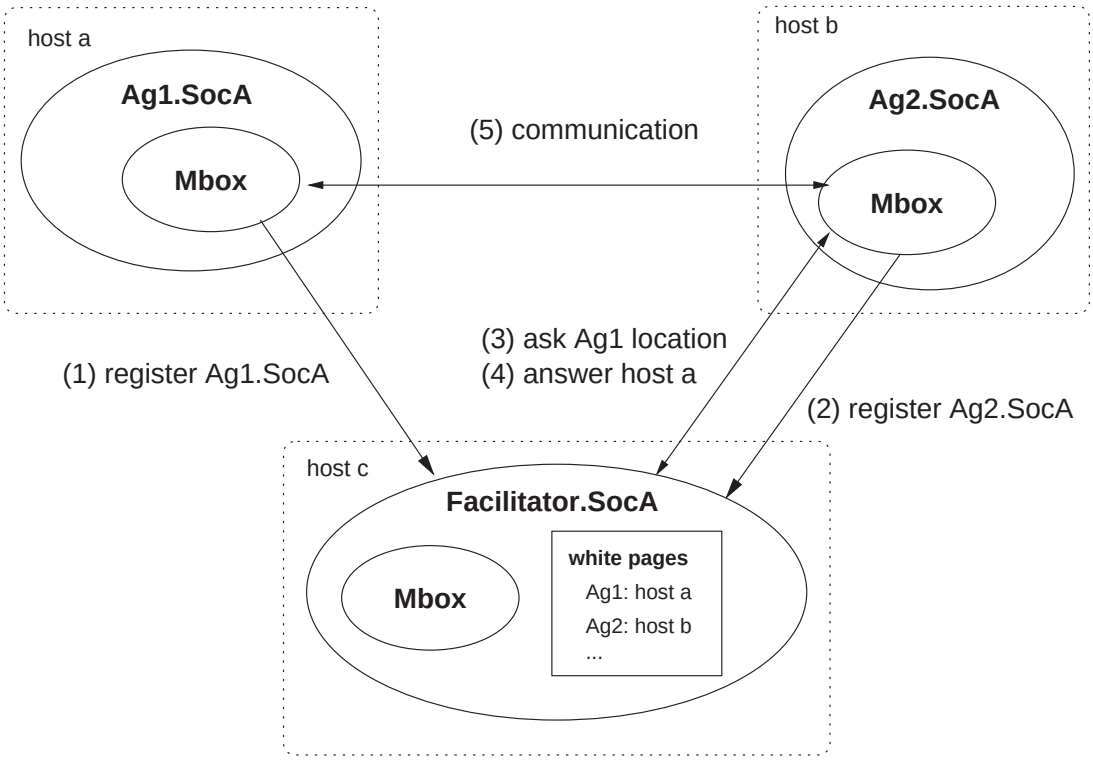

Figure 3.3: White pages service

preconditions for an agent to send a message to another). At first, $Ag2$'s MBox needs to know $Ag1$'s location, so it asks $SocA$'s facilitator for the location (arrow (3)). Having the location (arrow (4)), $Ag2$ starts to communicate with $Ag1$ through its MBox (arrow (5)).

3.2.3 Announcing skills

In Saci societies, the agents can advertise their skills with the society's facilitator (analogous to a "yellow page" list) in order to be known inside its society. When an agent decides to ask another for a task and do not know the other's name, it may query the facilitator for a set of agents' names that have the required skill. For instance (see Figure 3.4), considering agent $Ag2$ has advertised a skill that $Ag1$ needs (arrow (1)), $Ag1$ asks the facilitator for such a skill (arrow (2)), the facilitator gives $Ag2$ as the answer (arrow (3)), and then $Ag1$ can start to communicate with agent $Ag2$ (arrow (4)). Indeed, this is just one model available for presentation. Saci agents can use other ways to be known, for instance: broadcasting its skills to all agents.

In the sample of Figure 3.4, the KQML message $Ag2$ uses for advertising one skill to the facilitator (arrow (1)) is

```
(advertise
    :receiver       Facilitator
```

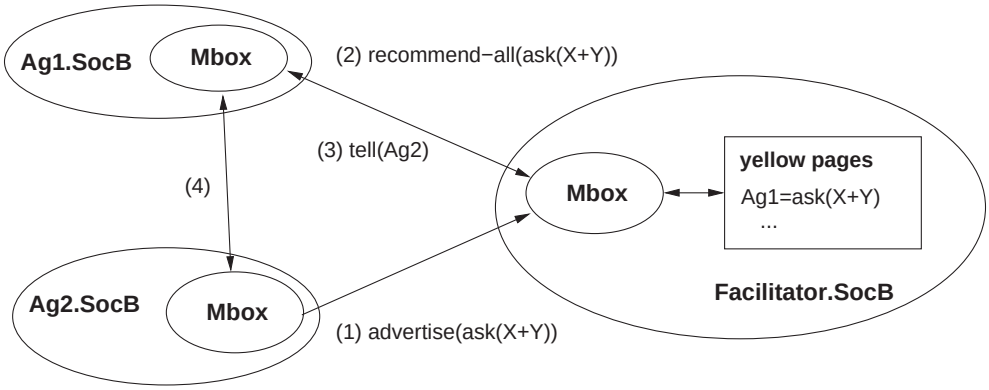

Figure 3.4: Yellow pages service

```
:sender      Ag2
:language    KQML
:ontology    yp
:content     (ask-one
                  :receiver    Ag2
                  :language    alg
                  :ontology    math
                  :content     "X + Y"))
```

that means agent $Ag2$ can answer messages with the performative ask-one, the language alg, the ontology math, and formulae with two operands. Then Agent $Ag1$ uses the following message to get a list of agents (arrow (2)):

```
(recommend-all
    :receiver      Facilitator
    :sender        Ag1
    :reply-with    id1
    :language      KQML
    :ontology      yp
    :content       (ask-one
                        :language    alg
                        :ontology    math
                        :content     "X + Y"))
```

and the answer from the facilitator (arrow (3)) is

```
(tell
    :receiver      Ag1
    :sender        Facilitator
    :in-reply-to   id1
```

:language	KQML
:ontology	yp
:content	(Ag2))

Chapter 4

Using Saci to Develop Agents

4.1 A simple sample

One of the easiest way to learn a new API is by an example. So let us consider an agent's source that sends and receives a message:

```
1  import saci.*;
2
3  public class SampleSaciAg extends Agent {
4
5  public void run() {
6    try {
7      Message m = new Message("(ask-one" +
8                  " :content \"2+4\"" +
9                  " :receiver APlusServer" +
10                 " :reply-with rAdd)");
11     mbox.sendMsg(m);
12     boolean ok = false;
13     while (! ok) {
14       Message r = mbox.polling();
15       String irt = (String)r.get("in-reply-to");
16       if (irt.equals("rAdd")) {
17         String ans = (String) r.get("content");
18         System.out.println ("Answer is " + ans);
19         ok = true;
20       }
21     }
22   } catch (Exception e) {
23     System.err.println ("Error " + e);
```

```
24      }
25  }
26
27  public static void main(String [] args) {
28      SampleSaciAg a = new SampleSaciAg();
29      if (a.enterSoc("SampleSaciAg")) {
30          a.run();
31          a.leaveSoc();
32          System.exit(0);
33      }
34  }}
```

The first thing to notice in the code is the main method (line 27). Agent's life cycle is coded there: the agent enters into a society (line 29), sends and receives messages (line 30, through run method), and leaves the society (line 31). In the run method, the agent creates a KQML message (line 7), sends it asynchronously using its inherited mail box interface (line 11), gets an incoming message (line 14), and, if the message is a reply to the previous sent message (line 16), shows the answer (line 18). The following run method is equivalent to the above code, but it uses the ask method instead of send and polling (the ask method sends the message and waits for an answer for it).

```
1   public void run() {
2     try {
3       Message m = new Message("(ask-one"+
4                   " :content \"2+4\""+
5                   " : receiver APlusServer"+
6                   " : reply-with rAdd)");
7       Message r = mbox.ask(m);
8       String ans = (String)r.get("content");
9       System.out.println ("Answer is " + ans);
10    } catch (Exception e) {
11      System.err.println ("Error " + e);
12    }
13  }
```

It is also possible to send messages by the KQML broker protocol. In this protocol, the agent do not need to know the name of the agent that will answer it. The following source samples how to use this protocol:

```
1   public void run() {
2     try {
3       Message m = new Message("(ask-one"+
4                   " :content \"2+4\""+
5                   " :language algebra)";
```

```
6         Message r = mbox.brokerOne(m);
7         String ans = (String)r.get("content");
8         System.out.println("Answer is " + ans);
9       } catch (Exception e) {
10        System.err.println("Error " + e);
11      }
12    }
```

4.1.1 Compiling an agent

To compile the program, make sure that Saci's saci.jar file is in the class path. Considering Saci was installed under /opt directory, the class path variable should be set and the program can be compiled with:

```
export CLASSPATH=/opt/saci/bin/saci.jar:$CLASSPATH [1]
javac SampleSaciAg.java
```

4.1.2 Running an agent

Saci agents can be started in many ways: by a Java Virtual Machine (JVM), by an agent launcher tool, or by a web browser. However, in order to start Saci agents in a host, a Saci launcher must be already running there. So, before starting the agent example compiled in the previous section, the launcher has to be started and the APlusServer agent, which will be asked by the SampleSaciAg agent to sum, must be running. Then, the SampleSaciAg agent can be started by a JVM with the command:[2]

```
java SampleSaciAg
```

Figure 4.1 illustrates the steps for starting this society:

1. Saci's Menu is started by running the saci or the saci.bat script that is in the Saci's bin directory (Figure 4.2 shows this menu);
2. Saci's Menu starts a launcher demon automatically;
3. a society launcher is started by Saci's Menu;
4. the society launcher asks the launcher demon to create a new society with no name ("" is the default society);
5. the launcher demon starts a new facilitator for the society "";
6. by Saci Menu, an agent launcher is started;

[1] In window systems where Saci was installed under "c:\" directory, this command should be: set CLASSPATH=c:\saci\bin\saci.jar;%CLASSPATH%

[2] See in Section 5.1.1 what to do to start this agent from Saci Agent Launcher tool.

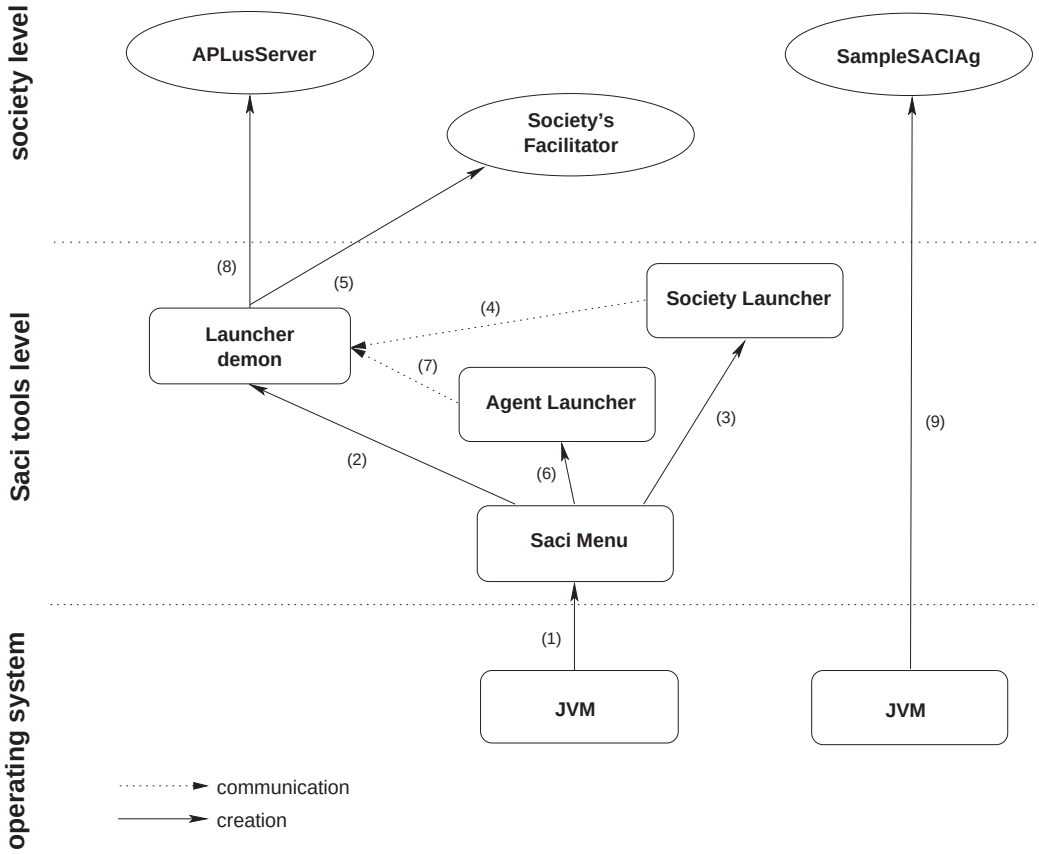

Figure 4.1: Starting a Saci Society

7. the agent launcher asks the launcher demon to create a new agent with the name "APlusServer" from PlusServer class;
8. the launcher demon starts such agent; and finally
9. the agent "SampleSaciAg" is stared by a JVM with the command java SampleSACIAg.

4.2 Entering and leaving societies

If one agent extends the Agent or AppletAgent class, the enterSoc and leaveSoc methods implement respectively the entry and exit in the society. The enterSoc method receives as a parameter the preferable name for the agent in the society. If the name already exists in the society, the facilitator will chose another one. The real agent's name in the society can be obtained by getMbox().getName() method. Since a Saci context can have many societies, sometimes it is necessary to give more parameters to the entry operation, for instance, the society's name that

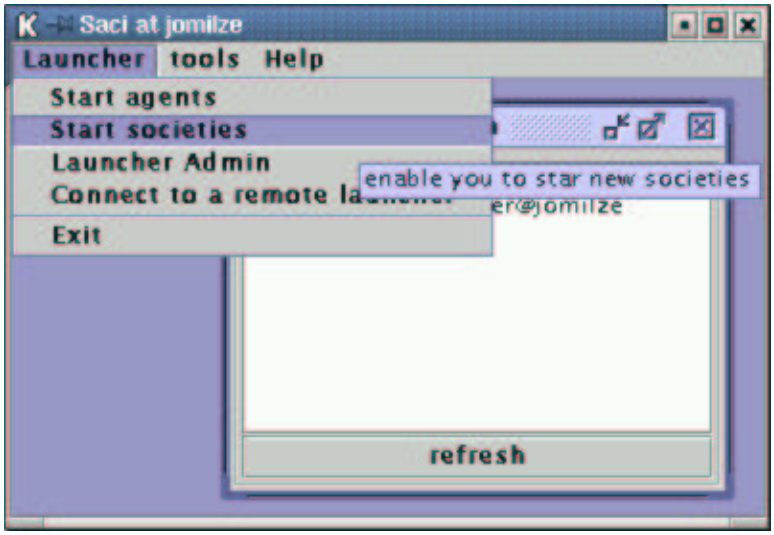

Figure 4.2: Saci Menu

the agent wants to enter. Thus, the enterSoc method also has an optional parameter: a Config object. Config could be set with two arguments (both are optional)[3]:

- society.name: society's name where the agent is entering, the default value is "". The society with name "" is also know as "<default society>". The default society is used when an agent does not specify one.
- facilitator.host: host's name where the society's facilitator is running, the default value is "localhost". If this parameter is not informed, the Saci will try to find out the host.

The following source samples the Config use for entering the society "simsoc" which facilitator is running at host "nantes.pcs.usp.br":

```
1    SampleSaciAg a = new SampleSaciAg();
2    Config c = new Config();
3    c.set("facilitator.host", "nantes.pcs.usp.br");
4    c.set("society.name", "simsoc");
5    if (a.enterSoc("SampleSaciAg", c)) {
6       System.out.println("My name is " + a.getMBox().agentName());
7        ...
```

If an agent's project does not allow extending the Saci's Agent class, it is possible to use an instance of the MBoxSAg class to enter and leave societies. An example may be:

```
1  import saci.*;
2  class Sample2 {
```

[3]If the agent launcher tool is used to start the agent, this tool will set Config's parameters automatically.

```
3     public static void main(String [] args) {
4        try {
5           MBoxSAg mbox = new MBoxSAg("AnAgName");
6           mbox.init();
7           mbox.sendMsg( new Message(
8              "( tell :content hello :receiver Rafa)" ));
9           mbox.disconnect();
10       } catch (Exception e) {
11          System.err.println ("Error " + e);
12       }
13       System.exit(0);
14    }
15 }
```

The MBoxSAg constructor receives the same parameters as the enterSoc method (an agent's name and a Config object) and its construction means that the agent will enter in a society (line 5). The MBoxSAg's init method starts the thread that delivers messages from an outgoing queue (see Figure 4.3). The agent leaves the society by using the disconnect method (line 9). Once the above agent does not extends the Agent class, this agent can only be started from the operating system, it can not be launched by the launcher tool (in order to make it launchable, see Section 5.1.1).

4.3 Sending and receiving messages

Before sending a message, an agent needs to create a Message object and set the corresponding KQML fields to it. Messages are simply mappings from field tags to values. In the Message construction, the string argument is parsed and the corresponded fields are put in the message's mapping. Because Message is a subclass of Hashtable (a Java mapping data structure), fields can also be set with the put(<field>, <value>) method. For instance:

```
1  Message r = new Message("(tell :content 45)");
2  r.put(" receiver ", "jomi");
3  r.put("in-reply-to", id3);
```

after line 3, the message r will have the following fields:

Field	Value
performative	tell
content	45
receiver	jomi
in-reply-to	id3

To send and receive messages, the following methods of the MBox interface may be used (see Figure 4.3):

- sendMsg(*Message*) sends a KQML message by adding it in the outcome queue. If there is neither "sender" nor "reply-with" fields in the message, these KQML fields will be automatically added.
- sendSyncMsg(*Message*) is the same as sendMsg, but it sends the message synchronously (the sender will wait until the receiver reads the message).
- receive() returns the first message in the agent's mail box.
- receive(*Pattern*) is the same as receive(), but select the first message that match *Pattern*. *Pattern* is a Message object with some fields used to filter the existing messages.
- polling([*timeout*]) is the same as receive(), but if there isn't a message, it will wait for one. If time-out is reached, it returns null. If time-out is not informed, this method will wait until one message comes (i.e., timeout is infinite).
- polling(*Pattern*, *timeout*) is the same as polling(), but waits for a message that matches *Pattern*.
- ask(*Message* [,*timeout*]) sends *Message* and waits until the agent receives a message with the appropriate "in-reply-with" for the *Message*. If time-out is reached, it returns null. If time-out is not informed, it will wait until one answer comes.
- getMessages(*Pattern* [, *quantity*, *timeout*, *remove*]) returns at least *quantity* messages from the MailBox that match *Pattern*. If there is not such number of messages, the method waits *timeout* milliseconds (at most) for them. If *Pattern* is null, waits at most *timeout* milliseconds for *quantity* messages. If *timeout* is -1, waits until *quantity* messages arrive (may be forever). If some error happens, returns null. If *remove* is true, the messages will be removed from the MailBox.

The following methods just encapsulate the use of facilitator's services:

- broadcast(*Message*) sends a message to the facilitator which broadcasts it to all the society's agents.
- brokerOne(*Message*[, *timeOut*) sends a message to the facilitator which will find one agent to answer it, say agent b. The method returns the answer b gives to the facilitator (see [7] for more information about the KQML broker protocol).
- forward(*Message*) sends a message through the facilitator. The facilitator will send it to the "receive" agent. It is useful if an agent can not contact an other agent with its network resources. In that case, the agent can ask the society's facilitator to send the message.

Instead of using these methods, one can send messages to facilitator to ask him services (see [7]). For example, instead of using the forward(new Message("(tell :receiver Ag3 :language alg :ontology math :content "1 + 3")")) method, one can send the following message:

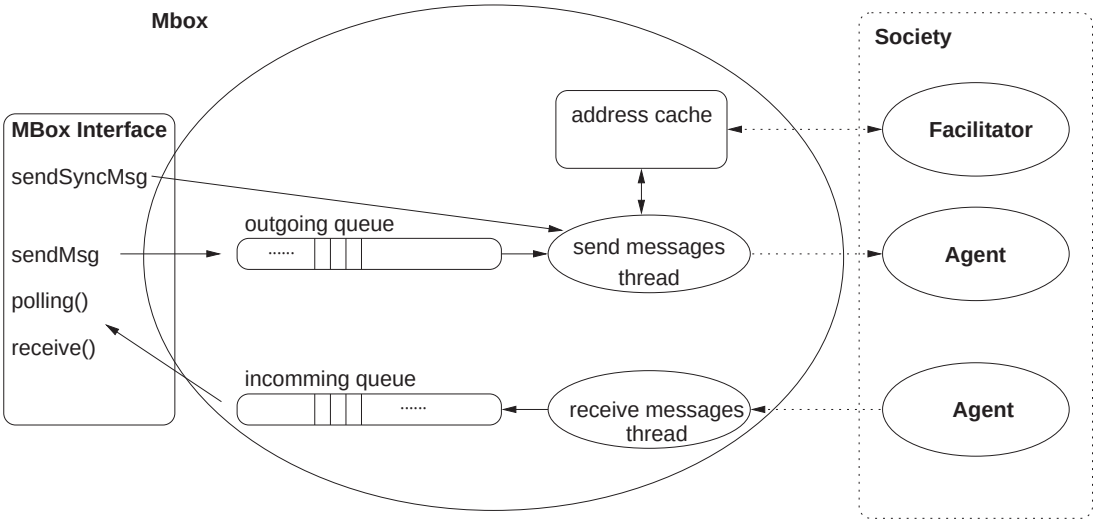

Figure 4.3: Agent Mail Box Functioning

```
(forward
    :from       Ag2
    :to         Ag3
    :receiver   Facilitator
    :sender     Ag2
    :language   kqml
    :ontology   kqml-ontology
    :content    (tell
                    :receiver   Ag3
                    :language   alg
                    :ontology   math
                    :content    "1 + 3"))
```

Instead of using the brokerOne(new Message("(ask-one :receiver AgS :language alg :ontology math :content "1 + 3")")) method, one can send the following message:

```
(broker-one
    :receiver   Facilitator
    :sender     Ag2
    :language   kqml
    :ontology   kqml-ontology
    :content    (ask-one
                    :language   alg
                    :ontology   math
                    :content    "1 + 3"))
```

4.4 Announcing skills

To announce skills, agents may send an advertise message to the society's facilitator (Cf. Section 3.2.3) or they should use MBox's method (that wraps the sending of the message):

- advertise(*performative*, *language*, *ontology*, *term*): advertises facilitator that the agent can receive messages of *performative*, *language*, *ontology*, and *term*. If some of the parameters are not relevant, they may have the null value.

In order to consult the facilitator's yellow pagea table, agents either send an *recommend-all* message or they should use the following *MBox* method that encapsulates the communication:

- recommdAll(*performative*, *language*, *ontology*, *term*): asks the facilitator to give a list of agents that advertise such skill, then it returns a Vector object with agent's names.

The source bellow shows the code for the $Ag2$ of Figure 3.4:

```
1  import saci.*;
2
3  public class PlusServer extends Agent {
4      public static void main(String[] args) {
5          Agent a = new PlusServer();
6          if (a.enterSoc("Ag2")) {
7              a.initAg(null);
8              a.run();
9          }
10     }
11
12     public void initAg(String[] args) {
13         try {
14             mbox.advertise("ask-one", null, "math", "X + Y");
15         } catch (Exception e) {
16             System.err.println("Error " + e);
17         }
18     }
19
20     public void run() {
21         while (true) {
22             Message m;
23             try {
24                 m = mbox.polling();
25             } catch (Exception e) {
26                 m = null;
27             }
28             if (m == null)
```

```
29                  continue;
30
31              if (m.get("performative").equals("ask-one")) {
32                  Message r = new Message("(tell)");
33                  r.put("receiver", m.get("sender"));
34                  r.put("in-reply-to", m.get("reply-with"));
35                  r.put("content", sum( (String)m.get("content") ));
36                  mbox.sendMsg(r);
37              }
38          }
39      }
40
41      String sum(String formula) {
42          try {
43              formula = formula.substring(1,formula.length()-1);
44              int posMais = formula.indexOf("+");
45              String opEsq = formula.substring(0,posMais);
46              String opDir = formula.substring(posMais+1);
47              float fEsq = (new Float(opEsq)).floatValue();
48              float fDir = (new Float(opDir)).floatValue();
49              return ""+(fEsq + fDir);
50          } catch (Exception e) {
51              return "Error in formula";
52          }
53      }
54  }
```

In the initAg() method (line 14) the agent advertises its skill to the facilitator.[4] Notice that the $Ag2$ ignores the language used to ask its services, since the language parameter is null.

The $Ag1$'s source should be:

```
1  import saci.*;
2
3  public class PlusClient extends Agent {
4      public static void main(String[] args) {
5          PlusClient a = new PlusClient();
6          if (a.enterSoc("Ag1")) {
7              a.run(args[0]);
8              a.leaveSoc();
9          }
10     }
```

[4]The agent must be initialized in the method initAg() because this method is called by the launcher before the agent starts running (see Section 5.1.1).

```
11
12      public void run(String exp) {
13          try {
14              Vector receptores =
15                  mbox.recommendAll("ask-one", null, "math", "X + Y");
16
17              if ( receptores.size() == 0) {
18                  System.out.println ("There is no agent to answer me!");
19              } else {
20                  for (int i=0; i < receptores.size (); i++) {
21                      String ag = (String) receptores.elementAt(i);
22                      Message m = new Message(
23                                  "(ask-one :content \"" + exp + "\")");
24                      m.put("receiver", ag);
25                      m.put("reply-with", "rSum");
26                      mbox.sendMsg(m);
27                  }
28                  Message pattern = new Message();
29                  pattern.put("in-reply-to", "rSum");
30                  Vector answers = mbox.getMessages( pattern,
31                                                     receptores.size (),
32                                                     4000, true);
33                  for (int i=0; i<answers.size (); i++) {
34                      Message answer = (Message)answers.elementAt(i);
35                      System.out.println ("Answer from" +
36                                  answer.get("sender") + " was " +
37                                  answer.get("content"));
38                  }
39              }
40          } catch (Exception e) {
41              System.err.println ("Error " + e);
42          }
43      }
44  }
```

In line 15 $Ag1$ gets as a result from method MBox.recommendAll() a Vector with all agents in its society that can add two number. If there is at least one, it sends a message to each one (in the loop of lines 20–27), and waits for the answers (lines 30 ss.).

4.5 Message handlers

Instead of using the agent main thread to handle incoming messages, as the agent PlusServer does, one can use message handlers. Message handlers are objects plugged in the MBox that "listen" messages which match a pattern and call back a method when a message arrives. The PlusServer agent could be coded as:

```
1  import saci.*;
2
3  public class PlusServer extends Agent {
4
5      public static void main(String[] args) {
6          Agent a = new PlusServer();
7          try {
8              if (a.enterSoc("APlusServer")) {
9                  a.initAg(null);
10                 //a.run(); no run for this agent!
11                 // It uses message handlers to anwser messages
12             }
13         } catch (Exception e) {
14             System.err.println("Error="+e);
15         }
16     }
17
18     /**
19      * initiate the agent
20      */
21     public void initAg(String[] args) {
22         try {
23             mbox.advertise("ask-one", "alg", "math", "X + Y");
24
25             // add a message handler that prints the arriving messages
26             mbox.addMessageHandler(null,null,null,null, new MessageHandler() {
27                     public boolean processMessage(Message m) {
28                         System.out.println("Msg="+m);
29                         return false; // other message handler also gives this message
30                     }
31                 });
32
33             // add a message handler to answer sum asks
34             // this handler filter is
35             //  . any content (null)
```

```
36              //   . performative "ask-one"
37              //   . language "alg"
38              //   . ontology "math"
39              mbox.addMessageHandler(null,"ask-one","alg","math", new MessageHandler() {
40                      public boolean processMessage(Message m) {
41                          try {
42                              System.out. println ("Adding..." );
43                              Message r = new Message("(tell)");
44                              r.put(" receiver ", m.get("sender" ));
45                              r.put("in-reply-to", m.get("reply-with"));
46                              r.put("ontology", m.get("ontology"));
47                              r.put("content", sum( (String)m.get("content" ) ));
48                              System.out. println ("sending" + r);
49                              mbox.sendMsg(r);
50                          } catch (Exception e) {
51                              System.err. println ("Error sending message\n"+e);
52                          }
53                          return true; // no other message handler gives this message
54                      }
55                  });
56
57          } catch (Exception e) {
58              System.err. println ("Error starting agent:"+e);
59          }
60      }
61
62
63      String sum(String formula) {
64          try {
65              formula = formula. substring (1,formula. length()-1);
66              int posPlus = formula.indexOf("+");
67              String opL = formula.substring (0,posPlus);
68              String opR = formula.substring(posPlus+1);
69              float fL = (new Float(opL)).floatValue ();
70              float fR = (new Float(opR)).floatValue();
71              return ""+ (fL + fR);
72          } catch (Exception e) {
73              return "Erro";
74          }
75      }
76  }
```

If an incoming message match the content, performative, language, and ontology of some handler (null means any), the *processMessage* method of this handler will be called. In case this method returns *true*, no other handler will try the message neither the message will be included in the mailbox.

4.6 Making agents applets

Implementing a Saci agent as an Applet is very simple: one might just extend the AppletAgent class instead of the Agent class. AppletAgent uses a remote MBox that runs on a web server host instead of the browser's JVM.[5] This is necessary because MBox needs special privileges to run (it is a message receiver server) and usually an Applet does not have them.

The applet agent could be coded as follow:

```
1  import java.awt.*;
2  import saci.*;
3  public class PlusClientApplet extends AppletAgent {
4
5  public void init () {
6      if (enterSoc("Ag1")) {
7          run();
8          leaveSoc();
9      }
10 }
11 public void run() {
12     try {
13         Vector receptores =
14         mbox.recommendAll("ask-one", null, "math", "X + Y");
15
16         if (receptores.size() == 0) {
17             add (new Label("There is no agent to answer me!"));
18         } else {
19             Message m = new Message("(ask-one :content \"2+4\")");
20             m.put("receiver", receptores.elementAt(0));
21             Message r = mbox.ask(m);
22             String ans = (String) r.get("content");
23             add (new Label(ans));
24         }
25     } catch (Exception e) {
26         System.err.println ("Error "+e);
27     }
```

[5]Thus, a launcher must be running on that web server host (see Section 5.1).

```
28  }
29  }
```

Although this program uses a remote MBox, MBox's methods are called as if they were local methods (lines 14 and 21).

To publish an applet at some web server, it is necessary to put in the appropriate web server directory (i) the agent's classes, (ii) a HTML page that calls the agent, and (iii) the saciapplet.jar file (this file contains all Saci's classes needed by a Saci applet). Here is an example of a HTML page that calls the agent coded above:

```
1  <h1>Demo Saci Agent Applet</h1>
2  <applet
3       code=PlusClientApplet
4       archive="saciapplet.jar"
5       width = 200    height = 50
6  ></applet>
```

Chapter 5

Saci's Tools

Last Chapter has shown how to program an agent using Saci's API and this Chapter will show how to maintain societies of agents using those agents. Saci has two kinds of tools: some that should be used in order to create agents and societies; and some that should be used to see what is happening in a society.

5.1 Launcher Demon

The launcher demon main purpose is to enable agents and societies to be started in a distributed environment. Every machine that will run agents has to have a launcher demon running (Figure 4.1 gives a good example of the launcher role). The launcher can be asked for the following services[1]:

- Agents and societies' facilitators creation in the local host or in a remote host. If launcher is asked to start an agent or society, it first checks if it is a local or remote creation. In the first case, the launcher demon creates the agent in its own JVM, and, in the second case, it looks for the launcher at the remote host and asks him to proceed the creation. This service is used by agent and society launcher tools (see Section 5.2).
- Remote mail box creation: in this case the agent's mail box will run in launcher host and not in agent's host (it happens with applet agents because MBox can't run in web browsers). It is also useful if the agents decide to de-active themselves and stop running, so the incoming messages will be received by the launcher.
- Kill an agent: the launcher kills the agent if it is local, otherwise ask the appropriate launcher to proceed the killing.

[1]It is a partial list, a complete and detailed launcher's RMI interface is available as java doc in Saci's home pages (see http://www.lti.pcs.usp.br/saci/doc/api).

In order to get a launcher running, the Saci'sMenu can be used (see Section 4.1.2) or the launcher can be started by the launcherd script. In the last case, the following parameters may be informed:

- -applications *filename*: indicates a file where a list of resources (agents classes, for example) used by an application are defined. The default value is "applications.xml".
- -facilitator [*societyName*]: starts a society's facilitator, optionally with a specific *society name*. The user may start several facilitator within the same command.
- -agent *class name* [*society*]: starts a new agent that will be created from *class* with preferred *name* and optionally in the society *society*.
- -connect *host*[:*port*] : connects to another launcher, so the other launcher will be known by this launcher.
- -registry *host*[:*port*] : specifies a host/port where the launcher could find a rmiregistry running. If not specified, localhost will be used.

5.1.1 Enabling an agent to be launched

Launcher demon can start agents that satisfy the following conditions:

1. The agent implements the LaunchableAg interface. Thus, an agent is launchable whether it extends the Agent/AppletAgent class or it implements this interface behavior.[2] This interface has the following methods:

   - enterSoc(*name*, *config*): this method has to implement the entrance in society with *name* and *config* (as saw in Section 4.2);
   - leaveSoc(): implements the leaving society act;
   - initAg(String[] args): initializes the agent before it starts running and gives it an array of arguments sent by the launcher;
   - run(): implements the agent behaviour;
   - stopAg(): this agent's method is called when the infrastructure or the user want the agent stops running;
   - getMBox(): returns the agent's mail box;
   - setProperty(String id, Object value): set an agent's property.
   - Object getProperty(String id): get an agent's property.

   When the launcher creates a new agent, it calls the LaunchableAg methods in the following order: enterSoc, initAg, and run. Optionally the launcher also calls getMBox, stopAg, setProperty, and leaveSoc. Note that agent's main method will not be executed when the agent is launched.

[2] It is very easy to give such an interface to an agent, see <saci dir>/src/saci/tools/Agent.java source for a sample.

2. A description of the agent has to be added in the appropriate application resource file (in the default file "applications.xml" there are instructions about how to give this description).

3. The agent's classes have to be found by the JVM where the launcher runs. So, before starting the launcher, either set the CLASSPATH variable to a directory where your agent's classes are or put your classes/jar files in the ulib directory (the ulib directory and its jar files are already in the classpath).

For example, considering the program of Section 4.1, which has the LaunchableAg behavior since it extends Agent class, the following commands will turn it launchable:

- put the agent's classes in the launcher class path:

  cp SampleSaciAg.class <saci dir>/ulib[3]

- add the following lines at the end of the "applications.xml" file:

```
<application id="Demo">

<agentType id="d1"
           description="My First Agent"
           class="SampleSaciAg" />
</application>
```

In summary, a Saci Agent can be started in many ways: as a Java application (Section 4.1.2, page 16), as an home-page applet (Section 4.6, page 27), or by the launcher agent tool (as seen in this Section).

5.2 Agent Launcher

Agent launcher is a launcher demon client. It asks launcher demon to create new agents. To do this operation, the launcher demon requires as parameters the agent's name, the agents's arguments, the agent's type[4], the agent's society, and the host where the new agent will run. Figure 5.1 shows how the user interface agent launcher gets these parameters:

- The first parameter (agent's type), is a choice list whose options the agent launcher has got from the launcher demon.

- The agent's society parameter is also a choice list whose options the agent launcher has got from the launcher demon. The demon makes this list from the societies it has created plus the societies created by other known launchers.

[3]The *<saci dir>/ulib* directory is always in the launcher demon class path. So, instead of setting the CLASSPATH variable, one can copy the classes to this directory.

[4]Indeed, it is the Java class that encodes the agent and implements LauchableAg interface.

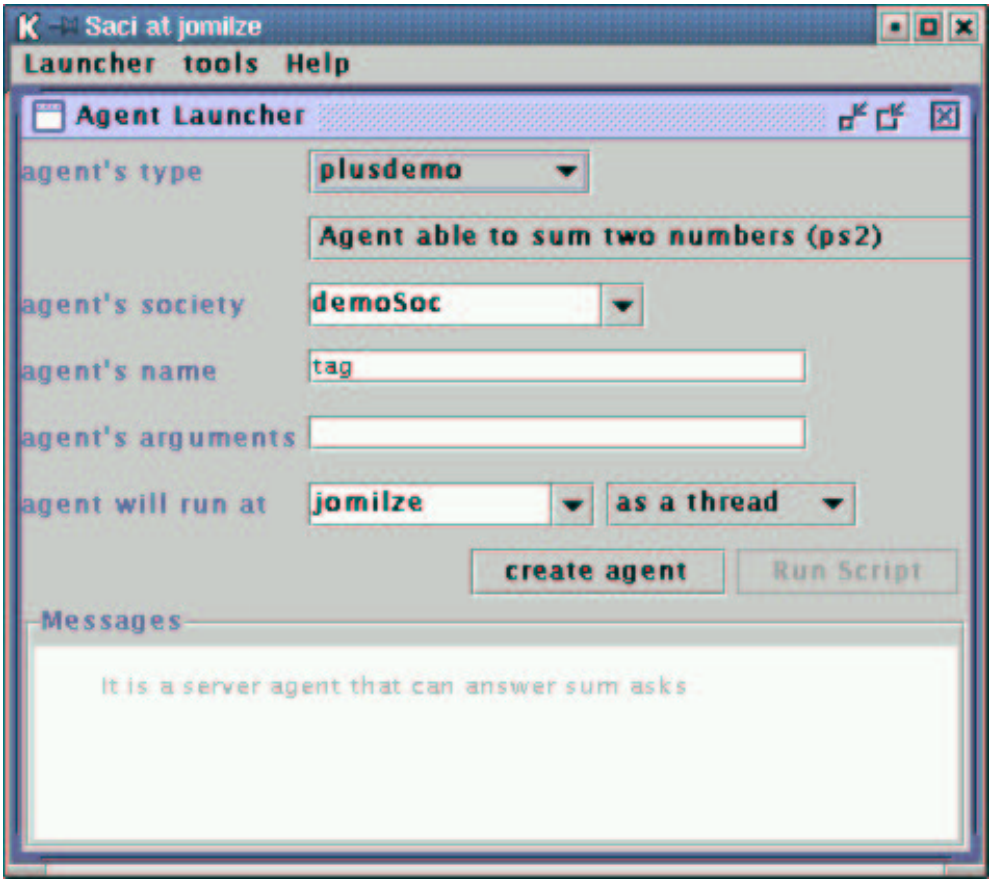

Figure 5.1: Agent Launcher Interface

- The host where the new agent will run is a text field or a choice list. The choice list is created from the information given by the launcher demon (i.e., the launcher demon's host plus the hosts it knows there are launchers running).
- The agent's name which is a text field.
- The agent's arguments which is a text field. The arguments must be separeted by spaces.

There is a status area, in the middle of the window, where the results of the creation process are shown. Below, both known agents and known societies are listed.

This agent launcher can also run as an applet in a home-page. In this case, one should add the folloing lines in a home-page in order to get the applet running:

```
7  <applet
8      code=saci.launcher.AppletAgLauncher
9      archive="saciapplet.jar,appletAgLauncher.jar"
10     width=600 height=500>
11 </applet>
```

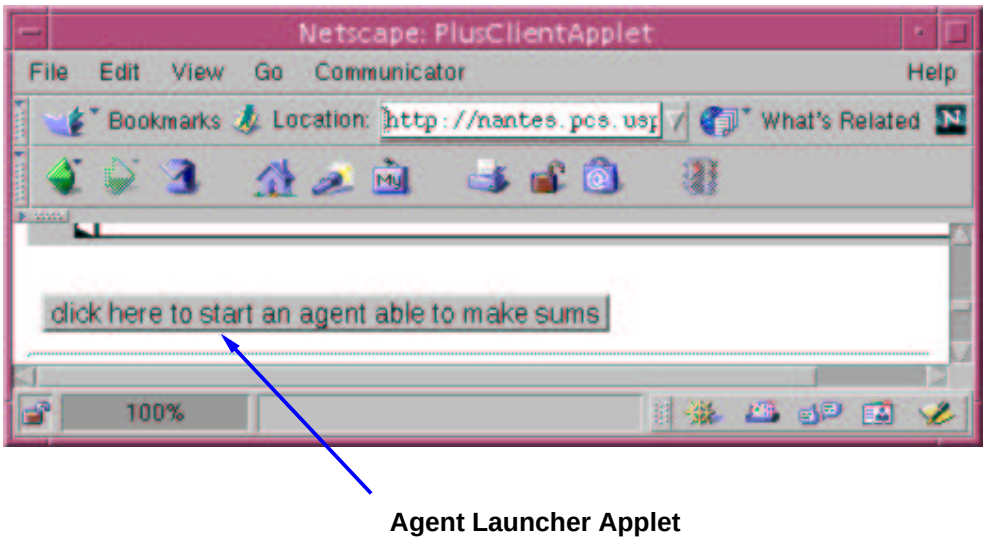

Figure 5.2: Agent launcher running as an applet.

Optionally, some parameters can be added to the applet tag:

- askHost: values can be true or false. If the value is true, the hosts choice is shown in the applet's user interface, otherwise the agent will run at the launcher's host. If the parameter is omitted, the default value is true.
- showStatus: values can be true or false. If the value is false, the applet's interface will not have the status area. If the parameter is omitted, the default value is true.
- agentSoc: value is a String with the society's name the agent will enter. If the parameter is omitted, the applet's interface will show the society choice list.
- agentName: value is a String with the name of the agent. If the parameter is omitted, the applet's interface will show the agent's name field.
- agentArgs: value is a String with arguments, separeted by spaces, for agent initialization (the agent will receive this value in his initAg method). If the parameter is omitted, the applet's interface will show the agent's argument field.
- agentType: value is a String with the Java class for the agent creation. If the parameter is omitted, the applet's interface will show the agent's type choice list.
- buttonName: value is a String with the text for the creation button. If the parameter is omitted, the text will be "create agent".
- askMethod: value can be true or false. If the value is false, the applet's interface will not give the user the choice for the creation method (thread or process). If the parameter is omitted, the default value is true.

For instance, the page

```
<applet
    code=saci.launcher.AppletAgLauncher
    archive="saciapplet.jar,appletAgLauncher.jar"
    width=200 height=50>
<param name="askHost"   value="false">
<param name="askMethod" value="false">
<param name="showStatus" value="false">
<param name="agentSoc" value="">
<param name="agentName" value="agT">
<param name="agentArgs" value="">
<param name="agentType" value="PlusServer2">
<param name="buttonName" value="Start a PlusServer agent">
</applet>
```

will show only a button with text "start PlusServer agent". When clicked, it creates an agent in default society with name agT, type PlusServer2, and which will run at the launcher's host (see Figure 5.2).

Instead of using the agent launcher, you can write a program that asks the launcher to create/kill/stop/move agents, run scripts, etc. For more information, see the launcher API documentation and the sample/geral/RemoteAgentCreation.java example.

Bibliography

[1] Luiz Otavio Alvares and Jaime Simão Sichman. Introdução aos sistemas multiagentes. In Cláudia Maria Bauzer Medeiros, editor, *Jornada de Atualização em Informática*, volume 16, chapter 1, page 1ss. SBC, Brasília, agosto 1997.

[2] Rafael Heitor Bordini, Renata Vieira, and Álvaro Freitas Moreira. Fundamentos de sistemas multiagentes. In Carlos Eduardo Ferreira, editor, *XX Jornada de Atualização em Informática (JAI)*, volume 2, chapter 1, pages 3–44. SBC, Fortaleza, CE, Brazil, 2001.

[3] Yves Demazeau. From interactions to collective behaviour in agent-based systems. In *Proceedings of the European Conference on Cognitive Science*, Saint-Malo (France), 1995.

[4] Yves Demazeau and Jean-Pierre Müller, editors. *Decentralized Artificial Intelligence*. Elsevier, Amsterdam, 1990.

[5] Stan Franklin and Art Graesser. Is it an agent or just a program? a taxonomy for autonomous agents. In Jörg P. Müller, Michael Wooldridge, and Nicholas R. Jennings, editors, *Proceedings of the 3rd International Workshop on Agent Theories, Architectures, and Languages (ATAL'96)*, Lecture Notes in Computer Science, Vol. 1193, pages 21–35. Springer, 1997.

[6] Nicholas R. Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, (1):7–38, 1998.

[7] Yannis Labrou and Tim Finin. *A proposal for a new KQML specification*. UMBC, Baltimore, 1997.

[8] Yannis Labrou, Tim Finin, and Yun Peng. Agent communication languages: the current landscape. *IEEE Intelligent Systems*, 14(2):45–52, March/April 1999.

[9] Gerhard Weiß, editor. *Multiagent Systems: A modern approach to distributed artificial intelligence*. MIT Press, London, 1999.

[10] Michel Wooldridge. Intelligent agents. In Gerhard Weiß, editor, *Multiagent Systems: A modern approach to distributed artificial intelligence*, chapter 1, pages 27–78. MIT Press, London, 1999.