Mica User's Guide

Mohammed Waleed Kadous
Claude Anthony Sammut
James Henry Westendorp

Mica User's Guide

by Mohammed Waleed Kadous, Claude Anthony Sammut, and James Henry Westendorp

Published 2005

Table of Contents

List of Figures

Chapter 1. Introduction

MICA is a toolkit that makes it easy to build applications that involve different modes of interaction, and different autonomous agents. MICA stands for Multimodal Interagent Communication Architecture[1].

It is being developed by the Smart Internet Technology CRC. It is specifically designed to support the following features:

- Write applications in such a way that the application itself is separated from the interface to that application; in this way different interfaces can be written for different modalities. This makes it possible, for instance, to write an e-mail agent; and then have both a voice interface and a GUI interface to it at once.
- Having the same information accessible across different devices and modalities.
- Maintainenance of state across different devices and modalities so that an interaction started on one device can quickly be moved to another.
- Combinations of both input modalities from various devices and output modalities.
- Support for learning, by exposing the interactions between agents so that learning programs can observe them.

To support these capabilities, MICA:

- Is a platform for exchange of data that is modality and device independent;
- Provides storage of information and state at a single logical "access point" (information may be physically distributed but the location should be hidden);
- Requires that communication between agents should be visible to learning agents.

Basic introduction to Mica

MICA uses a blackboard architecture to provide the capabilities discussed above. The blackboard concept was introduced by Erman et al (1980) in the HEARSAY-II speech understanding system. Since then, it has been used in a variety of AI systems and variations of the blackboard have been used to provide rendezvous mechanisms in multi-agent systems.

In MICA, we extend the blackboard to support:

- distributed execution;
- multiple devices;
- network connection;
- object-oriented storage;
- security and privacy.

In its simplest form, a blackboard is a shared memory. An agent may perform some task and post its results to the blackboard. When data of a particular type is written to the blackboard, that event may trigger another agent into action and this process repeats. The advantage of agents communicating indirectly through the blackboard is that they do not have to be aware of each other. This isolation makes it possible to introduce new agents or replace existing ones without affecting other agents. Furthermore, indirect communication allows the information going through the blackboard to be observed by third parties, which allows learning and knowledge acquisition.

[1]Actually, this is the latest version of the acronym. Previously it used to stand for "Multimodal Internet Conversation Architecture", but then it was realised (a) it wasn't restricted to the internet (b) it wasn't restricted to conversation either. But the name has stuck, in any case.

In practice, blackboards have more structure than a simple shared memory. The memory is often divided into different regions and some agents may be restricted to accessing only certain regions. Blackboards also commonly include an agenda mechanism for scheduling events.

In MICA, an object-oriented approach is taken to the blackboard; items written to the blackboard have a particular type. Types may be inherited.

The blackboard is structured as follows:

- Agents connect to the blackboard.
- Agents register interests in particular types of objects.
- Agents write objects to the blackboard.
- The blackboard manager advises registered agents of new objects.
- Agents can search blackboards using a query language (in the present implementation, the query language is based on SQL).

Generally, there are several types of agents: *interface agents*, which provide a way of interacting with the system's user, *computation agents* that provide services such as e-mail, or text-to-speech and *environment agents* that report information about the user's context, e.g. GPS trackers. There is no formal distinction made between these types of agents, but it goes some way to showing the variety of different systems that can be connected to MICA.

The blackboard provides two important functions: information storage -- by writing information to the blackboard; and information communication -- by informing other agents when objects are written to the blackboard. The fact that they are both done using the same mechanism has several advantages. It means that, for example, a GPS tracker can report information to the blackboard, even if no one is listening, and later agents can "track back" through older location information. Agents that connect to a discussion "late" can retrieve state from before they connected. Agents can also "watch" the blackboard and contribute to the information on the blackboard -- e.g. learning agents.

Comparing MICA to other systems

MICA can be compared with many other systems; since the problem MICA addresses -- storage and communication, are two of the universal problems in computer science. To help show how MICA fits in with these alternatives, it is compared here with existing systems.

MICA as a type of database

MICA can be thought of as a database, in fact, to be accurate, an object-oriented active database. It is object-oriented, because objects can be stored directly in the database, and active, because active databases support the execution of code under certain conditions when the database changes. When clients write objects to the blackboard, this can be thought of as writing a record to a table in a database. Similarly for reading, deleting and querying.

The important difference is the way that MICA conveys information about changes to the database to interested third parties; who wish to be informed about whatever changes occur to a database.

There are to our knowledge, no active object-oriented databases particularly ones that communicate information of changes to the database.

MICA as a Publish/Subscribe Model

Another way to think of MICA is as a publish/subscribe model of interaction (like the InfoBus architecture), or an Observer pattern in languages like Java or as a content-based routing system (such as elvin).

The main difference between MICA and such systems is that MICA integrates storage into such an architecture. Though this change seems small, it does actually have some significant implications.

MICA as a web service and/or RMI and/or RPC

The popularity of XML-based web services has increased dramatically over the last few years; but really they are examples of general remote procedure calls systems.

The main differences compared to MICA are that these systems are "one-way" -- queries are launched by clients to the server, and the client gets a response; and secondly, that there is no inherent storage capability built in -- though it would be possible to build one.

Having said that, it would be possible to implement MICA as a two-way web service. It would be messy, but possible.

MICA compared to CORBA

CORBA could be used to implement a MICA service; however, CORBA is incredibly complex and this would be like implementing a simple counter with a Pentium processor. Further, though the protocol supports more advanced forms of interaction than the client-server protocol mentioned above, such features are rarely implemented in CORBA systems.

MICA as an agent architecture

MICA can be considered a type of agent architecture. Compared to most other agent architectures, however, it is much simpler. MICA is like an agent architecture where the only elements allowed in the agent communication language are objects.

Having said that, it is easy to interface non-MICA agents to MICA. For comparison, we compare it to a popular agent architecture: SRI's Open Agent Architecture.

In many ways, MICA is a simplified form of OAA. MICA could be implemented using OAA, but it would be like trying to build a lower level abstraction on a higher one -- possible, but for serious work, you want a clean implementation. There are also a few other differences in security and data model.

OAA has a "facilitator" agent that manages interaction between different agents. In MICA, this is the blackboard manager. OAA uses the idea of "solvables"-- things or queries that can be resolved. So an agent in OAA says that it provides a certain solvable to the facilitator. In MICA, solvables correspond pretty closely to objects. In MICA, an agent tells the blackboard to inform it of particular objects it is interested in.

The main differences are:

- MICA has no backtracking across agents to solve problems; whereas in OAA, you can post compound goals to the facilitator, which will then employ depth-first and/or breadth-first search to fulfill those goals. For example, in OAA, if you tried to make an enquiry like "Get me Bill Smith's manager's phone number" you could say:

  ```
  oaa_solve(manager("Bill Smith",M),fax(M,R))
  ```

  Figure 1.1. A typical OAA query

  (R is the return value here, and M is an "intermediate" variable) The "manager" solvable might be handled by one agent, and the "fax" solvable might be handled by two different agents, each with access to different fax numbers. Backtracking would allow this to be done seamlessly.

- OAA's security model is agent-based; ie. "only the agents I specify can use my solvables". MICA's is object-based, i.e. "this particular object should not be visible to this agent". The blackboard manager in informing agents or responding to their queries, has a policy expressed as a function of the agent to which the object is to be delivered and the object itself.

- MICA uses object inheritance to model objects on the blackboard; OAA can use any data structure supported by ICL (which is basically Prolog).
- OAA uses "procedural solvables" and "data solvables" -- the MICA equivalent for both of these is an Object.
- OAA specifies four different types of triggers: data triggers (tell me fact X), communication triggers (tell me when agent X talks to agent Y about Z), timed triggers (tell me every 5 seconds) and task triggers (tell me when someone wants to do X). MICA has one trigger only: tell me when an class X or one of its subclasses appears on the blackboard.

The manual

The remainder of this manual is designed to help users get started with MICA. It begins with a quick outline of the MICA design, before discussing the prototype implementation. It then walks through the code in order to build a simple MICA agent for implementing a shared notepad. A more complicated example, with five interacting agents, is then demonstrated.

Chapter 2. The MICA Design

Basic entities

The MICA system consists of several entities; and it is their interplay and coordination that makes it useful. There are really only three basic entities used in the MICA design:

The Blackboard

The blackboard is the core of MICA. An analogy can be made in the case of the blackboard to the old switches that used to be employed to handle telephone calls. The blackboard is similar to a switch with memory -- facts can also be stored on the blackboard. But all interactions between agents flow through the blackboard, in much the same way that all phone calls in a town go through a switch.

Much of the power of the blackboard comes from its ability to allow many agents to share information. It is expected, that in use, there would be one blackboard for each person. All agents and services related to that user would execute through the blackboard. It is also expected that forum blackboards, for groups of individuals, could also be created.

Agents

Agents are entities that wish to access and contribute to the information on the blackboard. As previously mentioed, there are several types of agents, but to the blackboard, they all look the same.

Agents take many shapes and forms. It may be a complex GUI running on a person's desktop computer, or it may be a proxy for an incoming call over the phone system. It may even be a physical device. It may be autonomous, with full reasoning and deliberative capabilities, or it may be a simple "time" agent, which writes "time" objects to the blackboard every ten seconds.

Agents *connect* to the blackboard. Once connected, clients can do several things:

- Write something to the blackboard.
- Read something from the blackboard.
- Query objects on the blackboard.
- Register interests for new things written on the blackboard. When a new thing -- of interest to the agent -- is written to the blackboard, the agent is informed of its arrival.

MICA Objects

MICA Objects (or *mobs* for short) are the basic unit of information in MICA. Mobs are the things that clients actually read and write from the blackboard.

Mobs are similar to objects in an object-oriented programming language. They have the following characteristics:

- *Name:* Each mob has a name, which is unique. Mobs are named when they are first written to the blackboard. The name is a string.
- *Type:* Each mob has a type. Much like an object-oriented programming language, types can inherit from one another. MICA's type system is discussed in greater detail later.
- *Slots:* Slots are similar to fields in object-oriented oriented programming; with one small exception: slots can have a list of values associated with them. Each slot has a name, and a list of values associated with it.

Typically, each type has one or more slots associated with it. Subtypes will typically have the slots of the base type, but will typically have additional slots.

A simple example of a mob is the `sharedPadLine` type used in the sharedPad demo (which is discussed in Chapter 5). It has four slots for each of the properties of a line: `oldX, oldY, newX, newY`. In this case, each slot only has a single value. A subtype might be `colouredPadLine` that in addition to the slots for a `sharedPadLine` above, also has a slot `colour` to describe the colour of the line.

Putting the pieces together

A valid question is how do all these pieces fit together. Figure 2.1 shows what this looks like.

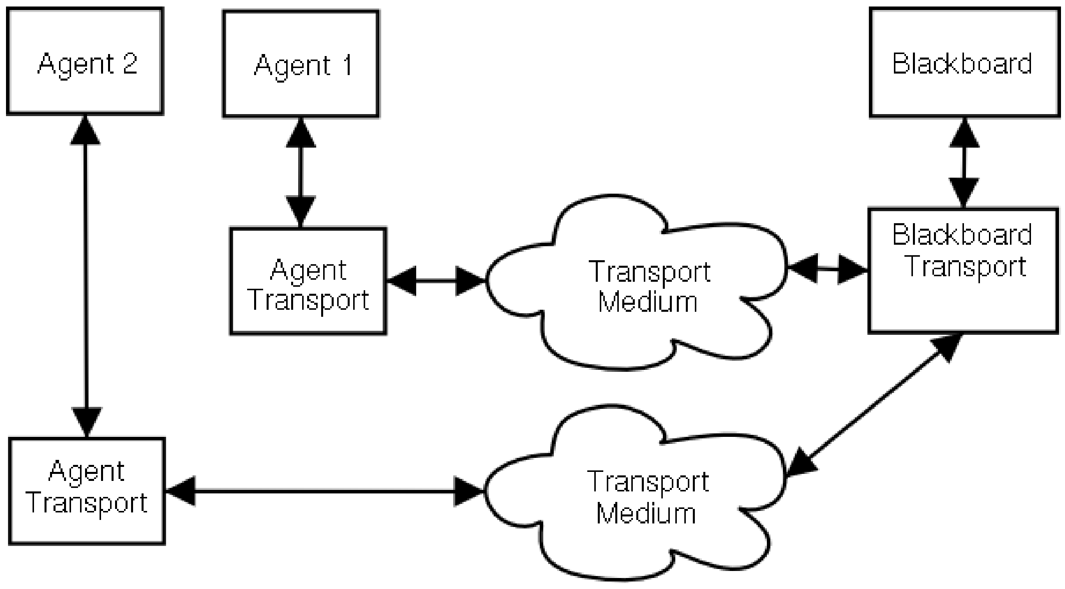


Figure 2.1. How MICA components are connected

In the Figure 2.1, the arrows represent conduits for messages. These messages are requests that do things such as request that an object be written to the blackboard, or that an agent be informed that a newly created mob.

The diagram shows two agents connected to one blackboard. Typically, there would be more than two agents connected, but two are the minimum number for any interesting application. Agents and blackboards do not communicate directly, but through a transport layer. This allows implementations of MICA to use different transport layers. For example, some possible transport layers are: local function calls, XML over a TCP/IP connection, some proprietary encrypted protocol, or ASCII messages over Bluetooth. For each one of these, a pair of classes -- an agent transport and a blackboard transport -- need to be developed. Agents talk to the Agent Transport layer through method calls; and likewise for the Blackboard.

The advantage of doing things in this way is that it gives a lot of flexibility in the domain MICA can be applied to; and allows developers to write agents that do not need to be aware of the underlying transport layer.

In order to clarify how requests are passed, consider what happens when Agent 1 writes a mob to the blackboard, which is of type `Message`. Also assume that Agent 2 has registered for any mobs whose type is `Message`. Agent 1 would create the mob and call an appropriate method in its agent transport. The agent transport would deliver it to the blackboard transport (using whatever transport medium was appropriate). The blackboard transport would do two things: firstly, it would give a name to the mob; say `Message_0`, and inform Agent 1 -- through Agent 1's transport -- of the new mob's name. Secondly, since Agent 2 was registered for mobs of type `Message`, the blackboard would send a message to Agent 2's transport, that there was a new mob called `Message_0` with certain slots. Agent 2's transport would then call a method in Agent 2 to handle the newly arrived Mob.

Note the following:

- Neither Agent 1 nor Agent 2 are explicitly aware of one another.
- Many different agents can register for mobs of a given type. There could be a dozen agents connected to the blackboard, and if six of them were registered for `Message`s, then all six of them would get it.
- The arrows are "two way" arrows. At times the blackboard will initiate communication with an agent, and at other times, it will be the other way around.
- Agents can register for mobs of any type, and they will be informed if mobs of that type or any sub-type are written to the blackboard. For example, if `ShortMessage` was a subtype of `Message`, and someone wrote a mob of type `ShortMessage` to the blackboard, Agent 2 would still be informed.

Chapter 3. Installing and running MICA

Getting MICA

MICA is currently CRC internal software; so the usual delivery mechanism at this point in time is via an e-mail attachment.

MICA requirements

MICA should run on any system that supports JDK1.3 or better. It is most extensively tested on Windows XP, but is also tested on Linux Redhat 9 systems.

Installing MICA

To install MICA, take the tar.gz (where version is the current version) file and use the following commands to unzip it:

```
% jar xvfz mica-2.0-{version}.jar
```

This sets up a subdirectory **mica-2.0-{version}** that contains all the mica files and documentation. You should move into that directory. The next step that needs to be completed is that the classpath must be set up. You should add the following jar files to your classpath (all from the jars directory):

- `hsqldb.jar`
- `mica.jar`
- `weka.jar` (optional: only necessary for supporting the LearningAgent)
- `framescript.jar` (optional: only necessary for supporting the WhizBang demo)

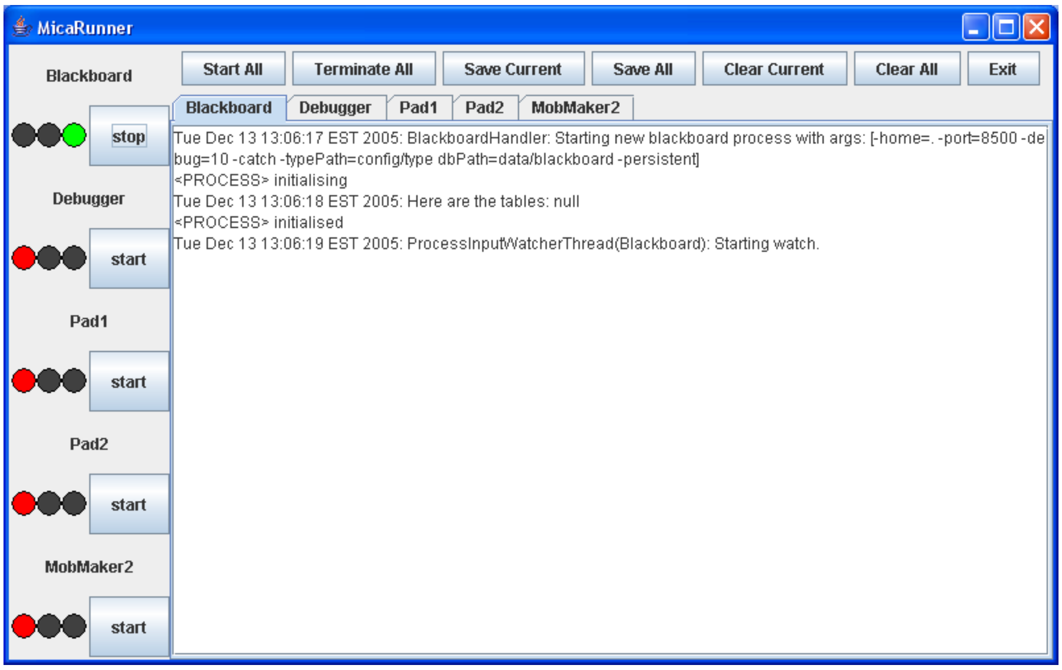
Exactly how you do this depends on your operating system. For example, under Windows, it can be done using (excluding the optional jars, don't forget to include them if you need them):

```
set CLASSPATH=jars/hsqldb.jar;jars/mica.jar
```

Under bash (e.g. Linux or MacOS) it might be:

```
export CLASSPATH=jars/hsqldb.jar:jars/mica.jar
```

Once you have set up the classpath, you can now start the MicaRunner. MicaRunner is a graphical user interface that allows the blackboard and the agents to be started independently. A picture of the MicaRunner is shown in figure Figure 3.1.



```
Tue Dec 13 13:06:17 EST 2005: BlackboardHandler: Starting new blackboard process with args: [-home=. -port=8500 -de
bug=10 -catch -typePath=config/type dbPath=data/blackboard -persistent]
<PROCESS> initialising
Tue Dec 13 13:06:18 EST 2005: Here are the tables: null
<PROCESS> initialised
Tue Dec 13 13:06:19 EST 2005: ProcessInputWatcherThread(Blackboard): Starting watch.
```

Figure 3.1. MicaRunner window

MicaRunner takes a file that tells it which agents it should load. In this particular case, in the mica directory, a file called `sharedpad-run.xml` specifies which agents to load. MicaRunner can be run by typing:

% java unsw.cse.mica.runner.MicaRunner examples/run/sharedpad-run.xml

Once loaded, hit the "Start all" button to start the agents. You should now see a total of three windows (they may actually be overlapping, so move them as appropriate). For the moment, put the "Mica Blackboard Display" window aside, and just move the Shared Pad windows into a convenient place. Drawing in one window should lead to lines appearing in the opposite window and vice versa. In this case, both of the SharedPad windows are clients that connect to the blackboard. When you drag the mouse in either window, then the client creates a mob with a description of the line and writes it to the blackboard. Since SharedPad clients register their interests in lines, both clients are informed of the newly added line and draw it on their canvases. If you now look at the "Mica Blackboard Display" window, and click on the "MICA Objects" node, there is a complete list of all the Mobs that have been written on the blackboard.

A more detailed look

Let's have a look at `sharedpad-run.xml` in a little more detail:

```
<runner>

  <host name="localhost"/>
  <port number="8500"/>
  <pos x="0" y="0" />
  <size width="800" height="500" />
  <catch value="false" />

  <blackboard/>

  <agent class="unsw.cse.mica.tools.Debugger">
    <needs name="blackboard"/>
    <arg param="scroll" value="true" />
    <arg param="x" value="0" />
    <arg param="y" value="500" />
    <arg param="width" value="500" />
    <arg param="height" value="400" />
  </agent>

  <agent class="unsw.cse.mica.demo.SharedPad" name="Pad1">
    <needs name="blackboard"/>
  </agent>

  <agent class="unsw.cse.mica.demo.SharedPad" name="Pad2">
    <needs name="blackboard"/>
  </agent>

  <agent class="unsw.cse.mica.tools.MobMaker2">
    <needs name="blackboard"/>
    <arg param="x" value="500" />
    <arg param="y" value="500" />
    <arg param="width" value="300" />
    <arg param="height" value="400" />
  </agent>
</runner>
```

Figure 3.2. `sharedpad-run.xml`

The first line of Figure 3.2 declares that the file is a runner file. the next two lines specify the particular blackboard to connect to, in this case the current machine running on port 8500. Specifications are also be given for the location for the position of the runner window, using the . This is followed by the specification of the blackboard. In this particular case, the blackboard does not try to restore the information on the blackboard from the last time it ran, and it also specifies the amount of debugging information that is given.

The others are entries for individual agents. The first is for the debugger tool (that allows you to see what's on the blackboard). The second and third are two instances of the SharedPad.

The Shared Pads need not run on the same computer as the blackboard. If you had two computers and installed MICA on both, then you could have Shared Pads running on different machines, with a slightly different version of the `sharedpad-run.xml` file (i.e., without the **blackboard** tag, and with a **runner** tag that pointed to the appropriate remote blackboard). Nor are you limited to only two, you could have as many shared pads as you liked.

More documentation on MicaRunner can be found in the API documentation, and more examples of the kinds of things that go in a run file can be seen in the `examples/run` folder of the mica distribution.

Chapter 4. The MICA implementation

Parts

MICA is implemented in Java. It consists of a number of packages. But when you get down to it, for implementing agents, there are only a few classes that matter:`Agent, AgentTransport, XMLOverTCPAgentTransport` from the `unsw.cse.mica.agent` package; `Mob` from the `unsw.cse.mica.data` package; `Blackboard, SQLBlackboard and XMLOverTCPBlackboardTransport` from the `unsw.cse.mica.blackboard` package. Also of use may be the `DefaultAgent`, `DefaultAgent2` and `GUIAgent` classes, which provide various partial implementations of an agent.

All of these classes have extensive Java documentation, so you may also wish to examine the javadocs that are included with MICA.

A quick walk through the MICA API

There are several important classes that make up the MICA implementation; in this section, we will go through the most important classes and their methods.

`unsw.cse.mica.data.Mob`

The Mob is the basic unit of information storage and communication in MICA. As previously mentioned, each Mob has a name, a type and a set of slots that define it. Let's have a look at two real Mobs (as output by its `toString()` method).

```
Mob sharedPadLine_103 of type sharedPadLine has slots:
  creationTime: [2003-08-07 12:13:21.246]
  creator: [sharedPad_1]
  oldX: [69]
  oldY: [151]
  newX: [68]
  newY: [151]
```

Figure 4.1. A simple mob

In the above code, the mob's name is `sharedPadLine_103` (a name that was allocated when this object was written to the blackboard) and its type is `sharedPadLine`. It has six slots, but all of the slots have a single value. Let's look at a more complex example.

```
Mob emailListReply_12 of type emailListReply has slots:
  creationTime: [2003-07-31 16:01:02.24]
  creator: [emailAgent]
  from: [waleed@cse.unsw.edu.au, foo@bar.com, bar@baz.com]
  subject: [What's up?, Bugs in MICA, Romeo and Juliet screening]
  count: [3]
```

Figure 4.2. A more complex Mob

Figure 4.2 shows slots containing multiple values. The current implementation allows only Strings to be stored in slots, but this is still a very flexible representation. For example, references to other mobs and complex data structures can be constructed using slots that contain references to other Mobs. Similarly, binary data can be stored using base-64 encoding.

The methods for mobs are consequently related to constructing a Mob. Basically, there are several families of methods. The complete documentation can be found in the java documentation.

- *Constructors:* The typical constructor used to make a Mob is the `Mob(String mobType)` method. This creates a new Mob of the specified type, but with an undefined name and no slots. Typically, the Mob has no name until a name is actually given to the Mob by writing it on the blackboard.
- *Setting up slots:* There are a number of functions to set up slots. `setSlot(String slotName, List slotValues)` allows you to set up a slot with all of its values, while `addSlot(String slotName, String slotValue)` allows you to add a new value to a slot. `addEmptySlot(String slotName)` allows you to set up a slot that has no values in it.
- *Getting values from slots:* There are two primary methods for getting values from slots: `getSlot(String slotName)` gets all the list of values for a slot, but since there is a very common case where the list contain a single value, there is also the `getSlot1(Strong slotName)` which returns a single String, rather than an entire list. To get a slot value as an integer, use `getSlot1AsInt(String slotName)`.

Transience

On occassion, you want to use MICA merely as a traditional publish/subscribe mechanism, i.e. you do not want to save the mob you are writing on the blackboard, you just want that mov delivered to anyone who is interested. Such mobs can be labelled as transient. Transient mobs are not stored on the blackboard, but other agents interested in the mobs are notified. To mark a mob as transient, use the `makeTransient()` method.

Note that as of Mica 2.0, transience can also be defined using the type manager. If a type is specified in this way, the blackboard will ensure that all Mobs of that type are transient, so agents creating these mobs are not required to do so.

Reserved slot names

The following slot names are reserved and are used by the system: `creator, creationTime, deleter, deletionTime` and `persistence`. Setting these slot names can lead to unexpected events and should be avoided.

unsw.cse.mica.agent.AgentTransport

`AgentTransport` is an interface that represents a connection to the blackboard. As such, the `AgentTransport` is the only "view" that the agent actually gets of the blackboard. For this reason, the `AgentTransport` encapsulates all the functionality of the board. Whenever an agent wants to communicate with the blackboard it does so through the agent transport. Similarly, whenever the agent transport receives information about a new mob, this is passed from the agent transport to the agent.

Because the `AgentTransport` is a proxy for the blackboard, all its methods are basically proxy commands that are forwarded to the blackboard. These are:

- *Connection commands:* `connect(String agentName)` and `disconnect()` are used to connect to the blackboard. `connect()` returns the agent name that was given to the agent. The agent can request a particular name, but this may not be granted since another agent with this name may already exist. `disconnect()` obviously disconnects from the blackboard.
- *Getting type information from the blackboard:* Agents can access the blackboard's type manager using the `getTypeManager()` command.
- *Registering and unregistering:* Registering is the way that you let the blackboard know that you are interested in objects of a particular type. For instance, if an agent is interested in lines, it registers for `sharedPadLine` mobs. Whenever a `sharedPadLine` mob is written to the blackboard, the agent is sent a copy of the new mob. To register, the `register(String mobType)` can be used. Similarly, if an agent is no longer interested in `sharedPadLine`s anymore, it unregisters with `unregister(String mobType)`
- *Writing things on the blackboard*: To write on the blackboard, `writeMob(Mob m)` can be used. `writeMob` returns the name that the object is finally given. The blackboard then handles the forwarding of this message to anyone who is interested -- the agent need not concern itself with how the information is distributed.

- *Deleting mobs from the blackboard*: `deleteMob(String mobName)` can be used to delete information from the blackboard. However, this should be done very carefully, since other mobs may refer to the mob.

- *Getting mob information from the blackboard*: To retrieve information from the blackboard, one can use `readMob(String mobName)` if the name of the mob is known or `mobSearch(String micaQuery)` if a mob with particular properties is sought. For a description of the query language, see the section called "The MICA query language". `mobSearch` will return all mobs that have the desired properties.

`Mica V2 provides two primary implementations of the AgentTransport interface:`

- LocalAgentTranport provides a direct programmatic connection between the agent's transport and the blackboard transport. Note that this implementation has not been thoroughly tested.

- `CompoundTransport` combines a connection type (`AgentConnection`) with a protocol (`AgentProtocol`) to create a versatile component-based transport system. Currently available connection types are `LocalAgentConnection` and `TCPAgentConnection`, while the only protocol currently implemented is `XMLAgentProtocol`.

- `XMLOverTCPAgentTransport` sends snippets of XML over a TCP network connection. This is the most frequently used transport method, so much so that a new class has been created to simplify the process of creating this transport type. Currently this is the only type of transport used by MicaRunner.

unsw.cse.mica.agent.Agent

`Agent` encapsulates a single autonomous unit for doing computation on the blackboard. It is implemented as an interface. Implementing an agent consists of writing seven methods; and `DefaultAgent` provides a reasonable default for most of those.

- `setAgentTransport(AgentTransport at)` and `getAgentTransport()` are used so that the agent can redirect calls to the blackboard through the appropriate agent. `DefaultAgent` provides a default for these two methods. In general, `DefaultAgent` should be extended, but in some cases it is more convenient to implement `Agent` so that the agent itself can inherit from another class.

- `init(MicaProperties)`: This method is called to start the agent operating. It is actually called by the `AgentTransport` (see the section below). Note that the `init()` method should return quickly; in particular, the agent transport should not start sending messages until `init()` returns. If some complicated execution takes place, it should be done in another thread. Also note that the `init()` method takes a MicaProperties object; these are usually extracted from the startup file; but it provides a generic mechanism for passing configuration info to the agent.

- `terminate()`: This method is called to stop the agent operating. It is typically called by the MicaRunner.

- `handleNewMob(Mob m)` This method is called whenever a Mob we are registered for arrives.

- `handleDeletedMob(Mob m)` This method is called whenever a Mob we are registered for is deleted. The mob being deleted is included. This is useful, as this gives us a last opportunity to make a local copy of the Mob. If you inherit from `DefaultAgent`, there is a default implementation that does nothing.

- `handleTypeManagerChanged()` this method is called whenever the blackboard's type manager changes. The defaul is to do nothing.

That's it! At its core, to implement most agents requires writing the the `init` and `handleNewMob` methods.

`DefaultAgent2` provides additional functionaliy over `DefaultAgent`, including transport connection and disconnection, and providing a type manager that is kept up-to-date with the blackboard's type manager. `GUIAgent` extends this further to provide skeleton support for an agent needing a GUI. It ensures that terminating the agent closes the GUI and vice-versa.

Setting up Agents and Agent Transports

The following section discusses how agents and agents are set up. However, if you use the `MicaRunner` tool, you need not concern yourself with such details; it handles the setting up of agents and agent transports. On first reading, this section can be glossed over.

Because the `Agent` and the `AgentTransport` have a close relationship and each can call the other, there are a few special hoops that have to be jumped through in order to get them to work together. This is because each need to know about the other, so it's tricky to set this up with constructors and the like. The steps involved are:

- Construct the `Agent`.
- Construct the `AgentTransport`, but ensure one of the parameters is the the Agent constructed in the previous step.
- In the constructor of `AgentTransport` call the method `Agent.setTransport(AgentTransport at)`. This lets the agent know what its transport is.
- When the agent is ready to go, call the method `Agent.init()` This is typically the point in the code when the agent will connect to the blackboard using, say, an `AgentTransport.connect()` call.

The MICA type system

MICA's type system -- the way that different objects are represented -- is similar to many object-oriented systems. As previously discussed, objects have a type; and types can inherit from one another. Each type typically has certain slots associated with it. Each slot can have a list of values.

The current implementation of MICA is simple. Firstly, slots can currently only have values that are Strings. Ideally, it would be useful to have slots that can have other types, e.g. ints, doubles, and so on[1]. Secondly, although it would be typical for a particular type to have particular slots, this is not enforced in any way. For example, consider the `sharedPadLine` mob in Figure 4.1. Although we expect that a sharedPadLine would have the six slots shown, it is perfectly possible to construct a `sharedPadLine` that has no `oldX` slot.

MICA supports multiple inheritance: a given type can have several superclasses. Unlike languages like Smalltalk where each object can have multiple types, however, a single mob can only have one type; although any type may have several supertypes.

To simplify management, there is a universal supertype, called "mob". Every type is assumed to inherit from "mob". When an object is created on the blackboard, the blackboard also creates two slots, each with a single value: `creationTime`, the time at which the agent was created; and `creator`, the agent that created a particular mob.

It is also important to note that the registration system heeds inheritance; in other words, if an agent registers for a mob of a given type, the agent is informed if any subtype is also written to the blackboard. This is an extremely important feature. For example, it makes writing a "debugger" agent that displays everything on the blackboard very easy. All the agent need do is register for any mobs that are of type "mob" and it will be informed of *any* mob written to the blackboard. Similarly, say some kind of complex agent has a whole family of mob types that it has to listen to. If all the mobs inherit from a common type, then it is easy to register for a whole family of mobs in a single go.

A further use of the type hierarchy is as a way of specifying transience. Although individual mobs can be specified as transient by the makeTransient method, it is often the case that all mobs of a particular type will be transient. If a mob type is specified as transient then all mobs of that type will be automaticlly made transient by the blackboard. This feature also obeys type inheritance, so that if some type is declared transient, all type inheriting from it will also be transient.

[1]One particularly useful type would be a reference to another Mob. However, similar functionality can be achieved by using the name of the mob and storing it in a slot as a string, and then using the `readMob()` method to find the useful information.

Using the Blackboard

Agents never directly see the blackboard, but only the AgentTransport. Thus, the interaction of most agents and the blackboard are limited. Usually, all that the developer needs to know what to do with the blackboard is to learn how to start and stop it.

The simplest way to start a blackboard is via MicaRunner. Simply adding a blackboard tag to the runner XML configuration file will provide support for a blackboard running on the local machine.

The alternative is to start a blackboard manually from the command line. For an SQL Blackboard with a XML-over-TCP transport, this is simply done using the `XMLOverTCPBlackboard` class:

% java unsw.cse.mica.blackboard.XMLOverTCPBlackboard

This starts a blackboard transport at port 8500 on the local host, with all the default settings. The default settings can be overridden using the following command-line parameters:

- **-port=PORT** specifies an alterative port number to use
- **-micaHome=DIR** specifies an alternative home directory for mica
- **-typePath=DIR** specifies an alternative directory (relative to the micaHome directory) to search for type specification files
- **-dbPath=DIR** specifies an alternative directory (relative to the micaHome directory) to use for the SQL database
- **-persistent=BOOLEAN** specifies whether or not the blackboards should attempt to load stored mobs on entry or save them on exit.

Configuring MICA

Agents can be configured by using the **arg** in the XML run files. Any such tags are passed to the `init()` method through a **MicaProperties** object. An example of a complex snippet of a MicaRunner XML file that shows an agent that takes multiple parameters is shown in figure Figure 4.3. These can be accessed through the methods of the `MicaProperties` class. Note that multiple parameters with different values are allowed; in this case to load different files into MicaBot.

```
<agent class="unsw.cse.framescript.mica.MicaBot">
  <arg param="file" value="scripts/system_mica.frs"/>
  <arg param="file" value="scripts/numbers.frs"/>
  <arg param="topic" value="all"/>
  <arg param="init" value="init"/>
  <debug level="information"/>
</agent>
```

Figure 4.3. A MicaRunner XML file snippet with complex arguments

Also note that the MicaRunner file supports a **home** element so that you can choose the directory that MICA is installed in. This is if you want to run agents in another directory. The MICA home directory is used for several purposes, namely to load data from `config/type` folder related to types (the **LearnerAgent** also uses the `config/learntask` folder to load learning tasks). The `data` folder from the mica home directory is also used to store temporary objects, like blackboard data files, or learnt concepts.

Giving information about types

The current implementation of MICA uses types that are read in at the beginning of the blackboard's execution. Future versions are likely to allow types and inheritance to be defined dynamically. But for now, on startup, any xml files in the type directory (normally `config/type`) are read in to define the hierarchy. Figure 4.4 shows a typical type definition file, for the problem of defining a hierarchy of shapes.

```
<typedesc>
  <mobdecl name="object"/>
  <mobdecl name="shape">
    <parent name="object"/>
  </mobdecl>
  <mobdecl name="polygon">
    <parent name="shape"/>
  </mobdecl>
  <mobdecl name="rectangle">
    <parent name="shape"/>
  </mobdecl>
  <mobdecl name="circle">
    <parent name="shape"/>
  </mobdecl>
  <mobdecl name="square">
    <parent name="rectangle"/>
    <parent name="polygon"/>
  </mobdecl>
</typedesc>
```

Figure 4.4. `shapes.xml`

As can be seen, each type declaration consists of a description of a mob, and then its parents. For example, `circle` inherits from `shape`, and `square` inherits from both `rectangle` and `polygon`. If a type is to be defined as transient, simply add the attribute `persistence="transient"` to the declaration. For example, to make all the types in the above example transient, define the object type as `<mobdecl name="object" persistence="transient" />`.

The MICA query language

The MICA query system takes advantages of the storage mechanism used by MICA. MICA uses HSQLDB as a database store in which it places all of the MICA objects. MICA's query language takes advantage of the HSQL interface. Without going in to too much detail, MICA objects reside in a SQL table with three columns: the name of the object, the type of the object and the object itself.

To use the MICA Query language, normal SQL has been enhanced with the following functions that are applicable to mobs. They closely mirror the methods that are available to the `unsw.cse.mica.data.Mob` class.

- `typeof(mob, 'type')` allows you to check the type of an object. This mthod returns a boolean.
- `hasslot(mob, 'slotName')` allows you to check whether a mob has a particular slot or not.
- `getslot1(mob, 'slotName')` gets the first value of a slot. It returns a string. There are two further versions of this method, `getslot1asint` and `getslot1asdbl` to get the first value in a slot as an integer and double respectively.
- `getslotn(mob,'slotName',pos)` is a useful method for getting arbitrary information from a multi-valued slot.
- `contains(mob, 'slotName', 'value')` allows you to check whether a mob has a particular value stored somewhere in a multi-valued slot.

Example uses

In order to use the system, consider the following examples that actually occur in the MICA codebase.

Getting all Mobs on the blackboard

To get all mobs from the blackboard, you can use:

```
select * from mobs
```

Getting all mobs of a particular type

To get all mobs of a particular type from the blackboard (including subtypes) you could use:

```
select * from mobs where typeof(mob, 'sharedPadObject')
```

This would retrieve all sharedPadObjects (sharedPadLines, sharedPadRectangles, etc) from the blackboard.

Selecting the order of objects

To get Mobs in a particular order, you can use the "order by" command, for example

```
select * from mobs where typeOf(mob, 'sharedPadObject') order by getSlot1(mob, 'creationTime')
```

"asc" and "desc" could be appended to get things in ascending or descending order, respectively.

Getting the first of a list of mobs

If we wanted the most recent sharedPadObject, this could be achieved as follows:

```
select top 1 * from mobs where typeof(mob, 'sharedPadObject') order by getslot1(mob, 'creationTime') desc
```

Finding Mobs with certain properties

If we wanted to find a mob with particular properties, this could be tested, for example, as follows:

```
select * from mobs where contains(mob, 'creator', 'myAgentName')
```

The same thing could be accomplished using

```
select * from mobs where getslot1(mob, 'creator') = 'myAgentName'
```

(assuming, of course, that the 'creator' slot is single-valued).

Chapter 5. A simple client: `SharedPad`

`SharedPad` is an agent that illustrates the basics of writing a MICA agent. This chapter includes a walk-through of the MICA-related code that is in `SharedPad`.

SharedPad allows any number of connected agents to share a single notepad and to draw on it using a mouse and/or stylus. Anything drawn on one shared pad is shared amongst the other pads. In addition, state is preserved, so that even if every shared pad disconnects, all the information is kept.

SharedPad uses a single type of Mob: `sharedPadLine`. As previously discussed, it has four fields that represent the start and end points of a line:

As mentioned previously, there are two interesting methods that need to be implemented for an agent: `init()` and `handleMob()`. `init()` is called -- typically by `main()` to start the agent.

```
public void init(MicaProperties mp) throws MicaException {
  at.connect("sharedPad");
  List existingLines
    = at.mobSearch("select m from m in sharedPadLine;");
  for(Iterator i = existingLines.iterator(); i.hasNext();) {
    drawSharedLine((Mob) i.next());
  }
  at.register("sharedPadLine");
}
```

Figure 5.1. The `init()` method for `SharedPad`

The first thing that the agent does is connect to the blackboard. To do this, it uses `at`, which is a field that stores the agent transport. By calling `at.connect("sharedPad")`, the agent connects to the blackboard and requests that it should be called sharedPad.

Once connected, we query for all existing lines on the blackboard. This is useful, since if someone has already started drawing, it allows the retrieval of existing doodles. The query uses MicaQL, a simple query language. The above statement says: get all objects m from the blackboard, whose type is sharedPadLine. For each mob, we then use the method `drawSharedLine()` to draw it. `drawSharedLine()` is outlined below.

Finally, once we have obtained all the existing data, we now register for any new `sharedPadLine` mobs that are written to the blackboard, using the `at.register()` call.

The above method called `drawSharedLine()`, which is used to actually draw it on to the screen.

```
public void drawSharedLine(Mob m) {
  drawnLines.add(m);
  int startX = m.getSlot1AsInt("oldX"));
  int startY = m.getSlot1AsInt("oldY"));
  int endX =  m.getSlot1AsInt("newX"));
  int endY =  m.getSlot1AsInt("newY"));
  Graphics g = getGraphics();
  g.drawLine(startX, startY, endX, endY);
}
```

Figure 5.2. The `drawSharedLine()` method for `SharedPad`

`drawSharedLine()` is an example of how the information stored in a mob can be extracted and used. Firstly, the mob is added to a list of drawn lines -- this isn't for any reason related to the blackboard, but simply so that if our sharedPad window gets covered by other GUI applications we can do an redraw easily. The next four lines extract data from the mob; each converting it into an integer. Note that we use `getSlot1AsInt()` in this case, since we only want one value for each slot and we want it to be returned as an integer. Finally we obtain the graphics context and draw the line based on the information we extracted from the mob.

The other major method an agent must define is `handleNewMob()`.

```
public void handleNewMob(Mob m) {
  if(m.getType().equals("sharedPadLine")){
    drawSharedLine(m);
  }
}
```

Figure 5.3. The `handleMob()` method for `SharedPad`

The code first checks to see if the type of the received mob is a `sharedPadLine`. Strictly speaking this is unnecessary, since `sharedPadLines` are the only type that is registered for. Once a new `sharedPadLine` is received, it is drawn, just like the lines that were retrieved from the blackboard initially.

All the functionality now required to implement the drawing of lines already on the blackboard is now complete. This code is now sufficient (together with the supporting Swing and Java boilerplate code) for implementing a passive sharedPad that watches while other sharedPad agents connected to the blackboard draw. What remains is the functionality to write to the blackboard.

This is implemented by having sharedPad object have a `MouseDragListener`. Every time the mouse is dragged, it calls the following newLine() method:

```
public void newLine(int oldX, int oldY, int newX, int newY) {
  Mob m = new Mob("sharedPadLine");
  m.addSlot("oldX", String.valueOf(oldX));
  m.addSlot("oldY", String.valueOf(oldY));
  m.addSlot("newX", String.valueOf(newX));
  m.addSlot("newY", String.valueOf(newY));
  at.writeMob(m);
}
```

Figure 5.4. The `newLine()` method for `SharedPad`

The parameters passed to `newLine()` are used to construct the appropriate mob. For each of the four values, we add a slot, and set the value of the slot to a string value. Finally, once it is all finished; the newly constructed mob is written to the blackboard through the agent transport `at`.

Once all these components have been put together, with the remainder of the Swing code; we now have an Agent that can share information and retain state. As can be seen, the code is relatively simple.

One interesting note is that a given `SharedPad` does not have a concept of a "local" and "remote" line; so much so that when a user drags the mouse, a line is not drawn locally; instead the agent just listens for generic `sharedPadLine` mobs, some of which may have been its own! It would be possible to immediately draw a line as soon as the mouse is dragged and then ignore mobs which was created by the agent itself; indeed, this would lead to the user experiencing a "snappier" response. This was not done in this case for two reasons: (a) it would complicate the code, (b) it is useful to experience the "round-trip time" of going to and from the blackboard to decide if MICA is suitable for "real-time" interaction.

Chapter 6. A collection of clients: the mail reading application

While `SharedPad` was a single client, many applications using MICA have several clients involved in the process. In this chapter we discuss the mail reading application, which involves four different MICA clients: a learning agent, an e-mail agent, a user interaction agent, a natural language processing agent and a debug agent.

A scenario

Imagine that in 2006, a user (say Aki) has the WhizBang 3000 PDA. The user carries it around with him everywhere. The PDA has some pretty cool technology on board. In particular, it has GPS, speech recognition and speech synthesis. It also interacts with sensors in the environment that let it know things like: the ambient noise level, whether the user is at home, in the car, or in the office, and what other people are around.

One funny thing about the WhizBang is that it does not have a screen, or a microphone, or even a speaker. Instead, it is just a small box that Aki carries in his briefcase. It knows how to communicate with its environment, so will use the car's speaker and microphone if it has one, and the desktop computer's screen if Aki is in the office or even the Bluetooth headset that Aki sometimes wears.

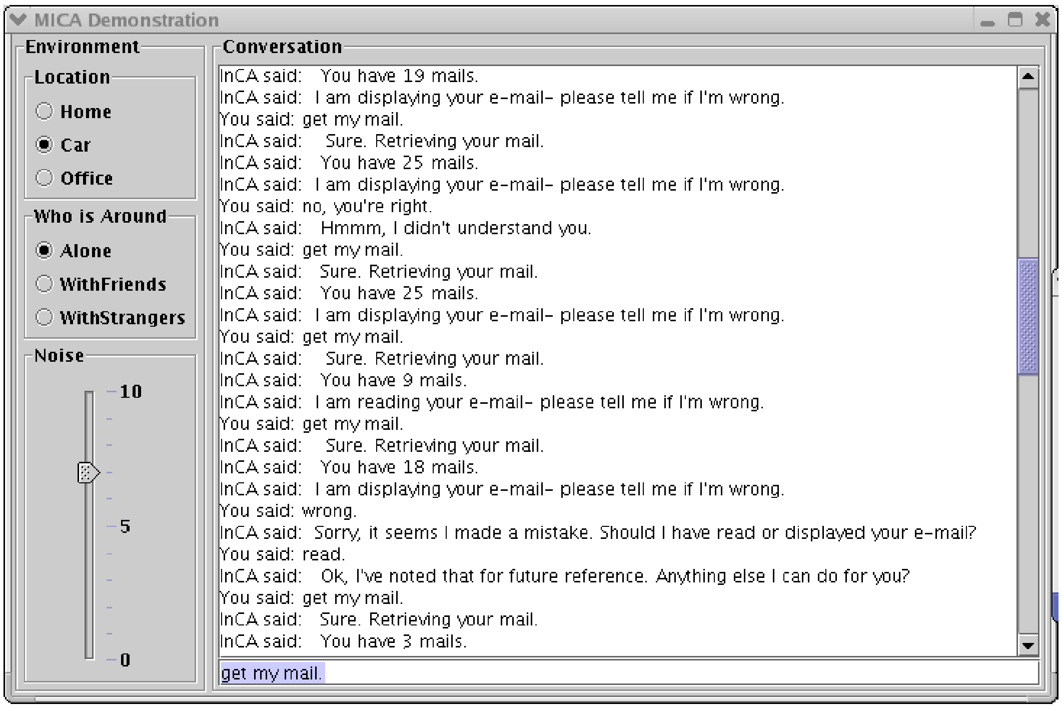
Now, since the PDA is so smart: the device can be used in the car, at home or in the office. However, one issue is to learn what modes the user prefers to use, depending on these factors. For example, if Aki receives e-mail, should it be read aloud to him using the car's speakers, or displayed using the in-dash LCD?

Ideally, we'd like the WhizBang to learn what Aki likes to do. It will learn, for example, that if Aki is in the car and it's not noisy, then it should read the e-mail out loud. But, if Aki's in the office, it should display them on his monitor.

In this chapter, we mock up some of the infrastructure, and show how such an application could be implemented.

Cast of characters

What agents would be involved in such an application? Well, first, there's the agent that represents the WhizBang itself. For this demonstration, we'll mock up the WhizBang as shown in figure Figure 6.1.



Location: Home [✓ Car] Office

Who is Around: [✓ Alone] WithFriends WithStrangers

```
InCA said:   You have 19 mails.
InCA said:  I am displaying your e-mail- please tell me if I'm wrong.
You said: get my mail.
InCA said:    Sure. Retrieving your mail.
InCA said:   You have 25 mails.
InCA said:  I am displaying your e-mail- please tell me if I'm wrong.
You said: no, you're right.
InCA said:   Hmmm, I didn't understand you.
You said: get my mail.
InCA said:   Sure. Retrieving your mail.
InCA said:   You have 25 mails.
InCA said:  I am displaying your e-mail- please tell me if I'm wrong.
You said: get my mail.
InCA said:    Sure. Retrieving your mail.
InCA said:   You have 9 mails.
InCA said:  I am reading your e-mail- please tell me if I'm wrong.
You said: get my mail.
InCA said:    Sure. Retrieving your mail.
InCA said:   You have 18 mails.
InCA said:  I am displaying your e-mail- please tell me if I'm wrong.
You said: wrong.
InCA said:  Sorry, it seems I made a mistake. Should I have read or displayed your e-mail?
You said: read.
InCA said:   Ok, I've noted that for future reference. Anything else I can do for you?
You said: get my mail.
InCA said:   Sure. Retrieving your mail.
InCA said:   You have 3 mails.
```

```
get my mail.
```

Figure 6.1. Screenshot of the Whizbang interface

To "mock up" the ability to detect information like who is around, we have a series of panels on the left hand side for detecting the location, people who are around and the general background noise level. The rest of the panel is for a speech conversation between the user and WhizBang. For this demo, we'll use text.

We also make use of a bogus email retrieval agent. When queried, this agent returns a list of e-mails. Currently, the number of e-mails is generated randomly, and the user may receive anywhere between 0 and 30 e-mails.

There is also a natural language processing agent. This agent "wraps around" the a C program that supports an NLP system. This agent called the MicaBot Agent, is responsible for parsing what the user says.

Finally, there is a learning agent that we use to learn when to read e-mail and when to display it.

There are some additional agents available: one is the debugger program that allows the user to see everything on the blackboard, and the other is one for looking at the decision trees generated in the process of learning.

Setting up Mobs

In order to set up the system, decisions must first be made about what types of mobs agents will use to communicate. First of all, to communicate text to and from the user, the type declaration in Figure 6.2 is used.

```
<typedesc>
  <mobdecl name="text">
    <slot name="utterance"/>
  </mobdecl>
  <mobdecl name="textFromUser">
    <parent name="text"/>
  </mobdecl>
  <mobdecl name="textForUser">
    <slot name="speaker"/>
    <parent name="text"/>
  </mobdecl>
</typedesc>
```

Figure 6.2. The type declaration for types of text used between agents.

In Figure 6.2, three mob types are declared: a generic text mob, with a single slot called "utterance" to store what is being said[1]. There are two subtypes: `textFromUser` and `textForUser`. `textForUser` has an additional slot to describe who it is who is talking.

These mobs are used to communicate between the WhizBang interface and the Natural Language Processing (NLP) agent.

The WhizBang interface also uses several other mobs to describe the environment; such as the `envNoise` mob to describe noise levels. Every time the noise level changes, it will write a new mob to the blackboard. Similarly, for the location of the user, and the people around them.

The e-mail agent only listens for one mob: `emailListRequest`. It then responds with an emailListReply, which contains three slots: `count` -- the number of new e-mails, `from` and `subject`. The latter two are multi-valued. In this particular case, the information is randomly generated.

The learning agent is rather complex. First of all, the learning task is defined. This is done by way of a learning task configuration. Similar to the way that new types are declared, learning tasks are defined in `con-fig/learntask`. Each file in that directory is read. The configuration file for the learning agent is shown in Figure 6.3.

[1]The current implementation of MICA will read the `slot` element of the xml document, but won't actually do anything about it. It's there as a form of documentation,

```
<learntask
  name="readOrDisplayEmail"
  learner="weka.classifiers.trees.j48.J48"
  datafile="/tmp/readordisplay.arff"
  modelfile="/tmp/readordisplay.mdl">

  <attribute name="location" type="discrete"
        sourcemob="envLocation" sourceslot="location" >
    <value label="home"/>
    <value label="office"/>
    <value label="car"/>
  </attribute>

  <attribute name="noiseLevel" type="continuous"
        sourcemob="envNoise" sourceslot="noiseLevel"/>

  <attribute name="whosaround" type="discrete"
        sourcemob="envWhosAround" sourceslot="whosAround">
    <value label="alone"/>
    <value label="withfriends"/>
    <value label="withstrangers"/>
  </attribute>

  <attribute name="numMails" type="continuous"
        sourcemob="emailListReply" sourceslot="count"/>

  <class name="readOrDisplayEmail">
    <value label="askuser"/>
    <value label="readmail"/>
    <value label="display"/>
  </class>
</learntask>
```

Figure 6.3. `readordisplay.xml`

For a given classification task (in this case "readOrDisplayEmail"), we define a learning algorithm for the task as well as files for temporary results to be stored. A learning task consists of a set of attributes, and a final class the learner is trying to classify. In this case, the possible actions are "readmail" or "display". The attributes are things like the noise level, who is around and how many e-mails were received. For each of these attributes, the learning task extracts the value from information on the blackboard. It does this by listening to any new mobs of the types important for this classification task; and thus has an idea of the "current" context.

Although this part is very complex, once a learning task is defined, other agents can use it easily. In order to provide an example to the learner; a `learnerTrain` is written to the blackboard, specifying the learning task and the actual class, given the current context. The learner then extracts the appropriate information from the blackboard and stores the current situation as an example.

In order to use the learner to decide what it should do given the current context, it writes a `learnerTest` mob, with the following fields: a `requestId`, so that when the learner agent replies, the other agent will know what the learner agent is replying to, and secondly the learning task. The learning agent then replies with a `learnerReply` mob with the copied `requestId`, the predicted class and the confidence of the prediction.

Figure 6.4 shows a sequence diagram for the interactions that occur in this process.

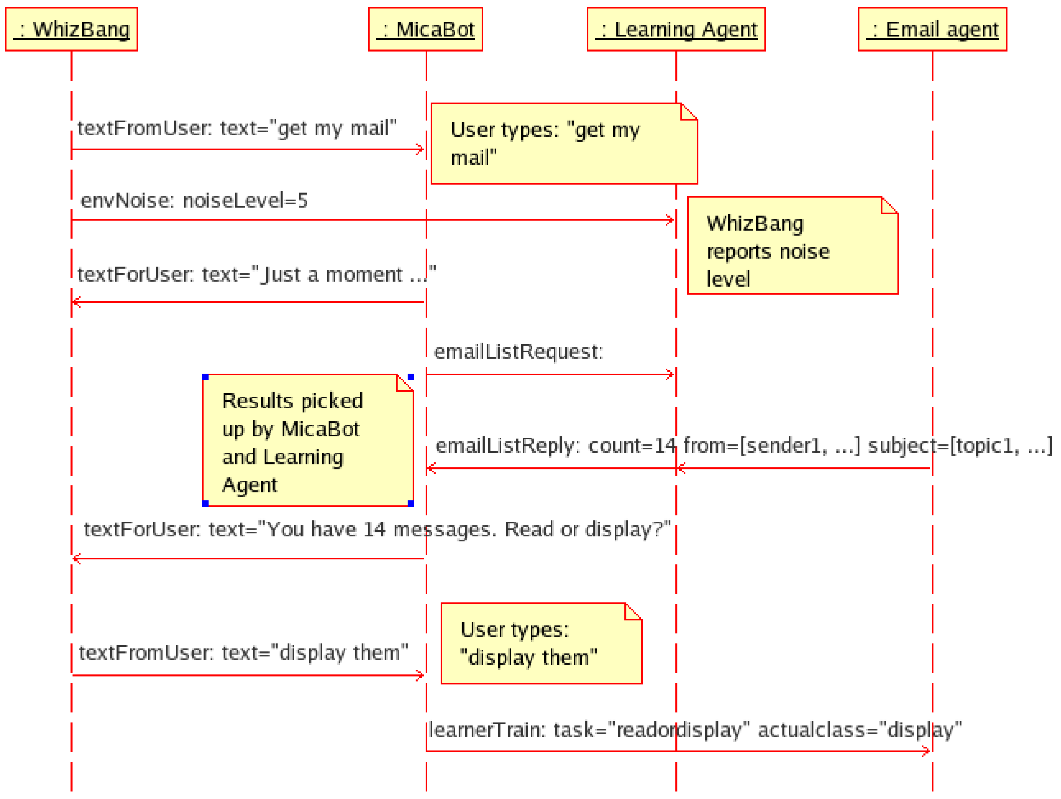
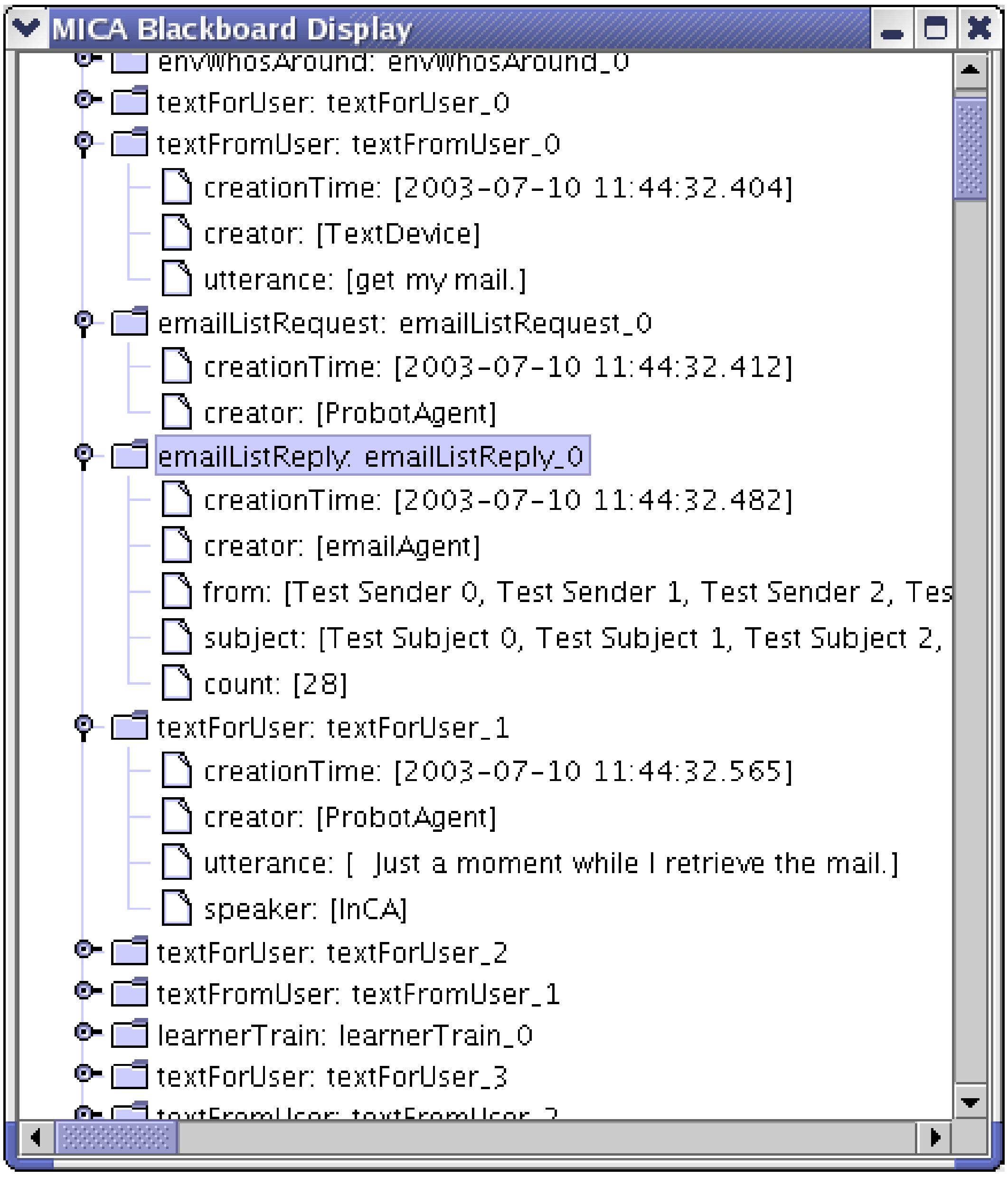

: WhizBang
: MicaBot
: Learning Agent
: Email agent
textFromUser: text="get my mail"
User types: "get my mail"
envNoise: noiseLevel=5
WhizBang reports noise level
textForUser: text="Just a moment ..."
emailListRequest:
Results picked up by MicaBot and Learning Agent
emailListReply: count=14 from=[sender1, ...] subject=[topic1, ...]
textForUser: text="You have 14 messages. Read or display?"
User types: "display them"
textFromUser: text="display them"
learnerTrain: task="readordisplay" actualclass="display"

Figure 6.4. Sequence diagram for training the WhizBang

In Figure 6.4, the blackboard is not explicitly shown; rather the information flows through the blackboard are shown. "Notes" are tools to aid in understanding. So, the user types in "get my mail". The interface writes this as a mob to the blackboard. Since the MicaBot agent (the natural language agent) has registered for "textFromUser" mobs, it is informed. Similarly, the learning agent is also informed when new information about environment is added to the blackboard. The process continues, with the MicaBot agent making a query of the email agent; and generating a text description for the user.

The MicaBot agent in this case is designed to begin by eliciting user preferences, and eventually to use the elicited preferences to predict what the user would like. In this case, it tells the user, then responds to the user's reply by providing an example to the learning system using a `learnerTrain` mob.

In order to run the demonstration of all of these agents running, firstly kill any blackboards or other agents running (this is not strictly necessary, but it helps to make sure the script will run). Then in the Mica directory, type **java unsw.cse.mica.runner.MicaRunner examples/run/learnemail-run.xml**. When the "start all" buttons is hit, in addition to the agents mentioned above, this will open two other windows: firstly, a "MICA debugger". This application is generally useful, and allows you to see all the information on the blackboard. An example of the debugger is shown in Figure 6.5



MICA Blackboard Display

```
envWhosAround: envWhosAround_0
textForUser: textForUser_0
textFromUser: textFromUser_0
    creationTime: [2003-07-10 11:44:32.404]
    creator: [TextDevice]
    utterance: [get my mail.]
emailListRequest: emailListRequest_0
    creationTime: [2003-07-10 11:44:32.412]
    creator: [ProbotAgent]
emailListReply: emailListReply_0
    creationTime: [2003-07-10 11:44:32.482]
    creator: [emailAgent]
    from: [Test Sender 0, Test Sender 1, Test Sender 2, Tes
    subject: [Test Subject 0, Test Subject 1, Test Subject 2,
    count: [28]
textForUser: textForUser_1
    creationTime: [2003-07-10 11:44:32.565]
    creator: [ProbotAgent]
    utterance: [  Just a moment while I retrieve the mail.]
    speaker: [InCA]
textForUser: textForUser_2
textFromUser: textFromUser_1
learnerTrain: learnerTrain_0
textForUser: textForUser_3
textFromUser: textFromUser_2
```

Figure 6.5. The MICA debugger

The second window opened displays the decision tree learned in the process of answering the question of whether to read or display the e-mail. You should hit the "Reload" button occasionally to reload the tree. It is shown in Figure 6.6.

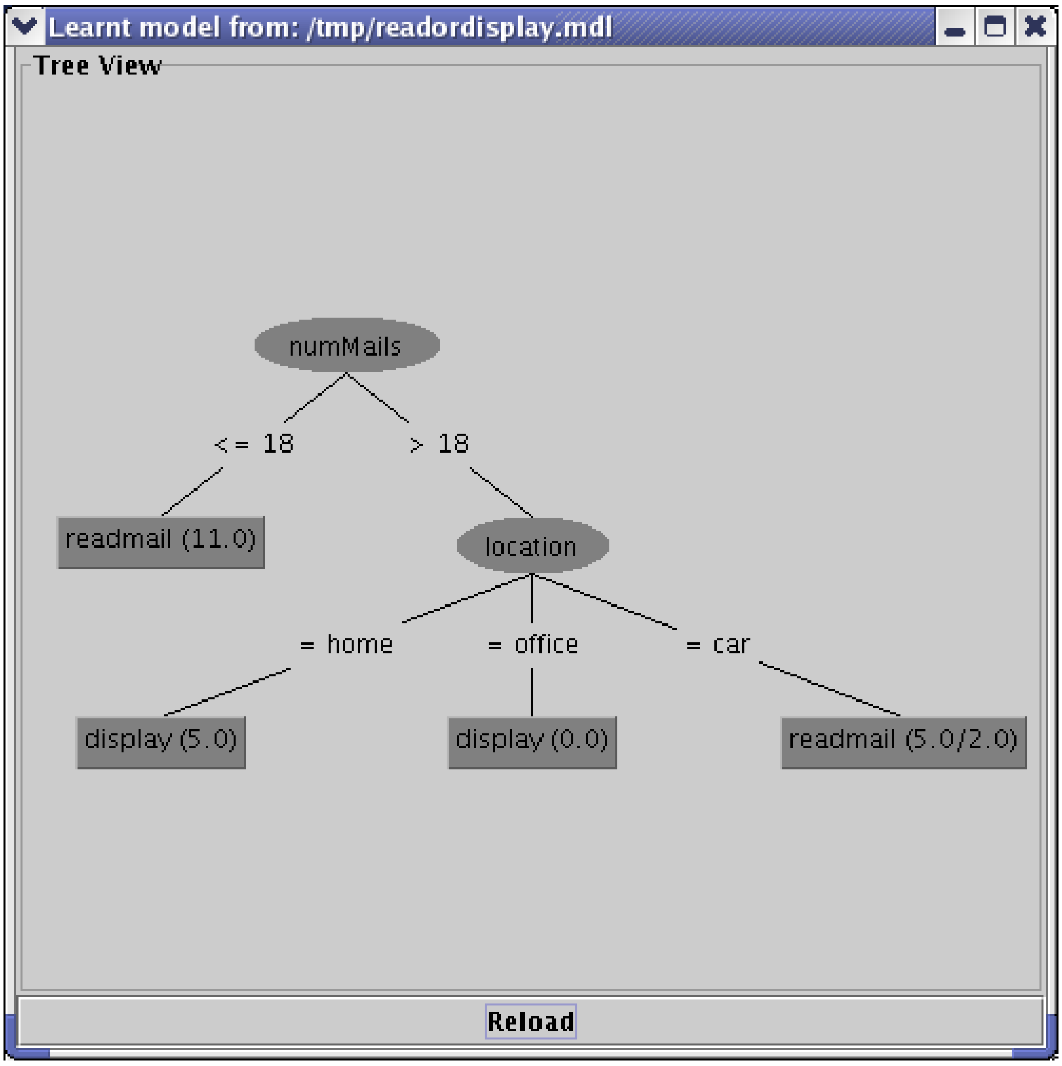


Figure 6.6. Tree learnt from conversation with user

The above tree shows the concept learnt after a few rounds with the user. It has learnt that it should read e-mails in the car, and if at home, read e-mails if there is less than 18 and display them if there is more than 18.

For a closer examination, users should consider reading the source code.

Chapter 7. Writing your first clients

In this chapter, we write from scratch a pair of clients to implement a simple "Knock knock" joke system.

Chapter 8. Writing your own transport medium

Not yet written.

Chapter 9. Writing your own security policy

Not yet written.

Chapter 10. Writing your own blackboard

Not yet written