using Boris in Java

brief

Boris is a multiagent platform with support for agents written in Java and other languages. Boris is designed to be generic in style rather than imposing a specific type of agency (like BDI for example) so can be used as a middleware solution as well as a framework for building multiagent systems.

This document gives a quick guide to the basics of using Boris to write Java agents.

agents & portals

Boris uses 2 types of entity: agents & portals. Agents are what you (the programmer) write to carry out the behaviour you want for your programs. Portals are provided by Boris to manage the message passing between agents.

It is possible to write agents which do not communicate but normally we want them send each other messages. To allow this we need to link our agents to a portal and write a message handler for them.

receiving messages

Boris treats agent-messages as if they were agent-events and uses an analogy (in Java) with other event handling mechanisms, like a awt Button and an ActionListener for example. This means that the code needed to build a button, place it on a panel & write an ActionListener is similar to that required to build an agent, add it to a Portal & write a MessageListener. See the examples below...

```
// building a button & a panel
Panel p = new Panel();               // create a Panel
Button b = new Button( text );       // create a Button

// specify an ActionListener for the Button
b.addActionListener( new ActionListener()
{   public void actionPerformed( ActionEvent event )
    {   ...code body...
    }
});
p.add( b );                          // add the Button to the Panel
```

```
// building an agent & a portal
Portal p = new Portal( portal-name );     // create a Portal
Agent a = new Agent( agent-name );        // create a new Agent

// specify a MessageListener for the Agent
a.addMessageListener(new MessageListener()
{  public void messageReceived(String frm, String to, String msg, MsgId id)
   {   ...code body...
   }
});
p.addAgent( a );                          // add the Agent to the Portal
```

The example (above) shows how to specify an agent which responds to incoming messages. The handler method for MessageListener messages is

```
void messageReceived(String from, String to, String msg, MsgId id)
```

The arguments for *messageReceived* are as follows...

- from the name of the agent who sent the message;
- to the name of the agent the message was addressed to (usually this is the same as the agent which receives the message but it is possible for agents to share message listeners so this information can be useful);
- msg the message itself;
- id the message id – a multi-faceted structure which can help keep track of dialog sessions.

sending messages

Agents send messages simply by issuing a call to...

```
void sendMessage( String to, String msg )
```

with arguments as follows...

- to the name of the agent the message should be sent to;
- msg the message.

See following example...

```
Portal p = new Portal( portal-name );
Agent sue = new Agent( "sue" );
sue.addMessageListener(new MessageListener()
{  public void messageReceived(String frm, String to, String msg, MsgId id)
   {   ...code body...
   }
});
p.addAgent( sue );
Agent sam = new Agent( "sam" );
p.addAgent( sam );

...
sam.sendMessage( "sue", "hello sue" );
...
```

a working example

This is based on the chat-room example which can be found on www.agent-domain.org (Chat3Server & Chat3Client).

In this example the chat-room server receives messages from its clients and broadcasts them to all other clients (the number of clients is not limited).

As you will see from the example, the server uses two agents – one (the management agent) deals with client agents joining & leaving the service, the other receives & broadcasts messages from registered clients. If the chat-room is called "room1" then the server's management agent is called "room1manager" and its message broadcasting agent is called "room1".

coding the client

Each client uses one agent. When joining & leaving a room it contacts the room-server's management agent. When posting messages to the room, the client contacts the server's message broadcasting agent.

step 1 – the client agent & its message receiver

The clients receive messages broadcast by the server and display them in a TextArea called *msgArea*.

```
//--- client agent ------------------
final Agent agent = new Agent( name );
agent.addMessageListener(new MessageListener()
{  public void messageReceived(String frm, String to, String msg, MsgId id)
   {  msgArea.append( "\n" + msg );
   }
});
portal.addAgent( agent, Portal.GLOBAL );
```

Note the use of: `portal.addAgent( agent, Portal.GLOBAL );`
Agents can have limited visibility/scope in Boris. Portal.GLOBAL ensures that agents are visible even when distributed across different JVMs and physical network nodes.

step 2 – sending messages to the server

Clients hold the names of the two server agents in string variables called "room" & "roomManager". Clients have a minimal GUI with Buttons to initiate clients joining & leaving a chat room and sending messages to the chat room.

The processes of joining & leaving are similar (only join is shown below) and involve contacting the server's management agent. The joining/leaving messages are very simple "join" or "leave". To post a message in the chat-room (& so have the message broadcast to other clients) the message is sent to the server's broadcasting agent. The clients use a TextField called "chatMsg" which allow users to enter text.

```
Button joinBtn = new Button("join");
joinBtn.addActionListener(new ActionListener()
{  public void actionPerformed(ActionEvent e)    // contact the room manager
   {  agent.sendMessage( roomManager, "join" );   // & join the chatroom
   }
});
add( joinBtn );

Button sendBtn = new Button("send");
sendBtn.addActionListener(new ActionListener()
{  public void actionPerformed(ActionEvent e)          // contact the room &
   {  agent.sendMessage( room, chatMsg.getText() );     // send it some text
   }
});
add( sendBtn );
```

coding the server

As outlined above the server creates two agents. The manager deals with "join" & "leave" messages, the other handles broadcasting.

These are specified as shown below. Servers maintain a collection of chat-room members (with methods addMember & removeMember) and the String variable "name" holds the name of the server)...

```
//--- manager agent ------------------
manager = new Agent( name+"manager" );
manager.addMessageListener(new MessageListener()
{  public void messageReceived(String frm, String to, String msg, MsgId id)
   {  if (msg.equals("join"))
         addMember( frm );
      else if (msg.equals("leave"))
         removeMember( frm );
      manager.sendMessage( frm, msg+" status=ok" );
   }
});
portal.addAgent(manager, Portal.GLOBAL);

//--- chat agent ---------------------
chatAgent = new Agent( name );
chatAgent.addMessageListener(new MessageListener()
{  public void messageReceived(String frm, String to, String msg, MsgId id)
   {  if( members.contains( frm.intern() ))
      {  String msgTxt = frm + " \t" + msg;
         msgArea.append( "\n" + msgTxt );
         broadcast( msgTxt );
      }
   }
});
portal.addAgent( chatAgent, Portal.GLOBAL );
```

constructor methods & the IDE

You can use Boris IDE (also called the Boris console) to load your compiled agent code (clients & servers). For this to be possible there must be a constructor method with one of the following signatures...

```
    public SomeClass( Portal portal )
    public SomeClass( Portal portal, String cmd )
```

The IDE passes the constructor a prebuilt Portal and (if specified) a command string which you can specify when you load a class with the IDE.

The chat client class uses the following constructors...

```
public Chat3Client( Portal portal )
{  this( portal, "" );
}

public Chat3Client( Portal portal, String cmd )
{  ...construction code...
}
```

distributing your agents across a network

There are two common approaches to distributing your code across a network, briefly these are...

1. Boris consoles also operate as message routers. You can open a Boris console/router on each network node and connect them (part of the main panel of the console provides the ability to connect one console/router to another);
2. Portals support methods which allow them to connect to consoles (and other types of router) by specifying their IP address (see Portal.connectToGrid and the relevant Javadoc for boris.jar).

Writing distributed Boris agent-ware is described in more detail elsewhere.

the code structure: agents & objects

Some multi-agent platforms encapsulate everything as agents, others introduce new types of OO patterns to define agent behaviours and messaging. Boris intentionally uses established OO collaboration patterns and takes an approach which will be familiar to many Java programmers.

This approach imposes few constraints and allows agents to be freely mixed with objects. There are some significant advantages to this but it also allows programmers to produce poorly structured code. To avoid this we recommend sketching out an agent-level structure for your systems before you start to write code or plan which design patterns to use.

where to go next

This document intends only to provide a brief introduction. There are many features of Boris which have not been described. Please look at other documentation provided at www.agent-domain.org or email borismolecule@gmail.com for support.