1. messaging

1.1 sending messages

1.1.1 send-message

The most basic means of communication between agents uses a simple send-message call. In Java this is done using...

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

To send the message "*have a nice day*" to an agent named "*sid*" the call would be...

```
sendMessage( "sid", "have a nice day" )
```

Note: the API provides various sendMessage forms, these are described in other parts of this document. All sendMessage forms return a value, this is also described elsewhere.

1.1.2 message-received

Agents receive messages through a message handler method. In Java this is called messageReceived and is specified in the MessageListener interface. To write an in-line definition of an agent, providing it with a message listener you need to use something similar to the code below.

```
Agent sid = new Agent( "sid" );
sid.addMessageListener( new MessageListener()
{       public void messageReceived( String from, String to, String msg, MsgId msgId )
        {       // message handling code
        }
});
```

Notice the analogy between the use of a MessageListener and listener models from the Java AWT (the ActionListener for example).

The messageReceived method takes 4 arguments, these are...

from	the name of the agent which sent the message;
to	the name of the agent that the message was addressed to. This is normally only useful if the same Message-Listener is shared between agents;
msg	the text of the message;
msgId	a unique (and faceted) identifier for the message, MsgIds are described later in this document.

1.1.3 sending messages / clones & non-clones

Agents can be *cloneable* or *non-cloneable*. *Cloneable* agents process their messages as soon as they receive them. If a cloneable agent is busy with an existing task a new clone is created to process the new message. *Non-cloneable* agents have their messages queued when they are busy.

In practice Boris handles clones by creating a new process thread each time their MessageListeners are triggered. Clones share global variables (including instance variables for agents which extend the Agent class) but local method variables are not shared.

1.2 replying to messages

1.2.1 sessions

Messages are often sent in a series of exchanges which together form some kind of dialog. These exchanges are known as sessions. In Boris sessions are initiated with send-message but then continued by sending replies to earlier messages (or earlier replies).

1.2.2 send-reply

Replies are sent (in Java) using...

```
sendReply( MsgId mfor, String reply )
```

The first argument for sendReply is the MsgId of the message that is being replied to. This is most often a message that has been recently received.

The code below defines a simple agent class which send a reply "*thanks for the message*" for any message it receives...

```
class MyAgent extends Agent implements MessageListener
{       public MyAgent( String name )
        {       super( name );
                addMessageListener( this );
        }
        public void messageReceived( String from, String to, String msg, MsgId msgId )
        {       sendReply( msgId, "thanks for the message" );
        }
}
```

1.2.3 reply-receivers

While messages are handled by a MessageListener, replies are handled by a ReplyListener. This makes it easier for agents to track multiple sessions. ReplyListeners use a handler method call replyReceived with the following signature...

replyReceived(String from, MsgId mfor, String reply, MsgId id)

The replyReceived method takes 4 arguments, these are...

from	the name of the agent which sent the message;
mfor	the MsgId of the message that this is a reply to;
reply	the text of the reply;
msgId	a unique (and faceted) identifier for the reply, MsgIds are described later in this document.

Replies are only sent to those agents which request them. An agent requests a reply by including a ReplyListener as a third argument in a call to sendMessage or sendReply. The ReplyListener and its associated replyReceived method is automatically invoked when a reply is received, eg:

```
ReplyListener rl = new ReplyListener()
{       public void replyReceived( String from, MsgId mfor, String reply, MsgId id )
        {       // place code to handle the reply here
        }
};
sendMessage( "sid", "have a nice day", rl );
```

In the above example the replyReceived method will be called when "*Sid*" sends a reply to the "*have a nice day*" message.

1.2.4 message id.s

Message id.s are faceted (ie: they have different parts), most of these are only ever used by the Boris kernel but one facet is the session id. The session id is a unique string identifier which can be used to identify any given session. All messages which are part of the same session will have the same session id and every session will have a different id.

Programmers writing agents to handle sessions in some specialised way may need to access session id.s, they can be retrieved with the MsgId method getSessionId...

String MsgIg.getSessionId()

1.2.5 clones & non-clones

Message replys for both cloneable and non-cloneable agents are routed through ReplyListeners but the scheduling of this activity differs.

With cloneable agents, ReplyListeners run concurrently with other ReplyListeners and also with any other activity of any given agent. With non-cloneable agents, ReplyListener activity occurs non-concurrently, in sequence with other ReplyListeners and also with any other activity of any given agent. This sequential activity is scheduled on a first-come first-served basis.

1.2.6 waiting

In some cases it is convent for an agent to send a message (or a reply) to another agent then delay further activity until a reply is received. This *waiting model* is supported in Boris but there is a risk of deadlock and/or hanging associated with this approach.

Deadlock exists when two or more activities are waiting for each other to finish and processing hangs as a result. There is a risk of this when using a waiting model -- particularly when non-cloneable agents are used.

With a distributed MAS, agent activity can also stall if some kind of hang-up (perhaps a part of the network going down) is propagated across the system. This is more likely to occur when using a waiting model (as above this is more likely when non-cloneable agents are used).

Despite the risks there are some tasks which are more easily programmed using a waiting model and Boris provides forms for *sendAndWait* and *replyAndWait*.

1.3 waiting

1.3.1 send and wait

Call sendMessage and suspend processing on the current agent until a reply is received (see below for details of the WaitReply class)...

public WaitReply sendAndWait(String to, String msg)

1.3.2 reply and wait

Call sendReply and suspend processing on the current agent until a reply is received (see below for details of the WaitReply class)...

public WaitReply replyAndWait(MsgId mfor, String reply)

1.3.3 the WaitReply class

sendMessage and sendReply methods return an instance of a WaitReply object which contains details about the reply, itsMsgId, the sender, etc. Accessor methods for WaitReply are as follows...

public String getFrom()
    get the name of the agent who sent the reply;

public String getReply()
    get the reply;

public MsgId getId()
    get the message id for the reply;

public MsgId getMfor()
    get the message id this reply was sent for.

1.3.4 clones & non-clones

Clones and non-clones work with waiting models in a way consistent with their design philosophy. If a cloneable agent receives a message while it is in a waiting state then a new clone is produced to handle the new message. The new message will be processed concurrently with any waiting operation so the new message is not delayed and there is no deadlock.

With non-cloneable agents the situation is different. When a non-cloneable agent is in a waiting state it will not continue to process new messages. New messages are delayed until the wait has completed (along with any other pending activity). This means that deadlock can occur with non-cloneable agents.