Aglet Communication

- Agent Communication
- Speech Act Theory
- Agent Communication Languages
  - Knowledge Query Manipulation Language (KQML)
  - FIPA Agent Communication Language (ACL)
- Messaging - very important for Agent Interaction, Coordination, Collaboration etc.
- Remember - Difference between Objects & Agents?
  - Object - invoke method
  - Agent - request a service

Simple Messaging

- Principal means of Aglet Communication
- Message Passing
- InterAglet Messaging is based on a simple event model
- Requires an aglet to implement “handlers” only for the messages that it is supposed to understand
- The main message handling method is:
  ```java
  public boolean Aglet.handleMessage(Message myMessage)
  ```
- To send a message:
  - Invoke the `sendMessage()` method on the Aglet’s proxy - not directly on the Aglet
  ```java
  public Object AgletProxy.sendMessage(Message myMessage)
  ```
- Aglet Proxy acts as a message gateway
Example – Send Message

- You need the Proxy of the Aglet you wish to send a message to.
- The following example creates an Aglet called SimpleMessageChild and uses the proxy that it gets while creating it to send messages to it.
- Trivial example
- Typical applications will require you to communicate with Aglets that you don’t really know.

```java
public class SimpleMessageExample extends Aglet {
public void run() {
  AgletProxy proxy = getAgletContext().createAglet(getCodeBase(),
  "SimpleMessageChild", null);
  try {
    proxy.sendMessage(new Message("Hello"));
  } catch (NotHandledException e) {
    System.out.println("Message Not Handled!");
  }
}
```

Example – Handle Message

```java
public class SimpleMessageChild extends Aglet {
  public boolean handleMessage(Message msg) {
    if (msg.sameKind("Hello")) {
      System.out.println("I got a Hello Message...");
      return true;
    } else {
      return false;
    }
  }
}
```

Message Class

- Messages are objects.
- A Message Object is characterised by its "kind".
- This string property is used to distinguish messages from each other.
- The Message class supports a range of constructors with different mandatory arguments.
- Message objects can also contain an optional argument field for data associated with a particular message.
- The argument field can be either:
  - Atomic - String, int etc.
  - Tabular - Hashtable

Message Creation

- `public Message(String kind)`
  - Constructs a message of a specific kind.
  - Creates a default argument table that can hold a set of key-value pairs.
  - `setArg` method can be used to initialise the argument table.
- `public Message(String kind, Object arg)`
  - Constructs a message of the specified kind with an Object as its initial bonding.
Message Creation

- public Message (String kind, int arg)
  Constructs a message of the specified kind with an int as its initial argument
- public Message (String kind, float arg)
  Constructs a message of the specified kind with a float as its initial argument
- public Message (String kind, long arg)
  Constructs a message of the specified kind with a long as its initial argument
- public Message (String kind, double arg)
  Constructs a message of the specified kind with a double as its initial argument
- public Message (String kind, char arg)
  Constructs a message of the specified kind with a char as its initial argument
- public Message (String kind, boolean arg)
  Constructs a message of the specified kind with a boolean as its initial argument

Message Creation - Examples

- Message msg = new Message("Greeting", "Happy Birthday");
- Message msg = new Message("Price", 200)

Receiving Messages

- public String Message.getKind()
  Gets the kind of the message
- public boolean Message.sameKind(String)
  Compares the kind of the message with the String Argument
- public Object Message.getArg()
  Gets the argument of the message

Receiving Messages - Example

```java
public class SimpleMessageChild extends Aglet {
    public boolean handleMessage (Message msg) {
        if (msg.sameKind("Hello")) {
            System.out.println("I got a Hello Message...");
            return true;
        }
    }
}
```
Receiving Messages - Example

```java
else if (msg.sameKind("Greeting")) {
    String msgContents = new String((msg.getArg().toString()));
    System.out.println("I got the following Greeting" + msgContents);
    return true;
} else
    return false;
```

Non-atomic Messages

- Nonatomic arguments - have more than one argument to the receiver
- Such Arguments are most effectively handled as key-value pairs
- `setArg` and `getArg` methods are useful for organising multiple arguments into a table
- `public void Message.setArg(String key, Object value)`
  - Maps the specified key to the specified value in the argument table
- `public Object Message.getArg(String key)`
  - Gets the value to which the key is mapped in the argument table, null is there is no mapped value for the key

Non-atomic Messages - Example

```java
public class MessageHashExample extends Aglet {
    public void run() {
        try {
            AgletProxy proxy = getAgletContext().createAglet(getCodeBase(), "MessageHashChild", null);
            try {
                Message msg = new Message("Location");
                msg.setArg("Horizontal", 60);
                msg.getArg("Vertical", 70);
                proxy.sendMessage(msg);
            } catch (Exception e) {
                System.out.println("Message Not Sent!");
            }
        } catch (Exception e) {
            System.out.println("Child Not Created!");
        }
    }
}
```

Non-atomic Messages - Example

```java
public class MessageHashChild extends Aglet {
    public boolean handleMessage(Message msg) {
        if (msg.sameKind("Location")) {
            Integer h = new Integer(msg.getArg("Horizontal"));
            int int_h = h.intValue();
            Integer v = new Integer(msg.getArg("Vertical"), 70);
            System.out.println("Horizontal:" + h + "Vertical" + v);
            return true;
        }
        return false;
    }
}
```

Replying to Messages

- `public void Message.sendReply()` - Sends a reply without any specific value ("null")
- `public void Message.sendReply(Object reply)` - Sends an "Object" reply
- `public void Message.sendReply(int reply)` - Sends an numeric reply
- `public void Message.sendReply(long reply)` - Sends an numeric reply
- `public void Message.sendReply(double reply)` - Sends an numeric reply
- `public void Message.sendReply(float reply)` - Sends an numeric reply
- `public void Message.sendReply(char reply)` - Sends a char reply
- `public void Message.sendReply(boolean reply)` - Sends a boolean reply
Replying to Messages - Example

public class MessageReplyExample extends Aglet {
    public void run() {
        try {
            AgletProxy proxy =
                getAgletContext().createAglet(getCodeBase(),
                "MessageReplyChild", null);
            try {
                Message msg = new Message("What is your
                height?");
                proxy.sendMessage(msg);
            }
            catch (Exception e) {
                System.out.println("Message Not Sent!");
            }
        }
        catch (Exception e) {
            System.out.println("Child Not Created!");
        }
    }
}

Getting the Reply

- How the sender manages the reply
- Synchronous Messaging
  - Sender waits for the reply by not continuing the thread of Execution
- Asynchronous Messaging
  - Uses the FutureReply object and its API to support continuing its thread while waiting for a reply

Getting the Reply - Asynchronous Messaging

- Based on future object
- The future object is returned by the proxy’s sendFutureMessage() method
  public FutureReply AgletProxy.sendFutureReply(Message msg)
- Methods in the FutureReply class used for Asynchronous messaging
  - public boolean FutureReply.isAvailable()
  - public void FutureReply.waitForReply(long duration)
  - public void FutureReply.waitForReply()
  - public Object FutureReply.getReply()

Getting the Reply - Synchronous Example

public class MessageReplyExample extends Aglet {
    public void run() {
        try {
            AgletProxy proxy =
                getAgletContext().createAglet(getCodeBase(),
                "MessageReplyChild", null);
            try {
                Message msg = new Message("What is your
                height?");
                Object reply = proxy.sendMessage(msg);
                int height = (Integer) reply.intValue();
                System.out.println("Her Height is: "+ height);
            }
            catch (Exception e) {
                System.out.println("Message Not Sent!");
            }
        }
        catch (Exception e) {
            System.out.println("Child Not Created!");
        }
    }
}

Aglets can send exceptions as results:
- public void sendException(Exception e)

The sender of the message should be able to accept an exception as a reply
- Handle the Exception!

Getting the Reply – Asynchronous Example

- Based on future object
  public FutureReply isAvailable()
  call isAvailable() to check whether a reply is coming in
- public void waitForReply(long duration)
- public void waitForReply()
- public Object getReply()
Message Management

- Each Aglet has a “Message Manager”
- Handles incoming messages by placing them in a queue and passing the contents to the Aglet
- Ensures that the next message in the queue is not forwarded until the current message is handled
- Facilitates
  - Serial or Sequential Messages
  - Message Priorities
  - Parallel Message Handling
  - Synchronised Message Handling

Serial Message Handling

```
public class MsgManagerExample extends Aglet {
    public void run() {
        try {
            AgletProxy proxy = getAgletContext().createAglet(getCodeBase(),
                "MsgManagerChild", null);
            proxy.sendFutureMessage(new Message("one"));
            proxy.sendFutureMessage(new Message("three"));
            proxy.sendFutureMessage(new Message("two"));
        } catch (Exception e) {} 
    }
}
```

```java
public class MsgManagerChild extends Aglet {
    public boolean handleMessage(Message msg) {
        if (msg.sameKind("one")) {
            System.out.println("Handled Message =one");
            return true;
        }
        else {
            return false;
        }
    }
}
```

Message Priorities

- Override the serial order in which messages are sent by adding priorities
- Priority Values: 1 - 10
- Default Message Priority: 5
- Higher Value, Higher Priority
- Lower Value, Lower Priority
- public void MessageManager.setPriority(String kind, int priority)
  - Sets the priority of the message of the type specified in the String “kind”
Message Priorities

```java
public class MsgManagerChild extends Aglet {
    public void onCreation(Object obj) {
        getMessageManager().setPriority("three", 10);
    }
    public boolean handleMessage(Message msg) {
        if (msg.sameKind("one")) {
            System.out.println("Handled Message =one");
            return true;
        }
        if (msg.sameKind("two")) {
            System.out.println("Handled Message =two");
            return true;
        }
        if (msg.sameKind("three")) {
            System.out.println("Handled Message =three");
            return true;
        }
        else
            return false;
    }
}
```

Parallel Message Handling

- Parallel message handling can be enabled by calling the Message Manager's `exitMonitor()` method
- Allows the Aglet to continue handling the current message, while simultaneously allowing a new message-handling thread to be started
```java
public class MsgManagerParallelChild extends Aglet {
    public boolean handleMessage(Message msg) {
        if (msg.sameKind("one")) {
            System.out.println("Handled Message =one");
            return true;
        }
        if (msg.sameKind("two")) {
            System.out.println("Handled Message =two");
            getMessageManager().exitMonitor();
            System.out.println("Hello 1");
            System.out.println("Hello 2");
            return true;
        }
        if (msg.sameKind("three")) {
            System.out.println("Handled Message =three");
            notifyMessage();
            return true;
        }
        else
            return false;
    }
}
```

Synchronised Message Handling

- If the handling of a message cannot proceed (external reasons?), the thread can be suspended until further notice
- In other words, subsequent messages can be handled
  - Three methods are available:
    - `public void Aglet.waitMessage();`
      - Suspends the execution of this message handling thread. The thread waits until further notice
    - `public void Aglet.notifyMessage();`
      - Notifies a single waiting message in the MessageManager
    - `public void Aglet.notifyAllMessages();`
      - Notifies all waiting messages in the MessageManager
```java
public class MsgWaitingChild extends Aglet {
    public boolean handleMessage(Message msg) {
        if (msg.sameKind("one")) {
            System.out.println("Handled Message =one");
            return true;
        }
        if (msg.sameKind("two")) {
            System.out.println("Handled Message =two");
            waitMessage();
            System.out.println("Hello 1");
            System.out.println("Hello 2");
            return true;
        }
        if (msg.sameKind("three")) {
            System.out.println("Handled Message =three");
            notifyMessage();
            return true;
        }
        else
            return false;
    }
}
```

Remote Messaging

- Remote messaging involves the same messaging model and constructs as in local messaging
- Location transparency
- The remote Aglet’s proxy must be obtained
```java
```
Multicasting

- So far: Peer-to-Peer Messaging
  - Aglet must know the proxy or the identity of the receiver

- Problems???
  - Coordination of multiple Aglets, whose origins etc are unknown?
  - Large numbers of Aglets?
  How can an incoming Aglet announce its arrival to all the other Aglets in the context?

- The Context supports Multicasting
  - NOTE: Within a single context
  - Facilitates Collaboration and Interaction

- HOW?
  - Subscribe to one or more multicast messages
  - Implement handlers for those messages

---

Multicast Example

Public class MulticastExample extends Aglet {
    public void run() {
        try {
            try {
                Message msg = new Message("Hello Everybody");
                getAgletContext().multicastMessage(msg);
            } catch (Exception e) {
                // handle exception
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}

---

Receiving Multiple Replies

- ReplySet class
  - Used to hold multiple FutureReply objects

- The ReplySet class has two methods that allow receiving multiple replies
  - public boolean hasMoreFutureReplies()
    - Checks whether there are any more replies in the set
  - public FutureReply ReplySet.getNextFutureReply()
    - Gets the next FutureReply that is available

---

Public class MulticastExample extends Aglet {
    public void run() {
        try {
            Message msg = new Message("Hello Everybody");
            ReplySet replies = getAgletContext().multicastMessage(msg);
            while (replies.hasMoreFutureReplies()) {
                FutureReply future = replies.getNextFutureReply();
                System.out.println(future.getReply());
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
Receiving Multiple Examples

```java
public class MulticastSubscriber extends Aglet {
    public void onCreation(Object obj) {
        subscribeMessage("Hello Everybody");
    }
    public boolean handleMessage(Message msg) {
        if (msg.sameKind("Hello Everybody")) {
            System.out.println("Handled Message");
            msg.sendReply("Hi!");
            return true;
        } else {
            return false;
        }
    }
}
```

References