REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

OpenAIR 1.0 Specification

Written and Edited by

Kristinn R. Thórisson
CADIA, Reykjavik University, thorisson@ru.is

and

Thor List
Communicative Machines, list@cmlabs.com

Authors and Contributors

Thor List, Christopher C. Pennock,
John DiPirro and Kristinn R. Thórisson

RUTR-CS07005

Reykjavik University – School of Computer Science

Technical Report

1 Introduction

OpenAIR is a routing and communication protocol based on a publish-subscribe architecture. It is intended to be the "glue" that allows numerous A.I. researchers to share code more effectively – "AIR to share". It is a definition or a blueprint of the "post office and mail delivery system" for distributed, multi-module systems. OpenAIR provides a core foundation upon which subsequent markup languages and semantics can be based, for e.g. gesture recognition and generation, computer vision, hardware-software interfacing etc; for a recent example see CVML.

The OpenAIR specification was initiated by Communicative Machines and is now exclusively managed by MINDMAKERS ("MINDMAKERS.ORG" is the legal 'original author' of this work). It is historically related to efforts such as KQML and Open Agent Architecture (it is not affiliated with those efforts). OpenAIR is based around standard TCP/IP and XML and consists of a simple but solid message semantics and network protocols. The full OpenAIR specification can be found below, released under the very liberal Creative Commons Attribution License. We expect to have the Java reference implementation of OpenAIR on this page soon.

2 Terminology and Conventions

If a feature is REQUIRED or if it MUST be supported it means that the implementation is not complete without support for that feature, and cannot therefore claim OpenAIR Compliance. If a feature is NOT REQUIRED it means it MAY be omitted; if a feature MAY be implemented it means that it is NOT REQUIRED. If a feature is RECOMMENDED it means that it is optional - yet desirable - to include it in an implementation. SHOULD means the same as RECOMMENDED.

Notice that in the below, if a content-holding data element in a Message – a slot – is REQUIRED it means that support for the slot (parsing and semantics) MUST be included for the Basic OpenAIR implementation. (It does not mean that the slot or its tag must be included when a Message is transmitted.)

In implementations of OpenAIR the terms "Basic OpenAIR Compliance" means that all REQUIRED features are supported; the terms "Full OpenAIR Compliance" means that all REQUIRED, NON-REQUIRED and all RECOMMENDED features are supported; the terms "Extended OpenAIR Support" means Full OpenAIR plus some special extensions. Such extensions MUST be documented to be released on MINDMAKERS.ORG.

The terms "Register" and "Subscribe" are used interchangably.

The term "slot" is used here in a general object-oriented way to refer to major content-holding aspects of an OpenAIR Message; the term "tag" refers to the generic definition of an XML tag.

Regular words, such as "post" and "subscription", when written in all-lower-case, refer to the common concepts that these words refer to in the English language. When such words are written with a capital first letter (e.g. "Post", "Subscription") they refer to the particular technical concept they have been defined to mean in this spec.

In the examples below, demonstration content that is not part of the spec is gray, non-bold and italic.

3 System Model

The term Component refers to any processing element in an OpenAIR system. There are four main types of Components, CNS, AIRServer, Modules and Dispatchers. A CNS is a Central Naming Service

which holds the unique names of all Components. The AIRServer is the active component that holds the CNS and Dispatcher(s). OpenAIR has four naming requirements for Components. 1. The central naming service MUST be called "CNS". 2. The Server MUST be called "AIRServer". 3. At runtime there MUST exist one Dispatcher named "AIRCentral." Modules MUST have system-wide unique names.

A Dispatcher's role is to disseminate messages from one Module to one or more other Modules. A Dispatcher can be implemented as a blackboard, or any kind of infobus; the protocol makes no difference between these. However, a few requirements are made of Dispatchers: They MUST understand the Basic OpenAIR message semantics; they MUST support the Basic and Full AIR message syntax (but not semantics)*; they MUST implement a publish-subscribe mechanism, as described below; they MUST support the dot-delimited Message type format (see below); they MUST support queries using Type, message ID and two timestamps specifying an interval. They MAY implement a priority or quality-of-service scheme by making use of priority= parameter.

Modules are Components that are built to process Messages, primarily the content of Messages – Modules can be thought of as the "application-specific" part of an OpenAIR system.

There are two roles that the Modules and Dispatchers can take: Sender and Receiver. The Sender is the one who creates a message, fills in the slots and sends it off. The Receiver is the one who receives that message. Any Component in the system can have either or both roles at any one time. The term "Poster" refers to a Module sending a Message to a Dispatcher.

To receive Messages, Modules can (a) Subscribe (also called "Register") for Message Types and/or (b) Retrieve Messages (by querying for ID, timestamps or Type) from a Storage Location, which can be e.g. a database or a Dispatcher.

Dispatchers typically do not Subscribe, Retrieve or Post Messages to other Dispatchers, their primary job is to dispatch Messages to Modules. Upon receivinga Message of Type A, a Disptacher sends a copy of it to anyone who has Subscribed to Messages of Type A. Dispatchers also service requests for Retrieving Messages.

All components in AIR MUST have unique names, which are represented as ASCII strings which MUST contain no spaces.

*In other words, this is a way to ensure that the Dispatcher can ignore any advanced functionality so that Basic-level Dispatchers can work with Full OpenAIR adapters.

4 Message Format and Semantics

The message format is defined in XML. This section explains the Message slots and their semantics (meaning and use). We use upper case "Message" when we are referring to AIR messages; lower-case "message" refers to the generic use of the term. When transmitting Messages across the network, emtpy slots MAY be omitted. Notice that in the below, if a slot is REQUIRED it means that support for this slot MUST be included for the Basic OpenAIR implementation. (It does not mean that the slot must be included when a Message is transmitted.)

5 AIR Message Format: Major Slot Names and Semantics

Message

The outermost wrapper for an OpenAIR Message. This tag is REQUIRED. It MUST be the first and last tag in the transmission of the Message.

```
<message priority="" timetolive="">
    ...
</message>
```

The ... represents the rest of the XML tags for the Message.

SLOT NAME	REQUIRED	SHORT DESCRIPTION
Message	+	The outer wrapper for all Messages
Priority=	recom	A float number between 0 and 6 indicating the relative priority of the Module that posted the Message
TimeToLive=	no	The duration for which the contents of this Message are valid; set by the Poster
ID	+	Global unique identifier (GUID)
Type	+	A dot-delimited string which represents the name of the type of this Message
From	+	The unique name of the Poster - the system component (module) that authored and posted this Message
To	+	The main Receiver of this Message, which is always a dispatcher (e.g. blackboard or infobus) that handles the dissemination of this Message
Cc	no	Any unique module name which should receive this Message, even though it may not be subscribed to receive this particular Message's Type
PostedTime	+	The time at which this Message was released, or posted, by the module that authored and posted it
RecievedTime	+	The time at which a receiving party received this Message
Content	+	The content of the Message - the main body. This slot has a parameter called language, which specifies the particular language of the content, e.g. XML, Prolog, Lisp, KQML, etc.
InReplyTo	no	A single Reference to the Message to which this Message is a reply, grouped by the Reference tag
Reference	no	Used as a quick reference "library card" to an already-posted Message
Stored	no	Name of storage (blackboard or database) where this Message can be accessed or retrieved from (after having been been dispatched by the storage or other dispatcher)
History	no	A stack containing a list of Reference tags, tracing the history of prior Messages relevant to this Message
Group	no	A tag for grouping Reference tags
Comment	no	Free-form text
Origin	+	[Low-level slot] The IP address of the computer that hosts the Component that Posted this Message
IsResponse	+	[Low-level slot] Used by network layer for keeping tabs on which messages are replies and which are unsolicited

Table 1: Slots for OpenAIR messages.

Priority=

The relative priority of the Module that posted the Message. This parameter is RECOMMENDED.

```
<message priority="5"/>
```

There are five levels of execution priority for all Components in an OpenAIR system, represented by a float:

0.0 - Highest. Used for I/O, network and other systems-related processes

1.0 - Very High. Message scheduling; distributed blackboard / Dispatcher syncrhonization

2.0 - High. Used for processes that take care of reactive and reflex behaviors, as well as real-time data gathering like vision, hearing, etc.

3.0 - Medium. Used for e.g. short-term dialog planning and short-term action control (up to roughly 2 seconds*)

4.0 - Slow. Used for reflective processes, e.g. long-term memory access, knowledge manipulation, task planning

5.0 - Slower. Used e.g. for background knowledgebase management, variuos garbage collection, and logging

6.0 - Slowest. Background tasks, non-real-time tasks.

The priority= parameter is NOT REQUIRED, however, a Dispatcher that implements a priority scheme MUST understand the semantics of these particular priority levels. (Alternative priority schemes are allowed to live side by side with this one, should this be called for; in this case a different tag than priority= MUST be used.)

Level 0 is reserved for network and other systems and highest-priority processes; level 1 is reactive and reflex behavior; level 2 is fast behavior, typically deliberative; level 3 is relatively fast reflective behavior, level 4 is long-term planning and slow behaviors and level 5 is background processes. Each level can be divided as need be by decimal numbers. For example priority level 4.1 is higher than priority level 4.12. No specification is provided for the decimal priorities - this will be determined in due time.

*Based on human performance data from A. Newell's (1990) *Unified Theories of Cognition.*

TimeToLive=

The (predicted or known) duration, measured in milliseconds from the point of the PostedTime timestamp, for which the contents of this Message are valid or true. Value is in milliseconds. This slot is RECOMMENDED.

```
<message priority="" timetolive="200"/>
```

A value of zero in this parameter means the same as leaving the parameter out of the Message tag– i.e. "no value specified".

Type

The type of a Message is stored in the Type slot. All messages MUST contain a Type. The Type MUST be a dot-delimited string on which the Dispatcher can do matching to Message Types which Modules have Subscribed to. The Message Type implementation MUST be case-sensitive.

```
<type>Internal.Perception.Hearing.Voice</type>
```

It is RECOMMENDED that the Message Types be based on a thorough taxonomy or (preferably) an ontology of system message types. Future work by MINDMAKERS includes the development and standardization of such taxonomies for various domains and purposes, enabling better cross-compatibility between separate software development efforts.

For discussion about matching against Type for queries and Subscriptions see Subscription and Query protocols.

ID

The unique identification code for this particular Message. All messages MUST contain this ID. The ID MUST be a global unique identifier (GUID) that uniquely identifies this Message across Dispatchers and across time, for all eternity, until the end of the Known Universe.

```
<id>1f7c9745-db80-4b31-af64-5e089fddf623</id>
```

From

This is the unique name of the Poster. All messages MUST contain a From.

```
<from>Unique-Module-Name-123</from>
```

To

The unique name of the Receiver. For Modules this is the (unique) name of a Dispatcher, for a Dispatcher this is the (unique) name of a Module. This slot MUST be filled.

```
<to>Blackboard-3000</to>
```

Cc

The unique name of the Receiver. Used for getting messages of Type X to Modules that have not registered for Messages of Type X. For Modules this is another Module. Dispatchers make no active use of this slot. This slot is NOT REQUIRED.

```
<cc>MyOtherModule-2001</cc>
```

PostedTime

The precise moment in time at which the Poster let go of (i.e. Sent, Posted) the Message. This timestamp defines the very moment in time at which the Poster committed to the content and meaning of the message as being true. This timestamp MUST be in the Message.

```
<postedtime="1076264657" msec="110"  text="08.02.2004␣18:24:17:110"/>
```

The sec= and msec= parameters MUST be included. The more human-readable and information-complete text= parameter MAY be included.

ReceivedTime

The time at which the Receiver slot receives the Message. This is the Dispatcher listed in the To slot if the Message originated with a Module; it is the receiving Module that is registered for the message's Type, or the Module listed in the Cc slot, if the message originated with the Dispatcher. This timestamp is REQUIRED. (To have a message delivered directly to a module, regardless of whether it is registered for messages of that type or not, use the Cc slot.)

```
<receivedtime sec="1076264657" msec="111" text="08.02.2004␣18:24:17:111" />
```

Content

The actual content of the Message, equivalent to the letter inside an envelope. The Content slot has a parameter called language which specifies which language the content is listed in, e.g. XML, Lisp, English, etc. This slot is REQUIRED.

```
<content language="english" version="modern">
   This is modern english
</content>
```

The parameter language= is REQUIRED when the Content slot is filled.

Reference

A reference to an already-posted message. This tag is used as part of the History and the InReplyTo slots. Use of this tag is NOT REQUIRED.

```
<reference id="" stored="" postedsec="" postedmsec="" />
```

All parameters in this tag are REQUIRED.

InReplyTo

The (unique) ID of the original message to which this message is a reply. Another way to view it: The ID of an intial Message that initiated a series of one or more Message postings by one or more Modules. If Module A Posts Message M1 and fills in this slot with the ID of the same Message M1, and subsequently a Module B receives M1, to which it produces its own Message M2 in reply, then Module A MUST copy the value of this field from M1 into the same field in its own new Message M2. This slot is NOT REQUIRED.

```
<inreplyto>
   <reference id="" stored="" postedsec="" postedmsec="" />
</inreplyto>
```

The id= parameter MUST reference a valid Message GUID; the stored= parameter MAY be included and should reference a unique name of a Storage Location; postedsec= and postedmsec= MUST hold the Posted-Time of the particular Message to which the Reference refers, the one whose ID is listed in the id= parameter.

Stored

The (unique) name of the system Component where the full message, along with its content, is stored. This slot is NOT REQUIRED.

```
<stored>Blackboard-1000</stored>
```

History

A push-down stack containing ID-PostedTime data of the trace history from the originating Message to the current message. The first ID in this slot is always the same as that in the InReplyTo slot, if the InReplyTo slot is filled.

```
<history>
   <reference id="" stored="" postedsec="" postedmsec="" />
</history>
```

References to the messages that contained the content on which this Message is based; a trail of Message ID-PostedTime pairs, stored as a time-ordered collection of message IDs (the oldest messages last). This slot is used to trace which inputs lead to which outputs; in an implemented system it provides a way to trace causal Message chains. For example, we could have an initial Message stemming from a perception of a sound, to the recognition of what that sound is (e.g. the word "hello"), to an output plan (e.g. "greet person"), to an action plan (e.g. "use manual gesture"), to which motor spec (e.g. "raise hand to face"), to actual output events (e.g. the robot waves). Using this history list, deveoper can retrieve all the messages that the previous Modules used to compute the previous results leading up the current one. The recording of this history should be turned on and off depending on which state of development a system is in.

References MAY be grouped using the group tag:

```
<history>
   <group author="">
      <reference id="1" stored="" postedsec="" postedmsec="" />
      <reference id="2" stored="" postedsec="" postedmsec="" />
   </group>
</history>
```

The Group tag has an author= parameter, which is NOT REQUIRED, but if it is included, MUST contain the name of the Component that added the Group.

Comment

A free-form slot for holding ASCII data. Meant for human consumption and debugging purposes. This slot is NOT REQUIRED.

```
<comment>this is an example comment</comment>
```

6 XML Message Format Example

Here we show the XML format required for the above message format.

```
<message priority="5" timetolive="500">
    <id>1f7c9745-db80-4b31-af64-5e089fddf623</id>
    <type>Internal.Status.Report</type>
    <from>Domino-Module-3000-B</from>
    <to>Blackboard-1</to>
    <cc>Domino-5500</cc>
    <comment>My own comment</comment>
    <stored>Blackboard-1000</stored>
    <inreplyto>
          <reference
                  id="1590997-db80-4b31-af64-5e089fddf623882590"
                  stored="Blackboard-1000"
                  sec="1076264657"
                  msec="101"/>
    </inreplyto>
    <postedtime
          sec="1076264657"
          msec="110"
          text="08.02.2004 18:24:17:110"/>
    <receivedtime
          sec="1076264657"
          msec="111"
          text="08.02.2004 18:24:17:111"/>
    <content language="XML" version="1.0">
         <mycontentspecifictag>My private content</mycontentspecifictag>
    </content>
    <history>
          <group author="CoolModule-200">
                  <reference
                          id="1590997-db80-4b31-af64-5e089fddf623882590"
                          stored="Blackboard-1000"
                          sec="1076264659"
                          msec="101"/>
          </group>
          <group author="CoolModule-200">
                  <reference
                          id="4b31-1590997-db80-af64-5e089fddf623882590"
                          stored="Blackboard-1000"
                          sec="1076263550"
                          msec="101"/>
                  <reference
                          id="db80-1590997-4b31-af64-5e089fddf623882590"
                          stored="Blackboard-1000"
                          sec="1076264457"
                          msec="101"/>
          </group>
    </history>
    <origin>92.168.0.1</origin>
</message>
```

7 Network Transmission and Routing

All Components in OpenAIR communicate via the network. Current reference implementations use an Ethernet TCP/IP network, but there is no reason for not allowing future extensions to use the UDP or GSM/GPRS standards, or even communicating directly in-memory for Components in the same executable, or via streams if running on the same computer. Before any data transmission can happen, a Component in an OpenAIR system must make a valid connection to one or more Dispatchers.

When a Message is Sent/Posted to the network it is converted from whatever internal format it has to XML. As detailed above, the XML MUST be en closed in the outer tag

```
<message>...</message>
```

where ... is any XML formatting, which MAY include any number of <TAB>s and <CR>s.

When a Component A wishes to transmit a Message to another Component B, it first needs to locate the network address of B. OpenAIR does not specify how this should be done; a RECOMMENDED way is having a central naming server (CNS) that can be queried for the location of Component B. Such a CNS would most likely be passive and merely receive binding messages from each Component as they start up, keep a list of all the Components it knows about, and lookup the answer when asked. A CNS MUST be Basic OpenAIR Compliant.

The transmission protocol further includes a preceding Header, a trailing Reception Acknowledgment, as well as a Timeout.

Transmission Data

When Component A knows the network address of Component B it opens a Socket connection to this address and sends the entire message right away, prepending a message header, if connected. The Receiver (Component B) should be listening for incoming connections and when one is requested it MUST start listening for the header, which is 12 bytes long and has the format

```
[M] [e] [s] [s] [a] [g] [e] IVZ [a] [b] [c] [d]
```

IVZ means the integer value zero; a b c d are 32-bit signed integers (not the characters abcd).

The 32-bit signed integer byte order is Little Endian with 'a' being the least significant byte and 'd' the most significant byte. The integer specifies the number of bytes yet to come in the transmission, meaning the actual number of characters in the XML to be received (remaining to be transmitted).

Reception Acknowledgment

Once Component A has received the full Message in XML form from the Sender it MUST check that the XML is valid and process the Message with the intent to produce an acknowledgement Message, which can be either (a) a default I received the message fine Message, which has the form RECEIVE_ACCEPT, or (b) the non-default I received a message, but didnt understand it Message, which has the form RECEIVE_FAILED.

Timeout

When the Receiver receives the header it counts this as a valid connection from another Component. The Receiver MUST keep listening on this Socket until the Sender decides to close the connection, which it may do without further warning. If a Component does not receive any data on such a connection within a given timeout of 8 to 10 seconds, or if something other than the above header is received over the connection, the Receiver MUST consider the connection invalid and close it without warning.

Low-Level Protocol

There are two low-level slots in the protocol; these are fields/tags that have to do with efficiency and "book-keeping" of the transmission. As a result only the network level should see them - they SHOULD not be visible in the programming API. These slots are called IsResponse and Origin. They are REQUIRED.

A Component MAY support a standard set of lower-level protocol Messages which do not propagate up into the Component logic above (including the programming API). These are RECOMMENDED to include message types such as PING to be responded to by a message with type PING_SUCCESS, which is the standard way of checking that a Component is alive and well, as well as testing the network transmission speed.

A future extension to support transmission of binary data, in addition to the XML, may be implemented in OpenAIR at a later date.

Origin

Upon reception of a Message via the network the Socket layer MUST add a low-level field to the Message structure called Origin, which MUST contain the IP Address of the computer on which the software runs which sent the Message. This can usually only be done by the Socket layer as only it knows, or can query, the Message Senders IP address. The XML for the Origin field should be

```
<origin>92.168.0.1</origin>
```

This is done because the sender might in fact not know its own global IP address. An example is a home computer: It often thinks it has the IP address given to it by its router, which might be 10.0.1.2 or 192.168.1.15, etc. But if someone in the world wants to contact such a home computer they need to know its public IP address. One can verify a home computer's IP address by going to a website which publishes it, e.g. WhatIsMyIP. Such a website does exactly what the mechanism here describes, since the Receiver can look back and see where the message came from, thus retreiving the router address, and thus the public IP of the computer which initiated the socket connection.

IsResponse

Any Message which is a reply to a prior Message to should include this low-level Message field. When a Sender sends Message M1 to Receiver, where M1 is an unsolicited message requiring a response (e.g. a RECEIVE_ACCEPT), the Receiver MUST include this tag with filled-in content. This way, should a stray reply message accidentally be received unexpectedly the Receiver knows to disregard it and not treat it as a new message, which could cause a message feedback loop. The XML for the IsReponse field is

```
<isresponse>...</isresponse>
```

where ... is any any non-zero length text; it IS RECOMMENDED that it be the GUID of the original unsolicited Message to which the Receiver is responding.

Re-Connect Protocol

The Component which opens the Socket connection MUST to maintain it. If the connection is unintentionally broken, the Component MAY try to re-establish the connection over a given time-out period at a regular interval. The Receiver MUST accept a broken connection as an end of communication, and it MUST allow new connections to be made in the future. Any Component MAY allow multiple simultaneous connections from many other Components.

8 Subscription and Query Protocols

To receive Messages, Modules can (a) Subscribe for Message Types and/or (b) Retrieve Messages (by querying for ID, timestamps or Type) from a Storage Location, which can be e.g. a database or a Dispatcher.

Subscription

To Subscribe to one or more Message Types a Module MUST send a Message with Type AIR.Subscribe to a Dispatcher and one or more Message Types that the Module wants to Subscribe to. The Message Types to be Subscribed to MUST be included in the Content slot of the Message, using the following syntax:

```
<triggers>
   <trigger from="Dispatcher-0" type="Input.Hearing.Voice">
</triggers>
```

The from= MUST reference a valid Dispatcher; the type= parameter MAY reference a valid Message Type. A from= can also be specified for the tiggers tag:

```
<triggers from="MyDispatcher">
```

in which case it applies to all listed triggers inside the tag.

Self-trigger blocking MUST be the default behavior of a Dispatcher; in other words, if a Module posts a Message Type to which it is also Subscribed, the Dispatcher MUST not dispatch the Message back to the same Module. In case self-triggering is a desired behavior, the following syntax MAY be used for either or both tags (when used in both tags, the ¡trigger¿ overrides the ¡triggers¿ setting):

```
<triggers allowselftriggering="yes">
<trigger from="" type="" allowselftriggering="yes">
```

It MUST be possible to Subscribe to a message Type using patterns with implicit wildcards to the right of the string, e.g. x.y matches any string which starts with x.y. To take an example, x.y MUST match x.y.a, x.y.b, and x.y.z; x.y MUST not match a.x.y or a.x.y.z. This applies to the extensions as well: Subscriptions for Types including an extension :xxx MUST be supported; i.e. x.y:b MUST not match x.y:a but x, x.y and x.y:a MUST match x.y:a. Thus, when a Module Subscribes to x.y it is equivalent to Subscribing to x.y.*:* .

Dispatchers MUST service requests for Retrieving Messages. The queries can be done by specifying a Message ID, a Message Type, a pair of timestamps indicating an interval, and a combination of the last two, i.e. a Type plus a time interval.

Retrieval - Preliminary Suggestions

To Retrieve one or more MessageTypes a Module MUST send a Message with Type Internal.Retrieve to a Dispatcher and one or more Message Types that the Module wants to Retrieve. The Message Type to be retrieved, and the Dispatcher from which it is to be retrieved, MUST be included in the Content slot of the Message.

```
<retrieves>
   <retrieve from="Blackboard-1" type="Output.Plan.Speech">
</retreives>
```

An extended retrieval specification MUST follow this syntax:

```
<retrieves>
   <retrieve from="BBX" type="Input.Sens.UniM.Hear"/>
   <retrieve from="BBX" type="Input.Person.Found.True">
      <latest>10</latest>
      <aftertime sec="10" msec="233"/>
      <untiltime sec="12" msec="466"/>
      <lastmsec>200</lastmsec>
   </retrieve>
</retrieves>
```

Any of the constraining tags (latest, aftertime, untiltime, lastmsec) may be omitted when constructing a query. Aftertime is exclusive of the time specified; untiltime is inclusive of time specified; lastmsec is inclusive of the time specified.

9 Implementation

The following is intended to help standardize the programmer's experience when using adapters that implement the OpenAIR specification.

The Java OpenAIR implementation has a number of basic objects for working with time, Internet addresses, and collections and dictionaries of strings and objects. It also contains an advanced network client and server, which can transmit any of the basic objects seamlessly to be re-instantiated by the receiver on the other side.

Source

JAVA AIR implementation can be downloaded from Mindmakers.org/projects/openair/

Plug Instantiation

The main OpenAIR object MUST be called JavaAIRPlug. It can either be instantiated by inheritance so your object in effect is an AIRPlug, or it can be used as a variable in your object, where you call functions inside to communicate with the server.

When instantiating the JavaAIRPlug the following four pieces of information MUST be provided:

- A unique name identifying the Plug
- The hostname of the AIR Server
- The TCP/IP port number of the AIR Server (10000 is RECOMMENDED)
- The handler of message received from the server

The handler is either itself in the case of inheritance or another object which implements the MessageAcceptor interface. The Plug MUST include the following methods.

The method

```
public boolean init();
```

will connect to the AIR Server and return true if successful. From then on the Plug will try to maintain the connection and automatically try to re-establish the connection if broken. One can ask the plug if it is currently connected to the AIR Server by calling the method

```
public boolean isConnected();
```

which returns true if the call succeeded, false otherwise.

Publishing (aka Posting, Sending)

When one wishes to publish, or post, information to a Dispatcher via the AIR Server these are the methods one can call:

```
public boolean postMessage(String dispatcher, String messageType, String cc);
public boolean postMessage(String dispatcher, String messageType,
       String content, String language, String cc);
public boolean postMessage(String dispatcher, Message messageType, String cc);
```

These methods are REQUIRED. The cc field is usually left empty unless the message is intended for another client that hasn't subscribed to that Message Type specifically.

Subscribing

The following functionality is REQUIRED:

When one wishes to subscribe to a Message Type with a particular Dispatcher, via the AIR Server, these are the methods one can call:

```
public String addTrigger(String dispatcher, String messageType);
public String addTrigger(String dispatcher, String messageType,
       boolean allowSelfTriggering);
public String addTrigger(String dispatcher, String messageType,
       String retrieveXML);
public String addTrigger(String dispatcher, String messageType,
       String retrieveXML,
       boolean allowSelfTriggering);
```

Each of these methods will add a trigger, which is what a Subscribed-to Message Type is called, to the current list of triggers and inform the AIR Server, and return an string ID for the trigger. To see which triggers are currently subscribed to, use

```
public ObjectDictionary getTriggers();
```

An ObjectDictionary is a dictionary of named Objects, a collection of string = Object where each string is unique.

To remove a single trigger or all, use

```
public boolean removeTrigger(String id);
public boolean removeAllTriggers();
```

To "manually" retrieve messages from a Dispatcher, use

```
public ObjectCollection retrieveMessages(String dispatcher, String xml);
```

An ObjectCollection is a collection of unnamed Objects, much like the ObjectDictionary above, without the unique string entry names.

The following functionality is NOT REQUIRED:

To be notified about disconnection and reconnection events, i.e. when the AIR Server connection is lost and regained, one can overwrite the following two methods:

```
public boolean primaryReconnectNotify();
public boolean primaryDisconnectNotify();
```

The JavaAIRPlug object relies heavily on other objects providing the functionality needed for network connections, sending and receiving messages and more basic objects such as Time and TCPLocation.

The reference implementation communicates fully with the C++ library the CoreLibrary (on SourceForge: http://corelibrary.sourceforge.net/) which has scores of basic objects including seamless multiplatform networking and threading and the ability to transmit any object anywhere.

Acknowledgments

This work was in part supported by a grant from the Icelandic Research Fund and by a Marie Curie European Reintegration Grants within the 6th European Community Framework Programme.

– EOF –