UNIVERSITE DE TECHNOLOGIE DE COMPIÈGNE
Département de Génie Informatique

OMAS - Vocal Interface

Jean-Paul Barthès

BP 349 COMPIÈGNE
Tel +33 3 44 23 44 23

Email: barthes@utc.fr

N275 v1.0
September 2012

Warning

This document discusses how to program a vocal interface for OMAS Personal Assistant agents. It applies to OMAS version 9.0.3 and up)

Keywords

Personal Assistant, vocal interface

Revisions

Version	**Date**	**Author**	**Remarks**
1.0	Sep 12	Barthès	First Issue

Contents

1 Introduction

We consider the problem of giving a PA the possibility of communicating using a vocal interface. There are many ways of implementing the interface and we discuss the ones more in line with the OMAS approach. We assume that we have a speech-to-text and a text-to-speech software that runs in parallel with OMAS agents. There are essentially two ways of connecting a PA with the voice system: (i) directly using sockets; (ii) indirectly using messages.

2 Direct Socket Connection

The direct socket connection is justified whenever the user wants to be connected directly to her PA. This means that the voice interface implements a direct communication between the voice component and the PA and is not meant to be heard by other PAs in the coterie. In that case a tight coupling makes sense. We thus use a direct UDP socket connection as shown Fig.1.

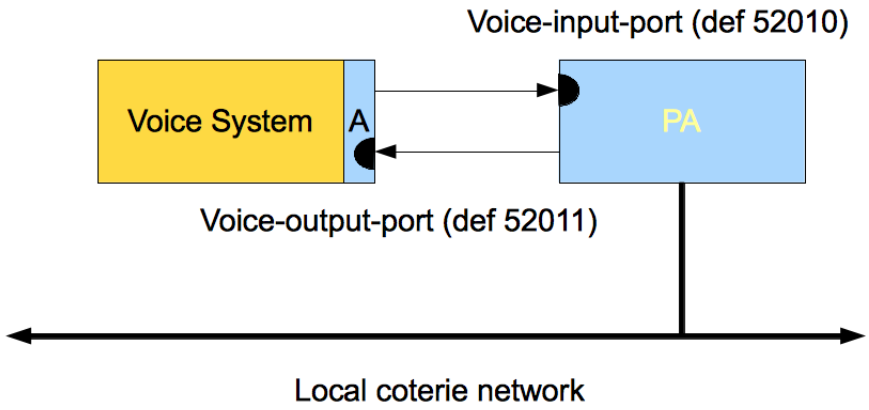

Figure 1: Direct connection using sockets

In Fig.1 the voice-input-port is the PA port receiving the string resulting from a speech to text conversion of the Voice System. The voice-output-port is the port of the voice system receiving the string to be converted to speech.

2.1 Implementation

To implement this solution, one must tell OMAS what are the input port (PA side) and the output port (voice component side) of the connection. This is done when creating the PA by using the :voice-input socket and :voice-output-socket options. If the options are not used then sockets 52010 and 52011 are used by default.

The code that deals with the messages on the OMAS approach is now described.

When the PA is created with a :voice t option, then the voice interface is initialized as follows:

```
;;;------------------------------------------------------------ NET-INITIALIZE-VOICE

(defun net-initialize-voice (agent)
  "This function opens a socket if necessary.
Arguments:
```

```
   agent: assistant agent controling the vocal interface
Return:
   :done"
  ;; create a receiving socket to be used on the PA side
  (unless (voice-input-socket agent)
    (setf (voice-input-socket agent)
      (socket:make-socket
       :type :datagram
       :local-port (voice-input-port agent)
       ))
    )
  ;; create a sending socket
  (unless (voice-output-socket agent)
    (setf (voice-output-socket agent)
      (socket:make-socket :type :datagram))
    )
  ;; set up receiving process unless it already exists
  (unless (voice-receiving-process agent)
    (setf (voice-receiving-process agent)
          (mp:process-run-function "Net Voice Receive" #'voice-receiving agent)))
  :done)
```

This function creates two sockets: (i) a receiving socket (voice-input-socket agent); and (ii) a sending socket (voice-sending-socket agent). The sockets are defined on the local machine and use a UDP protocol. The receiving socket is used by a function called voice-receiving that is essentially a loop:

```
;;;------------------------------------------------------------------ VOICE-RECEIVING

(defun voice-receiving (agent &aux text)
  "receives a message from the voice input and puts it into the to-do slot of the
agent."
  (loop
    (multiple-value-bind
          (raw-buffer size)
        (socket:receive-from (voice-input-socket agent)
                             (voice-max-message-length agent)
                             :extract t)
      ;; transform unicode string into internal string
      ;; truncate is added as a safety measure
      (setq text (excl::octets-to-string raw-buffer ;:external-format :fat
                                         :truncate t))
      ;; process the received string
      (omas::assistant-process-master-text agent text)
      )))
```

When the PA wants to send a string to the voice system it uses the net-voice-send function defined as follows:

```
;;;------------------------------------------------------------------ NET-VOICE-SEND
```

```
(defun net-voice-send (agent message)
  "sends a message to the voice interface using the UDP protocol.
Arguments:
   message: a String less than *max-message-length* bytes long
Return
   code returned by the send-to function."
  ;; check arg
  (unless (stringp message)
    (error "message should be a string rather than: ~S" message))
  (if (> (length message) (voice-max-message-length agent))
      (error "message too long:~%~S" message))
  ;; trace
  (format t "~%; net-voice-send / sending message to ~A:~S~%;  ~%;   ~S"
    (voice-ip agent) (voice-output-port agent) message)
  ;; send message
  (when (voice-output-socket agent)
    (socket:send-to (voice-output-socket agent) message (length message)
                    :remote-host (voice-ip agent)
                    :remote-port (voice-output-port agent))))
```

2.2 Setting the Stage

In order to activate the voice mechanism, one has to use the following options when defining the personal assistant:

```
(defassistant :albert :voice t :voice-input-port 4000 :voice-output-port 4001 ...)
```

2.3 Discussion

The direct connection approach is well suited for accessing a PA with minimal overhead. It inserts the string resulting from the speech-to-text conversion into the TO-DO slot of the PA, which wakes up the converse process managing the dialog. The string is then processed as if it had been typed into the master's pane of the interface window.

Then, because the voice connection is enabled, every time something is written to the assistant pane, it will be sent to the voice synthesizer.

2.4 Voice Component on a Different Machine

One can install the voice component on a different machine as long as it sends the messages to the right socket of the PA.

On the PA side it is necessary to specify the IP of the machine containing the voice component and its port number if different from the default one, e.g.

```
(defassistant :albert :voice t :voice-input-port 4000 :voice-output-port 4001
      :voice-ip "skopelos" ...)
```

This means that the PA will receive transcribed vocal messages on port 4000 and the voice component located on skopelos will receive strings to transform into speech on port 4001.

3 Message Connection

In this approach we get the string corresponding to the user's utterance and send it to the PA using an OMAS message. There are two possible ways of implementing it: (i) directly producing OMAS messages and sending them on the local coterie loop; or (ii) using a postman to produce the message.

3.1 Synthesizing OMAS Messages

The approach consists in wrapping the voice recognition software in a piece of software that will send an OMAS message on the local coterie loop. Then, the answer message will be analyzed and sent to the speech-to-text processor. The structure is described Fig.2.

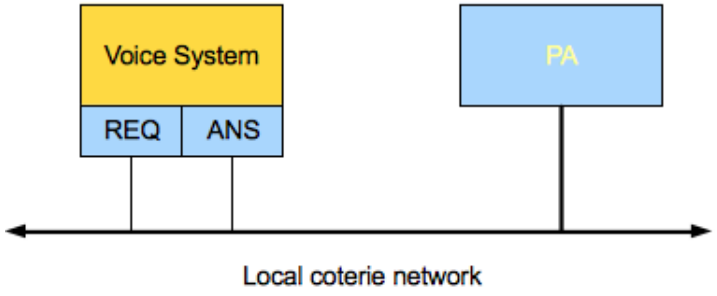

Figure 2: Vocal Connection using OMAS messages

In this approach the voice system is interfaced to the OMAS platform by a software layer that transforms the vocal input into an OMAS message. The message is then put onto the local coterie LAN and sent to the PA. The PA then sends an answer back to the ANS part of the vocal interface.

3.2 Implementation

The voice component is implemented with a language chosen by the designer, as long as the produced message obeys the OMAS syntax and is broadcast onto the local coterie loop. The answer message must then be picked up from the OMAS messages circulating on the local coterie loop.

On the PA side, processing the message is easy, provided that a specific skill is defined in the PA. For example:

```
;;;=============================================================================
;;;                            PROCESS-VOICE-INPUT
;;;============================================================================= skill
;;; receives a label, checks if the category exists. If not, creates it, and sends
;;; a message to the NEWS-PUBLISHER agent.

(defskill :process-voice-input :CIT-STEVENS
  :static-fcn static-process-voice-input
  ;:dynamic-fcn dynamic-process-voice-input
  ;:timeout-handler timeout-handler-process-voice-input
  )

(defun static-process-voice-input (agent message text)
```

```
   "function that receives a string from the speech to text program and posts it ~
    into the to-do slot of the PA.
Arguments:
   agent: the PA
   message: the input message
   text: the input text string
Return
   nothing significant."
   (declare (ignore message))
   (format t "~%; static-process-voice-input / text: ~S" text)
   ;; post the received string into the to-do slot of the PA
   (setf (omas::to-do agent) text)
   ;; this will trigger the converse process which presumably is waiting on
   ;; some state
   ;; the output of the converse process should be both a string to be printed into
   ;; the window and sent to the text to speech converter (to do in the dialog part)
   (static-exit agent :done))
```

This skill will send the string to the converse process to be examined by the dialog mechanism.

Returning an answer from the dialog is more difficult and requires to add a special function for broadcasting the answer onto the local coterie net, e.g.:

```
(defun send-voice-answer (text)
   "oribe sends an answer message"
   (let ((answer (make-instance 'omas::message :from :oribe :to :all :type :inform
            :action :dummy :args (list text))))
     (send-message answer)))
```

In the example the PA is called :CIT-STEVENS.

Variant Another possibility is to specify the :voice parameter in the defassistant macro and to overload the net-voice-send function as follows:

```
;;; clobber the current function
(fmakunbound 'omas::net-voice-send)

;;; redefine OMAS function
(defun omas::net-voice-send (agent text)
  "broadcasts a message using the UDP protocol.
Arguments:
   agent: our PA
   text: a string less than max-message-length bytes long
Return
   code returned by the send-to function."
  ;; check arg
  (unless (stringp text)
    (error "message should be a string rather than: ~S" text))
  (if (> (length text) (omas::voice-max-message-length agent))
      (error "message too long:~%~S" text))
```

```
(let (message)
  ;; trace
  (format t "~%; net-voice-send / sending message to ~A:~S~%;  ~%;   ~S"
    (omas::voice-ip agent) (omas::voice-output-port agent) text)
  ;; make answer message
  (setq message (make-instance 'omas::message
                  :from :CIT-STEVENS :to :all :action :speak :type :answer
                  :args (list text)))
  ;; put the message on the LAN of the local coterie
  (send-message message)
  ))
```

Note that our assistant is :CIT-STEVENS and that the message is sent as a broadcast message. Its type is :answer but it could also be an inform message or any type that the voice component could parse.

3.3 Discussion

The method is not recommended for several reasons:

- synthesizing an OMAS message is not difficult but if the OMAS internal syntax of the messages changes, the interface will no longer work.
- parsing the OMAS answer is more tricky, and again assumes that the syntax of the internal messages will not change.
- grabbing the answer obliges to process every message that circulates on the loop.
- processing a request message on the PA side involves a lot of useless overhead, making the program prone to failures.

The approach is thus not recommended.

4 Using a Postman

The approach here consists of using a postman to receive the input translation of the vocal message and to put it onto the local coterie loop. This way, it can be sent to the PA or broadcast to all the agents on the loop. It can be used if we want to send the vocal input to several agents (or to all agents) of the local coterie. The approach consists of assigning a specific postman (transfer agent) to the vocal component as shown Fig.3.

4.1 Implementation

Here the vocal component is connected to the Postman directly using sockets as for the PA direct connection case. The main point is that the postman will take care of formatting the OMAS messages using the right syntax. The postman embodies a proxy agent for the vocal system. If we call the postman :VOICE for example, then we can send and receive messages from the voice component.

The following code is an *untested* example of a possible :VOICE postman.

```
;;;-*- Mode: Lisp; Package: "CIT-VOICE" -*-
;;;============================================================================
;;;12/09/08
```

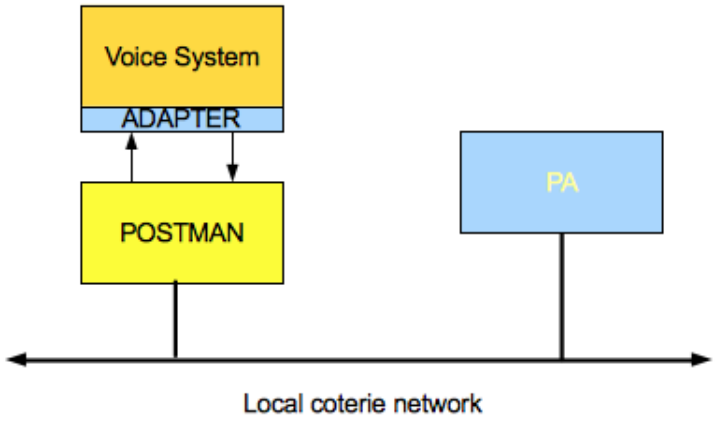

Figure 3: Connection to the vocal component using a postman

```
;;;                              AGENT POSTMAN :CIT-VOICE
;;; Copyright barthes UTC, 2012
;;; Postman to handle voice input
;;;=============================================================================

;;; the postman is only valid for ACL
;;; the voice component is assumed to run on the same machine as the postman
;;; uses ports 900 and 9001


#|
2012
 0908 creation
|#

(defpackage :CIT-VOICE (:use :moss :omas :cl #+MCL :ccl))
(in-package :CIT-VOICE)

;;;=============================================================================
;;;                              Creating postman
;;;=============================================================================
;;; the :raw parameter indicates that we will NOT use default skills

(omas::defpostman :CIT-VOICE
    :known-postmen ((:VOICE . "127.1")) :raw t)

;;;=============================================================================
;;;                                 Globals
;;;=============================================================================

(defparameter *voice-input-port* 9000 "port for receiving voice input")
```

```
(defparameter *voice-output-port* 9001 "socket for sending string to voice")

(defparameter *voice-input-socket* nil)
(defparameter *voice-output-socket* nil)

(defparameter *voice-receive-process* nil)

(defparameter *max-message-length* 4096)

;;;============================================================================
;;;                            Service functions
;;;============================================================================

(defun voice-receiving (agent)
  "receive a message from the voice input and put it into the to-do slot of the
agent."
  (declare (special *voice-input-socket* *max-message-length*))
  (let (raw-buffer size)
  (loop
    (multiple-value-bind
          (raw-buffer size)
        (socket:receive-from *voice-input-socket* *max-message-length* :extract t)
      (format *debug-io* "+++ ~S ~S" raw-buffer size)
      ;; writes text into the assistant pane of the interface window and puts the
      ;; string into the to-do slot of the agent, reviving the dialog process
      (omas::assistant-process-master-text agent
                                           (make-string-from-buffer raw-buffer size))
      ))))

;;;---------------------------------------------------------------------- VOICE-SEND

(defun voice-send (message)
  "Sends a message to the voice interface using the UDP protocol.
Arguments:
   message: a String less than *max-message-length* bytes long
Return
   code returned by the send-to function."
  ;; check arg
  (unless (stringp message)
    (error "message should be a string rather than: ~S" message))
  (if (> (length message) *max-message-length*)
      (error "message too long:~%~S" message))
  ;; otherwise send message
  (when *voice-output-socket*
    (socket:send-to *voice-output-socket* message (length message)
                    :remote-host "127.1"
                    :remote-port *voice-output-port* agent)))
```

```
;;;=============================================================================
;;;================================ SKILLS =====================================
;;;=============================================================================

;;;============================================================================= skill
;;;                                  CONNECT
;;;=============================================================================
;;; the connect skill sets up the sockets for communication with the voice component

(defskill :CONNECT :CIT-VOICE
  :static-fcn static-connect
  )

(defun static-connect (agent message)
  "This function opens communication sockets if necessary.
Arguments:
   agent: assistant agent controlling the vocal interface
Return:
:done"
  (declare (ignore message)
           (special *voice-input-socket* *voice-output-socket*
                    *net-voice-receive-process*))
  ;; create a receiving socket
  (unless *voice-input-socket*
    (setq *voice-input-socket*
          (socket:make-socket
           :type :datagram
           :local-port *voice-input-port*
           ))
    (format t "~%;*** net-initialize-voice / *voice-input-socket* ~S"
      *voice-input-socket*))

  ;; create a sending socket
  (unless *voice-output-socket*
    (setf *voice-output-socket*
      (socket:make-socket :type :datagram))
    (format t "~%;*** net-initialize-voice / *voice-output-socket* ~S"
      *voice-output-socket*))

  ;; add info to the list of connected agents
  (unless (assoc :voice (omas::connected-postmen-info agent))
    (setf (omas::connected-postmen-info agent)
      (append (omas::connected-postmen-info agent)
              (list (cons :VOICE "127.1")))))

  ;; refresh postman window, which will show connections
  (omas::agent-display (omas::%agent-from-key (omas::key agent)))
```

```
  ;; set up receiving process unless it already exists
  (unless *net-voice-receive-process*
    (setq *net-voice-receive-process*
          (mp:process-run-function "Net Voice Receive" #'voice-receiving agent)))
  (static-exit agent :done))


;;;============================================================================ skill
;;;                                DISCONNECT
;;;============================================================================
;;; do we need that? sockets are created with a reuse-address option. If they are
;;; not closed on output, they can be reused when restarting...

(defskill :disconnect :voice
  :static-fcn static-disconnect)

(defun static-disconnect (agent message site-key)
  "closes all sockets and kills receiving process"
  (declare (ignore message))
  (when *voice-input-socket*
    (close *voice-input-socket*)
    (setq *voice-input-socket* nil))
  (when *voice-output-socket*
    (close *voice-output-socket*)
    (setq *voice-output-socket* nil))
  (when *receiving-process*
    (mp:process-kill *receiving-process*)
    (setq *receiving-process* nil))
  ;; must update postman window
  (setf (connected-postmen-info agent)
    (remove site-key (connected-postmen-info agent) :key #'car))
  ;; refresh postman window
  (agent-display (%agent-from-key (key agent)))
  (static-exit agent :done))

;;;============================================================================ skill
;;;                                  SEND
;;;============================================================================

(defskill :send :voice
  :static-fcn static-send)

(defun static-send (agent in-message message-string message)
  "skill that sees every message and filters those for the voice system,
arguments:
   agent: postman
   in-message: incoming message
   message-string: message to send, a string (ignored)
```

```
   message: message to send, an object"
  (declare (ignore in-message message-string))
  (let (text)
    ;; check if message is for voice
    (when (eql :voice (omas::to! message))
      ;; yes transfer the content
      (voice-send (omas::args message)))
    ;; no, get out
    (static-exit agent :done)))

;;; ============================================================================

:EOF
```

4.2 Discussion

The postman implementation of the VOICE interface is intended to be used when the voice input could be heard by all agents. It has some overhead but the important point with respect to the previous approach is that all OMAS message formatting or parsing is done by OMAS.