KPNQWEST Italia

/Author: Max Muzi <massimiliano.muzi@kqi.it>
Last update: 2011-01-13
Location: http://soap.kqumg.it/chaka/current/0.9.5/docs/kqumsg0.9.5.manual.en.pdf
Helpdesk: <soap-support@kqumg.it>

KQ UMG SOAP Gateway (KQUMSG1)

User manual

Covering release 0.9.5

1. Overview

The KQ UMG SOAP Gateway (KQUMSG1) enables customer applications to access most of the main features of the *kqumg.it* messaging platform through the SOAP protocol. The service is available over HTTP and HTTP-SSL/TLS.

Currently (April 2010), the following messaging types are supported:

- standard Fax/SMS messages and bulk SMS messages
- bulk mailmerge-like trackable email (*Unimail*)
- Microsoft® "mailmerge" Fax messages (*mailmerge-ng*)

Users can compose and inject new messages combining one or more documents and a recipient list (§2.2.2). Both documents (§2.2.1) and recipient lists (§2.2.3) can be uploaded separately and reused in distinct messages throughout the same session. The gateway can also be instructed to retrieve and use remote documents available through HTTP.

The status of pending outbound messages can be queried by calling special methods, either specifying a message *id* or a time range (§2.3). Besides, users can request that all destination status changes be asynchronously notified to their host, provided they can set up and expose an appropriate HTTP service (§3).

The full WSDL description of service operations and data structures is available at the following URLs:
http://soap.kqumg.it/chaka/current/0.9.5/Chaka.wsdl (KQUMGS1 service)
http://soap.kqumg.it/chaka/current/0.9.5/Chaka_CB.wsdl (client-side callback service)

The service is also available though HTTP+SSL/TLS at the following URL:
https://ssl.kqumg.it/chaka/current/0.9.5/Chaka.wsdl

2. Operations

Available operations can be grouped under three categories: *(1)* session activation, *(2)* upload and sending actions (§2.2), *(3)* destination status querying (§2.3).

All operations in group 2 and 3 assume that a session has already been successfully activated (§2.1).

2.1. Sessions

A new session is initiated by calling the method *start* with the following arguments:

userid	user-id
password	password
options	options

Upon success, a special token called *auth* is returned, which is to be used as the first argument in all subsequent operations. If an operation fails with numerical code 403 or 500 (see §4), the client should call *start* again, and repeat all intended operations using the newly returned *auth* token.

The *options* values determine some aspects of the gateway's behavior when performing operations within the current session. As of release 0.9.5, some of the *options* are still unimplemented.

contact-person	an email address (*required*)
soap-callback	user-side HTTP callback endpoint for status asynchronous notifications; service must implement both *ping* and *processStatusNotifications* operations as declared in Chaka_CB.wsdl (see §3).
max-browse-block100-count	max number of 100-record blocks returned by each call to *browseMessagesByDate* o *browseDestinationsBySeqId*. (Not yet implemented.)
max-remote-operation-time	max completion time for remote retrievals. (Not yet implemented.)
max-method-completion-time	max completion time for SOAP operations . (Not yet implemented.)

If *soap-callback* is specified, the gateway will try to invoke the *ping* operation as part of the session activation process (see §3).
The *contact-person* field is not optional: this address should allow to reach the person or the group responsible for developing or maintaining the client application, if serious problems arise or urgent communications are required.

Clients can call the *check* method to make sure the session token is still valid and know when it will actually expire. This methods returns a single *expires* element of type *xsd:Datatime*.

2.2. Document uploading and message injection

2.2.1. *storeDocuments*

Documents can be uploaded using either the *storeDocument* or *storeDocuments* operations. They have basically the same function, except the former can only handle one document at a time.

storeDocuments operation accepts two arguments:

auth	session token (see §2.1)
sources	sequence of elements of type *DocumentSource*

and returns a sequence of *storeDocumentResult* structures.

Each item in the *sources* sequence should include a single element that can be either of type *Document* or type *RemoteDocument*. In the former case, the document's content will be embedded in the *data* sub-element using base64 encoding. In the latter case, the *location* sub-element will specify the remote location (URI) where the document can be retrieved from. (Only the 'http' schema is supported for the moment). The system will automatically try and fetch every file specified as a *remoteDocument*. Each document shall have a name that uniquely identifies it during the session's lifetime (*unique-name*).

Every stored document can be later referred to by means of a *DocumentRef* element (see §2.2.2).

The *storeDocumentResult* items returned by this method describe the results of the call and comprise the following information for each of the successfully stored documents:

name	the document's unique name (`string '.' extension`)
size	size in bytes
md5	MD5 digest in hex format

Attribute *name* must be in the form *string* '.' *extension*, where *extension* is a well-known file extension identifying the file's content type, such as “doc”, “pdf”, “txt”, etc.

2.2.2. *sendMessage*

The *sendMessage* function will be used to assemble and submit a new message. The following arguments must be specified:

auth	session token (§2.1)
request-uid	(see below)
message	the message

A structure is then returned including the numerical id of the message and other relevant information. The main components of the *message* structure are the following:

subject	the subject
options	the options (*flags*, *scheduling*, etc..)

fax|*sms*|*umail*|*faxmsmerge*|*bulksms*	elements representing the actual message to be injected

An *sms/bulksms* message (standard/bulk SMS) has the following structure:

destinations	sequence of elements of type *Destination*
or *destination-list-name*	name of an uploaded destination list
short-msg	text of the SMS message

A *fax* message (standard Fax) has the following structure:

destinations	sequence of elements of type *Destination*
or *destination-list-name*	name of an uploaded destination list
documents	sequence of elements of type *Document* or *DocumentRef*

An *umail* message (Unimail) has the following structure:

destinations	sequence of elements of type *Destination*
or *destination-list-name*	name of an uploaded destination list
documents	sequence of elements of type *Document* or *DocumentRef*
options	special extra options

A *faxmsmerge* message (mailmerge-ng) has the following structure:

destinations	sequence of elements of type *Destination*
or *destination-list-name*	name of an uploaded destination list
msword-doc	Document/DocumentRef element including o referring to an MS-Word file

The message destinations (i.e. the *destinations* element in *fax*, *sms*, bulksms, *umail, faxmsmerge* elements) consist in a sequence of elements each including *address*, *cc1* and *cc1* fields and an optional *extra* element. The first field represents the address of the destination (namely, a phone number or an email address), while the other two include additional information to be linked to the destination and later returned as part of the querying operation results or in the feedback status callback calls. The *extra* element is a set of 10 free text attributes to be used as Unimail placeholder macro values.

Each destination in the list is assigned a unique numerical sequential id representing the position of the destination within the sequence, starting off with 1. That value will be returned to user as a *seq-id* element (see §2.3.1).

The document list of a Fax and Unimail message (i.e. the *documents* element) may embed full documents as *Document* elements (see above §2.2.1) or *DocumentRef* elements referring to previously stored documents (via *unique-name)*.

Unimail messages (i.e. the *umail* elements) include a special *options* element used to access and control some of the service's advanced features:

embed-files	if true, outbound generated messages will embed docs and images
from-name	sender's display name used in outbound messages
from-email	sender's address used in outbound messages
subject	subject in outbound messages (if unspecified, the request's subject will be used)
no-receipt-request	don't include return-receipt requests in outbound messages

The *request-uid* attribute helps cope with situations when the network connection gets broken before the client could see the server's response. If a call fails due to network problems, a client may (and normally should) resubmit the request using the same *request-uid*. If the previous call has indeed been successfully processed, the server will return a "cached" result and won't generate a duplicate message.

2.2.3. storeDestinationList

Clients can upload one or more destination lists using the method *storeDestinationList*. Uploaded lists are available throughout the session and can be used in any message by specifying their unique name in the *destination-list-name* element (see §2.2.2).

storeDestinationList accepts the following arguments:

unique-name	unique name of the list
*table	list*	a *DestinationTable* element or a sequence of *Destination* element (see §2.2.2)

A *DestinationTable* structure includes the following members:

address-column, cc1-column, cc2-column	names of the columns hosting *address*, *cc2* and *cc2* fields
columns	string of type *xsd:NMTOKENS,* representing the field names of the table. If it's not specified, the table's first row is assumed to be a column header
format	format used for the table data (currently, one of *semicolon-csv*, *comma-csv*, *tab-csv*, *json-array*)
data	table raw data (base64 encoded)
expected_count	(optional) number of rows the client expects to be read (if actual count doesn't match, an error is raised)

When a destination list table is being imported, the table data is parsed into a record list. Each record field is given a name according to the *columns* value or to the first row's content (if *columns* is not specified). Then, the address, cc1 and cc2 fields are selected based on the names specified in *address-column, cc1-column* and *cc2-column,* respectively. If any of these column names is not defined, the import process is aborted and an error is raised. On success a structure is returned including the list name (*name*) and number of the successfully imported records (*count*).

2.3. Destination status querying

The current status of message destinations can be queried by calling either *browseMessagesByDate* or *browseDestinationsBySeqId*, depending on whether a client is interested in a time range or in a single specific message.

The former allows a client to retrieve the detail summaries of all messages submitted within a given period, as defined by *since* and *until* arguments. The returned structure will include a sequence of items of type *MessageDetail*, each of which will in turn comprise the message unique id (*msg-id*), the subject (*subject*), the creation time (*created*), the overall number of recipients (*dest-count*) and the number of destinations having reached a final (or preudo-final) status (*completed-dest-count*). Additionally, the status details of the first destination (*first-dest*) is included for convenience.

The *browseDestinationBySeqId* operation returns the current status of a given message's specific destinations. Arguments *first* and *last* define a range of sequential id numbers within the destination list. The returned structure will include all destinations whose sequential id is equal or greater than *first* and less or equal to *last*. Each destination's status is represented by an element of type *DestinationDetail*, which among other includes the status string (*status*), a timestamp (*time*) and the sequential id (*seq-id*).

If a message was sent to a single recipient, the presence of the *first-dest* element is specially convenient in that it spares the client a call to *browseDestinationBySeqId*.

3. Asynchronous notifications

Users can request that all status updates for a given message be asynchronously notified to their own HTTP service exposing an appropriate interface.

User-side callback service must implement the SOAP operations described in Chaka_CB.wsdl, available at http://soap.kqumg.it/chaka/current/0.9.5/Chaka_CB.wsdl, and *must* run on ports 80 or 8080, or on a port in the range 40000-50000.

Chaka_CB service (with "CB" standing for *call-back*) currently includes two operations:

ping	called by the gateway during the session activation
processStatusNotifications	invoked when new status details are available to be notified

The user-side service's endpoint URI is passed to the gateway as the *soap-callback* option of the *start* operation. The server will try to invoke the *ping* operation before returning. If the process completes successfully, all subsequent injected messages will be bound to the specified callback endpoint URI.

Both *ping* and *processStatusNotifications* operations are *one-way* methods. Their implementation must return an HTTP response with an empty body and a status code indicating success (200-299). Failure to successfully call the *ping* operation on the endpoint will result in the session not being activated and an error being returned. Failure to call the *processStatusNotifications* operation will result in the pending notifications being deferred and the call being retried at a later time.

Note that users do not actually need to set up a full-fledged SOAP service to take advantage of this feature. A simple web server and an XML parser will do, since no SOAP response needs to be generated.

4. Errors

When an error occurs, a *SOAP-ENV:Fault* response is returned. Here is a example:

```
<SOAP-ENV:Fault>
    <faultcode>SOAP-ENV:Client</faultcode>
    <faultstring>403 AUTH_BAD_CREDENTIALS</faultstring>
    <detail>Invalid or expired 'auth' token</detail>
  </SOAP-ENV:Fault>
```

The *faultstring* element includes an HTTP-like numerical code followed by an uppercase string code. The *detail* element contains an English description of the error.

5. Troubleshooting

Trouble reports and help requests must be directed to <soap-support@kqumg.it>. When reporting problems or unexpected errors/behaviors, you should attach a full log (including HTTP headers and XML body) of all relevant SOAP messages sent to and received from the server.

Interoperability tests have been carried out using clients developed in the following languages or environments:

- Perl + SOAP::Simple
- Python + NuSOAP/0.7.2
- PHP-SOAP/5.2.2
- 4D built-in SOAP Client
- Axis/1.4
- Java/1.5.0_06
- C#/.NET

No serious compatibility issues have arisen so far.

If your environment/language is not listed above and you indeed managed to successfully use the service, please, let us know. If you also provide us with some code samples, we might include them in a future release of this document as an additional help, along with a reference to your name, company or domain.

6. XML snippets

The following samples are for demostration pourpose only. They are not intended to be used as surrogate documentation and should not be relied on when writing client code. Samples are not even complete and all details that are not called for by the WSDL (such as the namespace prefixes *m1* and *m2*) may change at any time.

Here is a *start* request sample (*SOAP-ENV:Body* part) and a possible response:

```
<SOAP-ENV:Body xmlns:m1="urn:chaka-0.9.5" >
  <m1:start>
   <m1:userid>user@example.com</m1:userid>
   <m1:password>password</m1:password>
   <m1:options>
    <m1:contact-person>mailto:develop-staff@example.com</m1:contact-person>
    <m1:soap-callback>http://soap.example.com/soap-kq-callback</m1:soap-callback>
   </m1:options>
  </m1:start>

<SOAP-ENV:Body xmlns="urn:chaka-0.9.5">
  <m1:startResponse>
    <m1:auth>4uY29tbTIwMDAuaXQvY2dpLWJpbi9zb2FwY2Jsb2cucGwKCSA=</m1:auth>
  </m1:startResponse>
</SOAP-ENV:Body>
```

Below is a *storeDocuments* request sample (*SOAP-ENV:Body part*) with a possible response:

```
<SOAP-ENV:Body SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
 <storeDocuments xmlns="urn:chaka-0.9.5" >
 <auth> [...]  </auth>
 <sources>
 <source>
   <remote-document>
   <location>http://www.example.com/documents/myfax.pdf</location>
   <unique-name>myfax.pdf</unique-name>
   </remote-document>
 </source>
 <source>
   <document>   <unique-name>myfax.tif</unique-name>
   <data>
    SUkqABQSAQAAKJsVAGWzAqBsVgCUzQqAslkBUDYrAMpmBUDZrAAomxUAZbMCoGxWAJTNCoCy
    WQFQNisAymYFQNmsACibFQBlswKgbFYAlM0KgLJZAVA2KwDKZgVA2awAKJsVAGWzAqBsVgCU
    ICAwNjIzMzI3Nzc3ICBGcmkgTm92IDE5IDE2OjU0OjE3IDIwMDQKAAAAgBkAACAAAACAGAAA
    IAA=
  </data>
   </document>
   </source>
  </sources>
  </storeDocuments>
 </SOAP-ENV:Body>
```

```
<SOAP-ENV:Body xmlns:m1="urn:chaka-0.9.5">
  <m1:storeDocumentsResponse>
   <m1:stored-list>
    <m1:stored>
     <m1:name>myfax.pdf</m1:name>
     <m1:md5>4cf791b5c9d0b1e4627b0130a05a9a69</m1:md5>
     <m1:size>98891</m1:size>
     </m1:stored>
    <m1:stored>
     <m1:name>myfax.tif</m1:name>
     <m1:md5>375992b51f242f46ae284c72298bd928</m1:md5>
     <m1:size>70472</m1:size>
    </m1:stored>
   </m1:stored-list>
  </m1:storeDocumentsResponse>
</SOAP-ENV:Body>
```

Below is a *sendMessage* request sample (*SOAP-ENV:Body* part) for injecting a Fax message, followed by a possible response:

```
<SOAP-ENV:Body xmlns="urn:chaka-0.9.5" >
 <sendMessage>
  <auth>  [ auth string ] </auth>
  <request-uid>01234567898901234567890</request-uid>
   <message >
    <subject>Test message (via SOAP)</subject>
    <fax>
     <destinations>
      <item><address>0270030070</address><cc1>code1A</cc1><cc2>code1B</cc2> </item>
     </destinations>
     <documents>
      <item><document-ref><unique-name>myfax.pdf</unique-name></document-ref></item>
    </document>
  </fax>
 </message>
</sendMessage>

<SOAP-ENV:Body xmlns:m1="urn:chaka-0.9.5">
 <m1:sendMessageResponse>
      <m1:result>
       <m1:request-uid>01234567898901234567890</m1:request-uid>
       <m1:cached-result>false</m1:cached-result>
       <m1:info> <m1:msg-id>13283256</m1:msg-id>  </m1:info>
      </m1:result>
 </m1:sendMessageResponse>
</SOAP-ENV:Body>
```

Below is a *storeDestinationList* sample request for uploading a CSV destination table:

```
<SOAP-ENV:Body  xmlns:m1="urn:chaka-0.9.5">
  <m1:storeDestinationList>
  <m1:auth>  [ auth string ] </m1:auth>
   <m1:list>
    <m1:unique-name>phone-list-1</m1:unique-name>
    <m1:table>
       <m1:address-column>phone</m1:address-column>
       <m1:cc1-column>name</m1:cc1-column>
       <m1:cc2-column>company</m1:cc2-column>
       <m1:columns>
          name company address1 address2 city zip state fax phone email
       </m1:columns>
       <m1:format>semicolon-csv</m1:format>
       <m1:data>
          <!--  base64  data -->
       </m1:data>
     </m1:table>
   </m1:list>
  </m1:storeDestinationList>
```

Example of a call to *processStatusNotifications* (*SOAP-ENV:Body* part)*:*

```
<SOAP-ENV:Body xmlns:m2="urn:chaka-callback-0.9.5" xmlns:m1="urn:chaka-0.9.5" >
     <m2:processStatusNotifications>
        <m2:notif-list>
            <m2:notif>
             <m2:msg-id>13283256</m2:msg-id>
             <m2:request-uid>01234567898901234567 89</m2:request-uid>
             <m2:dest-count>992</m2:dest-count>
             <m2:completed-dest-count>992</m2:completed-dest-count>
             <m2:subject>Test message (via SOAP)</m2:subject>
             <m2:dest>
                <m1:seq-id>1</m1:seq-id>
                <m1:address>0270030070</m1:address>
                <m1:cc1>code1A</m1:cc1>
                <m1:cc2>code1B</m1:cc2>
                <m1:status>OK</m1:status>
                <m1:time>2006-11-20T10:24:40Z</m1:time>
                <m1:pages>1</m1:pages>
              </m1:dest>
           <m2:notif>
        <m2:noitif-list>
     <m2:processStatusNotifications>
```

5. Notes

The codename of KQUMSG1 is *chaka*, which means "bridge" in the Quechua language.