Collaborative Project

EU-BRIDGE

Bridges Across the Language Divide

Grant Number 287658 – ICT-2011.4.2 Language Technologies
Start date: 01 FEB 2012 - duration: 36 months

D4.1: Speech Processing Service API and specifications delivered to the Scientific Board

Version: 2.1 Date: 10.07.2012

Type of activity	RTD
Work package(s)	WP 4
Due date	31.05.2012
Submission date	10.07.2012
Main author(s)	Christian Fügen
Co-author(s)	
Reviewers	Volker Steinbiss, Roma Wiezorek
Contributors	
Version (status, date)	V2.1 (Final, 10.07.2012)
Dissemination level	RE = Restricted to a group specified by the consortium (incl. the Commission Services)
Nature	R
Keywords	WP4, service architecture, client server, speech translation

Part of the Seventh Framework Programme
Funded by the EC-DG INFSO

Executive Summary (Abstract)

This deliverable will report on the analysis of D1.1, D2.1, D3.1 and D5.1. It describes the APIs and the specifications (functional and operational, the associated plan and timetable, and the corresponding evaluation criteria, protocols and metrics) of the system to be deployed for accessing the transcription and translation services from outside and thus implementing the four use cases. It will be delivered to the Scientific Board for approval as a Milestone 401.

Note: The project officer has amended the original plan for this deliverable. The deliverable has been split into Part 1 (draft, due date 31.05.2012) and Part 2 (final, due date 30.06.2012).

Contents

Chapter 1

The Service Infrastructure

1.1 Overview

Figure 1.1 shows the placement of the service infrastructure within the project and its connection with the different work packages. As it can be seen, all use cases are connected to the service infrastructure. On the other side, the service infrastructure makes use of the different core components such as ASR, MT,

The service infrastructure is implemented as a client-server architecture. Several clients can connect to a mediator which distributes the requests and load amongst several connected workers. In our scenario, clients are the modules that implement the use cases and the workers represent different core components for ASR, MT, TTS, LID, Segmentation, Annotation, and Presentation. An overview of this service infrastructure is given in Figure 1.2.

Through its light-weighted API, the service infrastructure allows the integration of multiple engines from different partners and a flexible usage for different use cases. The mediator supports the parallel processing of service requests in a distributed architecture.

1.2 Terminology

In this section we would like to introduce the terminology that will be used throughout this deliverable.

1.2.1 Clients and Workers

The service architecture enables a connection-based communication with multiple service requests at the same time. A client connects to the mediator and the mediator connects the output media stream of the client with one or multiple workers in order to accomplish a specific service request.

The clients typically initiate the service request by specifying the type and language of the media stream that should be processed and by specifying the type and language to which the media stream should be converted. For example, for the captioning use case, an audio stream is sent to the mediator as an output media stream and the client requests automatically generated captions of the audio as an input stream.

In case of a worker, each worker has to register at the service architecture with one or multiple services that the worker is able to handle. But each worker accepts only one incoming service request per connection. In order to make sure that even in the case when the worker is unable to handle the incoming media stream in real-time, the stream is queued internally for later

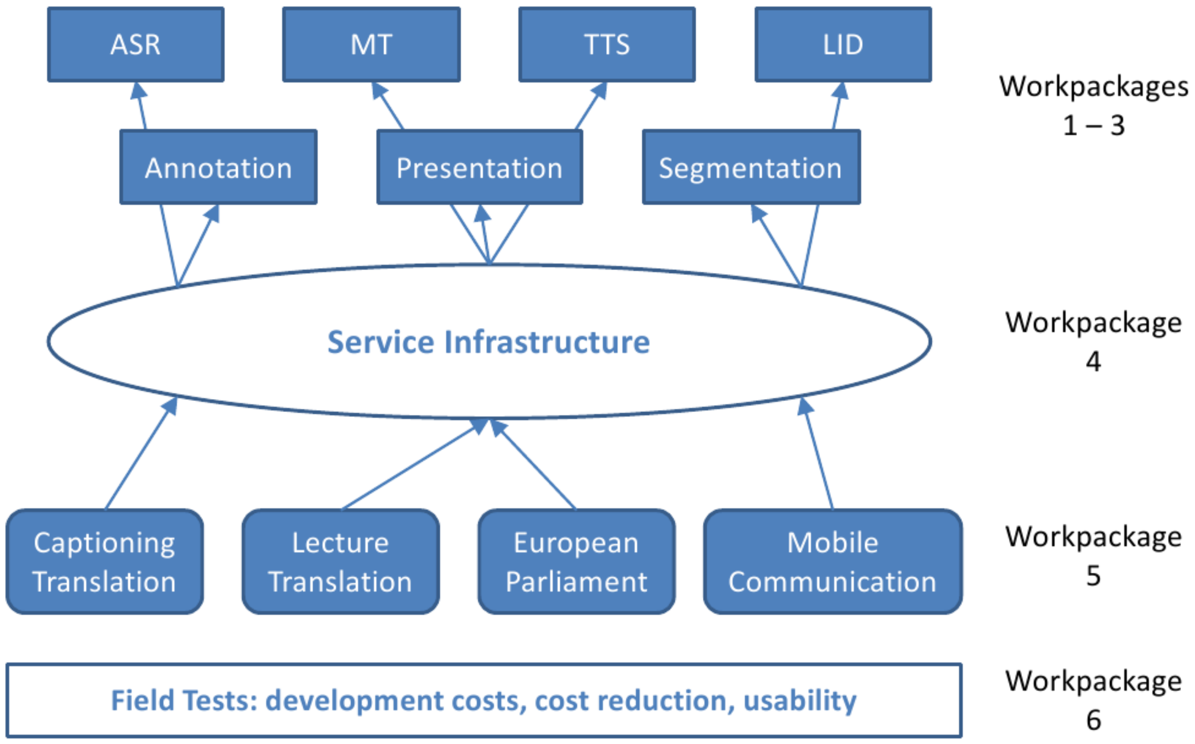

Figure 1.1: Placement of the service architecture within the project

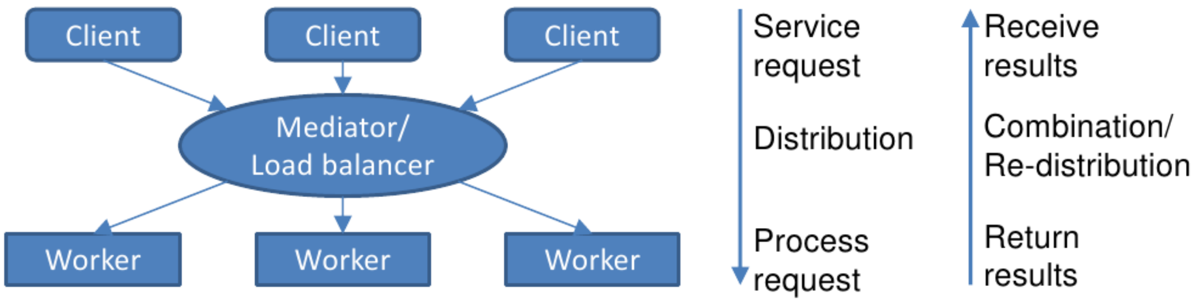

Figure 1.2: Overview of the service architecture

`ll`	language code (ISO369-1, 2 lowercase letters)
`LL`	optional country code (ISO3166, 2 uppercase letters)
`dddd`	optional additional properties (n letters)

Table 1.1: Structure of a fingerprint.

processing.

1.2.2 Fingerprints

Fingerprints are used to specify the exact language and genre of a media stream. They consist of a two-letter language code (ISO369-1) followed by an optional two-letter country code (ISO3166) and an optional additional string specifying other properties such as domain, type, version, or dialect. Different parts have to be separated by a minus sign (`ll[-LL[-dddd]]`). For example, `en-EU` means European accented English, `de-AT` denotes German that is spoken in Austria, and `en-US-weather` is the fingerprint of a media stream for the US American accented English in the weather-domain.

If a language is spoken across several countries, the country code shall be the uppercase language code. Table 1.1 summarizes the structure of a fingerprint.

1.2.3 Input and Output Types

Stream types are used to specify the type of a media stream. For a fully specified media stream, both the fingerprint and the stream type have to be given. So far, the following stream types are currently supported by the service architecture:

`audio`: audio containing speech

`image`: image data such as presentation slides

`text`: properly formatted textual data, e.g. captions

`unseg-text`: unsegmented textual data such as ASR hypotheses

Media streams coming from the mediator to the client or worker are classified as input streams, and media streams passing from the client or worker to the mediator are referred to as output streams.

1.2.4 Mediation – Simple Form

This section describes the mediation of a client's request. In order to accomplish a client's request, workers must be present that are able to convert a media stream specified by the input fingerprint and input type to the requested output fingerprint and output type. In other words, if a client is sending a media stream of the type `audio` with the fingerprint `en-EU` and requests an input stream of the type `unseg-text` an the fingerprint `en`, the mediator must find one or a concatenation of multiple workers that are able to convert audio containing European accented speech into unsegmented English text, i.e. a speech recognition worker in that case.

In order to make sure that a mediation is still possible even if there are no workers available which exactly match with the requested stream types and fingerprints, back-up strategies have been implemented, which are processed in the following order:

- Stream types must match

- For fingerprints, the match is a score according to the following criteria:
  - Exact match
  - Language code and country
  - Language code and domain
  - Language code

For a `en-EU-weather` fingerprint, workers would be selected in the following order: `en-EU-weather > en-EU > en-US-weather > en-US`. This back-up strategy order is chosen with respect to speech recognition, for which a match of the language and country, i.e. dialect or accent, is more important than a match of the domain.

More complicated forms of mediation are possible by specifying the whole workflow in XML using the `node` structure. Such forms might be necessary for instance for a system combination or when only specific workers should be selected.

1.2.5 Packet and Data Types

Messages between the mediator and clients/ workers are transferred in XML packets of a specific type. The following packet types are currently supported:

`data`: Packets containing data that is exchanged between client and worker. A data packet must be specified in greater detail by the data type:

> **`audio`:** For audio data containing speech
>
> **`text`:** For text data such as the recognition and translation results
>
> **`image`:** For image data such as the slides of a presentation

`done`: Status message that is sent either when the client has sent all data, or the worker has finished processing all data (but only after a `done` from the client has been received)

`error`: Status message that is sent whenever an error occurs such as an interrupted connection

`reset`: Status message that is sent whenever the client or worker should be reset to its initial state

`flush`: Status message that is sent, whenever the client or worker should finalize processing pending data and flush all corresponding results

1.3 Connecting, Requesting, and Announcing

This section explains the connection procedure for clients and workers to the service architecture. It is also describes how clients can request specific services and how workers announce the services they can provide.

1.3.1 Client – Flows and Streams

Figure 1.3 shows the steps that are necessary to be taken by the client for registering with the mediator and announcing the type of the output streams. Each client is allowed to have exactly one flow per connection to the mediator. The flow description provides human readable information about the flow and consists of a name and description of the flow in several different languages. A flow can consist of one or multiple different time synchronous media streams that belong to each other such as audio or image that are specified by its type and fingerprint. In addition, a stream ID is required in order to be able to request translations of a specific stream only, for example (see next section).

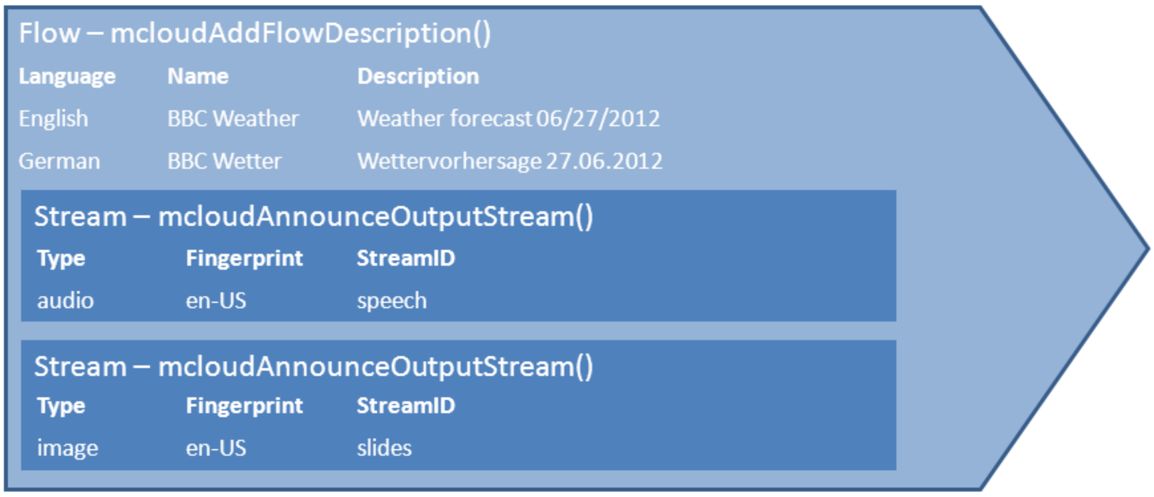
Flow – mcloudAddFlowDescription()
Language
Name
Description
English
BBC Weather
Weather forecast 06/27/2012
German
BBC Wetter
Wettervorhersage 27.06.2012
Stream – mcloudAnnounceOutputStream()
Type
Fingerprint
StreamID
audio
en-US
speech
Stream – mcloudAnnounceOutputStream()
Type
Fingerprint
StreamID
image
en-US
slides

Figure 1.3: Client: Registering a flow and announcing output streams.

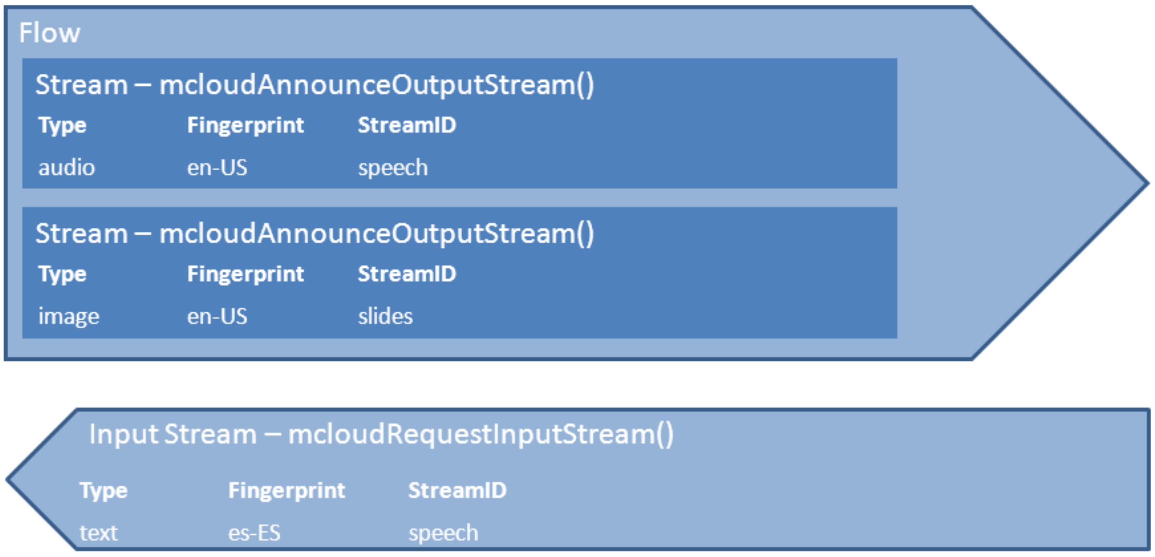

Figure 1.4: Client: Requesting for an input streams.

1.3.2 Client – Request for an Input Streams

Service requests can be raised for each of the output streams announced. A request is specified by a type and fingerprint along with a stream ID that specifies the stream that needs to be converted. A single client is even allowed to request multiple different input streams per stream ID, however, in order to accomplish a request, the required workers must be present. For the fingerprints and types given in Figure 1.4, the following sequence might occur: A worker converting an English US audio into an English US text (an ASR worker) and another worker for converting an English US text into a Spanish text (an MT worker).

1.3.3 Worker – Service Specification

On the other hand, a worker is required to register at the mediator with information about the services that it can provide. Note that a single worker is able to offer multiple services, however, only one service can be active at the same time per connection with the mediator.

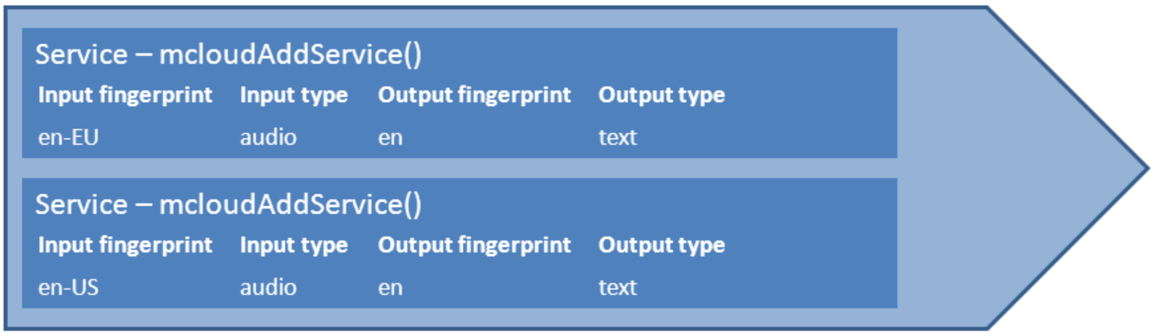

Figure 1.5: Worker: Specification of services that can be provided.

For example, the worker shown in Figure 1.5 offers the conversion of European and American accented English into English text.

1.3.4 Client – Example Program Flow

Figure 1.6 shows a flow chart of a typical client implementation. First, a handle to the MCloud structure is created that keeps all necessary information for the data exchange between the client and mediator. This is followed by adding a flow description to the handle specifying in several different languages the name and description of the flow that is handled by the client. With this information, the client can connect to the mediator that is running on a specific host and port. As soon as the connection is established, the client can announce different output streams that belong to the flow.

If the client also wants to receive some input media stream, type and fingerprint of the input media stream need to be specified calling `mcloudRequestInputStream`. In order to be able to receive and process packets belonging to the input streams while sending packets belonging to the output streams, the reception of packets should be done asynchronously to the main thread within a separate thread.

As soon as this is done, the data can be sent. This is also done asynchronously with the help of an extra thread that is only responsible to send out the packets that are put into the sending queue by calling `mcloudSendPacketAsync`. When no more data is to be sent, `mcloudWaitFinish` needs to be called in order to wait until the sending queue is empty.

If the client has requested an input stream, it needs to wait until the mediator signals by sending a `done` message that all workers involved have finished processing the output media streams. Afterwards, the client can disconnect from the mediator. In order to process a new media stream and service request, a new connection needs to be established.

Figure 1.7 shows the queues and the callback mechanism that has been implemented in order to support the asynchronous sending and processing of packets. For the process of sending messages, a sending queue is available to which packets can be appended for sending. In case of any errors, the callback functions set by `mcloudSetErrorCallback` and `mcloudSetBreakCallback` are called in order to handle the errors within the main thread. If the client has requested an input stream, the reception of packets should also be done in a separate thread, in this case by calling `RecvMessageAsync`. This thread simply wait for the incoming packets and is taking specific actions depending on the message type. In case of a `data` message, the packet is appended to the processing queue. In case of a `done` message, `mcloudWaitFinish` is called in order to wait until the processing of all pending packets in the processing queue has been finished, directly followed by leaving this thread. In case of a `error` or `reset` message, the processing is stopped immediately by calling `mcloudBreak` followed by exiting the thread.

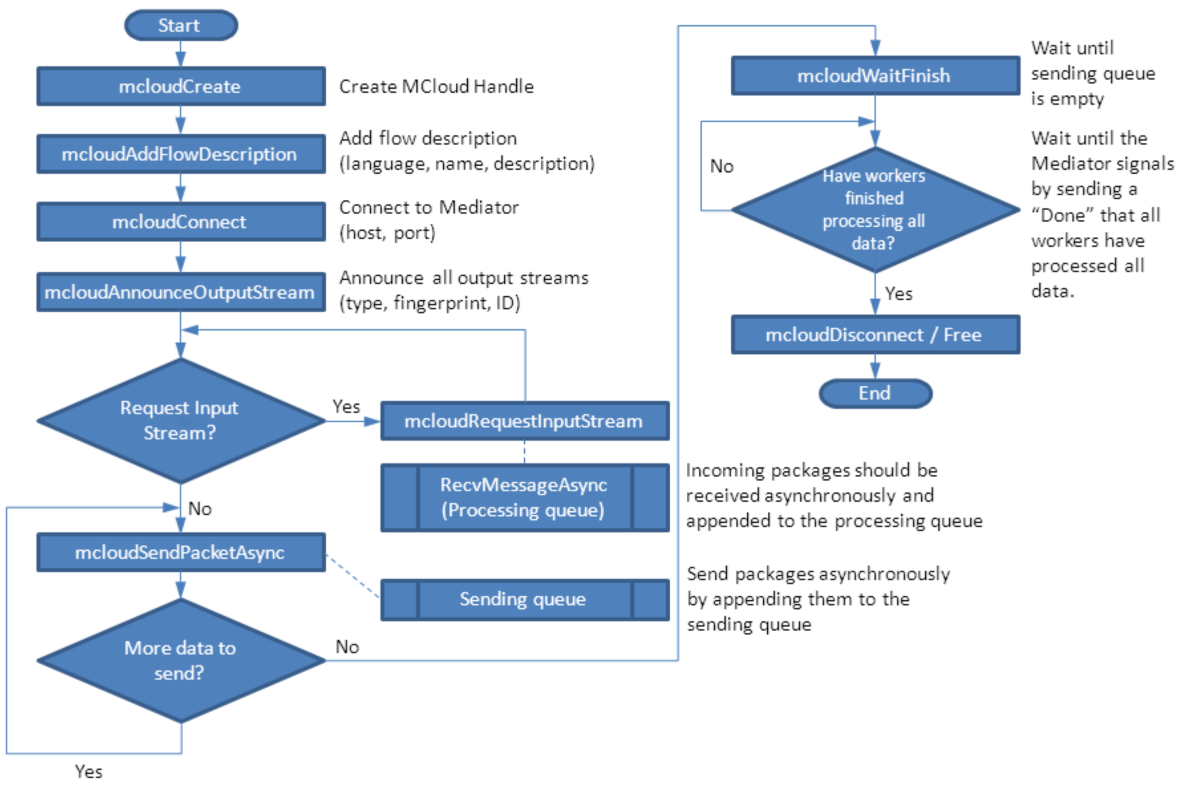
Start
mcloudCreate
Create MCloud Handle
mcloudAddFlowDescription
Add flow description (language, name, description)
mcloudConnect
Connect to Mediator (host, port)
mcloudAnnounceOutputStream
Announce all output streams (type, fingerprint, ID)
Request Input Stream?
Yes
mcloudRequestInputStream
RecvMessageAsync (Processing queue)
Incoming packages should be received asynchronously and appended to the processing queue
No
mcloudSendPacketAsync
Sending queue
Send packages asynchronously by appending them to the sending queue
More data to send?
No
Yes
mcloudWaitFinish
Wait until sending queue is empty
No
Have workers finished processing all data?
Wait until the Mediator signals by sending a "Done" that all workers have processed all data.
Yes
mcloudDisconnect / Free
End

Figure 1.6: Client: Example program flow.

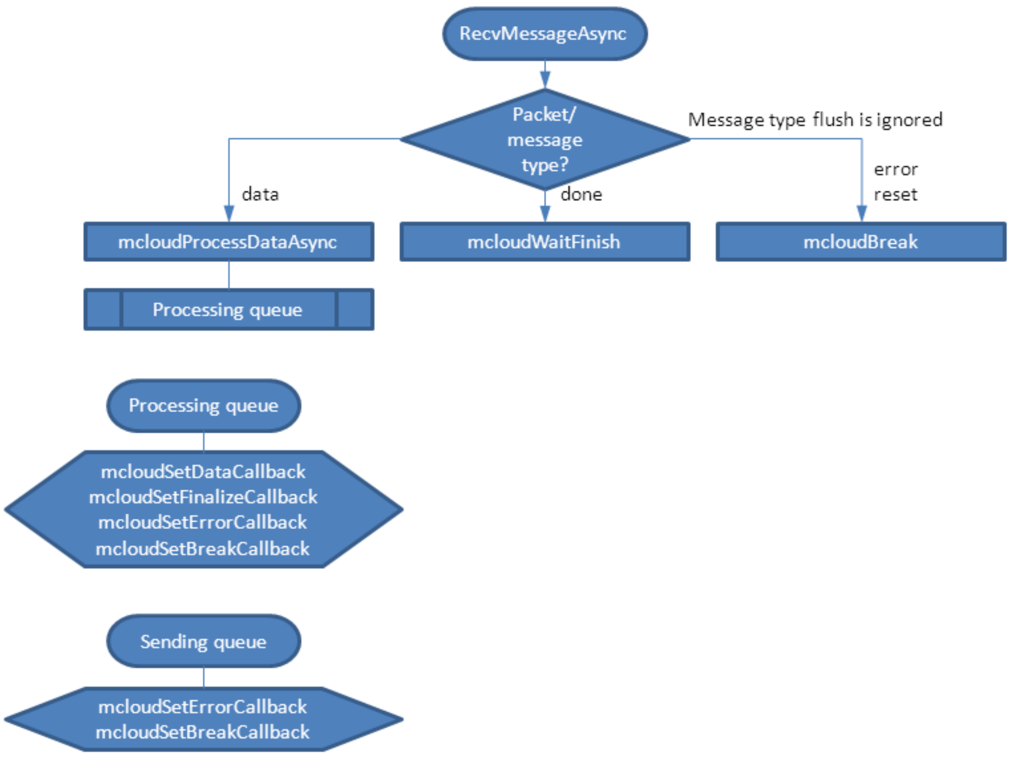

Figure 1.7: Client: Queues and callbacks.

For the processing queue, four different callback functions can be set:

`dataCallback`: This function is called for each packet in the queue sequentially. For the client, the requested input stream results should be handled within this function.

`finalizeCallback`: This function is called when the mediator signals that all workers involved in this service request have finished processing the data. This means, the client can safely disconnect from the mediator.

`errorCallback`: This function is called when an error occurs during the internal processing. This means that the processing should be stopped by resetting all the queues and the client should re-connect to the mediator.

`breakCallback`: This function is called when `mcloudBreak` has been called and means that the processing should be stopped by resetting all the queues and the client should reconnect to the mediator.

The message type `flush` can be ignored by the client, because there is typically no other component that depends on the requested input stream.

1.3.5 Worker – Example Program Flow

In the case of a worker, Figure 1.8 shows a flow chart of a typical worker implementation. First, a handle to the MCloud structure is created that keeps all necessary information for the data exchange between the worker and mediator. This is followed by adding several service descriptions that are offered by the worker. The service descriptions are specified by an input stream type, an input fingerprint, an output stream type and an output fingerprint. With this information, the worker can connect to the mediator that is running on a specific host and port. As soon as the connection is established, the worker is waiting until a new service request is received.

As soon as the new service request has been accepted, `mcloudGetNextPacket` is waiting for the incoming packets from the client's media stream to process, followed by specific actions depending on the message type. In case of a `error` or `reset` message, the processing is stopped immediately by calling `mcloudBreak`. The connection with the client is terminated and the worker is waiting for a new client to connect. In case of a `done` message, `mcloudWaitFinish` is called in order to let the worker wait until all pending packets in the processing queue have been processed, followed by terminating the connection with the client and waiting for a new client to connect. In case of a `data` message, the packets are simply appended to the processing queue and the worker is waiting for the next packet to arrive. In case of a `flush` message, the worker is waiting until all pending packets have been processed, by calling `mcloudWaitFinish` then sends a flush message informing e.g. the next worker to flush its queues as well, and is waiting for the next packet to arrive.

Contrary to the client, the worker always keeps the connection with the mediator.

For the processing queue, four callback functions can be set:

`dataCallback`: This function is called for each packet in the queue sequentially. In case of the worker, the packet content has to be processed in this function and available results have to be sent immediately in order to keep the latency low. The data is sent by appending it to the sending queue.

`finalizeCallback`: This function is called as soon as the mediator signals that the client has finished sending the data and directly after the last packet in the processing queue has been processed.

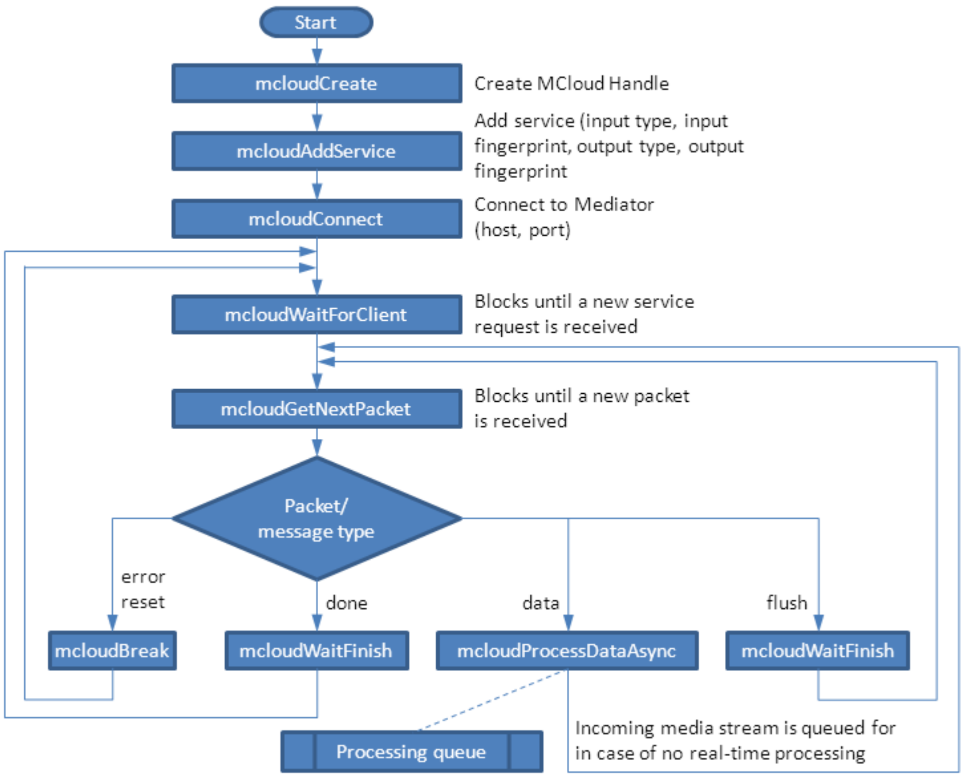
Start
mcloudCreate
Create MCloud Handle
mcloudAddService
Add service (input type, input fingerprint, output type, output fingerprint
mcloudConnect
Connect to Mediator (host, port)
mcloudWaitForClient
Blocks until a new service request is received
mcloudGetNextPacket
Blocks until a new packet is received
Packet/ message type
error reset
done
data
flush
mcloudBreak
mcloudWaitFinish
mcloudProcessDataAsync
mcloudWaitFinish
Processing queue
Incoming media stream is queued for in case of no real-time processing

Figure 1.8: Worker: Example program flow.

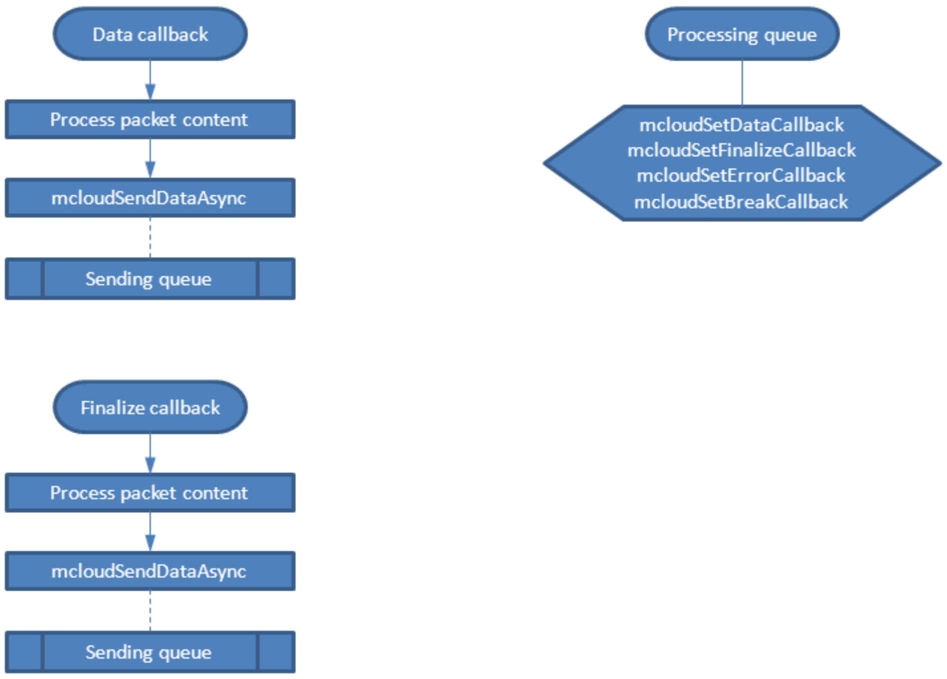

Figure 1.9: Worker: Queues and callbacks.

errorCallback: This function is called whenever an error occurs during the internal processing. This means that the processing should be stopped by resetting all the queues and the worker should wait for new incoming service requests.

breakCallback: This function is called as soon as `mcloudBreak` has been called and means that the processing should be stopped by resetting all the queues and the worker should wait for new incoming service requests.

1.4 Implementation Details

Besides the support of asynchronous processing of the sending and retrieval of packets, the service architecture API provides some more features that should simplify the implementation of other workers and clients.

1.4.1 Convenience Functions

First of all, several convenience functions exist that simplify the handling of data packets, i.e. the extraction of content from a packet and the preparation of a packet for sending. A packet is defined amongst others by a packet type, data type, stream ID, fingerprint, start and stop time stamps. The exact definition can be found in 4.1.

In order to prepare a packet for sending, the following functions can be used:

```
mcloudPacketInintFromAudio      (startTime, stopTime, fingerPrint, sampleA,
                                 sampleN)
mcloudPacketInintFromImage      (startTime, stopTime, fingerPrint, width,
                                 height, format, buffer, byteN)
mcloudPacketInintFromText       (startTime, stopTime, fingerPrint, text)
mcloudPacketInintFromWordTokenA (startTime, stopTime, fingerPrint, tokenA,
                                 tokenN)
```

In case of the audio, a pointer to an array of samples and the number of samples in the array has to be given. Sampling rate and sample size have to be defined by using `mcloudSetAttr` in advance. However, the sampling rate that is used for the media stream should not be lower than the sampling rate that is used by a worker, such as ASR for processing the audio. For images, the pointer and length of a buffer containing the image data along with the specification of the image (width, height, and format) have to be given. And for a text, simply a text has to be specified. Somewhat special is the function accepting word token arrays. This function is especially suitable for the ASR and MT results and will be described in detail in the next section. Note, that a packet generated with this function is also a data packet of a type text. There is even a text field present which is generated automatically out of the information stored in the token array. However, in addition to this text field, also a word token array structure is stored as well.

For extracting the content of a packet, similar functions exist:

```
mcloudPacketGetAudio
mcloudPacketGetImage
mcloudPacketGetText
mcloudPacketGetWordTokenA
```

It is also possible to directly access the packet's content in XML if the packet's content cannot be accessed with the help of the convenience functions.

As mentioned above, the API also allows to change some attributes that are used internally for sampling rate conversion and audio encoding, decoding and compression. So far, the following attributes are available:

`MCloudA_sAudioCodec`: The type of the audio codec for compression. The following codecs are available `pcm` for raw data, and `speex` for the Speex codec.

`MCloudA_iSampleRate`: In case of the client, the sampling rate in Hertz of the output media stream, and in case of the worker, the sampling rate that is required for processing the input media stream.

`MCloudA_iSampleSize`: As above, but for the sample size in bits.

`MCloudA_iChannelN`: As above, but for the number of chanels in the audio.

It is expected that more attributes appear in the future.

1.4.2 Word Tokens

In order to be able to pass some additional information from an ASR worker for example, an array of word tokens has been introduced. Amongst others, for each token, the internal form, written form, spoken form, a confidence, and some start and stop time stamps are specified. From these attributes, only the written form, i.e. the word and the start and stop time are required. The internal representation of the word and a spoken form of the word are optional. An additional attribute is available for marking tokens as filler words and not as regular words. The exact definition can be found in 4.2.

As mentioned above, a text string is automatically generated from each word token array. For this purpose, the written forms are simply concatenated.

1.4.3 Time Stamps

In order to be able to align the media streams produced by the workers with the media streams provided by the clients, time stamps are used in the headers of each packet as well as in the word token arrays.

The time stamps in the packet headers are absolute time stamps within the media stream. In case of the a client, each outgoing packet of an output media stream needs to be equipped with a time stamp, specifying the absolute time when the packages have been sent, and covering the period of time of the packet's content. It is required that a media stream should never be interrupted. The time stamps should be absolutely continuous. This means that the difference between the absolute time stamp of a packet and the absolute time stamp at the beginning of the media stream gives the position of the packet withing the media stream.

Contrary to it, the time stamps defined in the word token array are relative times stamps to the starting position of the media stream, i.e. the beginning of the media stream is set to 0.

1.5 Example Implementations

In the following sections, some example implementations for a worker and client are given. Note, that in all examples multi-threading is required in order to be able to receive new data while processing buffered data, because e.g. a worker cannot always guarantee that an incoming package has already been processed before the next package arrives.

The API is provided in a binary form as a C-library with header files along with some documentation and C source code examples which demonstrate its usage.

1.5.1 Schematic ASR Worker Implementation

```
mcloudCreate ()
mcloudAddService (en-GB, audio, en-GB, text)
mcloudSetDataCallback (userData) // set other callbacks if required

while (1) {
  mcloudConnect ()
  mcloudWaitForClient ()
  while (1) {
    proceed = 1
    while (proceed && p=mcloudGetNextPacket ()) {
      switch (p->type) {
        data:  mcloudProcessDataAsync (p); break
        flush: mcloudWaitFinish (); mcloudFlush (); break
        done:  mcloudWaitFinish (); proceed = 0; break
        error:
        reset: mcloudBreak (); proceed = 0; break
      }
    }
  }
}

// data callback function
dataCallback (p, userData) {
  switch (p->type) {
    audio: mcloudPacketGetAudio (p, &sampleA, &sampleN)
           asr_Decode
           asr_GetResult
           // copy result into token array
           p = mcloudPacketInitFromWordTokenA (startTime, stopTime,
                                               tokenA, tokenN);
           mcloudSendPacketAsync (p);
           break
  }
}
```

1.5.2 Schematic Client Implementation

```
mcloudCreate ()
mcloudAddFlowDescription (English, Weather, \Weather Forecast")
mcloudConnect ()
mcloudAnnounceOutputStream (audio, en-GB, speech)
userData->proceed = 1
mcloudSetDataCallback (userData) // set other callbacks if required
if (wantInputStream) {
  mcloudRequestInputStream (text, en-GB, speech)
  recvMessagesAsync (userData); // install thread for receiving data
}
while (mediaDataIsAvailable) {
  p = mcloudPacketInitFromAudio (startTime, stopTime, fingerPrint,
                                 buffer, bufferSize, isFinal)
  mcloudSendPacketAsync (p)
}
mcloudWaitFinish ()
while (userData->proceed) sleep (500)
mcloudDisconnect ()

// receiving messages asynchronously
recvMessagesAsyncMain (userData) {
  p = userData->packet
  while (userData->proceed && p=mcloudGetNextPacket ()) {
      switch (p->type) {
        data:  mcloudProcessDataAsync (p); break
        flush: break
        done:  mcloudWaitFinish (); userData->proceed = 0; break
        error:
        reset: mcloudBreak (); userData->proceed = 0; break
      }
    }
  }
}

dataCallback (p, userData) {
  mcloudPacketGetWordTokenA (p, &tokenA, &tokenN)
  // so something with packet content such as storing the transcript
}
```

1.6 XML Low Level API

In this section, the XML low level API is described. Note, that it is not recommended to use this kind of API. Instead, the API described above should be used, because it already takes care of the XML low level message flow and also implements asynchronous processing for receiving and sending the packets.

1.6.1 XML Low Level Message Flow

Figure 1.10 shows the message flow in the service architecture.

Workers can connect to the mediator by announcing their service. A service is described by its input type, input fingerprint, output type and output fingerprint. In other words, it accepts input streams of the given input type and input fingerprint only, and returns an output stream of the given type and fingerprint. Amongst others, a fingerprint is defined by the language of the stream and the type can be a text or audio, for example. Workers can announce multiple services at once, however, only one stream of a specific type has to be handled at a time.

Clients first have to send a flow description, which is defined by a name such as the title of a show, and some additional description. If that has been accepted by the mediator, several different output streams can be announced. As above, an output stream is defined by its type and fingerprint.

Clients can also request a specific service such as the transcription and translation of some audio into another language. This can be done by requesting an input stream of an available output stream of a specific type and fingerprint. For example, a client announces an output stream of the type audio with the fingerprint en-US (for US English). By requesting an input stream of the type text with the fingerprint es-ES (for European Spanish), it requests the mediator to first transcribe the audio into English, then translate the result from English into Spanish, and

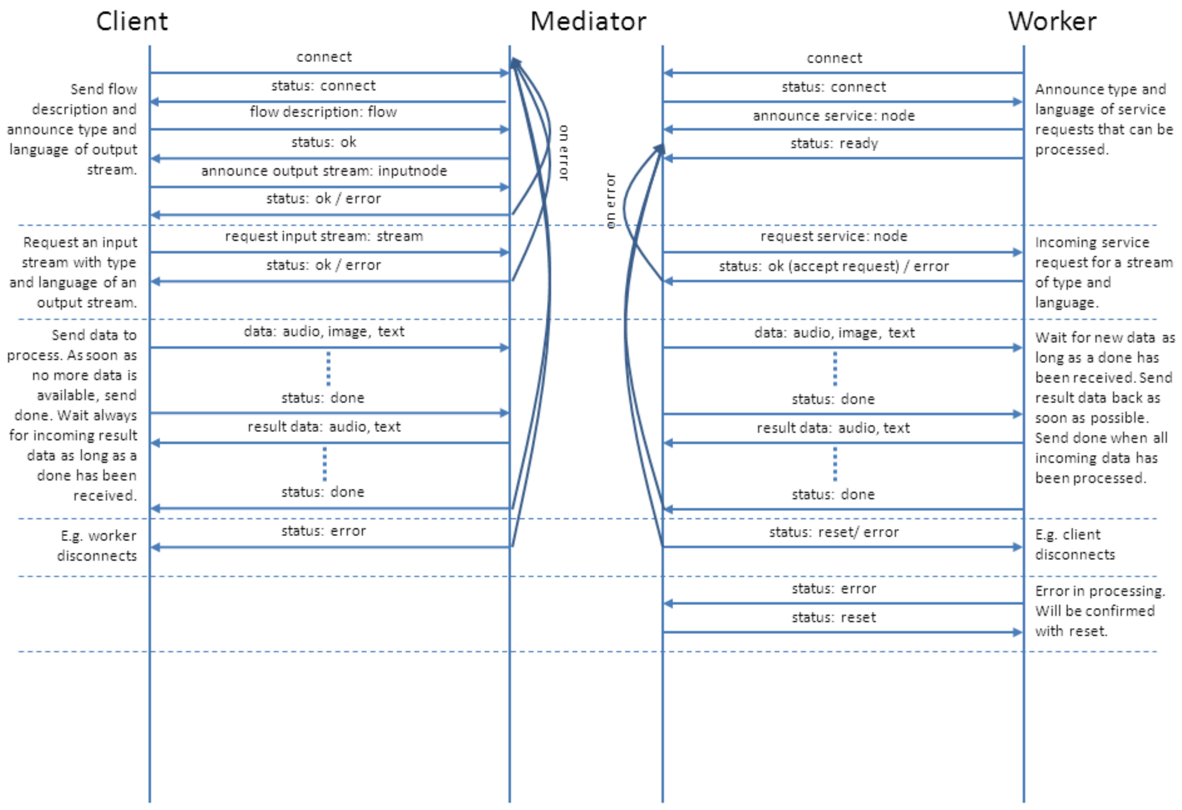
Client
Mediator
Worker
connect
status: connect
flow description: flow
status: ok
announce output stream: inputnode
status: ok / error
Send flow description and announce type and language of output stream.
request input stream: stream
status: ok / error
Request an input stream with type and language of an output stream.
data: audio, image, text
status: done
result data: audio, text
status: done
Send data to process. As soon as no more data is available, send done. Wait always for incoming result data as long as a done has been received.
status: error
E.g. worker disconnects
on error
on error
connect
status: connect
announce service: node
status: ready
Announce type and language of service requests that can be processed.
request service: node
status: ok (accept request) / error
Incoming service request for a stream of type and language.
data: audio, image, text
status: done
result data: audio, text
status: done
Wait for new data as long as a done has been received. Send result data back as soon as possible. Send done when all incoming data has been processed.
status: reset/ error
E.g. client disconnects
status: error
status: reset
Error in processing. Will be confirmed with reset.

Figure 1.10: Message flow within the service architecture

finally return it to the client. The mediator is responsible for selecting the workers necessary to do this job. This is done by sending the selected workers a service request.

As soon as the mediator tells the client that the service request has been accepted, the client can send the data. The data should be sent as a stream divided into several smaller packages. For instance, in the case of audio data, packages containing about 200ms of speech are ideal in order to keep the latency of the recognition results low. As soon as there is no more data available, the mediator has to be informed by sending a Done message.

Ideally, the selected workers are sending their results as soon as possible. They should not wait until the client signals that there is no more data coming. This is especially true for speech recognition. As soon as the worker has been informed that there is no more data coming, all buffered data should be processed and the mediator has to be informed by sending a Done message that there are no more results available.

As soon as both, the client and the worker, have signaled that they have finished processing the request, the connection between the both is terminated. In case of the worker, new service requests can be accepted from that very moment.

As long as the connection between a client and some workers is active, the workers are not able to accept other service requests at the same time. Therefore, multiple workers performing the same service have to be activated when parallel service requests of the same type are to be handled.

1.6.2 XML Schema description

This section defines all XML messages that are allowed to use

```
<!-- XSD for messages -->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<!-- client announces to the mediator the avilability of sessions, i.e. lectures
     currently only one flow is supported -->
  <xsd:element name="flow">
  <xsd:complexType>
    <xsd:all>
      <xsd:element name="sessioninfos" type="sessioninfos" minOccurs="1"
          maxOccurs="1"/>
    </xsd:all>
    <xsd:attribute name="password" type="xsd:string"/>
    <xsd:attribute name="sessionid" type="xsd:string" use="required"/>
  </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="sessioninfos">
    <xsd:sequence>
      <xsd:element name="sessioninfo" minOccurs="1" maxOccurs="unbounded"
          type="sessioninfo"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="sessioninfo">
      <xsd:attribute name="language" type="xsd:string" use="required"/>
      <xsd:attribute name="name" type="xsd:string" use="required"/>
      <xsd:attribute name="description" type="xsd:string" use="required"/>
  </xsd:complexType>

<!-- status messages from and to the mediator -->
<xsd:element name="status">
  <xsd:complexType>
    <xsd:attribute name="type" type="statustype" use="required"/>
    <xsd:attribute name="description" type="xsd:string"/>
    <xsd:anyAttribute/>
  </xsd:complexType>
```

```
</xsd:element>

<xsd:simpleType name="statustype">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="ok"/>
    <xsd:enumeration value="done"/>
    <xsd:enumeration value="ready"/>
    <xsd:enumeration value="error"/>
    <xsd:enumeration value="reset"/>
    <xsd:enumeration value="connect"/>
  </xsd:restriction>
</xsd:simpleType>


<xsd:element name="register">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="streams" type="streams" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
    <xsd:attribute name="type" type="registertype" use="required"/>
    <xsd:attribute name="sessionid" type="xsd:string" use="required"/>
  </xsd:complexType>
</xsd:element>


<xsd:simpleType name="registertype">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="add"/>
    <xsd:enumeration value="delete"/>
  </xsd:restriction>
</xsd:simpleType>


<xsd:element name="streams">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="stream" type="stream" minOccurs="1"
          maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>


<!-- client request the mediator to send him data of this type -->
<xsd:element name="stream">
  <xsd:complexType>
      <xsd:attribute name="type" type="xsd:string" use="required"/>
      <xsd:attribute name="fingerprint" type="xsd:string" use="required"/>
      <xsd:attribute name="streamid" type="xsd:string" use="required"/>
  </xsd:complexType>
</xsd:element>

<!-- client announces to the mediator that it will send data of this type
     this results in a node message, which is send to potential workers in order
         to process the data -->
<xsd:element name="inputnode">
  <xsd:complexType>
      <xsd:attribute name="type" type="xsd:string" use="required"/>
      <xsd:attribute name="fingerprint" type="xsd:string" use="required"/>
      <xsd:attribute name="streamid" type="xsd:string" use="required"/>
      <xsd:anyAttribute/>
  </xsd:complexType>
</xsd:element>

<!-- worker announces to the mediator the availibility of a service of this type
    -->
```

```
<xsd:element name="node">
  <xsd:complexType>
      <xsd:attribute name="name" type="xsd:string"/>
      <xsd:attribute name="sessionid" type="xsd:string"/>
      <xsd:attribute name="service" type="xsd:string"/>
      <xsd:attribute name="streamid" type="xsd:string"/>
      <xsd:attribute name="inputfingerprint" type="xsd:string"/>
      <xsd:attribute name="inputtype" type="xsd:string"/>
      <xsd:attribute name="outputfingerprint" type="xsd:string"/>
      <xsd:attribute name="outputtype" type="xsd:string"/>
      <xsd:anyAttribute/>
  </xsd:complexType>
</xsd:element>

<xsd:element name="edge" type="edge"/>
  <xsd:complexType name="edge">
      <xsd:attribute name="from" type="xsd:string"/>
      <xsd:attribute name="to" type="xsd:string"/>
</xsd:complexType>

<!-- client tells the mediator that the DisplayServer is allowed to output the
    data sent -->
<xsd:element name="announce">
  <xsd:complexType>
    <xsd:anyAttribute/>
  </xsd:complexType>
</xsd:element>

<!-- data messages -->
<xsd:element name="data">
        <xsd:complexType>
                <xsd:attribute name="creator" type="xsd:string" use="required"/>
                <xsd:attribute name="type" type="xsd:string" use="required"/>
                <xsd:attribute name="sessionid" type="xsd:string"
                    use="required"/>
                <xsd:attribute name="fingerprint" type="xsd:string"
                    use="required"/>
                <xsd:attribute name="streamid" type="xsd:string" use="required"/>
                <xsd:attribute name="start" type="xsd:string" use="required"/>
                <xsd:attribute name="stop" type="xsd:string" use="required"/>
                <xsd:anyAttribute/>
                <xsd:all>
                        <xsd:element name="audio" type="audioE" minOccurs="0"
                            maxOccurs="1"/>
                        <xsd:element name="text" type="textE" minOccurs="0"
                            maxOccurs="1"/>
                        <xsd:element name="image" type="imageE" minOccurs="0"
                            maxOccurs="1"/>
                </xsd:all>
        </xsd:complexType>
</xsd:element>

<xsd:complexType name="audioE">
        <xsd:simpleContent>
                <xsd:extension base="xsd:string">
                        <xsd:attribute name="bits" type="xsd:string"
                            use="required"/>
                        <xsd:attribute name="srate" type="xsd:string"
                            use="required"/>
                        <xsd:attribute name="channels" type="xsd:string"
                            use="required"/>
                        <xsd:attribute name="samples" type="xsd:string"
                            use="required"/>
                        <xsd:attribute name="codec" type="xsd:string"
                            use="required"/>
                        <xsd:attribute name="encoding" type="xsd:string"
                            use="required"/>
                </xsd:extension>
        </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="imageE">
        <xsd:simpleContent>
                <xsd:extension base="xsd:string">
                        <xsd:attribute name="width" type="xsd:string"
                            use="required"/>
                        <xsd:attribute name="height" type="xsd:string"
                            use="required"/>
                        <xsd:attribute name="format" type="xsd:string"
                            use="required"/>
                        <xsd:attribute name="encoding" type="xsd:string"
                            use="required"/>
                </xsd:extension>
        </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="textE">
        <xsd:simpleContent>
                <xsd:extension base="xsd:string">
                        <xsd:attribute name="encoding" type="xsd:string"
                            use="required"/>
                </xsd:extension>
        </xsd:simpleContent>
</xsd:complexType>

</xsd:schema>
```

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 MCloudPacket_S Struct Reference

`#include <MCloud.h>`

Data Fields

- **MCloudType packetType**
  Type of the packet.
- **MCloudType dataType**
  Type of the data included if packet is of type MCloudData.
- char ∗ **sessionID**
  The current sessionID which the packet belongs to.
- char ∗ **streamID**
  The current streamID which the packet belongs to.
- char ∗ **fingerPrint**
  Fingerprint of the packet.
- char ∗ **creator**
  Name of the creator of the packet.
- char ∗ **start**
  Human readable time stamp identifying the start time of the packet "dd/MM/YY-hh:mm:ss.-mss".
- char ∗ **stop**
  Human readable time stamp identifying the end time of the packet "dd/MM/YY-hh:mm:ss.-mss".
- char ∗ **statusDescription**
  Optional detailed status description in case of status messages.
- char ∗ **xmlString**
  Raw XML string.
- xmlDoc ∗ **doc**
  Reference to the whole XML document (libXML2 xmlDoc)

4.1.1 Field Documentation

4.1.1.1 char∗ MCloudPacket_S::creator

Name of the creator of the packet.

4.1.1.2 MCloudType MCloudPacket_S::dataType

Type of the data included if packet is of type MCloudData.

4.1.1.3 xmlDoc∗ MCloudPacket_S::doc

Reference to the whole XML document (libXML2 xmlDoc)

4.1.1.4 char∗ MCloudPacket_S::fingerPrint

Fingerprint of the packet.

4.1.1.5 MCloudType MCloudPacket_S::packetType

Type of the packet.

4.1.1.6 char∗ MCloudPacket_S::sessionID

The current sessionID which the packet belongs to.

4.1.1.7 char∗ MCloudPacket_S::start

Human readable time stamp identifying the start time of the packet "dd/MM/YY-hh:mm:ss.-mss".

4.1.1.8 char∗ MCloudPacket_S::statusDescription

Optional detailed status description in case of status messages.

4.1.1.9 char∗ MCloudPacket_S::stop

Human readable time stamp identifying the end time of the packet "dd/MM/YY-hh:mm:ss.-mss".

4.1.1.10 char∗ MCloudPacket_S::streamID

The current streamID which the packet belongs to.

4.1.1.11 char∗ MCloudPacket_S::xmlString

Raw XML string.

The documentation for this struct was generated from the following file:

- **MCloud.h**

4.2 MCloudWordToken_S Struct Reference

`#include <MCloud.h>`

Data Fields

- int **index**
- char ∗ **internal**
- char ∗ **written**
- char ∗ **spoken**
- float **confidence**
- unsigned int **startTime**
- unsigned int **stopTime**
- int **isFiller**

4.2.1 Field Documentation

4.2.1.1 float MCloudWordToken_S::confidence

4.2.1.2 int MCloudWordToken_S::index

4.2.1.3 char∗ MCloudWordToken_S::internal

4.2.1.4 int MCloudWordToken_S::isFiller

4.2.1.5 char∗ MCloudWordToken_S::spoken

4.2.1.6 unsigned int MCloudWordToken_S::startTime

4.2.1.7 unsigned int MCloudWordToken_S::stopTime

4.2.1.8 char∗ MCloudWordToken_S::written

The documentation for this struct was generated from the following file:

- **MCloud.h**

Chapter 5

File Documentation

5.1 MCloud.h File Reference

```
#include <libxml/tree.h>
```

Data Structures

- struct **MCloudPacket_S**
- struct **MCloudWordToken_S**

Defines

- #define **SHAREDDLL**
- #define **mcloudERROR**(...) **mcloudMsgHandler**(__FILE__, __LINE__, 1, __VA_ARGS__)
- #define **mcloudWARN**(...) **mcloudMsgHandler**(__FILE__, __LINE__, 2, __VA_ARGS__)
- #define **mcloudINFO**(...) **mcloudMsgHandler**(__FILE__, __LINE__, 3, __VA_ARGS__)

Typedefs

- typedef struct **MCloudPacket_S MCloudPacket**
- typedef struct **MCloudWordToken_S MCloudWordToken**
- typedef struct MCloud_S **MCloud**

  MCloud object.
- typedef int **MCloudCallbackFct** (**MCloud** ∗cloudP, void ∗userData)

  MCloud general callback function type used for finalize, break, and error.
- typedef int **MCloudPacketCallbackFct** (**MCloud** ∗cloudP, **MCloudPacket** ∗p, void ∗userData)

  MCloud packet callback function type used for data and init.

Enumerations

- enum **S2S_Result** { **S2S_Success** = 0, **S2S_Error** = 1 }

- enum **MCloudType** { **MCloudModeWorker** = 1, **MCloudModeClient**, **MCloud-Data**, **MCloudDone**, **MCloudError**, **MCloudReset**, **MCloudFlush**, **MCloud-Audio**, **MCloudText**, **MCloudImage**, **MCloudSendingQueue**, **MCloudProcessing-Queue** }
- enum **MCloudAttribute** { **MCloudA_sAudioCodec**, **MCloudA_iSampleRate**, **M-CloudA_iSampleSize**, **MCloudA_iChannelN** }

Functions

- **SHAREDDLL** void **mcloudMsgHandler** (const char ∗file, int line, int type, const char ∗format,...)

  Message handler that should be only used with the macros defined above.
- **SHAREDDLL MCloudWordToken** ∗ **mcloudWordTokenArrayCreate** (int n)

  Creates an array of MCloudWordTokens.
- **SHAREDDLL** void **mcloudWordTokenArrayFree** (**MCloudWordToken** ∗tokenA, int n)

  Free an array of MCloudWordTokens.
- **SHAREDDLL MCloudPacket** ∗ **mcloudPacketInitFromText** (**MCloud** ∗cloudP, const char ∗startTime, const char ∗stopTime, const char ∗fingerPrint, const char ∗text)

  Initialize a new packet from text for sending.
- **SHAREDDLL MCloudPacket** ∗ **mcloudPacketInitFromWordTokenA** (**MCloud** ∗cloudP, const char ∗startTime, const char ∗stopTime, const char ∗fingerPrint, **MCloud-WordToken** ∗tokenA, int tokenN)

  Initialize a new packet from a MCloudWordToken array for sending.
- **SHAREDDLL MCloudPacket** ∗ **mcloudPacketInitFromAudio** (**MCloud** ∗cloud-P, const char ∗startTime, const char ∗stopTime, const char ∗fingerPrint, const short ∗sampleA, int sampleN, int isFinal)

  Initialize a new packet from audio for sending.
- **MCloudPacket** ∗ **mcloudPacketInitFromImage** (**MCloud** ∗cloudP, const char ∗start-Time, const char ∗stopTime, const char ∗fingerPrint, int width, int height, const char ∗format, const char ∗buffer, int bufferN)

  Initialize a new packet from image for sending.
- **SHAREDDLL** void **mcloudPacketDeinit** (**MCloudPacket** ∗p)

  Free a packet.
- **SHAREDDLL S2S_Result mcloudPacketGetText** (**MCloud** ∗cloudP, **MCloud-Packet** ∗p, char ∗∗text)

  Convenient function for extracting the string embedded in <text></text>.
- **SHAREDDLL S2S_Result mcloudPacketGetWordTokenA** (**MCloud** ∗cloudP, **-MCloudPacket** ∗p, **MCloudWordToken** ∗∗tokenA, int ∗tokenN)

  Convenient function for extracting a word token array embedded in <wordtokens></wordtokens>.
- **SHAREDDLL S2S_Result mcloudPacketGetAudio** (**MCloud** ∗cloudP, **MCloud-Packet** ∗p, short ∗∗sampleA, int ∗sampleN)

  Convenient function for extracting the data embedded in <audio></audio>.
- **SHAREDDLL MCloud** ∗ **mcloudCreate** (const char ∗name, int mode)

  Create an MCloud object with a given name and mode.
- **SHAREDDLL** void **mcloudFree** (**MCloud** ∗cloudP)

  Free an MCloud object.
- **SHAREDDLL S2S_Result mcloudGetAttr** (**MCloud** ∗cloudP, **MCloudAttribute** attr, void ∗value)

Get the value of an attribute.

- **SHAREDDLL S2S_Result mcloudSetAttr** (**MCloud** ∗cloudP, **MCloudAttribute** attr, const void ∗value)

  Set the value of an attribute.

- **SHAREDDLL S2S_Result mcloudConnect** (**MCloud** ∗cloudP, const char ∗host, int port)

  Connect to the MCloud server running on the host at port given.

- **SHAREDDLL S2S_Result mcloudDisconnect** (**MCloud** ∗cloudP)

  Disconnect from the MCloud server.

- **SHAREDDLL S2S_Result mcloudAddService** (**MCloud** ∗cloudP, const char ∗name, const char ∗service, const char ∗inputFingerPrint, const char ∗inputType, const char ∗outputFingerPrint, const char ∗outputType, const char ∗specifier)

  Add a service description of a worker to an MCloud object.

- **SHAREDDLL S2S_Result mcloudAddFlowDescription** (**MCloud** ∗cloudP, const char ∗password, int logging, const char ∗language, const char ∗name, const char ∗description)

  Add a flow description of a client to an MCloud object.

- **SHAREDDLL S2S_Result mcloudAnnounceOutputStream** (**MCloud** ∗cloudP, const char ∗type, const char ∗fingerPrint, const char ∗streamID, const char ∗specifier)

  Announce an output stream of a client to an MCloud object.

- **SHAREDDLL S2S_Result mcloudRequestInputStream** (**MCloud** ∗cloudP, const char ∗type, const char ∗fingerPrint, const char ∗streamID)

  Request an input stream of a client from an MCloud object.

- **SHAREDDLL S2S_Result mcloudRequestForDisplay** (**MCloud** ∗cloudP)

  Request the display of an output stream.

- **SHAREDDLL S2S_Result mcloudWaitForClient** (**MCloud** ∗cloudP, char ∗∗streamID)

  Wait for a service request to process.

- **SHAREDDLL MCloudPacket** ∗ **mcloudGetNextPacket** (**MCloud** ∗cloudP)

  Wait for the next data package.

- **SHAREDDLL S2S_Result mcloudSendPacket** (**MCloud** ∗cloudP, **MCloudPacket** ∗p)

  Send a packet.

- **SHAREDDLL S2S_Result mcloudSendPacketAsync** (**MCloud** ∗cloudP, **MCloudPacket** ∗p, void ∗userData)

  Send a packet asynchronously.

- **SHAREDDLL S2S_Result mcloudSendDone** (**MCloud** ∗cP)

  Inform a client or a worker that there is no more data to receive.

- **SHAREDDLL S2S_Result mcloudSendFlush** (**MCloud** ∗cP)

  Inform subsequent worker to flush their output buffers.

- **SHAREDDLL S2S_Result mcloudProcessDataAsync** (**MCloud** ∗cloudP, **MCloudPacket** ∗p, void ∗userData)

  Process received packages asynchronously.

- **SHAREDDLL** int **mcloudPending** (**MCloud** ∗cloudP, **MCloudType** queueType)

  Return number of pending packages in queue.

- **SHAREDDLL S2S_Result mcloudWaitFinish** (**MCloud** ∗cloudP, **MCloudType** queueType, int done)

  Wait until all pending packages have been processed/ sent.

- **SHAREDDLL S2S_Result mcloudBreak** (**MCloud** ∗cloudP, **MCloudType** queueType)

  Stop processing/ sending pending packages immediately and reset queue.
- **SHAREDDLL** void **mcloudSetInitCallback** (**MCloud** ∗cloudP, **MCloudPacketCallbackFct** ∗callback, void ∗userData)

  Set an init callback function.
- **SHAREDDLL** void **mcloudSetFinalizeCallback** (**MCloud** ∗cloudP, **MCloudCallbackFct** ∗callback, void ∗userData)

  Set a finalize callback function.
- **SHAREDDLL** void **mcloudSetBreakCallback** (**MCloud** ∗cloudP, **MCloudType** queueType, **MCloudCallbackFct** ∗callback, void ∗userData)

  Set a break callback function.
- **SHAREDDLL** void **mcloudSetErrorCallback** (**MCloud** ∗cloudP, **MCloudType** queueType, **MCloudCallbackFct** ∗callback, void ∗userData)

  Set an error callback function.
- **SHAREDDLL** void **mcloudSetDataCallback** (**MCloud** ∗cloudP, **MCloudPacketCallbackFct** ∗callback)

  Set a data callback function.

5.1.1 Define Documentation

5.1.1.1 #define mcloudERROR(...) mcloudMsgHandler(__FILE__, __LINE__, 1, __VA_ARGS__)

5.1.1.2 #define mcloudINFO(...) mcloudMsgHandler(__FILE__, __LINE__, 3, __VA_ARGS__)

5.1.1.3 #define mcloudWARN(...) mcloudMsgHandler(__FILE__, __LINE__, 2, __VA_ARGS__)

5.1.1.4 #define SHAREDDLL

5.1.2 Typedef Documentation

5.1.2.1 typedef struct MCloud_S MCloud

MCloud object.

5.1.2.2 typedef int MCloudCallbackFct(MCloud ∗cloudP, void ∗userData)

MCloud general callback function type used for finalize, break, and error.

Parameters

in	*cloudP*	reference to an MCloud object
in	*userData*	user data

Returns

0 if success

5.1.2.3 typedef struct MCloudPacket_S MCloudPacket

5.1.2.4 typedef int MCloudPacketCallbackFct(MCloud *cloudP, MCloudPacket *p, void *userData)

MCloud packet callback function type used for data and init.

Parameters

`in`	*cloudP*	reference to an MCloud object
`in`	*p*	packet containing the data to process
`in`	*userData*	user data

Returns

0 if success

5.1.2.5 typedef struct MCloudWordToken_S MCloudWordToken

5.1.3 Enumeration Type Documentation

5.1.3.1 enum MCloudAttribute

Enumerator:

MCloudA_sAudioCodec	Audio codec type [pcm, adpcm, speex, fcm] (get/set) (default: pcm)
MCloudA_iSampleRate	input/ output sample rate in Hz (get/set) (default: 16000)
MCloudA_iSampleSize	input/ output sample size in bits (get/set) (default: 16)
MCloudA_iChannelN	input/ output number of channels (get/set) (default: 1)

5.1.3.2 enum MCloudType

Enumerator:

MCloudModeWorker	worker mode
MCloudModeClient	client mode
MCloudData	message of type data
MCloudDone	status message of type done
MCloudError	status message of type error
MCloudReset	status message of type reset
MCloudFlush	status message of type flush
MCloudAudio	data packet of type audio
MCloudText	data packet of type text
MCloudImage	data packet of type image
MCloudSendingQueue	sending queue
MCloudProcessingQueue	receiving/ processing queue

5.1.3.3 enum S2S_Result

Enumerator:

S2S_Success	success
S2S_Error	error

5.1.4 Function Documentation

5.1.4.1 SHAREDDLL S2S_Result mcloudAddFlowDescription (MCloud ∗ *cloudP*, const char ∗ *password*, int *logging*, const char ∗ *language*, const char ∗ *name*, const char ∗ *description*)

Add a flow description of a client to an MCloud object.

This function has to be called after an MCloud object has been created and before connecting to the MCloud. A client can add more than one flow being just translations of the same descriptions. Therefore, the password and logging has to be the same over all flows.

Parameters

in	*cloudP*	reference to an MCloud object
in	*password*	an optional password which has to be used in order to subscribe to this flow (display server), NULL for no password
	in	logging if set to 0, the flow will not be logged in the database
in	*language*	descriptive language identifier of the flow, e.g. English
in	*name*	name of the flow, e.g. title of a talk
in	*description*	additional description of the flow, e.g. abstract

Returns

S2S_Success if no error occurs

5.1.4.2 SHAREDDLL S2S_Result mcloudAddService (MCloud ∗ *cloudP*, const char ∗ *name*, const char ∗ *service*, const char ∗ *inputFingerPrint*, const char ∗ *inputType*, const char ∗ *outputFingerPrint*, const char ∗ *outputType*, const char ∗ *specifier*)

Add a service description of a worker to an MCloud object.

This function has to be called after an MCloud object has been created and before connecting to the MCloud.

Parameters

in	*cloudP*	reference to an MCloud object
in	*name*	name of the worker
in	*service*	name of the service (asr, smt, tts, ...)
in	*inputFinger-Print*	service input finger print
in	*inputType*	data input type (audio, text)
in	*outputFinger-Print*	service output finger print
in	*outputType*	data output type (audio, text)
in	*specifier*	an additional specifier, i.e. a speaker identifier

Returns

S2S_Success if no error occurs

5.1.4.3 SHAREDDLL S2S_Result mcloudAnnounceOutputStream (MCloud ∗ *cloudP*, const char ∗ *type*, const char ∗ *fingerPrint*, const char ∗ *streamID*, const char ∗ *specifier*)

Announce an output stream of a client to an MCloud object.

This function has to be called after the client has been connected to the MCloud.

Parameters

in	*cloudP*	reference to an MCloud object
in	*type*	data type (audio, text, image)
in	*fingerPrint*	finger print of the data stream
in	*streamID*	unique stream identifier
in	*specifier*	an additional specifier, i.e. a speaker identifier

Returns

S2S_Success if no error occurs

5.1.4.4 SHAREDDLL S2S_Result mcloudBreak (MCloud ∗ *cloudP*, MCloudType *queueType*)

Stop processing/ sending pending packages immediately and reset queue.

This function can be used to stop further processing or sending packages in the queue specified.

Parameters

in	*cloudP*	reference to an MCloud object
in	*queueType*	type of queue MCloudSendingQueue, or MCloudProcessingQueue

Returns

S2S_Success if no error occurs

5.1.4.5 SHAREDDLL S2S_Result mcloudConnect (MCloud ∗ *cloudP*, const char ∗ *host*, int *port*)

Connect to the MCloud server running on the host at port given.

This function has to be called after an MCloud object has been created and before waiting for a client or worker.

Parameters

in	*cloudP*	reference to an MCloud object
in	*host*	host name
in	*port*	port number

Returns

S2S_Success if no error occurs

5.1.4.6 SHAREDDLL MCloud∗ mcloudCreate (const char ∗ *name*, int *mode*)

Create an MCloud object with a given name and mode.

Parameters

`in`	*name*	descriptive name of the worker or client (used as 'creator' in XML messages)
`in`	*mode*	working mode, i.e. MCloudModeWorker, or MCloudModeClient

Returns

reference to an MCloud object or NULL if failed

5.1.4.7 SHAREDDLL S2S_Result mcloudDisconnect (MCloud ∗ *cloudP*)

Disconnect from the MCloud server.

Parameters

`in`	*cloudP*	reference to an MCloud object

Returns

S2S_Success if no error occurs

5.1.4.8 SHAREDDLL void mcloudFree (MCloud ∗ *cloudP*)

Free an MCloud object.

Parameters

`in`	*cloudP*	reference to an MCloud object

5.1.4.9 SHAREDDLL S2S_Result mcloudGetAttr (MCloud ∗ *cloudP*, MCloudAttribute *attr*, void ∗ *value*)

Get the value of an attribute.

Parameters

`in`	*cloudP*	reference to an MCloud object
`in`	*attr*	an MCloud attribute
`out`	*value*	reference to the returned value

Returns

S2S_Success if no error occurs

5.1.4.10 SHAREDDLL MCloudPacket∗ mcloudGetNextPacket (MCloud ∗ *cloudP*)

Wait for the next data package.

This function has to be called in order to wait for the next data packet. The function times out after an internally specified amount of time.

Parameters

`in`	*cloudP*	reference to an MCloud object

Returns

S2S_Success if no error occurs

5.1.4.11 SHAREDDLL void mcloudMsgHandler (const char ∗ *file*, int *line*, int *type*, const char ∗ *format*, ...)

Message handler that should be only used with the macros defined above.

Parameters

`in`	*file*	source code file name __FILE__
`in`	*line*	source code line __LINE__
`in`	*type*	message type
`in`	*format*	printf format
`in`	...	variable list of additional arguments

5.1.4.12 SHAREDDLL void mcloudPacketDeinit (MCloudPacket ∗ *p*)

Free a packet.

Parameters

`in`	*p*	reference to an MCloud package

5.1.4.13 SHAREDDLL S2S_Result mcloudPacketGetAudio (MCloud ∗ *cloudP*, MCloudPacket ∗ *p*, short ∗∗ *sampleA*, int ∗ *sampleN*)

Convenient function for extracting the data embedded in <audio></audio>.

Don't forget to free the memory allocated for sampleA.

Parameters

`in`	*cloudP*	reference to an MCloud object
`in`	*p*	reference to an MCloud package
`out`	*sampleA*	returned array of samples
`out`	*sampleN*	returned number of samples in array

Returns

S2S_Success if no error occured

5.1.4.14 SHAREDDLL S2S_Result mcloudPacketGetText (MCloud ∗ *cloudP*, MCloudPacket ∗ *p*, char ∗∗ *text*)

Convenient function for extracting the string embedded in <text></text>.

Don't forget to free the memory allocated for text.

Parameters

in	*cloudP*	reference to an MCloud object
in	*p*	reference to an MCloud package
out	*text*	returned text

Returns

S2S_Success if no error occured

5.1.4.15 SHAREDDLL S2S_Result mcloudPacketGetWordTokenA (MCloud ∗ *cloudP*, MCloudPacket ∗ *p*, MCloudWordToken ∗∗ *tokenA*, int ∗ *tokenN*)

Convenient function for extracting a word token array embedded in <wordtokens></wordtokens>.

Don't forget to free the memory allocated for tokenA.

Parameters

in	*cloudP*	reference to an MCloud object
in	*p*	reference to an MCloud package
out	*tokenA*	returned array of word tokens
out	*tokenN*	number of word tokens returned

Returns

S2S_Success if no error occured

5.1.4.16 SHAREDDLL MCloudPacket∗ mcloudPacketInitFromAudio (MCloud ∗ *cloudP*, const char ∗ *startTime*, const char ∗ *stopTime*, const char ∗ *fingerPrint*, const short ∗ *sampleA*, int *sampleN*, int *isFinal*)

Initialize a new packet from audio for sending.

Parameters

in	*cloudP*	reference to an MCloud object
in	*startTime*	human readable starting time stamp
in	*stopTime*	human readable ending time stamp
in	*fingerPrint*	finger print of the audio
in	*sampleA*	reference to an array of audio samples
in	*sampleN*	number of samples in array
in	*isFinal*	indicates whether the sample array given is the final one

Returns

reference to the created package or NULL if failed

5.1.4.17 MCloudPacket∗ mcloudPacketInitFromImage (MCloud ∗ *cloudP*, const char ∗ *startTime*, const char ∗ *stopTime*, const char ∗ *fingerPrint*, int *width*, int *height*, const char ∗ *format*, const char ∗ *buffer*, int *bufferN*)

Initialize a new packet from image for sending.

Parameters

in	*cloudP*	reference to an MCloud object
in	*startTime*	human readable starting time stamp
in	*stopTime*	human readable ending time stamp
in	*fingerPrint*	language finger print of the image (slide text language)
in	*width*	image width
in	*height*	image height
in	*format*	image format, i.e. PNG
in	*buffer*	reference to a buffer that keeps the image
in	*bufferN*	length of buffer in bytes

Returns

reference to the created package or NULL if failed

5.1.4.18 SHAREDDLL MCloudPacket∗ mcloudPacketInitFromText (MCloud ∗ *cloudP*, const char ∗ *startTime*, const char ∗ *stopTime*, const char ∗ *fingerPrint*, const char ∗ *text*)

Initialize a new packet from text for sending.

Parameters

in	*cloudP*	reference to an MCloud object
in	*startTime*	human readable starting time stamp
in	*stopTime*	human readable ending time stamp
in	*fingerPrint*	finger print of the text
in	*text*	the text string

Returns

reference to the created package or NULL if failed

5.1.4.19 SHAREDDLL MCloudPacket∗ mcloudPacketInitFromWordTokenA (MCloud ∗ *cloudP*, const char ∗ *startTime*, const char ∗ *stopTime*, const char ∗ *fingerPrint*, MCloudWordToken ∗ *tokenA*, int *tokenN*)

Initialize a new packet from a MCloudWordToken array for sending.

Parameters

`in`	*cloudP*	reference to an MCloud object
`in`	*startTime*	human readable starting time stamp
`in`	*stopTime*	human readable ending time stamp
`in`	*fingerPrint*	finger print of the text
`in`	*tokenA*	reference to an MCloudWordToken array
`in`	*tokenN*	number of word tokens in array

Returns

reference to the created package or NULL if failed

5.1.4.20 SHAREDDLL int mcloudPending (MCloud ∗ *cloudP*, MCloudType *queueType*)

Return number of pending packages in queue.

Parameters

`in`	*cloudP*	reference to an MCloud object
`in`	*queueType*	type of queue MCloudSendingQueue, or MCloudProcessingQueue

Returns

number of pending packages

5.1.4.21 SHAREDDLL S2S_Result mcloudProcessDataAsync (MCloud ∗ *cloudP*, MCloudPacket ∗ *p*, void ∗ *userData*)

Process received packages asynchronously.

This function can be used to process packets asnychronously. The packages will be placed into an internal queue and processed in the background by calling mcloudDataCallback. - Callback functions are used to forward status messages such as errors. Use mcloudBreak to stop processing pending packages. Use mcloudWaitFinish to wait until the last package has been processed. Packages are freed automatically after they have been sent. While processing, the data callback function is called for the next pending package. As soon as no more packages are pending and mcloudWaitFinish has been called, the finalize callback function is called. - The error callback function may be called in case of erorrs, and the break callback function if mcloudBreak has been called.

Parameters

`in`	*cloudP*	reference to an MCloud object
`in`	*p*	an MCloud packet
`in`	*userData*	additional user data, that is passed to the callback functions

Returns

S2S_Success if no error occurs

5.1.4.22 SHAREDDLL S2S_Result mcloudRequestForDisplay (MCloud ∗ *cloudP*)

Request the display of an output stream.

By calling this function, the client requests the display of the output stream on the display server. For cancelling the request for display, the client needs to disconnect.

Parameters

`in`	*cloudP*	reference to an MCloud object

Returns

S2S_Success if no error occurs

5.1.4.23 SHAREDDLL S2S_Result mcloudRequestInputStream (MCloud ∗ *cloudP*, const char ∗ *type*, const char ∗ *fingerPrint*, const char ∗ *streamID*)

Request an input stream of a client from an MCloud object.

This function has to be called in order to request a specific input stream from the MCloud such as ASR or MT results. Otherwise, the client will not receive any data. This function has to be called after the client has been connected to the MCloud.

Parameters

`in`	*cloudP*	reference to an MCloud object
`in`	*type*	data type (audio, text, image)
`in`	*fingerPrint*	finger print of the data stream
`in`	*streamID*	stream identifier of the (output) stream which the requested data stream belongs to

Returns

S2S_Success if no error occurs

5.1.4.24 SHAREDDLL S2S_Result mcloudSendDone (MCloud ∗ *cP*)

Inform a client or a worker that there is no more data to receive.

This function should be called to inform a worker that there is no more data to receive from a client, or to inform a client, that the worker finished processing of the data received. As soon as both, worker and client have sent a done, the session is teminated.

Parameters

`in`	*cloudP*	reference to an MCloud object

Returns

S2S_Success if no error occurs

5.1.4.25 SHAREDDLL S2S_Result mcloudSendFlush (MCloud ∗ *cP*)

Inform subsequent worker to flush their output buffers.

This function should be called to inform subsequent workers finalize processing data stored in the queue and to flush their buffers.

Parameters

`in`	*cloudP*	reference to an MCloud object

Returns

S2S_Success if no error occurs

5.1.4.26 SHAREDDLL S2S_Result mcloudSendPacket (MCloud ∗ *cloudP*, MCloudPacket ∗ *p*)

Send a packet.

This function has to be called to send a data packet to the MCloud. The data packet has to be created in advance by using the packet handling functions and has to be freed after usage.

Parameters

`in`	*cloudP*	reference to an MCloud object
`in`	*p*	an MCloud packet

Returns

S2S_Success if no error occurs

5.1.4.27 SHAREDDLL S2S_Result mcloudSendPacketAsync (MCloud ∗ *cloudP*, MCloudPacket ∗ *p*, void ∗ *userData*)

Send a packet asynchronously.

This function can be used for sending data packets asynchronously. The packages will be placed into an iternal queue and sent in the background. Callback functions are used to forward status messages such as errors. Use mcloudBreak to stop sending pending packages. Use mcloudWait-Finish to wait until the last package has been sent. Packages are freed automatically after they have been sent. While sending, the error callback function may be called in case of erorrs and the break callback function if mcloudBreak has been called. As soon as no more packages are pending and mcloudWaitFinish has been called, the finalize callback function is called.

Parameters

`in`	*cloudP*	reference to an MCloud object
`in`	*p*	an MCloud packet
`in`	*userData*	additional user data, that is passed to the callback functions

Returns

S2S_Success if no error occurs

5.1.4.28 SHAREDDLL S2S_Result mcloudSetAttr (MCloud ∗ *cloudP*, MCloudAttribute *attr*, const void ∗ *value*)

Set the value of an attribute.

Parameters

in	*cloudP*	reference to an MCloud object
in	*attr*	an MCloud attribute
in	*value*	reference to the value to be set

Returns

S2S_Success if no error occurs

5.1.4.29 SHAREDDLL void mcloudSetBreakCallback (MCloud ∗ *cloudP*, MCloudType *queueType*, MCloudCallbackFct ∗ *callback*, void ∗ *userData*)

Set a break callback function.

This function is called when the worker should stop the processing as soon as possible. This callback is available for both the sending and the processing queue.

Parameters

in	*cloudP*	reference to an MCloud object
in	*queueType*	type of queue MCloudSendingQueue, or MCloudProcessingQueue
in	*callback*	reference to the function that has to be called
in	*userData*	reference to some data that is passed to the callback function

5.1.4.30 SHAREDDLL void mcloudSetDataCallback (MCloud ∗ *cloudP*, MCloudPacketCallbackFct ∗ *callback*)

Set a data callback function.

This function is called for each incoming data package in a serial way, i.e. after a package has been processed it is called again if more packages are pending. Note that for this function no userData is given at the time of the set of the callback function. Instead, the userData is given per packet with mcloudProcessDataAsync or mcloudSendAsync. This callback is available for the processing queue only.

Parameters

in	*cloudP*	reference to an MCloud object
in	*callback*	reference to the function that has to be called

5.1.4.31 SHAREDDLL void mcloudSetErrorCallback (MCloud ∗ *cloudP*, MCloudType *queueType*, MCloudCallbackFct ∗ *callback*, void ∗ *userData*)

Set an error callback function.

This function is called as soon as an error occurs in the asynchronous processing. This callback is available for both the sending and the processing queue.

Parameters

`in`	*cloudP*	reference to an MCloud object
`in`	*queueType*	type of queue MCloudSendingQueue, or MCloudProcessingQueue
`in`	*callback*	reference to the function that has to be called
`in`	*userData*	reference to some data that is passed to the callback function

5.1.4.32 SHAREDDLL void mcloudSetFinalizeCallback (MCloud ∗ *cloudP*, MCloudCallbackFct ∗ *callback*, void ∗ *userData*)

Set a finalize callback function.

This function is called as soon as the processing of packets should be finalized, i.e. no more packets will follow and the worker should output the final results after all pending packets have been processed. This callback is only available for the processing queue.

Parameters

`in`	*cloudP*	reference to an MCloud object
`in`	*callback*	reference to the function that has to be called
`in`	*userData*	reference to some data that is passed to the callback function

5.1.4.33 SHAREDDLL void mcloudSetInitCallback (MCloud ∗ *cloudP*, MCloudPacketCallbackFct ∗ *callback*, void ∗ *userData*)

Set an init callback function.

This function is called as soon as an incoming service request has been accepted by the worker, i.e. in mcloudWaitForClient. The packet containing the service description is passed to the init callback function as argument. This callback is only available for the processing queue.

Parameters

`in`	*cloudP*	reference to an MCloud object
`in`	*callback*	reference to the function that has to be called
`in`	*userData*	reference to some data that is passed to the callback function

5.1.4.34 SHAREDDLL S2S_Result mcloudWaitFinish (MCloud ∗ *cloudP*, MCloudType *queueType*, int *done*)

Wait until all pending packages have been processed/ sent.

This function can be used to wait until all pending packages have been processed or sent in the queue specified.

Parameters

in	*cloudP*	reference to an MCloud object
in	*queueType*	type of queue MCloudSendingQueue, or MCloudProcessingQueue
in	*done*	if set to 1, indicates that processing of the request has been completed

Returns

S2S_Success if no error occurs

5.1.4.35 SHAREDDLL S2S_Result mcloudWaitForClient (MCloud ∗ *cloudP*, char ∗∗ *streamID*)

Wait for a service request to process.

This function has to be called after the worker has been sucessfully connected to the MCloud in order to wait for an incoming service request to process.

Parameters

in	*cloudP*	reference to an MCloud object
out	*streamID*	ID of input stream

Returns

S2S_Success if no error occurs

5.1.4.36 SHAREDDLL MCloudWordToken∗ mcloudWordTokenArrayCreate (int *n*)

Creates an array of MCloudWordTokens.

Parameters

in	*n*	number of elements

Returns

reference to the created token array or NULL if failed

5.1.4.37 SHAREDDLL void mcloudWordTokenArrayFree (MCloudWordToken ∗ *tokenA*, int *n*)

Free an array of MCloudWordTokens.

Parameters

in	*tokenA*	reference to an MCloudWorkToken array
in	*n*	number of elements

For copies of reports, updates on project activities and other EU-BRIDGE related information, contact:

Margit Rödder
roedder@kit.edu
KIT - Press and Communication
Adenauerring 2, Building 50.20
76131 Karlsruhe, Germany
Tel.: +49 721 608 48676

Copies of public reports and other material can also be accessed via the project's homepage: http://www.eu-bridge.eu