ERICSSON

Ericsson Internal

Approved: Per Berggren	No: 7/FD-MAS001 Uen

Copyright Ericsson AB All rights reserved	Author: Bernard Melsom/Erik Tegeback Title: FD – Stream	Version: B Date: 2008-08-05	1/21

FD – Stream

Content

History

Version	Date	Adjustments
A	2006-10-06	First revision. (MBEME)
B	2008-08-05	Updated codec handling and RTP packet failures. Updated in DP5 in order include Silence detection. (EERITEG/EMATSHA)

1 Introduction

This document describes the implementation of the Stream component.

Basically the Stream component streams media to a RTP stream. The media comes in as a MediaObject, see [1]. As a MediaObject may carry media of any type, WAVE, MOV etc, the Stream component uses the C++ library MediaLibrary to extract information about the media in order to set up the properties of the stream.

2 External dependencies

This component is dependent of the components Event Notifier and Media Translation Manager. This component also depends on the QuickTime library found in Foundation.

2.1 3pp

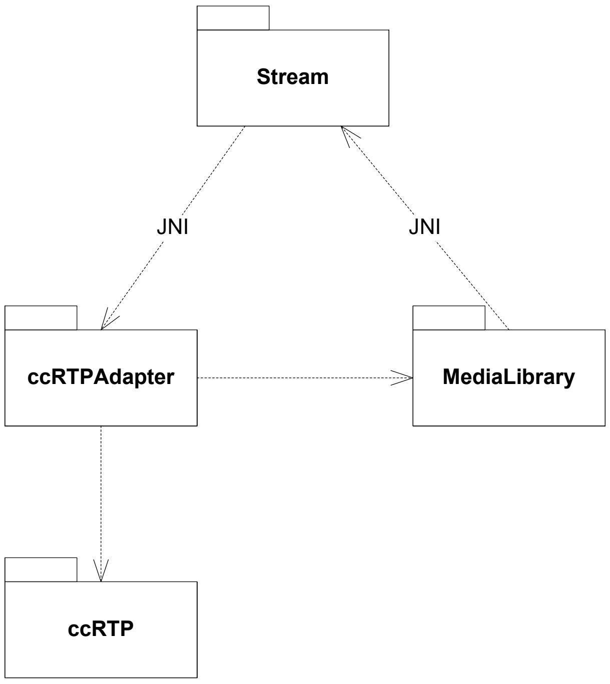
The core of the Stream RTP handling is based upon libraries from GNU Telephony Class Framework (see ref. [3]). The libraries are GNU Common C++ and GNU ccRTP.

3 Function Structure

Hereby follows an overview of the Stream design. The overview shows the structure of Stream both according to the OSI model and in a package diagram. The overview is followed by more detailed description of the individual packages.

3.1 Package Overview

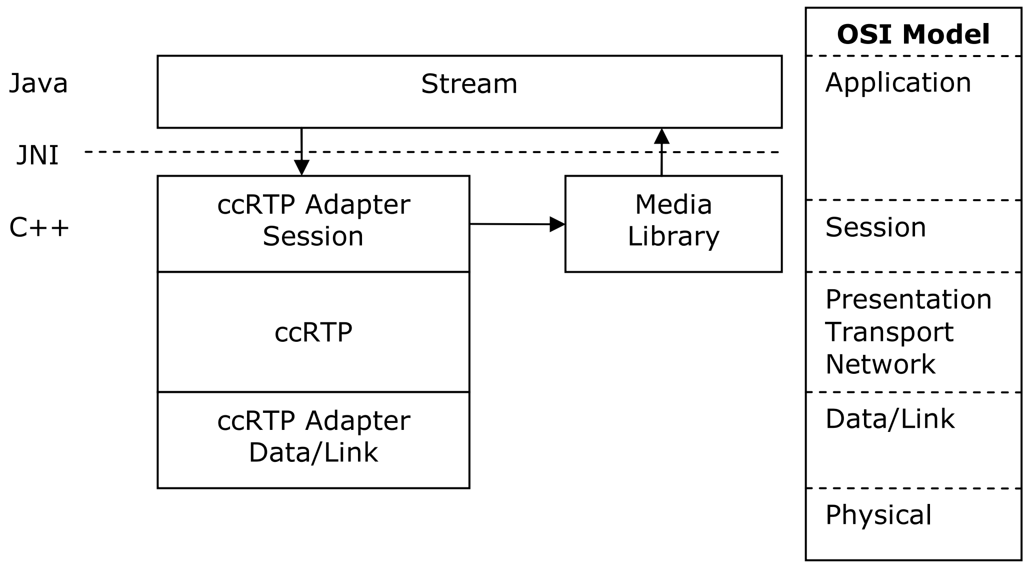
The Stream component is implemented in the following packages.



- Stream package
  this package is implemented in Java. The Stream package contains the classes that implement the Stream component interface. The Stream package acts as a proxy/façade of the RTP stack (implemented in C++).

- ccRTPAdapter package
  this package is implemented in C++. The ccRTPAdapter is wrapping/adapting the RTP stack implementation (ccRTP). The adapting includes implementation of both ISO Session and Data/Link layer. Thread handing is also implemented in ccRTPAdapter.
- MediaLibrary package
  this package is implemented in C++. The MediaLibrary implements the codec awareness of Stream. MediaLibrary uses JNI to access the Stream package (and media objects).
- ccRTP package
  this package, which is a 3PP, is implemented in C++. The ccRTP implements an RTP stack. No threads are created by ccRTP.

3.2 The OSI Model



3.2.1 Application Layer

Most of the application layer is implemented by the Stream package. The Stream package provides means to handle inbound and outbound streaming of media objects.

Unfortunately though is the ccRTPAdapter package designed to be codec aware and must have access to the media objects. As a consequence of this the application layer is handled in both the Java domain and C++ domain.

Some parts of the application layer is implemented (and even duplicated) in the packages MediaLibrary and ccRTPAdapter.

3.2.2 Session Layer

The session handling is implemented in the ccRTPAdapter package. The session handling involves threads for sending and receiving RTP.

3.2.3 Presentation, Transport and Network Layer

The ccRTP (GNU 3PP) implements the RTP stack, the presentation, transport and network layers.

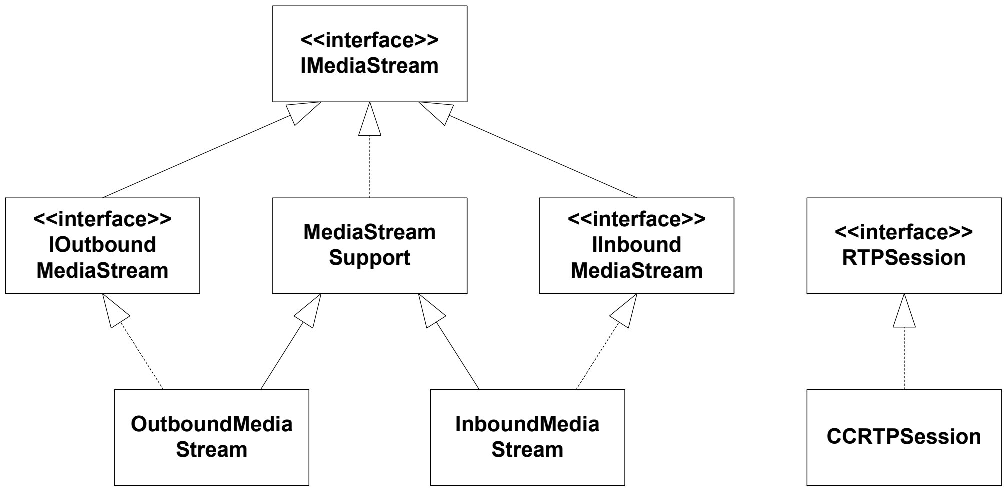
3.2.4 Data/Link Layer

The data/link layer is also implemented in the ccRTPAdapter. Specifically the RTP socket handling (read and write of RTP packages) is handled by ccRTPAdapter.

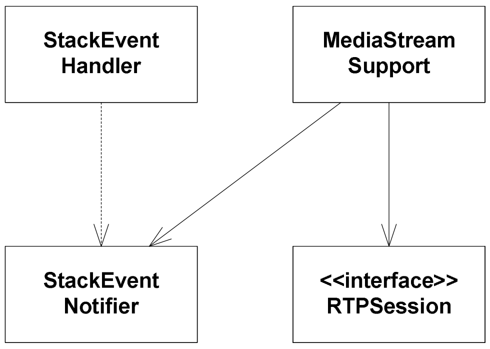
3.2.5 Physical Layer

3.3 Stream Package

Here is a class diagram showing the Stream package types.


<<interface>> IMediaStream
<<interface>> IOutbound MediaStream
MediaStream Support
<<interface>> IInbound MediaStream
<<interface>> RTPSession
OutboundMedia Stream
InboundMedia Stream
CCRTPSession

And here is a class showing the relations.



Please note that the most relevant classes are OutboundMediaStream, InboundMediaStream and CCRTPSession.

3.3.1 OutboundMediaStream

This class implements the handling of the outbound streaming service. The RTP session is accessed through CCRTPSession.

3.3.2 InboundMediaStream

This class implements the handling of the inbound streaming service. The RTP session is accessed through CCRTPSession.

3.3.3 CCRTPSession

This class acts as a proxy for an RTP session in the C++ domain. A CCRTPSession object holds a handle to an instance of SessionSupport (base class for InboundSession and OutboundSession, in the C++ domain, specifically ccRTPAdapter).

3.3.4 StackEventHandler

This class is a factory for creating instances of StackEventNotifier.

3.3.5 StackEventNotifier

The purpose of this class is to receive stack events from the ccRTPAdapter (over JNI) and forward them to an event dispatcher.

This class also provides means to perform blocked wait for stack events (e.g. wait for play finished).

3.3.6 IMediaStream

The main purpose of this interface is reuse. No class should implement this interface directly but implement one of the two sub interfaces instead.

3.3.7 MediaStreamSupport

This is a base class that contains common logic for inbound and outbound stream classes. The main responsibility of this class is to create an appropriate implementation of the IMediaStream interface.

3.3.8 RTPSession

Component-internal interface that defines the needed functionality of an RTP stack implementation. The methods are at a fairly high level. The reason is that the stack implementation is likely to be implemented in C/C++. To limit the times the boundary between Java and C/C++ have to be crossed some logic has to be implemented on top of the stack implementation, most of it in the same language as the stack implementation itself.

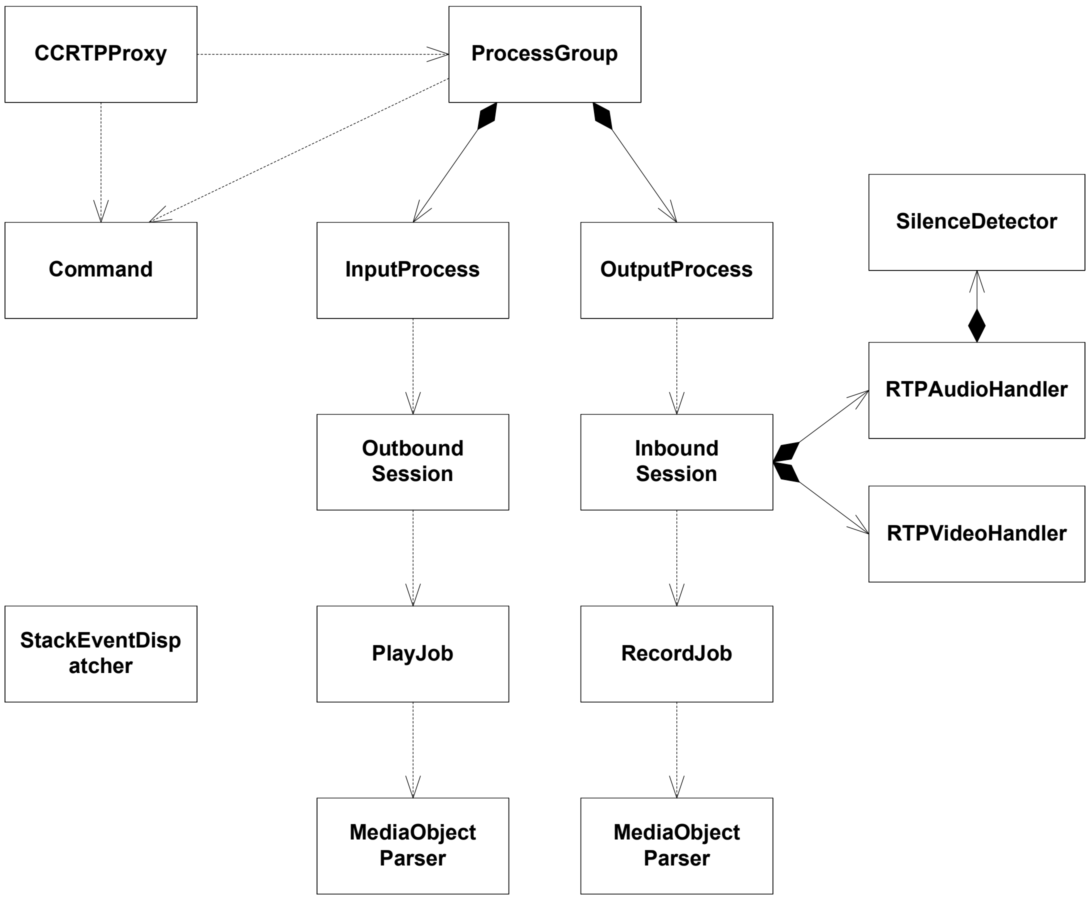
At the first glance it might be tempting to make this interface inherit from IInboundMediaStream and IOutboundMediaStream. IRTPSession is, however, an internal interface to the Stream component that should be possible to modify without consequences to the public interface.

3.4 ccRTP Adapter

The ccRTP Adapter implements both the Session and Data/Link level of the RTP handling in Stream.

Since ccRTP is agnostic with respect to specific payload and such it is up to the ccRTP Adapter to distinguish between audio, video, DTMF etc. and handle play, record and join.

The general idea is to have, for each CPU on the host machine, one thread handling the outbound RTP and four threads handling the inbound RTP.



3.4.1 ProcessorGroup

Handles load balancing of the In/Out-put Processors, and distribution of Commands to the input/output processors.

The purpose of the command procedure is to serialize the concurrent JNI accesses and in that manner achieve thread safe access.

3.4.2 InputProcessor

This class is a thread and handles the inbound RTP traffic. RTP packets are received from the UDP sockets and distributed to the inbound queues of the registered (currently active) RTP stacks.

The InputProcessor is also executing inbound stream related commands such as for example record and join.

3.4.3 OutputProcessor

This class is a thread and handles the outbound RTP traffic. RTP packets are retrieved from the outbound queues of the registered (currently active) RTP stacks.

The OutputProcessor is also executing outbound stream related commands such as for example play.

3.4.4 Command

This is a super class which is the base for all the used commands (e.g. play, join, etc.).

3.4.5 Player and PlayJob

These two classes (should perhaps have been one) are responsible for playing media. There is a one to one relation between a play and a media object. Once executed (by Player) the PlayJob takes the media object data and transfers the data to the appropriate RTP session (video or audio).

In order to perform the data transfer the PlayJob utilizes a MediaParser.

The Player/PlayJob is also responsible for communicating the status/result by issuing Play Finished and Play Failed events.

3.4.6 Recorder and RecordJob

These two classes (should perhaps have been one) are responsible for recording media. There is a one to one relation between a record and a media object. During record the RTP data is stored in memory. The recording is stopped automatically if either the maximum recording duration has been reached or if a period of silence, longer than the maximum silence duration is found.

Once the record is terminated the RecordJob will transfer the RTP data into the media object. If the recording was stopped due to silence detection, the silent part of the message will be removed (except for 900ms that is retained in order to prevent loss of spoken audio).

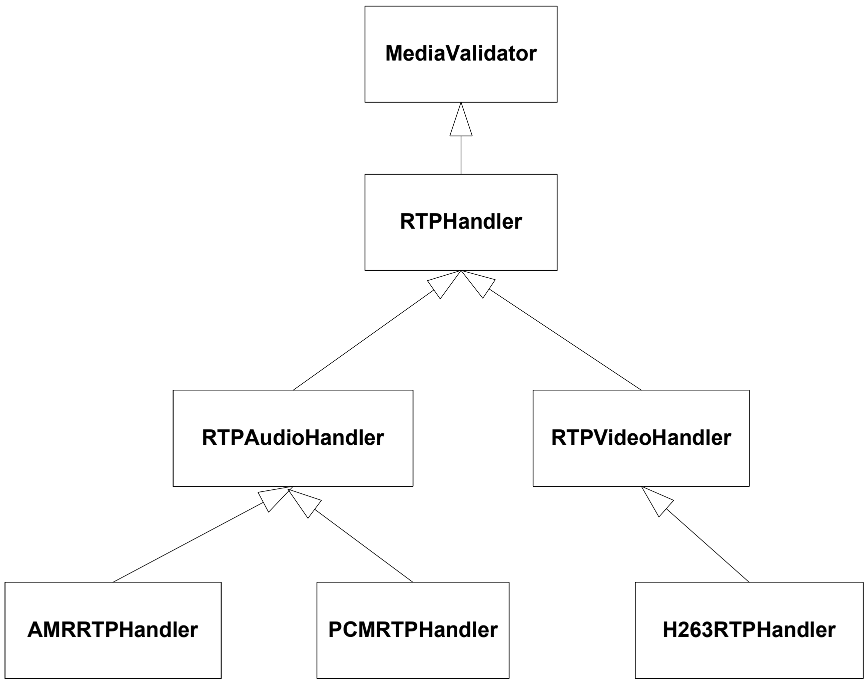
In order to perform the transfer the RecordJob utilizes a MediaBuilder.

The Recorder/RecordJob is also responsible for communicating the status/result by issuing Record Finished and Record Failed events.

3.4.7 RTPHandler

RTPHandler is the base for classes handling codec dependent functions during record. RTPHandler is subclassed into RTPAudioHandler and RTPVideoHandler which in turn are subclassed for each supported codec. The RTPHandler also implements the MediaValidator interface described in 3.5.8. The implemented RTPHandler are shown in the class diagram below.

The silence detector is part of the RTPAudioHandler only since silence detection is never used for video calls. The actual implementation of silence detection is codec dependant and currently only the PCMRTPHandler actually implements it. Therefore silence detection is only supported for G.711 μ-law/A-law and not for AMR.



3.4.8 SilenceDetector and PCM Conversion Table

These two classes are responsible for checking if recorded data can be considered being silence, i.e. when the energy contained in a certain percentage of the PCM samples in a frame (e.g. 40ms) is below a constant threshold value (for Static Silence Detection, Mode=1) or when the average energy level of a frame is below an adaptive threshold value (for Dynamic Silence Detection, Mode=2). Currently the silence detector is only capable of handling G.711 μ-law and G.711 A-law (PCMU and PCMA).

More details of the silence detection algorithms are described in FS Stream [2].

3.4.9 StreamMixer

Handles join. RTP data is forwarded from one InboundSession to one or more OuboundSessions.

A StreamMixer is owned by an InboundSession. When joining an InboundSession and an OutboundSession the StreamMixer creates and holds a StreamConnection object (which holds a reference to an OutboundSession).

3.4.10 StreamRTPSession

This class implements user specific RTP session behavior and is also acting as bridge for the Stream specific ccRTP adaptations.

StreamRTPSession is implementing a base class defined in the ccRTP stack.

3.4.11 InboundSession

This is the "application" layer of an inbound RTP session.

The InboundSession aggregates, amongst all classes involved in an Inbound Stream session, a video and audio RTP session (StreamRTPSession), and a StreamMixer.

3.4.12 OutboundSession

This is the "application" layer of an outbound RTP session.

The OutboundSession aggregates, amongst all classes involved in an Outbound Stream session, a video and audio RTP session (StreamRTPSession).

3.4.13 ccRTPProxy

This is the JNI interface (from Java to C++).

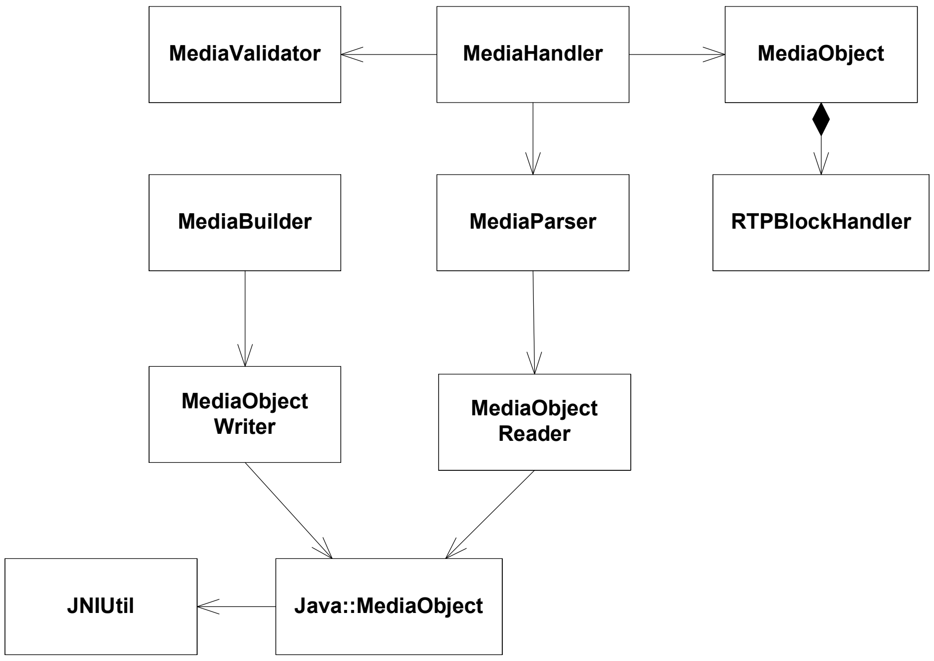
The ccRTP proxy is a collection of functions which are defined from the "native" methods of the corresponding Java class (CCRTPSession). These functions are called by the Java domain.

3.4.14 CommandToken

This class is the representation of a DTMF event which consist of information on which key was pressed, duration and volume.

3.5 MediaLibrary

The Media Library is responsible for the handling of media. The Media Object (see [1]) is central to the media handling. With respect to streaming the Media Objects are either playable or recordable. Hence the work includes translation of media data into RTP payload data back and forth. Besides from the Media Object Media Parsers and Media Builders are central here.



3.5.1 Java MediaObject

This class is the representation of a media object in the C++ domain. Java MediaObject encapsulates accesses to the Java domain (the media object and related objects). All JNI calls are made through JNIUtil.

3.5.2 MediaObject

This class represents all the information needed to stream a media object over RTP. The central part is the aggregated RTPBlockHandler object containing RTP payloads for the media together with meta information needed to create RTP packets to be streamed.

3.5.3 RTPBlockHandler

The RTPBlockHandler encapsulates a memory buffer containing RTP payloads and meta data needed to create RTP packets to be streamed.

RTPBlockHandler also reserves memory for the object structure created by the RTP stack CCRTP to manage the RTP packets, in this way the cost for allocating memory is decreased during RTP transmission.

3.5.4 MediaReader

The MediaReader encapsulates a MediaObject in order to provide a file access abstraction. The MediaReader provide a set of read methods which mimics random file read access.

3.5.5 MediaWriter

The MediaWriter encapsulates a MediaObject in order to provide a file access abstraction. The MediaWriter provide a set of write methods which mimics random file write access.

3.5.6 MediaParser

This is a base class for a media parser. A media parser is responsible for translating the media data (audio/video) to RTP payload data. The structure of the media data depends on the media currently there are three media parsers (WavParser, MovParser and AmrParser).

A MediaParser operates on a MediaReader.

3.5.7 MediaHandler

The MediaHandler encapsulates the parsing of java MediaObjects into the C++ media object holding the RTP payload data.

3.5.8 MediaValidator

This is a interface class for media validators. A media validator is used to validate a parsed media object against the requirements of the RTP session, for instance that the codec modes in an AMR media file corresponds to the codec modes negotiated by the SDP media descriptor.

3.5.9 MediaBuilder

This is a base class for a media builder. A media builder is responsible for translating RTP payload data (audio/video) into media data. The structure of the media data depends on the media. Currently there are three media builders (WavBuilder, MovBuilder and AmrBuilder).

A MediaBuilder operates on a MediaWriter.

3.5.10 JNIUtil

This is the JNI interface (from C++ to Java).

JNIUtil encapsulates the JNI environment. All JNI calls must be made through this class.

4 Function Behavior

This chapter describes how the more complex concepts are solved in stream. Note that the sequence diagrams are in no way complete. Their purpose is to clarify the text.

4.1 RTP packet failures

4.1.1 RTP packet sequence number validation

During record an RTP packet is considered valid if its sequence number is no more than 100 packets behind and no more than 3000 packets after the last received packet. Invalid packets are ignored. If packets starts to come in sync again after a drop of more than 3000 packets all earlier received packets are ignored and recording restarts. See appendix A.1 in [4] for more details on packet sequence number validation.

4.1.2 Reordered packets

During record RTP packets are maintained in a list sorted by the packets extended sequence number. The extended sequence number of an RTP packet is a 32 bit integer where the 16 most significant bits represents the RTP sequence number “wrap around” count and the 16 least significant bits the RTP sequence number. In this way packets that are received out of order are managed. When a packet is received the position in the list is found by searching from the back of the list for a packet with an extended sequence number less than the extended sequence number of the received packet.

4.1.3 Missed packets

Missed packets are handled differently depending on the codec used. For G.711 µ-Law, silence is inserted to fill up the gap. For AMR, DTX NO_DATA packets are inserted and for H.263 nothing is done. For more information on codec dependent functionality in record, see section 4.2.2.

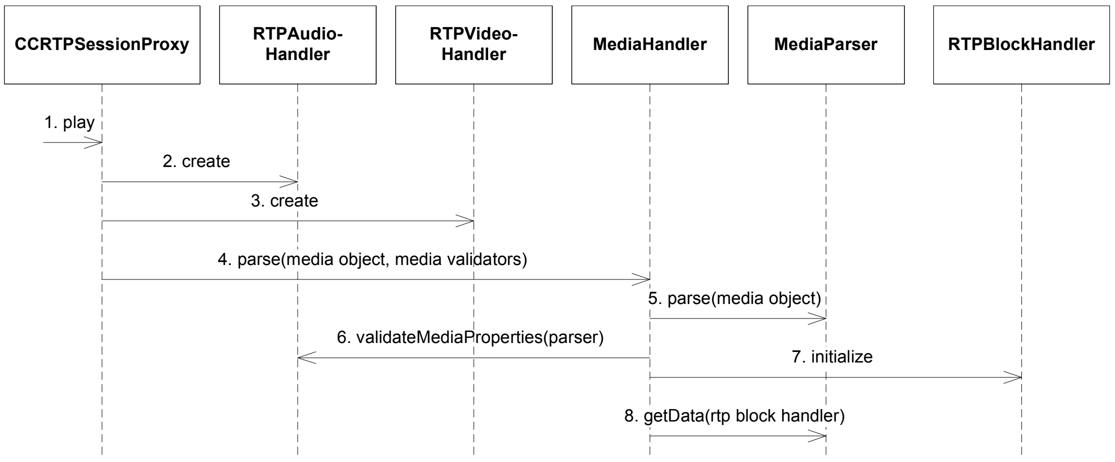
4.2 Codec handling

Algorithms for play and record are designed to be codec independent. Codec dependent functions are instead factored out to separate classes (RTPHandlers) that handle the codec dependent parts.

4.2.1 Play

In play the media parsers initializes an RTPBlockHandler object with the information needed to stream the media over RTP without any knowledge of the codec used. Some media properties have to be validated against the RTP payload properties setup by the SDP media negotiation. For example the codec modes in AMR media has to be validated against the codec modes setup by the media negotiation. This kind of validation is done by passing MediaValidator objects to the MediaHandler parse method. The validateMediaProperties method for each MediaValidator object are then called, with the media parser as argument. As described earlier there is one RTPHandler sublass for each supported codec who all implements the MediaValidator interface.

The initialization of the RTPBlockHandler and media validation is schematically described below:

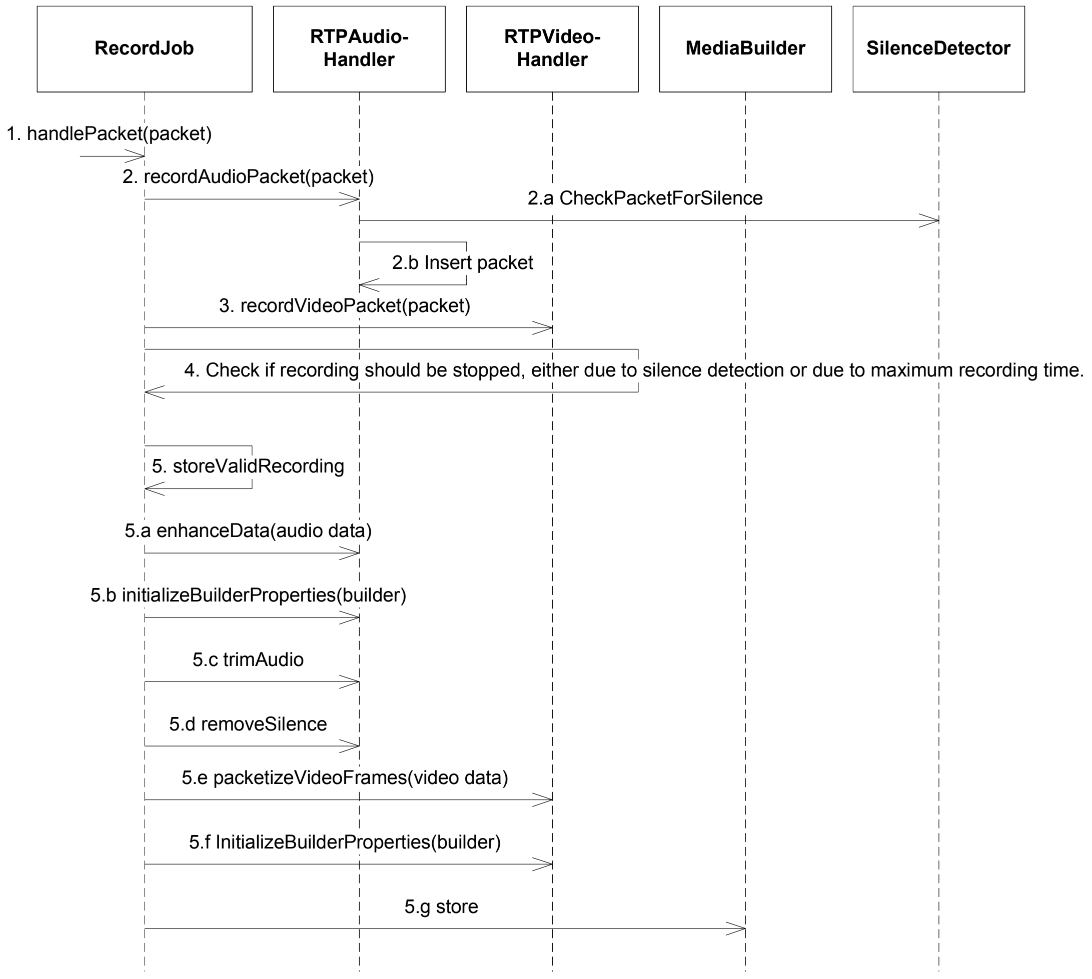


1. A play call is issued over JNI.
2. An RTPAudioHandler object is created for validating audio properties. Which class is instantiated depends on the codec used.
3. If it’s a video media, an RTPVideHandler object is created.
4. The MediaHandler is called to parse the mediaobject.
5. Mediahandler instantiates and delegates the parsing to the correct media parser.
6. Media validators are called to validate the parsed result.
7. The RTPBlockHandler is initialized.
8. The MediaHandler delegates to the parser to inject the RTP information into the RTPBlockhandler.

4.2.2 Record

The RecordJob object holds a list of audio packets and a list of video frames. A video frame is in itself a list of packets representing the packets in that frame. For each RTP packet received the record job is called which then delegates that information to RTPHandler objects. Audio and video packets are handled a bit differently. When an audio packet is received the RTP audio handler updates the list of audio packets owned by the record job. When a video packet is received the RTP video handler adds this packet to its own list of received packets. When the recording is stopped the record job delegates the handling of missed packets to the RTP audio handler and the video handler to package the received video packets into frames.

The recording process is schematically described below.


RecordJob
RTPAudio-Handler
RTPVideo-Handler
MediaBuilder
SilenceDetector
1. handlePacket(packet)
2. recordAudioPacket(packet)
2.a CheckPacketForSilence
2.b Insert packet
3. recordVideoPacket(packet)
4. Check if recording should be stopped, either due to silence detection or due to maximum recording time.
5. storeValidRecording
5.a enhanceData(audio data)
5.b initializeBuilderProperties(builder)
5.c trimAudio
5.d removeSilence
5.e packetizeVideoFrames(video data)
5.f InitializeBuilderProperties(builder)
5.g store

1. The record job is called for each received packet.
2. If it is an audio packet the RTP audio handler is called to manage that packet. The RTPAudioHandler will in turn invoke the implementation for the specific codec, eg. PCMRTPHandler.
    a. If silence detection is enabled and supported for the codec the packet will be analyzed for silence by the Silence Detector.
    b. The packet is inserted in a list of audio packets.
3. If it's a video packet the RTP video handler is called.
4. RecordJob checks if the criteria for silence detection is fulfilled and in that case stops the recording. A check is also made to ensure that the maximum recording time is not exceeded.
5. When the recording is stopped, the record job stores the recording using a media builder.
    a. The RTP audio handler compensates for missed packets. This is done differently in different audio handler, for PCMU/PCMA silence is added and for AMR DTX NO_DATA is added.

b. The RTP audio handler initializes the media builder with session dependent parameter, e.g. for AMR the codec modes are set.

c. A configurable amount of audio is removed from the beginning in order to avoid recording the beep.

d. If the recording was stopped due to silence detection then the silent part of the audio will be removed (less a 900ms margin).

e. The RTP video handler is called to package the video packets into video frames.

f. The RTP video handler initializes the media builder with session dependent properties. Currently no video properties are set.

g. The RTP data is stored using the media builder.

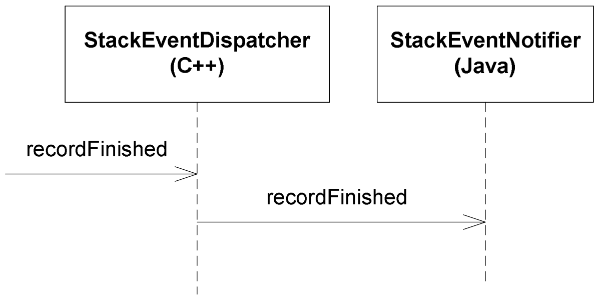
4.3 Events

Since the Stream package implements internal event handing, beside from the event handling provided by the EventNotifier, this is a topic that should be further explained.

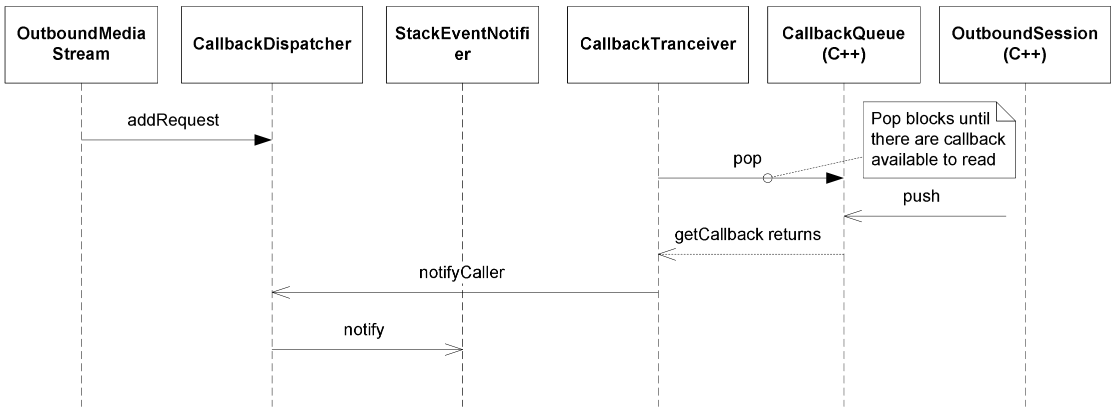
Since the communication between the Java and C++ is asynchronous and runs in different threads/domains the event handling is implemented.

Currently there are two different mechanisms for internal event handling. Originally there was only one mechanism but that involved synchronous calls into the java domain from the C++ domain oven JNI with the effect that the real time requirements of the play command could not be met. Therefore another event handling mechanism was developed for the events used in the play call, playFinished and playFailed.

Below is an example of the original event handling, the StackEventNotifier dispatches the event to the external receivers.



The mechanism used for the events in play are a bit more complex and is illustrated schematically below.



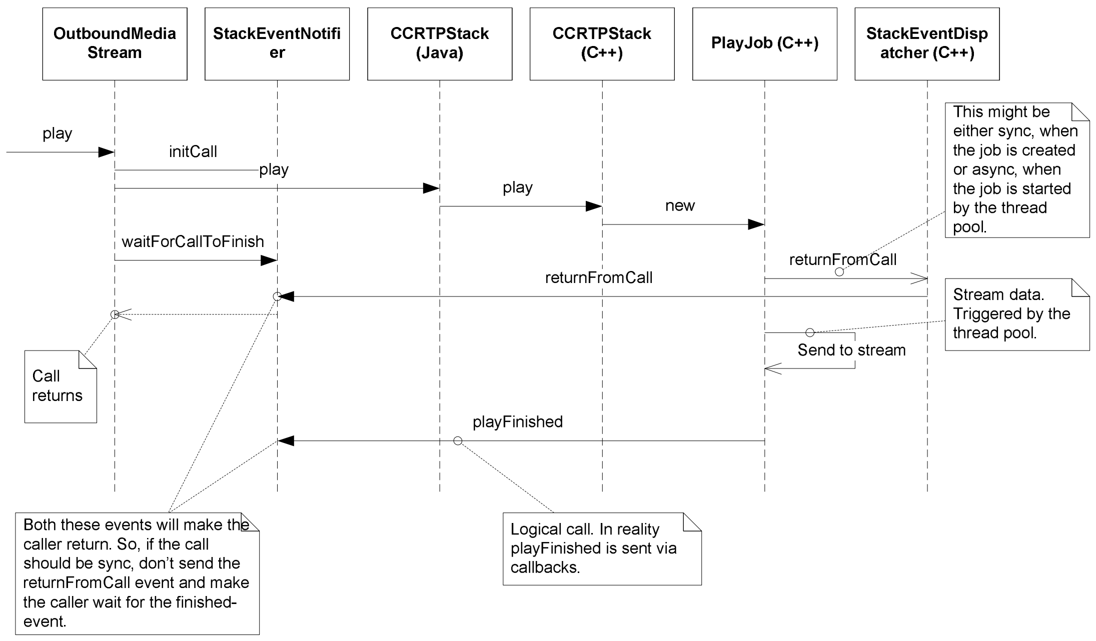
1. OutboundStream adds a request specifying a requested id and the StackEventNotifier that will handle the event to the singleton class CallbackDispatcher.
2. CallbackTranceiver threads polls the C++ callback queue over JNI.
3. When the C++ outbound session wants to post a playFinished or playFailed event it post a callback on the callback queue.
4. The CallbackTranceiver tells the CallbackDispatcher to notify the caller that an event has occurred.
5. The CallbackDispatcher picks up the request added earlier and calls the StackEventNotifier that will dispatch the event to the external receivers.

In this way there is no potentially time consuming synchronous calls to the java side from the C++ real time threads.

4.4 Asynchronous/Synchronous calls

Call to record can be either asynchronous or synchronous. Calls to play are always asynchronous but it might still be necessary to be able to control when the call may return. All asynchronous calls will return as soon as possible, which is when the call has passed the JNI-boundary and some initial processing is done, but before the actual streaming starts. This gives the possibility to communicate some types of errors synchronously.

In the case of a play, for example, it might be necessary to check that the media conforms to the parameters that was specified when the stream was created before returning.


OutboundMedia Stream
StackEventNotifier
CCRTPStack (Java)
CCRTPStack (C++)
PlayJob (C++)
StackEventDispatcher (C++)
play
initCall
play
play
new
waitForCallToFinish
returnFromCall
returnFromCall
Send to stream
playFinished
This might be either sync, when the job is created or async, when the job is started by the thread pool.
Stream data. Triggered by the thread pool.
Call returns
Both these events will make the caller return. So, if the call should be sync, don't send the returnFromCall event and make the caller wait for the finished-event.
Logical call. In reality playFinished is sent via callbacks.

This is the basic flow of control for an asynchronous operation. Note that the playFinish call from PlayJob to StackEventNotifier is logical and in reality is handled as the ecent mechanism described above. Each call is identified with an id; a `callId`. This id makes it possible for a stream to wait for a specific event for a specific call.

There are three possible scenarios.

4.4.1 Asynchronous Scenario

The call is "completely" asynchronous: No call to `waitForCallToFinish` is made and the call returns as soon as an asynchronous job has been created at the C++-side an added to the thread pool.

1. The scenario starts when play method is called.
2. The PlayJob is created and started.
3. The play method returns.
4. Once the PlayJob finishes a play finished is issued.

4.4.2 Semi Synchronous Scenario

The call is asynchronous but some additional processing is done at the C++-side: The stream calls `waitForCallToFinish` and the asynchronous job calls `returnFromCall` when the caller is allowed to return from the call.

1. The scenario starts when play method is called.
2. The PlayJob is created.
3. The play method returns.
4. The OutboundStream calls waitForCallToFinish (which is blocking).
5. The PlayJob is started.

6. The PlayJob calls returnFromCall.
7. A returnFromCall event propagates to the Java domain and the OutboundStream resumes execution.
8. Once the PlayJob finishes a play finished is issued.

4.4.3 Synchronous Scenario

The call is completely synchronous: The stream calls `waitForCallToFinish` and the asynchronous job does not call `returnFromCall` but calls `playFinished` when the whole operation is done.

1. The scenario starts when play method is called.
2. The PlayJob is created.
3. The play method returns.
4. The OutboundStream calls waitForCallToFinish (which is blocking).
5. The PlayJob is started.
6. Once the PlayJob finishes a play finished is issued.
7. The waitForCallToFinish returns and OutboundStream resumes execution.

4.5 Threads

In the C++ domain the RTP traffic is handled by a fixed number of threads. The number of threads is a multiple of the number CPUs. There is one OutputProcessor thread handling outbound RTP and four InputProcessor threads handling inbound RTP for each CPU, hence there are 5 threads per CPU.

In the Java domain one thread is created for each CallbackTranceiver, see the description of event mechanisms. There is a one to one correspondence between the number of CallbackTranceiver threads and the number of C++ OutputProcessor threads.

4.5.1 Input Processor

The inbound thread is monitoring the UDP sockets which are used for RTP input (there are two for each RTP (RTP/RTCP) session). Once data arrives the data is dispatched to the associated RTP session.

The inbound thread is also responsible for the execution of the active inbound sessions (e.g. join and record).

The inbound sessions are evenly distributed over the available input threads.

4.5.2 Output Processor

The outbound thread is responsible for UDP transmission by monitoring the outbound RTP sessions and transferring RTP packets from the sessions to the associated UDP sockets.

The outbound thread is also responsible for the execution of the active outbound sessions (e.g. the play operation).

The output sessions are evenly distributed over the output threads.

4.6 JNI

When transferring data between Java and C++ the following NIO-methods are used to improve performance:

- `NewDirectByteBuffer`
- `GetDirectBufferAddress`
- `GetDirectBufferCapacity`

Otherwise, all data will be copied in memory each time the boundary between Java and C/C++ is crossed.

Some Java-objects are cached and therefore global references are created for these objects. Such references are deleted when, either a "finished"-event is sent or an exception occurs.

No Java environment references are cached; such references are retrieved for each time access to Java-space is needed by attaching the current thread to the JVM. Each time the current thread is attached it must be detached when the Java-call is done.

5 References

[1] FD – Media Object
13/FD-MAS0001

[2] FS – Stream
7/FS-MAS0001

[3] GNU Telephony Class Framework
http://wiki.gnutelephony.org

[4] RFC 3550 - RTP: A Transport Protocol for Real-Time Applications

6 Terminology

Term	Explanation

7 Appendix

From the bottom an up we have the RTP stack. RTP application and the Stream interface. The RTP stack is a 3PP (ccRTP) source code from GNU. The lowest level, socket handling etc. is implemented at Mobeon. Since the RTP stack (and specification) is quite generic a lot of RTP is handled in the application layer. In Stream the application layer is represented by ccRTP Adapter. The Stream functionality is implemented on top of the ccRTP Adapter.

Since ccRTP is implemented in C++ the communication between the Java and C++ domains is handled through JNI. Hence there is also a JNI layer in the Stream design.

In the current design the C++ domain is media aware and responsible for parsing the media data. This yields that the C++ is tightly coupled to the Java domain. In order to fulfill the streaming the C++ domain must have access to most of the Java objects which are involved in the process.

Media Library implements most of the accesses of the Java domain which are made by the C++ domain. Media Library also handles the media parsing and building (media data read and write). The actual file accesses are handled outside of Stream.