

八 RTSPClient分析

有RTSPServer，当然就要有RTSPClient。

如果按照Server端的架构，想一下Client端各部分的组成可能是这样：

因为要连接RTSP server，所以RTSPClient要有TCP socket。当获取到server端的DESCRIBE后，应建立一个对应于ServerMediaSession的ClientMediaSession。对应每个Track，ClientMediaSession中应建立ClientMediaSubsession。当建立RTP Session时，应分别为所拥有的Track发送SETUP请求连接，在获取回应后，分别为所有的track建立RTP socket，然后请求PLAY，然后开始传输数据。事实是这样吗？只能分析代码了。

testProgs中的OpenRTSP是典型的RTSPClient示例，所以分析它吧。

main()函数在playCommon.cpp文件中。main()的流程比较简单，跟服务端差别不大：建立任务计划对象——建立环境对象——处理用户输入的参数(RTSP地址)——创建RTSPClient实例——发出第一个RTSP请求（可能是OPTIONS也可能是DESCRIBE）——进入Loop。

RTSP的tcp连接是在发送第一个RTSP请求时才建立的，在RTSPClient的那几个发请求的函数sendXXXXXXCommand()中最终都调用sendRequest()，sendRequest()中会跟据情况建立起TCP连接。在建立连接时马上向任务计划中加入处理从这个TCP接收数据的socket handler：RTSPClient::incomingDataHandler()。

下面就是发送RTSP请求，OPTIONS就不必看了，从请求DESCRIBE开始：

```
[cpp]
01. void getSDPDescription(RTSPClient::responseHandler* afterFunc)
02. {
03.     ourRTSPClient->sendDescribeCommand(afterFunc, ourAuthenticator);
04. }
05. unsigned RTSPClient::sendDescribeCommand(responseHandler* responseHandler,
06.     Authenticator* authenticator)
07. {
08.     if (authenticator != NULL)
09.         fCurrentAuthenticator = *authenticator;
10.     return sendRequest( new RequestRecord(++fCSeq, "DESCRIBE", responseHandler));
11. }
```

参数responseHandler是调用者提供的回调函数，用于在处理完请求的回应后再调用之。并且在这个回调函数中会发出下一个请求——所有的请求都是这样依次发出的。使用回调函数的原因主要是因为socket的发送与接收不是同步进行的。类RequestRecord就代表一个请求，它不但保存了RTSP请求相关的信息，而且保存了请求完成后的回调函数——就是responseHandler。有些请求发出时还没建立tcp连接，不能立即发送，则加入fRequestsAwaitingConnection队列；有些发出后要等待Server端的回应，就加入fRequestsAwaitingResponse队列，当收到回应后再从队列中把它取出。

由于RTSPClient::sendRequest()太复杂，就不列其代码了，其无非是建立起RTSP请求字符串然后用TCP socket发送之。

现在看一下收到DESCRIBE的回应后如何处理它。理论上是跟据媒体信息建立起MediaSession了，看看是不是这样：

```

01. void continueAfterDESCRIBE(RTSPClient*, int resultCode, char * resultString)
02. {
03.     char * sdpDescription = resultString;
04.     //根据SDP创建MediaSession.
05.     // Create a media session object from this SDP description:
06.     session = MediaSession::createNew(*env, sdpDescription);
07.     delete [] sdpDescription;
08.
09.     // Then, setup the "RTPSource"s for the session:
10.     MediaSubsessionIterator iter(*session);
11.     MediaSubsession *subsession;
12.     Boolean madeProgress = False;
13.     char const * singleMediumToTest = singleMedium;
14.     //循环所有的MediaSubsession, 为每个设置其RTPSource的参数
15.     while ((subsession = iter.next()) != NULL) {
16.         //初始化subsession, 在其中会建立RTP/RTCP socket以及RTPSource.
17.         if (subsession->initiate(simpleRTPoffsetArg)) {
18.             madeProgress = True;
19.             if (subsession->rtpSource() != NULL) {
20.                 // Because we're saving the incoming data, rather than playing
21.                 // it in real time, allow an especially large time threshold
22.                 // (1 second) for reordering misordered incoming packets:
23.                 unsigned const thresh = 1000000; // 1 second
24.                 subsession->rtpSource()->setPacketReorderingThresholdTime(thresh);
25.
26.                 // Set the RTP source's OS socket buffer size as appropriate - either if we were exp
27.                 // licitly asked (using -B),
28.                 // or if the desired FileSink buffer size happens to be larger than the current OS s
29.                 // ocket buffer size.
30.                 // (The latter case is a heuristic, on the assumption that if the user asked for a l
31.                 // arge FileSink buffer size,
32.                 // then the input data rate may be large enough to justify increasing the OS socket
33.                 // buffer size also.)
34.                 int socketNum = subsession->rtpSource()->RTPgs()->socketNum();
35.                 unsigned curBufferSize = getReceiveBufferSize(*env, socketNum);
36.                 if (socketInputBufferSize > 0 || fileSinkBufferSize > curBufferSize) {
37.                     unsigned newBufferSize = socketInputBufferSize > 0 ?
38.                     socketInputBufferSize : fileSinkBufferSize;
39.                     newBufferSize = setReceiveBufferTo(*env, socketNum, newBufferSize);
40.                     if (socketInputBufferSize > 0) { // The user explicitly asked for the new socket buf
41.                         fer size; announce it:
42.                         *env
43.                         << "Changed socket receive buffer size for the \""
44.                         << subsession->mediumName() << "/"
45.                         << subsession->codecName()
46.                         << "\" subsession from " << curBufferSize
47.                         << " to " << newBufferSize << " bytes\n" ;
48.                     }
49.                 }
50.             }
51.             if (!madeProgress)
52.                 shutdown();
53.
54.             // Perform additional 'setup' on each subsession, before playing them:
55.             //下一步就是发送SETUP请求了。需要为每个Track分别发送一次。
56.             setupStreams();
57.         }
58.     }
59. }

```

此函数被删掉很多枝叶，所以发现与原版不同请不要惊掉大牙。

的确在DESCRIBE回应后建立起了MediaSession，而且我们发现Client端的MediaSession不叫ClientMediaSession，SubSession亦不是。我现在很想看看MediaSession与MediaSubsession的建立过程：

```

01. MediaSession* MediaSession::createNew(UsageEnvironment& env, char const * sdpDescrip
02. tion)
03. {
04.     MediaSession* newSession = new MediaSession(env);
05.     if (newSession != NULL) {
06.         if (!newSession->initializeWithSDP(sdpDescription)) {
07.             delete newSession;
08.             return NULL;
09.         }
10.     }
11.     return newSession;
12. }

```

我可以告诉你，MediaSession的构造函数没什么可看的，那么就来看initializeWithSDP()：

内容太多，不必看了，我大略说说吧：就是处理SDP，跟据每一行来初始化一些变量。当遇到“m=”行时，就建立一个MediaSubsession，然后再处理这一行之下，下一个“m=”行之上的行们，用这些参数初始化MediaSubsession的变量。循环往复，直到尽头。然而这其中并没有建立RTP socket。我们发现在continueAfterDESCRIBE()中，创建MediaSession之后又调用了subsession->initiate(simpleRTPoffsetArg)，那么socket是不是在它里面创建的呢？look：


```

89.     delete existing; // in case it wasn't NULL
90.     continue ;
91. }
92.
93. // Make sure we can use the next (i.e., odd) port number, for RTP:
94. portNumBits rtcpPortNum = fClientPortNum | 1;
95. if (isSSM()) {
96.     fRTCPsocket = new Groupsock(env(), tempAddr,
97.     fSourceFilterAddr, rtcpPortNum);
98. } else {
99.     fRTCPsocket = new Groupsock(env(), tempAddr, rtcpPortNum,
100.    255);
101. }
102. if (fRTCPsocket != NULL && fRTCPsocket->socketNum() >= 0) {
103.     // Success! Use these two sockets.
104.     success = True;
105.     break ;
106. } else {
107.     // We couldn't create the RTP socket (perhaps that port number's already in use elsew
108.     here?).
109.     delete fRTCPsocket;
110.
111.     // Record the first socket in our table, and keep trying:
112.     unsigned key = (unsigned) fClientPortNum;
113.     Groupsock* existing = (Groupsock*) socketHashTable->Add(
114.     ( char const *) key, fRTCPsocket);
115.     delete existing; // in case it wasn't NULL
116.     continue ;
117. }
118.
119. // Clean up the socket hash table (and contents):
120. Groupsock* oldGS;
121. while ((oldGS = (Groupsock*) socketHashTable->RemoveNext()) != NULL) {
122.     delete oldGS;
123. }
124. delete socketHashTable;
125.
126. if (!success)
127.     break ; // a fatal error occurred trying to create the RTP and RTPC sockets; we can't
128.     continue
129. }
130.
131. // Try to use a big receive buffer for RTP - at least 0.1 second of
132. // specified bandwidth and at least 50 KB
133. unsigned rtpBufSize = fBandwidth * 25 / 2; // 1 kbps * 0.1 s = 12.5 bytes
134. if (rtpBufSize < 50 * 1024)
135.     rtpBufSize = 50 * 1024;
136. increaseReceiveBufferTo(env(), fRTCPsocket->socketNum(), rtpBufSize);
137.
138. // ASSERT: fRTCPsocket != NULL && fRTCPsocket != NULL
139. if (isSSM()) {
140.     // Special case for RTPC SSM: Send RTPC packets back to the source via unicast:
141.     fRTCPsocket->changeDestinationParameters(fSourceFilterAddr, 0, ~0);
142. }
143.
144. //创建RTPSource的地方
145. // Create "fRTPSource" and "fReadSource":
146. if (!createSourceObjects(useSpecialRTPoffset))
147.     break ;
148.
149. if (fReadSource == NULL) {
150.     env().setResultMsg( "Failed to create read source" );
151.     break ;
152. }
153.
154. // Finally, create our RTPC instance. (It starts running automatically)
155. if (fRTPSource != NULL) {
156.     // If bandwidth is specified, use it and add 5% for RTPC overhead.
157.     // Otherwise make a guess at 500 kbps.
158.     unsigned totSessionBandwidth =
159.     fBandwidth ? fBandwidth + fBandwidth / 20 : 500;
160.     fRTPCInstance = RTPCInstance::createNew(env(), fRTCPsocket,
161.     totSessionBandwidth, (unsigned char const *) fParent.CNAME(),
162.     NULL /* we're a client */, fRTPSource);
163.     if (fRTPCInstance == NULL) {
164.         env().setResultMsg( "Failed to create RTPC instance" );
165.         break ;
166.     }
167. }
168.
169. return True;
170. } while (0);
171.
172. //失败时执行到这里
173. delete fRTCPsocket;
174. fRTCPsocket = NULL;
175. delete fRTCPsocket;
176. fRTCPsocket = NULL;
177. Medium::close(fRTPCInstance);
178. fRTPCInstance = NULL;

```

```

178.   Medium::close(fReadSource);
179.   fReadSource = fRTPSource = NULL;
180.   fClientPortNum = 0;
181.   return False;
182. }

```

是的，在其中创建了RTP/RTCP socket并创建了RTPSource，创建RTPSource在函数createSourceObjects()中，看一下：

```

[cpp]
01. Boolean MediaSubsession::createSourceObjects( int useSpecialRTPoffset)
02. {
03.     do {
04.         // First, check "fProtocolName"
05.         if (strcmp(fProtocolName, "UDP" ) == 0) {
06.             // A UDP-packetized stream (*not* a RTP stream)
07.             fReadSource = BasicUDPSource::createNew(env(), fRTPSocket);
08.             fRTPSource = NULL; // Note!
09.
10.             if (strcmp(fCodecName, "MP2T" ) == 0) { // MPEG-2 Transport Stream
11.                 fReadSource = MPEG2TransportStreamFramer::createNew(env(),
12.                     fReadSource);
13.                 // this sets "durationInMicroseconds" correctly, based on the PCR values
14.             }
15.         } else {
16.             // Check "fCodecName" against the set of codecs that we support,
17.             // and create our RTP source accordingly
18.             // (Later make this code more efficient, as this set grows #####)
19.             // (Also, add more fmts that can be implemented by SimpleRTPSource#####)
20.             Boolean createSimpleRTPSource = False; // by default; can be changed below
21.             Boolean doNormalMBitRule = False; // default behavior if "createSimpleRTPSource" is
                True
22.             if (strcmp(fCodecName, "QCELP" ) == 0) { // QCELP audio
23.                 fReadSource = QCELPAudioRTPSource::createNew(env(), fRTPSocket,
24.                     fRTPSource, fRTPPayloadFormat, fRTPTimestampFrequency);
25.                 // Note that fReadSource will differ from fRTPSource in this case
26.             } else if (strcmp(fCodecName, "AMR" ) == 0) { // AMR audio (narrowband)
27.                 fReadSource = AMRAudioRTPSource::createNew(env(), fRTPSocket,
28.                     fRTPSource, fRTPPayloadFormat, 0 /*isWideband*/ ,
29.                     fNumChannels, fOctetalign, fInterleaving,
30.                     fRobustsorting, fCRC);
31.                 // Note that fReadSource will differ from fRTPSource in this case
32.             } else if (strcmp(fCodecName, "AMR-WB" ) == 0) { // AMR audio (wideband)
33.                 fReadSource = AMRAudioRTPSource::createNew(env(), fRTPSocket,
34.                     fRTPSource, fRTPPayloadFormat, 1 /*isWideband*/ ,
35.                     fNumChannels, fOctetalign, fInterleaving,
36.                     fRobustsorting, fCRC);
37.                 // Note that fReadSource will differ from fRTPSource in this case
38.             } else if (strcmp(fCodecName, "MPA" ) == 0) { // MPEG-1 or 2 audio
39.                 fReadSource = fRTPSource = MPEG1or2AudioRTPSource::createNew(
40.                     env(), fRTPSocket, fRTPPayloadFormat,
41.                     fRTPTimestampFrequency);
42.             } else if (strcmp(fCodecName, "MPA-ROBUST" ) == 0) { // robust MP3 audio
43.                 fRTPSource = MP3ADURTPSource::createNew(env(), fRTPSocket,
44.                     fRTPPayloadFormat, fRTPTimestampFrequency);
45.                 if (fRTPSource == NULL)
46.                     break ;
47.
48.                 // Add a filter that deinterleaves the ADUs after depacketizing them:
49.                 MP3ADUdeinterleaver* deinterleaver = MP3ADUdeinterleaver::createNew(
50.                     env(), fRTPSource);
51.                 if (deinterleaver == NULL)
52.                     break ;
53.
54.                 // Add another filter that converts these ADUs to MP3 frames:
55.                 fReadSource = MP3FromADUSource::createNew(env(), deinterleaver);
56.             } else if (strcmp(fCodecName, "X-MP3-DRAFT-00" ) == 0) {
57.                 // a non-standard variant of "MPA-ROBUST" used by RealNetworks
58.                 // (one 'ADU'ized MP3 frame per packet; no headers)
59.                 fRTPSource = SimpleRTPSource::createNew(env(), fRTPSocket,
60.                     fRTPPayloadFormat, fRTPTimestampFrequency,
61.                     "audio/MPA-ROBUST" /*hack*/ );
62.                 if (fRTPSource == NULL)
63.                     break ;
64.
65.                 // Add a filter that converts these ADUs to MP3 frames:
66.                 fReadSource = MP3FromADUSource::createNew(env(), fRTPSource,
67.                     False /*no ADU header*/ );
68.             } else if (strcmp(fCodecName, "MP4A-LATM" ) == 0) { // MPEG-4 LATM audio
69.                 fReadSource = fRTPSource = MPEG4LATMAudioRTPSource::createNew(
70.                     env(), fRTPSocket, fRTPPayloadFormat,
71.                     fRTPTimestampFrequency);
72.             } else if (strcmp(fCodecName, "AC3" ) == 0
73.                 || strcmp(fCodecName, "EAC3" ) == 0) { // AC3 audio
74.                 fReadSource = fRTPSource = AC3AudioRTPSource::createNew(env(),
75.                     fRTPSocket, fRTPPayloadFormat, fRTPTimestampFrequency);
76.             } else if (strcmp(fCodecName, "MP4V-ES" ) == 0) { // MPEG-4 Elem Str vid
77.                 fReadSource = fRTPSource = MPEG4ESVideoRTPSource::createNew(
78.                     env(), fRTPSocket, fRTPPayloadFormat,
79.                     fRTPTimestampFrequency);
80.             } else if (strcmp(fCodecName, "MPEG4-GENERIC" ) == 0) {

```

```

81. fReadSource = fRTPSource = MPEG4GenericRTPSource::createNew(
82. env(), fRTPSocket, fRTPPayloadFormat,
83. fRTPTimestampFrequency, fMediumName, fMode, fSizeLength,
84. fIndexLength, fIndexDeltaLength);
85. } else if (strcmp(fCodecName, "MPV" ) == 0) { // MPEG-1 or 2 video
86. fReadSource = fRTPSource = MPEG1or2VideoRTPSource::createNew(
87. env(), fRTPSocket, fRTPPayloadFormat,
88. fRTPTimestampFrequency);
89. } else if (strcmp(fCodecName, "MP2T" ) == 0) { // MPEG-2 Transport Stream
90. fRTPSource = SimpleRTPSource::createNew(env(), fRTPSocket,
91. fRTPPayloadFormat, fRTPTimestampFrequency, "video/MP2T" ,
92. 0, False);
93. fReadSource = MPEG2TransportStreamFramer::createNew(env(),
94. fRTPSource);
95. // this sets "durationInMicroseconds" correctly, based on the PCR values
96. } else if (strcmp(fCodecName, "H261" ) == 0) { // H.261
97. fReadSource = fRTPSource = H261VideoRTPSource::createNew(env(),
98. fRTPSocket, fRTPPayloadFormat, fRTPTimestampFrequency);
99. } else if (strcmp(fCodecName, "H263-1998" ) == 0
100. || strcmp(fCodecName, "H263-2000" ) == 0) { // H.263+
101. fReadSource = fRTPSource = H263plusVideoRTPSource::createNew(
102. env(), fRTPSocket, fRTPPayloadFormat,
103. fRTPTimestampFrequency);
104. } else if (strcmp(fCodecName, "H264" ) == 0) {
105. fReadSource = fRTPSource = H264VideoRTPSource::createNew(env(),
106. fRTPSocket, fRTPPayloadFormat, fRTPTimestampFrequency);
107. } else if (strcmp(fCodecName, "DV" ) == 0) {
108. fReadSource = fRTPSource = DVVideoRTPSource::createNew(env(),
109. fRTPSocket, fRTPPayloadFormat, fRTPTimestampFrequency);
110. } else if (strcmp(fCodecName, "JPEG" ) == 0) { // motion JPEG
111. fReadSource = fRTPSource = JPEGVideoRTPSource::createNew(env(),
112. fRTPSocket, fRTPPayloadFormat, fRTPTimestampFrequency,
113. videoWidth(), videoHeight());
114. } else if (strcmp(fCodecName, "X-QT" ) == 0
115. || strcmp(fCodecName, "X-QUICKTIME" ) == 0) {
116. // Generic QuickTime streams, as defined in
117. // <http://developer.apple.com/quicktime/icefloe/dispatch026.html>
118. char * mimeType = new char [strlen(mediumName())
119. + strlen(codecName()) + 2];
120. sprintf(mimeType, "%s/%s" , mediumName(), codecName());
121. fReadSource = fRTPSource = QuickTimeGenericRTPSource::createNew(
122. env(), fRTPSocket, fRTPPayloadFormat,
123. fRTPTimestampFrequency, mimeType);
124. delete [] mimeType;
125. } else if (strcmp(fCodecName, "PCMU" ) == 0 // PCM u-law audio
126. || strcmp(fCodecName, "GSM" ) == 0 // GSM audio
127. || strcmp(fCodecName, "DVI4" ) == 0 // DVI4 (IMA ADPCM) audio
128. || strcmp(fCodecName, "PCMA" ) == 0 // PCM a-law audio
129. || strcmp(fCodecName, "MP1S" ) == 0 // MPEG-1 System Stream
130. || strcmp(fCodecName, "MP2P" ) == 0 // MPEG-2 Program Stream
131. || strcmp(fCodecName, "L8" ) == 0 // 8-bit linear audio
132. || strcmp(fCodecName, "L16" ) == 0 // 16-bit linear audio
133. || strcmp(fCodecName, "L20" ) == 0 // 20-bit linear audio (RFC 3190)
134. || strcmp(fCodecName, "L24" ) == 0 // 24-bit linear audio (RFC 3190)
135. || strcmp(fCodecName, "G726-16" ) == 0 // G.726, 16 kbps
136. || strcmp(fCodecName, "G726-24" ) == 0 // G.726, 24 kbps
137. || strcmp(fCodecName, "G726-32" ) == 0 // G.726, 32 kbps
138. || strcmp(fCodecName, "G726-40" ) == 0 // G.726, 40 kbps
139. || strcmp(fCodecName, "SPEEX" ) == 0 // SPEEX audio
140. || strcmp(fCodecName, "T140" ) == 0 // T.140 text (RFC 4103)
141. || strcmp(fCodecName, "DAT12" ) == 0 // 12-bit nonlinear audio (RFC 3190)
142. ) {
143. createSimpleRTPSource = True;
144. useSpecialRTPOffset = 0;
145. } else if (useSpecialRTPOffset >= 0) {
146. // We don't know this RTP payload format, but try to receive
147. // it using a 'SimpleRTPSource' with the specified header offset:
148. createSimpleRTPSource = True;
149. } else {
150. env().setResultMsg(
151. "RTP payload format unknown or not supported" );
152. break ;
153. }
154.
155. if (createSimpleRTPSource) {
156. char * mimeType = new char [strlen(mediumName())
157. + strlen(codecName()) + 2];
158. sprintf(mimeType, "%s/%s" , mediumName(), codecName());
159. fReadSource = fRTPSource = SimpleRTPSource::createNew(env(),
160. fRTPSocket, fRTPPayloadFormat, fRTPTimestampFrequency,
161. mimeType, (unsigned) useSpecialRTPOffset,
162. doNormalMBitRule);
163. delete [] mimeType;
164. }
165. }
166.
167. return True;
168. } while (0);
169.
170. return False; // an error occurred
171. }

```

可以看到，这个函数里主要是跟据前面分析出的媒体和传输信息建立合适的Source。

socket建立了，Source也创建了，下一步应该是连接Sink，形成一个流。到此为止还未看到Sink的影子，应该是在下一步SETUP中建立，我们看到在continueAfterDESCRIBE()的最后调用了setupStreams ()，那么就探索一下setupStreams():

```
[cpp]
01. void setupStreams()
02. {
03.     static MediaSubsessionIterator* setupIter = NULL;
04.     if (setupIter == NULL)
05.         setupIter = new MediaSubsessionIterator(*session);
06.
07.     //每次调用此函数只为一个Subsession发出SETUP请求。
08.     while ((subsession = setupIter->next()) != NULL) {
09.         // We have another subsession left to set up:
10.         if (subsession->clientPortNum() == 0)
11.             continue ; // port # was not set
12.
13.         //为一个Subsession发送SETUP请求。请求处理完成时调用continueAfterSETUP(),
14.         //continueAfterSETUP()又调用了setupStreams(), 在此函数中为下一个SubSession发送SETUP请求。
```

```
[cpp]
01. <span style="white-space:pre" >         </span> //直到处理完所有的SubSession
02.     setupSubsession(subsession, streamUsingTCP, continueAfterSETUP);
03.     return ;
04. }
05.
06. //执行到这里时，已循环完所有的SubSession了
07. // We're done setting up subsessions.
08. delete setupIter;
09. if (!madeProgress)
10.     shutdown();
11.
12. //创建输出文件，看来是在这里创建Sink了。创建sink后，就开始播放它。这个播放应该只是把socket的handler加入到
13. //计划任务中，而没有数据的接收或发送。只有等到发出PLAY请求后才有数据的收发。
14. // Create output files:
15. if (createReceivers) {
16.     if (outputQuickTimeFile) {
17.         // Create a "QuickTimeFileSink", to write to 'stdout':
18.         qtOut = QuickTimeFileSink::createNew(*env, *session, "stdout",
19.             fileSinkBufferSize, movieWidth, movieHeight, movieFPS,
20.             packetLossCompensate, syncStreams, generateHintTracks,
21.             generateMP4Format);
22.         if (qtOut == NULL) {
23.             *env << "Failed to create QuickTime file sink for stdout: "
24.             << env->getResultMsg();
25.             shutdown();
26.         }
27.
28.         qtOut->startPlaying(sessionAfterPlaying, NULL);
29.     } else if (outputAVIFile) {
30.         // Create an "AVIFileSink", to write to 'stdout':
31.         aviOut = AVIFileSink::createNew(*env, *session, "stdout",
32.             fileSinkBufferSize, movieWidth, movieHeight, movieFPS,
33.             packetLossCompensate);
34.         if (aviOut == NULL) {
35.             *env << "Failed to create AVI file sink for stdout: "
36.             << env->getResultMsg();
37.             shutdown();
38.         }
39.
40.         aviOut->startPlaying(sessionAfterPlaying, NULL);
41.     } else {
42.         // Create and start "FileSink"s for each subsession:
43.         madeProgress = False;
44.         MediaSubsessionIterator iter(*session);
45.         while ((subsession = iter.next()) != NULL) {
46.             if (subsession->readSource() == NULL)
47.                 continue ; // was not initiated
48.
49.             // Create an output file for each desired stream:
50.             char outFileName[1000];
51.             if (singleMedium == NULL) {
52.                 // Output file name is
53.                 // "<filename-prefix><medium_name>-<codec_name>-<counter>"
54.                 static unsigned streamCounter = 0;
55.                 snprintf(outFileName, sizeof outFileName, "%s-%s-%d",
56.                     fileNamePrefix, subsession->mediumName(),
57.                     subsession->codecName(), ++streamCounter);
58.             } else {
59.                 sprintf(outFileName, "stdout");
60.             }
61.             FileSink* fileSink;
62.             if (strcmp(subsession->mediumName(), "audio") == 0
63.                 && (strcmp(subsession->codecName(), "AMR") == 0
64.                     || strcmp(subsession->codecName(), "AMR-WB") == 0))
65.                 -- 011 }
```



```

65.  -- 0, 1
66.  // For AMR audio streams, we use a special sink that inserts AMR frame hdrs:
67.  fileSink = AMRAudioFileSink::createNew(*env, outFileName,
68.  fileSinkBufferSize, oneFilePerFrame);
69.  } else if (strcmp(subsession->mediumName(), "video" ) == 0
70.  && (strcmp(subsession->codecName(), "H264" ) == 0)) {
71.  // For H.264 video stream, we use a special sink that insert start_codes:
72.  fileSink = H264VideoFileSink::createNew(*env, outFileName,
73.  subsession->fmtp_spproparametersets(),
74.  fileSinkBufferSize, oneFilePerFrame);
75.  } else {
76.  // Normal case:
77.  fileSink = FileSink::createNew(*env, outFileName,
78.  fileSinkBufferSize, oneFilePerFrame);
79.  }
80.  subsession->sink = fileSink;
81.  if (subsession->sink == NULL) {
82.  *env << "Failed to create FileSink for \"" << outFileName
83.  << "\": " << env->getResultMsg() << "\n" ;
84.  } else {
85.  if (singleMedium == NULL) {
86.  *env << "Created output file: \"" << outFileName
87.  << "\"\n" ;
88.  } else {
89.  *env << "Outputting data from the \""
90.  << subsession->mediumName() << "/"
91.  << subsession->codecName()
92.  << "\" subsession to 'stdout'\n" ;
93.  }
94.
95.  if (strcmp(subsession->mediumName(), "video" ) == 0
96.  && strcmp(subsession->codecName(), "MP4V-ES" ) == 0 &&
97.  subsession->fmtp_config() != NULL) {
98.  // For MPEG-4 video RTP streams, the 'config' information
99.  // from the SDP description contains useful VOL etc. headers.
100.  // Insert this data at the front of the output file:
101.  unsigned          configLen;
102.  unsigned char * configData
103.  = parseGeneralConfigStr(subsession->fmtp_config(), configLen);
104.  struct timeval timeNow;
105.  gettimeofday(&timeNow, NULL);
106.  fileSink->addData(configData, configLen, timeNow);
107.  delete [] configData;
108.  }
109.
110.  //开始传输
111.  subsession->sink->startPlaying(*(subsession->readSource()),
112.  subsessionAfterPlaying, subsession);
113.
114.  // Also set a handler to be called if a RTCP "BYE" arrives
115.  // for this subsession:
116.  if (subsession->rtcpInstance() != NULL) {
117.  subsession->rtcpInstance()->setByeHandler(
118.  subsessionByeHandler, subsession);
119.  }
120.
121.  madeProgress = True;
122.  }
123.  }
124.  if (!madeProgress)
125.  shutdown();
126.  }
127.  }
128.
129.  // Finally, start playing each subsession, to start the data flow:
130.  if (duration == 0) {
131.  if (scale > 0)
132.  duration = session->playEndTime() - initialSeekTime; // use SDP end time
133.  else if (scale < 0)
134.  duration = initialSeekTime;
135.  }
136.  if (duration < 0)
137.  duration = 0.0;
138.
139.  endTime = initialSeekTime;
140.  if (scale > 0) {
141.  if (duration <= 0)
142.  endTime = -1.0f;
143.  else
144.  endTime = initialSeekTime + duration;
145.  } else {
146.  endTime = initialSeekTime - duration;
147.  if (endTime < 0)
148.  endTime = 0.0f;
149.  }
150.
151.  //发送PLAY请求, 之后才能从Server端接收数据
152.  startPlayingSession(session, initialSeekTime, endTime, scale,
153.  continueAfterPLAY);
154.  }

```


仔细看看注释，应很容易了解此函数。

原文地址：http://blog.csdn.net/niu_gao/article/details/6927461

live555源代码（VC6）：<http://download.csdn.net/detail/leixiaohua1020/6374387>

文章标签：[live555](#) [rtsp](#) [Client](#) [源代码](#)

个人分类：[Live555](#)

所属专栏：[开源多媒体项目源代码分析](#)

此PDF由spygg生成,请尊重原作者版权!!!

我的邮箱:liushidc@163.com