live555学习笔记－RTSPClient分析

2013年09月20日 17:27:32　阅读数：14797

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
void getSDPDescription(RTSPClient::responseHandler* afterFunc)
{
ourRTSPClient->sendDescribeCommand(afterFunc, ourAuthenticator);
}
unsigned RTSPClient::sendDescribeCommand(responseHandler* responseHandler,
Authenticator* authenticator)
{
if (authenticator != NULL)
fCurrentAuthenticator = *authenticator;
return sendRequest( new RequestRecord(++fCSeq, "DESCRIBE" , responseHandler));
}
```

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：

```cpp
void continueAfterDESCRIBE(RTSPClient*, int resultCode, char * resultString)
{
char * sdpDescription = resultString;
//跟据SDP创建MediaSession。
// Create a media session object from this SDP description:
session = MediaSession::createNew(*env, sdpDescription);
delete [] sdpDescription;

// Then, setup the "RTPSource"s for the session:
MediaSubsessionIterator iter(*session);
MediaSubsession *subsession;
Boolean madeProgress = False;
char const * singleMediumToTest = singleMedium;
//循环所有的MediaSubsession，为每个设置其RTPSource的参数
while ((subsession = iter.next()) != NULL) {
//初始化subsession，在其中会建立RTP/RTCP socket以及RTPSource。
if (subsession->initiate(simpleRTPoffsetArg)) {
madeProgress = True;
if (subsession->rtpSource() != NULL) {
// Because we're saving the incoming data, rather than playing
// it in real time, allow an especially large time threshold
// (1 second) for reordering misordered incoming packets:
unsigned const thresh = 1000000; // 1 second
subsession->rtpSource()->setPacketReorderingThresholdTime(thresh);

// Set the RTP source's OS socket buffer size as appropriate - either if we were explicitly asked (using -B),
// or if the desired FileSink buffer size happens to be larger than the current OS socket buffer size.
// (The latter case is a heuristic, on the assumption that if the user asked for a large FileSink buffer size,
// then the input data rate may be large enough to justify increasing the OS socket buffer size also.)
int socketNum = subsession->rtpSource()->RTPgs()->socketNum();
unsigned curBufferSize = getReceiveBufferSize(*env,socketNum);
if (socketInputBufferSize > 0  || fileSinkBufferSize > curBufferSize) {
unsigned newBufferSize =  socketInputBufferSize > 0 ?
socketInputBufferSize : fileSinkBufferSize;
newBufferSize = setReceiveBufferTo(*env, socketNum, newBufferSize);
if (socketInputBufferSize > 0) { // The user explicitly asked for the new socket buffer size; announce it:
*env
<< "Changed socket receive buffer size for the \""
<< subsession->mediumName() << "/"
<< subsession->codecName()
<< "\" subsession from " << curBufferSize
<< " to " << newBufferSize << " bytes\n" ;
}
}
}
}
}
if (!madeProgress)
shutdown();

// Perform additional 'setup' on each subsession, before playing them:
//下一步就是发送SETUP请求了。需要为每个Track分别发送一次。
setupStreams();
}
```

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ClientMediaSesson，SubSession亦不是。我现在很想看看MediaSession与MediaSubsession的建立过程:

```cpp
MediaSession* MediaSession::createNew(UsageEnvironment& env, char const * sdpDescription)
{
MediaSession* newSession = new MediaSession(env);
if (newSession != NULL) {
if (!newSession->initializeWithSDP(sdpDescription)) {
delete newSession;
return NULL;
}
}

return newSession;
}
```

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，我大体说说吧：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:

[cpp]

```cpp
Boolean MediaSubsession::initiate( int useSpecialRTPoffset)
{
if (fReadSource != NULL)
return True; // has already been initiated

do {
if (fCodecName == NULL) {
env().setResultMsg( "Codec is unspecified" );
break ;
}

//创建RTP/RTCP sockets
// Create RTP and RTCP 'Groupsocks' on which to receive incoming data.
// (Groupsocks will work even for unicast addresses)
struct in_addr tempAddr;
tempAddr.s_addr = connectionEndpointAddress();
// This could get changed later, as a result of a RTSP "SETUP"

if (fClientPortNum != 0) {
//当server端指定了建议的client端口
// The sockets' port numbers were specified for us.  Use these:
fClientPortNum = fClientPortNum & ~1; // even
if (isSSM()) {
fRTPSocket = new Groupsock(env(), tempAddr, fSourceFilterAddr,
fClientPortNum);
} else {
fRTPSocket = new Groupsock(env(), tempAddr, fClientPortNum,
255);
}
if (fRTPSocket == NULL) {
env().setResultMsg( "Failed to create RTP socket" );
break ;
}

// Set our RTCP port to be the RTP port +1
portNumBits const rtcpPortNum = fClientPortNum | 1;
if (isSSM()) {
fRTCPSocket = new Groupsock(env(), tempAddr, fSourceFilterAddr,
rtcpPortNum);
} else {
fRTCPSocket = new Groupsock(env(), tempAddr, rtcpPortNum, 255);
}
if (fRTCPSocket == NULL) {
char tmpBuf[100];
sprintf(tmpBuf, "Failed to create RTCP socket (port %d)" ,
rtcpPortNum);
env().setResultMsg(tmpBuf);
break ;
}
} else {
//Server端没有指定client端口，我们自己找一个。之所以做的这样复杂，是为了能找到连续的两个端口
//RTP/RTCP的端口号不是要连续吗？还记得不？
// Port numbers were not specified in advance, so we use ephemeral port numbers.
// Create sockets until we get a port-number pair (even: RTP; even+1: RTCP).
// We need to make sure that we don't keep trying to use the same bad port numbers over and over again.
// so we store bad sockets in a table, and delete them all when we're done.
HashTable* socketHashTable = HashTable::create(ONE_WORD_HASH_KEYS);
if (socketHashTable == NULL)
break ;
Boolean success = False;
NoReuse dummy; // ensures that our new ephemeral port number won't be one that's already in use

while (1) {
// Create a new socket:
if (isSSM()) {
fRTPSocket = new Groupsock(env(), tempAddr,
fSourceFilterAddr, 0);
} else {
fRTPSocket = new Groupsock(env(), tempAddr, 0, 255);
}
if (fRTPSocket == NULL) {
env().setResultMsg(
"MediaSession::initiate(): unable to create RTP and RTCP sockets" );
break ;
}

// Get the client port number, and check whether it's even (for RTP):
Port clientPort(0);
if (!getSourcePort(env(), fRTPSocket->socketNum(),
clientPort)) {
break ;
}
fClientPortNum = ntohs(clientPort.num());
if ((fClientPortNum & 1) != 0) { // it's odd
// Record this socket in our table, and keep trying:
unsigned key = (unsigned) fClientPortNum;
Groupsock* existing = (Groupsock*) socketHashTable->Add(
( char const *) key, fRTPSocket);
```

```
( char const *) key, fRTPSocket);
delete existing; // in case it wasn't NULL
continue ;
}

// Make sure we can use the next (i.e., odd) port number, for RTCP:
portNumBits rtcpPortNum = fClientPortNum | 1;
if (isSSM()) {
fRTCPSocket = new Groupsock(env(), tempAddr,
fSourceFilterAddr, rtcpPortNum);
} else {
fRTCPSocket = new Groupsock(env(), tempAddr, rtcpPortNum,
255);
}
if (fRTCPSocket != NULL && fRTCPSocket->socketNum() >= 0) {
// Success! Use these two sockets.
success = True;
break ;
} else {
// We couldn't create the RTCP socket (perhaps that port number's already in use elsewhere?).
delete fRTCPSocket;

// Record the first socket in our table, and keep trying:
unsigned key = (unsigned) fClientPortNum;
Groupsock* existing = (Groupsock*) socketHashTable->Add(
( char const *) key, fRTPSocket);
delete existing; // in case it wasn't NULL
continue ;
}
}

// Clean up the socket hash table (and contents):
Groupsock* oldGS;
while ((oldGS = (Groupsock*) socketHashTable->RemoveNext()) != NULL) {
delete oldGS;
}
delete socketHashTable;

if (!success)
break ; // a fatal error occurred trying to create the RTP and RTCP sockets; we can't continue
}

// Try to use a big receive buffer for RTP - at least 0.1 second of
// specified bandwidth and at least 50 KB
unsigned rtpBufSize = fBandwidth * 25 / 2; // 1 kbps * 0.1 s = 12.5 bytes
if (rtpBufSize < 50 * 1024)
rtpBufSize = 50 * 1024;
increaseReceiveBufferTo(env(), fRTPSocket->socketNum(), rtpBufSize);

// ASSERT: fRTPSocket != NULL && fRTCPSocket != NULL
if (isSSM()) {
// Special case for RTCP SSM: Send RTCP packets back to the source via unicast:
fRTCPSocket->changeDestinationParameters(fSourceFilterAddr, 0, ~0);
}

//创建RTPSource的地方
// Create "fRTPSource" and "fReadSource":
if (!createSourceObjects(useSpecialRTPoffset))
break ;

if (fReadSource == NULL) {
env().setResultMsg( "Failed to create read source" );
break ;
}

// Finally, create our RTCP instance. (It starts running automatically)
if (fRTPSource != NULL) {
// If bandwidth is specified, use it and add 5% for RTCP overhead.
// Otherwise make a guess at 500 kbps.
unsigned totSessionBandwidth =
fBandwidth ? fBandwidth + fBandwidth / 20 : 500;
fRTCPInstance = RTCPInstance::createNew(env(), fRTCPSocket,
totSessionBandwidth, (unsigned char const *) fParent.CNAME(),
NULL /* we're a client */ , fRTPSource);
if (fRTCPInstance == NULL) {
env().setResultMsg( "Failed to create RTCP instance" );
break ;
}
}

return True;
} while (0);

//失败时执行到这里
delete fRTPSocket;
fRTPSocket = NULL;
delete fRTCPSocket;
fRTCPSocket = NULL;
Medium::close(fRTCPInstance);
fRTCPInstance = NULL;
```

```
Medium::close(fReadSource);
fReadSource = fRTPSource = NULL;
fClientPortNum = 0;
return False;
}
```

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]

```cpp
Boolean MediaSubsession::createSourceObjects( int useSpecialRTPoffset)
{
do {
// First, check "fProtocolName"
if (strcmp(fProtocolName, "UDP" ) == 0) {
// A UDP-packetized stream (*not* a RTP stream)
fReadSource = BasicUDPSource::createNew(env(), fRTPSocket);
fRTPSource = NULL; // Note!

if (strcmp(fCodecName, "MP2T" ) == 0) { // MPEG-2 Transport Stream
fReadSource = MPEG2TransportStreamFramer::createNew(env(),
fReadSource);
// this sets "durationInMicroseconds" correctly, based on the PCR values
}
} else {
// Check "fCodecName" against the set of codecs that we support,
// and create our RTP source accordingly
// (Later make this code more efficient, as this set grows #####)
// (Also, add more fmts that can be implemented by SimpleRTPSource#####)
Boolean createSimpleRTPSource = False; // by default; can be changed below
Boolean doNormalMBitRule = False; // default behavior if "createSimpleRTPSource" is True
if (strcmp(fCodecName, "QCELP" ) == 0) { // QCELP audio
fReadSource = QCELPAudioRTPSource::createNew(env(), fRTPSocket,
fRTPSource, fRTPPayloadFormat, fRTPTimestampFrequency);
// Note that fReadSource will differ from fRTPSource in this case
} else if (strcmp(fCodecName, "AMR" ) == 0) { // AMR audio (narrowband)
fReadSource = AMRAudioRTPSource::createNew(env(), fRTPSocket,
fRTPSource, fRTPPayloadFormat, 0 /*isWideband*/ ,
fNumChannels, fOctetalign, fInterleaving,
fRobustsorting, fCRC);
// Note that fReadSource will differ from fRTPSource in this case
} else if (strcmp(fCodecName, "AMR-WB" ) == 0) { // AMR audio (wideband)
fReadSource = AMRAudioRTPSource::createNew(env(), fRTPSocket,
fRTPSource, fRTPPayloadFormat, 1 /*isWideband*/ ,
fNumChannels, fOctetalign, fInterleaving,
fRobustsorting, fCRC);
// Note that fReadSource will differ from fRTPSource in this case
} else if (strcmp(fCodecName, "MPA" ) == 0) { // MPEG-1 or 2 audio
fReadSource = fRTPSource = MPEG1or2AudioRTPSource::createNew(
env(), fRTPSocket, fRTPPayloadFormat,
fRTPTimestampFrequency);
} else if (strcmp(fCodecName, "MPA-ROBUST" ) == 0) { // robust MP3 audio
fRTPSource = MP3ADURTPSource::createNew(env(), fRTPSocket,
fRTPPayloadFormat, fRTPTimestampFrequency);
if (fRTPSource == NULL)
break ;

// Add a filter that deinterleaves the ADUs after depacketizing them:
MP3ADUdeinterleaver* deinterleaver = MP3ADUdeinterleaver::createNew(
env(), fRTPSource);
if (deinterleaver == NULL)
break ;

// Add another filter that converts these ADUs to MP3 frames:
fReadSource = MP3FromADUSource::createNew(env(), deinterleaver);
} else if (strcmp(fCodecName, "X-MP3-DRAFT-00" ) == 0) {
// a non-standard variant of "MPA-ROBUST" used by RealNetworks
// (one 'ADU'ized MP3 frame per packet; no headers)
fRTPSource = SimpleRTPSource::createNew(env(), fRTPSocket,
fRTPPayloadFormat, fRTPTimestampFrequency,
"audio/MPA-ROBUST" /*hack*/ );
if (fRTPSource == NULL)
break ;

// Add a filter that converts these ADUs to MP3 frames:
fReadSource = MP3FromADUSource::createNew(env(), fRTPSource,
False /*no ADU header*/ );
} else if (strcmp(fCodecName, "MP4A-LATM" ) == 0) { // MPEG-4 LATM audio
fReadSource = fRTPSource = MPEG4LATMAudioRTPSource::createNew(
env(), fRTPSocket, fRTPPayloadFormat,
fRTPTimestampFrequency);
} else if (strcmp(fCodecName, "AC3" ) == 0
|| strcmp(fCodecName, "EAC3" ) == 0) { // AC3 audio
fReadSource = fRTPSource = AC3AudioRTPSource::createNew(env(),
fRTPSocket, fRTPPayloadFormat, fRTPTimestampFrequency);
} else if (strcmp(fCodecName, "MP4V-ES" ) == 0) { // MPEG-4 Elem Str vid
fReadSource = fRTPSource = MPEG4ESVideoRTPSource::createNew(
env(), fRTPSocket, fRTPPayloadFormat,
fRTPTimestampFrequency);
} else if (strcmp(fCodecName, "MPEG4-GENERIC" ) == 0) {
```

```
fReadSource = fRTPSource = MPEG4GenericRTPSource::createNew(
env(), fRTPSocket, fRTPPayloadFormat,
fRTPTimestampFrequency, fMediumName, fMode, fSizelength,
fIndexlength, fIndexdeltalength);
} else if (strcmp(fCodecName, "MPV" ) == 0) { // MPEG-1 or 2 video
fReadSource = fRTPSource = MPEG1or2VideoRTPSource::createNew(
env(), fRTPSocket, fRTPPayloadFormat,
fRTPTimestampFrequency);
} else if (strcmp(fCodecName, "MP2T" ) == 0) { // MPEG-2 Transport Stream
fRTPSource = SimpleRTPSource::createNew(env(), fRTPSocket,
fRTPPayloadFormat, fRTPTimestampFrequency, "video/MP2T" ,
0, False);
fReadSource = MPEG2TransportStreamFramer::createNew(env(),
fRTPSource);
// this sets "durationInMicroseconds" correctly, based on the PCR values
} else if (strcmp(fCodecName, "H261" ) == 0) { // H.261
fReadSource = fRTPSource = H261VideoRTPSource::createNew(env(),
fRTPSocket, fRTPPayloadFormat, fRTPTimestampFrequency);
} else if (strcmp(fCodecName, "H263-1998" ) == 0
|| strcmp(fCodecName, "H263-2000" ) == 0) { // H.263+
fReadSource = fRTPSource = H263plusVideoRTPSource::createNew(
env(), fRTPSocket, fRTPPayloadFormat,
fRTPTimestampFrequency);
} else if (strcmp(fCodecName, "H264" ) == 0) {
fReadSource = fRTPSource = H264VideoRTPSource::createNew(env(),
fRTPSocket, fRTPPayloadFormat, fRTPTimestampFrequency);
} else if (strcmp(fCodecName, "DV" ) == 0) {
fReadSource = fRTPSource = DVVideoRTPSource::createNew(env(),
fRTPSocket, fRTPPayloadFormat, fRTPTimestampFrequency);
} else if (strcmp(fCodecName, "JPEG" ) == 0) { // motion JPEG
fReadSource = fRTPSource = JPEGVideoRTPSource::createNew(env(),
fRTPSocket, fRTPPayloadFormat, fRTPTimestampFrequency,
videoWidth(), videoHeight());
} else if (strcmp(fCodecName, "X-QT" ) == 0
|| strcmp(fCodecName, "X-QUICKTIME" ) == 0) {
// Generic QuickTime streams, as defined in
// <http://developer.apple.com/quicktime/icefloe/dispatch026.html>
char * mimeType = new char [strlen(mediumName())
+ strlen(codecName()) + 2];
sprintf(mimeType, "%s/%s" , mediumName(), codecName());
fReadSource = fRTPSource = QuickTimeGenericRTPSource::createNew(
env(), fRTPSocket, fRTPPayloadFormat,
fRTPTimestampFrequency, mimeType);
delete [] mimeType;
} else if (strcmp(fCodecName, "PCMU" ) == 0 // PCM u-law audio
|| strcmp(fCodecName, "GSM" ) == 0 // GSM audio
|| strcmp(fCodecName, "DVI4" ) == 0 // DVI4 (IMA ADPCM) audio
|| strcmp(fCodecName, "PCMA" ) == 0 // PCM a-law audio
|| strcmp(fCodecName, "MP1S" ) == 0 // MPEG-1 System Stream
|| strcmp(fCodecName, "MP2P" ) == 0 // MPEG-2 Program Stream
|| strcmp(fCodecName, "L8" ) == 0 // 8-bit linear audio
|| strcmp(fCodecName, "L16" ) == 0 // 16-bit linear audio
|| strcmp(fCodecName, "L20" ) == 0 // 20-bit linear audio (RFC 3190)
|| strcmp(fCodecName, "L24" ) == 0 // 24-bit linear audio (RFC 3190)
|| strcmp(fCodecName, "G726-16" ) == 0 // G.726, 16 kbps
|| strcmp(fCodecName, "G726-24" ) == 0 // G.726, 24 kbps
|| strcmp(fCodecName, "G726-32" ) == 0 // G.726, 32 kbps
|| strcmp(fCodecName, "G726-40" ) == 0 // G.726, 40 kbps
|| strcmp(fCodecName, "SPEEX" ) == 0 // SPEEX audio
|| strcmp(fCodecName, "T140" ) == 0 // T.140 text (RFC 4103)
|| strcmp(fCodecName, "DAT12" ) == 0 // 12-bit nonlinear audio (RFC 3190)
) {
createSimpleRTPSource = True;
useSpecialRTPoffset = 0;
} else if (useSpecialRTPoffset >= 0) {
// We don't know this RTP payload format, but try to receive
// it using a 'SimpleRTPSource' with the specified header offset:
createSimpleRTPSource = True;
} else {
env().setResultMsg(
"RTP payload format unknown or not supported" );
break ;
}

if (createSimpleRTPSource) {
char * mimeType = new char [strlen(mediumName())
+ strlen(codecName()) + 2];
sprintf(mimeType, "%s/%s" , mediumName(), codecName());
fReadSource = fRTPSource = SimpleRTPSource::createNew(env(),
fRTPSocket, fRTPPayloadFormat, fRTPTimestampFrequency,
mimeType, (unsigned) useSpecialRTPoffset,
doNormalMBitRule);
delete [] mimeType;
}
}

return True;
} while (0);

return False; // an error occurred
}
```

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（），那么就来探索一下setupStreams():

[cpp]

```cpp
void setupStreams()
{
static MediaSubsessionIterator* setupIter = NULL;
if (setupIter == NULL)
setupIter = new MediaSubsessionIterator(*session);

//每次调用此函数只为一个Subsession发出SETUP请求。
while ((subsession = setupIter->next()) != NULL) {
// We have another subsession left to set up:
if (subsession->clientPortNum() == 0)
continue ; // port # was not set

//为一个Subsession发送SETUP请求。请求处理完成时调用continueAfterSETUP(),
//continueAfterSETUP()又调用了setupStreams()，在此函数中为下一个SubSession发送SETUP请求。
```

[cpp]

```cpp
<span style= "white-space:pre" >      </span> //直到处理完所有的SubSession
setupSubsession(subsession, streamUsingTCP, continueAfterSETUP);
return ;
}

//执行到这里时，已循环完所有的SubSession了
// We're done setting up subsessions.
delete setupIter;
if (!madeProgress)
shutdown();

//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
//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。
// Create output files:
if (createReceivers) {
if (outputQuickTimeFile) {
// Create a "QuickTimeFileSink", to write to 'stdout':
qtOut = QuickTimeFileSink::createNew(*env, *session, "stdout" ,
fileSinkBufferSize, movieWidth, movieHeight, movieFPS,
packetLossCompensate, syncStreams, generateHintTracks,
generateMP4Format);
if (qtOut == NULL) {
*env << "Failed to create QuickTime file sink for stdout: "
<< env->getResultMsg();
shutdown();
}

qtOut->startPlaying(sessionAfterPlaying, NULL);
} else if (outputAVIFile) {
// Create an "AVIFileSink", to write to 'stdout':
aviOut = AVIFileSink::createNew(*env, *session, "stdout" ,
fileSinkBufferSize, movieWidth, movieHeight, movieFPS,
packetLossCompensate);
if (aviOut == NULL) {
*env << "Failed to create AVI file sink for stdout: "
<< env->getResultMsg();
shutdown();
}

aviOut->startPlaying(sessionAfterPlaying, NULL);
} else {
// Create and start "FileSink"s for each subsession:
madeProgress = False;
MediaSubsessionIterator iter(*session);
while ((subsession = iter.next()) != NULL) {
if (subsession->readSource() == NULL)
continue ; // was not initiated

// Create an output file for each desired stream:
char outFileName[1000];
if (singleMedium == NULL) {
// Output file name is
//     "<filename-prefix><medium_name>-<codec_name>-<counter>"
static unsigned streamCounter = 0;
snprintf(outFileName, sizeof outFileName, "%s%s-%s-%d" ,
fileNamePrefix, subsession->mediumName(),
subsession->codecName(), ++streamCounter);
} else {
sprintf(outFileName, "stdout" );
}
FileSink* fileSink;
if (strcmp(subsession->mediumName(), "audio" ) == 0
&& (strcmp(subsession->codecName(), "AMR" ) == 0
|| strcmp(subsession->codecName(), "AMR-WB" )
== 0)) {
```

```
== 0)) {
// For AMR audio streams, we use a special sink that inserts AMR frame hdrs:
fileSink = AMRAudioFileSink::createNew(*env, outFileName,
fileSinkBufferSize, oneFilePerFrame);
} else if (strcmp(subsession->mediumName(), "video" ) == 0
&& (strcmp(subsession->codecName(), "H264" ) == 0)) {
// For H.264 video stream, we use a special sink that insert start_codes:
fileSink = H264VideoFileSink::createNew(*env, outFileName,
subsession->fmtp_spropparametersets(),
fileSinkBufferSize, oneFilePerFrame);
} else {
// Normal case:
fileSink = FileSink::createNew(*env, outFileName,
fileSinkBufferSize, oneFilePerFrame);
}
subsession->sink = fileSink;
if (subsession->sink == NULL) {
*env << "Failed to create FileSink for \"" << outFileName
<< "\": " << env->getResultMsg() << "\n" ;
} else {
if (singleMedium == NULL) {
*env << "Created output file: \"" << outFileName
<< "\"\n" ;
} else {
*env << "Outputting data from the \""
<< subsession->mediumName() << "/"
<< subsession->codecName()
<< "\" subsession to 'stdout'\n" ;
}

if (strcmp(subsession->mediumName(), "video" ) == 0
&& strcmp(subsession->codecName(), "MP4V-ES" ) == 0 &&
subsession->fmtp_config() != NULL) {
// For MPEG-4 video RTP streams, the 'config' information
// from the SDP description contains useful VOL etc. headers.
// Insert this data at the front of the output file:
unsigned                    configLen;
unsigned char * configData
= parseGeneralConfigStr(subsession->fmtp_config(), configLen);
struct timeval timeNow;
gettimeofday(&timeNow, NULL);
fileSink->addData(configData, configLen, timeNow);
delete [] configData;
}

//开始传输
subsession->sink->startPlaying(*(subsession->readSource()),
subsessionAfterPlaying, subsession);

// Also set a handler to be called if a RTCP "BYE" arrives
// for this subsession:
if (subsession->rtcpInstance() != NULL) {
subsession->rtcpInstance()->setByeHandler(
subsessionByeHandler, subsession);
}

madeProgress = True;
}
}
if (!madeProgress)
shutdown();
}
}

// Finally, start playing each subsession, to start the data flow:
if (duration == 0) {
if (scale > 0)
duration = session->playEndTime() - initialSeekTime; // use SDP end time
else if (scale < 0)
duration = initialSeekTime;
}
if (duration < 0)
duration = 0.0;

endTime = initialSeekTime;
if (scale > 0) {
if (duration <= 0)
endTime = -1.0f;
else
endTime = initialSeekTime + duration;
} else {
endTime = initialSeekTime - duration;
if (endTime < 0)
endTime = 0.0f;
}

//发送PLAY请求，之后才能从Server端接收数据
startPlayingSession(session, initialSeekTime, endTime, scale,
continueAfterPLAY);
}
```

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