# 1－引子

一直想研究live555，没有时间，终于因为项目的原因可以深入无间地研究一下了．所以在此著文以记之．

一　如何编译live555

利用mingw环境很容易：在live555文件夹下，

genMakefiles mingw

make

即可．

可以用genWindowsMakefiles.cmd生成VC可用的makefile，但是对比较新的vc版本支持不好，需要自己改很多东西．

用VC编译有一种更好的办法：

手动为每个库都生成一个lib项目，为mediaServer生成exe项目，设置好各库之间的依赖关系，就可以用VC编译了．由于live555代码中没有单独支持gcc的东西，所以编译是相当的容易．这样就可以用VC编译和调试了．

我现在怕麻烦，只用eclipse+mingw．eclipse的调试也很好用了．

# 二　基础类

讲几个重要的基础类：

BasicUsageEnvironment和UsageEnvironment中的类都是用于整个系统的基础功能类．比如UsageEnvironment代表了整个系统运行的环境，它提供了错误记录和错误报告的功能，无论哪一个类要输出错误，就需要保存UsageEnvironment的指针．而TaskScheduler则提供了任务调度功能．整个程序的运行发动机就是它，它调度任务，执行任务（任务就是一个函数）．TaskScheduler由于在全局中只有一个，所以保存在了UsageEnvironment中．而所有的类又都保存了UsageEnvironment的指针，所以谁想把自己的任务加入调度中，那是很容易的．在此还看到一个结论：整个live555（服务端）只有一个线程．

类HashTable：不用多说，实现了哈稀表．

类DelayQueue：译为＂延迟队列＂，它是一个队列，每一项代表了一个要调度的任务（在它的fToken变量中保存）．同时保存了这个任务离执行时间点的剩余时间．可以预见，它就是在TaskScheduler中用于管理调度任务的东西．注意，此队列中的任务只被执行一次！执行完后这一项即被无情抛弃！

类HandlerSet：Handler集合．Handler是什么呢？它是一种专门用于执行socket操作的任务（函数），HandlerSet被TaskScheduler用来管理所有的socket任务（增删改查）．所以TaskScheduler中现在已调度两种任务了：socket任务（handlerSet）和延迟任务(DelayQueue)．其实TaskScheduler还调度第三种任务：Event，介个后面再说．

类Groupsock：这个是放在单独的库Groupsock中。它封装了socket操作，增加了多播放支持和一对多单播的功能．但我只看到它对UDP的支持，好像不支持TCP。它管理着一个本地socket和多个目的地址，因为是UDP，所以只需知道对方地址和端口即可发送数据。Groupsock的构造函数有一个参数是struct in\_addr const& groupAddr，在构造函数中首先会调用父类构造函数创建socket对象，然后判断这个地址，若是多播地址，则加入多播组。Groupsock的两个成员变量destRecord\* fDests和DirectedNetInterfaceSet fMembers都表示目的地址集和，但我始终看不出DirectedNetInterfaceSet fMembers有什么用，且DirectedNetInterfaceSet是一个没有被继承的虚类，看起来fMembers没有什么用。仅fDesk也够用了，在addDestination()和removeDestination()函数中就是操作fDesk，添加或删除目的地址。

解释一下Groupsock::changeDestinationParameters()函数：

1. //改变目的地址的参数
2. //newDestAddr是新的目的地址
3. //newDestPort是新的目的端口
4. //newDestTTL是新的TTL
5. void Groupsock::changeDestinationParameters(
6. struct in\_addr const& newDestAddr,
7. Port newDestPort,
8. **int** newDestTTL)
9. {
10. if (fDests == NULL)
11. return;
13. //获取第一个目的地址（此处不是很明白：fDest是一个单向链表，每次添加一个目的地
14. //址， 都会把它插入到最前目，难道这个函数仅改变最后一个添加的目的地址？）
15. struct in\_addr destAddr = fDests->fGroupEId.groupAddress();
16. if (newDestAddr.s\_addr != 0) {
17. if (newDestAddr.s\_addr != destAddr.s\_addr
18. && IsMulticastAddress(newDestAddr.s\_addr))
19. {
20. //如果目的地址是一个多播地址，则离开老的多播组，加入新的多播组。
21. socketLeaveGroup(env(), socketNum(), destAddr.s\_addr);
22. socketJoinGroup(env(), socketNum(), newDestAddr.s\_addr);
23. }
24. destAddr.s\_addr = newDestAddr.s\_addr;
25. }
27. portNumBits destPortNum = fDests->fGroupEId.portNum();
28. if (newDestPort.num() != 0) {
29. if (newDestPort.num() != destPortNum &&
30. IsMulticastAddress(destAddr.s\_addr))
31. {
32. //如果端口也不一样，则先更改本身socket的端口
33. //（其实是关掉原先的socket的，再以新端口打开一个socket）。
34. changePort(newDestPort);
35. //然后把新的socket加入到新的多播组。
36. // And rejoin the multicast group:
37. socketJoinGroup(env(), socketNum(), destAddr.s\_addr);
38. }
39. destPortNum = newDestPort.num();
40. fDests->fPort = newDestPort;
41. }
43. u\_int8\_t destTTL = ttl();
44. if (newDestTTL != ~0)
45. destTTL = (u\_int8\_t) newDestTTL;
47. //目标地址的所有信息都在fGroupEId中，所以改变成员fGroupEId。
48. fDests->fGroupEId = GroupEId(destAddr, destPortNum, destTTL);
50. //(看起来这个函数好像只用于改变多播时的地址参数，
51. //以上分析是否合理，肯请高人指点)
52. }

# 三 消息循环

看服端的主体：live555MediaServer.cpp中的main()函数，可见其创建一个RTSPServer类实例后，即进入一个函数env->taskScheduler().doEventLoop()中，看名字很明显是一个消息循坏，执行到里面后不停地转圈，生名不息，转圈不止。那么在这个人生的圈圈中如何实现RTSP服务和RTP传输呢？别想那么远了，还是先看这个圈圈中实现了什么功能吧。

1. void BasicTaskScheduler0::doEventLoop(**char**\* watchVariable) {
2. // Repeatedly loop, handling readble sockets and timed events:
3. while (1) {
4. if (watchVariable != NULL && \*watchVariable != 0)
5. break;
6. SingleStep();
7. }
8. }

BasicTaskScheduler0从TaskScheduler派生，所以还是一个任务调度对象，所以依然说明任务调度对象是整个程序的发动机。

循环中每次走一步：SingleStep()。这走一步中都做些什么呢？

总结为以下四步：

１为所有需要操作的socket执行select。

２找出第一个应执行的socket任务(handler)并执行之。

３找到第一个应响应的事件，并执行之。

４找到第一个应执行的延迟任务并执行之。

可见，每一步中只执行三个任务队列中的一项。下面详细分析函数SingleStep()：

1. //循坏中主要执行的函数
2. void BasicTaskScheduler::SingleStep(unsigned maxDelayTime) {
3. fd\_set readSet = fReadSet; // make a copy for this select() call
4. fd\_set writeSet = fWriteSet; // ditto
5. fd\_set exceptionSet = fExceptionSet; // ditto
7. //计算select socket们时的超时时间。
8. DelayInterval const& timeToDelay = fDelayQueue.timeToNextAlarm();
9. struct timeval tv\_timeToDelay;
10. tv\_timeToDelay.tv\_sec = timeToDelay.seconds();
11. tv\_timeToDelay.tv\_usec = timeToDelay.useconds();
12. // Very large "tv\_sec" values cause select() to fail.
13. // Don't make it any larger than 1 million seconds (11.5 days)
14. const **long** MAX\_TV\_SEC = MILLION;
15. if (tv\_timeToDelay.tv\_sec > MAX\_TV\_SEC) {
16. tv\_timeToDelay.tv\_sec = MAX\_TV\_SEC;
17. }
18. // Also check our "maxDelayTime" parameter (if it's > 0):
19. if (maxDelayTime > 0
20. && (tv\_timeToDelay.tv\_sec > (**long**) maxDelayTime / MILLION
21. || (tv\_timeToDelay.tv\_sec == (**long**) maxDelayTime / MILLION
22. && tv\_timeToDelay.tv\_usec  > (**long**) maxDelayTime % MILLION))) {
23. tv\_timeToDelay.tv\_sec = maxDelayTime / MILLION;
24. tv\_timeToDelay.tv\_usec = maxDelayTime % MILLION;
25. }
27. //先执行socket的select操作，以确定哪些socket任务（handler）需要执行。
28. **int** selectResult = select(fMaxNumSockets,
29. &readSet, &writeSet,&exceptionSet,
30. &tv\_timeToDelay);
32. if (selectResult < 0) {
33. //#if defined(\_\_WIN32\_\_) || defined(\_WIN32)
34. **int** err = WSAGetLastError();
35. // For some unknown reason, select() in Windoze sometimes fails with WSAEINVAL if
36. // it was called with no entries set in "readSet".  If this happens, ignore it
37. if (err == WSAEINVAL && readSet.fd\_count == 0) {
38. err = EINTR;
39. // To stop this from happening again, create a dummy socket:
40. **int** dummySocketNum = socket(AF\_INET, SOCK\_DGRAM, 0);
41. FD\_SET((unsigned) dummySocketNum, &fReadSet);
42. }
43. if (err != EINTR) {
44. internalError();
45. }
46. }
48. // Call the handler function for one readable socket:
49. HandlerIterator iter(\*fHandlers);
50. HandlerDescriptor\* handler;
51. // To ensure forward progress through the handlers, begin past the last
52. // socket number that we handled:
53. if (fLastHandledSocketNum >= 0) {
54. //找到上次执行的socket handler的下一个
55. while ((handler = iter.next()) != NULL) {
56. if (handler->socketNum == fLastHandledSocketNum)
57. break;
58. }
59. if (handler == NULL) {
60. fLastHandledSocketNum = -1;
61. iter.reset(); // start from the beginning instead
62. }
63. }
65. //从找到的handler开始，找一个可以执行的handler，不论其状态是可读，可写，还是出错，执行之。
66. while ((handler = iter.next()) != NULL) {
67. **int** sock = handler->socketNum; // alias
68. **int** resultConditionSet = 0;
69. if (FD\_ISSET(sock, &readSet)
70. && FD\_ISSET(sock, &fReadSet)/\*sanity check\*/)
71. resultConditionSet |= SOCKET\_READABLE;
72. if (FD\_ISSET(sock, &writeSet)
73. && FD\_ISSET(sock, &fWriteSet)/\*sanity check\*/)
74. resultConditionSet |= SOCKET\_WRITABLE;
75. if (FD\_ISSET(sock, &exceptionSet)
76. && FD\_ISSET(sock, &fExceptionSet)/\*sanity check\*/)
77. resultConditionSet |= SOCKET\_EXCEPTION;
78. if ((resultConditionSet & handler->conditionSet)
79. != 0 && handler->handlerProc != NULL) {
80. fLastHandledSocketNum = sock;
81. // Note: we set "fLastHandledSocketNum" before calling the handler,
82. // in case the handler calls "doEventLoop()" reentrantly.
83. (\*handler->handlerProc)(handler->clientData, resultConditionSet);
84. break;
85. }
86. }
88. //如果寻找完了依然没有执行任何handle，则从头再找。
89. if (handler == NULL && fLastHandledSocketNum >= 0) {
90. // We didn't call a handler, but we didn't get to check all of them,
91. // so try again from the beginning:
92. iter.reset();
93. while ((handler = iter.next()) != NULL) {
94. **int** sock = handler->socketNum; // alias
95. **int** resultConditionSet = 0;
96. if (FD\_ISSET(sock, &readSet)&& FD\_ISSET(sock, &fReadSet))
97. resultConditionSet |= SOCKET\_READABLE;
98. if (FD\_ISSET(sock, &writeSet)&& FD\_ISSET(sock, &fWriteSet))
99. resultConditionSet |= SOCKET\_WRITABLE;
100. if (FD\_ISSET(sock, &exceptionSet)&& FD\_ISSET(sock, &fExceptionSet))
101. resultConditionSet |= SOCKET\_EXCEPTION;
102. if ((resultConditionSet & handler->conditionSet)
103. != 0 && handler->handlerProc != NULL) {
104. fLastHandledSocketNum = sock;
105. // Note: we set "fLastHandledSocketNum" before calling the handler,
106. // in case the handler calls "doEventLoop()" reentrantly.
107. (\*handler->handlerProc)(handler->clientData, resultConditionSet);
108. break;
109. }
110. }
112. //依然没有找到可执行的handler。
113. if (handler == NULL)
114. fLastHandledSocketNum = -1; //because we didn't call a handler
115. }
117. //响应事件
118. // Also handle any newly-triggered event
119. // (Note that we do this \*after\* calling a socket handler,
120. // in case the triggered event handler modifies The set of readable sockets.)
121. if (fTriggersAwaitingHandling != 0) {
122. if (fTriggersAwaitingHandling == fLastUsedTriggerMask) {
123. // Common-case optimization for a single event trigger:
124. fTriggersAwaitingHandling = 0;
125. if (fTriggeredEventHandlers[fLastUsedTriggerNum] != NULL) {
126. //执行一个事件处理函数
127. (\*fTriggeredEventHandlers[fLastUsedTriggerNum])(fTriggeredEventClientDatas[fLastUsedTriggerNum]);
128. }
129. } else {
130. // Look for an event trigger that needs handling
131. // (making sure that we make forward progress through all possible triggers):
132. unsigned i = fLastUsedTriggerNum;
133. EventTriggerId mask = fLastUsedTriggerMask;
135. do {
136. i = (i + 1) % MAX\_NUM\_EVENT\_TRIGGERS;
137. mask >>= 1;
138. if (mask == 0)
139. mask = 0x80000000;
141. if ((fTriggersAwaitingHandling & mask) != 0) {
142. //执行一个事件响应
143. fTriggersAwaitingHandling &= ~mask;
144. if (fTriggeredEventHandlers[i] != NULL) {
145. (\*fTriggeredEventHandlers[i])(fTriggeredEventClientDatas[i]);
146. }
148. fLastUsedTriggerMask = mask;
149. fLastUsedTriggerNum = i;
150. break;
151. }
152. } while (i != fLastUsedTriggerNum);
153. }
154. }
156. //执行一个最迫切的延迟任务。
157. // Also handle any delayed event that may have come due.
158. fDelayQueue.handleAlarm();
159. }

# 四 计划任务(TaskScheduler)深入探讨

我们且把三种任务命名为：**socket handler**,**event handler**,**delay task**。

这三种任务的特点是，前两个加入执行队列后会一直存在，而delay task在执行完一次后会立即弃掉。

socket handler保存在队列BasicTaskScheduler0::HandlerSet\* fHandlers中;

event handler保存在数组BasicTaskScheduler0::TaskFunc \* fTriggeredEventHandlers[MAX\_NUM\_EVENT\_TRIGGERS]中;

delay task保存在队列BasicTaskScheduler0::DelayQueue fDelayQueue中。

下面看一下三种任务的执行函数的定义：  
socket handler为  
typedef void BackgroundHandlerProc(void\* clientData, int mask);  
event handler为  
typedef void TaskFunc(void\* clientData);  
delay task 为  
typedef void TaskFunc(void\* clientData);//跟event handler一样。

再看一下向任务调度对象添加三种任务的函数的样子：  
delay task为：  
void setBackgroundHandling(int socketNum, int conditionSet　,BackgroundHandlerProc\* handlerProc, void\* clientData)  
event handler为:  
EventTriggerId createEventTrigger(TaskFunc\* eventHandlerProc)  
delay task为：  
TaskToken scheduleDelayedTask(int64\_t microseconds, TaskFunc\* proc,void\* clientData)

socket handler添加时为什么需要那些参数呢？socketNum是需要的，因为要select socket（socketNum即是socket()返回的那个socket对象）。conditionSet也是需要的，它用于表明socket在select时查看哪种装态，是可读？可写？还是出错？proc和clientData这两个参数就不必说了（真有不明白的吗？）。再看BackgroundHandlerProc的参数，clientData不必解释，mask是什么呢？它正是对应着conditionSet，但它表明的是select之后的结果，比如一个socket可能需要检查其读/写状态，而当前只能读，不能写，那么mask中就只有表明读的位被设置。

event handler是被存在数组中。数组大小固定，是32项，用EventTriggerId来表示数组中的项，EventTriggerId是一个32位整数，因为数组是32项，所以用EventTriggerId中的第n位置１表明对应数组中的第n项。成员变量fTriggersAwaitingHandling也是EventTriggerId类型，它里面置1的那些位对应了数组中所有需要处理的项。这样做节省了内存和计算，但降低了可读性，呵呵，而且也不够灵活，只能支持32项或64项，其它数量不被支持。以下是函数体

1. EventTriggerId BasicTaskScheduler0::createEventTrigger( TaskFunc\* eventHandlerProc)
2. {
3. unsigned i = fLastUsedTriggerNum;
4. EventTriggerId mask = fLastUsedTriggerMask;
6. //在数组中寻找一个未使用的项，把eventHandlerProc分配到这一项。
7. do {
8. i = (i + 1) % MAX\_NUM\_EVENT\_TRIGGERS;
9. mask >>= 1;
10. if (mask == 0)
11. mask = 0x80000000;
13. if (fTriggeredEventHandlers[i] == NULL) {
14. // This trigger number is free; use it:
15. fTriggeredEventHandlers[i] = eventHandlerProc;
16. fTriggeredEventClientDatas[i] = NULL; // sanity
18. fLastUsedTriggerMask = mask;
19. fLastUsedTriggerNum = i;
21. return mask; //分配成功，返回值表面了第几项
22. }
23. } while (i != fLastUsedTriggerNum);//表明在数组中循环一圈
25. //数组中的所有项都被占用，返回表明失败。
26. // All available event triggers are allocated; return 0 instead:
27. return 0;
28. }

可以看到最多添加32个事件，且添加事件时没有传入clientData参数。这个参数在触发事件时传入，见以下函数：

1. void BasicTaskScheduler0::triggerEvent(EventTriggerId eventTriggerId,void\* clientData)
2. {
3. // First, record the "clientData":
4. if (eventTriggerId == fLastUsedTriggerMask) {
5. // common-case optimization:直接保存下clientData
6. fTriggeredEventClientDatas[fLastUsedTriggerNum] = clientData;
7. } else {
8. //从头到尾查找eventTriggerId对应的项，保存下clientData
9. EventTriggerId mask = 0x80000000;
10. for (unsigned i = 0; i < MAX\_NUM\_EVENT\_TRIGGERS; ++i) {
11. if ((eventTriggerId & mask) != 0) {
12. fTriggeredEventClientDatas[i] = clientData;
14. fLastUsedTriggerMask = mask;
15. fLastUsedTriggerNum = i;
16. }
17. mask >>= 1;
18. }
19. }
21. // Then, note this event as being ready to be handled.
22. // (Note that because this function (unlike others in the library)
23. // can be called from an external thread, we do this last, to
24. // reduce the risk of a race condition.)
25. //利用fTriggersAwaitingHandling以bit mask的方式记录需要响应的事件handler们。
26. fTriggersAwaitingHandling |= eventTriggerId;
27. }

看，clientData被传入了，这表明clientData在每次触发事件时是可以变的。

此时再回去看SingleStep()是不是更明了了？

delay task添加时，需要传入task延迟等待的微秒（百万分之一秒）数(第一个参数)，这个弱智也可以理解吧？嘿嘿。分析一下介个函数：

1. TaskToken BasicTaskScheduler0::scheduleDelayedTask(int64\_t microseconds,TaskFunc\* proc, void\* clientData)
2. {
3. if (microseconds < 0)
4. microseconds = 0;
5. //DelayInterval 是表示时间差的结构
6. DelayInterval timeToDelay((**long**) (microseconds / 1000000),(**long**) (microseconds % 1000000));
7. //创建delayQueue中的一项
8. AlarmHandler\* alarmHandler = new AlarmHandler(proc, clientData,timeToDelay);
9. //加入DelayQueue
10. fDelayQueue.addEntry(alarmHandler);
11. //返回delay task的唯一标志
12. return (void\*) (alarmHandler->token());
13. }
15. delay task的执行都在函数fDelayQueue.handleAlarm()中，handleAlarm()在类DelayQueue中实现。看一下handleAlarm():
16. void DelayQueue::handleAlarm()
17. {
18. //如果第一个任务的执行时间未到，则同步一下（重新计算各任务的等待时间）。
19. if (head()->fDeltaTimeRemaining != DELAY\_ZERO)
20. synchronize();
21. //如果第一个任务的执行时间到了，则执行第一个，并把它从队列中删掉。
22. if (head()->fDeltaTimeRemaining == DELAY\_ZERO) {
23. // This event is due to be handled:
24. DelayQueueEntry\* toRemove = head();
25. removeEntry(toRemove); // do this first, in case handler accesses queue
26. //执行任务，执行完后会把这一项销毁。
27. toRemove->handleTimeout();
28. }
29. }

可能感觉奇怪，其它的任务队列都是先搜索第一个应该执行的项，然后再执行，这里干脆，直接执行第一个完事。那就说明第一个就是最应该执行的一个吧？也就是等待时间最短的一个吧？那么应该在添加任务时，将新任务跟据其等待时间插入到适当的位置而不是追加到尾巴上吧？猜得对不对还得看fDelayQueue.addEntry(alarmHandler)这个函数是怎么执行的。

1. void DelayQueue::addEntry(DelayQueueEntry\* newEntry)
2. {
3. //重新计算各项的等待时间
4. synchronize();
6. //取得第一项
7. DelayQueueEntry\* cur = head();
8. //从头至尾循环中将新项与各项的等待时间进行比较
9. while (newEntry->fDeltaTimeRemaining >= cur->fDeltaTimeRemaining) {
10. //如果新项等待时间长于当前项的等待时间，则减掉当前项的等待时间。
11. //也就是后面的等待时几只是与前面项等待时间的差，这样省掉记录插入时的时间的变量。
12. newEntry->fDeltaTimeRemaining -= cur->fDeltaTimeRemaining;
13. //下一项
14. cur = cur->fNext;
15. }
17. //循环完毕，cur就是找到的应插它前面的项，那就插它前面吧
18. cur->fDeltaTimeRemaining -= newEntry->fDeltaTimeRemaining;
20. // Add "newEntry" to the queue, just before "cur":
21. newEntry->fNext = cur;
22. newEntry->fPrev = cur->fPrev;
23. cur->fPrev = newEntry->fPrev->fNext = newEntry;
24. }

有个问题，while循环中为什么没有判断是否到达最后一下的代码呢？难道肯定能找到大于新项的等待时间的项吗？是的！第一个加入项的等待时间是无穷大的，而且这一项永远存在于队列中。

# 五　RTSP服务运作

基础基本搞明白了，那么RTSP,RTP等这些协议又是如何利用这些基础机制运作的呢？

首先来看RTSP.

RTSP首先需建立TCP侦听socket。可见于此函数：

1. DynamicRTSPServer\* DynamicRTSPServer::createNew(UsageEnvironment& env,
2. Port ourPort,
3. UserAuthenticationDatabase\* authDatabase,
4. unsigned reclamationTestSeconds) {
5. **int** ourSocket = setUpOurSocket(env, ourPort); //建立TCP socket
6. if (ourSocket == -1)
7. return NULL;
9. return new DynamicRTSPServer(env, ourSocket, ourPort, authDatabase,
10. reclamationTestSeconds);
11. }

要帧听客户端的连接，就需要利用任务调度机制了，所以需添加一个socket handler。可见于此函数：

1. RTSPServer::RTSPServer(UsageEnvironment& env,
2. **int** ourSocket,
3. Port ourPort,
4. UserAuthenticationDatabase\* authDatabase,
5. unsigned reclamationTestSeconds) :
6. Medium(env),
7. fRTSPServerSocket(ourSocket),
8. fRTSPServerPort(ourPort),
9. fHTTPServerSocket(-1),
10. fHTTPServerPort(0),
11. fClientSessionsForHTTPTunneling(NULL),
12. fAuthDB(authDatabase),
13. fReclamationTestSeconds(reclamationTestSeconds),
14. fServerMediaSessions(HashTable::create(STRING\_HASH\_KEYS))
15. {
16. #ifdef USE\_SIGNALS
17. // Ignore the SIGPIPE signal, so that clients on the same host that are killed
18. // don't also kill us:
19. signal(SIGPIPE, SIG\_IGN);
20. #endif
22. // Arrange to handle connections from others:
23. env.taskScheduler().turnOnBackgroundReadHandling(
24. fRTSPServerSocket,
25. (TaskScheduler::BackgroundHandlerProc\*) &incomingConnectionHandlerRTSP,
26. this);
27. }

当收到客户的连接时需保存下代表客户端的新socket，以后用这个socket与这个客户通讯。每个客户将来会对应一个rtsp会话，而且各客户的RTSP请求只控制自己的rtsp会话，那么最好建立一个会话类，代表各客户的rtsp会话。于是类RTSPServer::RTSPClientSession产生，它保存的代表客户的socket。下为RTSPClientSession的创建过程

1. void RTSPServer::incomingConnectionHandler(**int** serverSocket)
2. {
3. struct sockaddr\_in clientAddr;
4. SOCKLEN\_T clientAddrLen = sizeof clientAddr;
6. //接受连接
7. **int** clientSocket = accept(serverSocket,
8. (struct sockaddr\*) &clientAddr,
9. &clientAddrLen);
11. if (clientSocket < 0) {
12. **int** err = envir().getErrno();
13. if (err != EWOULDBLOCK) {
14. envir().setResultErrMsg("accept() failed: ");
15. }
16. return;
17. }
19. //设置socket的参数
20. makeSocketNonBlocking(clientSocket);
21. increaseSendBufferTo(envir(), clientSocket, 50 \* 1024);
23. #ifdef DEBUG
24. envir() << "accept()ed connection from " << our\_inet\_ntoa(clientAddr.sin\_addr) << "\n";
25. #endif
27. //产生一个sesson id
28. // Create a new object for this RTSP session.
29. // (Choose a random 32-bit integer for the session id (it will be encoded as a
30. // 8-digit hex number). We don't bother checking for a collision;
31. // the probability of two concurrent sessions getting the same session id is very low.)
32. // (We do, however, avoid choosing session id 0, because that has a special use (by "OnDemandServerMediaSubsession").)
33. unsigned sessionId;
34. do {
35. sessionId = (unsigned) our\_random();
36. } while (sessionId == 0);
38. //创建RTSPClientSession，注意传入的参数
39. (void) createNewClientSession(sessionId, clientSocket, clientAddr);
40. }

  RTSPClientSession要提供什么功能呢？可以想象：需要监听客户端的rtsp请求并回应它，需要在DESCRIBE请求中返回所请求的流的信息，需要在SETUP请求中建立起RTP会话，需要在TEARDOWN请求中关闭RTP会话，等等...

RTSPClientSession要侦听客户端的请求，就需把自己的socket handler加入计划任务。证据如下：

1. RTSPServer::RTSPClientSession::RTSPClientSession(
2. RTSPServer& ourServer,
3. unsigned sessionId,
4. **int** clientSocket,
5. struct sockaddr\_in clientAddr) :
6. fOurServer(ourServer),
7. fOurSessionId(sessionId),
8. fOurServerMediaSession(NULL),
9. fClientInputSocket(clientSocket),
10. fClientOutputSocket(clientSocket),
11. fClientAddr(clientAddr),
12. fSessionCookie(NULL),
13. fLivenessCheckTask(NULL),
14. fIsMulticast(False),
15. fSessionIsActive(True),
16. fStreamAfterSETUP(False),
17. fTCPStreamIdCount(0),
18. fNumStreamStates(0),
19. fStreamStates(NULL),
20. fRecursionCount(0)
21. {
22. // Arrange to handle incoming requests:
23. resetRequestBuffer();
24. envir().taskScheduler().turnOnBackgroundReadHandling(fClientInputSocket, (TaskScheduler::BackgroundHandlerProc\*) &incomingRequestHandler, this);
25. noteLiveness();
26. }

下面重点讲一下下RTSPClientSession响应DESCRIBE请求的过程：

1. void RTSPServer::RTSPClientSession::handleCmd\_DESCRIBE(
2. **char** const\* cseq,
3. **char** const\* urlPreSuffix,
4. **char** const\* urlSuffix,
5. **char** const\* fullRequestStr)
6. {
7. **char**\* sdpDescription = NULL;
8. **char**\* rtspURL = NULL;
9. do {
10. //整理一下下RTSP地址
11. **char** urlTotalSuffix[RTSP\_PARAM\_STRING\_MAX];
12. if (strlen(urlPreSuffix) + strlen(urlSuffix) + 2  > sizeof urlTotalSuffix) {
13. handleCmd\_bad(cseq);
14. break;
15. }
16. urlTotalSuffix[0] = '\0';
17. if (urlPreSuffix[0] != '\0') {
18. strcat(urlTotalSuffix, urlPreSuffix);
19. strcat(urlTotalSuffix, "/");
20. }
21. strcat(urlTotalSuffix, urlSuffix);
23. //验证帐户和密码
24. if (!authenticationOK("DESCRIBE", cseq, urlTotalSuffix, fullRequestStr))
25. break;
27. // We should really check that the request contains an "Accept:" #####
28. // for "application/sdp", because that's what we're sending back #####
30. // Begin by looking up the "ServerMediaSession" object for the specified "urlTotalSuffix":
31. //跟据流的名字查找ServerMediaSession，如果找不到，会创建一个。每个ServerMediaSession中至少要
32. //包含一个ServerMediaSubsession。一个ServerMediaSession对应一个媒体，可以认为是Server上的一个文件，或一个实时获取设备。其包含的每个ServerMediaSubSession代表媒体中的一个Track。所以一个ServerMediaSession对应一个媒体，如果客户请求的媒体名相同，就使用已存在的ServerMediaSession，如果不同，就创建一个新的。一个流对应一个StreamState，StreamState与ServerMediaSubsession相关，但代表的是动态的，而ServerMediaSubsession代表静态的。
33. ServerMediaSession\* session = fOurServer.lookupServerMediaSession(urlTotalSuffix);
34. if (session == NULL) {
35. handleCmd\_notFound(cseq);
36. break;
37. }

40. // Then, assemble a SDP description for this session:
41. //获取SDP字符串，在函数内会依次获取每个ServerMediaSubSession的字符串然连接起来。
42. sdpDescription = session->generateSDPDescription();
43. if (sdpDescription == NULL) {
44. // This usually means that a file name that was specified for a
45. // "ServerMediaSubsession" does not exist.
46. snprintf((**char**\*) fResponseBuffer, sizeof fResponseBuffer,
47. "RTSP/1.0 404 File Not Found, Or In Incorrect Format\r\n"
48. "CSeq: %s\r\n"
49. "%s\r\n", cseq, dateHeader());
50. break;
51. }
52. unsigned sdpDescriptionSize = strlen(sdpDescription);
54. // Also, generate our RTSP URL, for the "Content-Base:" header
55. // (which is necessary to ensure that the correct URL gets used in
56. // subsequent "SETUP" requests).
57. rtspURL = fOurServer.rtspURL(session, fClientInputSocket);
59. //形成响应DESCRIBE请求的RTSP字符串。
60. snprintf((**char**\*) fResponseBuffer, sizeof fResponseBuffer,
61. "RTSP/1.0 200 OK\r\nCSeq: %s\r\n"
62. "%s"
63. "Content-Base: %s/\r\n"
64. "Content-Type: application/sdp\r\n"
65. "Content-Length: %d\r\n\r\n"
66. "%s", cseq, dateHeader(), rtspURL, sdpDescriptionSize,
67. sdpDescription);
68. } while (0);
69. delete[] sdpDescription;
70. delete[] rtspURL;
72. //返回后会被立即发送（没有把socket write操作放入计划任务中）。
73. }

fOurServer.lookupServerMediaSession(urlTotalSuffix)中会在找不到同名ServerMediaSession时新建一个，代表一个RTP流的ServerMediaSession们是被RTSPServer管理的，而不是被RTSPClientSession拥有。为什么呢？因为ServerMediaSession代表的是一个静态的流，也就是可以从它里面获取一个流的各种信息，但不能获取传输状态。不同客户可能连接到同一个流，所以ServerMediaSession应被RTSPServer所拥有。创建一个ServerMediaSession过程值得一观：

1. static ServerMediaSession\* createNewSMS(UsageEnvironment& env,**char** const\* fileName, **FILE**\* /\*fid\*/)
2. {
3. // Use the file name extension to determine the type of "ServerMediaSession":
4. **char** const\* extension = strrchr(fileName, '.');
5. if (extension == NULL)
6. return NULL;
8. ServerMediaSession\* sms = NULL;
9. Boolean const reuseSource = False;
10. if (strcmp(extension, ".aac") == 0) {
11. // Assumed to be an AAC Audio (ADTS format) file:
12. NEW\_SMS("AAC Audio");
13. sms->addSubsession(
14. ADTSAudioFileServerMediaSubsession::createNew(env, fileName, reuseSource));
15. } else if (strcmp(extension, ".amr") == 0) {
16. // Assumed to be an AMR Audio file:
17. NEW\_SMS("AMR Audio");
18. sms->addSubsession(
19. AMRAudioFileServerMediaSubsession::createNew(env, fileName, reuseSource));
20. } else if (strcmp(extension, ".ac3") == 0) {
21. // Assumed to be an AC-3 Audio file:
22. NEW\_SMS("AC-3 Audio");
23. sms->addSubsession(
24. AC3AudioFileServerMediaSubsession::createNew(env, fileName, reuseSource));
25. } else if (strcmp(extension, ".m4e") == 0) {
26. // Assumed to be a MPEG-4 Video Elementary Stream file:
27. NEW\_SMS("MPEG-4 Video");
28. sms->addSubsession(
29. MPEG4VideoFileServerMediaSubsession::createNew(env, fileName, reuseSource));
30. } else if (strcmp(extension, ".264") == 0) {
31. // Assumed to be a H.264 Video Elementary Stream file:
32. NEW\_SMS("H.264 Video");
33. OutPacketBuffer::maxSize = 100000; // allow for some possibly large H.264 frames
34. sms->addSubsession(
35. H264VideoFileServerMediaSubsession::createNew(env, fileName, reuseSource));
36. } else if (strcmp(extension, ".mp3") == 0) {
37. // Assumed to be a MPEG-1 or 2 Audio file:
38. NEW\_SMS("MPEG-1 or 2 Audio");
39. // To stream using 'ADUs' rather than raw MP3 frames, uncomment the following:
40. //#define STREAM\_USING\_ADUS 1
41. // To also reorder ADUs before streaming, uncomment the following:
42. //#define INTERLEAVE\_ADUS 1
43. // (For more information about ADUs and interleaving,
44. //  see <http://www.live555.com/rtp-mp3/>)
45. Boolean useADUs = False;
46. Interleaving\* interleaving = NULL;
47. #ifdef STREAM\_USING\_ADUS
48. useADUs = True;
49. #ifdef INTERLEAVE\_ADUS
50. unsigned **char** interleaveCycle[] = {0,2,1,3}; // or choose your own...
51. unsigned const interleaveCycleSize = (sizeof interleaveCycle)/(sizeof (unsigned **char**));
52. interleaving = new Interleaving(interleaveCycleSize, interleaveCycle);
53. #endif
54. #endif
55. sms->addSubsession(
56. MP3AudioFileServerMediaSubsession::createNew(env, fileName,
57. reuseSource, useADUs, interleaving));
58. } else if (strcmp(extension, ".mpg") == 0) {
59. // Assumed to be a MPEG-1 or 2 Program Stream (audio+video) file:
60. NEW\_SMS("MPEG-1 or 2 Program Stream");
61. MPEG1or2FileServerDemux\* demux = MPEG1or2FileServerDemux::createNew(env,
62. fileName, reuseSource);
63. sms->addSubsession(demux->newVideoServerMediaSubsession());
64. sms->addSubsession(demux->newAudioServerMediaSubsession());
65. } else if (strcmp(extension, ".ts") == 0) {
66. // Assumed to be a MPEG Transport Stream file:
67. // Use an index file name that's the same as the TS file name, except with ".tsx":
68. unsigned indexFileNameLen = strlen(fileName) + 2; // allow for trailing "x\0"
69. **char**\* indexFileName = new **char**[indexFileNameLen];
70. sprintf(indexFileName, "%sx", fileName);
71. NEW\_SMS("MPEG Transport Stream");
72. sms->addSubsession(
73. MPEG2TransportFileServerMediaSubsession::createNew(env,
74. fileName, indexFileName, reuseSource));
75. delete[] indexFileName;
76. } else if (strcmp(extension, ".wav") == 0) {
77. // Assumed to be a WAV Audio file:
78. NEW\_SMS("WAV Audio Stream");
79. // To convert 16-bit PCM data to 8-bit u-law, prior to streaming,
80. // change the following to True:
81. Boolean convertToULaw = False;
82. sms->addSubsession(
83. WAVAudioFileServerMediaSubsession::createNew(env, fileName, reuseSource, convertToULaw));
84. } else if (strcmp(extension, ".dv") == 0) {
85. // Assumed to be a DV Video file
86. // First, make sure that the RTPSinks' buffers will be large enough to handle the huge size of DV frames (as big as 288000).
87. OutPacketBuffer::maxSize = 300000;
89. NEW\_SMS("DV Video");
90. sms->addSubsession(
91. DVVideoFileServerMediaSubsession::createNew(env, fileName, reuseSource));
92. } else if (strcmp(extension, ".mkv") == 0) {
93. // Assumed to be a Matroska file
94. NEW\_SMS("Matroska video+audio+(optional)subtitles");
96. // Create a Matroska file server demultiplexor for the specified file.  (We enter the event loop to wait for this to complete.)
97. newMatroskaDemuxWatchVariable = 0;
98. MatroskaFileServerDemux::createNew(env, fileName, onMatroskaDemuxCreation, NULL);
99. env.taskScheduler().doEventLoop(&newMatroskaDemuxWatchVariable);
101. ServerMediaSubsession\* smss;
102. while ((smss = demux->newServerMediaSubsession()) != NULL) {
103. sms->addSubsession(smss);
104. }
105. }
107. return sms;
108. }

可以看到NEW\_SMS("AMR Audio")会创建新的ServerMediaSession，之后马上调用sms->addSubsession（）为这个ServerMediaSession添加一个 ServerMediaSubSession 。看起来ServerMediaSession应该可以添加多个ServerMediaSubSession，但这里并没有这样做。如果可以添加多个 ServerMediaSubsession 那么ServerMediaSession与流名字所指定与文件是没有关系的，也就是说它不会操作文件，而文件的操作是放在 ServerMediaSubsession中的。具体应该是在ServerMediaSubsession的sdpLines()函数中打开。

# 六　建立RTP会话

首先更正一个概念：ServerMediaSession原先说代表一个流，其实是不准确的。它代表的是server端的一个媒体的名字，而说ServerMediaSubsession代表一个Track是准确的。以后流指的是那些有数据流动的组合。

RTP的建立：

RTP的建立过程无非是这样：client告诉server自己的rtp/rtcp端口号，server建立自己的rtp/rtcp socket，然后在收到PLAY请求时向客户端发数据。看起来非常简单。

在收到SETUP请求时才建立连接，让我们看一下处理这个命令的函数：

1. void RTSPServer::RTSPClientSession::handleCmd\_SETUP(
2. **char** const\* cseq,
3. **char** const\* urlPreSuffix,
4. **char** const\* urlSuffix,
5. **char** const\* fullRequestStr)
6. {
7. // Normally, "urlPreSuffix" should be the session (stream) name,
8. // and "urlSuffix" should be the subsession (track) name.
9. // However (being "liberal in what we accept"), we also handle
10. // 'aggregate' SETUP requests (i.e., without a track name),
11. // in the special case where we have only a single track.  I.e.,
12. // in this case, we also handle:
13. //    "urlPreSuffix" is empty and "urlSuffix" is the session (stream) name, or
14. //    "urlPreSuffix" concatenated with "urlSuffix" (with "/" inbetween)
15. //    is the session (stream) name.
16. **char** const\* streamName = urlPreSuffix; // in the normal case
17. **char** const\* trackId = urlSuffix; // in the normal case
18. **char**\* concatenatedStreamName = NULL; // in the normal case
20. do {
21. // First, make sure the specified stream name exists:
22. fOurServerMediaSession = fOurServer.lookupServerMediaSession(streamName);
23. if (fOurServerMediaSession == NULL) {
24. // Check for the special case (noted above), before we up:
25. if (urlPreSuffix[0] == '\0') {
26. streamName = urlSuffix;
27. } else {
28. concatenatedStreamName = new **char**[strlen(urlPreSuffix)
29. + strlen(urlSuffix) + 2]; // allow for the "/" and the trailing '\0'
30. sprintf(concatenatedStreamName, "%s/%s", urlPreSuffix,  urlSuffix);
31. streamName = concatenatedStreamName;
32. }
33. trackId = NULL;
35. // Check again:
36. fOurServerMediaSession = fOurServer.lookupServerMediaSession(streamName);
37. }
38. if (fOurServerMediaSession == NULL) {
39. handleCmd\_notFound(cseq);
40. break;
41. }
43. fOurServerMediaSession->incrementReferenceCount();
45. //如没有则为一个流中所有的track都分配一个stream state
46. if (fStreamStates == NULL) {
47. // This is the first "SETUP" for this session.  Set up our
48. // array of states for all of this session's subsessions (tracks):
49. ServerMediaSubsessionIterator iter(\*fOurServerMediaSession);
50. for (fNumStreamStates = 0; iter.next() != NULL; ++fNumStreamStates) {
51. } // begin by counting the number of subsessions (tracks)
52. fStreamStates = new streamState[fNumStreamStates];
54. iter.reset();
55. ServerMediaSubsession\* subsession;
56. for (unsigned i = 0; i < fNumStreamStates; ++i) {
57. subsession = iter.next();
58. fStreamStates[i].subsession = subsession;
59. fStreamStates[i].streamToken = NULL; // for now; it may be changed by the "getStreamParameters()" call that comes later
60. }
61. }
63. //查找当前请求的track的信息
64. // Look up information for the specified subsession (track):
65. ServerMediaSubsession\* subsession = NULL;
66. unsigned streamNum;
67. if (trackId != NULL && trackId[0] != '\0') { // normal case
68. for (streamNum = 0; streamNum < fNumStreamStates; ++streamNum) {
69. subsession = fStreamStates[streamNum].subsession;
70. if (subsession != NULL  && strcmp(trackId, subsession->trackId()) == 0)
71. break; //找到啦！
72. }
73. if (streamNum >= fNumStreamStates) {
74. // The specified track id doesn't exist, so this request fails:
75. handleCmd\_notFound(cseq);
76. break;
77. }
78. } else {
79. // Weird case: there was no track id in the URL.
80. // This works only if we have only one subsession:
81. if (fNumStreamStates != 1) {
82. handleCmd\_bad(cseq);
83. break;
84. }
85. streamNum = 0;
86. subsession = fStreamStates[streamNum].subsession;
87. }
88. // ASSERT: subsession != NULL
90. //分析RTSP请求字符串中的传输要求
91. // Look for a "Transport:" header in the request string, to extract client parameters:
92. StreamingMode streamingMode;
93. **char**\* streamingModeString = NULL; // set when RAW\_UDP streaming is specified
94. **char**\* clientsDestinationAddressStr;
95. u\_int8\_t clientsDestinationTTL;
96. portNumBits clientRTPPortNum, clientRTCPPortNum;
97. unsigned **char** rtpChannelId, rtcpChannelId;
98. parseTransportHeader(fullRequestStr, streamingMode, streamingModeString,
99. clientsDestinationAddressStr, clientsDestinationTTL,
100. clientRTPPortNum, clientRTCPPortNum, rtpChannelId,
101. rtcpChannelId);
102. if (streamingMode == RTP\_TCP && rtpChannelId == 0xFF
103. || streamingMode != RTP\_TCP && fClientOutputSocket != fClientInputSocket) {
104. // An anomolous situation, caused by a buggy client.  Either:
105. //     1/ TCP streaming was requested, but with no "interleaving=" fields.  (QuickTime Player sometimes does this.), or
106. //     2/ TCP streaming was not requested, but we're doing RTSP-over-HTTP tunneling (which implies TCP streaming).
107. //  In either case, we assume TCP streaming, and set the RTP and RTCP channel ids to proper values:
108. streamingMode = RTP\_TCP;
109. rtpChannelId = fTCPStreamIdCount;
110. rtcpChannelId = fTCPStreamIdCount + 1;
111. }
112. fTCPStreamIdCount += 2;
114. Port clientRTPPort(clientRTPPortNum);
115. Port clientRTCPPort(clientRTCPPortNum);
117. // Next, check whether a "Range:" header is present in the request.
118. // This isn't legal, but some clients do this to combine "SETUP" and "PLAY":
119. **double** rangeStart = 0.0, rangeEnd = 0.0;
120. fStreamAfterSETUP = parseRangeHeader(fullRequestStr, rangeStart,
121. rangeEnd) || parsePlayNowHeader(fullRequestStr);
123. // Then, get server parameters from the 'subsession':
124. **int** tcpSocketNum = streamingMode == RTP\_TCP ? fClientOutputSocket : -1;
125. netAddressBits destinationAddress = 0;
126. u\_int8\_t destinationTTL = 255;
127. #ifdef RTSP\_ALLOW\_CLIENT\_DESTINATION\_SETTING
128. if (clientsDestinationAddressStr != NULL) {
129. // Use the client-provided "destination" address.
130. // Note: This potentially allows the server to be used in denial-of-service
131. // attacks, so don't enable this code unless you're sure that clients are trusted.
132. destinationAddress = our\_inet\_addr(clientsDestinationAddressStr);
133. }
134. // Also use the client-provided TTL.
135. destinationTTL = clientsDestinationTTL;
136. #endif
137. delete[] clientsDestinationAddressStr;
138. Port serverRTPPort(0);
139. Port serverRTCPPort(0);
141. // Make sure that we transmit on the same interface that's used by
142. // the client (in case we're a multi-homed server):
143. struct sockaddr\_in sourceAddr;
144. SOCKLEN\_T namelen = sizeof sourceAddr;
145. getsockname(fClientInputSocket, (struct sockaddr\*) &sourceAddr, &namelen);
146. netAddressBits origSendingInterfaceAddr = SendingInterfaceAddr;
147. netAddressBits origReceivingInterfaceAddr = ReceivingInterfaceAddr;
148. // NOTE: The following might not work properly, so we ifdef it out for now:
149. #ifdef HACK\_FOR\_MULTIHOMED\_SERVERS
150. ReceivingInterfaceAddr = SendingInterfaceAddr = sourceAddr.sin\_addr.s\_addr;
151. #endif
153. //获取rtp连接信息，在其中已建立起了server端的rtp和rtcp socket，返回
154. //fStreamStates[streamNum].streamToken表示数据流已经建立起来了
155. subsession->getStreamParameters(fOurSessionId,
156. fClientAddr.sin\_addr.s\_addr, clientRTPPort, clientRTCPPort,
157. tcpSocketNum, rtpChannelId, rtcpChannelId, destinationAddress,
158. destinationTTL, fIsMulticast, serverRTPPort, serverRTCPPort,
159. fStreamStates[streamNum].streamToken);
160. SendingInterfaceAddr = origSendingInterfaceAddr;
161. ReceivingInterfaceAddr = origReceivingInterfaceAddr;
163. //形成RTSP回应字符串
164. struct in\_addr destinationAddr;
165. destinationAddr.s\_addr = destinationAddress;
166. **char**\* destAddrStr = strDup(our\_inet\_ntoa(destinationAddr));
167. **char**\* sourceAddrStr = strDup(our\_inet\_ntoa(sourceAddr.sin\_addr));
168. if (fIsMulticast) {
169. switch (streamingMode) {
170. case RTP\_UDP:
171. snprintf(
172. (**char**\*) fResponseBuffer,
173. sizeof fResponseBuffer,
174. "RTSP/1.0 200 OK\r\n"
175. "CSeq: %s\r\n"
176. "%s"
177. "Transport: RTP/AVP;multicast;destination=%s;source=%s;port=%d-%d;ttl=%d\r\n"
178. "Session: %08X\r\n\r\n", cseq, dateHeader(),
179. destAddrStr, sourceAddrStr, ntohs(serverRTPPort.num()),
180. ntohs(serverRTCPPort.num()), destinationTTL,
181. fOurSessionId);
182. break;
183. case RTP\_TCP:
184. // multicast streams can't be sent via TCP
185. handleCmd\_unsupportedTransport(cseq);
186. break;
187. case RAW\_UDP:
188. snprintf(
189. (**char**\*) fResponseBuffer,
190. sizeof fResponseBuffer,
191. "RTSP/1.0 200 OK\r\n"
192. "CSeq: %s\r\n"
193. "%s"
194. "Transport: %s;multicast;destination=%s;source=%s;port=%d;ttl=%d\r\n"
195. "Session: %08X\r\n\r\n", cseq, dateHeader(),
196. streamingModeString, destAddrStr, sourceAddrStr,
197. ntohs(serverRTPPort.num()), destinationTTL,
198. fOurSessionId);
199. break;
200. }
201. } else {
202. switch (streamingMode) {
203. case RTP\_UDP: {
204. snprintf(
205. (**char**\*) fResponseBuffer,
206. sizeof fResponseBuffer,
207. "RTSP/1.0 200 OK\r\n"
208. "CSeq: %s\r\n"
209. "%s"
210. "Transport: RTP/AVP;unicast;destination=%s;source=%s;client\_port=%d-%d;server\_port=%d-%d\r\n"
211. "Session: %08X\r\n\r\n", cseq, dateHeader(),
212. destAddrStr, sourceAddrStr, ntohs(clientRTPPort.num()),
213. ntohs(clientRTCPPort.num()), ntohs(serverRTPPort.num()),
214. ntohs(serverRTCPPort.num()), fOurSessionId);
215. break;
216. }
217. case RTP\_TCP: {
218. snprintf(
219. (**char**\*) fResponseBuffer,
220. sizeof fResponseBuffer,
221. "RTSP/1.0 200 OK\r\n"
222. "CSeq: %s\r\n"
223. "%s"
224. "Transport: RTP/AVP/TCP;unicast;destination=%s;source=%s;interleaved=%d-%d\r\n"
225. "Session: %08X\r\n\r\n", cseq, dateHeader(),
226. destAddrStr, sourceAddrStr, rtpChannelId, rtcpChannelId,
227. fOurSessionId);
228. break;
229. }
230. case RAW\_UDP: {
231. snprintf(
232. (**char**\*) fResponseBuffer,
233. sizeof fResponseBuffer,
234. "RTSP/1.0 200 OK\r\n"
235. "CSeq: %s\r\n"
236. "%s"
237. "Transport: %s;unicast;destination=%s;source=%s;client\_port=%d;server\_port=%d\r\n"
238. "Session: %08X\r\n\r\n", cseq, dateHeader(),
239. streamingModeString, destAddrStr, sourceAddrStr,
240. ntohs(clientRTPPort.num()), ntohs(serverRTPPort.num()),
241. fOurSessionId);
242. break;
243. }
244. }
245. }
246. delete[] destAddrStr;
247. delete[] sourceAddrStr;
248. delete[] streamingModeString;
249. } while (0);

252. delete[] concatenatedStreamName;
253. //返回后，回应字符串会被立即发送
254. }

live555 中有两个 streamstate，一个是类 StreamState ，一个是此处的结构 struct streamState。类 SteamState 就是 streamToken，而 struct streamState 中保存了 MediaSubsession (即track) 和类 StreamState 的对应。类 StreamState 代表一个真正流动起来的数据流。这个数据流是从源流到 Sink 。客户端与服务端的一个 rtp 会话中，有两个数据流，服务端是从 XXXFileSouce 流到 RTPSink，而客户端则是从 RTPSource 流到 XXXFileSink 。建立数据流的过程就是把 Source 与 Sink 连接起来。

为何不把 StreamToken 保存在 MediaSubsession 中呢？看起来在 struct streamState 中是一个 MediaSubsession 对应一个 streamToken 呀？ 因为 MediaSubsession  代表一个 track 的静态数据，它是可以被其它 rtp 会话重用的。比如不同的用户可能会连接到同一个媒体的同一个 track 。所以　streamToken　与　MediaSubsession　独立存在，只是被　RTSPClientSession　给对应了起来。

streamToken的建立过程存在于函数subsession->getStreamParameters（）中，让我们看一下：

1. void OnDemandServerMediaSubsession::getStreamParameters(
2. unsigned clientSessionId,
3. netAddressBits clientAddress,
4. Port const& clientRTPPort,
5. Port const& clientRTCPPort,
6. **int** tcpSocketNum,
7. unsigned **char** rtpChannelId,
8. unsigned **char** rtcpChannelId,
9. netAddressBits& destinationAddress,
10. u\_int8\_t& /\*destinationTTL\*/,
11. Boolean& isMulticast,
12. Port& serverRTPPort,
13. Port& serverRTCPPort,
14. void\*& streamToken)
15. {
16. if (destinationAddress == 0)
17. destinationAddress = clientAddress;
19. struct in\_addr destinationAddr;
20. destinationAddr.s\_addr = destinationAddress;
21. isMulticast = False;
23. //ServerMediaSubsession并没有保存所有基于自己的数据流，而是只记录了最后一次建立的数据流。
24. //利用这个变量和fReuseFirstSource可以实现多client连接到一个流的形式。
25. if (fLastStreamToken != NULL && fReuseFirstSource) {
26. //如果这个ServerMediaSubsession已经建好连接，并且希望使用这个连接则直接返回这个连接。
27. // Special case: Rather than creating a new 'StreamState',
28. // we reuse the one that we've already created:
29. serverRTPPort = ((StreamState\*) fLastStreamToken)->serverRTPPort();
30. serverRTCPPort = ((StreamState\*) fLastStreamToken)->serverRTCPPort();
31. ++((StreamState\*) fLastStreamToken)->referenceCount();
32. streamToken = fLastStreamToken;
33. } else {
34. // Normal case: Create a new media source:
35. unsigned streamBitrate;
36. FramedSource\* mediaSource = createNewStreamSource(clientSessionId,streamBitrate);
38. // Create 'groupsock' and 'sink' objects for the destination,
39. // using previously unused server port numbers:
40. RTPSink\* rtpSink;
41. BasicUDPSink\* udpSink;
42. Groupsock\* rtpGroupsock;
43. Groupsock\* rtcpGroupsock;
44. portNumBits serverPortNum;
45. if (clientRTCPPort.num() == 0) {  //不使用RTCP
46. // We're streaming raw UDP (not RTP). Create a single groupsock:
47. NoReuse dummy; // ensures that we skip over ports that are already in use
48. for (serverPortNum = fInitialPortNum;; ++serverPortNum) {
49. struct in\_addr dummyAddr;
50. dummyAddr.s\_addr = 0;
52. serverRTPPort = serverPortNum;
53. rtpGroupsock = new Groupsock(envir(), dummyAddr, serverRTPPort, 255);
54. if (rtpGroupsock->socketNum() >= 0)
55. break; // success
56. }
58. rtcpGroupsock = NULL;
59. rtpSink = NULL;
60. udpSink = BasicUDPSink::createNew(envir(), rtpGroupsock);
61. } else {
62. // Normal case: We're streaming RTP (over UDP or TCP).  Create a pair of
63. // groupsocks (RTP and RTCP), with adjacent port numbers (RTP port number even):
64. NoReuse dummy; // ensures that we skip over ports that are already in use
65. for (portNumBits serverPortNum = fInitialPortNum;; serverPortNum += 2) {
66. struct in\_addr dummyAddr;
67. dummyAddr.s\_addr = 0;
69. serverRTPPort = serverPortNum;
70. rtpGroupsock = new Groupsock(envir(), dummyAddr, serverRTPPort, 255);
71. if (rtpGroupsock->socketNum() < 0) {
72. delete rtpGroupsock;
73. continue; // try again
74. }
76. serverRTCPPort = serverPortNum + 1;
77. rtcpGroupsock = new Groupsock(envir(), dummyAddr,serverRTCPPort, 255);
78. if (rtcpGroupsock->socketNum() < 0) {
79. delete rtpGroupsock;
80. delete rtcpGroupsock;
81. continue; // try again
82. }
84. break; // success
85. }
87. unsigned **char** rtpPayloadType = 96 + trackNumber() - 1; // if dynamic
88. rtpSink = createNewRTPSink(rtpGroupsock, rtpPayloadType, mediaSource);
89. udpSink = NULL;
90. }
92. // Turn off the destinations for each groupsock.  They'll get set later
93. // (unless TCP is used instead):
94. if (rtpGroupsock != NULL)
95. rtpGroupsock->removeAllDestinations();
96. if (rtcpGroupsock != NULL)
97. rtcpGroupsock->removeAllDestinations();
99. if (rtpGroupsock != NULL) {
100. // Try to use a big send buffer for RTP -  at least 0.1 second of
101. // specified bandwidth and at least 50 KB
102. unsigned rtpBufSize = streamBitrate \* 25 / 2; // 1 kbps \* 0.1 s = 12.5 bytes
103. if (rtpBufSize < 50 \* 1024)
104. rtpBufSize = 50 \* 1024;
105. increaseSendBufferTo(envir(), rtpGroupsock->socketNum(),rtpBufSize);
106. }
108. // Set up the state of the stream.  The stream will get started later:
109. streamToken = fLastStreamToken = new StreamState(\*this, serverRTPPort,
110. serverRTCPPort, rtpSink, udpSink, streamBitrate, mediaSource,
111. rtpGroupsock, rtcpGroupsock);
112. }
114. // Record these destinations as being for this client session id:
115. Destinations\* destinations;
116. if (tcpSocketNum < 0) { // UDP
117. destinations = new Destinations(destinationAddr, clientRTPPort, clientRTCPPort);
118. } else { // TCP
119. destinations = new Destinations(tcpSocketNum, rtpChannelId, rtcpChannelId);
120. }
122. //记录下所有clientSessionID对应的目的rtp/rtcp地址，是因为现在不能把目的rtp,rtcp地址加入到
123. //server端rtp的groupSocket中。试想在ReuseFirstSource时，这样会引起client端立即收到rtp数据。
124. //其次，也可以利用这个hash table找出client的rtp/rtcp端口等信息，好像其它地方还真没有可以保存的
125. //RTSPClientSession中的streamstates在ReuseFirstSource时也不能准确找出client端的端口等信息。
126. fDestinationsHashTable->Add((**char** const\*) clientSessionId, destinations);
127. }

流程不复杂：如果需要重用上一次建立的流，就利用之（这样就可以实现一rtp server对应多个rtp client的形式）；如果不需要，则创建合适的source，然后创建rtp sink，然后利用它们创建streamSoken。

启动一个流：

当RTSPClientSession收到PLAY请求时，就开始传输RTP数据。下面看一下流启动的代码：

1. void RTSPServer::RTSPClientSession::handleCmd\_PLAY(
2. ServerMediaSubsession\* subsession,
3. **char** const\* cseq,
4. **char** const\* fullRequestStr)
5. {
6. **char**\* rtspURL = fOurServer.rtspURL(fOurServerMediaSession,fClientInputSocket);
7. unsigned rtspURLSize = strlen(rtspURL);
9. // Parse the client's "Scale:" header, if any:
10. **float** scale;
11. Boolean sawScaleHeader = parseScaleHeader(fullRequestStr, scale);
13. // Try to set the stream's scale factor to this value:
14. if (subsession == NULL /\*aggregate op\*/) {
15. fOurServerMediaSession->testScaleFactor(scale);
16. } else {
17. subsession->testScaleFactor(scale);
18. }
20. **char** buf[100];
21. **char**\* scaleHeader;
22. if (!sawScaleHeader) {
23. buf[0] = '\0'; // Because we didn't see a Scale: header, don't send one back
24. } else {
25. sprintf(buf, "Scale: %f\r\n", scale);
26. }
27. scaleHeader = strDup(buf);
29. //分析客户端对于播放范围的要求
30. // Parse the client's "Range:" header, if any:
31. **double** rangeStart = 0.0, rangeEnd = 0.0;
32. Boolean sawRangeHeader = parseRangeHeader(fullRequestStr, rangeStart,rangeEnd);
34. // Use this information, plus the stream's duration (if known), to create
35. // our own "Range:" header, for the response:
36. **float** duration = subsession == NULL /\*aggregate op\*/
37. ? fOurServerMediaSession->duration() : subsession->duration();
38. if (duration < 0.0) {
39. // We're an aggregate PLAY, but the subsessions have different durations.
40. // Use the largest of these durations in our header
41. duration = -duration;
42. }
44. // Make sure that "rangeStart" and "rangeEnd" (from the client's "Range:" header) have sane values
45. // before we send back our own "Range:" header in our response:
46. if (rangeStart < 0.0)
47. rangeStart = 0.0;
48. else if (rangeStart > duration)
49. rangeStart = duration;
50. if (rangeEnd < 0.0)
51. rangeEnd = 0.0;
52. else if (rangeEnd > duration)
53. rangeEnd = duration;
54. if ((scale > 0.0 && rangeStart > rangeEnd && rangeEnd > 0.0)
55. || (scale < 0.0 && rangeStart < rangeEnd)) {
56. // "rangeStart" and "rangeEnd" were the wrong way around; swap them:
57. **double** tmp = rangeStart;
58. rangeStart = rangeEnd;
59. rangeEnd = tmp;
60. }
62. **char**\* rangeHeader;
63. if (!sawRangeHeader) {
64. buf[0] = '\0'; // Because we didn't see a Range: header, don't send one back
65. } else if (rangeEnd == 0.0 && scale >= 0.0) {
66. sprintf(buf, "Range: npt=%.3f-\r\n", rangeStart);
67. } else {
68. sprintf(buf, "Range: npt=%.3f-%.3f\r\n", rangeStart, rangeEnd);
69. }
70. rangeHeader = strDup(buf);
72. // Create a "RTP-Info:" line.  It will get filled in from each subsession's state:
73. **char** const\* rtpInfoFmt = "%s" // "RTP-Info:", plus any preceding rtpInfo items
74. "%s"// comma separator, if needed
75. "url=%s/%s"
76. ";seq=%d"
77. ";rtptime=%u";
78. unsigned rtpInfoFmtSize = strlen(rtpInfoFmt);
79. **char**\* rtpInfo = strDup("RTP-Info: ");
80. unsigned i, numRTPInfoItems = 0;
82. // Do any required seeking/scaling on each subsession, before starting streaming:
83. for (i = 0; i < fNumStreamStates; ++i) {
84. if (subsession == NULL /\* means: aggregated operation \*/
85. || subsession == fStreamStates[i].subsession) {
86. if (sawScaleHeader) {
87. fStreamStates[i].subsession->setStreamScale(fOurSessionId,
88. fStreamStates[i].streamToken, scale);
89. }
90. if (sawRangeHeader) {
91. **double** streamDuration = 0.0; // by default; means: stream until the end of the media
92. if (rangeEnd > 0.0 && (rangeEnd + 0.001) < duration) { // the 0.001 is because we limited the values to 3 decimal places
93. // We want the stream to end early.  Set the duration we want:
94. streamDuration = rangeEnd - rangeStart;
95. if (streamDuration < 0.0)
96. streamDuration = -streamDuration; // should happen only if scale < 0.0
97. }
98. u\_int64\_t numBytes;
99. fStreamStates[i].subsession->seekStream(fOurSessionId,
100. fStreamStates[i].streamToken, rangeStart,
101. streamDuration, numBytes);
102. }
103. }
104. }
106. // Now, start streaming:
107. for (i = 0; i < fNumStreamStates; ++i) {
108. if (subsession == NULL /\* means: aggregated operation \*/
109. || subsession == fStreamStates[i].subsession) {
110. unsigned **short** rtpSeqNum = 0;
111. unsigned rtpTimestamp = 0;
112. //启动流
113. fStreamStates[i].subsession->startStream(fOurSessionId,
114. fStreamStates[i].streamToken,
115. (TaskFunc\*) noteClientLiveness, this, rtpSeqNum,
116. rtpTimestamp, handleAlternativeRequestByte, this);
117. const **char** \*urlSuffix = fStreamStates[i].subsession->trackId();
118. **char**\* prevRTPInfo = rtpInfo;
119. unsigned rtpInfoSize = rtpInfoFmtSize + strlen(prevRTPInfo) + 1
120. + rtspURLSize + strlen(urlSuffix) + 5 /\*max unsigned short len\*/
121. + 10 /\*max unsigned (32-bit) len\*/
122. + 2  /\*allows for trailing \r\n at final end of string\*/;
123. rtpInfo = new **char**[rtpInfoSize];
124. sprintf(rtpInfo, rtpInfoFmt, prevRTPInfo,
125. numRTPInfoItems++ == 0 ? "" : ",", rtspURL, urlSuffix,
126. rtpSeqNum, rtpTimestamp);
127. delete[] prevRTPInfo;
128. }
129. }
130. if (numRTPInfoItems == 0) {
131. rtpInfo[0] = '\0';
132. } else {
133. unsigned rtpInfoLen = strlen(rtpInfo);
134. rtpInfo[rtpInfoLen] = '\r';
135. rtpInfo[rtpInfoLen + 1] = '\n';
136. rtpInfo[rtpInfoLen + 2] = '\0';
137. }
139. // Fill in the response:
140. snprintf((**char**\*) fResponseBuffer, sizeof fResponseBuffer,
141. "RTSP/1.0 200 OK\r\n"
142. "CSeq: %s\r\n"
143. "%s"
144. "%s"
145. "%s"
146. "Session: %08X\r\n"
147. "%s\r\n", cseq, dateHeader(), scaleHeader, rangeHeader,
148. fOurSessionId, rtpInfo);
149. delete[] rtpInfo;
150. delete[] rangeHeader;
151. delete[] scaleHeader;
152. delete[] rtspURL;
153. }

有个问题，如果这个streamToken使用的是已存在的（还记得ReuseFirstSource吗），为它再次调用startStream()时，究竟会做什么呢？你猜啊！呵呵我猜吧：应该是把这个客户端的地址和rtp/rtcp端口传给rtp server的groupSocket，rtp server自然就开始向这个客户端发送数据了。是不是这样尼？看代码吧：

1. void StreamState::startPlaying(
2. Destinations\* dests,
3. TaskFunc\* rtcpRRHandler,
4. void\* rtcpRRHandlerClientData,
5. ServerRequestAlternativeByteHandler\* serverRequestAlternativeByteHandler,
6. void\* serverRequestAlternativeByteHandlerClientData)
7. {
8. //目标ip address rtp && rtcp port
9. if (dests == NULL)
10. return;
12. //创建RTCPInstance对象，以计算和收发RTCP包
13. if (fRTCPInstance == NULL && fRTPSink != NULL) {
14. // Create (and start) a 'RTCP instance' for this RTP sink:
15. fRTCPInstance = RTCPInstance::createNew(fRTPSink->envir(), fRTCPgs,
16. fTotalBW, (unsigned **char**\*) fMaster.fCNAME, fRTPSink,
17. NULL /\* we're a server \*/);
18. // Note: This starts RTCP running automatically
19. }
21. if (dests->isTCP) {
22. //如果RTP over TCP
23. // Change RTP and RTCP to use the TCP socket instead of UDP:
24. if (fRTPSink != NULL) {
25. fRTPSink->addStreamSocket(dests->tcpSocketNum, dests->rtpChannelId);
26. fRTPSink->setServerRequestAlternativeByteHandler(
27. dests->tcpSocketNum, serverRequestAlternativeByteHandler,
28. serverRequestAlternativeByteHandlerClientData);
29. }
30. if (fRTCPInstance != NULL) {
31. fRTCPInstance->addStreamSocket(dests->tcpSocketNum,
32. dests->rtcpChannelId);
33. fRTCPInstance->setSpecificRRHandler(dests->tcpSocketNum,
34. dests->rtcpChannelId, rtcpRRHandler,
35. rtcpRRHandlerClientData);
36. }
37. } else {
38. //向RTP和RTCP的groupsocket增加这个目标
39. // Tell the RTP and RTCP 'groupsocks' about this destination
40. // (in case they don't already have it):
41. if (fRTPgs != NULL)
42. fRTPgs->addDestination(dests->addr, dests->rtpPort);
43. if (fRTCPgs != NULL)
44. fRTCPgs->addDestination(dests->addr, dests->rtcpPort);
45. if (fRTCPInstance != NULL) {
46. fRTCPInstance->setSpecificRRHandler(dests->addr.s\_addr,
47. dests->rtcpPort, rtcpRRHandler, rtcpRRHandlerClientData);
48. }
49. }
51. if (!fAreCurrentlyPlaying && fMediaSource != NULL) {
52. //如果还没有启动传输，现在启动之。
53. if (fRTPSink != NULL) {
54. fRTPSink->startPlaying(\*fMediaSource, afterPlayingStreamState,
55. this);
56. fAreCurrentlyPlaying = True;
57. } else if (fUDPSink != NULL) {
58. fUDPSink->startPlaying(\*fMediaSource, afterPlayingStreamState,this);
59. fAreCurrentlyPlaying = True;
60. }
61. }
62. }

嘿嘿，还真的这样嘀！

# 七　RTP打包与发送

rtp传送开始于函数：MediaSink::startPlaying()。想想也有道理，应是sink跟source要数据，所以从sink上调用startplaying（嘿嘿，相当于directshow的拉模式）。

看一下这个函数：

1. Boolean MediaSink::startPlaying(MediaSource& source,
2. afterPlayingFunc\* afterFunc, void\* afterClientData)
3. {
4. //参数afterFunc是在播放结束时才被调用。
5. // Make sure we're not already being played:
6. if (fSource != NULL) {
7. envir().setResultMsg("This sink is already being played");
8. return False;
9. }
11. // Make sure our source is compatible:
12. if (!sourceIsCompatibleWithUs(source)) {
13. envir().setResultMsg(
14. "MediaSink::startPlaying(): source is not compatible!");
15. return False;
16. }
17. //记下一些要使用的对象
18. fSource = (FramedSource\*) &source;
20. fAfterFunc = afterFunc;
21. fAfterClientData = afterClientData;
22. return continuePlaying();
23. }

为了进一步封装（让继承类少写一些代码），搞出了一个虚函数continuePlaying()。让我们来看一下：

1. Boolean MultiFramedRTPSink::continuePlaying() {
2. // Send the first packet.
3. // (This will also schedule any future sends.)
4. buildAndSendPacket(True);
5. return True;
6. }

MultiFramedRTPSink是与帧有关的类，其实它要求每次必须从source获得一个帧的数据，所以才叫这个name。可以看到continuePlaying()完全被buildAndSendPacket()代替。看一下buildAndSendPacket():

1. void MultiFramedRTPSink::buildAndSendPacket(Boolean isFirstPacket)
2. {
3. //此函数中主要是准备rtp包的头，为一些需要跟据实际数据改变的字段留出位置。
4. fIsFirstPacket = isFirstPacket;
6. // Set up the RTP header:
7. unsigned rtpHdr = 0x80000000; // RTP version 2; marker ('M') bit not set (by default; it can be set later)
8. rtpHdr |= (fRTPPayloadType << 16);
9. rtpHdr |= fSeqNo; // sequence number
10. fOutBuf->enqueueWord(rtpHdr);//向包中加入一个字
12. // Note where the RTP timestamp will go.
13. // (We can't fill this in until we start packing payload frames.)
14. fTimestampPosition = fOutBuf->curPacketSize();
15. fOutBuf->skipBytes(4); // leave a hole for the timestamp　在缓冲中空出时间戳的位置
17. fOutBuf->enqueueWord(SSRC());
19. // Allow for a special, payload-format-specific header following the RTP header:
20. fSpecialHeaderPosition = fOutBuf->curPacketSize();
21. fSpecialHeaderSize = specialHeaderSize();
22. fOutBuf->skipBytes(fSpecialHeaderSize);
24. // Begin packing as many (complete) frames into the packet as we can:
25. fTotalFrameSpecificHeaderSizes = 0;
26. fNoFramesLeft = False;
27. fNumFramesUsedSoFar = 0; // 一个包中已打入的帧数。
28. //头准备好了，再打包帧数据
29. packFrame();
30. }

继续看packFrame()：

1. void MultiFramedRTPSink::packFrame()
2. {
3. // First, see if we have an overflow frame that was too big for the last pkt
4. if (fOutBuf->haveOverflowData()) {
5. //如果有帧数据，则使用之。OverflowData是指上次打包时剩下的帧数据，因为包可能容纳不了一帧。
6. // Use this frame before reading a new one from the source
7. unsigned frameSize = fOutBuf->overflowDataSize();
8. struct timeval presentationTime = fOutBuf->overflowPresentationTime();
9. unsigned durationInMicroseconds =fOutBuf->overflowDurationInMicroseconds();
10. fOutBuf->useOverflowData();
12. afterGettingFrame1(frameSize, 0, presentationTime,durationInMicroseconds);
13. } else {
14. //一点帧数据都没有，跟source要吧。
15. // Normal case: we need to read a new frame from the source
16. if (fSource == NULL)
17. return;
19. //更新缓冲中的一些位置
20. fCurFrameSpecificHeaderPosition = fOutBuf->curPacketSize();
21. fCurFrameSpecificHeaderSize = frameSpecificHeaderSize();
22. fOutBuf->skipBytes(fCurFrameSpecificHeaderSize);
23. fTotalFrameSpecificHeaderSizes += fCurFrameSpecificHeaderSize;
25. //从source获取下一帧
26. fSource->getNextFrame(fOutBuf->curPtr(),//新数据存放开始的位置
27. fOutBuf->totalBytesAvailable(),//缓冲中空余的空间大小
28. afterGettingFrame,  //因为可能source中的读数据函数会被放在任务调度中，所以把获取帧后应调用的函数传给source
29. this,
30. ourHandleClosure, //这个是source结束时(比如文件读完了)要调用的函数。
31. this);
32. }
33. }

可以想像下面就是source从文件（或某个设备）中读取一帧数据，读完后返回给sink，当然不是从函数返回了，而是以调用afterGettingFrame这个回调函数的方式。所以下面看一下afterGettingFrame():

1. void MultiFramedRTPSink::afterGettingFrame(void\* clientData,
2. unsigned numBytesRead, unsigned numTruncatedBytes,
3. struct timeval presentationTime, unsigned durationInMicroseconds)
4. {
5. MultiFramedRTPSink\* sink = (MultiFramedRTPSink\*) clientData;
6. sink->afterGettingFrame1(numBytesRead, numTruncatedBytes, presentationTime,
7. durationInMicroseconds);
8. }

没什么可看的，只是过度为调用成员函数，所以afterGettingFrame1()才是重点：

1. void MultiFramedRTPSink::afterGettingFrame1(
2. unsigned frameSize,
3. unsigned numTruncatedBytes,
4. struct timeval presentationTime,
5. unsigned durationInMicroseconds)
6. {
7. if (fIsFirstPacket) {
8. // Record the fact that we're starting to play now:
9. gettimeofday(&fNextSendTime, NULL);
10. }
12. //如果给予一帧的缓冲不够大，就会发生截断一帧数据的现象。但也只能提示一下用户
13. if (numTruncatedBytes > 0) {
15. unsigned const bufferSize = fOutBuf->totalBytesAvailable();
16. envir()
17. << "MultiFramedRTPSink::afterGettingFrame1(): The input frame data was too large for our buffer size ("
18. << bufferSize
19. << ").  "
20. << numTruncatedBytes
21. << " bytes of trailing data was dropped!  Correct this by increasing \"OutPacketBuffer::maxSize\" to at least "
22. << OutPacketBuffer::maxSize + numTruncatedBytes
23. << ", \*before\* creating this 'RTPSink'.  (Current value is "
24. << OutPacketBuffer::maxSize << ".)\n";
25. }
26. unsigned curFragmentationOffset = fCurFragmentationOffset;
27. unsigned numFrameBytesToUse = frameSize;
28. unsigned overflowBytes = 0;
30. //如果包只已经打入帧数据了，并且不能再向这个包中加数据了，则把新获得的帧数据保存下来。
31. // If we have already packed one or more frames into this packet,
32. // check whether this new frame is eligible to be packed after them.
33. // (This is independent of whether the packet has enough room for this
34. // new frame; that check comes later.)
35. if (fNumFramesUsedSoFar > 0) {
36. //如果包中已有了一个帧，并且不允许再打入新的帧了，则只记录下新的帧。
37. if ((fPreviousFrameEndedFragmentation && !allowOtherFramesAfterLastFragment())
38. || !frameCanAppearAfterPacketStart(fOutBuf->curPtr(), frameSize))
39. {
40. // Save away this frame for next time:
41. numFrameBytesToUse = 0;
42. fOutBuf->setOverflowData(fOutBuf->curPacketSize(), frameSize,
43. presentationTime, durationInMicroseconds);
44. }
45. }
47. //表示当前打入的是否是上一个帧的最后一块数据。
48. fPreviousFrameEndedFragmentation = False;
50. //下面是计算获取的帧中有多少数据可以打到当前包中，剩下的数据就作为overflow数据保存下来。
51. if (numFrameBytesToUse > 0) {
52. // Check whether this frame overflows the packet
53. if (fOutBuf->wouldOverflow(frameSize)) {
54. // Don't use this frame now; instead, save it as overflow data, and
55. // send it in the next packet instead.  However, if the frame is too
56. // big to fit in a packet by itself, then we need to fragment it (and
57. // use some of it in this packet, if the payload format permits this.)
58. if (isTooBigForAPacket(frameSize)
59. && (fNumFramesUsedSoFar == 0 || allowFragmentationAfterStart())) {
60. // We need to fragment this frame, and use some of it now:
61. overflowBytes = computeOverflowForNewFrame(frameSize);
62. numFrameBytesToUse -= overflowBytes;
63. fCurFragmentationOffset += numFrameBytesToUse;
64. } else {
65. // We don't use any of this frame now:
66. overflowBytes = frameSize;
67. numFrameBytesToUse = 0;
68. }
69. fOutBuf->setOverflowData(fOutBuf->curPacketSize() + numFrameBytesToUse,
70. overflowBytes, presentationTime, durationInMicroseconds);
71. } else if (fCurFragmentationOffset > 0) {
72. // This is the last fragment of a frame that was fragmented over
73. // more than one packet.  Do any special handling for this case:
74. fCurFragmentationOffset = 0;
75. fPreviousFrameEndedFragmentation = True;
76. }
77. }
79. if (numFrameBytesToUse == 0 && frameSize > 0) {
80. //如果包中有数据并且没有新数据了，则发送之。（这种情况好像很难发生啊！）
81. // Send our packet now, because we have filled it up:
82. sendPacketIfNecessary();
83. } else {
84. //需要向包中打入数据。
86. // Use this frame in our outgoing packet:
87. unsigned **char**\* frameStart = fOutBuf->curPtr();
88. fOutBuf->increment(numFrameBytesToUse);
89. // do this now, in case "doSpecialFrameHandling()" calls "setFramePadding()" to append padding bytes
91. // Here's where any payload format specific processing gets done:
92. doSpecialFrameHandling(curFragmentationOffset, frameStart,
93. numFrameBytesToUse, presentationTime, overflowBytes);
95. ++fNumFramesUsedSoFar;
97. // Update the time at which the next packet should be sent, based
98. // on the duration of the frame that we just packed into it.
99. // However, if this frame has overflow data remaining, then don't
100. // count its duration yet.
101. if (overflowBytes == 0) {
102. fNextSendTime.tv\_usec += durationInMicroseconds;
103. fNextSendTime.tv\_sec += fNextSendTime.tv\_usec / 1000000;
104. fNextSendTime.tv\_usec %= 1000000;
105. }
107. //如果需要，就发出包，否则继续打入数据。
108. // Send our packet now if (i) it's already at our preferred size, or
109. // (ii) (heuristic) another frame of the same size as the one we just
110. //      read would overflow the packet, or
111. // (iii) it contains the last fragment of a fragmented frame, and we
112. //      don't allow anything else to follow this or
113. // (iv) one frame per packet is allowed:
114. if (fOutBuf->isPreferredSize()
115. || fOutBuf->wouldOverflow(numFrameBytesToUse)
116. || (fPreviousFrameEndedFragmentation
117. && !allowOtherFramesAfterLastFragment())
118. || !frameCanAppearAfterPacketStart(
119. fOutBuf->curPtr() - frameSize, frameSize)) {
120. // The packet is ready to be sent now
121. sendPacketIfNecessary();
122. } else {
123. // There's room for more frames; try getting another:
124. packFrame();
125. }
126. }
127. }

看一下发送数据的函数：

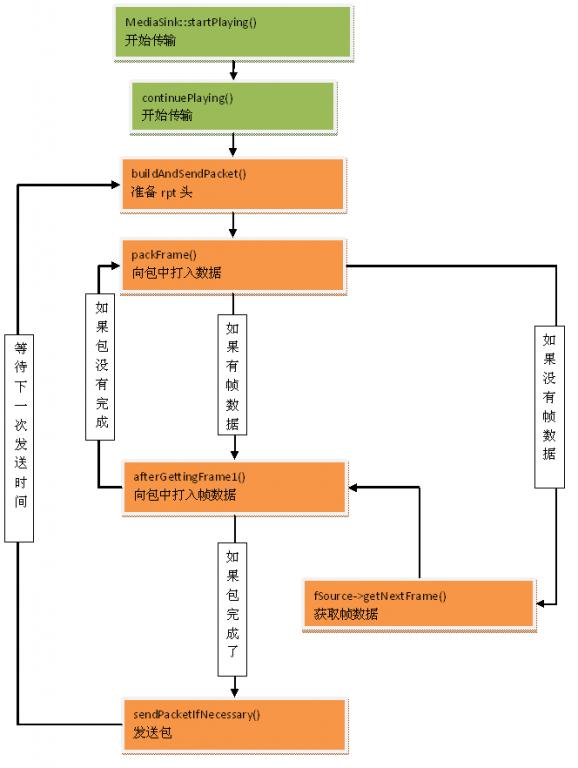
1. void MultiFramedRTPSink::sendPacketIfNecessary()
2. {
3. //发送包
4. if (fNumFramesUsedSoFar > 0) {
5. // Send the packet:
6. #ifdef TEST\_LOSS
7. if ((our\_random()%10) != 0) // simulate 10% packet loss #####
8. #endif
9. if (!fRTPInterface.sendPacket(fOutBuf->packet(),fOutBuf->curPacketSize())) {
10. // if failure handler has been specified, call it
11. if (fOnSendErrorFunc != NULL)
12. (\*fOnSendErrorFunc)(fOnSendErrorData);
13. }
14. ++fPacketCount;
15. fTotalOctetCount += fOutBuf->curPacketSize();
16. fOctetCount += fOutBuf->curPacketSize() - rtpHeaderSize
17. - fSpecialHeaderSize - fTotalFrameSpecificHeaderSizes;
19. ++fSeqNo; // for next time
20. }

23. //如果还有剩余数据，则调整缓冲区
24. if (fOutBuf->haveOverflowData()
25. && fOutBuf->totalBytesAvailable() > fOutBuf->totalBufferSize() / 2) {
26. // Efficiency hack: Reset the packet start pointer to just in front of
27. // the overflow data (allowing for the RTP header and special headers),
28. // so that we probably don't have to "memmove()" the overflow data
29. // into place when building the next packet:
30. unsigned newPacketStart = fOutBuf->curPacketSize()-
31. (rtpHeaderSize + fSpecialHeaderSize + frameSpecificHeaderSize());
32. fOutBuf->adjustPacketStart(newPacketStart);
33. } else {
34. // Normal case: Reset the packet start pointer back to the start:
35. fOutBuf->resetPacketStart();
36. }
37. fOutBuf->resetOffset();
38. fNumFramesUsedSoFar = 0;
40. if (fNoFramesLeft) {
41. //如果再没有数据了，则结束之
42. // We're done:
43. onSourceClosure(this);
44. } else {
45. //如果还有数据，则在下一次需要发送的时间再次打包发送。
46. // We have more frames left to send.  Figure out when the next frame
47. // is due to start playing, then make sure that we wait this long before
48. // sending the next packet.
49. struct timeval timeNow;
50. gettimeofday(&timeNow, NULL);
51. **int** secsDiff = fNextSendTime.tv\_sec - timeNow.tv\_sec;
52. int64\_t uSecondsToGo = secsDiff \* 1000000
53. + (fNextSendTime.tv\_usec - timeNow.tv\_usec);
54. if (uSecondsToGo < 0 || secsDiff < 0) { // sanity check: Make sure that the time-to-delay is non-negative:
55. uSecondsToGo = 0;
56. }
58. // Delay this amount of time:
59. nextTask() = envir().taskScheduler().scheduleDelayedTask(uSecondsToGo,
60. (TaskFunc\*) sendNext, this);
61. }
62. }

可以看到为了延迟包的发送，使用了delay task来执行下次打包发送任务。

sendNext()中又调用了buildAndSendPacket()函数，呵呵，又是一个圈圈。

总结一下调用过程：



最后，再说明一下包缓冲区的使用：

MultiFramedRTPSink中的帧数据和包缓冲区共用一个，只是用一些额外的变量指明缓冲区中属于包的部分以及属于帧数据的部分（包以外的数据叫做overflow data）。它有时会把overflow data以mem move的方式移到包开始的位置，有时把包的开始位置直接设置到overflow data开始的地方。那么这个缓冲的大小是怎样确定的呢？是跟据调用者指定的的一个最大的包的大小+60000算出的。这个地方把我搞胡涂了：如果一次从source获取一个帧的话，那这个缓冲应设为不小于最大的一个帧的大小才是，为何是按包的大小设置呢？可以看到，当缓冲不够时只是提示一下：

1. if (numTruncatedBytes > 0) {
2. unsigned const bufferSize = fOutBuf->totalBytesAvailable();
3. envir()
4. << "MultiFramedRTPSink::afterGettingFrame1(): The input frame data was too large for our buffer size ("
5. << bufferSize
6. << ").  "
7. << numTruncatedBytes
8. << " bytes of trailing data was dropped!  Correct this by increasing \"OutPacketBuffer::maxSize\" to at least "
9. << OutPacketBuffer::maxSize + numTruncatedBytes
10. << ", \*before\* creating this 'RTPSink'.  (Current value is "
11. << OutPacketBuffer::maxSize << ".)\n";
12. }

当然此时不会出错，但有可能导致时间戳计算不准，或增加时间戳计算与source端处理的复杂性(因为一次取一帧时间戳是很好计算的)。

# 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：

1. void getSDPDescription(RTSPClient::responseHandler\* afterFunc)
2. {
3. ourRTSPClient->sendDescribeCommand(afterFunc, ourAuthenticator);
4. }
5. unsigned RTSPClient::sendDescribeCommand(responseHandler\* responseHandler,
6. Authenticator\* authenticator)
7. {
8. if (authenticator != NULL)
9. 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：

1. void continueAfterDESCRIBE(RTSPClient\*, **int** resultCode, **char**\* resultString)
2. {
3. **char**\* sdpDescription = resultString;
4. //跟据SDP创建MediaSession。
5. // Create a media session object from this SDP description:
6. session = MediaSession::createNew(\*env, sdpDescription);
7. delete[] sdpDescription;
9. // 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);
26. // Set the RTP source's OS socket buffer size as appropriate - either if we were explicitly asked (using -B),
27. // or if the desired FileSink buffer size happens to be larger than the current OS socket buffer size.
28. // (The latter case is a heuristic, on the assumption that if the user asked for a large FileSink buffer size,
29. // then the input data rate may be large enough to justify increasing the OS socket buffer size also.)
30. **int** socketNum = subsession->rtpSource()->RTPgs()->socketNum();
31. unsigned curBufferSize = getReceiveBufferSize(\*env,socketNum);
32. if (socketInputBufferSize > 0　|| fileSinkBufferSize > curBufferSize) {
33. unsigned newBufferSize =　socketInputBufferSize > 0 ?
34. socketInputBufferSize : fileSinkBufferSize;
35. newBufferSize = setReceiveBufferTo(\*env, socketNum, newBufferSize);
36. if (socketInputBufferSize > 0) { // The user explicitly asked for the new socket buffer size; announce it:
37. \*env
38. << "Changed socket receive buffer size for the \""
39. << subsession->mediumName() << "/"
40. << subsession->codecName()
41. << "\" subsession from " << curBufferSize
42. << " to " << newBufferSize << " bytes\n";
43. }
44. }
45. }
46. }
47. }
48. if (!madeProgress)
49. shutdown();
51. // Perform additional 'setup' on each subsession, before playing them:
52. //下一步就是发送SETUP请求了。需要为每个Track分别发送一次。
53. setupStreams();
54. }

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:

1. MediaSession\* MediaSession::createNew(UsageEnvironment& env,**char** const\* sdpDescription)
2. {
3. MediaSession\* newSession = new MediaSession(env);
4. if (newSession != NULL) {
5. if (!newSession->initializeWithSDP(sdpDescription)) {
6. delete newSession;
7. return NULL;
8. }
9. }
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:

1. Boolean MediaSubsession::initiate(**int** useSpecialRTPoffset)
2. {
3. if (fReadSource != NULL)
4. return True; // has already been initiated
6. do {
7. if (fCodecName == NULL) {
8. env().setResultMsg("Codec is unspecified");
9. break;
10. }
12. //创建RTP/RTCP sockets
13. // Create RTP and RTCP 'Groupsocks' on which to receive incoming data.
14. // (Groupsocks will work even for unicast addresses)
15. struct in\_addr tempAddr;
16. tempAddr.s\_addr = connectionEndpointAddress();
17. // This could get changed later, as a result of a RTSP "SETUP"
19. if (fClientPortNum != 0) {
20. //当server端指定了建议的client端口
21. // The sockets' port numbers were specified for us.  Use these:
22. fClientPortNum = fClientPortNum & ~1; // even
23. if (isSSM()) {
24. fRTPSocket = new Groupsock(env(), tempAddr, fSourceFilterAddr,
25. fClientPortNum);
26. } else {
27. fRTPSocket = new Groupsock(env(), tempAddr, fClientPortNum,
28. 255);
29. }
30. if (fRTPSocket == NULL) {
31. env().setResultMsg("Failed to create RTP socket");
32. break;
33. }
35. // Set our RTCP port to be the RTP port +1
36. portNumBits const rtcpPortNum = fClientPortNum | 1;
37. if (isSSM()) {
38. fRTCPSocket = new Groupsock(env(), tempAddr, fSourceFilterAddr,
39. rtcpPortNum);
40. } else {
41. fRTCPSocket = new Groupsock(env(), tempAddr, rtcpPortNum, 255);
42. }
43. if (fRTCPSocket == NULL) {
44. **char** tmpBuf[100];
45. sprintf(tmpBuf, "Failed to create RTCP socket (port %d)",
46. rtcpPortNum);
47. env().setResultMsg(tmpBuf);
48. break;
49. }
50. } else {
51. //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
52. //RTP/RTCP的端口号不是要连续吗？还记得不？
53. // Port numbers were not specified in advance, so we use ephemeral port numbers.
54. // Create sockets until we get a port-number pair (even: RTP; even+1: RTCP).
55. // We need to make sure that we don't keep trying to use the same bad port numbers over and over again.
56. // so we store bad sockets in a table, and delete them all when we're done.
57. HashTable\* socketHashTable = HashTable::create(ONE\_WORD\_HASH\_KEYS);
58. if (socketHashTable == NULL)
59. break;
60. Boolean success = False;
61. NoReuse dummy; // ensures that our new ephemeral port number won't be one that's already in use
63. while (1) {
64. // Create a new socket:
65. if (isSSM()) {
66. fRTPSocket = new Groupsock(env(), tempAddr,
67. fSourceFilterAddr, 0);
68. } else {
69. fRTPSocket = new Groupsock(env(), tempAddr, 0, 255);
70. }
71. if (fRTPSocket == NULL) {
72. env().setResultMsg(
73. "MediaSession::initiate(): unable to create RTP and RTCP sockets");
74. break;
75. }
77. // Get the client port number, and check whether it's even (for RTP):
78. Port clientPort(0);
79. if (!getSourcePort(env(), fRTPSocket->socketNum(),
80. clientPort)) {
81. break;
82. }
83. fClientPortNum = ntohs(clientPort.num());
84. if ((fClientPortNum & 1) != 0) { // it's odd
85. // Record this socket in our table, and keep trying:
86. unsigned key = (unsigned) fClientPortNum;
87. Groupsock\* existing = (Groupsock\*) socketHashTable->Add(
88. (**char** const\*) key, fRTPSocket);
89. delete existing; // in case it wasn't NULL
90. continue;
91. }
93. // Make sure we can use the next (i.e., odd) port number, for RTCP:
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 RTCP socket (perhaps that port number's already in use elsewhere?).
108. delete fRTCPSocket;
110. // Record the first socket in our table, and keep trying:
111. unsigned key = (unsigned) fClientPortNum;
112. Groupsock\* existing = (Groupsock\*) socketHashTable->Add(
113. (**char** const\*) key, fRTPSocket);
114. delete existing; // in case it wasn't NULL
115. continue;
116. }
117. }
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;
126. if (!success)
127. break; // a fatal error occurred trying to create the RTP and RTCP sockets; we can't continue
128. }
130. // Try to use a big receive buffer for RTP - at least 0.1 second of
131. // specified bandwidth and at least 50 KB
132. unsigned rtpBufSize = fBandwidth \* 25 / 2; // 1 kbps \* 0.1 s = 12.5 bytes
133. if (rtpBufSize < 50 \* 1024)
134. rtpBufSize = 50 \* 1024;
135. increaseReceiveBufferTo(env(), fRTPSocket->socketNum(), rtpBufSize);
137. // ASSERT: fRTPSocket != NULL && fRTCPSocket != NULL
138. if (isSSM()) {
139. // Special case for RTCP SSM: Send RTCP packets back to the source via unicast:
140. fRTCPSocket->changeDestinationParameters(fSourceFilterAddr, 0, ~0);
141. }
143. //创建RTPSource的地方
144. // Create "fRTPSource" and "fReadSource":
145. if (!createSourceObjects(useSpecialRTPoffset))
146. break;
148. if (fReadSource == NULL) {
149. env().setResultMsg("Failed to create read source");
150. break;
151. }
153. // Finally, create our RTCP instance. (It starts running automatically)
154. if (fRTPSource != NULL) {
155. // If bandwidth is specified, use it and add 5% for RTCP overhead.
156. // Otherwise make a guess at 500 kbps.
157. unsigned totSessionBandwidth =
158. fBandwidth ? fBandwidth + fBandwidth / 20 : 500;
159. fRTCPInstance = RTCPInstance::createNew(env(), fRTCPSocket,
160. totSessionBandwidth, (unsigned **char** const\*) fParent.CNAME(),
161. NULL /\* we're a client \*/, fRTPSource);
162. if (fRTCPInstance == NULL) {
163. env().setResultMsg("Failed to create RTCP instance");
164. break;
165. }
166. }
168. return True;
169. } while (0);
171. //失败时执行到这里
172. delete fRTPSocket;
173. fRTPSocket = NULL;
174. delete fRTCPSocket;
175. fRTCPSocket = NULL;
176. Medium::close(fRTCPInstance);
177. fRTCPInstance = 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：

1. Boolean MediaSubsession::createSourceObjects(**int** useSpecialRTPoffset)
2. {
3. do {
4. // First, check "fProtocolName"
5. if (strcmp(fProtocolName, "UDP") == 0) {
6. // A UDP-packetized stream (\*not\* a RTP stream)
7. fReadSource = BasicUDPSource::createNew(env(), fRTPSocket);
8. fRTPSource = NULL; // Note!
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;
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;
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;
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. }
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. }
167. return True;
168. } while (0);
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（），那么就来探索一下setupStreams():

1. void setupStreams()
2. {
3. static MediaSubsessionIterator\* setupIter = NULL;
4. if (setupIter == NULL)
5. setupIter = new MediaSubsessionIterator(\*session);
7. //每次调用此函数只为一个Subsession发出SETUP请求。
8. while ((subsession = setupIter->next()) != NULL) {
9. // We have another subsession left to set up:
10. if (subsession->clientPortNum() == 0)
11. continue; // port # was not set
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。
15. <span style="white-space: pre;">        </span>//直到处理完所有的SubSession
16. setupSubsession(subsession, streamUsingTCP, continueAfterSETUP);
17. return;
18. }
20. //执行到这里时，已循环完所有的SubSession了
21. // We're done setting up subsessions.
22. delete setupIter;
23. if (!madeProgress)
24. shutdown();
26. //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
27. //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。
28. // Create output files:
29. if (createReceivers) {
30. if (outputQuickTimeFile) {
31. // Create a "QuickTimeFileSink", to write to 'stdout':
32. qtOut = QuickTimeFileSink::createNew(\*env, \*session, "stdout",
33. fileSinkBufferSize, movieWidth, movieHeight, movieFPS,
34. packetLossCompensate, syncStreams, generateHintTracks,
35. generateMP4Format);
36. if (qtOut == NULL) {
37. \*env << "Failed to create QuickTime file sink for stdout: "
38. << env->getResultMsg();
39. shutdown();
40. }
42. qtOut->startPlaying(sessionAfterPlaying, NULL);
43. } else if (outputAVIFile) {
44. // Create an "AVIFileSink", to write to 'stdout':
45. aviOut = AVIFileSink::createNew(\*env, \*session, "stdout",
46. fileSinkBufferSize, movieWidth, movieHeight, movieFPS,
47. packetLossCompensate);
48. if (aviOut == NULL) {
49. \*env << "Failed to create AVI file sink for stdout: "
50. << env->getResultMsg();
51. shutdown();
52. }
54. aviOut->startPlaying(sessionAfterPlaying, NULL);
55. } else {
56. // Create and start "FileSink"s for each subsession:
57. madeProgress = False;
58. MediaSubsessionIterator iter(\*session);
59. while ((subsession = iter.next()) != NULL) {
60. if (subsession->readSource() == NULL)
61. continue; // was not initiated
63. // Create an output file for each desired stream:
64. **char** outFileName[1000];
65. if (singleMedium == NULL) {
66. // Output file name is
67. //     "<filename-prefix><medium\_name>-<codec\_name>-<counter>"
68. static unsigned streamCounter = 0;
69. snprintf(outFileName, sizeof outFileName, "%s%s-%s-%d",
70. fileNamePrefix, subsession->mediumName(),
71. subsession->codecName(), ++streamCounter);
72. } else {
73. sprintf(outFileName, "stdout");
74. }
75. FileSink\* fileSink;
76. if (strcmp(subsession->mediumName(), "audio") == 0
77. && (strcmp(subsession->codecName(), "AMR") == 0
78. || strcmp(subsession->codecName(), "AMR-WB")
79. == 0)) {
80. // For AMR audio streams, we use a special sink that inserts AMR frame hdrs:
81. fileSink = AMRAudioFileSink::createNew(\*env, outFileName,
82. fileSinkBufferSize, oneFilePerFrame);
83. } else if (strcmp(subsession->mediumName(), "video") == 0
84. && (strcmp(subsession->codecName(), "H264") == 0)) {
85. // For H.264 video stream, we use a special sink that insert start\_codes:
86. fileSink = H264VideoFileSink::createNew(\*env, outFileName,
87. subsession->fmtp\_spropparametersets(),
88. fileSinkBufferSize, oneFilePerFrame);
89. } else {
90. // Normal case:
91. fileSink = FileSink::createNew(\*env, outFileName,
92. fileSinkBufferSize, oneFilePerFrame);
93. }
94. subsession->sink = fileSink;
95. if (subsession->sink == NULL) {
96. \*env << "Failed to create FileSink for \"" << outFileName
97. << "\": " << env->getResultMsg() << "\n";
98. } else {
99. if (singleMedium == NULL) {
100. \*env << "Created output file: \"" << outFileName
101. << "\"\n";
102. } else {
103. \*env << "Outputting data from the \""
104. << subsession->mediumName() << "/"
105. << subsession->codecName()
106. << "\" subsession to 'stdout'\n";
107. }
109. if (strcmp(subsession->mediumName(), "video") == 0
110. && strcmp(subsession->codecName(), "MP4V-ES") == 0 &&
111. subsession->fmtp\_config() != NULL) {
112. // For MPEG-4 video RTP streams, the 'config' information
113. // from the SDP description contains useful VOL etc. headers.
114. // Insert this data at the front of the output file:
115. unsigned                    configLen;
116. unsigned **char**\* configData
117. = parseGeneralConfigStr(subsession->fmtp\_config(), configLen);
118. struct timeval timeNow;
119. gettimeofday(&timeNow, NULL);
120. fileSink->addData(configData, configLen, timeNow);
121. delete[] configData;
122. }
124. //开始传输
125. subsession->sink->startPlaying(\*(subsession->readSource()),
126. subsessionAfterPlaying, subsession);
128. // Also set a handler to be called if a RTCP "BYE" arrives
129. // for this subsession:
130. if (subsession->rtcpInstance() != NULL) {
131. subsession->rtcpInstance()->setByeHandler(
132. subsessionByeHandler, subsession);
133. }
135. madeProgress = True;
136. }
137. }
138. if (!madeProgress)
139. shutdown();
140. }
141. }
143. // Finally, start playing each subsession, to start the data flow:
144. if (duration == 0) {
145. if (scale > 0)
146. duration = session->playEndTime() - initialSeekTime; // use SDP end time
147. else if (scale < 0)
148. duration = initialSeekTime;
149. }
150. if (duration < 0)
151. duration = 0.0;
153. endTime = initialSeekTime;
154. if (scale > 0) {
155. if (duration <= 0)
156. endTime = -1.0f;
157. else
158. endTime = initialSeekTime + duration;
159. } else {
160. endTime = initialSeekTime - duration;
161. if (endTime < 0)
162. endTime = 0.0f;
163. }
165. //发送PLAY请求，之后才能从Server端接收数据
166. startPlayingSession(session, initialSeekTime, endTime, scale,
167. continueAfterPLAY);
168. }

仔细看看注释，应很容易了解此函数。

# 九 h264 RTP传输详解

## 1

前几章对Server端的介绍中有个比较重要的问题没有仔细探究：如何打开文件并获得其SDP信息。我们就从这里入手吧。

当RTSPServer收到对某个媒体的DESCRIBE请求时，它会找到对应的ServerMediaSession,调用ServerMediaSession::generateSDPDescription()。generateSDPDescription()中会遍历调用ServerMediaSession中所有的调用ServerMediaSubsession，通过subsession->sdpLines()取得每个Subsession的sdp，合并成一个完整的SDP返回之。  
我们几乎可以断定，文件的打开和分析应该是在每个Subsession的sdpLines()函数中完成的，看看这个函数：

1. **char** const\* OnDemandServerMediaSubsession::sdpLines()
2. {
3. if (fSDPLines == NULL) {
4. // We need to construct a set of SDP lines that describe this
5. // subsession (as a unicast stream).  To do so, we first create
6. // dummy (unused) source and "RTPSink" objects,
7. // whose parameters we use for the SDP lines:
8. unsigned estBitrate;
9. FramedSource\* inputSource = createNewStreamSource(0, estBitrate);
10. if (inputSource == NULL)
11. return NULL; // file not found
13. struct in\_addr dummyAddr;
14. dummyAddr.s\_addr = 0;
15. Groupsock dummyGroupsock(envir(), dummyAddr, 0, 0);
16. unsigned **char** rtpPayloadType = 96 + trackNumber() - 1; // if dynamic
17. RTPSink\* dummyRTPSink = createNewRTPSink(&dummyGroupsock,
18. rtpPayloadType, inputSource);
20. setSDPLinesFromRTPSink(dummyRTPSink, inputSource, estBitrate);
21. Medium::close(dummyRTPSink);
22. closeStreamSource(inputSource);
23. }
25. return fSDPLines;
26. }

其所为如是：Subsession中直接保存了对应媒体文件的SDP，但是在第一次获取时fSDPLines为NULL，所以需先获取fSDPLines。其做法比较费事，竟然是建了临时的Source和RTPSink，把它们连接成一个StreamToken，Playing一段时间之后才取得了fSDPLines。createNewStreamSource()和createNewRTPSink()都是虚函数，所以此处创建的source和sink都是继承类指定的，我们分析的是H264，也就是H264VideoFileServerMediaSubsession所指定的，来看一下这两个函数：

1. FramedSource\* H264VideoFileServerMediaSubsession::createNewStreamSource(
2. unsigned /\*clientSessionId\*/,
3. unsigned& estBitrate)
4. {
5. estBitrate = 500; // kbps, estimate
7. // Create the video source:
8. ByteStreamFileSource\* fileSource = ByteStreamFileSource::createNew(envir(),
9. fFileName);
10. if (fileSource == NULL)
11. return NULL;
12. fFileSize = fileSource->fileSize();
14. // Create a framer for the Video Elementary Stream:
15. return H264VideoStreamFramer::createNew(envir(), fileSource);
16. }
18. RTPSink\* H264VideoFileServerMediaSubsession::createNewRTPSink(
19. Groupsock\* rtpGroupsock,
20. unsigned **char** rtpPayloadTypeIfDynamic,
21. FramedSource\* /\*inputSource\*/)
22. {
23. return H264VideoRTPSink::createNew(envir(), rtpGroupsock,
24. rtpPayloadTypeIfDynamic);
25. }

可以看到，分别创建了H264VideoStreamFramer和H264VideoRTPSink。可以肯定H264VideoStreamFramer也是一个Source，但它内部又利用了另一个source--ByteStreamFileSource。后面会分析为什么要这样做，这里先不要管它。还没有看到真正打开文件的代码，继续探索：

1. void OnDemandServerMediaSubsession::setSDPLinesFromRTPSink(
2. RTPSink\* rtpSink,
3. FramedSource\* inputSource,
4. unsigned estBitrate)
5. {
6. if (rtpSink == NULL)
7. return;
9. **char** const\* mediaType = rtpSink->sdpMediaType();
10. unsigned **char** rtpPayloadType = rtpSink->rtpPayloadType();
11. struct in\_addr serverAddrForSDP;
12. serverAddrForSDP.s\_addr = fServerAddressForSDP;
13. **char**\* const ipAddressStr = strDup(our\_inet\_ntoa(serverAddrForSDP));
14. **char**\* rtpmapLine = rtpSink->rtpmapLine();
15. **char** const\* rangeLine = rangeSDPLine();
16. **char** const\* auxSDPLine = getAuxSDPLine(rtpSink, inputSource);
17. if (auxSDPLine == NULL)
18. auxSDPLine = "";
20. **char** const\* const sdpFmt = "m=%s %u RTP/AVP %d\r\n"
21. "c=IN IP4 %s\r\n"
22. "b=AS:%u\r\n"
23. "%s"
24. "%s"
25. "%s"
26. "a=control:%s\r\n";
27. unsigned sdpFmtSize = strlen(sdpFmt) + strlen(mediaType) + 5 /\* max short len \*/
28. + 3 /\* max char len \*/
29. + strlen(ipAddressStr) + 20 /\* max int len \*/
30. + strlen(rtpmapLine) + strlen(rangeLine) + strlen(auxSDPLine)
31. + strlen(trackId());
32. **char**\* sdpLines = new **char**[sdpFmtSize];
33. sprintf(sdpLines, sdpFmt, mediaType, // m= <media>
34. fPortNumForSDP, // m= <port>
35. rtpPayloadType, // m= <fmt list>
36. ipAddressStr, // c= address
37. estBitrate, // b=AS:<bandwidth>
38. rtpmapLine, // a=rtpmap:... (if present)
39. rangeLine, // a=range:... (if present)
40. auxSDPLine, // optional extra SDP line
41. trackId()); // a=control:<track-id>
42. delete[] (**char**\*) rangeLine;
43. delete[] rtpmapLine;
44. delete[] ipAddressStr;
46. fSDPLines = strDup(sdpLines);
47. delete[] sdpLines;
48. }

此函数中取得Subsession的sdp并保存到fSDPLines。打开文件应在rtpSink->rtpmapLine()甚至是Source创建时已经做了。我们不防先把它放一放，而是先把SDP的获取过程搞个通透。所以把焦点集中到getAuxSDPLine()上。

1. **char** const\* OnDemandServerMediaSubsession::getAuxSDPLine(
2. RTPSink\* rtpSink,
3. FramedSource\* /\*inputSource\*/)
4. {
5. // Default implementation:
6. return rtpSink == NULL ? NULL : rtpSink->auxSDPLine();
7. }

很简单，调用了rtpSink->auxSDPLine()那么我们要看H264VideoRTPSink::auxSDPLine()：不用看了，很简单，取得source 中保存的PPS,SPS等形成a=fmpt行。但事实上并没有这么简单，H264VideoFileServerMediaSubsession重写了getAuxSDPLine()！如果不重写，则说明auxSDPLine已经在前面分析文件时获得了，那么既然重写，就说明前面没有获取到，只能在这个函数中重写。look　H264VideoFileServerMediaSubsession中这个函数:

1. **char** const\* H264VideoFileServerMediaSubsession::getAuxSDPLine(
2. RTPSink\* rtpSink,
3. FramedSource\* inputSource)
4. {
5. if (fAuxSDPLine != NULL)
6. return fAuxSDPLine; // it's already been set up (for a previous client)
8. if (fDummyRTPSink == NULL) { // we're not already setting it up for another, concurrent stream
9. // Note: For H264 video files, the 'config' information ("profile-level-id" and "sprop-parameter-sets") isn't known
10. // until we start reading the file.  This means that "rtpSink"s "auxSDPLine()" will be NULL initially,
11. // and we need to start reading data from our file until this changes.
12. fDummyRTPSink = rtpSink;
14. // Start reading the file:
15. fDummyRTPSink->startPlaying(\*inputSource, afterPlayingDummy, this);
17. // Check whether the sink's 'auxSDPLine()' is ready:
18. checkForAuxSDPLine(this);
19. }
21. envir().taskScheduler().doEventLoop(&fDoneFlag);
23. return fAuxSDPLine;
24. }

注释里面解释得很清楚，H264不能在文件头中取得PPS/SPS，必须在播放一下后（当然，它是一个原始流文件，没有文件头）才行。也就是说不能从rtpSink中取得了。为了保证在函数退出前能取得AuxSDP，把大循环搬到这里来了。afterPlayingDummy()是在播放结束也就是取得aux sdp之后执行。在大循环之前的checkForAuxSDPLine()做了什么呢？

void H264VideoFileServerMediaSubsession::checkForAuxSDPLine1()

{

char const\* dasl;

if (fAuxSDPLine != NULL) {

// Signal the event loop that we're done:

setDoneFlag();

} else if (fDummyRTPSink != NULL

&& (dasl = fDummyRTPSink->auxSDPLine()) != NULL) {

fAuxSDPLine = strDup(dasl);

fDummyRTPSink = NULL;

// Signal the event loop that we're done:

setDoneFlag();

} else {

// try again after a brief delay:

int uSecsToDelay = 100000; // 100 ms

nextTask() = envir().taskScheduler().scheduleDelayedTask(uSecsToDelay,

(TaskFunc\*) checkForAuxSDPLine, this);

}

}

它检查是否已取得Aux sdp，如果取得了，设置结束标志，直接返回。如果没有，就检查是否sink中已取得了aux sdp，如果是，也设置结束标志，返回。如果还没有取得，则把这个检查函数做为delay task加入计划任务中。每100毫秒检查一次，每检查一次主要就是调用一次fDummyRTPSink->auxSDPLine()。大循环在检测到fDoneFlag改变时停止，此时已取得了aux sdp。但是如果直到文件结束也没有得到aux sdp，则afterPlayingDummy（）被执行，在其中停止掉这个大循环。然后在父Subsession类中关掉这些临时的source和sink。在直正播放时重新创建。

## 2

一章并没有把打开文件分析文件的代码找到，因为发现它隐藏得比较深，而且H264的Source又有多个，形成了连环计。所以此章中就将文件处理与H264的Source们并在一起分析吧。  
从哪里开始呢？从source开始吧!为什么要从它开始呢？我就想从这里开始，行了吧？

1. FramedSource\* H264VideoFileServerMediaSubsession::createNewStreamSource(
2. unsigned /\*clientSessionId\*/,
3. unsigned& estBitrate)
4. {
5. estBitrate = 500; // kbps, estimate
7. // Create the video source:
8. ByteStreamFileSource\* fileSource = ByteStreamFileSource::createNew(envir(),
9. fFileName);
10. if (fileSource == NULL)
11. return NULL;
12. fFileSize = fileSource->fileSize();
14. // Create a framer for the Video Elementary Stream:
15. return H264VideoStreamFramer::createNew(envir(), fileSource);
16. }

先创建一个ByteStreamFileSource，显然这是一个从文件按字节读取数据的source，没什么可细说的。但是，打开文件，读写文件操作的确就在其中。最终来处理h264文件，分析其格式，解析出帧或nal的应是这个source: H264VideoStreamFramer。打开文件的地方找到了，但分析文件的代码才是更有价值的。那我们只能来看H264VideoStreamFramer。  
H264VideoStreamFramer继承自MPEGVideoStreamFramer，MPEGVideoStreamFramer继承自FramedFilter,FramedFilter继承自FramedSource。  
啊，中间又冒出个Filter。看到它，是不是联想到了DirectShow的filter？或者说Photoshop中的filter？它们的意义应该都差不多吧？即插入到source和render(sink)之间的处理媒体数据的东东？如果这样理解，还是更接近于photoshop中的概念。唉，说实话，我估计自己说的也不全对，反正就这样认识吧，谬不了一千里。既然我们这样认识了，那么我们就有理由相信可能会出现多个filter们一个连一个,然后高唱：手牵着脚脚牵着手一起向前走...  
H264VideoStreamFramer继承自MPEGVideoStreamFramer，MPEGVideoStreamFramer比较简单，它只是把一些工作交给了MPEGVideoStreamParser（又出来个parser，这可是个新东西哦，先不要管它吧），重点来看一下。  
构造函数：

1. H264VideoStreamFramer::H264VideoStreamFramer(UsageEnvironment& env,
2. FramedSource\* inputSource,
3. Boolean createParser,
4. Boolean includeStartCodeInOutput)
5. : MPEGVideoStreamFramer(env, inputSource),
6. fIncludeStartCodeInOutput(includeStartCodeInOutput),
7. fLastSeenSPS(NULL),
8. fLastSeenSPSSize(0),
9. fLastSeenPPS(NULL),
10. fLastSeenPPSSize(0)
11. {
12. fParser = createParser ?
13. new H264VideoStreamParser(this, inputSource,
14. includeStartCodeInOutput) : NULL;
15. fNextPresentationTime = fPresentationTimeBase;
16. fFrameRate = 25.0; // We assume a frame rate of 25 fps,
17. //unless we learn otherwise (from parsing a Sequence Parameter Set NAL unit)
18. }

由于createParser肯定为真，所以主要内容是创建了H264VideoStreamParser对象（先不管这个parser）。  
其它的函数就没什么可看的了，都集中在所保存的PPS与SPS上。看来分析工作移到了H264VideoStreamParser，Parser嘛，就是分析器。分析器的基类是StreamParser。StreamParser做了不少的工作，那我们就先搞明白StreamParser做了哪些工作吧，并且可能为继承者提供什么样的调用框架呢？.....我看完了，呵呵。直接说分析结果吧：  
StreamParser的主要工作是实现了对数据以位为单位进行访问。因为在处理媒体格式时，按位分析是很常见的情况。这两个函数skipBits(unsigned　numBits)和unsigned getBits(unsigned numBits)很明显是基于位的操作。unsigned char\* fBank[2]这个变量中的两个缓冲区被轮换使用。这个类中保存了一个Source，理所当然地它应该保存ByteStreamFileSource的实例，而不是FramedFilter的。那些getBytes()或getBits()最终会导致读文件的操作。从文件读取一次数据后，StreamParser::afterGettingBytes1（）被调用，StreamParser::afterGettingBytes1（）中做一点简单的工作后便调用fClientContinueFunc这个回调函数。fClientContinueFunc可能指向Frame的函数体也可能是指向RtpSink的函数体－－因为Framer完全可以把RtpSink的函数体传给Parser。至于到底指向哪个，只能在进一步分析之后才得知。

下面再来分析StreamParser的儿子:MPEGVideoStreamParser。

1. MPEGVideoStreamParser::MPEGVideoStreamParser(
2. MPEGVideoStreamFramer\* usingSource,
3. FramedSource\* inputSource)
4. : StreamParser(inputSource,
5. FramedSource::handleClosure,
6. usingSource,
7. &MPEGVideoStreamFramer::continueReadProcessing,
8. usingSource),
9. fUsingSource(usingSource)
10. {
11. }

MPEGVideoStreamParser的构造函数中有很多有意思的东西。  
首先参数usingSource是什么意思？表示正在使用这个Parser的Source? inputSource 很明确，就是能获取数据的source，也就是 ByteStreamFileSource。而且很明显的，StreamParser中保存的source是ByteStreamFileSource。从传入给StreamParser的回调函数以及它们的参数可以看出，这些回调函数全是指向的StreamParser的子类的函数（为啥不用虚函数的方式？哦，回调函数全是静态函数，不能成为虚函数）。这说明在每读一次数据后，MPEGVideoStreamFramer::continueReadProcessing()被调用，在其中对帧进行界定和分析，完成后再调用RTPSink的相应函数，RTPSink中对帧进行打包和发送（还记得吗，不记得了请回头看以前的章节）。  
MPEGVideoStreamParser的fTo是RTPSink传入的缓冲指针，其saveByte(),save4Bytes()是把数据从StreamParser的缓冲把数据复制到fTo中，是给继承类使用的。saveToNextCode()是复制数据直到遇到一个同步字节串（比如h264中分隔nal的那一陀东东，当然此处的跟h264还不一样），也是给继承类使用的。纯虚函数parse()很明显是留继承类去写帧分析代码的地方。registerReadInterest()被调用者用来告诉MPEGVideoStreamParser其接收帧的缓冲地址与容量。  
现在应该来分析一下MPEGVideoStreamFramer，以明确MPEGVideoStreamFramer与MPEGVideoStreamParser是怎样配合的。  
MPEGVideoStreamFramer中用到Parser的重要的函数只有两个，一是：

1. void MPEGVideoStreamFramer::doGetNextFrame()
2. {
3. fParser->registerReadInterest(fTo, fMaxSize);
4. continueReadProcessing();
5. }

很简单，只是告诉了Parser保存帧的缓冲和缓冲的大小，然后执行continueReadProcessing()，那么来看一下continueReadProcessing()：

1. void MPEGVideoStreamFramer::continueReadProcessing()
2. {
3. unsigned acquiredFrameSize = fParser->parse();
4. if (acquiredFrameSize > 0) {
5. // We were able to acquire a frame from the input.
6. // It has already been copied to the reader's space.
7. fFrameSize = acquiredFrameSize;
8. fNumTruncatedBytes = fParser->numTruncatedBytes();
10. // "fPresentationTime" should have already been computed.
12. // Compute "fDurationInMicroseconds" now:
13. fDurationInMicroseconds =
14. (fFrameRate == 0.0 || ((**int**) fPictureCount) < 0) ?
15. 0 : (unsigned) ((fPictureCount \* 1000000) / fFrameRate);
16. fPictureCount = 0;
18. // Call our own 'after getting' function.  Because we're not a 'leaf'
19. // source, we can call this directly, without risking infinite recursion.
20. afterGetting(this);
21. } else {
22. // We were unable to parse a complete frame from the input, because:
23. // - we had to read more data from the source stream, or
24. // - the source stream has ended.
25. }
26. }

先利用Parser进行分析（应该是解析出一帧吧？），分析完成后，帧数据已到了MPEGVideoStreamFramer的缓冲fTo中。计算出帧的持续时间后，调用FrameSource的afterGetting()，最终会调用RTPSink的函数。  
看到这里，可以总结一下，其实看来看去，Parser直正被外部使用的函数几乎只有一个：parse()。  
  
下面可以看H264VideoStreamParser了。其实也很简单，多了一些用于分析h264格式的函数，当然是非公开的，只供自己使用的。在哪里使用呢？当然是在parser()中使用。至于H264VideoStreamFramer前面已经说过了，没什么太多的东西，所以就不看了。总结起来也就是这样：RTPSink向H264VideoStreamFramer要下一帧（其实h264中肯定不是一帧了，而是一个NAL Unit），H264VideoStreamFramer告诉H264VideoStreamParser输出缓冲和容内的字节数，然后调用H264VideoStreamParser的parser()函数，parser()中调用ByteStreamFileSource从文件中读取数据，直到parser()获得完整的一帧，parser()返回，H264VideoStreamFramer进行一些自己的处理后把这一帧返回给了RTPSink（当然是以回调函数的方式返回）。  
还有一个东西，H264FUAFragmenter，被H264VideoRTPSink所使用，继承自FramedFilter。它最初在RTPSink开始播放后创建，如下：

1. Boolean H264VideoRTPSink::continuePlaying()
2. {
3. // First, check whether we have a 'fragmenter' class set up yet.
4. // If not, create it now:
5. if (fOurFragmenter == NULL) {
6. fOurFragmenter = new H264FUAFragmenter(envir(), fSource,
7. OutPacketBuffer::maxSize,
8. ourMaxPacketSize() - 12/\*RTP hdr size\*/);
9. fSource = fOurFragmenter;
10. }
12. // Then call the parent class's implementation:
13. return MultiFramedRTPSink::continuePlaying();
14. }

并且它取代了H264VideoStreamFramer成为直接与RTPSink发生关系的source.如此一来，RTPSink要获取帧时，都是从它获取的．看它最主要的一个函数吧：

1. void H264FUAFragmenter::doGetNextFrame()
2. {
3. if (fNumValidDataBytes == 1) {
4. // We have no NAL unit data currently in the buffer.  Read a new one:
5. fInputSource->getNextFrame(&fInputBuffer[1], fInputBufferSize - 1,
6. afterGettingFrame, this, FramedSource::handleClosure, this);
7. } else {
8. // We have NAL unit data in the buffer.  There are three cases to consider:
9. // 1. There is a new NAL unit in the buffer, and it's small enough to deliver
10. //    to the RTP sink (as is).
11. // 2. There is a new NAL unit in the buffer, but it's too large to deliver to
12. //    the RTP sink in its entirety.  Deliver the first fragment of this data,
13. //    as a FU-A packet, with one extra preceding header byte.
14. // 3. There is a NAL unit in the buffer, and we've already delivered some
15. //    fragment(s) of this.  Deliver the next fragment of this data,
16. //    as a FU-A packet, with two extra preceding header bytes.
18. if (fMaxSize < fMaxOutputPacketSize) { // shouldn't happen
19. envir() << "H264FUAFragmenter::doGetNextFrame(): fMaxSize ("
20. << fMaxSize << ") is smaller than expected\n";
21. } else {
22. fMaxSize = fMaxOutputPacketSize;
23. }
25. fLastFragmentCompletedNALUnit = True; // by default
26. if (fCurDataOffset == 1) { // case 1 or 2
27. if (fNumValidDataBytes - 1 <= fMaxSize) { // case 1
28. memmove(fTo, &fInputBuffer[1], fNumValidDataBytes - 1);
29. fFrameSize = fNumValidDataBytes - 1;
30. fCurDataOffset = fNumValidDataBytes;
31. } else { // case 2
32. // We need to send the NAL unit data as FU-A packets.  Deliver the first
33. // packet now.  Note that we add FU indicator and FU header bytes to the front
34. // of the packet (reusing the existing NAL header byte for the FU header).
35. fInputBuffer[0] = (fInputBuffer[1] & 0xE0) | 28; // FU indicator
36. fInputBuffer[1] = 0x80 | (fInputBuffer[1] & 0x1F); // FU header (with S bit)
37. memmove(fTo, fInputBuffer, fMaxSize);
38. fFrameSize = fMaxSize;
39. fCurDataOffset += fMaxSize - 1;
40. fLastFragmentCompletedNALUnit = False;
41. }
42. } else { // case 3
43. // We are sending this NAL unit data as FU-A packets.  We've already sent the
44. // first packet (fragment).  Now, send the next fragment.  Note that we add
45. // FU indicator and FU header bytes to the front.  (We reuse these bytes that
46. // we already sent for the first fragment, but clear the S bit, and add the E
47. // bit if this is the last fragment.)
48. fInputBuffer[fCurDataOffset - 2] = fInputBuffer[0]; // FU indicator
49. fInputBuffer[fCurDataOffset - 1] = fInputBuffer[1] & ~0x80; // FU header (no S bit)
50. unsigned numBytesToSend = 2 + fNumValidDataBytes - fCurDataOffset;
51. if (numBytesToSend > fMaxSize) {
52. // We can't send all of the remaining data this time:
53. numBytesToSend = fMaxSize;
54. fLastFragmentCompletedNALUnit = False;
55. } else {
56. // This is the last fragment:
57. fInputBuffer[fCurDataOffset - 1] |= 0x40; // set the E bit in the FU header
58. fNumTruncatedBytes = fSaveNumTruncatedBytes;
59. }
60. memmove(fTo, &fInputBuffer[fCurDataOffset - 2], numBytesToSend);
61. fFrameSize = numBytesToSend;
62. fCurDataOffset += numBytesToSend - 2;
63. }
65. if (fCurDataOffset >= fNumValidDataBytes) {
66. // We're done with this data.  Reset the pointers for receiving new data:
67. fNumValidDataBytes = fCurDataOffset = 1;
68. }
70. // Complete delivery to the client:
71. FramedSource::afterGetting(this);
72. }
73. }

如果输入缓冲中没有数据，调用fInputSource->getNextFrame()，fInputSource是H264VideoStreamFramer，H264VideoStreamFramer的getNextFrame()会调用H264VideoStreamParser的parser()，parser()又调用ByteStreamFileSource获取数据，然后分析，parser()完成后会调用：

1. void H264FUAFragmenter::afterGettingFrame1(
2. unsigned frameSize,
3. unsigned numTruncatedBytes,
4. struct timeval presentationTime,
5. unsigned durationInMicroseconds)
6. {
7. fNumValidDataBytes += frameSize;
8. fSaveNumTruncatedBytes = numTruncatedBytes;
9. fPresentationTime = presentationTime;
10. fDurationInMicroseconds = durationInMicroseconds;
12. // Deliver data to the client:
13. doGetNextFrame();
14. }

然后又调用回H264FUAFragmenter::doGetNextFrame()，此时输入缓冲中有数据了，H264FUAFragmenter就进行分析处理．H264FUAFragmenter又对数据做了什么呢？

## 3

书接上回：H264FUAFragmenter又对数据做了什么呢？

1. void H264FUAFragmenter::doGetNextFrame()
2. {
3. if (fNumValidDataBytes == 1) {
4. // We have no NAL unit data currently in the buffer.  Read a new one:
5. fInputSource->getNextFrame(&fInputBuffer[1], fInputBufferSize - 1,
6. afterGettingFrame, this, FramedSource::handleClosure, this);
7. } else {
8. // We have NAL unit data in the buffer.  There are three cases to consider:
9. // 1. There is a new NAL unit in the buffer, and it's small enough to deliver
10. //    to the RTP sink (as is).
11. // 2. There is a new NAL unit in the buffer, but it's too large to deliver to
12. //    the RTP sink in its entirety.  Deliver the first fragment of this data,
13. //    as a FU-A packet, with one extra preceding header byte.
14. // 3. There is a NAL unit in the buffer, and we've already delivered some
15. //    fragment(s) of this.  Deliver the next fragment of this data,
16. //    as a FU-A packet, with two extra preceding header bytes.
18. if (fMaxSize < fMaxOutputPacketSize) { // shouldn't happen
19. envir() << "H264FUAFragmenter::doGetNextFrame(): fMaxSize ("
20. << fMaxSize << ") is smaller than expected\n";
21. } else {
22. fMaxSize = fMaxOutputPacketSize;
23. }
25. fLastFragmentCompletedNALUnit = True; // by default
26. if (fCurDataOffset == 1) { // case 1 or 2
27. if (fNumValidDataBytes - 1 <= fMaxSize) { // case 1
28. memmove(fTo, &fInputBuffer[1], fNumValidDataBytes - 1);
29. fFrameSize = fNumValidDataBytes - 1;
30. fCurDataOffset = fNumValidDataBytes;
31. } else { // case 2
32. // We need to send the NAL unit data as FU-A packets.  Deliver the first
33. // packet now.  Note that we add FU indicator and FU header bytes to the front
34. // of the packet (reusing the existing NAL header byte for the FU header).
35. fInputBuffer[0] = (fInputBuffer[1] & 0xE0) | 28; // FU indicator
36. fInputBuffer[1] = 0x80 | (fInputBuffer[1] & 0x1F); // FU header (with S bit)
37. memmove(fTo, fInputBuffer, fMaxSize);
38. fFrameSize = fMaxSize;
39. fCurDataOffset += fMaxSize - 1;
40. fLastFragmentCompletedNALUnit = False;
41. }
42. } else { // case 3
43. // We are sending this NAL unit data as FU-A packets.  We've already sent the
44. // first packet (fragment).  Now, send the next fragment.  Note that we add
45. // FU indicator and FU header bytes to the front.  (We reuse these bytes that
46. // we already sent for the first fragment, but clear the S bit, and add the E
47. // bit if this is the last fragment.)
48. fInputBuffer[fCurDataOffset - 2] = fInputBuffer[0]; // FU indicator
49. fInputBuffer[fCurDataOffset - 1] = fInputBuffer[1] & ~0x80; // FU header (no S bit)
50. unsigned numBytesToSend = 2 + fNumValidDataBytes - fCurDataOffset;
51. if (numBytesToSend > fMaxSize) {
52. // We can't send all of the remaining data this time:
53. numBytesToSend = fMaxSize;
54. fLastFragmentCompletedNALUnit = False;
55. } else {
56. // This is the last fragment:
57. fInputBuffer[fCurDataOffset - 1] |= 0x40; // set the E bit in the FU header
58. fNumTruncatedBytes = fSaveNumTruncatedBytes;
59. }
60. memmove(fTo, &fInputBuffer[fCurDataOffset - 2], numBytesToSend);
61. fFrameSize = numBytesToSend;
62. fCurDataOffset += numBytesToSend - 2;
63. }
65. if (fCurDataOffset >= fNumValidDataBytes) {
66. // We're done with this data.  Reset the pointers for receiving new data:
67. fNumValidDataBytes = fCurDataOffset = 1;
68. }
70. // Complete delivery to the client:
71. FramedSource::afterGetting(this);
72. }
73. }

当fNumValidDataBytes等于１时，表明buffer(fInputBuffer)中没有Nal Unit数据，那么就读入一个新的．从哪里读呢？还记得前面讲过的吗？H264FUAFragmenter在第一次读数据时代替了H264VideoStreamFramer，同时也与H264VideoStreamFramer还有ByteStreamFileSource手牵着脚，脚牵着手形成了链结构．文件数据从ByteStreamFileSource读入，经H264VideoStreamFramer处理传给H264FUAFragmenter．ByteStreamFileSource返回给H264VideoStreamFramer一段数据，H264VideoStreamFramer返回一个H264FUAFragmenter一个Nal unit ．  
H264FUAFragmenter对Nal Unit做了什么呢？先看注释：  
当我们有了nal unit，要处理３种情况：  
１有一个完整的nal unit，并且它小到能够被打包进rtp包中．  
２有一个完整的nal unit，但是它很大，那么就得为它分片传送了，把第一片打入一个FU-A包，此时利用了缓冲中前面的一个字节的头部．  
３一个nal unit的已被发送了一部分，那么我们继续按FU-A包发送．此时利用了缓冲中前面的处理中已使用的两个字节的头部．  
fNumValidDataBytes是H264FUAFragmenter缓冲fInputBuffer中有效数据的字节数．可以看到fNumValidDataBytes重置时被置为１，为什么不是０呢？因为fInputBuffer的第一个字节一直被留用作AU-A包的头部．如果是single nal打包，则从fInputBuffer的第二字节开始把nal unit复制到输出缓冲fTo，如果是FU-A包，则从fInputBuffer的第一字节开始复制．  
  
  
结合下文，可以很容易地把此段函数看明白（转自http://blog.csdn.net/perfectpdl/article/details/6633841）  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

H.264 视频 RTP 负载格式

1. 网络抽象层单元类型 (NALU)

NALU 头由一个字节组成, 它的语法如下:

      +---------------+  
      |0|1|2|3|4|5|6|7|  
      +-+-+-+-+-+-+-+-+  
      |F|NRI|  Type   |  
      +---------------+

F: 1 个比特.  
  forbidden\_zero\_bit. 在 H.264 规范中规定了这一位必须为 0.

NRI: 2 个比特.  
  nal\_ref\_idc. 取 00 ~ 11, 似乎指示这个 NALU 的重要性, 如 00 的 NALU 解码器可以丢弃它而不影响图像的回放. 不过一般情况下不太关心

这个属性.

Type: 5 个比特.  
  nal\_unit\_type. 这个 NALU 单元的类型. 简述如下:

  0     没有定义  
  1-23  NAL单元  单个 NAL 单元包.  
  24    STAP-A   单一时间的组合包  
  25    STAP-B   单一时间的组合包  
  26    MTAP16   多个时间的组合包  
  27    MTAP24   多个时间的组合包  
  28    FU-A     分片的单元  
  29    FU-B     分片的单元  
  30-31 没有定义

2. 打包模式

  下面是 RFC 3550 中规定的 RTP 头的结构.

       0                   1                   2                   3  
       0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1  
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
      |V=2|P|X|  CC   |M|     PT      |       sequence number         |  
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
      |                           timestamp                           |  
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
      |           synchronization source (SSRC) identifier            |  
      +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+  
      |            contributing source (CSRC) identifiers             |  
      |                             ....                              |  
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

  负载类型 Payload type (PT): 7 bits  
  序列号 Sequence number (SN): 16 bits  
  时间戳 Timestamp: 32 bits  
    
  H.264 Payload 格式定义了三种不同的基本的负载(Payload)结构. 接收端可能通过 RTP Payload   
  的第一个字节来识别它们. 这一个字节类似 NALU 头的格式, 而这个头结构的 NAL 单元类型字段  
  则指出了代表的是哪一种结构,

  这个字节的结构如下, 可以看出它和 H.264 的 NALU 头结构是一样的.  
      +---------------+  
      |0|1|2|3|4|5|6|7|  
      +-+-+-+-+-+-+-+-+  
      |F|NRI|  Type   |  
      +---------------+  
  字段 Type: 这个 RTP payload 中 NAL 单元的类型. 这个字段和 H.264 中类型字段的区别是, 当 type  
  的值为 24 ~ 31 表示这是一个特别格式的 NAL 单元, 而 H.264 中, 只取 1~23 是有效的值.  
     
  24    STAP-A   单一时间的组合包  
  25    STAP-B   单一时间的组合包  
  26    MTAP16   多个时间的组合包  
  27    MTAP24   多个时间的组合包  
  28    FU-A     分片的单元  
  29    FU-B     分片的单元  
  30-31 没有定义

  可能的结构类型分别有:

  1. 单一 NAL 单元模式  
     即一个 RTP 包仅由一个完整的 NALU 组成. 这种情况下 RTP NAL 头类型字段和原始的 H.264的  
  NALU 头类型字段是一样的.

  2. 组合封包模式  
    即可能是由多个 NAL 单元组成一个 RTP 包. 分别有4种组合方式: STAP-A, STAP-B, MTAP16, MTAP24.  
  那么这里的类型值分别是 24, 25, 26 以及 27.

  3. 分片封包模式  
    用于把一个 NALU 单元封装成多个 RTP 包. 存在两种类型 FU-A 和 FU-B. 类型值分别是 28 和 29.

2.1 单一 NAL 单元模式

  对于 NALU 的长度小于 MTU 大小的包, 一般采用单一 NAL 单元模式.  
  对于一个原始的 H.264 NALU 单元常由 [Start Code] [NALU Header] [NALU Payload] 三部分组成, 其中 Start Code 用于标示这是一个

NALU 单元的开始, 必须是 "00 00 00 01" 或 "00 00 01", NALU 头仅一个字节, 其后都是 NALU 单元内容.  
  打包时去除 "00 00 01" 或 "00 00 00 01" 的开始码, 把其他数据封包的 RTP 包即可.

       0                   1                   2                   3  
       0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1  
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
      |F|NRI|  type   |                                               |  
      +-+-+-+-+-+-+-+-+                                               |  
      |                                                               |  
      |               Bytes 2..n of a Single NAL unit                 |  
      |                                                               |  
      |                               +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
      |                               :...OPTIONAL RTP padding        |  
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

  如有一个 H.264 的 NALU 是这样的:

  [00 00 00 01 67 42 A0 1E 23 56 0E 2F ... ]

  这是一个序列参数集 NAL 单元. [00 00 00 01] 是四个字节的开始码, 67 是 NALU 头, 42 开始的数据是 NALU 内容.

  封装成 RTP 包将如下:

  [ RTP Header ] [ 67 42 A0 1E 23 56 0E 2F ]

  即只要去掉 4 个字节的开始码就可以了.

2.2 组合封包模式

  其次, 当 NALU 的长度特别小时, 可以把几个 NALU 单元封在一个 RTP 包中.

       0                   1                   2                   3  
       0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1  
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
      |                          RTP Header                           |  
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
      |STAP-A NAL HDR |         NALU 1 Size           | NALU 1 HDR    |  
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
      |                         NALU 1 Data                           |  
      :                                                               :  
      +               +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
      |               | NALU 2 Size                   | NALU 2 HDR    |  
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
      |                         NALU 2 Data                           |  
      :                                                               :  
      |                               +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
      |                               :...OPTIONAL RTP padding        |  
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

2.3 Fragmentation Units (FUs).

  而当 NALU 的长度超过 MTU 时, 就必须对 NALU 单元进行分片封包. 也称为 Fragmentation Units (FUs).  
    
       0                   1                   2                   3  
       0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1  
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
      | FU indicator  |   FU header   |                               |  
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+                               |  
      |                                                               |  
      |                         FU payload                            |  
      |                                                               |  
      |                               +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  
      |                               :...OPTIONAL RTP padding        |  
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

      Figure 14.  RTP payload format for FU-A

   The FU indicator octet has the following format:

      +---------------+  
      |0|1|2|3|4|5|6|7|  
      +-+-+-+-+-+-+-+-+  
      |F|NRI|  Type   |  
      +---------------+

   The FU header has the following format:

      +---------------+  
      |0|1|2|3|4|5|6|7|  
      +-+-+-+-+-+-+-+-+  
      |S|E|R|  Type   |  
      +---------------+

－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

H264FUAFragmenter中只支持single和FU-A模式，不支持其它模式．

我们现在还得出一个结论，我们可以看出RTPSink与Source怎样分工：RTPSink只做形成通用RTP包头的工作，各种媒体格式的Source才是实现媒体数据RTP封包的地方，其实按习惯感觉XXXRTPSink才是进行封包的地方．但是，从文件的安排上，H264FUAFragmenter被隐藏在H264VideoRTPSink中，并在程序中暗渡陈仓地把H264VideoStreamFramer替换掉，其实还是按习惯的架构（设计模式）来做的，所以如果把H264FUAFragmenter的工作移到H264VideoRTPSink中也是没问题的．

# 十二 h264 rtp包的时间戳

这次我们一起来分析一下live555中是怎样为rtp包打时间戳的．就以h264为例吧．

1. void H264VideoRTPSink::doSpecialFrameHandling(unsigned /\*fragmentationOffset\*/,
2. unsigned **char**\* /\*frameStart\*/,
3. unsigned /\*numBytesInFrame\*/,
4. struct timeval framePresentationTime,
5. unsigned /\*numRemainingBytes\*/)
6. {
7. // Set the RTP 'M' (marker) bit iff
8. // 1/ The most recently delivered fragment was the end of (or the only fragment of) an NAL unit, and
9. // 2/ This NAL unit was the last NAL unit of an 'access unit' (i.e. video frame).
10. if (fOurFragmenter != NULL) {
11. H264VideoStreamFramer\* framerSource = (H264VideoStreamFramer\*) (fOurFragmenter->inputSource());
12. // This relies on our fragmenter's source being a "H264VideoStreamFramer".
13. if (fOurFragmenter->lastFragmentCompletedNALUnit()
14. && framerSource != NULL && framerSource->pictureEndMarker()) {
15. setMarkerBit();
16. framerSource->pictureEndMarker() = False;
17. }
18. }
20. setTimestamp(framePresentationTime);
21. }

函数中先检测是否是一个帧的最后一个包，如果是，打上'M'标记．然后就设置时间戳．这个间戳是哪来的呢？需看函数doSpecialFrameHandling()是被谁调用的，经查找，是被MultiFramedRTPSink::afterGettingFrame1()调用的．MultiFramedRTPSink::afterGettingFrame1()的参数presentationTime传给了doSpecialFrameHandling()．MultiFramedRTPSink::afterGettingFrame1()是在调用source的getNextFrame()时传给了source．传给哪个source呢？传给了H264FUAFragmenter,还记得暗渡陈仓那件事吗？所以H264FUAFragmenter在获取一个nal unit后调用了MultiFramedRTPSink::afterGettingFrame1()．也就是H264FUAFragmenter::afterGettingFrame1()调用了MultiFramedRTPSink::afterGettingFrame1()．

H264FUAFragmenter::afterGettingFrame1()是被它自己的source的afterGettingFrame1()调用的．H264FUAFragmenter的source是谁呢？是H264VideoStreamFramer，是在暗渡陈仓时传给H264FUAFragmenter的构造函数的．

H264VideoStreamFramer的afterGettingFrame1()是没有的，代替之的是MPEGVideoStreamFramer::continueReadProcessin()．它被MPEGVideoStreamParser暗中传给了StreamParser的构造函数．所以StreamParser在分析完一帧（或nal unit)之后，调用的就是MPEGVideoStreamFramer::continueReadProcessin()．以下即是证明：**(补充：以下函数并不是在parser分析完一帧（或nal unit）之后调用，而是parser利用ByteStreamFileSuorce获取到原始数据后调用，然后MPEGVideoStreamFramer再调用Parser的parser()函数分析原始数据)**

1. void StreamParser::afterGettingBytes(void\* clientData,
2. unsigned numBytesRead,
3. unsigned /\*numTruncatedBytes\*/,
4. struct timeval presentationTime,
5. unsigned /\*durationInMicroseconds\*/)
6. {
7. StreamParser\* parser = (StreamParser\*) clientData;
8. if (parser != NULL)
9. parser->afterGettingBytes1(numBytesRead, presentationTime);
10. }
12. void StreamParser::afterGettingBytes1(unsigned numBytesRead,
13. struct timeval presentationTime)
14. {
15. // Sanity check: Make sure we didn't get too many bytes for our bank:
16. if (fTotNumValidBytes + numBytesRead > BANK\_SIZE) {
17. fInputSource->envir()
18. << "StreamParser::afterGettingBytes() warning: read "
19. << numBytesRead << " bytes; expected no more than "
20. << BANK\_SIZE - fTotNumValidBytes << "\n";
21. }
23. fLastSeenPresentationTime = presentationTime;
25. unsigned **char**\* ptr = &curBank()[fTotNumValidBytes];
26. fTotNumValidBytes += numBytesRead;
28. // Continue our original calling source where it left off:
29. restoreSavedParserState();
30. // Sigh... this is a crock; things would have been a lot simpler
31. // here if we were using threads, with synchronous I/O...
32. fClientContinueFunc(fClientContinueClientData, ptr, numBytesRead,
33. presentationTime);
34. }

fClientContinueFunc就是MPEGVideoStreamFramer::continueReadProcessin()，而且我们看到时间戳被传入fClientContinueFunc．  
然而，MPEGVideoStreamFramer::continueReadProcessin()中跟本就不理这个时间戳，因为这个时间戳是ByteStreamFileSource计算出来的，它跟本就不可能正确．

1. void MPEGVideoStreamFramer::continueReadProcessing(void\* clientData,
2. unsigned **char**\* /\*ptr\*/,
3. unsigned /\*size\*/,
4. struct timeval /\*presentationTime\*/)
5. {
6. MPEGVideoStreamFramer\* framer = (MPEGVideoStreamFramer\*) clientData;
7. framer->continueReadProcessing();
8. }

看来真正的时间戳是在MPEGVideoStreamFramer中计算的，但是H264VideoStreamFramer并没有用到MPEGVideoStreamFramer中那些计算时间戳的函数，而是另外计算，其实H264VideoStreamFramer也没有自己去计算，而是利用H264VideoStreamParser计算的．是在哪个函数中呢？在parser()中！

1. unsigned H264VideoStreamParser::parse()
2. {
3. try {
4. // The stream must start with a 0x00000001:
5. if (!fHaveSeenFirstStartCode) {
6. // Skip over any input bytes that precede the first 0x00000001:
7. u\_int32\_t first4Bytes;
8. while ((first4Bytes = test4Bytes()) != 0x00000001) {
9. get1Byte();
10. setParseState(); // ensures that we progress over bad data
11. }
12. skipBytes(4); // skip this initial code
14. setParseState();
15. fHaveSeenFirstStartCode = True; // from now on
16. }
18. if (fOutputStartCodeSize > 0) {
19. // Include a start code in the output:
20. save4Bytes(0x00000001);
21. }
23. // Then save everything up until the next 0x00000001 (4 bytes) or 0x000001 (3 bytes), or we hit EOF.
24. // Also make note of the first byte, because it contains the "nal\_unit\_type":
25. u\_int8\_t firstByte;
26. if (haveSeenEOF()) {
27. // We hit EOF the last time that we tried to parse this data,
28. // so we know that the remaining unparsed data forms a complete NAL unit:
29. unsigned remainingDataSize = totNumValidBytes() - curOffset();
30. if (remainingDataSize == 0)
31. (void) get1Byte(); // forces another read, which will cause EOF to get handled for real this time
32. if (remainingDataSize == 0)
33. return 0;
34. firstByte = get1Byte();
35. saveByte(firstByte);
37. while (--remainingDataSize > 0) {
38. saveByte(get1Byte());
39. }
40. } else {
41. u\_int32\_t next4Bytes = test4Bytes();
42. firstByte = next4Bytes >> 24;
43. while (next4Bytes != 0x00000001
44. && (next4Bytes & 0xFFFFFF00) != 0x00000100) {
45. // We save at least some of "next4Bytes".
46. if ((unsigned) (next4Bytes & 0xFF) > 1) {
47. // Common case: 0x00000001 or 0x000001 definitely doesn't begin anywhere in "next4Bytes", so we save all of it:
48. save4Bytes(next4Bytes);
49. skipBytes(4);
50. } else {
51. // Save the first byte, and continue testing the rest:
52. saveByte(next4Bytes >> 24);
53. skipBytes(1);
54. }
55. next4Bytes = test4Bytes();
56. }
57. // Assert: next4Bytes starts with 0x00000001 or 0x000001, and we've saved all previous bytes (forming a complete NAL unit).
58. // Skip over these remaining bytes, up until the start of the next NAL unit:
59. if (next4Bytes == 0x00000001) {
60. skipBytes(4);
61. } else {
62. skipBytes(3);
63. }
64. }
66. u\_int8\_t nal\_ref\_idc = (firstByte & 0x60) >> 5;
67. u\_int8\_t nal\_unit\_type = firstByte & 0x1F;
69. switch (nal\_unit\_type) {
70. case 6: { // Supplemental enhancement information (SEI)
71. analyze\_sei\_data();
72. // Later, perhaps adjust "fPresentationTime" if we saw a "pic\_timing" SEI payload??? #####
73. break;
74. }
75. case 7: { // Sequence parameter set
76. // First, save a copy of this NAL unit, in case the downstream object wants to see it:
77. usingSource()->saveCopyOfSPS(fStartOfFrame + fOutputStartCodeSize,
78. fTo - fStartOfFrame - fOutputStartCodeSize);
80. // Parse this NAL unit to check whether frame rate information is present:
81. unsigned num\_units\_in\_tick, time\_scale, fixed\_frame\_rate\_flag;
82. analyze\_seq\_parameter\_set\_data(num\_units\_in\_tick, time\_scale,
83. fixed\_frame\_rate\_flag);
84. if (time\_scale > 0 && num\_units\_in\_tick > 0) {
85. usingSource()->fFrameRate = time\_scale
86. / (2.0 \* num\_units\_in\_tick);
87. } else {
88. }
89. break;
90. }
91. case 8: { // Picture parameter set
92. // Save a copy of this NAL unit, in case the downstream object wants to see it:
93. usingSource()->saveCopyOfPPS(fStartOfFrame + fOutputStartCodeSize,
94. fTo - fStartOfFrame - fOutputStartCodeSize);
95. }
96. }
98. //更新时间戳变量
99. usingSource()->setPresentationTime();
101. // If this NAL unit is a VCL NAL unit, we also scan the start of the next NAL unit, to determine whether this NAL unit
102. // ends the current 'access unit'.  We need this information to figure out when to increment "fPresentationTime".
103. // (RTP streamers also need to know this in order to figure out whether or not to set the "M" bit.)
104. Boolean thisNALUnitEndsAccessUnit = False; // until we learn otherwise
105. if (haveSeenEOF()) {
106. // There is no next NAL unit, so we assume that this one ends the current 'access unit':
107. thisNALUnitEndsAccessUnit = True;
108. } else {
109. Boolean const isVCL = nal\_unit\_type <= 5 && nal\_unit\_type > 0; // Would need to include type 20 for SVC and MVC #####
110. if (isVCL) {
111. u\_int32\_t first4BytesOfNextNALUnit = test4Bytes();
112. u\_int8\_t firstByteOfNextNALUnit = first4BytesOfNextNALUnit
113. >> 24;
114. u\_int8\_t next\_nal\_ref\_idc = (firstByteOfNextNALUnit & 0x60)
115. >> 5;
116. u\_int8\_t next\_nal\_unit\_type = firstByteOfNextNALUnit & 0x1F;
117. if (next\_nal\_unit\_type >= 6) {
118. // The next NAL unit is not a VCL; therefore, we assume that this NAL unit ends the current 'access unit':
119. thisNALUnitEndsAccessUnit = True;
120. } else {
121. // The next NAL unit is also a VLC.  We need to examine it a little to figure out if it's a different 'access unit'.
122. // (We use many of the criteria described in section 7.4.1.2.4 of the H.264 specification.)
123. Boolean IdrPicFlag = nal\_unit\_type == 5;
124. Boolean next\_IdrPicFlag = next\_nal\_unit\_type == 5;
125. if (next\_IdrPicFlag != IdrPicFlag) {
126. // IdrPicFlag differs in value
127. thisNALUnitEndsAccessUnit = True;
128. } else if (next\_nal\_ref\_idc != nal\_ref\_idc
129. && next\_nal\_ref\_idc \* nal\_ref\_idc == 0) {
130. // nal\_ref\_idc differs in value with one of the nal\_ref\_idc values being equal to 0
131. thisNALUnitEndsAccessUnit = True;
132. } else if ((nal\_unit\_type == 1 || nal\_unit\_type == 2
133. || nal\_unit\_type == 5)
134. && (next\_nal\_unit\_type == 1
135. || next\_nal\_unit\_type == 2
136. || next\_nal\_unit\_type == 5)) {
137. // Both this and the next NAL units begin with a "slice\_header".
138. // Parse this (for each), to get parameters that we can compare:
140. // Current NAL unit's "slice\_header":
141. unsigned frame\_num, pic\_parameter\_set\_id, idr\_pic\_id;
142. Boolean field\_pic\_flag, bottom\_field\_flag;
143. analyze\_slice\_header(
144. fStartOfFrame + fOutputStartCodeSize, fTo,
145. nal\_unit\_type, frame\_num, pic\_parameter\_set\_id,
146. idr\_pic\_id, field\_pic\_flag, bottom\_field\_flag);
148. // Next NAL unit's "slice\_header":
149. u\_int8\_t next\_slice\_header[NUM\_NEXT\_SLICE\_HEADER\_BYTES\_TO\_ANALYZE];
150. testBytes(next\_slice\_header, sizeof next\_slice\_header);
151. unsigned next\_frame\_num, next\_pic\_parameter\_set\_id,
152. next\_idr\_pic\_id;
153. Boolean next\_field\_pic\_flag, next\_bottom\_field\_flag;
154. analyze\_slice\_header(next\_slice\_header,
155. &next\_slice\_header[sizeof next\_slice\_header],
156. next\_nal\_unit\_type, next\_frame\_num,
157. next\_pic\_parameter\_set\_id, next\_idr\_pic\_id,
158. next\_field\_pic\_flag, next\_bottom\_field\_flag);
160. if (next\_frame\_num != frame\_num) {
161. // frame\_num differs in value
162. thisNALUnitEndsAccessUnit = True;
163. } else if (next\_pic\_parameter\_set\_id
164. != pic\_parameter\_set\_id) {
165. // pic\_parameter\_set\_id differs in value
166. thisNALUnitEndsAccessUnit = True;
167. } else if (next\_field\_pic\_flag != field\_pic\_flag) {
168. // field\_pic\_flag differs in value
169. thisNALUnitEndsAccessUnit = True;
170. } else if (next\_bottom\_field\_flag
171. != bottom\_field\_flag) {
172. // bottom\_field\_flag differs in value
173. thisNALUnitEndsAccessUnit = True;
174. } else if (next\_IdrPicFlag == 1
175. && next\_idr\_pic\_id != idr\_pic\_id) {
176. // IdrPicFlag is equal to 1 for both and idr\_pic\_id differs in value
177. // Note: We already know that IdrPicFlag is the same for both.
178. thisNALUnitEndsAccessUnit = True;
179. }
180. }
181. }
182. }
183. }
185. //注意！注意！注意！此处计算时间戳！！
186. if (thisNALUnitEndsAccessUnit) {
187. usingSource()->fPictureEndMarker = True;
188. ++usingSource()->fPictureCount;
190. // Note that the presentation time for the next NAL unit will be different:
191. struct timeval& nextPT = usingSource()->fNextPresentationTime; // alias
192. nextPT = usingSource()->fPresentationTime;
193. **double** nextFraction = nextPT.tv\_usec / 1000000.0
194. + 1 / usingSource()->fFrameRate;
195. unsigned nextSecsIncrement = (**long**) nextFraction;
196. nextPT.tv\_sec += (**long**) nextSecsIncrement;
197. nextPT.tv\_usec = (**long**) ((nextFraction - nextSecsIncrement)
198. \* 1000000);
199. }
200. setParseState();
202. return curFrameSize();
203. } catch (**int** /\*e\*/) {
204. return 0; // the parsing got interrupted
205. }
206. }

每当开始一个新帧时，计算新的时间戳．时间戳保存在fNextPresentationTime中，在usingSource()->setPresentationTime()中传给fPresentationTime.  
哇，我们看到live555的类之间调用关系曲折复杂，的确有点不易维护啊！同时我写的也不够清析，自己看着都晕，如果把你搞晕了，这很正常哦！

fPresentationTime是64位的时间，经convertToRTPTimestamp转换为32的rtp时间戳，见函数：

1. u\_int32\_t RTPSink::convertToRTPTimestamp(struct timeval tv)
2. {
3. // Begin by converting from "struct timeval" units to RTP timestamp units:
4. u\_int32\_t timestampIncrement = (fTimestampFrequency \* tv.tv\_sec);
5. timestampIncrement += (u\_int32\_t)(
6. (2.0 \* fTimestampFrequency \* tv.tv\_usec + 1000000.0) / 2000000);
7. // note: rounding
9. // Then add this to our 'timestamp base':
10. if (fNextTimestampHasBeenPreset) {
11. // Make the returned timestamp the same as the current "fTimestampBase",
12. // so that timestamps begin with the value that was previously preset:
13. fTimestampBase -= timestampIncrement;
14. fNextTimestampHasBeenPreset = False;
15. }
17. u\_int32\_t const rtpTimestamp = fTimestampBase + timestampIncrement;
19. return rtpTimestamp;
20. }

其实时间戳的转换主要就是把以秒为单位的时间，提升成按频率为单位的时间．也就是提升后，时间间隔不是以秒为单位，而是以1/fTimestampFrequency为单位，也就是1/9000秒。然后再强转为32。

# 十三：RTPInterface详解

大家应该已理解了GroupSocket这个类。理论上讲那些需要操作udp socket 的类应保存GroupSocket的实例。但事实并不是这样，可以看一下RTPSink，RTPSource，RTCPInstance等，它们都没有保存GroupSocket型的变量。那它们通过哪个类进行socket操作呢？是RTPInterface！！  
这些类接收的GroupSocket指针最后都传给了 RTPInterface 。为什么用RTPInterface而不直接用GroupSocket呢？这里面有个故事...扯远了。

要解答这个问题，让我们先提出问题吧。  
首先请问，Live555即支持rtp over udp，又支持rtp over tcp。那么在rtp over tcp情况下，用 GroupSocket 怎么实现呢？GroupSocket可是仅仅代表UDP啊！  
那么RTPInterface既然用于网络读写，它就应该既支持tcp收发，也支持udp收发。而且它还要像GroupSocket那样支持一对多。因为服务端是一对多个客户端哦。我们看一下RTPInterface的成员：  
Groupsock\* fGS;  
tcpStreamRecord\* fTCPStreams; // optional, for RTP-over-TCP streaming/receiving  
嘿嘿，这两个紧靠着，说明它们关系不一般啊（难道他们有一腿？）。fGS－－代表了一个udp socket和它对应的多个目的端，fTCPStreams--代表了多个TCP socket，当然这些socket都是从一个socket accept()出来的客户端socket（tcpStreamRecord是一个链表哦）。  
看到这个架式，我想大家都要得出结论了：RTPInterface还真是男女通吃啊！不论你客户端与我建立的是tcp连接，还是udp连接，我RTPInterface一律能接收你们的数据，并向你们发出数据！

证据一：向所有客户端发出数据：

1. Boolean RTPInterface::sendPacket(unsigned **char**\* packet, unsigned packetSize)
2. {
3. Boolean success = True; // we'll return False instead if any of the sends fail
5. // Normal case: Send as a UDP packet:
6. if (!fGS->output(envir(), fGS->ttl(), packet, packetSize))
7. success = False;
9. // Also, send over each of our TCP sockets:
10. for (tcpStreamRecord\* streams = fTCPStreams; streams != NULL;
11. streams = streams->fNext) {
12. if (!sendRTPOverTCP(packet, packetSize, streams->fStreamSocketNum,
13. streams->fStreamChannelId)) {
14. success = False;
15. }
16. }
18. return success;
19. }

很明显啊，先发送udp数据，一对多的问题在GroupSocket中解决。再发送tcp数据，一对多的问题本地解决。  
证据二：从所有客户端读取数据：  
我现在找不到直接的证据，所以我就憶想一下吧：当udp端口或tcp端口收到数据时，分析后，是哪个客户端的数据就发给对应这个客户端的RTPSink或RTCPInstance。  
好像已经把最开始的问题解答完了。下面让我们来分析一下RTPInterface吧。

1. void RTPInterface::setStreamSocket(**int** sockNum, unsigned **char** streamChannelId)
2. {
3. fGS->removeAllDestinations();
4. addStreamSocket(sockNum, streamChannelId);
5. }
7. void RTPInterface::addStreamSocket(**int** sockNum, unsigned **char** streamChannelId)
8. {
9. if (sockNum < 0)
10. return;
12. for (tcpStreamRecord\* streams = fTCPStreams; streams != NULL;
13. streams = streams->fNext) {
14. if (streams->fStreamSocketNum == sockNum
15. && streams->fStreamChannelId == streamChannelId) {
16. return; // we already have it
17. }
18. }
20. fTCPStreams = new tcpStreamRecord(sockNum, streamChannelId, fTCPStreams);
21. }

setStreamSocket()没必要说了吧，看一下addStreamSocke()。从字面意思应能了解，添加一个流式Socket，也就是添加tcp   
socket了。循环中查找是否已经存在了，最后如果不存在，就创建之，在tcpStreamRecord的构造函数中己经把自己加入了链表。对于参数，sockNum很易理解，就是socket()返回的那个SOCKET型  
数据呗，streamChannelId是什么呢？我们不防再猜测一下（很奇怪，我每次都能猜对，嘿嘿...）：rtp over tcp时，这个tcp连接是直接利用了RTSP所用的那个tcp连接，如果同时有很多rtp   
session，再加上rtsp session，大家都用这一个socket通信，怎么区分你的还是我的?我想这个channel   
id就是用于解决这个问题。给每个session分配一个唯一的id，在发送自己的包时为包再加上个头部，头部中需要有session的标记－－也就是这个channel id，包的长度等等字段。这样大家就可以穿一条裤子了，术语叫多路复用，但要注意只有tcp才进行多路复用，udp是不用的，因为udp是一个session对应一个socket（加上RTCP是两个）。  
想像一下，服务端要从这个tcp socket读写数据，必须把一个handler加入TaskScheduler中，这个handler在可读数据时进行读，在可写数据时进行写。在读数据时，对读出的数据进行分析，取得数据包的长度，以及其channel id，跟据channel id找到相应的处handler和对象，交给它们去处理自己的数据。  
试想两个建立在tcp上的rtp session，这个两个tcp socket既担负着rtsp通讯，又担负着rtp通讯。如果这两个rtp session共用一个stream，那么最终负责这两个session通信的就只有一个RTPInterface，那么这个RTPInterface中的fTCPStreams这个链表中就会有两项，分别对应这两个session。tcpStreamRecord主要用于socket number与channel id的对应。这些tcpStreamRecord是通过addStreamSocket()添加的。处理数据的handler是通过startNetworkReading（）添加的，看一下下：

1. void RTPInterface::startNetworkReading(TaskScheduler::BackgroundHandlerProc\* handlerProc)
2. {
3. // Normal case: Arrange to read UDP packets:
4. envir().taskScheduler().turnOnBackgroundReadHandling(fGS->socketNum(),handlerProc,
5. fOwner);
7. // Also, receive RTP over TCP, on each of our TCP connections:
8. fReadHandlerProc = handlerProc;
9. for (tcpStreamRecord\* streams = fTCPStreams; streams != NULL;
10. streams = streams->fNext) {
11. // Get a socket descriptor for "streams->fStreamSocketNum":
12. SocketDescriptor\* socketDescriptor = lookupSocketDescriptor(envir(),
13. streams->fStreamSocketNum);
15. // Tell it about our subChannel:
16. socketDescriptor->registerRTPInterface(streams->fStreamChannelId, this);
17. }
18. }

用UDP时很简单，直接把处理函数做为handler加入taskScheduler即可。而TCP时，需向所有的session的socket都注册自己。可以想像，socketDescriptor代表一个tcp socket，并且它有一个链表之类的东西，其中保存了所有的对这个socket感兴趣的RTPInterface，同时也记录了RTPInterface对应的channal id。只有向socketDescriptor注册了自己，socketDescriptor在读取数据时，才能跟据分析出的channel id找到对应的RTPInterface，才能调用RTPInterface中的数据处理handler，当然，这个函数也不是RTPInteface自己的，而是从startNetworkReading()这个函数接收到的调用者的。  
上述主要讲的是一个RTPInterface对应多个客户端tcp socket的情形。现在又发现一个问题：SocketDescriptor为什么需要对应多个RTPInterface呢？上面已经讲了，是为了多路复用，因为这个socket即负担rtsp通信又负担rtp通信还负担RTCP通信。SocketDescriptor记录多路复用数据（也就是RTPInterface与channel id）用了一个Hash table：HashTable\* fSubChannelHashTable。SocketDescriptor读数据使用函数：static void tcpReadHandler(SocketDescriptor\*, int mask)。证据如下：

1. void SocketDescriptor::registerRTPInterface(
2. unsigned **char** streamChannelId,
3. RTPInterface\* rtpInterface)
4. {
5. Boolean isFirstRegistration = fSubChannelHashTable->IsEmpty();
6. fSubChannelHashTable->Add((**char** const\*) (**long**) streamChannelId,
7. rtpInterface);
9. if (isFirstRegistration) {
10. // Arrange to handle reads on this TCP socket:
11. TaskScheduler::BackgroundHandlerProc\* handler =
12. (TaskScheduler::BackgroundHandlerProc\*) &tcpReadHandler;
13. fEnv.taskScheduler().turnOnBackgroundReadHandling(fOurSocketNum,
14. handler, this);
15. }
16. }

可见在注册第一个多路复用对象时启动reand handler。看一下函数主体：

1. void SocketDescriptor::tcpReadHandler1(**int** mask)
2. {
3. // We expect the following data over the TCP channel:
4. //   optional RTSP command or response bytes (before the first '$' character)
5. //   a '$' character
6. //   a 1-byte channel id
7. //   a 2-byte packet size (in network byte order)
8. //   the packet data.
9. // However, because the socket is being read asynchronously, this data might arrive in pieces.
11. u\_int8\_t c;
12. struct sockaddr\_in fromAddress;
13. if (fTCPReadingState != AWAITING\_PACKET\_DATA) {
14. **int** result = readSocket(fEnv, fOurSocketNum, &c, 1, fromAddress);
15. if (result != 1) { // error reading TCP socket, or no more data available
16. if (result < 0) { // error
17. fEnv.taskScheduler().turnOffBackgroundReadHandling(
18. fOurSocketNum); // stops further calls to us
19. }
20. return;
21. }
22. }
24. switch (fTCPReadingState) {
25. case AWAITING\_DOLLAR: {
26. if (c == '$') {
27. fTCPReadingState = AWAITING\_STREAM\_CHANNEL\_ID;
28. } else {
29. // This character is part of a RTSP request or command, which is handled separately:
30. if (fServerRequestAlternativeByteHandler != NULL) {
31. (\*fServerRequestAlternativeByteHandler)(
32. fServerRequestAlternativeByteHandlerClientData, c);
33. }
34. }
35. break;
36. }
37. case AWAITING\_STREAM\_CHANNEL\_ID: {
38. // The byte that we read is the stream channel id.
39. if (lookupRTPInterface(c) != NULL) { // sanity check
40. fStreamChannelId = c;
41. fTCPReadingState = AWAITING\_SIZE1;
42. } else {
43. // This wasn't a stream channel id that we expected.  We're (somehow) in a strange state.  Try to recover:
44. fTCPReadingState = AWAITING\_DOLLAR;
45. }
46. break;
47. }
48. case AWAITING\_SIZE1: {
49. // The byte that we read is the first (high) byte of the 16-bit RTP or RTCP packet 'size'.
50. fSizeByte1 = c;
51. fTCPReadingState = AWAITING\_SIZE2;
52. break;
53. }
54. case AWAITING\_SIZE2: {
55. // The byte that we read is the second (low) byte of the 16-bit RTP or RTCP packet 'size'.
56. unsigned **short** size = (fSizeByte1 << 8) | c;
58. // Record the information about the packet data that will be read next:
59. RTPInterface\* rtpInterface = lookupRTPInterface(fStreamChannelId);
60. if (rtpInterface != NULL) {
61. rtpInterface->fNextTCPReadSize = size;
62. rtpInterface->fNextTCPReadStreamSocketNum = fOurSocketNum;
63. rtpInterface->fNextTCPReadStreamChannelId = fStreamChannelId;
64. }
65. fTCPReadingState = AWAITING\_PACKET\_DATA;
66. break;
67. }
68. case AWAITING\_PACKET\_DATA: {
69. // Call the appropriate read handler to get the packet data from the TCP stream:
70. RTPInterface\* rtpInterface = lookupRTPInterface(fStreamChannelId);
71. if (rtpInterface != NULL) {
72. if (rtpInterface->fNextTCPReadSize == 0) {
73. // We've already read all the data for this packet.
74. fTCPReadingState = AWAITING\_DOLLAR;
75. break;
76. }
77. if (rtpInterface->fReadHandlerProc != NULL) {
78. rtpInterface->fReadHandlerProc(rtpInterface->fOwner, mask);
79. }
80. }
81. return;
82. }
83. }
84. }

最开始的注释中解释了多路复用头的格式。这一段引起了我的兴趣：

1. case AWAITING\_DOLLAR: {
2. if (c == $) {
3. fTCPReadingState = AWAITING\_STREAM\_CHANNEL\_ID;
4. } else {
5. // This character is part of a RTSP request or command, which is handled separately:
6. if (fServerRequestAlternativeByteHandler != NULL) {
7. (\*fServerRequestAlternativeByteHandler)(
8. fServerRequestAlternativeByteHandlerClientData, c);
9. }
10. }
11. break;
12. }

啊！原来ServerRequestAlternativeByteHandler是用于处理RTSP数据的。也就是从这个socket收到RTSP数据时，调用ServerRequestAlternativeByteHandler。如果收到RTP/RTCP数据时，先查看其channel id，跟据id找到RTPInterface(RTCP也是用了RTPIterface进行通信)，设置RTPInterface中与读缓冲有关的变量，然后当读到包数据的开始位置时，调用rtpInterface中保存的数据处理handler。还记得吧，rtpInterface中的这个数据处理handler在UDP时也被使用，在这个函数中要做的是读取一个包的数据，然后处理这个包。而SocketDescriptor把读取位置置于包数据开始的位置再交给数据处理handler，正好可以使用与UDP相同的数据处理handler！  
还有，socketDescriptor们并不属于任何RTPInterface，而是单独保存在一个Hash table中，这样多个RTPInterface都可以注册到一个socketDescriptor中，以实现多路复用。  
总结一下通过RTPInterface，live555不仅实现了rtp over udp，还实现了rtp over tcp，而且还实现了同时即有rtp over tcp，又有rtp over udp！  
最后，channel id是从哪里来的呢？是在RTSP请求中指定的。在哪个请求中呢？自己找去吧。

# 十四: live555多线程论

江湖传闻:live555如果不改为多线程,在多核心机器上效率会降低.  
虽然我没做过测试,但比较相信此传闻的真实性

所以在我试论述一下live555如何对多核进行支持,其实就是改为多线程,嘿嘿.

先看此文:http://www.live555.com/liveMedia/faq.html#threads

跟据它的说法,live555改多线程似乎不难,因为所有全局性的东西几乎都保存在UsageEnvironment的liveMediaPriv和groupsockPriv中,groupsockPriv里面放所有的GroupSock,而liveMediaPriv指向了一个HashTab类:\_Tables,\_Tables中有两个变量:mediaTable和socketTable.分别指向两个Hash Tab,mediaTable中存放所有从Meduim派生出来的类对象,socketTable存放的是StreamSocket们(我猜的,嘿嘿),比如SocketDescriptor.总之,全局性的东西们都放在UsageEnvironment内.

所以,如果开了多线程,为每个线程创建一个UsageEnvironment,然后调用各自UsageEnvironment的TaskSchedule的EventLoop(),理论上应该能实现各线程各自为战,互不干扰.

但是RTSPServer却只能有一个,所以,各线程之间还必须有少量的交集.而且,RTSPServer最好单独放在一个线程中吧?因它总揽全局,所以正好放在主线程中.当然主线程也要有自己的UsageEnvironment和event loop.

如果真的实现了多线程,我们完全可以跟据CPU的数量确定线程的个数.那在什么时机,如何创建新线程呢?

想一下各RtspClientSession 的创建过程:RTSPServer收到新客户端请求后,先创建与客户对应的RTSPClientSession,RTSPClientSession在收到DESCRIBE请求后查找对应的ServerMediaSession,如果找不到,就创建一个新的,那么这几个对象的创建过程,从哪个开始进入新线程呢?

其实从RTSPClientSession开始,就可以放入新线程中,无非是RTSPServer要操作RTSPClientSession时进行同步保护而已.但是我还发现一个问题,那就是RTSPServer中并没有保存RTSPClientSession的列表.RTSPClientSession被创建出来就不管了,哦!从RTSPClientSession开始进入另外线程真是绝佳的时机!那RTSPClientSession被保存在哪里呢?它其实最终被保存在ServerMediaSusession的stremstate中了.当一个流播放完毕时,它自然就要被销毁了,是吧?

但是还有问题:RTSPServer中还担负者查找所有ServerMediaSession的任务,当然它是受RTSPClientSession委托的,因为ServerMediaSession们保存在RTSPServer中是理所当然的事.如果RTSPClientSession在不同的线程中呢?RTSPClientSession再查找ServerMediaSession就要进行同步保护了.还有个更严重的问题:我们希望把各个StreamState分散到不同的线程中,但它们又被保存在ServerMediaSub session中,麻烦又来了...

如果把ServerMediaSession保存到不同的线程中呢?看起来是可以的!但是又带来了问题,一个线程中的RTSPClientSession只能在自己线程的ServerMediaSession列表中查找是否已存在某个ServerMediaSession,其它线程中即使已存在了,也不能用,只能另创建一个,因为RTSPClientSession在被创建后应马上找到其UsageEnvironment,否则它就不能利用event loop接收数据了.所以ReuseSource是否能真正起作用,只能靠运气了.可不可以这样:先在主线程中执行RTSPClientSession的OPTION和DESCRIBE响应,再跟据其ServerMediaSession所在的线程,把它移到那个线程中去?我认为,这是完全能够做到的!看起来这样做,似乎有点完美了....

当然真正的实现上,如果能做到各线程之间的交互只是把DelayTask Handle放到目的线程的EventLoop中的话,并行计算的能力就真的要发挥出来了.

# 十五：RTCPInstance类小结

RTCPInstance是对rtcp通信的封装．RTCP主要是功能是统计包的收发为流量控制提供依据．RTCPInstance统计数据的取得仅依赖于RTPSink的一些函数（因为RTPSink发送RTP包），所以RTCPInstance与其它类（GroupSock,RTPInterface等基础类除外）基本关系不大，封装的比较完整．

RTCPInstance靠RTPInterface提供网络通讯支持，所以它既支持rtcp over udp,又支持rtcp over tcp．  
RTCPInstance接收到的包在函数static void incomingReportHandler(RTCPInstance\* instance, int /\*mask\*/)中处理．  
最值得关注的是这个成员函数：void setSpecificRRHandler(netAddressBits fromAddress, Port fromPort,TaskFunc\* handlerTask, void\* clientData)．它的作用是让调用者可以设置回调函数，调用者就可以在收到RR包时做出一定的动作．参数fromAddress和fromPort指明要对哪个客户端的RR包做出响应．  
利用这个机制的例子是RTSPServer::RTSPClientSession．它会把自己的RRHandler函数经过层层传递，最终传给RTCPInstance．于是RTSPServer::RTSPClientSession就可以在每次收到对应的客户端的RR包时调用它传入的函数，这个函数是void RTSPServer::RTSPClientSession::noteClientLiveness(RTSPClientSession\* clientSession).此函数只是以下函数的过渡：

1. void RTSPServer::RTSPClientSession::noteLiveness()
2. {
3. #ifdef DEBUG
4. fprintf(stderr, "Liveness indication from client at %s\n", our\_inet\_ntoa(fClientAddr.sin\_addr));
5. #endif
6. if (fOurServer.fReclamationTestSeconds > 0) {
7. envir().taskScheduler().rescheduleDelayedTask(fLivenessCheckTask,
8. fOurServer.fReclamationTestSeconds \* 1000000,
9. (TaskFunc\*) livenessTimeoutTask, this);
10. }
11. }

可以看到，每收到一次指定客户端的RR包，就重置一下livenessTimeoutTask()的运行时间，如果livenessTimeoutTask()一旦运行，看一下livenessTimeoutTask():

1. void RTSPServer::RTSPClientSession::livenessTimeoutTask(RTSPClientSession\* clientSession)
2. {
3. // If this gets called, the client session is assumed to have timed out,
4. // so delete it:
5. #ifdef DEBUG
6. fprintf(stderr, "RTSP client session from %s has timed out (due to inactivity)\n", our\_inet\_ntoa(clientSession->fClientAddr.sin\_addr));
7. #endif
8. delete clientSession;
9. }

那么RTSPServer::RTSPClientSession就会自杀（真是想不开啊）．也就是说fOurServer.fReclamationTestSeconds \* 1000000是超时时间（默认好像是60秒）．

如果你想监视一个客户端，最好的方式就是向RTCPInstance注册RRHandle．

# 十六　几个重要对象的生命期

live555中很多类，类与类之间的关系复杂，从属关系不明显，层次上看起来也有些乱．所以源代码读起来比较困难，对于一些对象生命的来龙去脉也很难厘清．

但这并不能说明live555的架构不好，最适合的才是最好的，对于流媒体的处理来说，live555架构已是相当精巧，当然，这是在你深入了解它的基础上才会有的体会．

live555作为服务器，大家都很关心对内存的利用效率，是否过多的吃内存？是否造成太多的内存碎片？

我个人认为不必太担心这方面的事，live555对于内存的使用效率还是比较高的，当然要求太高的可能要自己实现内存池之类的东西．

然而，我在使用它的过程中，还是发现了一点小小的问题，这个问题只在某些情况下起作用．

在此不对内存管理做全面的阐述，只是探讨一下live555中一些重要类的对象实体是怎样被销毁的，同时说明那点小问题．

首先说创世者：是RTSPServer：它需永存，其余对象都是由它创建或由它引起了它们的创建．  
RTSPServer直接掌管的是ServerMediaSession和RTSPClientSession(只主其生，不掌其死)．  
ServerMediaSession对应一个媒体文件，而RTSPClientSession对应一个RTSP客户连接．RTSPClientSession在客户发出RTSP的TCP连接时建立，而ServerMediaSession在客户发出对一个新文件的DESCRIBE时建立．建立ServerMediaSession的同时也建立了ServerMediaSubsession们，被ServerMediaSession所管理，代表一个文件中的track们．  
ServerMediaSession的建立规则值得一说：RTSPClientSession在收到客户的DESCRIBE请求时，跟据地址中的媒体名字，去查找ServerMediaSession的列表，如果已有对应此媒体名字的ServerMediaSession，则利用它获取SDP信息．如果没找到，则跟据媒体名字中的扩展名部分，建立对应此类媒体的新ServerMediaSession对象．所以可以明确一点：一个ServerMediaSubsession对应一个文件！  
但是，如果测试，你会发现当一个文件播放完毕之后，并没有删除对应的ServerMediaSession．同时，与ServerMediaSubsession相关的那一坨东西（Demux和ServerMediaSubsession)也没有被销毁．但是它们终究还是要面临死亡的．什么时候死呢？RTSPServer销毁的什候（或对应的文件不存在了时）！哦，看到问题了吧？如果你做点播服务器，每打开一个文件就会创建一个ServerMediaSession以及相关的一坨东西们，如果文件太多，内存终究有用完的时候．  
再说一下RTSPClientSession，RTSPClientSession有两种结束生命的方式，一是在对应流(StreamState)接收不到RTCP数据了，还记得前面讲过RTCPInstance的setSpecificRRHandler()吗？RTSPClientSession就是通过它来监视客户端的心跳的．二种方式是收到客户端的TEARDOWN请求时自杀．RTSPClientSession自杀的同时会把流对象StreamState以及流上的Source和sink全干掉．  
所以说，除了RTSPClientSession那一坨之外，其余的对象还是可以在适当的时候销毁的．基本上是代表静态数据的对象不销毁，而代表动态数据的对象销毁．  
如果你做的是实时流媒体，那么这正是所需要的．而做点播服务呢？总不能文件关了，代表文件的对象还在内存中吧？  
那我们如何去改呢？  
其实很简单，我们只要在没有任何对ServerMediaSession的引用时把它删除不就行了．而且ServerMediaSession中已经实现了引用计数，见如下三个函数：

1. unsigned referenceCount() const
2. {
3. return fReferenceCount;
4. }
5. void incrementReferenceCount()
6. {
7. ++fReferenceCount;
8. }
9. void decrementReferenceCount()
10. {
11. if (fReferenceCount > 0)
12. --fReferenceCount;
13. }

现在的问题是何时减少这个引用计数．可以想象，基本情况是在建立一个新的StreamState时或建立RTSPClientSession时，ServerMediaSession的引用就会增加１.那么理应在RTSPClientSession关闭时减去１.我们看看源码，是否是这样做了？  
经查找，是在建立新的StreamState时．在函数void RTSPServer::RTSPClientSession::handleCmd\_SETUP(char const\* cseq,char const\* urlPreSuffix, char const\* urlSuffix,char const\* fullRequestStr)中可以看到．再找一下减少引用的代码：

1. RTSPServer::RTSPClientSession::~RTSPClientSession()
2. {
3. closeSockets();
5. if (fSessionCookie != NULL)
6. {
7. // We were being used for RTSP-over-HTTP tunneling.  Remove ourselves from the 'session cookie' hash table before we go:
8. fOurServer.fClientSessionsForHTTPTunneling->Remove(fSessionCookie);
9. delete[] fSessionCookie;
10. }
12. reclaimStreamStates();
14. if (fOurServerMediaSession != NULL)
15. {
16. fOurServerMediaSession->decrementReferenceCount();
17. if (fOurServerMediaSession->referenceCount() == 0
18. && fOurServerMediaSession->deleteWhenUnreferenced())
19. {
20. fOurServer.removeServerMediaSession(fOurServerMediaSession);
21. fOurServerMediaSession = NULL;
22. }
23. }
24. }

是在RTSPClientSession销毁时减少引用．同时我们还看到

1. if (fOurServerMediaSession->referenceCount() == 0
2. && fOurServerMediaSession->deleteWhenUnreferenced())
3. {
4. fOurServer.removeServerMediaSession(fOurServerMediaSession);
5. fOurServerMediaSession = NULL;
6. }

这样的语句，翻译过来就是：当引用为０并且可以在引用为０时删除，那么就删除它！原来在这里！我们只要让他deleteWhenUnreferenced()能返回True就解决上面所说的那个小问题了．  
等等，似乎还有问题，ServerMediaSession是RTSPClientSession在建立StreamState时增加引用，而在RTSPClientSession销毁时减少引用，如果有多个Track，StreamState是要被创建多次的？好像引用增加与减少对不起来啊！真的是这样吗？我没测试我不敢说，嘿嘿，那就先留个悬念吧．

# 十七：H264VideoStreamParser详解

很多人要做实时H264 RTP传输，那么如何充分利用live555来做呢？  
大家可以看到现有的H264VideoFileServerMediaSubsession中，sink使用了H264VideoRTPSink，source使用了H264VideoStreamFramer，然而这个连接是很复杂的，在这两个节点间要插入了很多其它的节点，其实际情况是这样的：ByteStreamFileSource-->H264VideoStreamParser-->H264VideoStreamFramer-->H264FUAFragmenter-->H264VideoRTPSink．哇！真的这么复杂吗？一点没错！  
当然你可以不用理它们的来龙去脉，你只需自己实现一个source，能采集图像并进行h264编码的source（当然你可以用CPU也可以用DSP进行编码），然后用它替代ByteStreamFileSource，就成了，比如你这个source可以叫做H264ByteStreamSource．当然为了提高效率，采集和编码部分应放在另一个线程中执行．

然而，我还是很想了解H264VideoStreamParser到底是什么，Parser到底有什么用？它做了什么？它与H264VideoStreamFramer是如何配合的？它们之间有内存copy发生吗？

先设想一个问题：

H264VideoStreamFramer是什么角色？跟据H264VideoFileServerMediaSubsession的代码，H264VideoStreamFramer是真正代表source的，Sink所面对的Source就是它．但是它又连接了一个ByteStreamFileSource．look一下这部分代码：

1. FramedSource\* H264VideoFileServerMediaSubsession::
2. createNewStreamSource(unsigned /\*clientSessionId\*/,
3. unsigned& estBitrate)
4. {
5. estBitrate = 500; // kbps, estimate
7. // Create the video source:
8. ByteStreamFileSource\* fileSource = ByteStreamFileSource::createNew(envir(),
9. fFileName);
10. if (fileSource == NULL)
11. return NULL;
12. fFileSize = fileSource->fileSize();
14. // Create a framer for the Video Elementary Stream:
15. return H264VideoStreamFramer::createNew(envir(), fileSource);
16. }

是吧？我没有忽悠吧？  
ByteStreamFileSource是从文件取得数据的，它不管是到底什么媒体格式，它只是读文件．所以很明显H264VideoStreamFramer利用ByteStreamFileSource从文件取得数据，然后H264VideoStreamFramer再对数据进行分析．比如找出每个NALU，然后传给Sink．但是H264VideoStreamFramer没有自己去分析，而是利用了Parser，所以那一串中就多了一个H264VideoStreamParser．  
H264VideoStreamParser拥有两个source指针，一个是FramedSource\* fInputSource,另一个是H264VideoStreamFramer\* fUsingSource．可以看出，H264VideoStreamParser把fInputSource和fUsingSource串了起来，那么fInputSource就是ByteStreamFileSource．  
  
我们想像一下H264VideoStreamParser的所作所为：H264VideoStreamFramer把自己的缓冲（其实是sink的）传给H264VideoStreamParser,每当H264VideoStreamFramer要获取一个NALU时，就跟H264VideoStreamParser要，H264VideoStreamParser就从ByteStreamFileSource读一坨数据，然后进行分析，如果取得了一个NALU，就传给H264VideoStreamFramer.唉，H264VideoStreamFramer真是个不劳而获的坏家伙！  
  
看一下实际的流程：

1. //Sink调用Source(H264VideoStreamFramer)的GetNextFrame()获取数据，
2. //H264VideoStreamFramer从MPEGVideoStreamFramer派生，所以下面的函数会被调用：
3. void MPEGVideoStreamFramer::doGetNextFrame()
4. {
5. fParser->registerReadInterest(fTo, fMaxSize);
6. continueReadProcessing();
7. }
9. void MPEGVideoStreamFramer::continueReadProcessing(void\* clientData,
10. unsigned **char**\* /\*ptr\*/,
11. unsigned /\*size\*/,
12. struct timeval /\*presentationTime\*/)
13. {
14. MPEGVideoStreamFramer\* framer = (MPEGVideoStreamFramer\*) clientData;
15. framer->continueReadProcessing();
16. }

上两个是过渡，最终在这里执行:

1. void MPEGVideoStreamFramer::continueReadProcessing()
2. {
3. //调用Parser的parser()分析出一个NALU.如果得到了一个NALU,则
4. //用afterGetting(this)返回给Sink.
5. unsigned acquiredFrameSize = fParser->parse();
6. if (acquiredFrameSize > 0)
7. {
8. // We were able to acquire a frame from the input.
9. // It has already been copied to the reader's space.
10. fFrameSize = acquiredFrameSize;
11. fNumTruncatedBytes = fParser->numTruncatedBytes();
13. // "fPresentationTime" should have already been computed.
15. // Compute "fDurationInMicroseconds" now:
16. fDurationInMicroseconds =
17. (fFrameRate == 0.0 || ((**int**) fPictureCount) < 0) ?
18. 0 : (unsigned) ((fPictureCount \* 1000000) / fFrameRate);
19. fPictureCount = 0;
21. // Call our own 'after getting' function.  Because we're not a 'leaf'
22. // source, we can call this directly, without risking infinite recursion.
23. afterGetting(this);
24. }
25. else
26. {
27. //执行到此处并不代表parser()中没有取得数据!!
28. // We were unable to parse a complete frame from the input, because:
29. // - we had to read more data from the source stream, or
30. // - the source stream has ended.
31. }
32. }

上面这个函数的else{}中的注释大家注意了没有?这里关连到一个很难搞懂的现象,后面会解释之.这里先看一下parser()函数是怎样取得数据并进行分析的.  
parser()中读新数据是由那些test4Bytes(),skipBytes()之类的函数引起的,它们都最终调用了ensureValidBytes1():

1. void StreamParser::ensureValidBytes1(unsigned numBytesNeeded)
2. {
3. // We need to read some more bytes from the input source.
4. // First, clarify how much data to ask for:
5. unsigned maxInputFrameSize = fInputSource->maxFrameSize();
6. if (maxInputFrameSize > numBytesNeeded)
7. numBytesNeeded = maxInputFrameSize;
9. // First, check whether these new bytes would overflow the current
10. // bank.  If so, start using a new bank now.
11. if (fCurParserIndex + numBytesNeeded > BANK\_SIZE)
12. {
13. // Swap banks, but save any still-needed bytes from the old bank:
14. unsigned numBytesToSave = fTotNumValidBytes - fSavedParserIndex;
15. unsigned **char** const\* from = &curBank()[fSavedParserIndex];
17. fCurBankNum = (fCurBankNum + 1) % 2;
18. fCurBank = fBank[fCurBankNum];
19. memmove(curBank(), from, numBytesToSave);
20. fCurParserIndex = fCurParserIndex - fSavedParserIndex;
21. fSavedParserIndex = 0;
22. fTotNumValidBytes = numBytesToSave;
23. }
25. // ASSERT: fCurParserIndex + numBytesNeeded > fTotNumValidBytes
26. //      && fCurParserIndex + numBytesNeeded <= BANK\_SIZE
27. if (fCurParserIndex + numBytesNeeded > BANK\_SIZE)
28. {
29. // If this happens, it means that we have too much saved parser state.
30. // To fix this, increase BANK\_SIZE as appropriate.
31. fInputSource->envir() << "StreamParser internal error ("
32. << fCurParserIndex << "+ " << numBytesNeeded << " > "
33. << BANK\_SIZE << ")\n";
34. fInputSource->envir().internalError();
35. }
37. // Try to read as many new bytes as will fit in the current bank:
38. unsigned maxNumBytesToRead = BANK\_SIZE - fTotNumValidBytes;
39. fInputSource->getNextFrame(&curBank()[fTotNumValidBytes], maxNumBytesToRead,
40. afterGettingBytes, this, onInputClosure, this);
42. throw NO\_MORE\_BUFFERED\_INPUT;
43. }

可以看到一个奇怪的现象:这个函数没有返回值,但最终抛出了一个异常,而且只要执行这个函数,就会抛出这个异常.  
还是先分析一下这个函数做了什么吧:  
首先判断自己的缓冲区是否能容纳所需的数据量,如果实在不能,也只能提示一下,最后从ByteStreamFileSource获取一坨数据.curBack()返回的就是Parser自己的缓冲.而afterGettingBytes这个回调函数是H264VideoStreamFramer传入的,所以获取数据之后会执行H264VideoStreamFramer的函数,中转几下后,最终执行的就是上面的void MPEGVideoStreamFramer::continueReadProcessing().哇,看到了一个问题:Parser()中嵌套执行Parser()!而第二次执行Parser()完成后,返回到ensureValidBytes1(),然后由于抛出异常而退出,退出到哪里了呢?退回到上次调用的Parser()中了,因为Parser()中写了try{}catch{}.catch{}中的代码如下:

1. catch (**int** /\*e\*/)
2. {
3. #ifdef DEBUG
4. fprintf(stderr, "H264VideoStreamParser::parse() EXCEPTION (This is normal behavior - \*not\* an error)\n");
5. #endif
6. return 0; // the parsing got interrupted
7. }

可见parser()此时返回0,parser()返回0就执行到MPEGVideoStreamFramer::continueReadProcessing()中的else{}部分了,回去看看吧,其实啥也没做.也就是说,第一次调用parser()时,它只是从ByteStreamFileSource获取数据,那么这个parser()获取数据后什么也不做,但实际上对NALU分析和处里在这次Parser()的调用中已经完成了,不是在它本身完成的,而是在它引起了parser()的嵌套调用中完成.好迷糊,理顺一下过程就知道了:  
sink要获取数据,执行到MPEGVideoStreamFramer::continueReadProcessing(),MPEGVideoStreamFramer::continueReadProcessing调用parser(),parser()要使用数据时发现没有,于是ensureValidBytes1()被调用来从ByteStreamFileSource获取数据,取得数据后MPEGVideoStreamFramer::afterGettingBytes()被调用,并中转到MPEGVideoStreamFramer::continueReadProcessing(),MPEGVideoStreamFramer::continueReadProcessing()被嵌套调用!,MPEGVideoStreamFramer::continueReadProcessing()中又会调用parser(),此时parser()要使用数据时发现有数据了,所以就进行分析,分析出一个NALU后,返回到MPEGVideoStreamFramer::continueReadProcessing(),MPEGVideoStreamFramer::continueReadProcessing()会调用afterGetting(this)把数据返回给sink.sink处理完数据后返回到MPEGVideoStreamFramer::continueReadProcessing(),MPEGVideoStreamFramer::continueReadProcessing()再返回到ensureValidBytes1(),ensureValidBytes1()抛出异常返回到第一次被调用的parser()的catch{}中,parser()返回到第一次调用的MPEGVideoStreamFramer::continueReadProcessing()中,MPEGVideoStreamFramer::continueReadProcessing()发现parser()没有取得NALU,于是啥也不做,返回到sink中,sink会继续通过source->getNextFrame()->MPEGVideoStreamFramer::continueReadProcessing()...这样再次获取NALU.  
  
好曲折离奇的故事!不过终于讲完了!  
  
可以看到,parser中是有自己的缓冲的,而且其大小是固定的:  
#define BANK\_SIZE 150000  
你自己写Source时,每次输出的是一帧数据,包含多个NALU,所以你只要确定你的一帧不超过150000字节,你就可以放心的往fTo中copy,如果你的帧太大,就改这个宏吧.  
  
在此公布一下**live555 QQ群号**: **185130569**.欢迎研究流媒体的人类加入,在其中可以讨论流媒体相关的任何东西,live555,ffmpeg,vlc,rtmp等等....

# live555传输Speex音频详解

* 1. Speex over RTP
     1. **RTP Speex 头部**

RTP头部在[RFC3550]中定义。本文定义RTP头部中各字段的用法。

Payload Type (PT): 本格式的负责类型号。

Marker (M) bit: 此位被用来标志一段无声后有声的开始。打在有声数据的第一个包上。 Speex支持声音检测，可以在无声时不产生帧数据。所以包可能是非连续传输的。

Extension (X) bit: 见RTP的规定。

Timestamp: 一个32位的整数，表示一个包中第一帧的采样时间。

* + 1. **Speex的RTP负载格式**

Speex的RTP负载如图1所示。本格式没有附加的头部，所以只什用标准的RTP头部  头部之后是一个或多个负载数据库（speex帧）。包尾部可能需要一些填补数据。

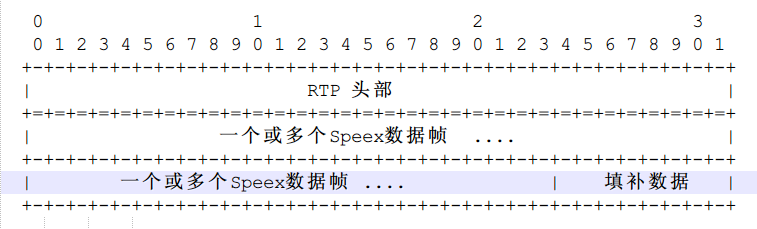


图 1: Speex的RTP负载

* + 1. **Speex** 负载

为了把编码后的数据打包进RTP，我们只需要考虑Speex编码器输出的比特流必须以相同的顺序出现在解码端。此处所说的负载格式保持了这个顺序。

一个典型的Speex帧，最大编码码率大约是110个字节。一个包中所有的Speex帧的总字节数应小于路径MTU以避免被分割。Speex帧绝不能被分割！

必须按时间序把帧打到包里。

一个RTP包中可能包含相同码率的帧也可能包含不同码率的帧。然而码率是在带内传送的，每帧中包含了自己的码率，所以打包时不必在意它。

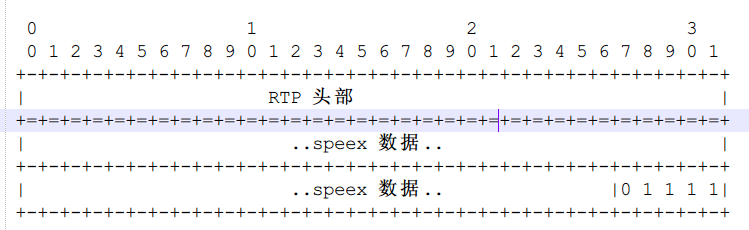
编码和解码算法可以以20毫秒的帧为边界改变码率。 码率改变的通知是在带内传送的。每个帧都包含采样率(窄带, 宽带, 或 超宽带)和"模式"(码率)信息。所以不需要带外数据通知解码器处理那些变化。

采样率必须是8000 Hz, 16000 Hz, 或 32000 Hz之一。

RTP负载必须被填补数据以保证能提供整数个字节的数据，这些填补位是LSB（最低有效位）-对齐的并且按网络字节序放置，它是由一个0跟着一群1组成的。填补数据仅被包中最后一个帧所需要。并且仅仅为了保证一个包的内容按字节边界结束。

* + 1. **Speex RTP包的例子**

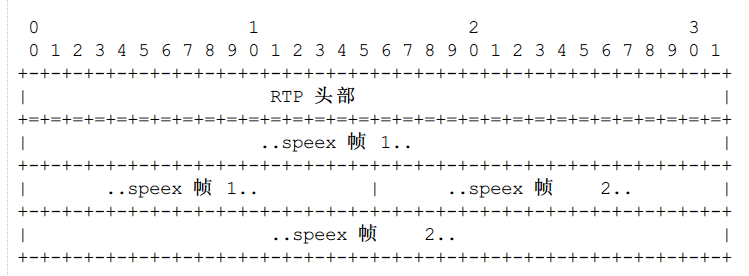
下面的示例中，我们的包中一个Speex帧，还有5位的填补数据来保证包的大小是字节对齐的。



* + 1. 多**Speex**帧的RTP包

下面的例子演示的是一个RTP包中有两个Speex帧。这个例子中的Speex帧的长度是字节对齐的，所以不需要填补数据。

Speex解码器可以从负载侦测码率，并负责在各帧之间检查20毫秒的的帧界限。



* + 1. **媒体类型**

媒体类型名字: audio

媒体字类型名字: speex

所需参数:

* rate: RTP 时间戳时钟频率，等于采样率Hz。采样率必须是8000, 16000, 或 32000。
* 可选的参数：
* ptime: 必须能被20毫秒整除 [RFC4566]
* maxptime: 必须能被20毫秒整除 [RFC4566]
* vbr: 可变码率 - 可为 'on', 'off', 或 'vad' (默认是'off')。如果是'on'，可变码率被使用。如果是 'off'，则不被使用。如果是'vad'，那么固定码率被使用，但是无声时段将被编码为特殊的短帧来表明那段时间没有声音。这个参数用于编码器。
* cng: 产生舒适噪音 - 可为 'on' 或 'off' (默认是'off')。如果为 'off'，无声帧就是无声，如果为'on'，那么这些帧将被以舒适噪音填充。此参数被用于编码器。
* mode: 以逗号分隔的多个speex支持的解码模式，按优先度排列。第一个具有最优先级，剩余的依次排列。对于窄带和宽带可用的模式值不一样，见以下定义：

\* {1,2,3,4,5,6,7,8,any} 用于窄带

\* {0,1,2,3,4,5,6,7,8,9,10,any} 用于宽带

mode'参数可能包含多个值。这此情况下，远端的编码器必须被配置成能支持模式列表中的第一个值。 当 'any' 被使用时，表明自己支持所有的解码模式。'mode' 参数必须永远有值。如果'mode' 没有出现，那么mode的值被置为：在窄带下是'mode="3,any"'，在宽带和超宽带下是'mode="8,any"' 。注意每个包含mode(或码率)的Speex帧必须被解码。因此，一个应用程序必须能解码任何Speex帧，除非在SDP中明确指明某些模式不被支持(例如，不含'mode="any"')。

解码端指定支持哪些模式意味着编码端也支持那些模式。

**1. Speex使用SDP**

当使用SDP来描述使用Speex格式的会话时，映射是下面这样的：

o 媒体类型 ("audio") 在"m="行中指定媒体的名字。

o 媒体子类型 ("speex") 在SDP "a=rtpmap"行中指定编码名字。所需的"rate"参数也在"a=rtpmap" 行中，表明时钟频率。

o 参数 "ptime" 和 "maxptime" 分别在SDP 的"a=ptime"行和"a=maxptime"中指明。

o 其它参数都在SDP的"a=fmtp"行中以parameter=value的形式指明。它们的名字与媒体内部的命名一致，并且以分号分分开不同的键值对们。

下面的表包含了mode与码率之间的对应，窄带，宽带和超宽带各不相同。并且也包含了对应的"Speex quality"的值(见Speex编码手册中的SPEEX\_SET\_QUALITY的解释)。

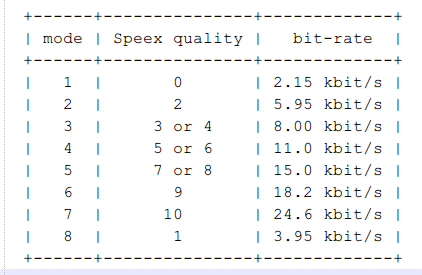


表 1: 窄带下的 Mode vs. Bit-Rate

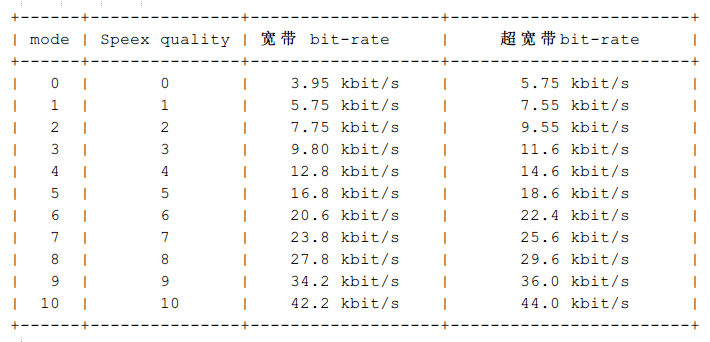


Table 2: 宽带和超宽带下的Mode vs. Bit-Rate

许多Speex特殊参数可以在一个a=fmtp行中提供，它们被分号分隔:

a=fmtp:97 mode="1,any";vbr=on

**1.1. 举例：支持所有模式，模式4优先**

m=audio 8088 RTP/AVP 97  
a=rtpmap:97 speex/8000  
a=fmtp:97 mode="4,any"

**1.2. 举例：只支持Modes 3 和 5**

m=audio 8088 RTP/AVP 97  
a=rtmap:97 speex/8000  
a=fmtp:97 mode="3,5"

**1.3. 举例：支持可变码率和舒适噪音**

m=audio 8088 RTP/AVP 97  
a=rtmap:97 speex/8000  
a=fmtp:97 vbr=on;cng=on

**1.4. 举例：支持声音动态检测**

使用vbr=vad parameter:  
m=audio 8088 RTP/AVP 97  
a=rtmap:97 speex/8000  
a=fmtp:97 vbr=vad

**1.5. 举例：支持多采样率**

表明一个Speex流支持16000Hz, mode 10 (42.2 kbit/s)；或8000Hz，mode 7 (24.6 kbit/s)。

m=audio 8088 RTP/AVP 97 98  
a=rtmap:97 speex/16000  
a=fmtp:97 mode="10,any"  
a=rtmap:98 speex/8000  
a=fmtp:98 mode="7,any"

**1.6. 举例：支持Ptime和多Speex帧**

SDP的"ptime"属性表示分包间隔(也就是，一个RTP包中编码了多少毫秒的音频数据)。因为Speex使用20毫秒一帧，ptime的值除以20就表明了一个包中有多少帧。建义使用的ptime的值可以被20整除。

如果ptime的值不能被20整除，那么需要先从它计算出最接近的能整除20的数，然后再计算帧数．例如，如果"ptime"为30，那么修正后的值就是40，然后用修正后的值来计算出帧数：为2．

下例中表明一个包中有两个帧:

m=audio 8088 RTP/AVP 97  
a=rtpmap:97 speex/8000  
a=ptime:40

**1.7. 举例：一个完整的发起者／回应者交互**

发起者表明它提供Speex 音频，采样率为16000Hz 或8000 Hz,并且发起者支持所有的模式，因为mode参数并没有指定．

m=audio 8088 RTP/AVP 97 98  
a=rtmap:97 speex/16000  
a=rtmap:98 speex/8000

回应者表明了它希望接收8000Hz的音频，这是它唯一支持的采样率．回应者支持所有的模式，因为没有指定mode参数.

m=audio 8088 RTP/AVP 99  
a=rtmap:99 speex/8000

**2. 实现方针**

支持Speex的实现者负责正确的解码Speex帧．

每个Speex帧包含所有解码需要的信息．尤其，SDP中的'mode' 和 'ptime'的值必不能在解码时使用：因为要解码一个RTP Speex流并不需要这些值．

**3. 安全相关**

本负载的安全问题符合[RFC3550]中的定义以及．这表明私密媒体流需被加密．因为本负载格式的数据压缩/解压是端对端的，加密是可以在压缩后进行的，所以两种操作不会有冲突．但是一个潜在的可能造成拒绝服务的危险是所采用的编码技术具有不一至的接收端计算负载．攻击者可以向流中注入有毒的数据报使解码变得复杂而引起接收端超负载．然而，本编码格式没有表现出任何明显的不一致问题．  
  
像其它任何的基于IP的协议，在某些环境下，一个接收者可能很简单地由于接收了太多的包而超出负载,不论是不是故意设计的．网络层的验证可以用于丢弃从不想要的源发来的包，但是验证计算本身可能耗费很多的计算．