RTP的调用过程

https://blog.csdn.net/leixiaohua1020/article/details/11850969

- RTP的传输开始于函数Boolean MediaSink::startPlaying (From MediaSink.cpp)

```
Boolean MediaSink::startPlaying(MediaSource& source,
        afterPlayingFunc* afterFunc,
        void* afterClientData) {
  // Make sure we're not already being played:
  if (fSource != NULL) {
    envir().setResultMsg("This sink is already being played");
    return False;
  }

  // Make sure our source is compatible:
  if (!sourceIsCompatibleWithUs(source)) {
    envir().setResultMsg("MediaSink::startPlaying(): source is not compatible!");
    return False;
  }
  fSource = (FramedSource*)&source;

  fAfterFunc = afterFunc;
  fAfterClientData = afterClientData;
  return continuePlaying();
}
```

- 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：(From MultiFramedRTPSink.cpp)

```
Boolean MultiFramedRTPSink::continuePlaying() {
  // Send the first packet.
  // (This will also schedule any future sends.)
  buildAndSendPacket(True);
  return True;
}
```

- 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():

```

void MultiFramedRTPSink::buildAndSendPacket(Boolean isFirstPacket) {
```

```
  nextTask() = NULL;
  fIsFirstPacket = isFirstPacket;

  // Set up the RTP header:
  unsigned rtpHdr = 0x80000000; // RTP version 2; marker ('M') bit not set (by default;
  rtpHdr |= (fRTPPayloadType<<16);
  rtpHdr |= fSeqNo; // sequence number
  fOutBuf->enqueueWord(rtpHdr);

  // Note where the RTP timestamp will go.
  // (We can't fill this in until we start packing payload frames.)
  fTimestampPosition = fOutBuf->curPacketSize();
  fOutBuf->skipBytes(4); // leave a hole for the timestamp

  fOutBuf->enqueueWord(SSRC());

  // Allow for a special, payload-format-specific header following the
  // RTP header:
  fSpecialHeaderPosition = fOutBuf->curPacketSize();
  fSpecialHeaderSize = specialHeaderSize();
  fOutBuf->skipBytes(fSpecialHeaderSize);

  // Begin packing as many (complete) frames into the packet as we can:
  fTotalFrameSpecificHeaderSizes = 0;
  fNoFramesLeft = False;
  fNumFramesUsedSoFar = 0;
  packFrame();
}
```

packFrame()

```

void MultiFramedRTPSink::packFrame() {
  // Get the next frame.

  // First, skip over the space we'll use for any frame-specific header:
  fCurFrameSpecificHeaderPosition = fOutBuf->curPacketSize();
  fCurFrameSpecificHeaderSize = frameSpecificHeaderSize();
  fOutBuf->skipBytes(fCurFrameSpecificHeaderSize);
```

```
  fTotalFrameSpecificHeaderSizes += fCurFrameSpecificHeaderSize;

  // See if we have an overflow frame that was too big for the last pkt
  if (fOutBuf->haveOverflowData()) {
    // Use this frame before reading a new one from the source
    unsigned frameSize = fOutBuf->overflowDataSize();
    struct timeval presentationTime = fOutBuf->overflowPresentationTime();
    unsigned durationInMicroseconds = fOutBuf->overflowDurationInMicroseconds();
    fOutBuf->useOverflowData();

    afterGettingFrame1(frameSize, 0, presentationTime, durationInMicroseconds);
  } else {
    // Normal case: we need to read a new frame from the source
    if (fSource == NULL) return;
    fSource->getNextFrame(fOutBuf->curPtr(), fOutBuf->totalBytesAvailable(),
        afterGettingFrame, this, ourHandleClosure, this);
  }
}
```

- 从source中获取一帧数据

```

void MultiFramedRTPSink
::afterGettingFrame(void* clientData, unsigned numBytesRead,
        unsigned numTruncatedBytes,
        struct timeval presentationTime,
        unsigned durationInMicroseconds) {
  MultiFramedRTPSink* sink = (MultiFramedRTPSink*)clientData;
  sink->afterGettingFrame1(numBytesRead, numTruncatedBytes,
         presentationTime, durationInMicroseconds);
}
```

- afterGettingFrame1

```

void MultiFramedRTPSink
::afterGettingFrame1(unsigned frameSize, unsigned numTruncatedBytes,
         struct timeval presentationTime,
         unsigned durationInMicroseconds) {
  if (fIsFirstPacket) {
    // Record the fact that we're starting to play now:
```

```
    gettimeofday(&fNextSendTime, NULL);
  }

  fMostRecentPresentationTime = presentationTime;
  if (fInitialPresentationTime.tv_sec == 0 && fInitialPresentationTime.tv_usec == 0) {
    fInitialPresentationTime = presentationTime;
  }

  if (numTruncatedBytes > 0) {
    unsigned const bufferSize = fOutBuf->totalBytesAvailable();
    envir() << "MultiFramedRTPSink::afterGettingFrame1(): The input frame data was too la
      << bufferSize << ").  "
      << numTruncatedBytes << " bytes of trailing data was dropped!  Correct this by inc
      << OutPacketBuffer::maxSize + numTruncatedBytes << ", *before* creating this 'RTPS
      << OutPacketBuffer::maxSize << ".)\n";
  }
  unsigned curFragmentationOffset = fCurFragmentationOffset;
  unsigned numFrameBytesToUse = frameSize;
  unsigned overflowBytes = 0;

  // If we have already packed one or more frames into this packet,
  // check whether this new frame is eligible to be packed after them.
  // (This is independent of whether the packet has enough room for this
  // new frame; that check comes later.)
  if (fNumFramesUsedSoFar > 0) {
    if ((fPreviousFrameEndedFragmentation
   && !allowOtherFramesAfterLastFragment())
  || !frameCanAppearAfterPacketStart(fOutBuf->curPtr(), frameSize)) {
      // Save away this frame for next time:
      numFrameBytesToUse = 0;
      fOutBuf->setOverflowData(fOutBuf->curPacketSize(), frameSize,
             presentationTime, durationInMicroseconds);
    }
  }
  fPreviousFrameEndedFragmentation = False;

  if (numFrameBytesToUse > 0) {
    // Check whether this frame overflows the packet
    if (fOutBuf->wouldOverflow(frameSize)) {
      // Don't use this frame now; instead, save it as overflow data, and
```

```
      // send it in the next packet instead.  However, if the frame is too
      // big to fit in a packet by itself, then we need to fragment it (and
      // use some of it in this packet, if the payload format permits this.)
      if (isTooBigForAPacket(frameSize)
          && (fNumFramesUsedSoFar == 0 || allowFragmentationAfterStart())) {
        // We need to fragment this frame, and use some of it now:
        overflowBytes = computeOverflowForNewFrame(frameSize);
        numFrameBytesToUse -= overflowBytes;
        fCurFragmentationOffset += numFrameBytesToUse;
      } else {
        // We don't use any of this frame now:
        overflowBytes = frameSize;
        numFrameBytesToUse = 0;
      }
      fOutBuf->setOverflowData(fOutBuf->curPacketSize() + numFrameBytesToUse,
             overflowBytes, presentationTime, durationInMicroseconds);
    } else if (fCurFragmentationOffset > 0) {
      // This is the last fragment of a frame that was fragmented over
      // more than one packet.  Do any special handling for this case:
      fCurFragmentationOffset = 0;
      fPreviousFrameEndedFragmentation = True;
    }
  }

  if (numFrameBytesToUse == 0 && frameSize > 0) {
    // Send our packet now, because we have filled it up:
    sendPacketIfNecessary();
  } else {
    // Use this frame in our outgoing packet:
    unsigned char* frameStart = fOutBuf->curPtr();
    fOutBuf->increment(numFrameBytesToUse);
        // do this now, in case "doSpecialFrameHandling()" calls "setFramePadding()" to a

    // Here's where any payload format specific processing gets done:
    doSpecialFrameHandling(curFragmentationOffset, frameStart,
         numFrameBytesToUse, presentationTime,
         overflowBytes);

    ++fNumFramesUsedSoFar;
```

```
    // Update the time at which the next packet should be sent, based
    // on the duration of the frame that we just packed into it.
    // However, if this frame has overflow data remaining, then don't
    // count its duration yet.
    if (overflowBytes == 0) {
      fNextSendTime.tv_usec += durationInMicroseconds;
      fNextSendTime.tv_sec += fNextSendTime.tv_usec/1000000;
      fNextSendTime.tv_usec %= 1000000;
    }

    // Send our packet now if (i) it's already at our preferred size, or
    // (ii) (heuristic) another frame of the same size as the one we just
    //      read would overflow the packet, or
    // (iii) it contains the last fragment of a fragmented frame, and we
    //      don't allow anything else to follow this or
    // (iv) only one frame per packet is allowed:
    if (fOutBuf->isPreferredSize()
        || fOutBuf->wouldOverflow(numFrameBytesToUse)
        || (fPreviousFrameEndedFragmentation &&
            !allowOtherFramesAfterLastFragment())
        || !frameCanAppearAfterPacketStart(fOutBuf->curPtr() - frameSize,
             frameSize) ) {
      // The packet is ready to be sent now
      sendPacketIfNecessary();
    } else {
      // There's room for more frames; try getting another:
      packFrame();
    }
  }
}
```

- 发送数据函数

```

void MultiFramedRTPSink::sendPacketIfNecessary() {
  if (fNumFramesUsedSoFar > 0) {
    // Send the packet:
#ifdef TEST_LOSS
    if ((our_random()%10) != 0) // simulate 10% packet loss #####
#endif
```

```
      if (!fRTPInterface.sendPacket(fOutBuf->packet(), fOutBuf->curPacketSize())) {
  // if failure handler has been specified, call it
  if (fOnSendErrorFunc != NULL) (*fOnSendErrorFunc)(fOnSendErrorData);
      }
    ++fPacketCount;
    fTotalOctetCount += fOutBuf->curPacketSize();
    fOctetCount += fOutBuf->curPacketSize()
      - rtpHeaderSize - fSpecialHeaderSize - fTotalFrameSpecificHeaderSizes;

    ++fSeqNo; // for next time
  }

  if (fOutBuf->haveOverflowData()
      && fOutBuf->totalBytesAvailable() > fOutBuf->totalBufferSize()/2) {
    // Efficiency hack: Reset the packet start pointer to just in front of
    // the overflow data (allowing for the RTP header and special headers),
    // so that we probably don't have to "memmove()" the overflow data
    // into place when building the next packet:
    unsigned newPacketStart = fOutBuf->curPacketSize()
      - (rtpHeaderSize + fSpecialHeaderSize + frameSpecificHeaderSize());
    fOutBuf->adjustPacketStart(newPacketStart);
  } else {
    // Normal case: Reset the packet start pointer back to the start:
    fOutBuf->resetPacketStart();
  }
  fOutBuf->resetOffset();
  fNumFramesUsedSoFar = 0;

  if (fNoFramesLeft) {
    // We're done:
    onSourceClosure();
  } else {
    // We have more frames left to send.  Figure out when the next frame
    // is due to start playing, then make sure that we wait this long before
    // sending the next packet.
    struct timeval timeNow;
    gettimeofday(&timeNow, NULL);
    int secsDiff = fNextSendTime.tv_sec - timeNow.tv_sec;
    int64_t uSecondsToGo = secsDiff*1000000 + (fNextSendTime.tv_usec - timeNow.tv_usec);
    if (uSecondsToGo < 0 || secsDiff < 0) { // sanity check: Make sure that the time-to-
```

```
      uSecondsToGo = 0;
    }

    // Delay this amount of time:
    nextTask() = envir().taskScheduler().scheduleDelayedTask(uSecondsToGo, (TaskFunc*)se
  }
}
```