RTSP

简介：

RTSP（Real Time Streaming Protocol）是由Real Network和Netscape共同提出的如何有效地在IP网络上传输流媒体数据的应用层协议。RTSP对流媒体提供了诸如暂停，快进等控制，**而它本身并不传输数据，RTSP的作用相当于流媒体服务器的远程控制。**服务器端可以自行选择使用TCP或UDP来传送串流内容，它的语法和运作跟HTTP 1.1类似，但并不特别强调时间同步，所以比较能容忍网络延迟

RTSP消息：

RTSP的消息有两大类，一是请求消息(request)，一是回应消息(response)，两种消息的格式不同。

LIVE555：

Live555是一个实现了RTSP协议的开源流媒体框架，Live555包含RTSP服务器端的实现以及RTSP客户端的实现。Live555可以将若干种格式的视频文件或者音频文件转换成视频流或者音频流在网络中通过RTSP协议分发传播，这便是流媒体服务器最核心的功能。

交叉编译live555

```
https://blog.csdn.net/tea1896/article/details/72972596
```

交叉编译openssl

```
https://www.jianshu.com/p/5f260723f5f8
```

移植

四个基本的库分别是:BasicUsageEnvironment, groupsock, liveMedia和UsageEnvironment

liveMedia库

liveMedia库包含了音视频相关的所有功能，包含音视频文件的解析，RTP传输封装等，我们可以看到这个目录下有对h264、AAC等文件解析的支持:

RTSP源码解析

各个模块简介

1. UsageEnvironment：代表了整个系统运行的环境，提供错误记录、报告和log输出错误，就需要保存UsageEnvironment的指针

```
 UsageEnvironment* env;
UsageEnvironment* env =BasicUsageEnvironment::createNew(*scheduler);
*env << "Play this stream usingthe URL \"
可以实现log的输出
```

2. TaskScheduler: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．
3. Groupsock:网络接口的封装，用于收发数据包。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．同时支持UDP和TCP协议传输，它管理着一个本地socket和多个目的地址，只需知道对方地址和端口即可发送数据。
4. BasicUsageEnvironmen： 主要是针对简单的控制台应用程序，利用select实现事件获取和处理。
5. DelayQueue：延时队列，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抛弃！
6. 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，后面再说．
7. *liveMedia*: 库中有一系列类，基类是*Medium*，这些类针对不同的流媒体类型和编码。
8.

```

```

类的实现：

BasicUsageEnvironment

- 继承关系：

```
BasicUsageEnvironment ->(继承) ->BasicUsageEnvironment0->UsageEnvironment 在include
```

```
class UsageEnvironment {
public:
  Boolean reclaim();
      // returns True iff we were actually able to delete our object

  // task scheduler:
```

```cpp
  TaskScheduler& taskScheduler() const {return fScheduler;}

  // result message handling:
  typedef char const* MsgString;
  virtual MsgString getResultMsg() const = 0;

  virtual void setResultMsg(MsgString msg) = 0;
  virtual void setResultMsg(MsgString msg1, MsgString msg2) = 0;
  virtual void setResultMsg(MsgString msg1, MsgString msg2, MsgString msg3) = 0;
  virtual void setResultErrMsg(MsgString msg, int err = 0) = 0;
  // like setResultMsg(), except that an 'errno' message is appended.  (If "err == 0", th

  virtual void appendToResultMsg(MsgString msg) = 0;

  virtual void reportBackgroundError() = 0;
  // used to report a (previously set) error message within
  // a background event

  virtual void internalError(); // used to 'handle' a 'should not occur'-type error cond

  // 'errno'
  virtual int getErrno() const = 0;

  // 'console' output:
  virtual UsageEnvironment& operator<<(char const* str) = 0;
  virtual UsageEnvironment& operator<<(int i) = 0;
  virtual UsageEnvironment& operator<<(unsigned u) = 0;
  virtual UsageEnvironment& operator<<(double d) = 0;
  virtual UsageEnvironment& operator<<(void* p) = 0;

  // a pointer to additional, optional, client-specific state
  void* liveMediaPriv;
  void* groupsockPriv;

protected:
  UsageEnvironment(TaskScheduler& scheduler); // abstract base class
  virtual ~UsageEnvironment(); // we are deleted only by reclaim()

```

```cpp
private:
  TaskScheduler& fScheduler;
};
```

1. UsageEnvironment 持有TaskScheduler的引用
2. 操作符重载是文件的输出操作：output 标准错误输出
3. setResultMsg() 函数重载：以appendToResultMsg为基础往缓存区中写入数据

BasicUsageEnvironment0：

虚函数的定义

TaskScheduler分析

- 继承关系：

```
BasicTaskScheduler继承了->BasicTaskScheduler0 继承了—-> TaskScheduler
```

- 这个类是关于任务的类

```
class TaskScheduler {
public:
  virtual ~TaskScheduler();

  virtual TaskToken scheduleDelayedTask(int64_t microseconds, TaskFunc* proc,
          void* clientData) = 0;
  // Schedules a task to occur (after a delay) when we next
  // reach a scheduling point.
  // (Does not delay if "microseconds" <= 0)
  // Returns a token that can be used in a subsequent call to
  // unscheduleDelayedTask() or rescheduleDelayedTask()
        // (but only if the task has not yet occurred).

  virtual void unscheduleDelayedTask(TaskToken& prevTask) = 0;
  // (Has no effect if "prevTask" == NULL)
        // Sets "prevTask" to NULL afterwards.
        // Note: This MUST NOT be called if the scheduled task has already occurred.

  virtual void rescheduleDelayedTask(TaskToken& task,
             int64_t microseconds, TaskFunc* proc,
             void* clientData);
        // Combines "unscheduleDelayedTask()" with "scheduleDelayedTask()"
        // (setting "task" to the new task token).
        // Note: This MUST NOT be called if the scheduled task has already occurred.

  // For handling socket operations in the background (from the event loop):
  typedef void BackgroundHandlerProc(void* clientData, int mask);
    // Possible bits to set in "mask".  (These are deliberately defined
    // the same as those in Tcl, to make a Tcl-based subclass easy.)
    #define SOCKET_READABLE    (1<<1)
    #define SOCKET_WRITABLE    (1<<2)
    #define SOCKET_EXCEPTION   (1<<3)
  virtual void setBackgroundHandling(int socketNum, int conditionSet, BackgroundHandlerPr
  void disableBackgroundHandling(int socketNum) { setBackgroundHandling(socketNum, 0, NU
  virtual void moveSocketHandling(int oldSocketNum, int newSocketNum) = 0;
        // Changes any socket handling for "oldSocketNum" so that occurs with "newSocketN

  virtual void doEventLoop(char volatile* watchVariable = NULL) = 0;
      // Causes further execution to take place within the event loop.
      // Delayed tasks, background I/O handling, and other events are handled, sequentia
      // (If "watchVariable" is not NULL, then we return from this routine when *watchVar

  virtual EventTriggerId createEventTrigger(TaskFunc* eventHandlerProc) = 0;
      // Creates a 'trigger' for an event, which - if it occurs - will be handled (from t
      // (Returns 0 iff no such trigger can be created (e.g., because of implementation l
  virtual void deleteEventTrigger(EventTriggerId eventTriggerId) = 0;

  virtual void triggerEvent(EventTriggerId eventTriggerId, void* clientData = NULL) = 0;
      // Causes the (previously-registered) handler function for the specified event to b
      // The handler function is called with "clientData" as parameter.
      // Note: This function (unlike other library functions) may be called from an exter
      // - to signal an external event.  (However, "triggerEvent()" should not be called
      // same 'event trigger id' from different threads.)

  // The following two functions are deprecated, and are provided for backwards-compatibi
  void turnOnBackgroundReadHandling(int socketNum, BackgroundHandlerProc* handlerProc, vo
    setBackgroundHandling(socketNum, SOCKET_READABLE, handlerProc, clientData);
  }
  void turnOffBackgroundReadHandling(int socketNum) { disableBackgroundHandling(socketNum

  virtual void internalError(); // used to 'handle' a 'should not occur'-type error condi

```

```
protected:
  TaskScheduler(); // abstract base class
};

```

BasicTaskScheduler0:

```
class BasicTaskScheduler0: public TaskScheduler {
public:
  virtual ~BasicTaskScheduler0();

  virtual void SingleStep(unsigned maxDelayTime = 0) = 0;
      // "maxDelayTime" is in microseconds.  It allows a subclass to impose a limit
      // on how long "select()" can delay, in case it wants to also do polling.
      // 0 (the default value) means: There's no maximum; just look at the delay queue

public:
  // Redefined virtual functions:
  virtual TaskToken scheduleDelayedTask(int64_t microseconds, TaskFunc* proc,
        void* clientData);
  virtual void unscheduleDelayedTask(TaskToken& prevTask);

  virtual void doEventLoop(char volatile* watchVariable);

  virtual EventTriggerId createEventTrigger(TaskFunc* eventHandlerProc);
  virtual void deleteEventTrigger(EventTriggerId eventTriggerId);
  virtual void triggerEvent(EventTriggerId eventTriggerId, void* clientData = NULL);

protected:
  BasicTaskScheduler0();

protected:
  // To implement delayed operations:
  DelayQueue fDelayQueue;

  // To implement background reads:
  HandlerSet* fHandlers;
  int fLastHandledSocketNum;

  // To implement event triggers:
```

```
  EventTriggerId volatile fTriggersAwaitingHandling; // implemented as a 32-bit bitmap
  EventTriggerId fLastUsedTriggerMask; // implemented as a 32-bit bitmap
  TaskFunc* fTriggeredEventHandlers[MAX_NUM_EVENT_TRIGGERS];
  void* fTriggeredEventClientDatas[MAX_NUM_EVENT_TRIGGERS];
  unsigned fLastUsedTriggerNum; // in the range [0,MAX_NUM_EVENT_TRIGGERS)
};
```

延时队列

DelayQueue:延时队列：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！

```

class DelayQueue: public DelayQueueEntry {
public:
  DelayQueue();
  virtual ~DelayQueue();

  void addEntry(DelayQueueEntry* newEntry); // returns a token for the entry
  void updateEntry(DelayQueueEntry* entry, DelayInterval newDelay);
  void updateEntry(intptr_t tokenToFind, DelayInterval newDelay);
  void removeEntry(DelayQueueEntry* entry); // but doesn't delete it
  DelayQueueEntry* removeEntry(intptr_t tokenToFind); // but doesn't delete it

  DelayInterval const& timeToNextAlarm();
  void handleAlarm();

private:
  DelayQueueEntry* head() { return fNext; }
  DelayQueueEntry* findEntryByToken(intptr_t token);
  void synchronize(); // bring the 'time remaining' fields up-to-date

  _EventTime fLastSyncTime;
};

```

DelayQueueEntry:这是延时队列中存储了一个成员的信息

```

class DelayQueueEntry {
public:
  virtual ~DelayQueueEntry();

  intptr_t token() {
    return fToken;
  }

protected: // abstract base class
  DelayQueueEntry(DelayInterval delay);

  virtual void handleTimeout();//子类需要实现这个函数，这个函数就是时间到的时候，需要执行的任

private:
  friend class DelayQueue;
  DelayQueueEntry* fNext;
  DelayQueueEntry* fPrev;
  // 用于表示延迟任务需要被执行的时间距当前时间的间隔,还需要多久能执行这个任务
  DelayInterval fDeltaTimeRemaining;//是当前结点和上一个结点要执行的时间差

  intptr_t fToken;//这个是标识哪一个结点的id，开始时由全局变量tokenCounter维护。
  static intptr_t tokenCounter;//这是一个静态变量，这个变量表示多少个条目
};
```

- addEntry()

```
void DelayQueue::addEntry(DelayQueueEntry* newEntry) {
  synchronize();

  DelayQueueEntry* cur = head();
  while (newEntry->fDeltaTimeRemaining >= cur->fDeltaTimeRemaining) {
    newEntry->fDeltaTimeRemaining -= cur->fDeltaTimeRemaining;
    cur = cur->fNext;
  }

  cur->fDeltaTimeRemaining -= newEntry->fDeltaTimeRemaining;

  // Add "newEntry" to the queue, just before "cur":
  newEntry->fNext = cur;
  newEntry->fPrev = cur->fPrev;
  cur->fPrev = newEntry->fPrev->fNext = newEntry;
}

```

1、延时队列是按照fDeltaTimeRemaining这个时间间隔的值排序的。

2，fDeltaTimeRemaining这个时间间隔是和上一个结点的fDeltaTimeRemaining这个值算出来的时间间隔。

- void synchronize():同步：**还没懂**

```
void DelayQueue::synchronize() {
  // First, figure out how much time has elapsed since the last sync:
  _EventTime timeNow = TimeNow();
  if (timeNow < fLastSyncTime) {
    // 系统时钟显然已经回到过去。 重置我们的同步时间并返回：
    fLastSyncTime  = timeNow;
    return;
  }
  DelayInterval timeSinceLastSync = timeNow - fLastSyncTime;    //算出两次同步的间隔时间
  fLastSyncTime = timeNow;    //保存现在的时间为最后同步时间

  // 然后，为时间到的所有条目调整延迟队列：
  DelayQueueEntry* curEntry = head();
  while (timeSinceLastSync >= curEntry->fDeltaTimeRemaining) {     //判断一下间隔任务中都
    timeSinceLastSync -= curEntry->fDeltaTimeRemaining;                     // 因为这是
    curEntry->fDeltaTimeRemaining = DELAY_ZERO;
    curEntry = curEntry->fNext;
  }
  curEntry->fDeltaTimeRemaining -= timeSinceLastSync;                           //  当
```

```
    //为什么后面的时间间隔不需要改变，是因为我们当初保存的是和上一个结点的时间差，时间差是不会改变的
}
```

涉及到的基本概念

- Source 、Sink

1. source：翻译为源、源头。表示数据的提供者，比方通过RTP读取数据，通过文件读取数据或者从内存读取数据，这些均能够作为Souce。
2. Sink:翻译为水槽。表述数据的流向，表示数据的流向、消费者。比方写文件、显示到屏幕等
3. Filter:翻译为过滤器；在数据流从Souce流到Sink的过程中能够设置Filter，用于过滤或做进一步加工。
4. 在整个LiveMedia中，数据都是从Souce，经过一个或多个Filter。终于流向Sink。在server中数据流是从文件或设备流向网络，而在client数据流是从网络流向文件或屏幕。

- ClientSession

对于每一个连接到server的client。server会为其创建一个ClientSession对象，保存该ClientSession的socket、ip地址等

- MediaSession、MediaSubsession、Track

Live555使用MediaSession管理一个包括音视频的媒体文件，每一个MediaSession使用文件名称唯一标识。使用SubSession管理MediaSession中的一个音频流或视频流