Apple
Darwin流媒体服务器
软件架构
以及源码分析

Andy Wu Zengde

https://www.linkedin.com/in/andyplusplus

<<index>>

概要

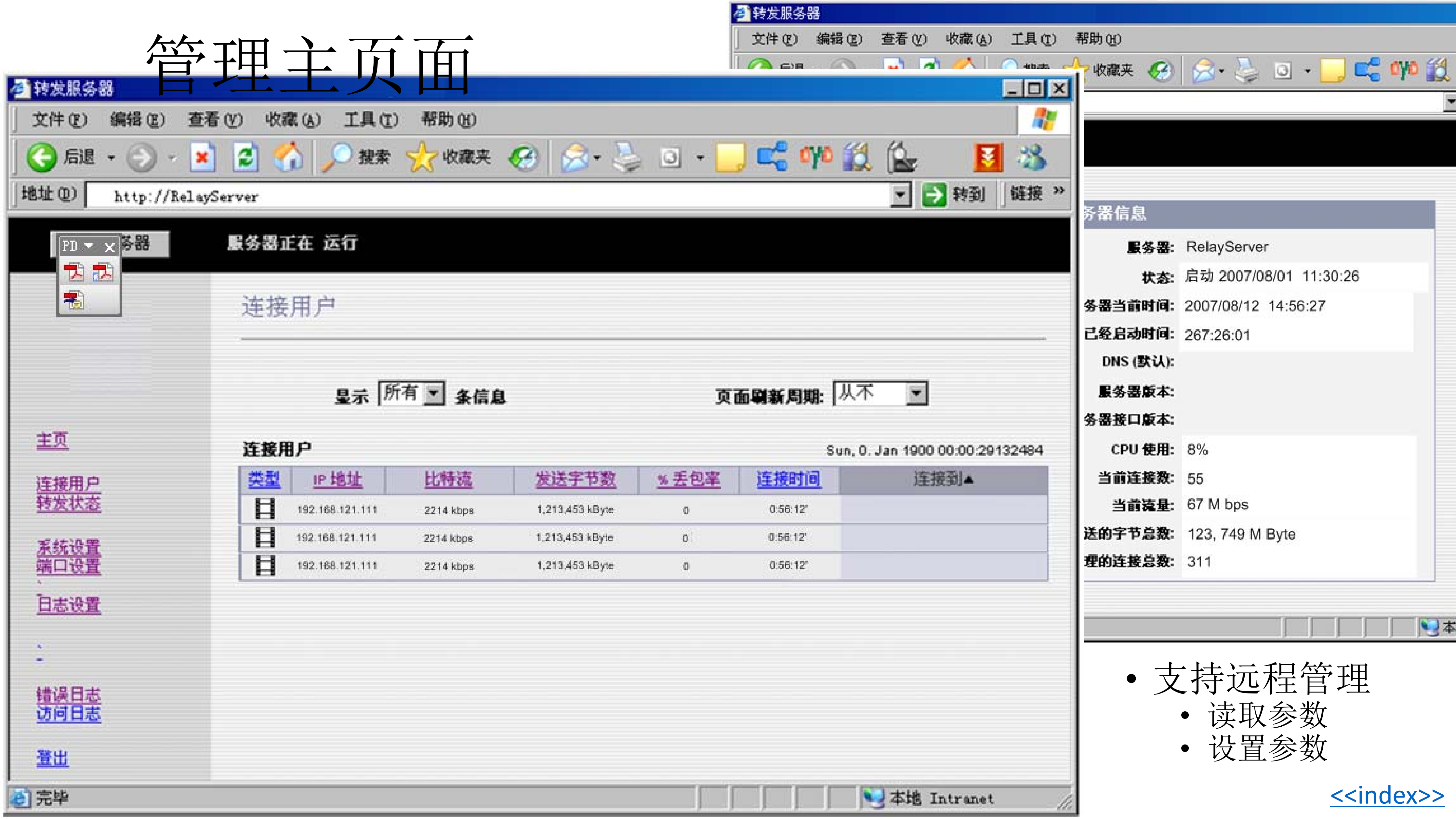
<<index>>

源码概况

- 开源于 1999
- 管理页面
  - Perl
- 服务器核心代码
  - C++
  - 代码复杂度
- 跨平台
  - Linux / Unix
  - Mac OS X
  - Windows

<<index>>

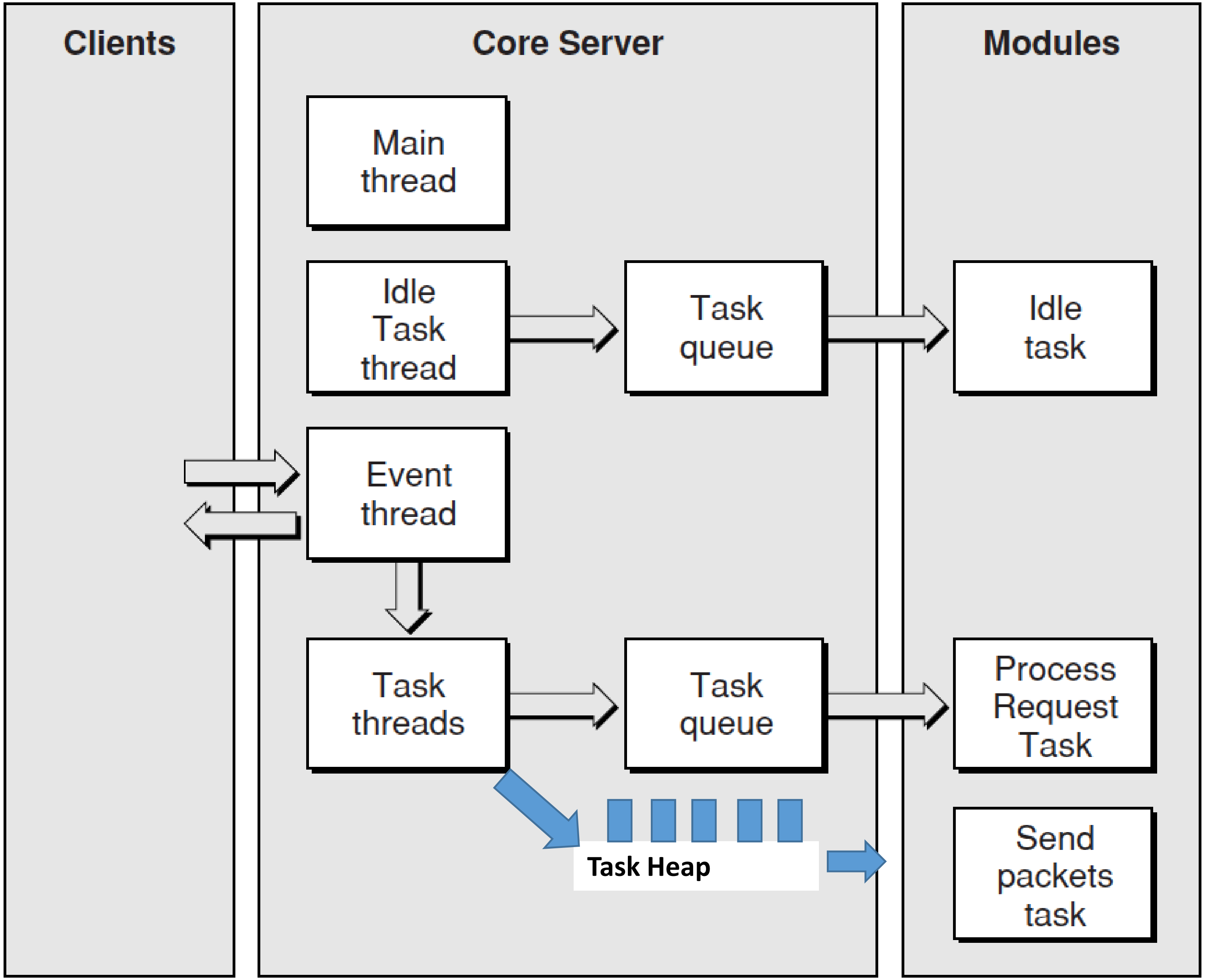
管理主页面

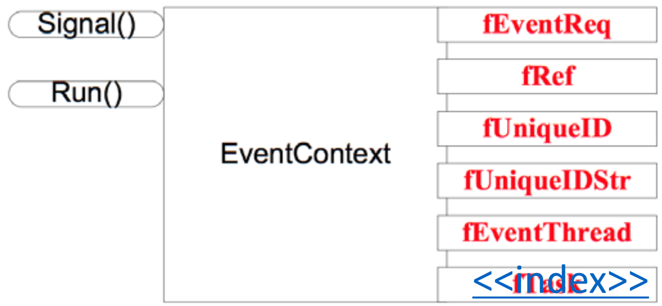


- 支持远程管理
  - 读取参数
  - 设置参数

<<index>>

体系结构

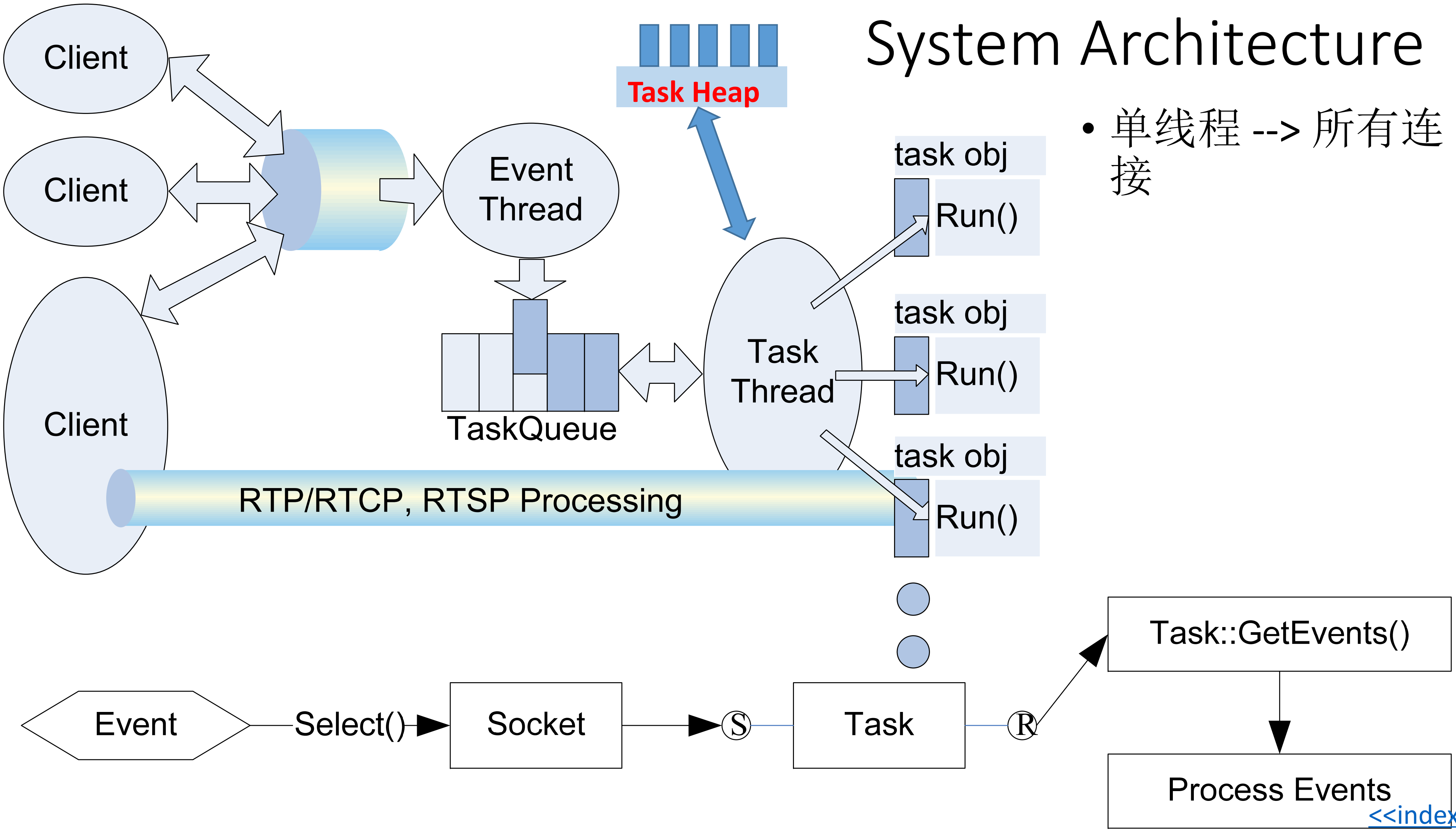

Clients
Core Server
Modules
Main thread
Idle Task thread
Task queue
Idle task
Event thread
Task threads
Task queue
Process Request Task
Task Heap
Send packets task


Signal()
Run()
EventContext
fEventReq
fRef
fUniqueID
fUniqueIDStr
fEventThread
fTask

<<index>>

System Architecture

- 单线程 --> 所有连接



<<index>>

支持的协议 16/18

- RTSP over TCP
- RTP over UTP
- RTP over Apple’s Reliable UDP
- RTSP/RTP in HTTP (Tunneled)
- RTP over RTSP (RTP over TCP)

<<index>>

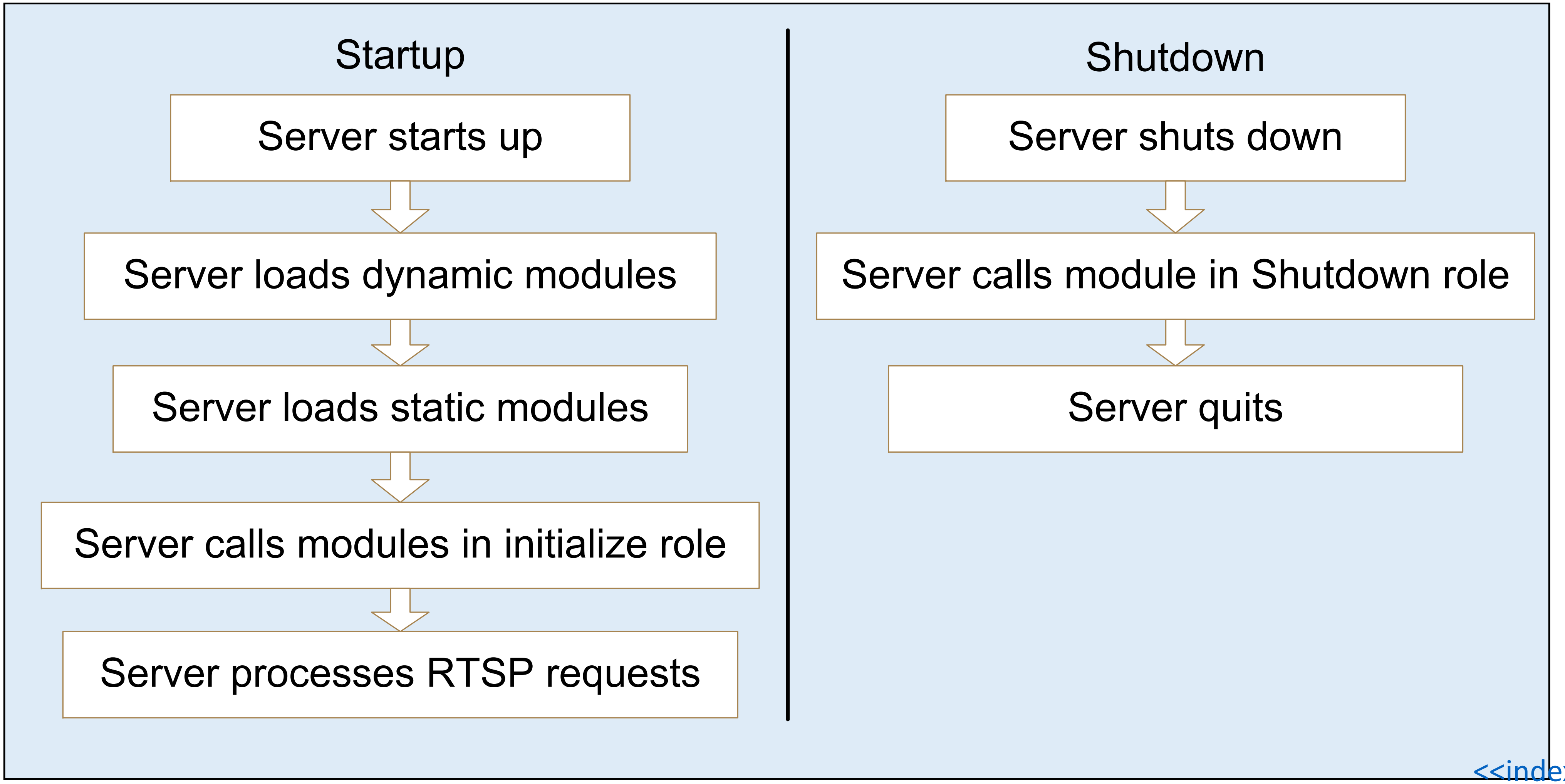
模块编程 22/24

```
QTSS_Error MyModule_Main(void* inPrivateArgs)
{
    return _stublibrary_main(inPrivateArgs, MyModuleDispatch);
}

void MyModuleDispatch(
        QTSS_Role inRole,
        QTSS_RoleParamPtr inParams);
```

<<index>>

Startup and Shutdown



<<index>>

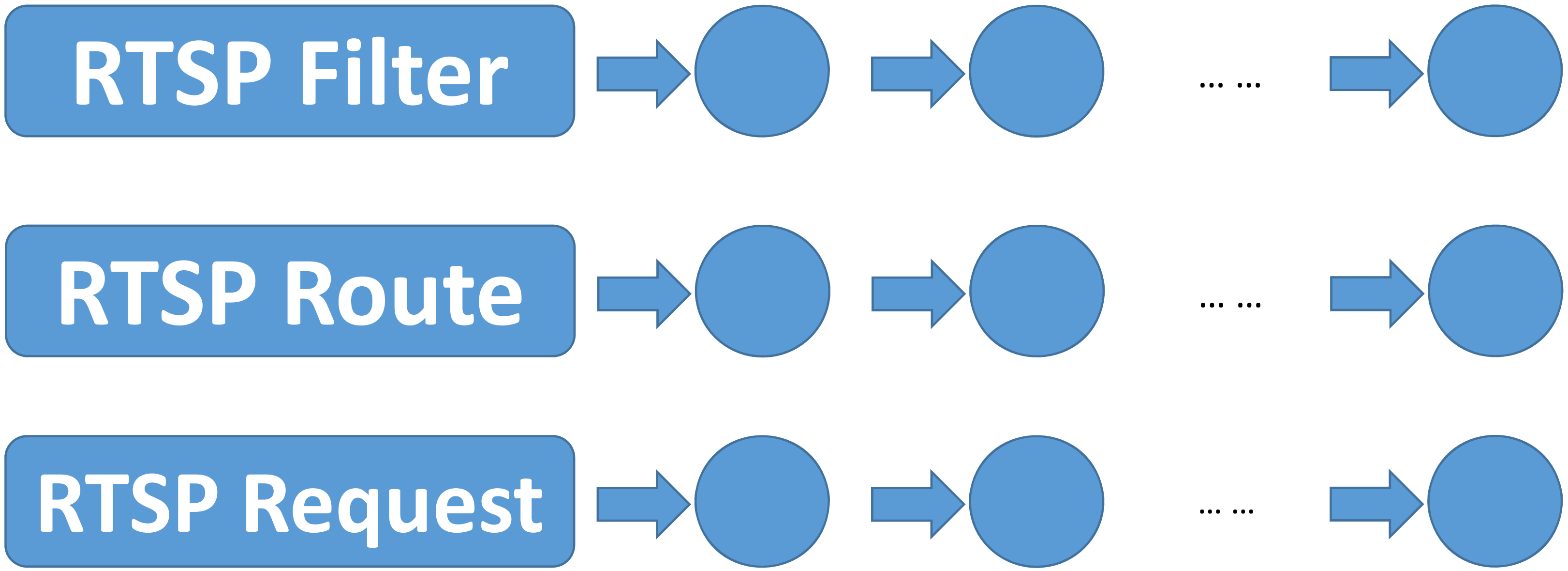
QTSSFileModule Example

```
QTSS_Error QTSSFileModule_Main(void* inPrivateArgs) {
   return _stublibrary_main(inPrivateArgs, QTSSFileModuleDispatch);
}

QTSS_Error  QTSSFileModuleDispatch(QTSS_Role inRole, QTSS_RoleParamPtr
inParamBlock) {
   switch (inRole)
   {
      case QTSS_Register_Role:
         return Register(&inParamBlock->regParams);
      case QTSS_Initialize_Role:
         return Initialize(&inParamBlock->initParams);
      case QTSS_RereadPrefs_Role:
         return RereadPrefs();
      … … … … …
   }
   return QTSS_NoErr;
}
```

<<index>>

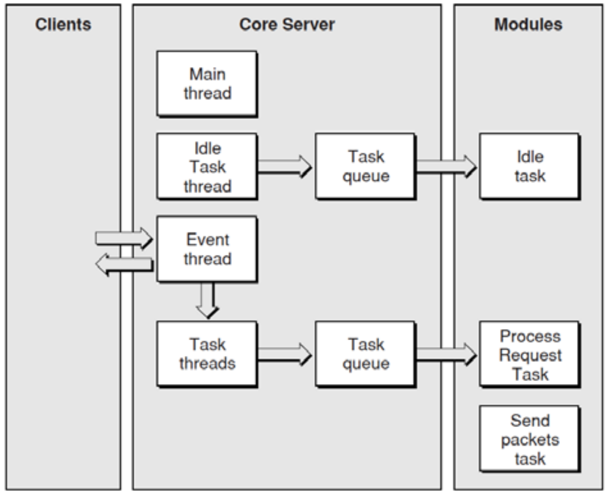
Module Role Array

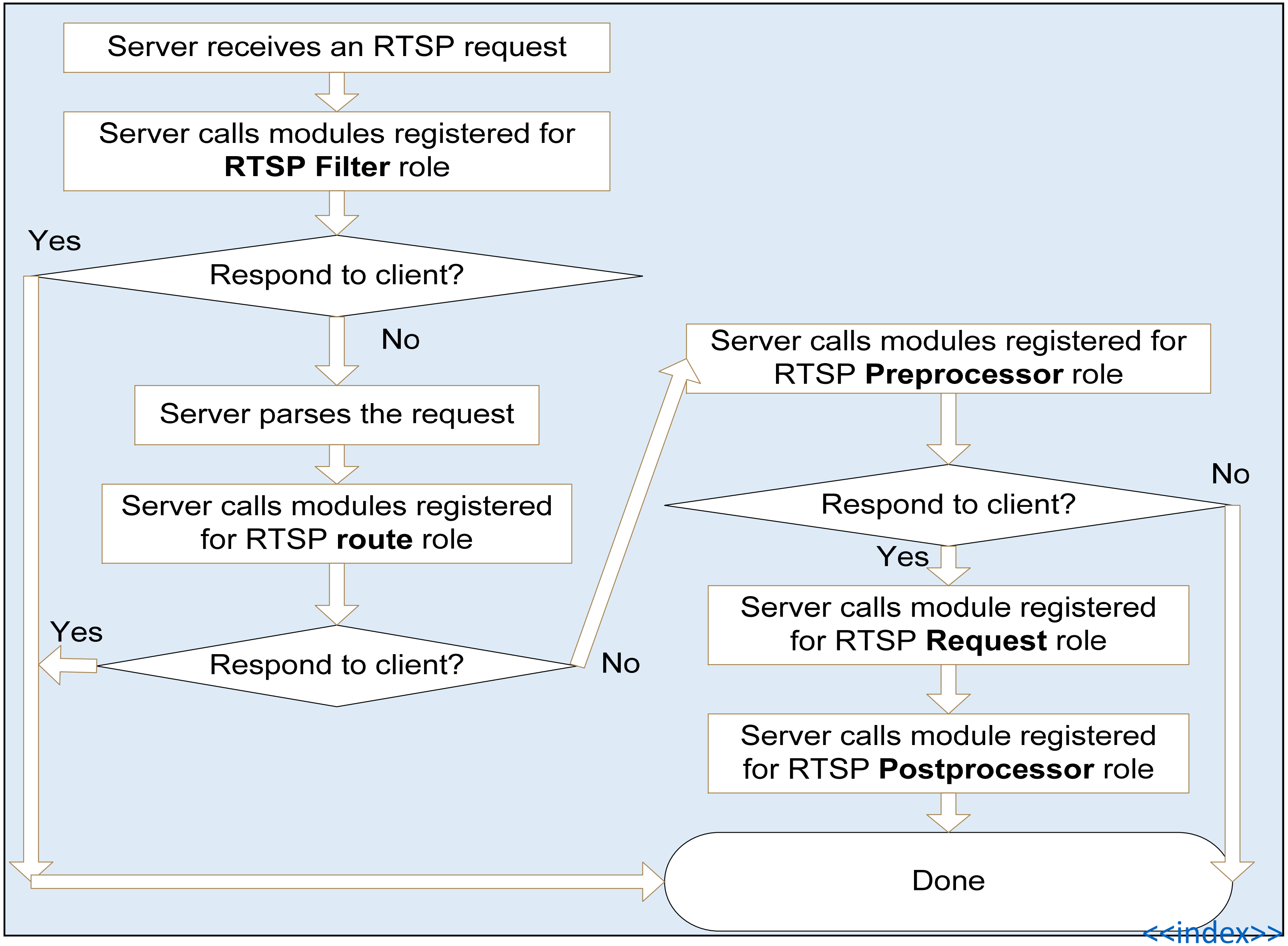


<<index>>

RTSP Request Processing

26/28



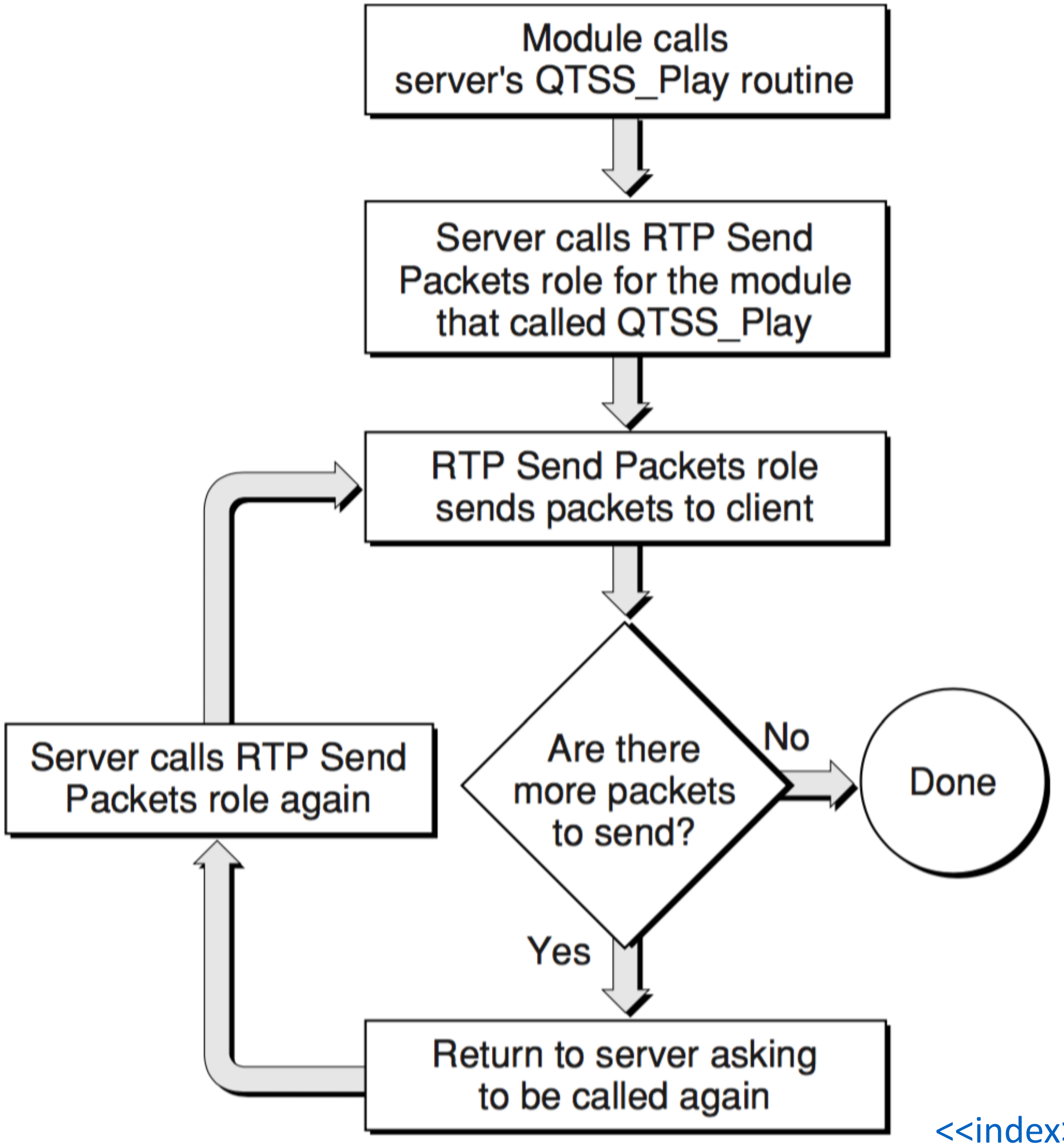


<<index>>


1. Recv an RTSP request
1. 创建一个RTSP请求对象，该对象是一些描述请求的属性的集合。
2. 此时qtssRTSPReqFullRequest属性是唯一有值的属性，其值组成了RTSP请求的完整内容。
2. RTSP Filter role
1. 服务器将RTSP请求对象作为参数传递给该角色。每个RTSP Filter角色都可以改变qtssRTSPReqFullRequest属性的值。
3. Parses the request
1. 填充RTSP对象剩余的属性值
2. 创建下面两个会话
●一个RTSP会话，当前请求相关联，客户端关闭与服务器之间的RTSP连接时，该会话随之关闭
●一个客户会话，和当前请求相关联，这个会话会一直保持，直到客户端的流播放结束。
4. RTSP Route role
1. 传入一个RTSP对象。
2. 角色使用RTSP对象中的属性值来
确定是否要改变qtssRTSPReqRootDir属性的值，进而改变用于处理当前请求的目录。
5. RTSP Preprocessor role
1. 角色通过qtssRTSPReqAbsoluteURL属性值判断当前请求是否和模块处理的请求类型相匹配
2. 匹配,角色就调用QTSS_Write或者QTSS_WriteV函数来向客户发送数据，对客户端进行响应
3. 如果只需要发送标准响应，则模块可以调用QTSS_SendStandardRTSPResponse，
或者QTSS_AppendRTSPHeader和QTSS_SendRTSPHeaders函数。
6. RTSP Request role
RTSP Request 角色负责响应所有没有被RTSP Preprocessor角色（的模块）处理过的RTSP请求。
7. RTSP Postprocessor role
RTSP Postprocessor 角色通常执行一些统计任务，比如记录各种统计信息。

Summary of the RTSP Preprocessor and RTSP Request roles

28/30



<<index>>

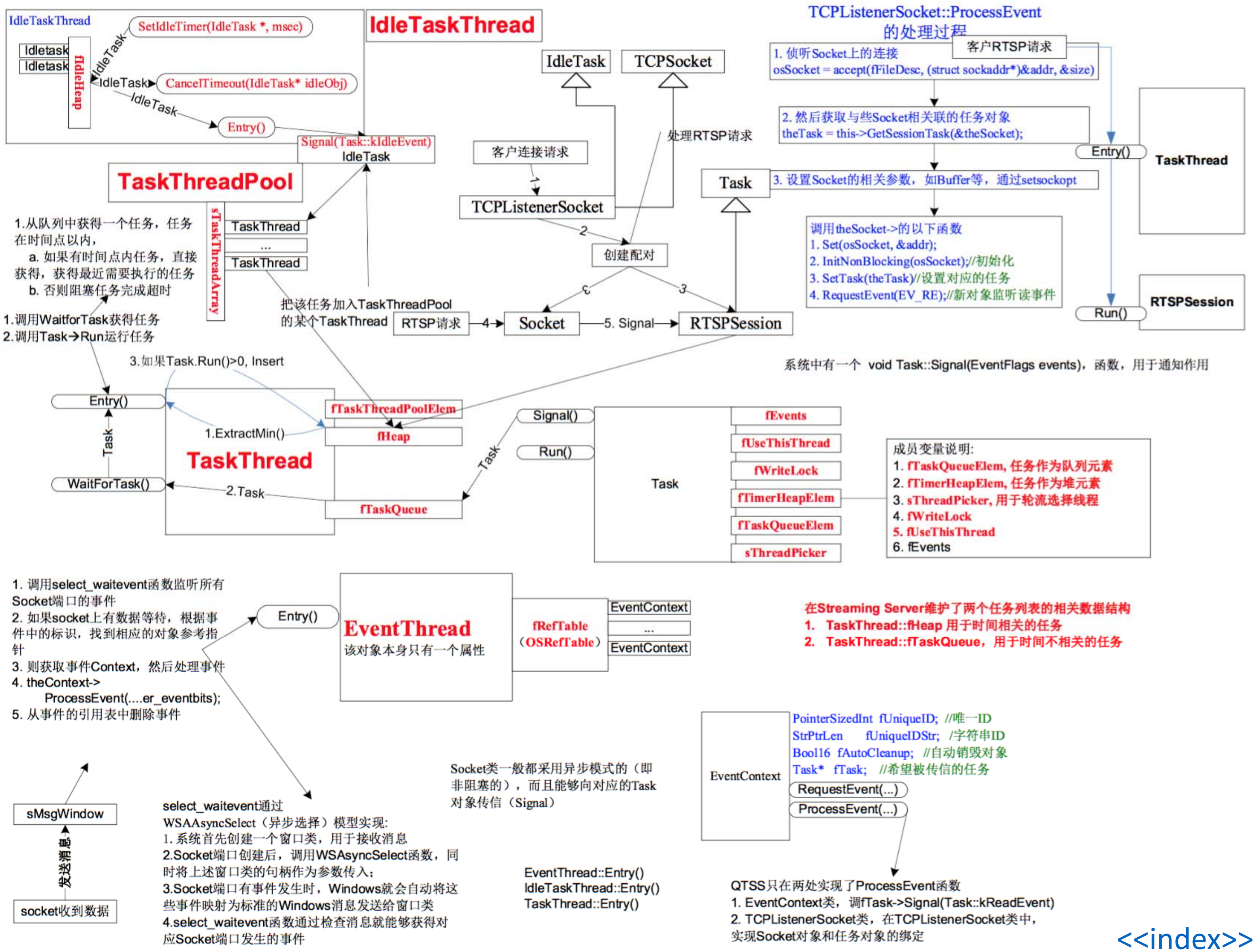
Roles

- Register role
- Initialize role
- Shutdown role
- Reread Preferences role
- Error Log role
- RTSP Filter role
- RTSP Route role
- RTSP Preprocessor role
- RTSP Request role
- RTSP Postprocessor role
- RTP Send Packets role
- Sends packets.
- Client Session Closing Role
- RTCP Process role
- Open File Preprocess role
- Open File role
- Advise File role
- Request Event File role
- Close File role

<<index>>

线程模型

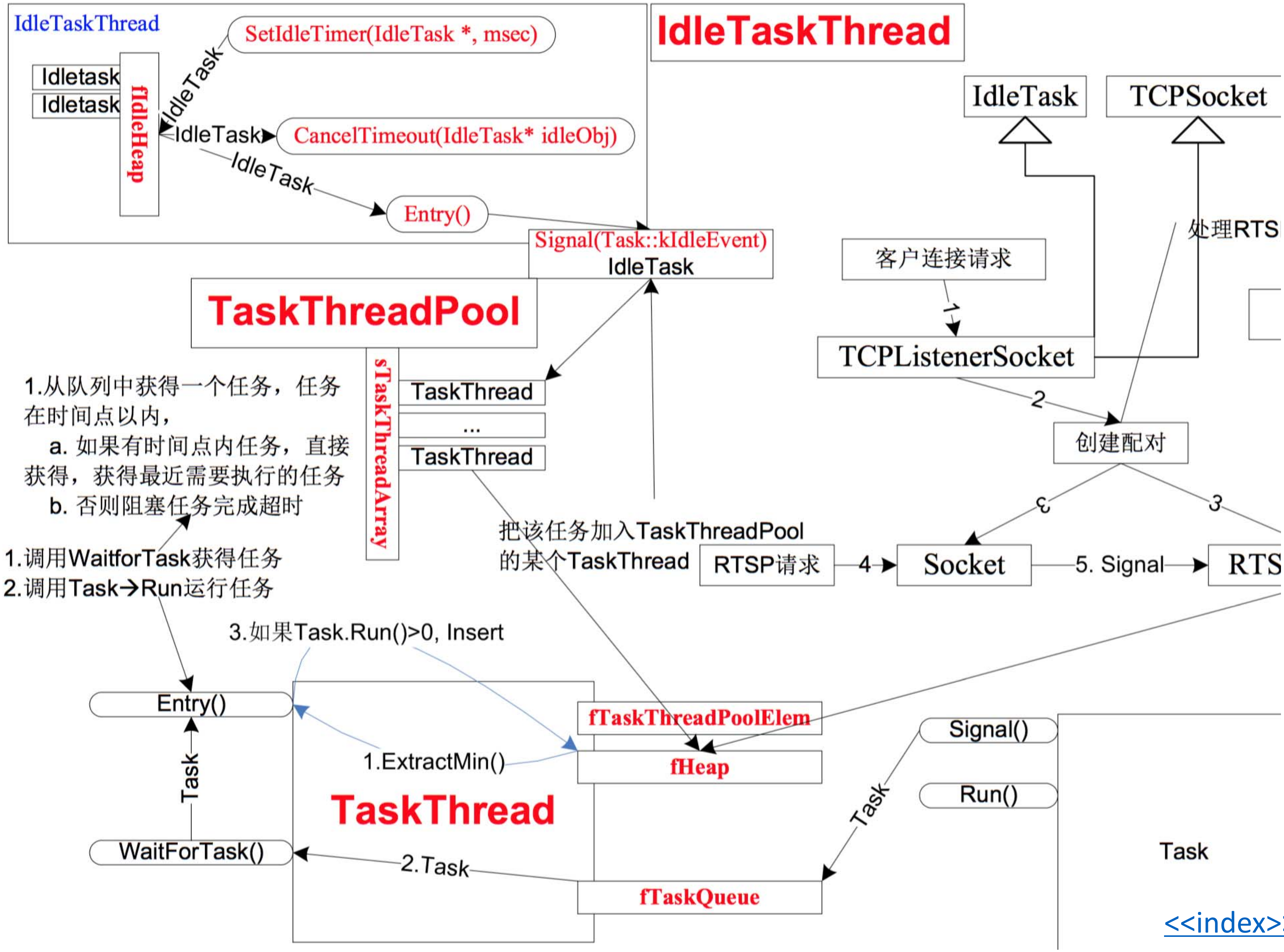
v12



<<index>>

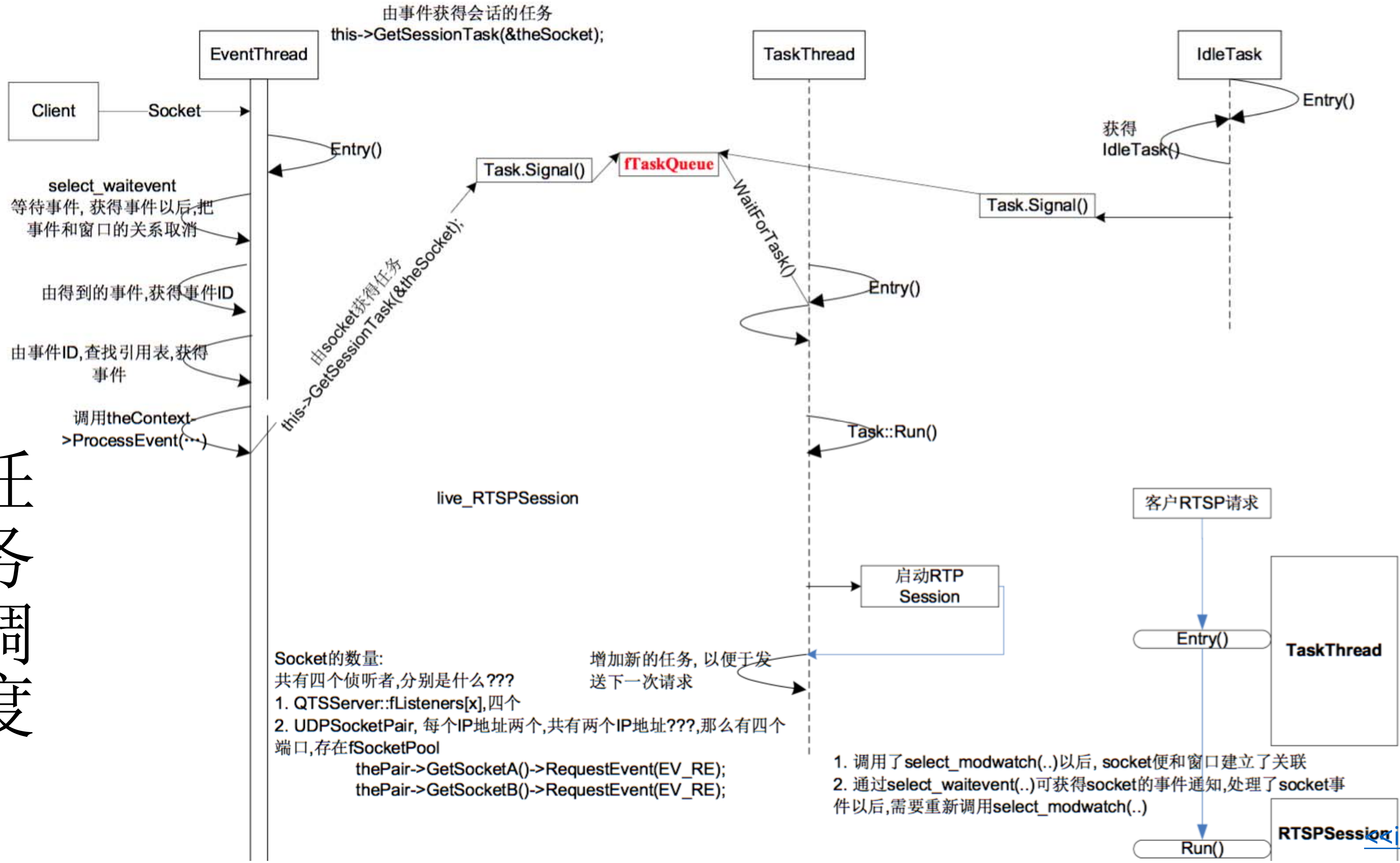
任务线程

v12



<<index>>

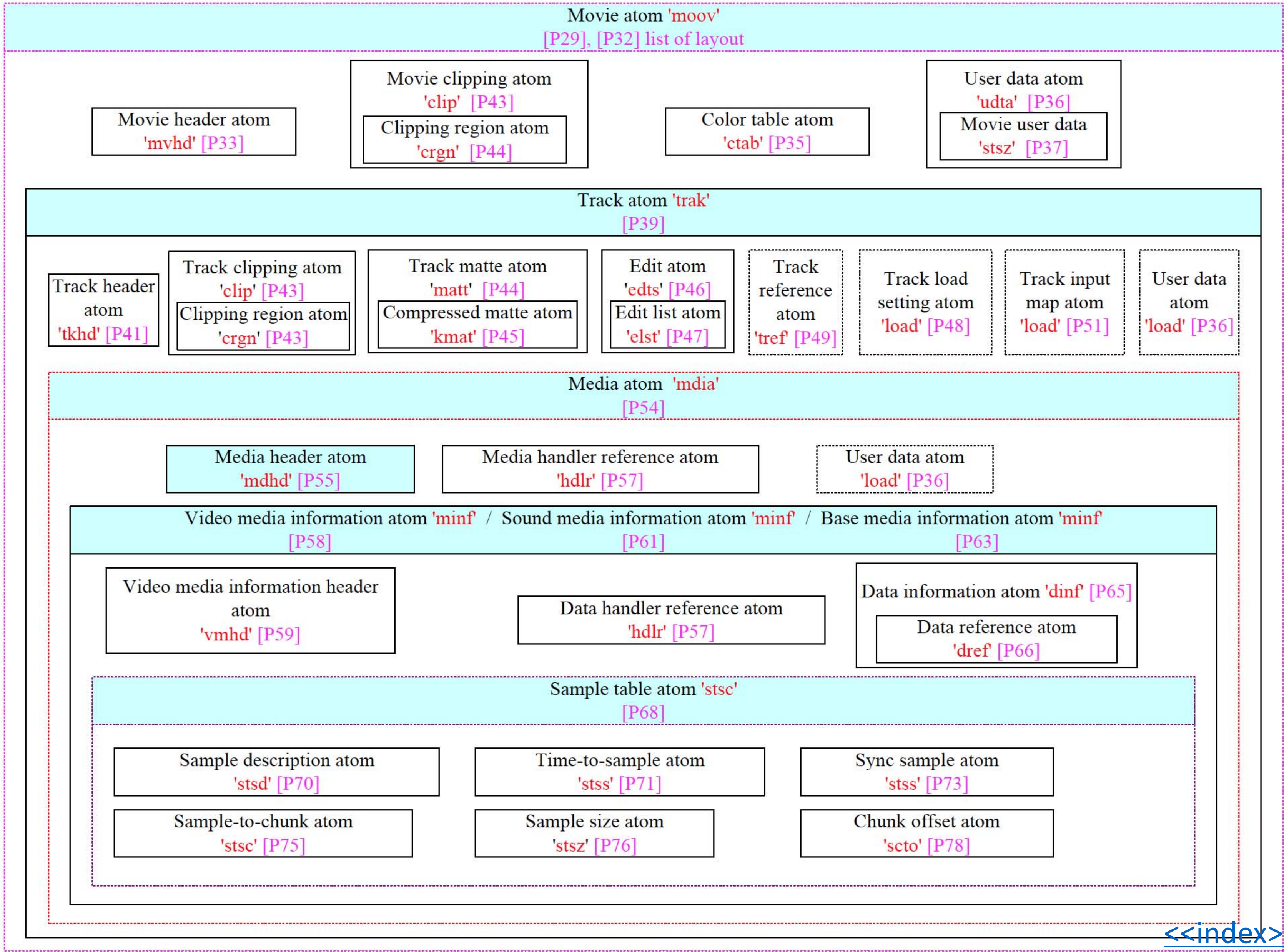
任务调度



v14

<<index>>

文件解析

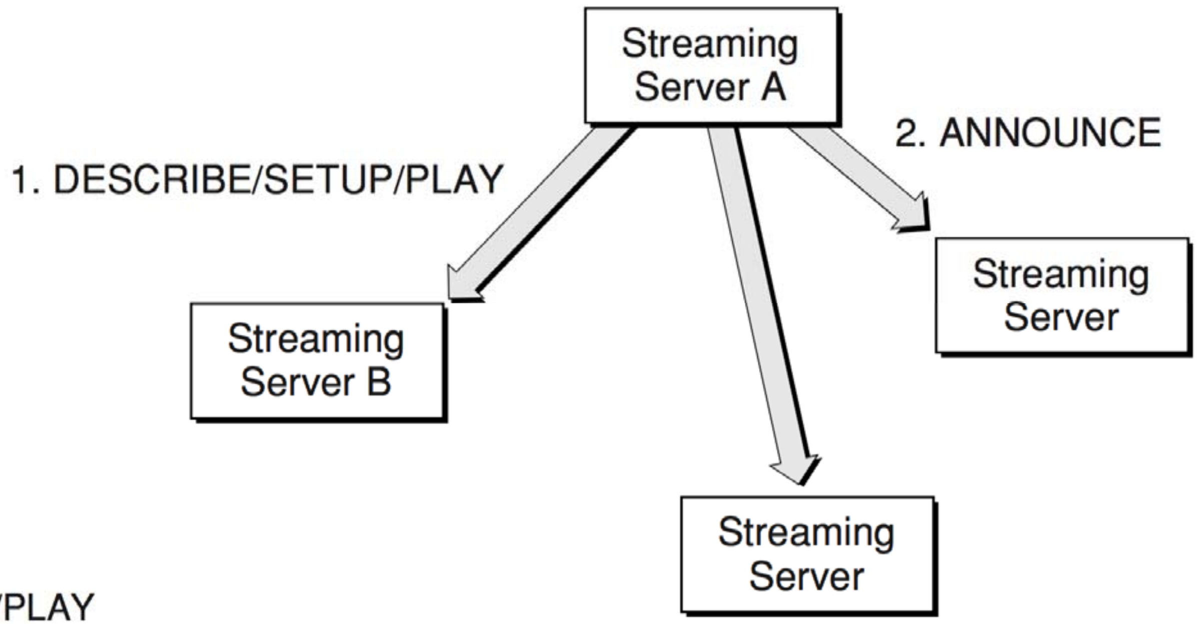


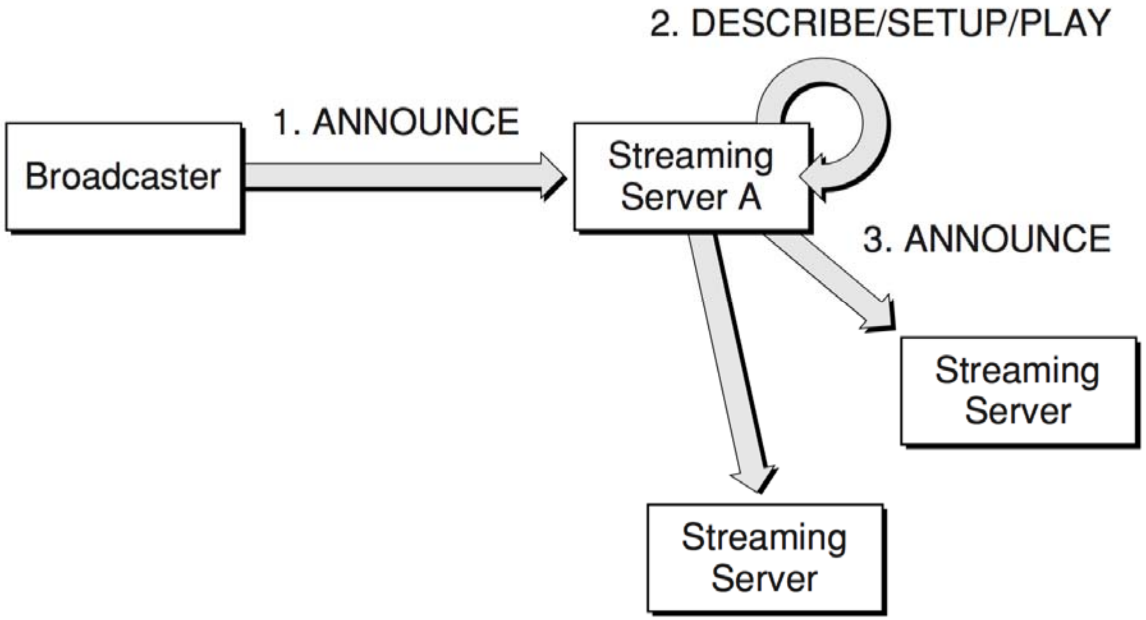
Movie atom 'moov'
[P29], [P32] list of layout
Movie header atom 'mvhd' [P33]
Movie clipping atom 'clip' [P43]
Clipping region atom 'crgn' [P44]
Color table atom 'ctab' [P35]
User data atom 'udta' [P36]
Movie user data 'stsz' [P37]
Track atom 'trak' [P39]
Track header atom 'tkhd' [P41]
Track clipping atom 'clip' [P43]
Clipping region atom 'crgn' [P43]
Track matte atom 'matt' [P44]
Compressed matte atom 'kmat' [P45]
Edit atom 'edts' [P46]
Edit list atom 'elst' [P47]
Track reference atom 'tref' [P49]
Track load setting atom 'load' [P48]
Track input map atom 'load' [P51]
User data atom 'load' [P36]
Media atom 'mdia' [P54]
Media header atom 'mdhd' [P55]
Media handler reference atom 'hdlr' [P57]
User data atom 'load' [P36]
Video media information atom 'minf' / Sound media information atom 'minf' / Base media information atom 'minf'
[P58] [P61] [P63]
Video media information header atom 'vmhd' [P59]
Data handler reference atom 'hdlr' [P57]
Data information atom 'dinf' [P65]
Data reference atom 'dref' [P66]
Sample table atom 'stsc' [P68]
Sample description atom 'stsd' [P70]
Time-to-sample atom 'stss' [P71]
Sync sample atom 'stss' [P73]
Sample-to-chunk atom 'stsc' [P75]
Sample size atom 'stsz' [P76]
Chunk offset atom 'scto' [P78]

<<index>>

automatic broadcasting

- Pull Then Push
- Listen Then Push





<<index>>

更多考虑

- 媒体服务器通过模块扩展实现
  - VAT
  - VTDU
  - BSU
  - IMS
- ……
- ……

- BSS通过HTTP查询实时显示
  - 查询媒体服务器状态信息
    - 服务器状态CPU使用
    - 当前连接数
    - 发送的总字节数
    - 处理的连接总数
  - 设置服务器参数
    - 转发设置
    - 存贮设置
    - ……
    - ……

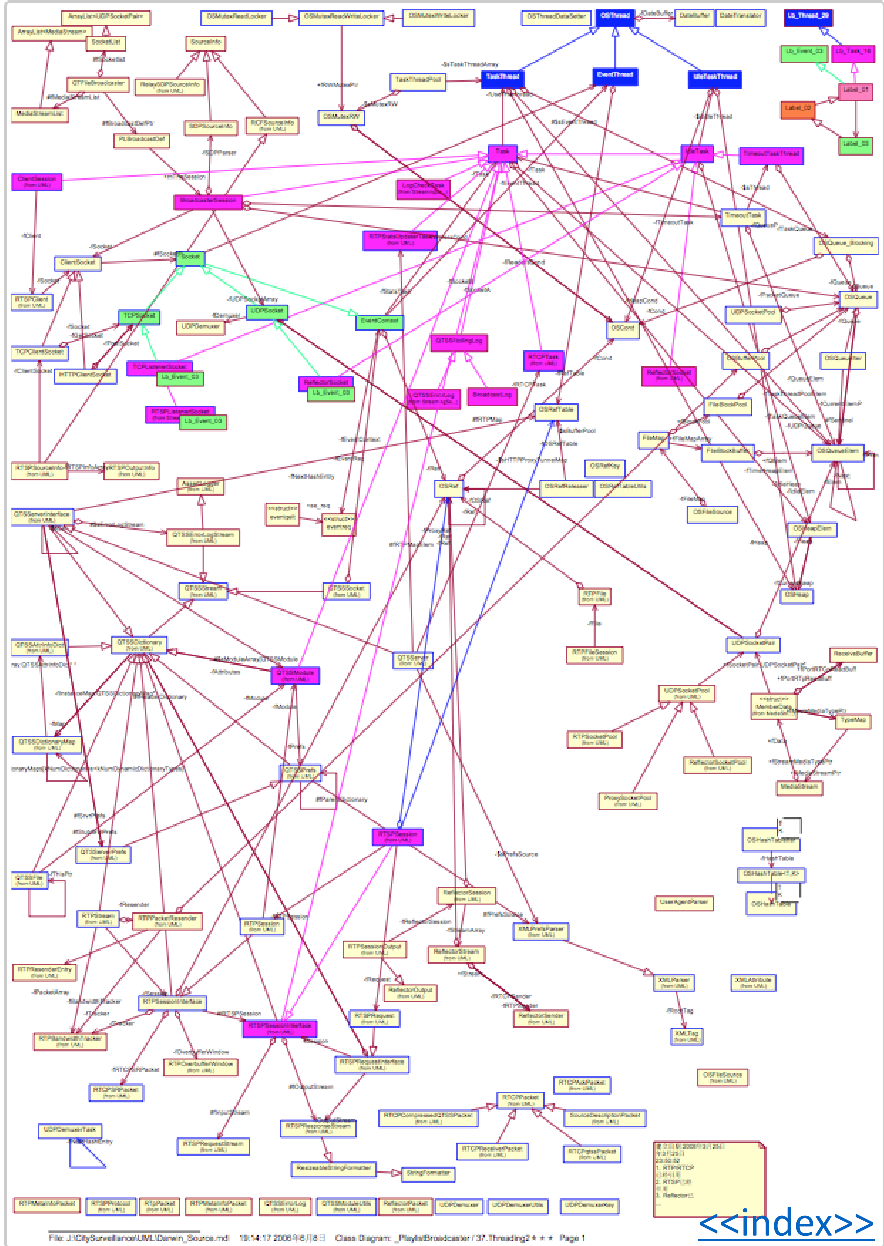
<<index>>

提问 & 讨论

<<index>>

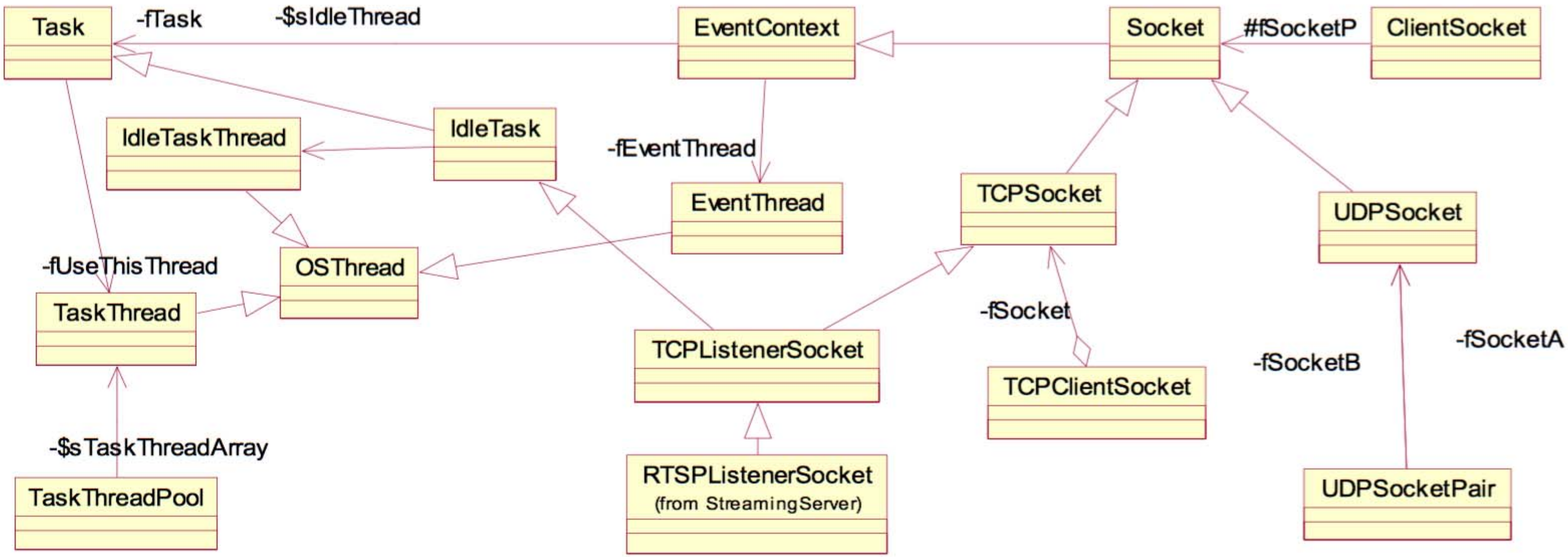
DSS类大图

- 参考
  - ref DSS主类图.pdf



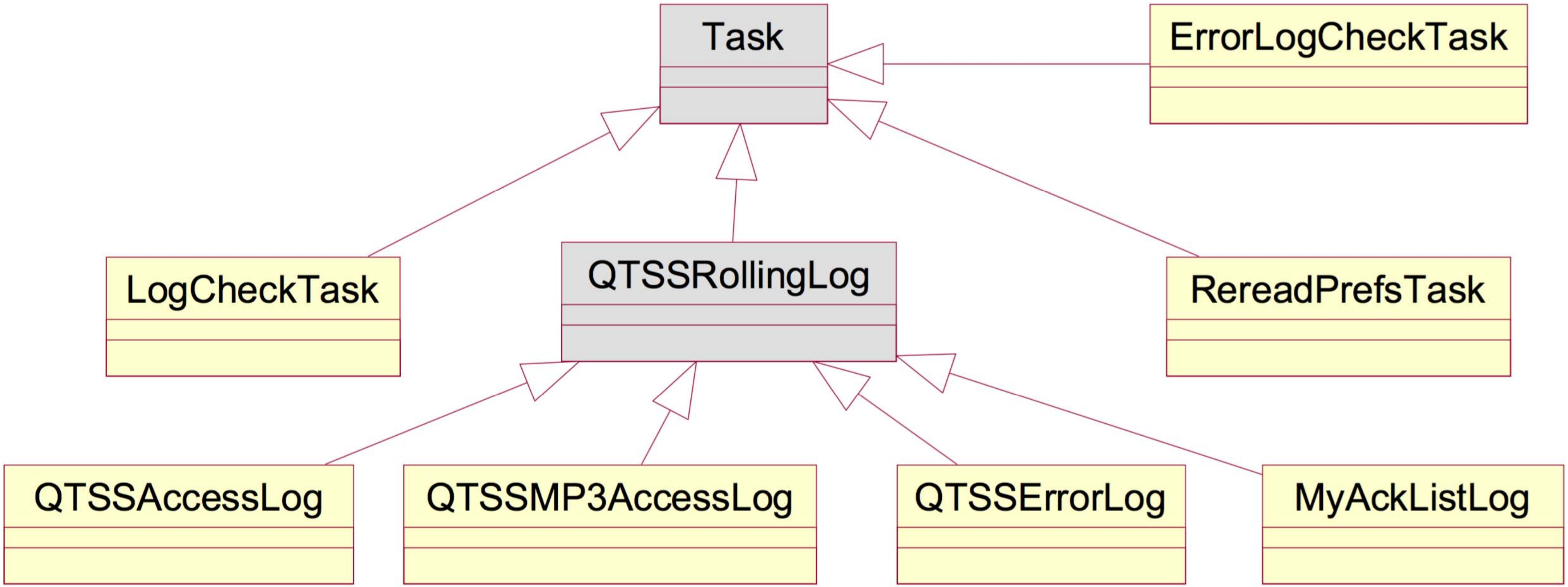
<<index>>

主要类图 v15

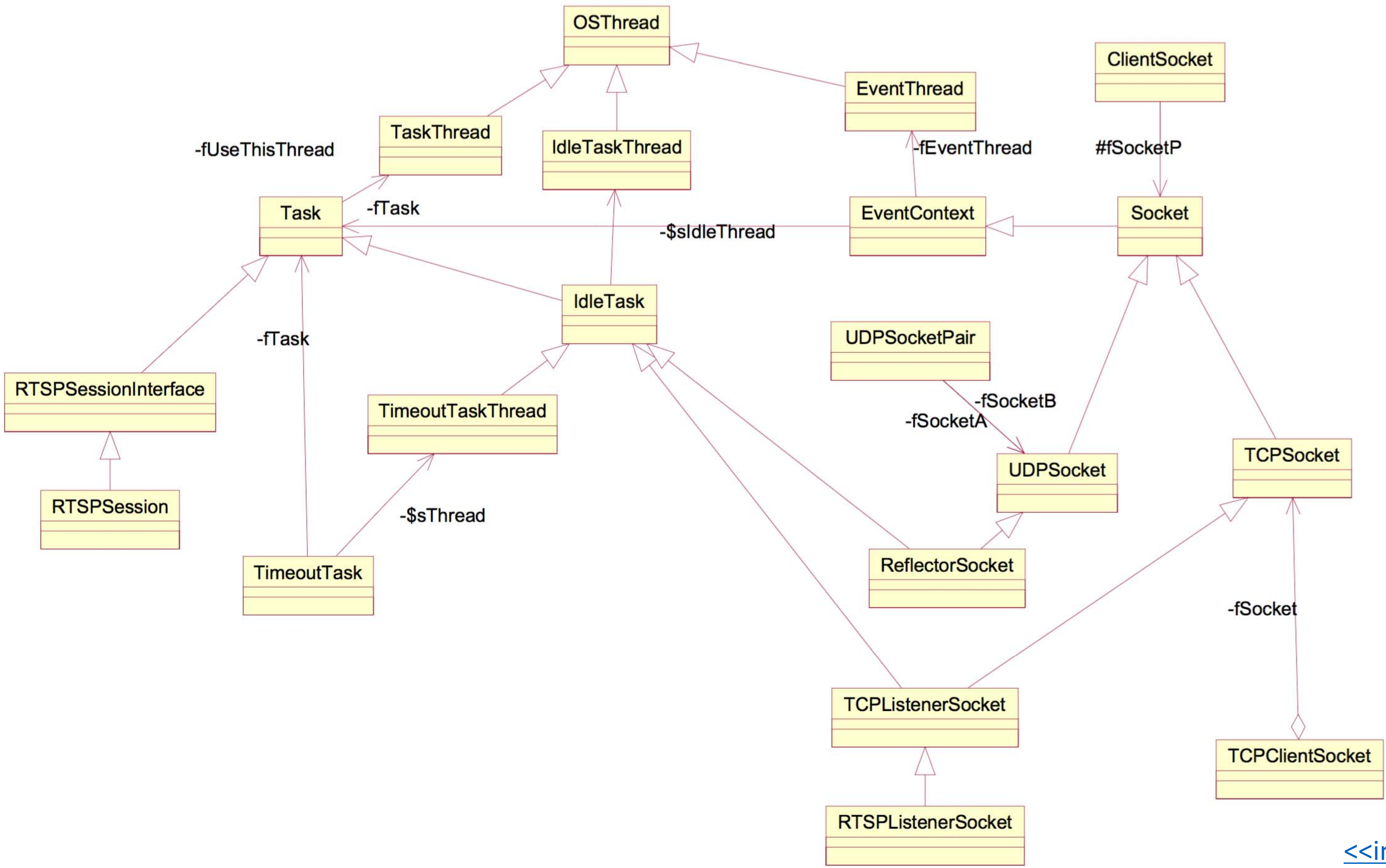


<<index>>

Task Class v15



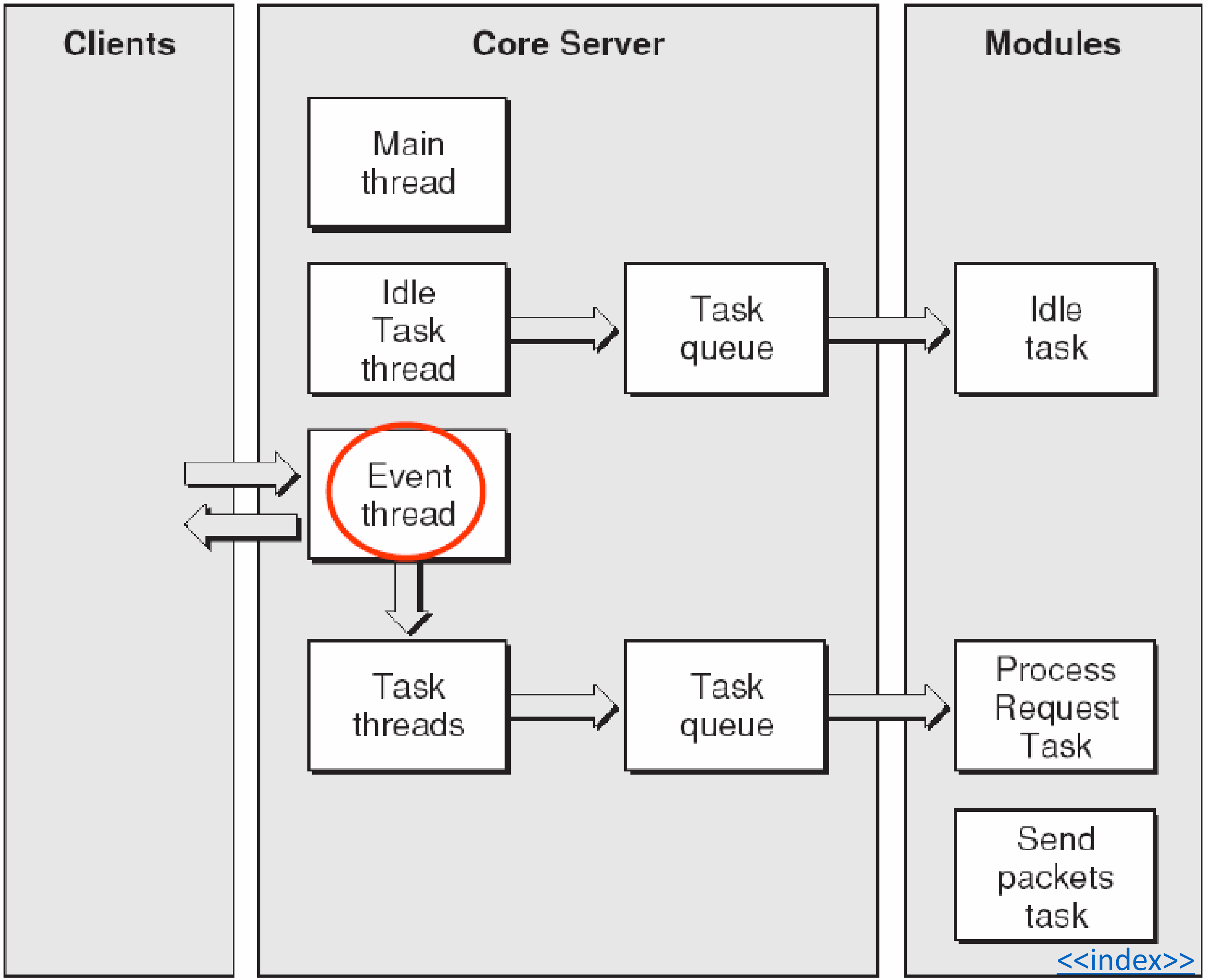
<<index>>


OSThread
ClientSocket
EventThread
TaskThread
IdleTaskThread
-fUseThisThread
-fEventThread
#fSocketP
Task
-fTask
-$sIdleThread
EventContext
Socket
IdleTask
UDPSocketPair
-fTask
RTSPSessionInterface
TimeoutTaskThread
-fSocketB
-fSocketA
UDPSocket
TCPSocket
RTSPSession
-$sThread
ReflectorSocket
TimeoutTask
-fSocket
TCPListenerSocket
TCPClientSocket
RTSPListenerSocket

<<index>>

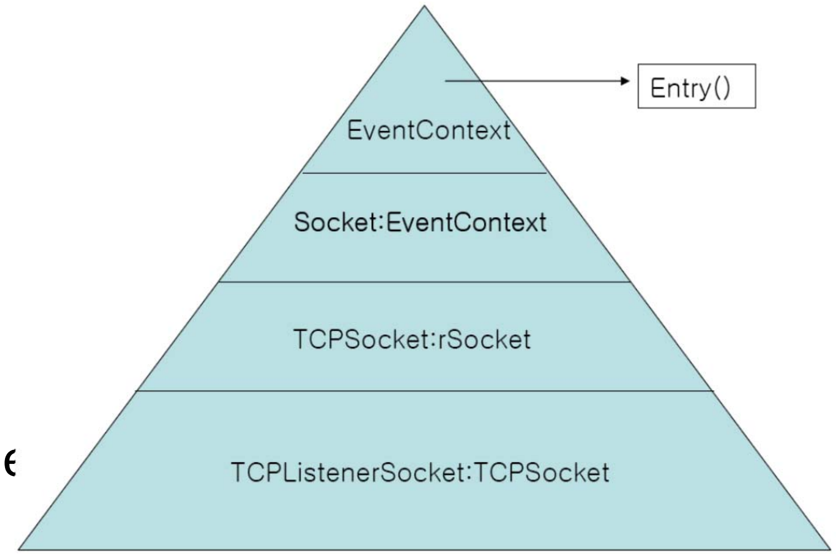
DSS事件线程

- 事件线程（Event Thread）。事件线程负责侦听socket事件，比如收到RTSP请求和RTP数据包，然后把事件传递给任务线程。



<<index>>

类层次



- EventContext
  - Event context提供了一种智能性，从UNIX file de
    EV_WR)获取事件，并传信一个任务。
- Socket
  - 封装了Socket操作及属性，如地址，端口，及其对socket的操作
- TCPSocket
  - TCP Socket的基本封装
- TCPListenerSocket
  - TCPListenerSocket用于监听TCP端口，当一个新连接请求到达后，该类将赋予这个新连接一个Socket对象和一个Task对象的配对。

Event Type

```
//EVENTS, CommonUtilitiesLib/Task.h
//here are all the events that can be sent to a task
 enum    {
     kKillEvent =    0x1 << 0x0, //Are all of type "EventFlags"
     kIdleEvent =   0x1 << 0x1,
     kStartEvent =   0x1 << 0x2,
     kTimeoutEvent = 0x1 << 0x3,

       //socket events
     kReadEvent =0x1 << 0x4, //All of type "EventFlags"
     kWriteEvent =   0x1 << 0x5,

        //update event
     kUpdateEvent =  0x1 << 0x6
};
```

EventThread

```
class EventContext   {
    ...
    virtual void ProcessEvent(int evtbits)  {
        fTask->Signal(Task::kReadEvent);
    }
    struct eventreq fEventReq; //事件请求结构
    OSRef   fRef;    //该Context的引用，并添加到fEventThread->fRefTable
    PointerSizedInt fUniqueID; //标识该Context的唯一ID,该变量的值通过
      EventContext::RequestEvent(int theMask)获得,大小为WM_User+1
    StrPtrLen      fUniqueIDStr; //标识该Context的字符串ID,该变量和fUniqueID的
      关系非常微妙,变量的Ptr只是取fUniqueID的地址,并强行转换,所以其值问题和fUniqueID一样
    EventThread*    fEventThread; //Context对应的事件线程
    Bool16  fWatchEventCalled;    //状态变量,初始为False,当为False时,需
      则需执行申请fUniqueID, 增加引用,并调用select_watchevent(...); 如为TRUE,则直接
      调用select_modwatch(...),而不申请fUniqueID

    Task*   fTask;    //希望被传信的任务
}
```

```
class EventThread : public OSThread   {
  public:
    EventThread() : OSThread() {}
    virtual ~EventThread() {}
  private:
    virtual void Entry();
    OSRefTable      fRefTable;
    friend class EventContext;
};
```

```
class OSThread
{
    void Start();
};
```

Thread Start & Entry

```
oid OSThread::Start()  {
    unsigned int theId = 0;
    fThreadID = (HANDLE)_beginthreadex(     NULL, // Inherit security    0, // Inherit stack size
                _Entry, // Entry function
                (void*)this,    // Entry arg                    0,      // Begin executing immediately
                &theId );
}
```

```
unsigned int WINAPI OSThread::_Entry(LPVOID inThread)   {
    OSThread* theThread = (OSThread*)inThread;
    BOOL theErr = ::TlsSetValue(sThreadStorageIndex, theThread);  //设置存贮索引，该线程的数据将存到设置的存贮索引里
     theThread->Entry();     //纯虚函数,所以调用的是子类的Entry()函数
    return NULL;
}
```

EventThread Start & Entry()

```
void EventThread::Entry()  { //该结构定义在ev.h中，记录Socket描述符和在该描述符上发生的事件
  struct eventreq theCurrentEvent;    ::memset( &theCurrentEvent, '\0', ... );
  while (true) {        //调用select_waitevent函数监听所有的Socket端口，直到有事件发生为止
    while ( theReturnValue == EINTR)
        theReturnValue = select_waitevent(&theCurrentEvent, NULL);
    //socket上有数据等待，发送wakeup，唤醒相应的Socket端口
    if (theCurrentEvent.er_data != NULL)        {//即获得消息的消息体theMessage.message, 即socketID
     //事件的cookie是一个对象名字，该对象已经在引用表中，解析为指针, 通过事件标识找到对象参考指针
      StrPtrLen idStr((char*)&theCurrentEvent.er_data, sizeof(theCurrentEvent.er_data));
      OSRef* ref = fRefTable.Resolve(&idStr);
      EventContext* theContext = (EventContext*)ref->GetObject();
      theContext->ProcessEvent(theCurrentEvent.er_eventbits);  //件在默认时只生成了读事件的命令，即fTask-Signal(Task::kReadEvent);
      fRefTable.Release(ref);        //减少引用,但并不删除任务, 与此对应的是,获得一个事件时,增加引用计数
    } //if      }//while
}
```

```
class Socket : public EventContext{
static void Initialize()
    { sEventThread = new
    EventThread(); }
static void StartThread()
    { sEventThread->Start(); }
static EventThread* GetEventThread()
    { return sEventThread; }
static EventThread* sEventThread;
}
```

ProcessEvent(...)

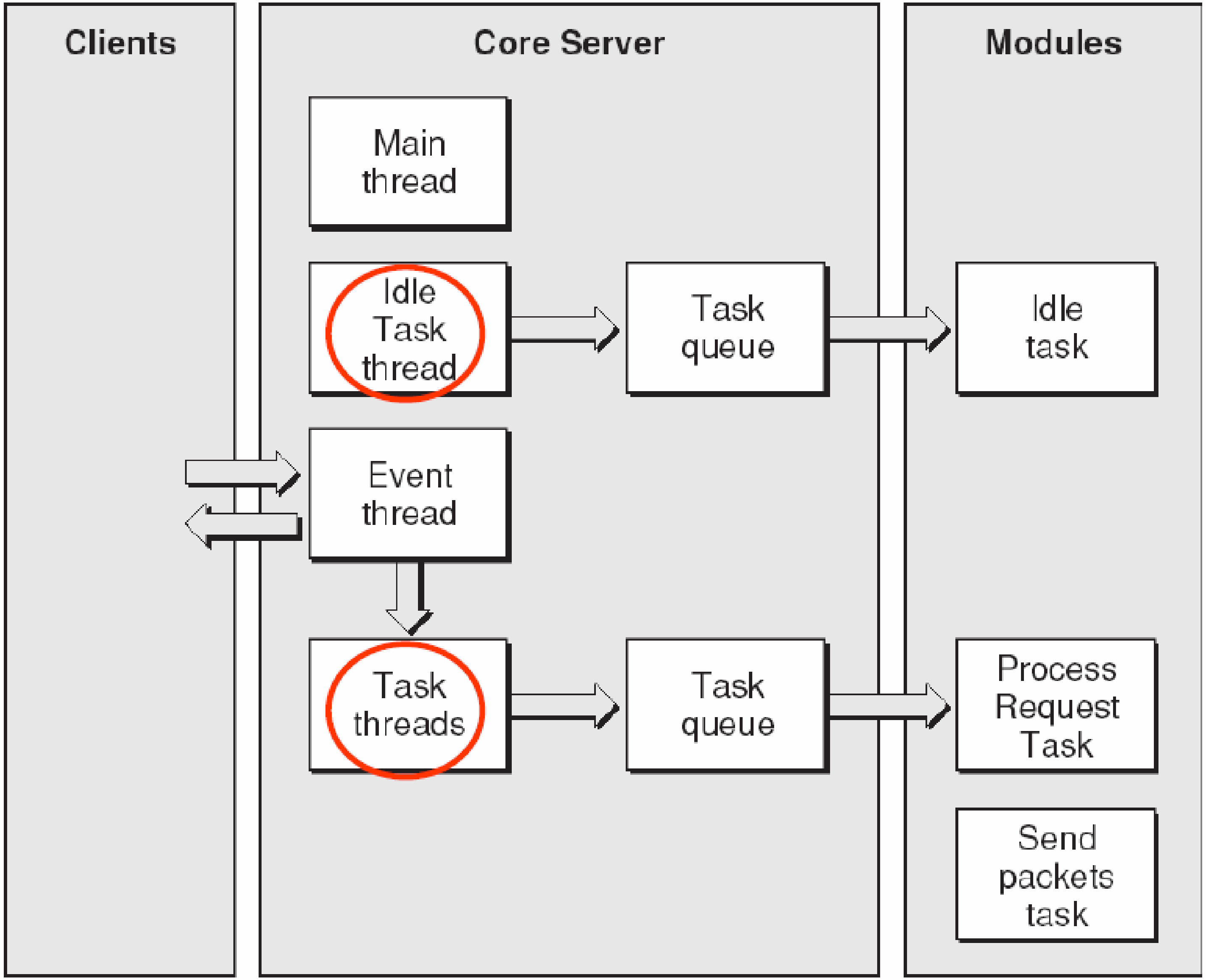
```
class EventContext {
    protected:
        virtual void ProcessEvent(int /*eventBits*/)
        {
            if (fTask != NULL)
            fTask->Signal(Task::kReadEvent);
        }
    private :
      Task* fTask;
}
```

Signaling to Task

```
void Task::Signal(EventFlags events)  {
   events  |= kAlive;               EventFlags oldEvents = atomic_or(&fEvents, events);
   if ((!(oldEvents & kAlive)) && (TaskThreadPool::sNumTaskThreads > 0))       {
           unsigned int theThread = atomic_add(&sThreadPicker, 1);
           theThread %= TaskThreadPool::sNumTaskThreads;
           TaskThreadPool::sTaskThreadArr[theThread]->fTaskQueue.EnQueue(&fTaskQueueElem);
   }   else
      qtss_printf("Task::Signal sent to dead Task");
}
```

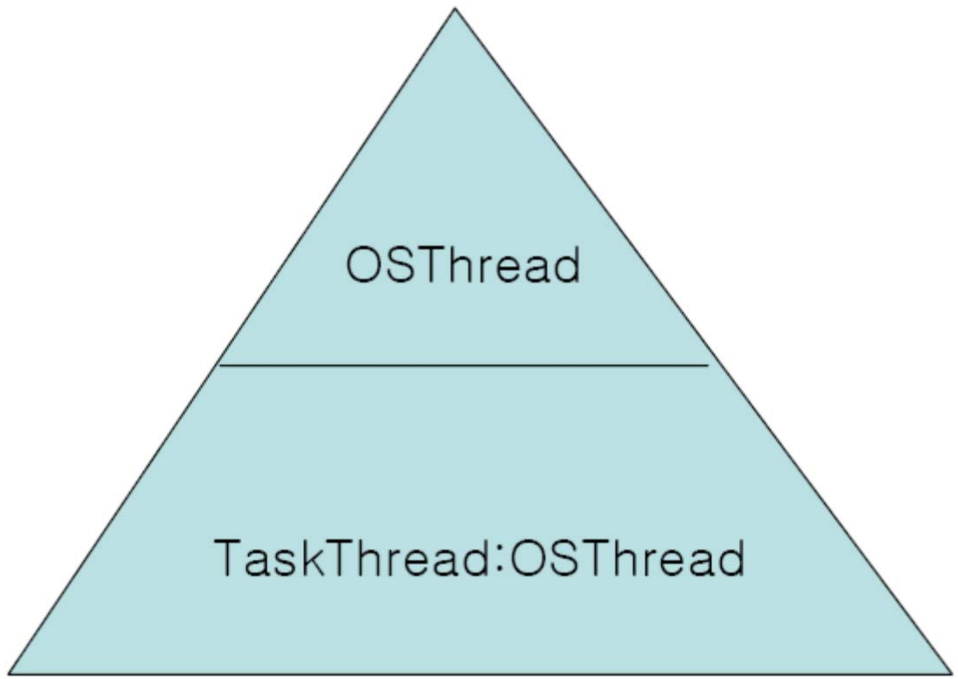
DSS任务线程

- 空闲任务线程（Idle Task Thread）。空闲任务线程管理一个周期性的任务队列。该任务队列有两种类型：超时任务和套接口任务。



<<index>>

Task类的层次结构



Task线程初始化

```
Bool16 TaskThreadPool::AddThreads(UInt32 numToAdd)  {
    sTaskThreadArray = new TaskThread*[numToAdd];
    for (UInt32 x = 0; x < numToAdd; x++)     {
        sTaskThreadArray[x] = NEW TaskThread();
        sTaskThreadArray[x]->Start();
    }
    sNumTaskThreads = numToAdd;
    return true;
}
```

```
class OSThread
{
    void Start();
};
```

Thread Start & Entry

```
void TaskThread::Entry()  { /*以下函数等待并运行任务*/
  Task* theTask = NULL；   //空任务
  while (true){  //监测是否有需要执行的任务，如果有就返回该任务；否则阻塞；线程循环执行到结束
     theTask = this->WaitForTask()；
    while (!doneProcessingEvent){      //该循环一直运行到结束
       SInt64 theTimeout = 0；      //Task中Run函数的返回值，重要核心部分：运行任务，根据返回值判断任务进度
       theTimeout = theTask->Run()；  //运行任务，得到返回值
 //监测Task中Run()函数的返回值，共有三种情况
```

```
//监测Task中Run()函数的返回值，共有三种情况
    if (theTimeout < 0){//1、返回负数，表明任务已经完全结束，删除任务对象
      theTask->fTaskName[0] = 'D';  //任务标记为Dead
      delete theTask;   theTask = NULL;    doneProcessingEvent = true;   //删除Task对象
    }
    else if (theTimeout=0){//2、返回0，表明任务希望在下次传信时被再次立即执行
      if (doneProcessingEvent)       theTask = NULL;
    }
   else{   //3、返回正数，表明任务希望在等待theTimeout时间后再次执行
       theTask->fTimerHeapElem.SetValue(OS::Milliseconds() + theTimeout);
       fHeap.Insert(&theTask->fTimerHeapElem);
       doneProcessingEvent = true;
    }
    //此处略...
 } //while (!doneProcessingEvent)
} //外面的大循环while (true)
```

WaitForTask()

```
Task* TaskThread::WaitForTask()   {
  while (true) {
    SInt64 theCurrentTime = OS::Milliseconds();
   //如果堆有任务，已到执行时间，返回该任务。PeekMin函数见OSHeap.h
    if ((fHeap.PeekMin() != NULL) && (fHeap.PeekMin()->GetValue() <= theCurrentTime))
        return (Task*)fHeap.ExtractMin()->GetEnclosingObject();  //从堆中取出第一个任务返回

   //如果堆有任务，未到执行时间，计算需要等待的时间
    if (fHeap.PeekMin() != NULL)              //计算还需等待的时间
       theTimeout = fHeap.PeekMin()->GetValue() - theCurrentTime;
   //等待theTimeout时间后从堆中取出任务返回
    OSQueueElem* theElem = fTaskQueue.DeQueueBlocking(this, theTimeout);
    if (theElem != NULL)
       return (Task*)theElem->GetEnclosingObject();
  } //while(TRUE)
}
```

<<index>>

IdleTaskThread

主要是维护一个heap对象

```
class IdleTaskThread : private OSThread  {
private:
    IdleTaskThread() : OSThread(), fHeapMutex() {}
    virtual ~IdleTaskThread() { Assert(fIdleHeap.CurrentHeapSize() == 0); }

    void SetIdleTimer(IdleTask *idleObj, SInt64 msec);
    void CancelTimeout(IdleTask *idleObj);

    virtual void Entry();
    OSHeap  fIdleHeap;
    OSMutex fHeapMutex;        OSCond  fHeapCond;       friend class IdleTask;
};
```

<<index>>

```
void IdleTaskThread::SetIdleTimer(IdleTask *activeObj, SInt64 msec)  {
      fIdleHeap.Insert(&activeObj->fIdleElem);          fHeapCond.Signal();
}
Void  IdleTaskThread::Entry()  {
   while (true)      {   //if there are no events to process, block.
      if (fIdleHeap.CurrentHeapSize() == 0)    fHeapCond.Wait(&fHeapMutex);
      SInt64 msec = OS::Milliseconds();
      //pop elements out of the heap as long as their timeout time has arrived
      while ((fIdleHeap.CurrentHeapSize() > 0) && (fIdleHeap.PeekMin()->GetValue() <= msec)) {
         IdleTask* elem = (IdleTask*)fIdleHeap.ExtractMin()->GetEnclosingObject();
         elem->Signal(Task::kIdleEvent);
      }
      //we are done sending idle events. If there is a lowest tick count, then we need to sleep until that time.
      if (fIdleHeap.CurrentHeapSize() > 0)         {
         SInt64 timeoutTime = fIdleHeap.PeekMin()->GetValue();
         UInt32 smallTime = (UInt32)timeoutTime;
         fHeapCond.Wait(&fHeapMutex, timeoutTime );
      }    } }
```

<<index>>

<<index>>

其它1

<<index>>

其它2

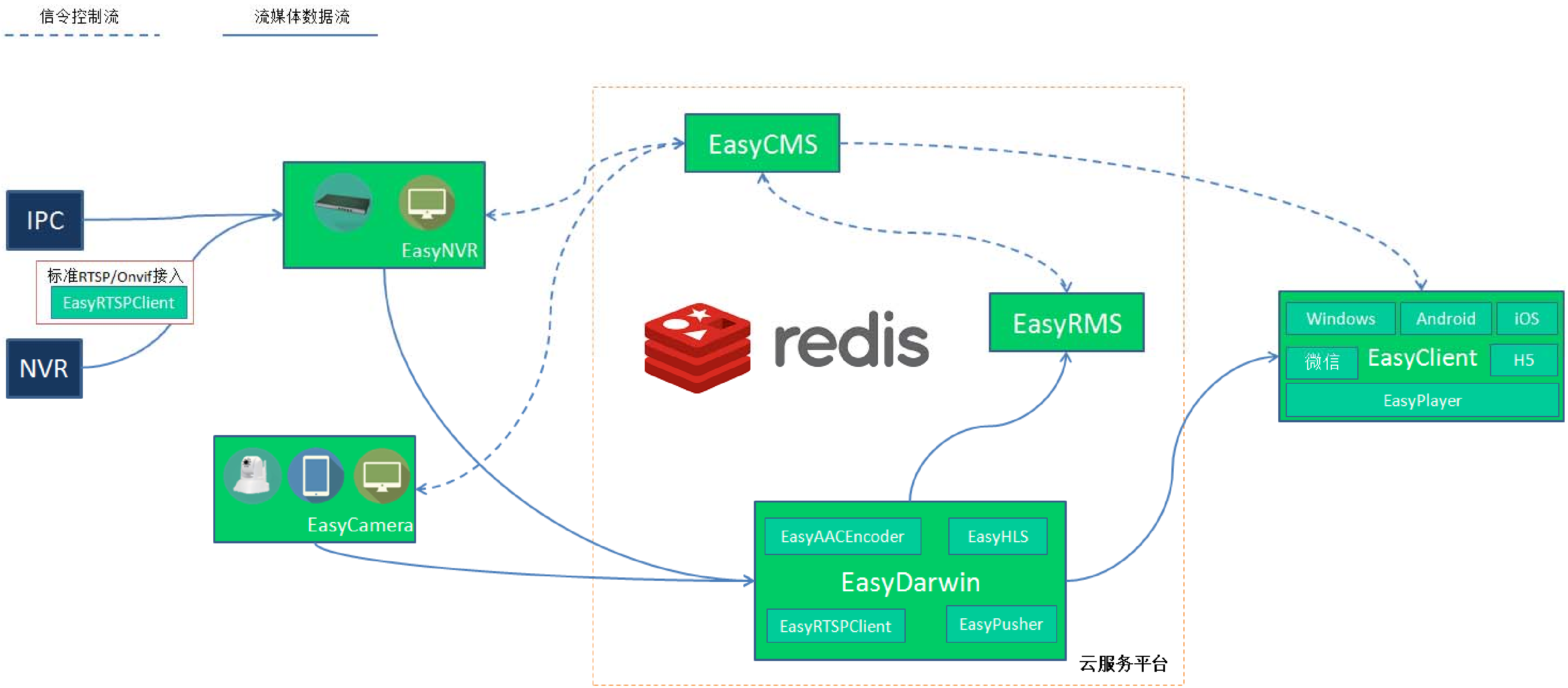
<<index>>

附录

- EasyDarwin
- Darwin代码剖析
  - 事件线程
  - 任务线程

<<index>>

EasyDarwin



<<index>>