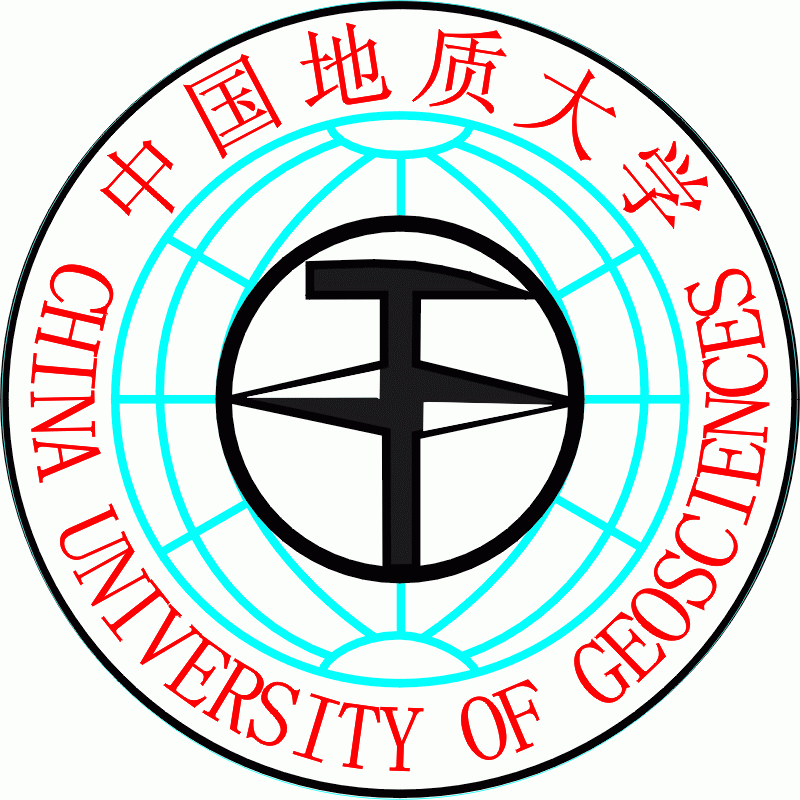
## 



《物联网安全技术课程设计》

前期规划

姓 名： 汪宇晖 班号： 193152

学 号：20151003737 组长： 汪宇晖

院（系）： 计算机学院 专业： 网络工程

指导教师： 姚 宏 职称： 副教授

2018 年 5 月

目录

[第一章 概要设计 1](#_Toc23028)

[1.1部署设计 1](#_Toc14304)

[1.1.1环境部署 1](#_Toc14933)

[1.1.2交互设置 1](#_Toc30710)

[1.1.3中间流程展示 2](#_Toc25347)

[1.2数据结构 2](#_Toc7486)

[1.2.1验证包数据结构 2](#_Toc1757)

[1.2.2聊天通信包数据结构 3](#_Toc8638)

[1.2.3聊天服务器数据结构 3](#_Toc10189)

[1.3多线程socket通信 4](#_Toc14261)

[1.4 log位置 5](#_Toc28464)

[第二章 UI设计 7](#_Toc27354)

[2.1服务器UI设计 7](#_Toc19070)

[1.2客户端UI设计 8](#_Toc18646)

[2.3 UI触发方式 9](#_Toc30319)

[2.3.1客户端 9](#_Toc12918)

[2.3.1客户端 11](#_Toc13278)

[2.3 UI联系方式 11](#_Toc17241)

[2.3.1聊天室客户机联系方式 11](#_Toc32066)

[2.3.2 FTP客户机联系方式 12](#_Toc27691)

[2.3.3服务器联系方法 12](#_Toc2165)

[第二章 模块布置与任务分配 14](#_Toc31666)

[2.1状态机 14](#_Toc28)

[2.1.1状态机简介 14](#_Toc29998)

[2.1.2状态机设计 15](#_Toc30963)

[2.2加解密方式 18](#_Toc18149)

[2.2.1密钥生成 18](#_Toc2753)

[2.2.2应用场景 19](#_Toc18888)

[2.3 多线程通信 19](#_Toc29135)

[2.4 模块分配 20](#_Toc31642)

[第三章拟定设计 23](#_Toc1286)

[3.1数据格式 23](#_Toc12038)

[3.1.1 数据包首部 23](#_Toc28205)

[3.1.2密钥 26](#_Toc2420)

[3.1.3函数命名 26](#_Toc15576)

[3.1.4其他信息 27](#_Toc9393)

[3.2简要流程 28](#_Toc29189)

[3.2.1 时序图 28](#_Toc6815)

[3.2.2 Kerberos流程图 28](#_Toc20864)

[3.2.3客户端工作流程图 29](#_Toc13184)

[3.2.4 AS和TGS服务器工作流程图 30](#_Toc28546)

[3.2.5服务器工作流程图 30](#_Toc29608)

# 概要设计

## 1.1部署设计

### 1.1.1环境部署

（1）编译工具：eclipse

（2）编译语言：java

（3）操作系统：windows10，mac等

（4）编译器版本：jdk1.8

（5）源代码管理工具：github

### 1.1.2交互设置

各个客户机之间不直接关联，如图3-5所示，聊天通信中，C1发送消息，通过服务器，消息将发送至连接上的所有客户机C2，C3，C4；在文件传输中，客户端与服务器相对独立，但是当有客户机向文件传输文件时，另一个客户机可通过服务器查看相关信息。

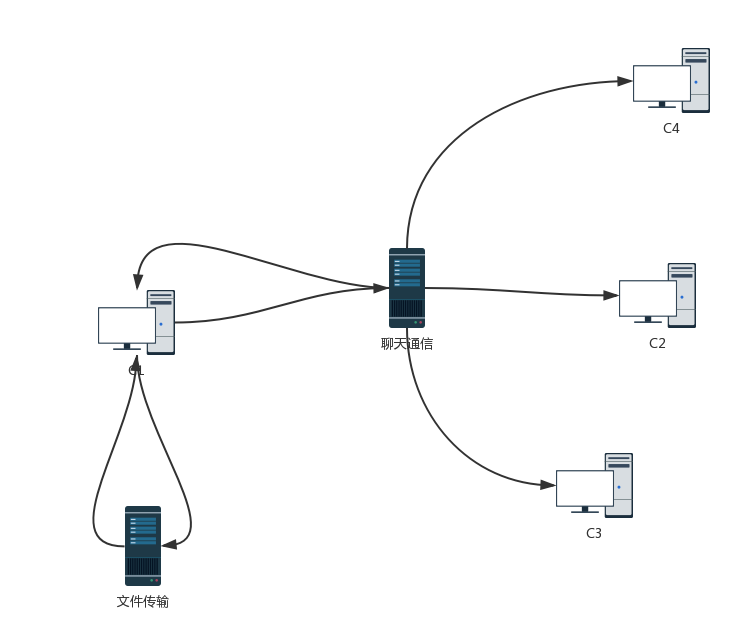


Figure 1‑1 客户机关系图

### 1.1.3中间流程展示

明文和密文的加解密过程是由服务器显示的，AS，TGS，V1，V2中的文本框就是显示发来消息和发送消息的明密文消息，详细情况看第一章UI设计图中文字注释。

## 1.2数据结构

### 1.2.1验证包数据结构

在通讯中，一开始预想使用的数据类型是用String代替一个包内所有的数据进行传输，包括了12位的包首部和每个包中不同的内容，但是这样做有几个不好的地方，首先在拆分中不方便拆分，使用分隔符的形式将字符串中的数据一个一个进行提取，这样做就需要设置一个分隔符，比如“||”，但如果把“||”作为分割符以为着“||”在加密的明文中是无法被使用的符号，否则直接导致整个拆包过程失败；其次在字符串的切分中需要约定一个顺序，因为一个字符串使用这样或那样的拆分方式后，是以一个字符串数组的形式去保存内步的数据的，message->message[n]，这个包内东西到底有多少，是以一个什么样的顺序进行一个存储，一旦在封包的时候顺序弄错，也会导致程序的失败；最后，有些包中会发送Ticket和Authenticator的数据，这两者的数据也是用“||”分割，会对正确拆分的结果造成干扰，所以要么重新约定两个分割符分别对Ticket、Authenticator和其他消息进行一个区分，否则只能用别的形式存储数据。对于以上三点，以一个分布式的程序来说，要检查错误十分麻烦，所以为了避免这样的错误，利用socket能发送对象的性质，直接发送相应的hashmap，发送的名称作为相应的key值，其中加密或者非加密的内容作为value，拆包方利用key来找到value就能避免产生如上的错误，key值的设置也比较简单，在验证过程中：

|  |  |
| --- | --- |
| **Key** | **Value** |
| Prelude | 包的首部12位标识 |
| IDc，IDv，IDtgs | 相应机器的id号 |
| TS | 时钟信息 |
| K(c,tgs) | c和tgs的Session-Key |
| Lifetime | 生存周期 |
| Authenticator | 验证票据信息 |
| K(c,v) | c和v的Session-key |
| AD | 防止非法使用 |
| Ticket | 相应票据（包括给tgs和v） |

Table 1-2 聊天通信包数据内容

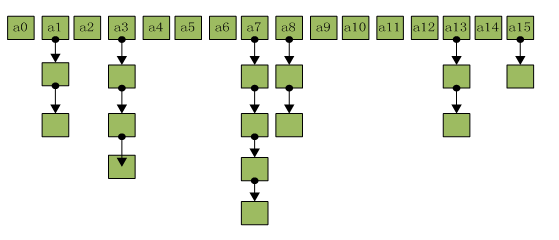


Figure 1‑2 HashMap数据结构

当然最终的过程中，Authenticator和Ticket还是会使用字符串分割的方式来解决，但是比较上面的方法，已经优化了不少。根据以上分析，自然而然使用的是hashmap的数据结构，其数据结构如图1-2所示。

### 1.2.2聊天通信包数据结构

通信包的数据结构按照验证包的思路，可以使用hashmap也可以使用自定义对象，因为首部的关系，还是使用hashmap的形式去发送传输相应数据：

|  |  |
| --- | --- |
| **Key** | **Value** |
| Prelude | 包的首部12位标识 |
| Message | 消息 |
| Times | 实现 |
| ID | 客户机ID |

Table 1-2 聊天通信包数据内容

### 1.2.3聊天服务器数据结构

在聊天服务器中，服务器需要把数据进行一个缓存，并将所有数据发送至相应的客户机，同时在离线的客户机在登陆上服务器时，服务器也能对消息进行一个处理，然后将未收到的消息转发给相应的客户机.由于没有大量的增删改，而且服务器的每个线程的操作相当于一个压栈出栈的过程，所以数据的存储的方式是链表的形式，对于线程的安全性，使用的是vector进行相应的数据存储，相应的处理方式如下所示：

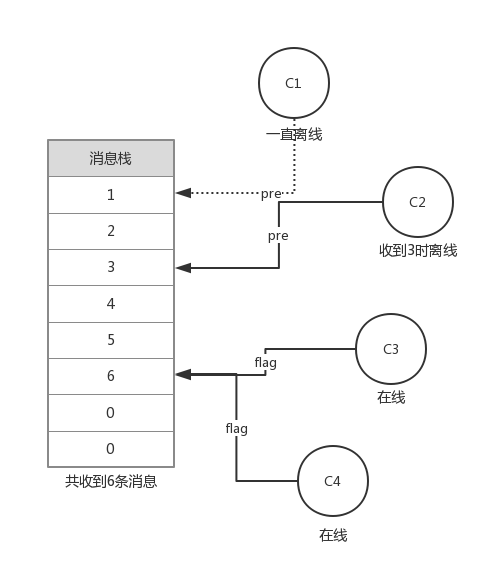


Figure 1-3 聊天栈数据内容

|  |
| --- |
| Time，Xxx：蓝啦烦请无法 |
| Time，Xxx：啊啊啊跟任何人 |
| Time，Yyy：个大傻逼大傻逼是 |
| Time，Zzz：啊恐惧啊擦 |
| Time，Vvv：i女i但是v |
| Time，Vvv：于遁迹空门 |
|  |
|  |

Table 1-3 聊天服务器数据内容

表1-3所示，有如上内容，相当于一个栈，后面更新的数据不断发送进来，服务器根据一个标识符pre标识每个用户已经读取到哪里的数据了，然后标记，如果后面有数据就将数据发送至相应客户机，如果客户机在通信之前都没上线，pre=0，如果上线后又下线了，再上线时，服务器会一直保留当前的pre，并随着flag的下移，发送相应消息，pre就相当于一个指针，具体情况如图1-3所示。

## 1.3多线程socket通信

如图1-4所示为多线程通信的框架，除了客户机以外，每个服务器都是用多个线程监听本线程要处理的客户机状态。

客户机1正在与AS中的线程1交互得到相应的验证信息。

客户机2正在与TGS中的线程2交互得到相应的验证信息。

客户机3正在与聊天室服务器中的线程3交互得到相应的通讯信息。

客户机4正在与FTP服务器中的线程4交互得到相应的文件命令以及传输情况。

同时客户机1对应所有服务器中的线程1，客户机2对应所有服务器中的线程2，依此类推。

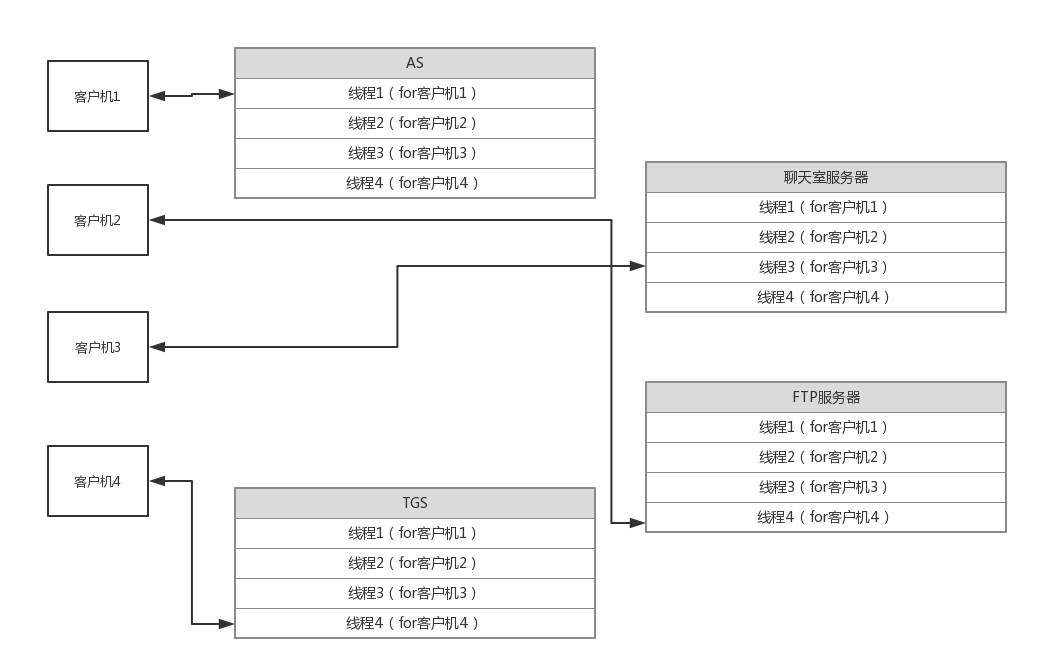


Figure 1-4 多线程通信框架

## 1.4 log位置

在分布式系统中，由于程序实现并不在一台机器中，所以需要用类似print的形式将错误的结果输出，所以需要将log设置在相应的位置。

在加密的时候使用log，主要是防止在加密的过程中，因为函数或者其他的原因产生的加密结果出现问题

加密->log:加密结果&&正确解密结果，这样就如同中间过程一样，为了和相应的中间票据服务器做一个类比，能快速知道是传输的问题还是函数的问题。

在接受中间服务器的数据时，对数据进行一个log输出，当数据出现错误的时能及时显示，至于拆包的过程可以在验证时候做一个print输出，可以更直观看出包的状态和内容。

总而言之，并行的麻烦之处就在于其通信的地方，在收包和发包的时候使用log打印包的相关问题就能知道哪方面的问题。

# 第二章 UI设计

## 2.1服务器UI设计

关于服务器的UI设计，主要是为了方便功能上的简单操作，由于我们预计要做的工作是文件传输和聊天交流，所以需要有两个服务器提供相关的服务，但是服务器上应该显示的是客户端名称，IP地址，交流的信息，所以这方面，两个服务器的UI设计应该相同或类似，预计效果如图1-1和图1-2所示（服务器的UI都类似，这里举两例FTP和AS）：



介绍各个客户端往来的信息，比如：

传输文件：

C1->”文件名”->“文件内容（密文显示）”

Figure 2‑1 ftp服务器UI设计



介绍各个客户端往来的信息，比如：

(K(c,tgs)||ID(tgs)||TS(2)||LifeTime2||Ticket(tgs))（明文）

(K(c,tgs)||ID(tgs)||TS(2)||LifeTime2||Ticket(tgs))（密文）

（空格）

Figure 2‑2 AS服务器 UI设计

## 1.2客户端UI设计

与服务器不同，客户端主要反应功能的实现，以及指引用户如何进行相应的操作，由于服务器的基本功能有传输文件和聊天交流，所以客户端的设计如下图1-3和图1-4所示：



聊天室的服务器对应只有一个，这里可以进行一个默认的设置

端口号设想是一个确定的端口号，无法修改，显示客户端socket通信的端口号

显示所有用户聊天的信息。

内容如：

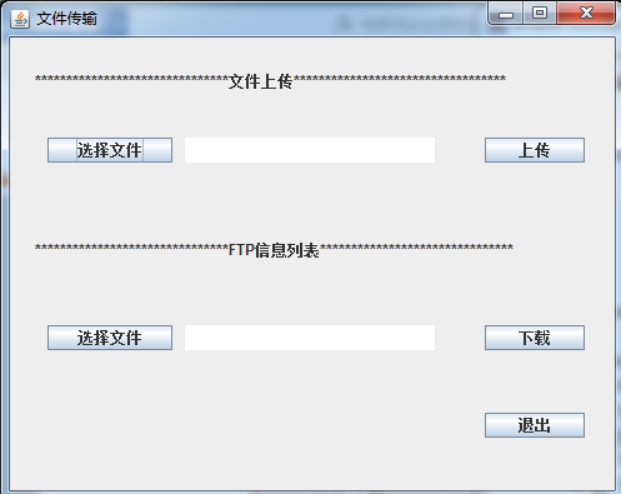
Xxx：xxxx

显示及在线用户，如：

Xxx

Yyy

Figure 2‑3 聊天室UI设计



由于这里的UI设置的比较具体，流程相应简单，选择文件后显示目录文件，点击其中一个进行相应操作即可

Figure 2‑4 客户端文件传输UI设计

## 2.3 UI触发方式

### 2.3.1客户端

1、“连接”按钮相关数据包

“连接”按钮上需要采用监听器，这里使用通过ActionListener来响应用户点击按钮的操作。

在UI设计中，“连接”按钮命名为button1，于是我们要针对button1来进行监听：

SimpleListener ourListener = new SimpleListener();//这里是创建了一个监听器对象

button1.addActionListener(ourListener);//将监听器绑定在“连接”按钮上

private class SimpleListener implements ActionListener

{

public void actionPerformed(ActionEvent e)

{

if (e.getSource() ==button1)

{

//在这里添加连接服务器的方法

//客户端希望与服务器建立连接，首先要主动访问AS

//实际操作中采用socket实现，send表示发包，receive表示收包，第一 个参数为目标服务器，第二个参数为包的内容

Send(AS,”ID(c)||ID(tgs)||TS1”);

//访问完毕后期待接受AS的数据

Receive(AS,”EK(c)[K(c,tgs)||ID(tgs)||TS2||Lifetime2||Ticket(tgs)]”)

//连接到TGS

Disassembly(”EK(c)[K(c,tgs)||ID(tgs)||TS2||Lifetime2||Ticket(tgs)]”);//拆包

Send(TGS,”ID(v)||Ticket(tgs)||Authenticator ”);

//收到TGS回复

Receive(TGS,”EK(c,tgs)[K(c,v)||ID(v)||TS4||Ticket(v)] ”);

Disassembly(”EK(c,tgs)[K(c,v)||ID(v)||TS4||Ticket(v)] ”);//拆包

//正式访问服务器

Send(V,”Ticket(v)||Authenticatorc”);

//收到服务器的确认

Receive(TGS,”K(c,v)||ID(c)||AD(c)||ID(v)||TS4||Lifetime(4) ”);

Disassembly(”K(c,v)||ID(c)||AD(c)||ID(v)||TS4||Lifetime(4) ”);//拆包

}

}

2、聊天室“发送”按钮和文本框相关数据包

Button2.addActionListener(ourListener);//将监听器绑定在“发送”按钮上

If(已经连接)

for()

{

//正式开始数据交换

Send(“Session-Key（时间||用户||信息）”);

Receive(“Session-Key（时间||用户||信息）”);

Disassembly(“Session-Key（时间||用户||信息）”);//拆包

//同时将收到的信息显示在文本框中

……;

}

else//未连接就发送会抛出异常

Error();

3、“文件上传”按钮相关数据包

点击“上传”按钮后，会弹出文件选择框，在这里触发器需要与该按钮绑定并在出发其中设置文件管理插件

If(已经连接)

for()

{

Send(“Session-Key（时间||用户||文件名）”);

Receive(“Session-Key（时间||用户||ACK）”);

Disassembly(“Session-Key（时间||用户||ACK）”);//拆包

for()//依据文件的大小可能需要分段传输

{

Send(“Session-Key（时间||用户||文件内容）”);

Receive(“Session-Key（时间||用户||文件内容）”);

……;

)

}

else//未连接就发送会抛出异常

Error();

1. “文件下载”按钮相关数据包

If(已经连接)

for()

{

Send(“Session-Key（时间||用户||文件名）”);

Receive(“Session-Key（时间||用户||ACK）”);

Disassembly(“Session-Key（时间||用户||ACK）”);//拆包

for()//依据文件的大小可能需要分段传输

{

Send(“Session-Key（时间||用户||文件内容）”);

Receive(“Session-Key（时间||用户||文件内容）”);

Disassembly(“Session-Key（时间||用户||文件内容）”);//拆包

……;

)

}

else//未连接就发送会抛出异常

Error();

### 2.3.1客户端

1. AS相关数据包

AS的UI只有文本框，用于显示包的信息。这里不需要使用触发器，只用将包的信息打印在屏幕上。

Receive(C,”ID(c)||ID(tgs)||TS1”);

Disassembly(”ID(c)||ID(tgs)||TS1”);//拆包

Print();

Send(AS,”EK(c)[K(c,tgs)||ID(tgs)||TS2||Lifetime2||Ticket(tgs)]”);

Print();

1. TGS相关数据包

同样TGS的UI也只有文本框，除了收发数据包不同，其余操作几乎一样

Receive(C,”ID(v)||Ticket(tgs)||Authenticator”);

Disassembly(”ID(v)||Ticket(tgs)||Authenticator”);//拆包

If()//判断票的真伪，票有问题就抛出异常

Print();

Send(AS,”EK(c,tgs)[K(c,v)||ID(v)||TS4||Ticket(v)]”);

Print();

1. FTP服务器相关数据包

FTP服务器只有文本框与记录文件上传下载状态，将所有包中的信息进行处理打印下来就可以了。

1. 聊天室服务器相关数据包

聊天室的情况与FTP类似，记录每个用户发送过来的信息，以time：xxx：劈哩叭啦，就是将每次进栈的数据输出出来。

## 2.3 UI联系方式

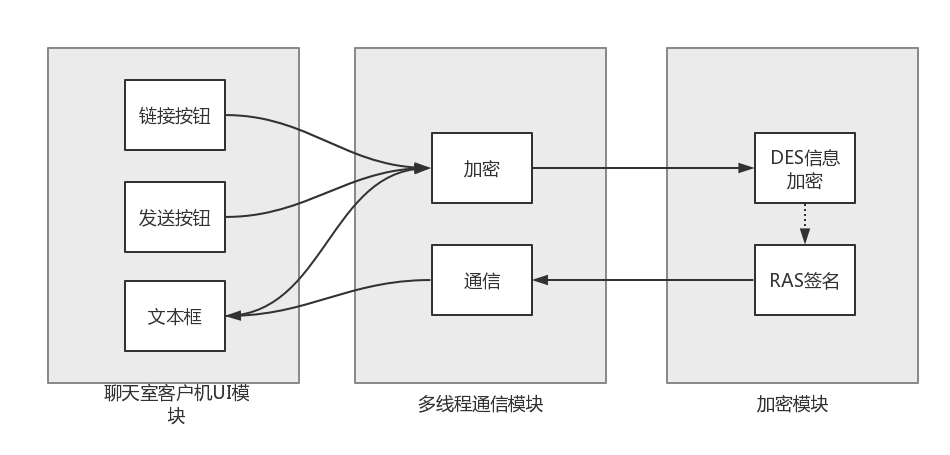
2.3.1聊天室客户机联系方式  


Figure 2‑5 聊天室客户机模块连接

2.3.2 FTP客户机联系方式

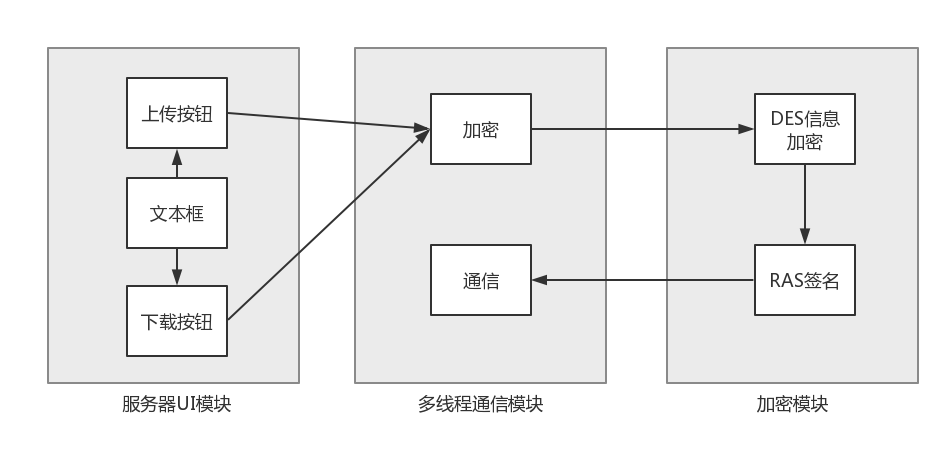


Figure 2‑6 FTP客户机模块连接

### 2.3.3服务器联系方法

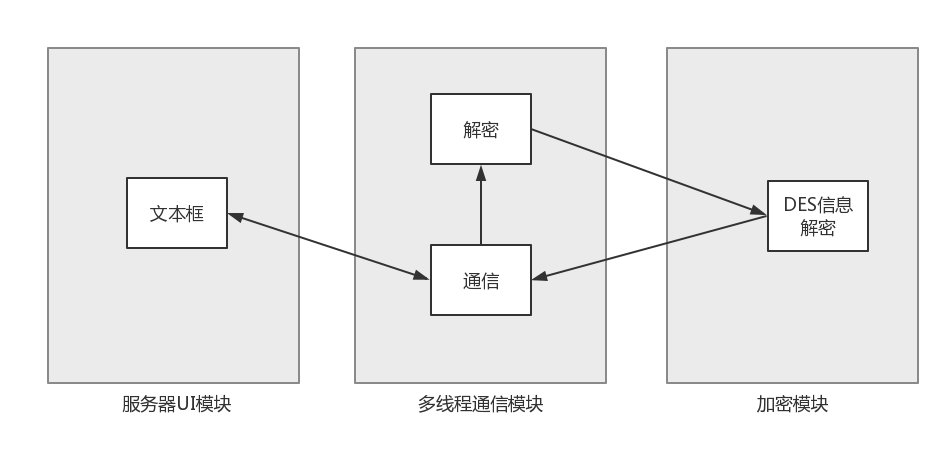


Figure 2‑7 服务器模块连接

# 第二章 模块布置与任务分配

## 2.1状态机

### 2.1.1状态机简介

有限状态机这一概念是从硬件领域衍生到软件领域的。有限状态机是描述有限个状态以及在这些状态之间的转移和动作等行为的数学模型。有限状态机通常由以下4个方面组成:现态，条件，动作，次态。在硬件领域中，通过不同的输入（0/1）来进行不同的操作，在满足条件的前提下，不同的操作又会将时序逻辑电路引导至不同的输出结果，这就是典型的动作引发的状态迁移（图2-1）。

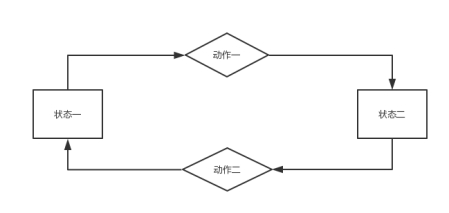


Figure 2‑0‑1 状态迁移图

如上图所示，一盏灯有两个状态，开和关。在关灯的状态下执行开灯的动作，灯就会从关灯的状态迁移至开灯的状态，同理，在开灯的情况下执行关灯的动作，灯的状态就会从开变为关。应当注意的是，动作只是引发状态迁移的引子，是否会发生状态变化则要看设计的机制。在逻辑电路中，对于一盏灯假设0为关的状态，1为开的状态，在状态为1的情况下，若再将状态置为1，灯也不会熄灭（因为灯灭的条件是0）。这种情况下，没有满足条件动作就是无效的，类似的无效动作在程序设计中是需要避免的。

每一个if判断就是一个状态机，它决定了整个程序的走向。往大了说，整个程序就是一个大型的状态机，通过点按不同的按钮，输入不同的数据等一连串的动作来实现不同的功能（达到不同的状态）。

根据tcp三次握手的流程图来详细讲解状态的迁移的细节。

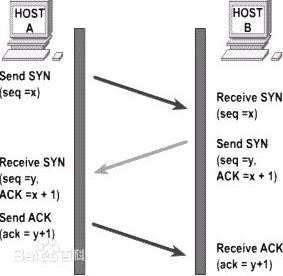


Figure 2-2 TCP握手流程

上图（图2-2）是三次握手的流程图，在主机A向主机B提交访问请求后，A的状态就从未连接转换到了监听，等待B的确认，此时B还是未连接的状态。在B收到连接请求之后，确认已经收到一个链接请求，此时B会向A回复一个SYN包，然后将自己的状态也切换至监听等待A的链接。A收到B回复的包，并确认包内ack编号无误（状态迁移的条件）后，状态从监听转换为连接，开始与B进行数据交换。

从细节来说，整个TCP三次握手的流程中，A状态的变化如下表（表2-1）：

|  |  |  |  |
| --- | --- | --- | --- |
| 次态  现态 | 未连接 | 监听 | 已连接 |
| 未连接 |  | 用户进行B的操作 |  |
| 监听 |  |  | 收到B的ack回复 |
| 已连接 |  |  |  |

Table 2-1 A的状态表

B的状态如下表所示（表2-2）

|  |  |  |  |
| --- | --- | --- | --- |
| 次态  现态 | 未连接 | 监听 | 已连接 |
| 未连接 |  | 收到A的SYN包 |  |
| 监听 |  |  | 收到A的ack回复 |
| 已连接 |  |  |  |

Table 2-2 B的状态表

以上就是基于TCP三次握手的状态机模型，TCP链接的过程中由于功能只有一个，所以状态的变化并不是很多，在实现Kerberos时，状态变化会十分频繁，但大致的思路不变，所以，在下面会详细描述关于Kerberos状态变化的设想。

### 2.1.2状态机设计

**FTP服务器状态机（如图2-3所示）**

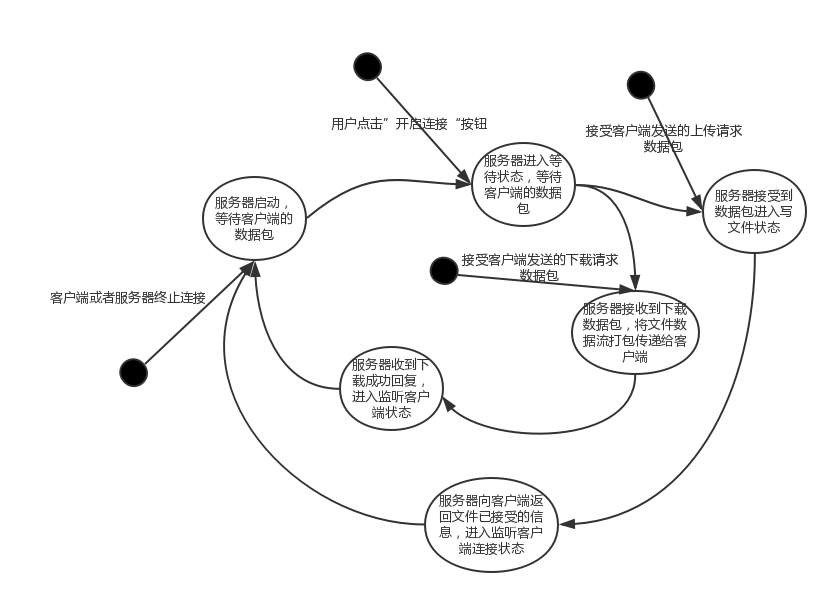


Figure 2‑3 服务器状态机

**聊天室服务器（如图2-4所示）**

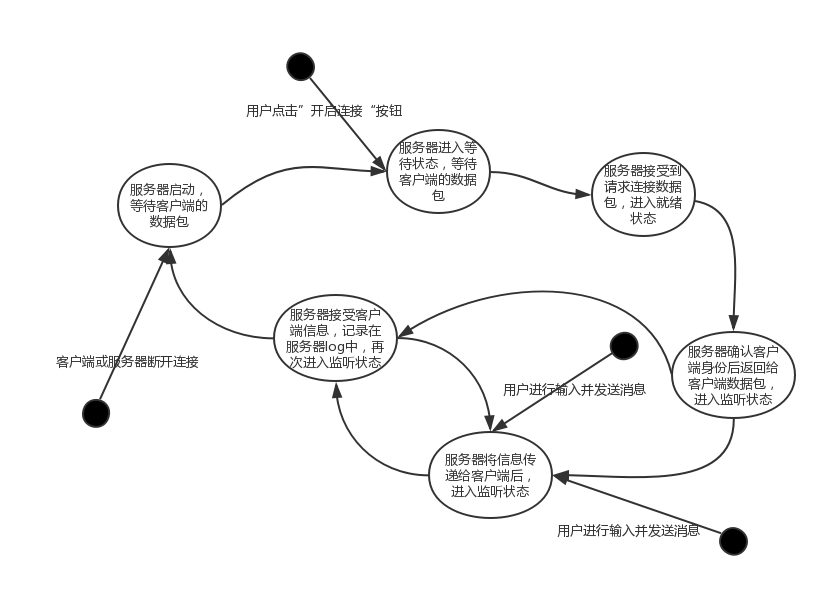


Figure 2‑4 客户端状态机

**聊天室客户端状态机（如图2-5所示）**

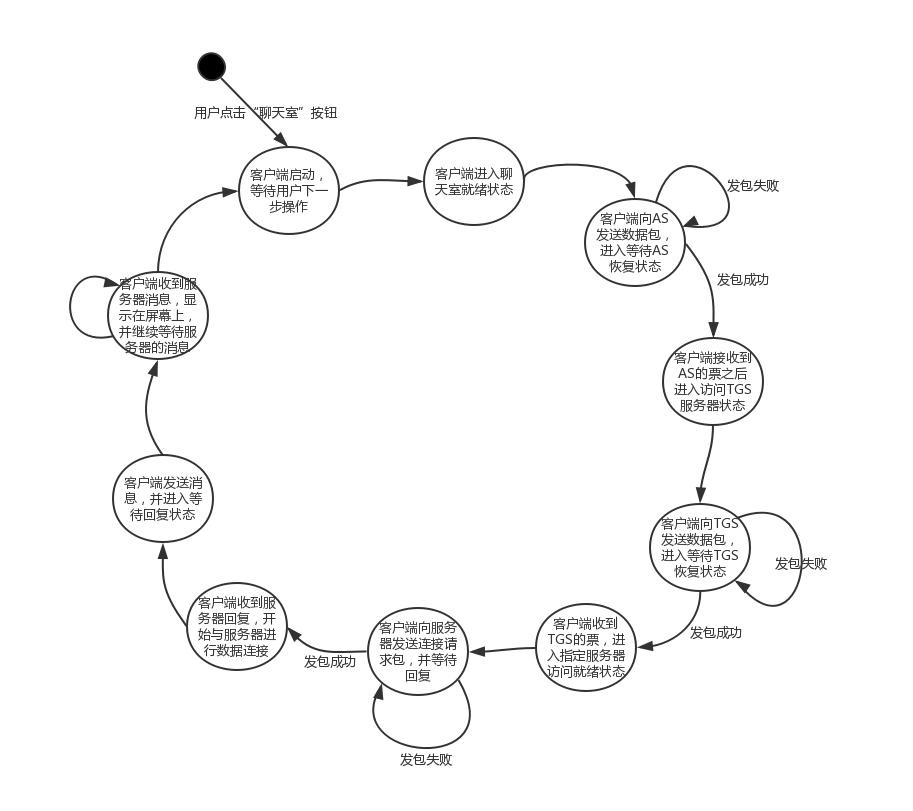


Figure 2‑5客户端状态机

**FTP客户端传输状态机（如图2-6所示）**

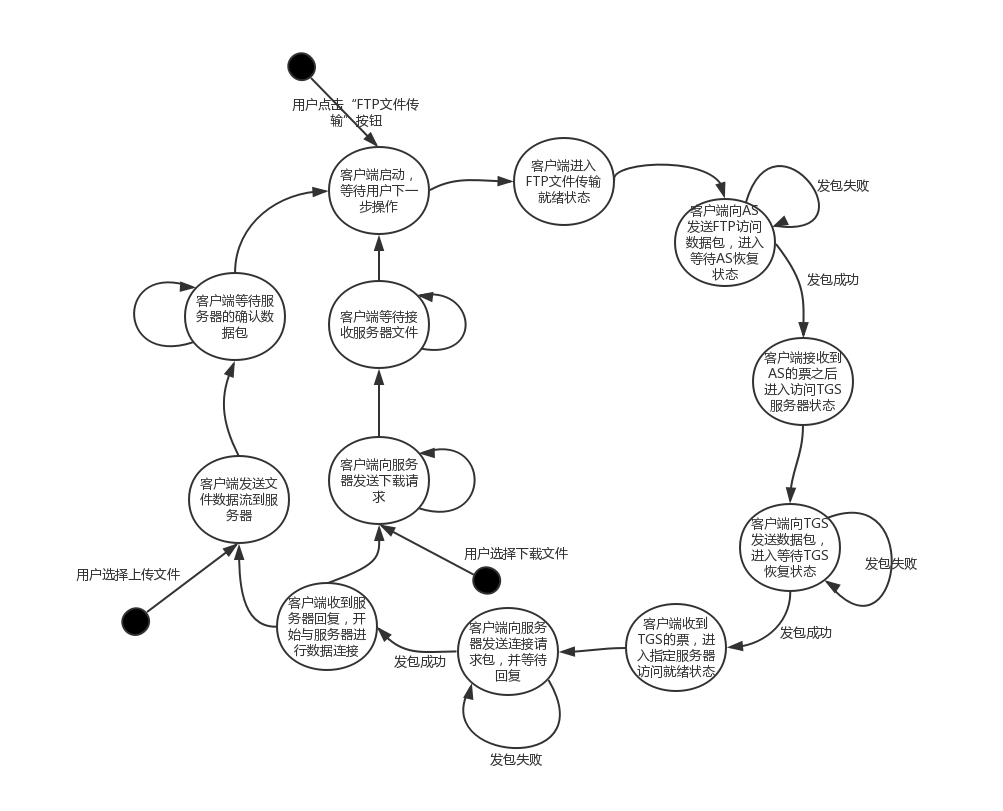
****

Figure 2‑6 客户端状态机

**AS与TGS服务器传输状态机（如图2-7所示）**

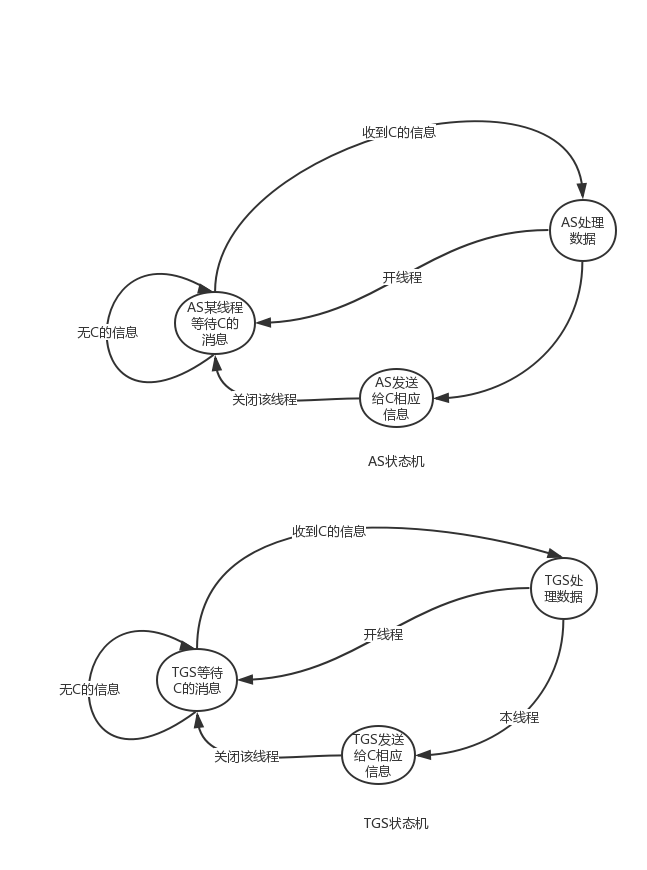
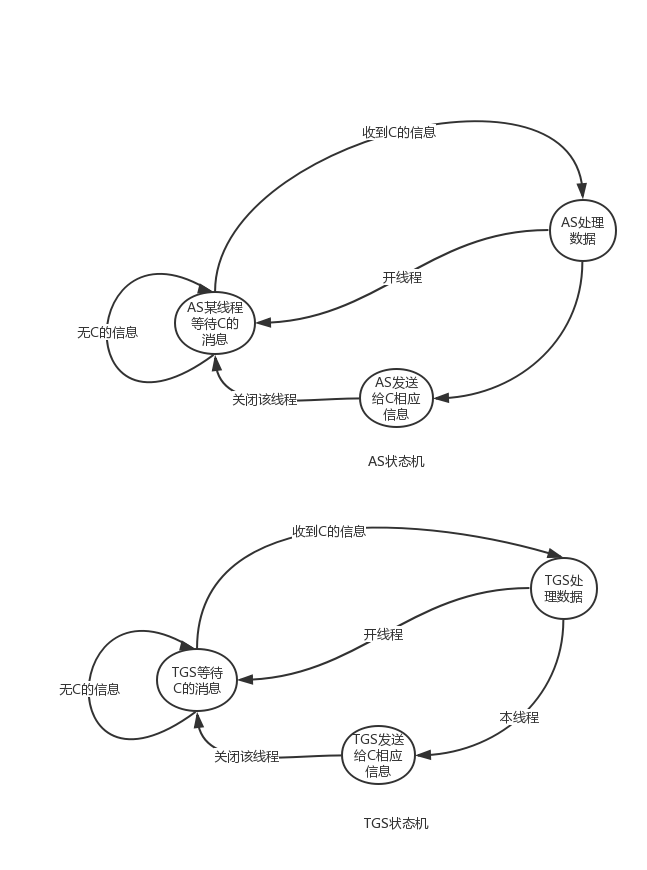


Figure 2‑7 AS与TGS状态机

## 2.2加解密方式

### 2.2.1密钥生成

根据Kerberos的理论，在通信中，需要两种加密方法，分别是DES堆成钥匙加密与RSA不对称钥加密，在Kerberos 中的Session-Key均为DES的对称密钥，Session-Key的密钥都可以利用Kerberos的流程，在AS和TGS的服务器中利用随机的数据生成一个Session-Key，但我在之前就注意到了，RSA算法的钥匙是在之前就约定好了的，否则，第一步中AS发送给Client的密文，Client无法解密，同样的道理，TGS发送给Client，其中Ticket(v)无法被V解密，思考到了这里，思路就非常清晰了，首先，AS与Client需要知道对方的RSA公钥，TGS和Server需要知道对方的公钥，因为公钥的传送过程不涉及到Kerberos的流程，所以在Kerberos中没有提及到，但是在实际的操作过程中，这都是需要知道的，可以用数据库的方式保存相应的钥匙，关于钥匙生成的过程如下图2-8所示

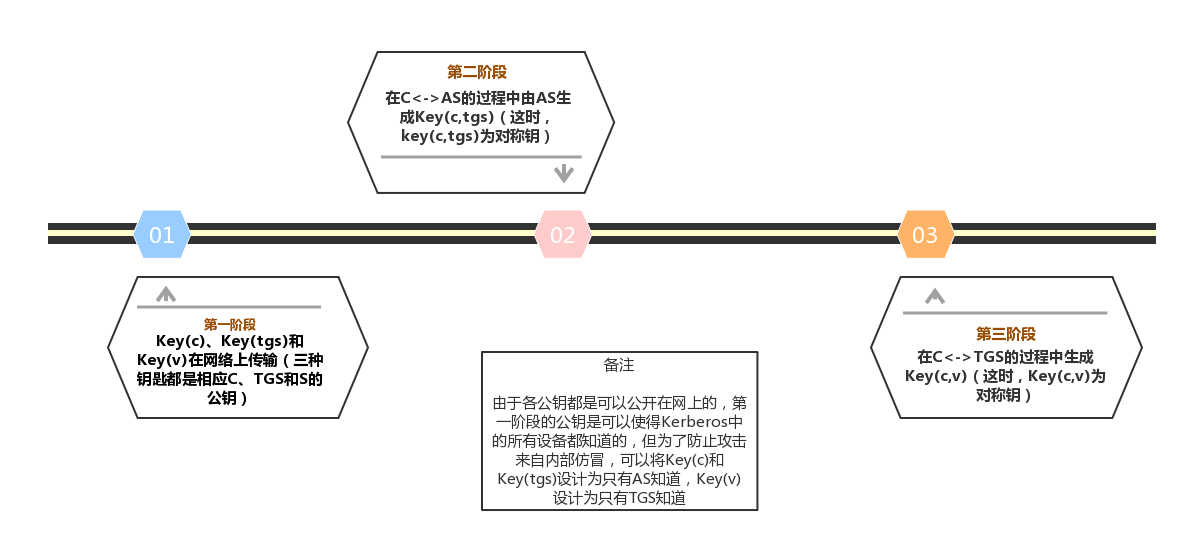


Figure 2‑8 密钥生成顺序

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 阶段 | 密钥 | 类型 | 方式 | 流程 |
| 第一阶段 | Key(c)  Key(tgs)  Key(v) | 不对称钥公钥 | DES | Key(c)、Key(tgs)和Key(v)在网络上传输（三种钥匙事先约定好） |
| 第二阶段 | Key(c,tgs) | 对称钥 | DES | 在C<->AS的过程中由AS生成Key(c,tgs)（这时，key(c,tgs)为对称钥） |
| 第三阶段 | Key(c,v) | 对称钥 | DES | 在C<->TGS的过程中生成Key(c,v)（这时，Key(c,v)为对称钥） |

Table 2-3 密钥生成

### 2.2.2应用场景

首先，需要实现得到3对不对称的密钥，这个密钥可以保存在数据库中，也可以直接定义在程序内部，由于这样的钥匙就如同证书一样，变化不会很大，变化很大的是Session-Key，所以Session-Key需要AS和TGS在程序中随机生成。

在实际运用中，利用不对称钥解密的场景主要由：C解读AS的信息、TGS解读C发送中的Ticket(tgs)的信息，以及V解读C发送的Ticket(v)的信息。

利用对称钥加密的场景主要有：TGS将重要信息发送给C，V将重要信息发送给C。

|  |  |  |  |
| --- | --- | --- | --- |
| **加密方式** | **加密主体** | **解密场景（顺序）** | **加密内容** |
| DES | AS | 1. C通过自己的私钥解密，得到与tgs的Session-Key； 2. TGS通过自己私钥得到了与C交互的Session-key； 3. V通过自己的私钥得到了与C交互的Session。 | 1. K(c,tgs)||ID(tgs)||TS(2)||LifeTime2   ||Ticket(tgs)；   1. K(c,tgs)||ID(c)||AD(c)||ID(tgs)||TS(2)   ||Lifetime2；   1. K(c,v)||ID(c)||AD(c)||ID(v)||TS(4)   ||Lifetime(4)； |
| DES | TGS、V | 1. C通过自己的Session-Key完成C与TGS的重要信息交互； 2. C通过自己的Session-Key完成C与V的重要信息交互。 | 1. K(c,v) || ID(v) || TS(4) || Ticket(v)； 2. TS(5)+1； |

Table 2-4 应用场景

## 多线程通信

由于在Kerberos中客户端C和服务器V都不是单单一台，所以这里必须使用多线程对应多端口的形式去实现Kerberos的完整流程。

由于对于服务器功能的设想是能够实现消息传递和文件传输，所以说有两个服务器，一个服务器负责汇总消息，将各个客户端的信息保存并转发给各个客户端，另一个服务器的功能是通过服务器对服务器所在的机器实现文件操作。综上，从功能上看，每个服务器都需要至少4个线程。

同理，由于AS和TGS都需要有同一时刻对应4个客户端的请求的能力，所以其要求与服务器相同，也是至少4个线程。

至于详细设计方面，由于每一个线程都要使用socket通信，而且C的数量不多且相对固定，所以这里不适用线程池的方法做线程分配，AS、TGS和V直接分配线程给每一个可能连接上的C，如图2-9所示：

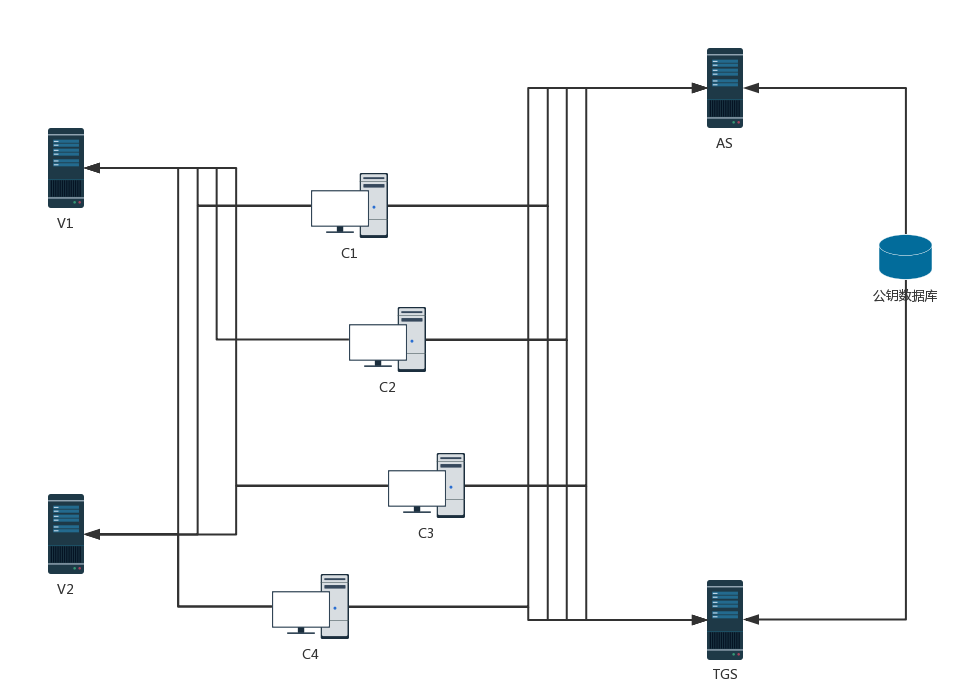


Figure 2‑9 部署图

## 模块分配

模块大致有以下五个模块，分别是DES对称加密模块、RSA+Hash签名模块、多线程Socket通信模块、UI设计模块和状态机模块，详细分配图2-10所示：

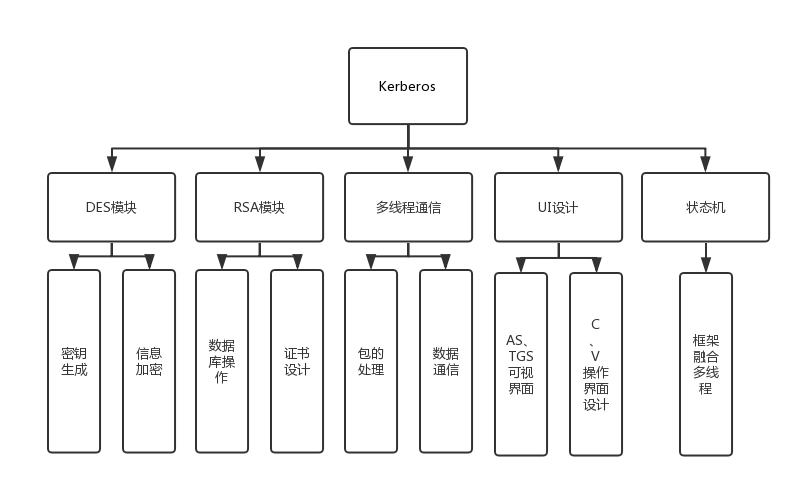


Figure 2‑10 模块图

|  |  |  |
| --- | --- | --- |
| 姓名 | 本职任务 | 协作任务 |
| 汪宇晖 | 设计完成多线程通信模块和RSA模块 | 配合组员完成各个分散的模块拼接完整； |
| 田宸威 | 设计完成状态机部分 | 将状态机和多线程通信框架相互融合； |
| 文钱坤 | 完成UI部分 | 将UI部分与操作控制部分衔接； |
| 钱志磊 | 完成DES部分 | 测试修改； |

Table 2-4 职能分配

根据此次分配的情况，加上我对自身和组员的能力了解，大概的工作进度规划如图2-11所示：

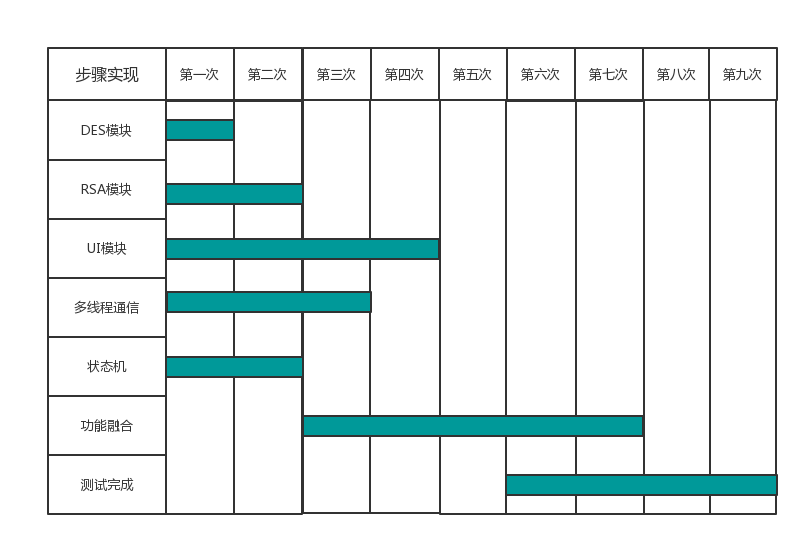


Figure 2‑11 甘特图

# 

# 第三章拟定设计

## 3.1数据格式

### 3.1.1 数据包首部

由于还没有实际进行操作，初步设想需要8位有用的首位，但为了以防万一，首位的设计为12位，留下4位补零以供扩充，8位实际有用位如表2-5所示：

|  |  |  |  |
| --- | --- | --- | --- |
| 发送方 | 接收方 | 功能 | 通信 |
| 00：C | 00：C | 00：认证 | 00：普通通信 |
| 01：AS | 01：AS | 01：C与V交互 | 01：文件名获取 |
| 10：TGS | 10：TGS | 10：错误 | 10：文件名发送 |
| 11：V | 11：V | 11：断开 | 11：文件内容 |

Table 3-1 数据包首部

用上面的方法，比如在第一步中C->AS，数据包的首部如下图所示：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

同理，第二步中TGS->C，数据包首部如下：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

在最后的功能操作中，比如C->V，发送文件内容：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

完整的数据包流程如下：

**验证包穷举如下所示：**

**第一步：**由C->AS，由AS收到相应的信息：

包的首部为：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

其中的数据内容以字符串的形式展现：

ID(c)||ID(tgs)||TS1

从首部AS可以看出是C的请求通信认证，然后将后面几位的字符串获取后按照“||”切割得到相应的信息，根据这些信息回复给C。

**第二步：**由AS->C，由C收到相应的信息：

包的首部为：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

其中的数据内容以字符串的形式展现：

EK(c)[K(c,tgs)||ID(tgs)||TS2||Lifetime2||Ticket(tgs)]

从首部C可以看出是AS的请求通信认证，然后将后面的字符串进行RSA解码，解码后通过“||”切割得到相应的信息，根据这些信息发送给AS。

**第三步：**由C->TGS，由TGS收到相应的信息：

包的首部为：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

其中的数据内容以字符串的形式展现：

ID(v)||Ticket(tgs)||Authenticator

从首部TGS可以看出是C的请求通信认证，然后将后面的字符串进行“||”切分得到相应的字符，根据这些字符进行相应的操作，再发包给客户端。

**第四步：**由TGS->C，由C收到相应的信息：

包的首部：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

其中的数据内容以字符串的形式展现：

EK(c,tgs)[K(c,v)||ID(v)||TS4||Ticket(v)]

从首部C可以看出是TGS的请求通信认证，先进行DES解密操作，然后将后面的字符串进行“||”切分得到相应的字符，根据这些字符进行相应的操作，再发包给V。

**第五步：**由C->V，由V收到相应的信息：

包的首部：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

其中的数据内容以字符串的形式展现：

Ticket(v)||Authenticatorc

从首部V可以看出C的请求通信认证，然后将后面的字符串进行“||”切分得到相应的字符串，根据这些字符串进行相应的操作，再发包给C。

**第六步：**由V->C，由C收到相应的信息：

包的首部：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

其中的数据内容以字符串的形式展现：

K(c,v)||ID(c)||AD(c)||ID(v)||TS4||Lifetime(4)

从首部C可以看出V的请求通信认证，然后将字符串进行DES的解密，然后将后面的字符串进行“||”切分得到相应的字符串，然后可以进行文件传输或者聊天通信。

**聊天通信包穷举如下所示：**

C->V，V->C

包首部：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

其中的内容有：

Session-Key（时间||用户||信息）

**文件通信包穷举如下所示：**

**文件下载：**C->V

包首部：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |

其中的内容有：

空

V->C

包首部：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

其中的内容有：

Session-Key（文件夹内文件的内容）

1. >V

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

其中的内容有：

Session-Key（文件名）

V->C

包首部：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

其中的内容有：

Session-Key（文件内容，定义一个上限，上限达到持续监听，上限未达到，完成传输）

回到文件传输的初始状态

**文件上传：**C->V

包首部：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

其中的内容有：

Session-Key（文件名）

V->C

包首部：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

其中的内容有：

空

C->V

包首部：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

其中的内容有：

Session-Key（文件内容，定义一个上限，上限达到持续监听，上限未达到，完成传输）

回到文件传输的初始状态

**断开请求：**如C-V：

包首部：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

V-C：

包首部：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

**错误信息：**如V-C:：

包首部：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

其中内容：

均为空。

算上除了包首部中表示接收方和发送方不同的地方（C->V错误信息，C->AS错误信息，AS->C错误信息，TGS->C错误信息，C->TGS错误信息），总共23个包。

### 3.1.2密钥

相关的加解密操作我在上面2.2已经详细说明了，Kc，Ktgs，Kv都是通过RSA加密，然后通过自己的私钥进行解密，但是Session-Key是通过Des对称钥匙进行加密，session-Key有Key(c,tgs)，Key(c,v)，这里都是可以长时间使用的对称密钥。

相关的私钥和密钥都保存在各自的私有成员变量中，不对外进行传递。

### 3.1.3函数命名

**客户机函数命名**

|  |  |
| --- | --- |
| 函数名 | 操作 |
| void DES\_E(Stirng key , HashMap message) | 利用DES密钥加密包信息 |
| Void Unpack(HashMap message) | 拆包 |
| String DES\_D(Stirng key , HashMap message) | 利用DES密钥加密包信息 |
| String RAS\_D(Stirng key , HashMap message) | 利用RSA进行签名操作 |
| Void Link\_IP(int port , String IP) | 链接相应的服务器（AS，TGS，V） |
| Void Send\_mes(HashMap message) | 向服务器验证发包 |
| HashMap Recv\_mes() | 接受服务器包 |

**AS服务器函数命名**

|  |  |
| --- | --- |
| 函数名 | 操作 |
| Void DES\_E(Stirng key , HashMap message) | 利用DES密钥加密包信息 |
| Void Unpack(HashMap message) | 拆包 |
| String DES\_D(Stirng key , HashMap message) | 利用DES密钥加密包信息 |
| String DES\_KEY() | 随机生成DES密钥 |
| String DES\_E(Stirng key , String message) | 利用DES密钥加密字符串信息 |
| Void Link\_IP(int port , String IP) | 链接C |
| Void Send\_mes(HashMap message) | 向客户机验证发包 |
| HashMap Recv\_mes() | 接受客户机包 |

**TGS服务器函数命名**

|  |  |
| --- | --- |
| 函数名 | 操作 |
| Void DES\_E(Stirng key , HashMap message) | 利用DES密钥加密包信息 |
| Void Unpack(HashMap message) | 拆包 |
| String DES\_D(Stirng key , HashMap message) | 利用DES密钥加密包信息 |
| String DES\_KEY() | 随机生成DES密钥 |
| String DES\_E(Stirng key , String message) | 利用DES密钥加密字符串信息 |
| Void Link\_IP(int port , String IP) | 链接C |
| Void Send\_mes(HashMap message) | 向客户机验证发包 |
| HashMap Recv\_mes() | 接受客户机包 |

**FTP服务器函数命名**

|  |  |
| --- | --- |
| 函数名 | 操作 |
| String FileSeach(String dir) | 得到当前目录所有文件文件名 |
| Void DES\_E(Stirng key , HashMap message) | 利用DES密钥加密包信息 |
| Void Unpack(HashMap message) | 拆包 |
| String DES\_D(Stirng key , HashMap message) | 利用DES密钥加密包信息 |
| String DES\_E(Stirng key , String message) | 利用DES密钥加密字符串信息 |
| Void Link\_IP(int port , String IP) | 链接C |
| Boolbean RSA\_E(Stirng key , String message) | 利用RSA公钥验证签名 |
| Void Send\_mes(HashMap message) | 向客户机验证发包 |
| HashMap Recv\_mes() | 接受客户机包 |
| Void FileCreat(String filename) | 创建文件 |
| Void FileCreat(byte[] io) | 向文件添加流数据 |

**聊天室服务器函数命名**

|  |  |
| --- | --- |
| 函数名 | 操作 |
| Void DES\_E(Stirng key , HashMap message) | 利用DES密钥加密包信息 |
| Void Unpack(HashMap message) | 拆包 |
| String DES\_D(Stirng key , HashMap message) | 利用DES密钥加密包信息 |
| String DES\_E(Stirng key , String message) | 利用DES密钥加密字符串信息 |
| Void Link\_IP(int port , String IP) | 链接C |
| Boolbean RSA\_E(Stirng key , String message) | 利用RSA公钥验证签名 |
| Void Send\_mess(int pre[] , vector stack) | 从栈中向相应的客户机发送信息 |
| Void Push(String message) | 将消息实现进栈操作 |
| HashMap Recv\_mes() | 接受客户机包 |

### 3.1.4其他信息

ID：ID用来记录C和V的标识号，如C1，C2……

TS：TS用来发送时间，时间可以直接获取系统中的月日时，4月29日19时52分

Lifetime：票据有效期

## 3.2简要流程

### 3.2.1 时序图

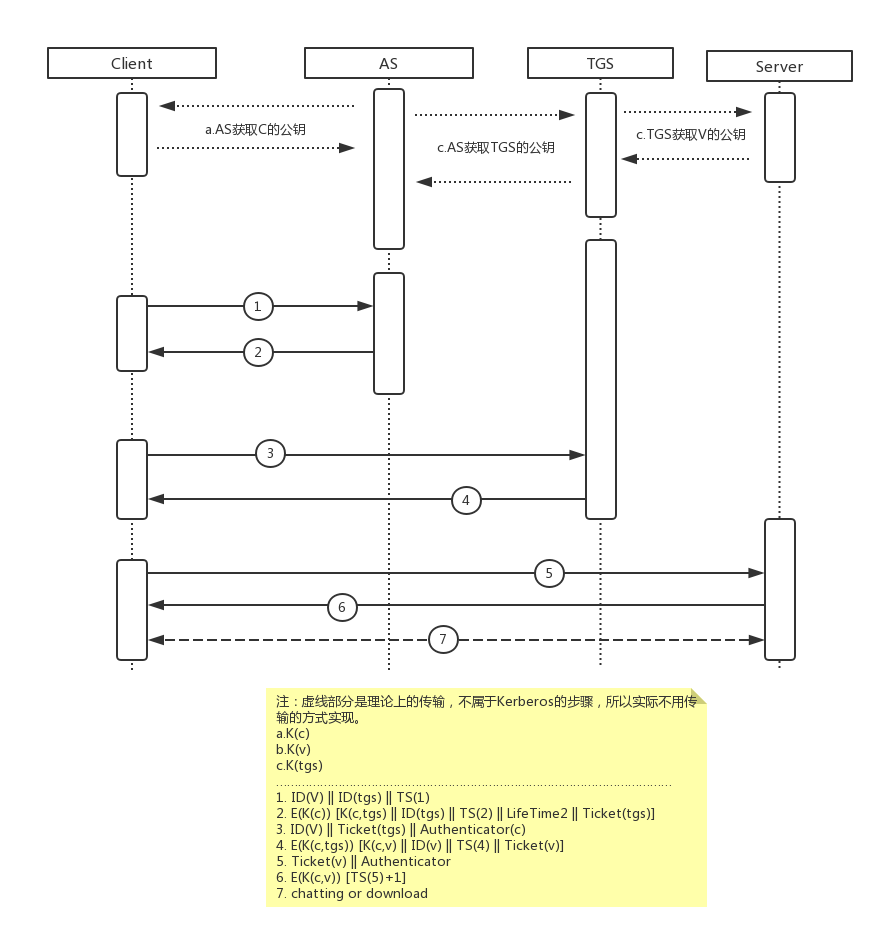


Figure 3‑1

### 3.2.2 Kerberos流程图

根据Kerberos的思路，由上面的时序图可以画出简要的流程图，如图3-2所示：

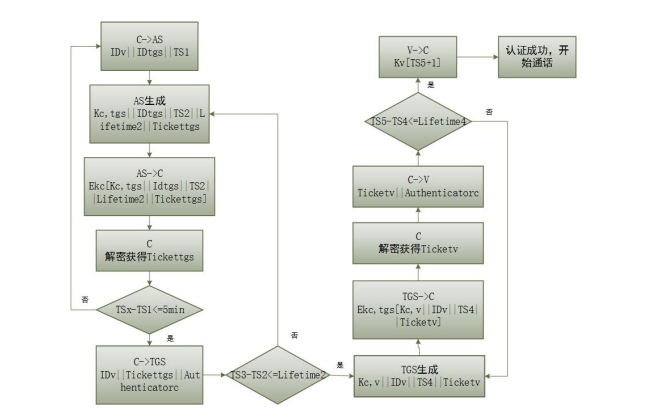


Figure 3‑2 kerberos简要流程图

### 3.2.3客户端工作流程图

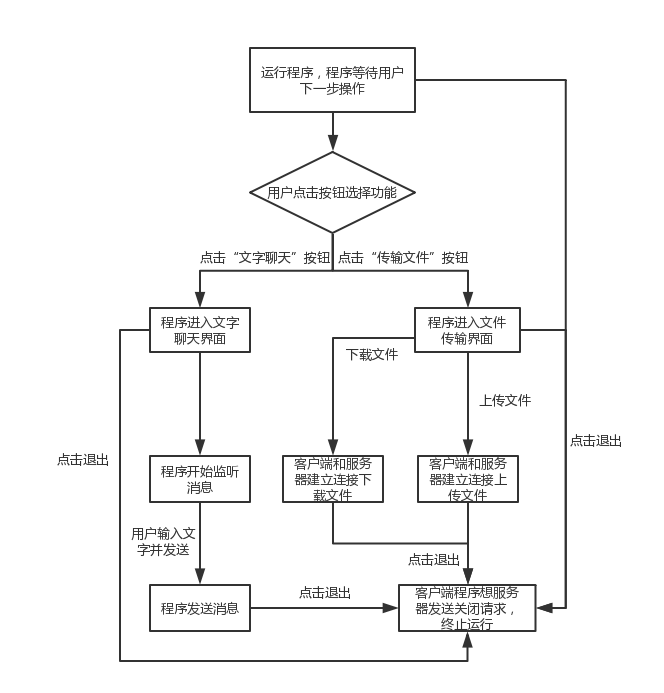


Figure 3‑3 客户端工作流程图

### 3.2.4 AS和TGS服务器工作流程图 未命名文件 (8)未命名文件 (8)

Figure 3‑4 AS和TGS工作流程图

### 3.2.5服务器工作流程图



Figure 3‑5 服务器工作流程图

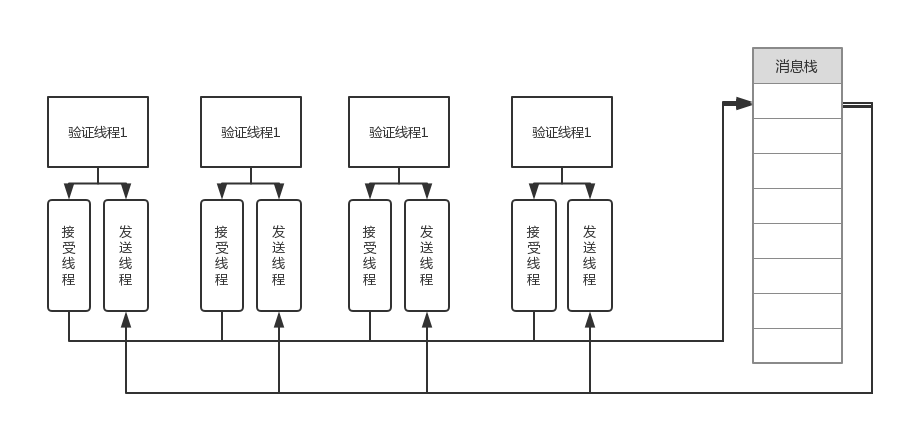
# 4 详细设计

## 4.1细节规约

### 4.1.1端口对应表格

|  |  |  |  |
| --- | --- | --- | --- |
| Client | AS | TGS | chatV |
| Client01 | 10000 | 10004 | 10008 |
| Cilent02 | 10001 | 10005 | 10009 |
| Client03 | 10002 | 10006 | 10010 |
| Client04 | 10003 | 10007 | 10011 |

当client和chatV做互联的时候，每个Client和chatV开放的每个端口，都需要创建两个新的线程去进行收发消息，如图4-1所示：



只要消息栈里有新的内容，就需要将其发送给每一个需要更新的线程，这里需要一个装填情况来确定有哪些客户机在线上，我在chatV的住进程中设置了一个static final int state[4]来确定线上客户机的状态，在数组中，state[0]、state[1]、 state[2]、 state[3]分别表示客户机1、2、3、4的状态，数组中的数字有0和1 ，0表示离线状态，1表示上线状态，在发送消息中，线程通过当前状态和自己的pre指针有没有指到消息栈的上限判断是不是要发送给对应客户机信息

消息栈第一章中的数据结构也介绍了，是一个Vector<Chatinfor>链表，ChatInfor类中存有User，Conv，Time三个String性的数据，分别代表用户，会话，时间。也是一个使用static final保存的独一的数据。

Client与chatV通信的详细端口（Client）：

|  |  |  |  |
| --- | --- | --- | --- |
| Client | 验证 | 发送消息（断开） | 接受消息（断开） |
| Client01 | 10008 | 验证+1000 | 验证+1100 |
| Client02 | 10009 | 验证+1000 | 验证+1100 |
| Client03 | 10010 | 验证+1000 | 验证+1100 |
| Client04 | 10011 | 验证+1000 | 验证+1100 |

Client与chatV通信的详细端口（chatV）：

|  |  |  |  |
| --- | --- | --- | --- |
| （from）Client | 验证 | 发送消息（断开） | 接受消息（断开） |
| Client01 | 10008 | 验证+1100 | 验证+1000 |
| Client02 | 10009 | 验证+1100 | 验证+1000 |
| Client03 | 10010 | 验证+1100 | 验证+1000 |
| Client04 | 10011 | 验证+1100 | 验证+1000 |

### 4.1.2包首部宏定义

在函数中定义一个类表示这些宏定义，类名为：Appoint\_Prelude

|  |  |  |
| --- | --- | --- |
| 名称 | 内容 | 作用 |
| C\_AS | 000100000000 | 验证消息（C->AS） |
| AS\_C | 010000000000 | 验证消息（As->C） |
| C\_TGS | 001000000000 | 验证消息（C->TGS） |
| TGS\_C | 100000000000 | 验证消息（TGS->C） |
| C\_V | 001100000000 | 验证消息（C->V） |
| V\_C | 110000000000 | 验证消息（V->C） |
| C\_V\_chat | 001101000000 | 聊天通信（C->V） |
| V\_C\_chat | 110001000000 | 聊天通信（V->C） |
| C\_V\_chatcut | 001111000000 | 中断连接（C->V） |
| V\_C\_chatcut | 110011000000 | 中断连接（V->C） |
| C\_AS\_error | 000111000000 | 验证错误消息（C->AS） |
| AS\_C\_error | 010011000000 | 验证错误消息（As->C） |
| C\_TGS\_error | 001011000000 | 验证错误消息（C->TGS） |
| TGS\_C\_error | 100011000000 | 验证错误消息（TGS->C） |
| C\_V\_error | 001111000000 | 验证错误消息（C->V） |
| V\_C\_error | 110011000000 | 验证错误消息（C->AS） |