一种应用于UDP网络通讯状态机的智能定时器的软件实现

李艳红
(中南民族大学 计算机科学学院,武汉 430074)

摘 要 分析了提高UDP数据报的传输可靠性的方法,设计了与该方法相配套的由有限状态机、智能定时器、线程协作的算法.所设计的智能定时器支持可变的定时间隔,非常适用于需要“退避算法”的应用场合.通过音视频终端登录服务器的实验,实现了所提出的算法.实验结果表明:智能定时器能按照预设的定时值重发那些没有收到服务器应答的UDP包,从而提高了UDP传输的可靠性.

关键词 状态机;智能定时器;线程;退避算法;UDP通讯;重传

中图分类号 TP311.1;TP393.0 **文献标识码** A **文章编号** 1672-4321(2013)01-0087-05

Software for State-Machine Smart Timer in UDP Communication

Li Yanhong
(College of Computer Science, South-Central University for Nationalities, Wuhan 430074, China)

Abstract In order to improve the reliability of UDP transmission, we design the algorithms matching with the finite state machine, smart timer and thread collaboration. Because the smart timer supports variable time intervals, it can apply to the case where back-off algorithm is particularly required. Consequently, the proposed algorithms are realized through the experiments in video terminal logging to server. The results show that the smart timer can resend UDP packet according to the preserved time intervals when its acknowledgement from the server has not been received.

Keywords state machine; start timer; thread; back-off algorithm; UDP communication; re-send

在很多网络应用程序中,需要使用UDP来传送消息和数据,UDP传输是不可靠的,数据报可能会在传送途中被丢弃.但是它也有很多优点,比如无连接、效率高、系统资源开销小、适用于防火墙穿透等,在互联网实时通讯以及本机进程间通讯中得以广泛使用.设计UDP网络通讯程序,需要解决丢包、重发、异步、并发等问题.TCP和UDP在网络通讯中都必不可少,他们有不同的特点,有各自的适用范围,均不可互相替代.UDP能够在不建立连接的情况下收发数据,也就是说在防火墙允许的情况下,可以发送数据到任何地址,也可以接收任何地址发来的数据.很多场合非常需要这种数据收发的方式.但是正是因为这个特点,丢包的现象也是不可避免的.研究者们关注UDP传输的可靠性问题,并提出了改进的算法[1,2],但是可靠性问题从原理上讲不可能完全解决.

在音视频数据传送时,一定程度丢包率是允许的,不会对会话质量造成很大的影响.但是传送控制信息时,需要用重发的方法来解决丢包问题.重发涉及到重发的最大次数、每次重发的时间间隔,以及状态机改变状态后才收到的(迟来的)应答包的处理方式等,这是异步操作必然要碰到的问题.一般的网络程序,往往需要同时处理一些并发的网络通讯任务,比如A终端与B终端进行视频会话,同时A终端还可能发送文件给C终端.

1 智能定时器的特点

定时器在过程控制中广泛使用,针对一些特定的应用环境,研究者们提出了相应的方案和研究成果[3-8],本文重点研究智能定时器的软件实现.大多数情况下,程序中使用固定间隔值的定时器即可满足定

收稿日期 2013-02-01
作者简介 李艳红(1973-),女,博士,讲师,研究方向:网络通讯,E-mail: liyanhong@mail.scuec.edu.cn
基金项目 国家自然科学基金资助项目(61173049);湖北省自然科学基金资助项目(2012FFB07401)

时的目的.VC、QT、JAVA等程序设计语言,无一例外实现了定时器类,提供了启动、停止、设置定时间隔值等方法,一般有一次性定时和重复定时两种定时模式.但是,这些定时器不能满足复杂的定时需求.例如运用于退避算法,定时间隔需要变化、然后定时间隔再趋于稳定的定时需求.而且,在不同场合间隔的变化方式也是不一样的.为此,本文设计了一种称之为智能定时器的C++类,来解决这个问题.

1.1 智能定时器类的设计

该定时器类的组成并不复杂,具体包括:一个数组、一个用于存放该数组元素个数的整数、一个时间戳、一个计数器、一个最大重试次数、判断是否超时的函数,以及判断是否重试的函数.

数组用来存放定时的间隔(超时值),也即每一次重试等待的时间.数组长度可长可短,由重试的频度变化策略决定,比如发送一个UDP包后,如果没收到应答,则需要按2s、4s、8s、16s、16s这样的间隔重发,那么该数组就是[0,2,4,8,16,16].由于数组是由构造函数的参数传入的,构造函数得到的是该数组的地址,在函数中无法求得其元素个数,所以需要将元素个数也作为参数传入构造函数.在构造函数中,将数组地址及其元素个数保存到成员变量中.

时间戳用来存放上次重试(重发)的系统时间.计数器用来统计重试的次数.最大重试次数用来判断是否超时,这里的超时表示重试结束.如果最大重试次数为0,则表示需要无限次重试.

如果计数器大于最大重试次数,并且最大重试次数大于0,则表示已经超时.判断是否需要重试的方法是,取出当前计数器对应数组元素的值,然后用当前系统时间减去时间戳的值,如果差值大于等于数组元素值,则表示需要重试.此时,将当前系统时间赋值给时间戳,作为下次判断的依据.

定时器类的设计具体如下:

```
class smartTimer{
private:
    struct timeb lastTime;//时间戳,存放上一次重试的时间
    unsigned int count;//重试次数,初始化为0
    unsigned int retryMax;//最大重试次数,构造函数默认为0,即无限次重试
    unsigned int * array;//重试时间间隔数组地址,由构造函数参数传入
    unsigned int arrayLen;//重试时间间隔数组元素个数,由构造函数参数传入
public:
    smartTimer(unsigned int * a, int n){ array=a; arrayLen =n; reset(0);} //构造函数,默认无限次重试
    void reset(unsigned int max =0){//reset函数设置重试次数
     retryMax = max; count = 0; ftime(&lastTime);//初始化max,count,lastTime
    }
    unsigned int getInterval(){return (count >= arrayLen)? array [arrayLen-1] : array[count];}
    bool isTimeOut(){return (retryMax >0 && count > retryMax)? true:false;}//是否重试超时
    bool isNextStep() {
        struct timeb t; ftime(&t);
        if (((t.time * 1000 + t.millitm) - (lastTime.time * 1000 + lastTime.millitm)) >= getInterval()) {
            count++; ftime(&lastTime);//可以重试,计数增1,重置时间戳
            return true;
        }
        else
            return false;
    }
};//end of class
```

1.2 智能定时器工作原理

上文所述的定时器是一个被动对象,也即没有自己的线程,需要在定时器线程以及状态机的配合下才能发挥作用.没有设计自己的线程,是为了节省系统的开销.因为一个稍微复杂的系统,可能需要多个定时器,如果每个定时器都有一个线程在运行,总体上会增加CPU空转的时间.

状态机一般是指有限状态机,由一系列的状态、事件、条件、动作组成.在某一时刻,状态机只能处于某个状态,称为当前状态.当发生某个事件时,如果条件满足,则状态机的状态将发生改变,这个改变称为状态转移.状态转移往往伴随着一些动作(任务)的执行.

线程是利用CPU时间片轮转来获得执行权的代

码片段. 在多数线程的使用中,这个代码片段会设计成循环体,只有在特殊条件下才退出循环来结束线程. 每一次循环内,调用状态机处理函数,处理一次状态机. 如果系统需要多个状态机,则每个状态机都处理一次,调用完所有的处理函数后是短暂的休眠. 休眠的时间与定时器的精度相关,比如精度为 20ms 则休眠 20ms,也即预定 1s 的定时,那么实际的定时间隔为 1000 ~ 1020ms. 除非是实时系统需要精确到微秒,一般的应用,精度为 20 ~ 100ms 就可以满足要求.

状态机处理函数因为要处理很多状态,适合用 switch…case 构建,该函数被调用时,执行的代码是状态机当前状态的 case,在此状态中,如果有重传,或者超时状态转移的需求,则可分别调用 isNextStep( ) 和 isTimeOut ( ) 来判断. 当 isTimeOut 返回 true 时,表示状态机需要转移到新的状态,而 isNextStep 返回 true 时,表示在当前状态下需要重复执行某些任务. 在 isNextStep 被调用时,计数器 count 和上次访问 last time 才会被更新. 这表明,增加智能定时器只是增加少量内存的使用,不会增加系统的 CPU 资源消耗.

状态机的状态转移,除了超时的原因外,更多的是人机交互事件和外部系统事件触发的,比如用户点击"上线"按钮,或者是收到服务器的应答信息.

2　智能定时器运用实例的具体实现

本文以音视频会话终端软件的登录过程作为实例,来说明智能定时器的功能和运用方法. 程序分终端程序和服务器端程序,均在 Linux 平台开发和运行,采用 C + + 语言编程.

2.1　登录状态机

终端登录的过程用状态机表示,如图 1 所示.

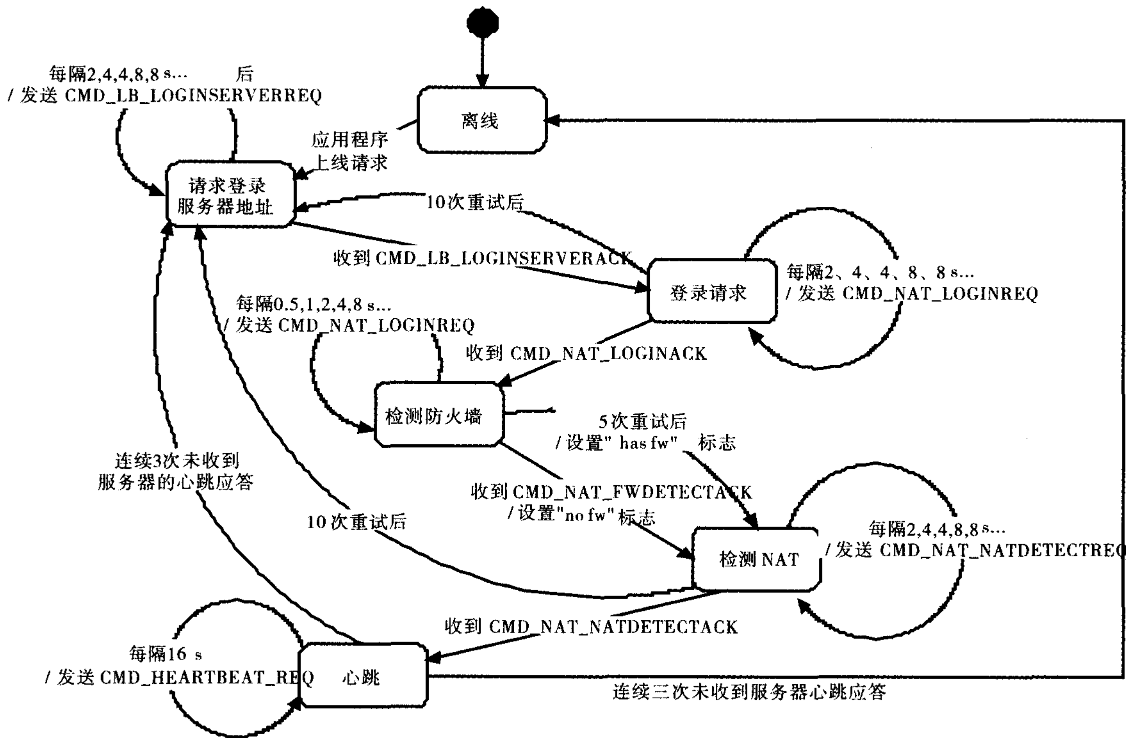

图 1　终端登录状态机

Fig. 1　Terminal login state machine

该状态机有"离线"、"请求登录服务器地址"、"登录请求"、"检测防火墙"、"检测 NAT"、"心跳"6 个状态. 除了离线状态,其余状态都使用了定时器. 这些定时器中,只有在心跳状态下终端才会以固定的时间间隔发送心跳包,其余状态的终端都要按照一定的退避算法来重发数据. 按退避算法的方式来重发的原因是,如果服务器忙于处理其他任务,不能及时给终端返回应答,终端应当适当放慢重试的频率;又由于终端向服务器发送的请求数据可能在网络上丢失,那么终端也不能无限制等待,希望服务器最终会返回应答.

2.2　定时间隔数组及定时器对象

根据登录过程的状态机,制定定时器策略. 终端登录过程共需要 3 种定时器:1) 登录请求定时器,分时复用于 3 个状态:获取登录服务器地址、登录请求、检测 NAT;2) 防火墙检测定时器,用于检测防火墙;3) 心跳维持定时器,用于心跳包定时发送. 如表 1 所示.

表1 定时策略表

Tab. 1 Timing strategy table

Timer	Intervals/ms
登录请求定时器	0,2000,4000,4000,8000,8000, …
防火墙检测定时器	0,500,1000,2000,4000,8000, …
心跳维持定时器	0,16000, …

根据定时策略表,定义3个数组:

unsigned int tmLogin [5] = {0,2000,4000,4000,8000};

unsigned int tmFwDetect [6] = {0, 500, 1000, 2000,4000,8000};

unsigned int tmHeartBeat [2] = {0,16000};

然后,声明智能定时器实例,构造函数中传入数组及其元素个数:

startTimer stLogin (tmLogin, sizeof (tmLogin)/ sizeof(tmLogin[0]));

startTimer stFwDetect (tmFwDetect, sizeof (tmFwDetect)/ sizeof(tmFwDetect [0]));

startTimer stHeartBeat (tmHeartBeat, sizeof (tmHeartBeat)/ sizeof(tmHeartBeat [0]));

2.3 登录过程中所涉及的算法

登录过程涉及到3个算法,分别是:定时器线程、登录状态机处理函数、服务器应答处理.

(1) 定时器线程算法.

```
Function TimerThread
pre-condition: sm is initialized to offline
1  While program not quitting do {
2    smLoginProcess();
3    sleep 100 ms;
4  }
```

(2) 登录状态机处理函数算法.

```
Function smLoginProcess
1  switch (sm) {
2  case offline:
3    break;
4  case login_server_req:
5    if (stLogin. isNextStep()){
6        send CMD_LB_LOGINSERVERREQ
         packet to load-balance server;
7    }
8    break;
9  case login_req:
10   if (stLogin. isTimeOut ()){//如果超时则返回到 login_server_req 状态
11     sm = login_server_req;
12     stLogin. reset(0);
13     break;
14   }
15   if (stLogin. isNextStep()){//否则如果时间间隔已满则重发登录请求到登录服务器
16     send CMD_NAT_LOGINREQ packet to login server;
17   }
18   break;
19 case firewall_detecting:
20   如果 stLogin 超时(5次重试无应答)则说明有防火墙,记录防火墙特性,复位stFwDetect,设置状态为 nat_detecting. 否则如果时间间隔已满则重发防火墙检测请求.
21   break;
22 case nat_detecting:
23   如果 stFwDetect 超时则返回 login_server_req 状态. 否则如果时间间隔已满则重发NAT 检测请求.
24   break;
25 case heat_beating:
26   如果超时则返回 login_server_req 状态,否则如果时间间隔已满则重发心跳包.
27   break;
28 }
```

(3) 服务器应答处理算法片段.

```
Function onServerAck(char * pkt)
1  switch (cmd in pkt) {//所有服务器返回的应答包中均含有 cmd(命令字)
2  case CMD_LB_LOGINSERVERACK:
3    if (login_server_req) {//该应答包只有在 login_server_req 状态下才有效
4      sm = login_req; //更新状态机的当前状态为 login_req
5      stLogin. reset(10); //重置定时器 stLogin,表示在 login_req 状态最多重发10次.
6      get ip,port of login server from pkt;//取得应答包中所含登录服务器的地址
7    }
8    break;
9  case CMD_NAT_LOGINACK:
10   …(略)
```

由于篇幅所限,不能完整描述这3个算法对所有状态和所有服务器应答包的处理情况,以下重点介绍终端由 offline(离线)状态和 login_server_req(获取登录服务器地址)状态相关的处理.

首先需要注意的是定时器线程算法 TimerThread,该线程每隔100ms会执行一次状态机处理函数 smLoginProcess.

终端程序初始化后,状态机处于 offline(离线). 在 offline 状态,smLoginProcess 处理函数不做任何操作,立即返回(第3行代码). 只有人机交互才能改变 offline 的状态,比如通过按钮发出"上线"请求,相应的代码会将状态机的状态由 offline 转移到 login_server_req,并调用 stLogin. reset(0)复位定时器,也即在该状态下将无限次地重试,直到从负载均衡服务器取到登录服务器地址为止.

状态机处理函数 smLoginProcess 对 login_server_req 状态的处理见4~8行. 在此调用 stLogin. isNextStep 来判断是否该发送 CMD_LB_LOGINSERVERREQ 到负载均衡服务器. 在 smartTimer 的 isNextStep 代码中可以看到,如果当前时间减去上次重试的时间大于当前重试间隔值,会返回 true. 在 login_server_req 状态下,stLogin. isNextStep 返回 true 时,程序将执行第6行所示的动作,发送 CMD_LB_LOGINSERVERREQ 到负载均衡服务器.

如果一直没有收到负载均衡服务器的应答,则状态机处理函数将永远执行4~8行的代码. 负载均衡服务器的应答在算法 onServerAck 的2~8行中处理,负载均衡服务器返回的应答包中包含了登录服务器的地址,这个应答包只有在终端的状态为 login_server_req 时才被处理,在其他状态则被忽略. 处理方法是:1)状态机变为 login_req;2)将定时器 stLogin 复位为10次超时,这10次的间隔值分别为0、2、4、4、8、8、8、8、8、8s;3)从应答包中取出登录服务器的地址.

状态机处理函数 smLoginProcess 对 login_req 状态的处理见9~18行. 先调用 stLogin. isTimeOut,如果超时则状态回转到 login_server_req 重新取登录服务器地址. 如果没有超时,则调用 stLogin. isNextStep 判断是否需要重试发送登录请求.

以上3个算法组成的代码框架,除了实现数据重发,也体现了并发、同步操作的理念. 终端发出请求包,服务器返回应答包是异步操作,不能设计成函数调用得到返回值的方式. smLoginProcess 中发送请求,而在 onServerAck 中处理应答,使用了异步处理的模式. 终端完成登录后进入心跳状态,在心跳状态终端每16s向服务器发送一次心跳包. 如果用户发起音视频会话,那么心跳包和会话信令包以及音视频数据包就需要并发处理.

3 结语

本文介绍了一种能够处理复杂状态机和程序逻辑的智能定时器,该智能定时器以一个类的形式进行封装. 智能定时器实例化时,从构造函数传入预设的定时数组,数组的元素个数可根据需要进行设定,每个元素的值代表某个操作重复执行的等待时间. 该智能定时器属于被动对象(没有自己的线程),因此需要在线程中去查询是否重复执行某操作,或者去查询是否超时. 本文所用的定时器线程和状态机处理函数,是支撑智能定时器的线程、状态机处理的典型代码框架. 千变万化的状态机模型,均可按照此框架来构建程序. 最后实现了将智能定时器应用于实时通讯软件系统的登录过程,实验结果表明所设计的智能定时器能精确控制UDP包的重发.

参考文献

[1] 李　国,巩光志,王冬冬. 一种提高UDP可靠性的数据传输方法研究[J]. 中国民航大学学报, 2012(01): 41-45.

[2] 王艳芳,戴　永,刘东华,等. 基于UDP的数据可靠传输技术研究与应用[J]. 计算机工程与应用, 2010(03): 105-108.

[3] 孙宏旭,邢　薇,陶　林. 基于有限状态机的模型转换方法的研究[J]. 计算机技术与发展,2012(02): 10-13.

[4] 李庆华,陈志刚,邓晓衡. 基于线性均方误差的无线自组网TCP定时器改进[J]. 中南大学学报:自然科学版, 2012(05):1780-1786.

[5] 朱正发,陈琳. 一种嵌入式基带系统定时器装置的研究[J]. 单片机与嵌入式系统应用,2012(11):12-14.

[6] 朱旭光. 单片机定时器应用探讨[J]. 自动化技术与应用, 2012(01):99-103.

[7] 王秀霞. 基于555定时器的开关电源的设计[J]. 微计算机信息, 2012(01):76-78.

[8] 任君玉,黎国文. 网络中的定时器技术[J]. 电脑知识与技术, 2011(21):5094-5095.