# *UDP\_domain\_server\_Tool* 代码编写总结

**------------written by 糖没味儿**

**2017年10月16日**

## 项目需求

UDP\_domain\_server\_Tool代码（以下简称Tool）的作用是连接RRM和RRMtester。由于RRM运行在开发板上，RRMtester运行在PC上，并且RRM对外只能过domain socket，所以RRM想与RRMtester通信，就必须通过Tool转发。

### 具体指标

* Tool工具将UDP与domain socket发出来的msg进行转发（不存储），从而实现UDP与domain socket消息的双向收发（udp <--> domain socket）。
* Tool工具涉及到两种通信方式：UDP和domain socket。具体编写代码时，要求domain socket地址、UDP端口灵活可配置。
* 代码最终在PowerPC平台上运行

### 输入格式要求

为实现具体指标中的第二条，可通过命令行传参的方式将客户端UDP端口、IP地址（非本机msg转发情况下，需传递IP）和domain socket的地址一并传入Tool中。

在PowerPC上编译完Tool工具后，

**输入：.*/server <domain\_addr> <udp\_ip> <udp\_port>***

即可将domain socket地址、客户端UDP的IP和端口传入Tool中。另外，如果忘了命令输入顺序，可执行***/server ?*** 聪明的Tool会提示你输入格式的。

## Tool设计及调试

### Tool设计

* Tool会转发RRM和RRMtester之间的消息，相当于有两个客户端会触发I/O请求，于是考虑Tool使用epoll机制处理多任务I/O并发请求。
* RRM端对外只能通过domain socket，而RRMtester运行在PC，因此Tool需绑定并监听两种套接字----UDP和domain socket。
* Tool能识别不同套接口来的msg，并进行接收（recvfrom）和转发（sendto）到另一种套接口中。

### Tool调试

放入真实平台前，为了方便准确定位BUG，需对Tool代码进行调试。

需模拟两个相同条件下的客户端，一个是本地domain socket客户端模拟RRM，一个是非本机UDP客户端模拟RRMtester。

#### 本地domain socket客户端

一开始对domain socket的设定是采用单线程，但在实际测试时发现，当此单线程domain socket客户端不发送消息，而tool转发了对端UDP客户端发来的消息，需要此单线程客户端接收消息时，此单线程domain socket客户端没办法正常接收消息，因为线程被阻塞到了recvfrom前面的sendto函数，即在接收消息前，必须先发送消息。

这样的工作机制在实际使用中让人很头疼，为了让它正常工作，得先让这个domain socket单线程客户端发送完消息后它才能正常接收对端UDP发来的消息，随之而来又有一个让人头疼的问题是，必须得采用“一收一发”的机制，才能让这个单线程客户端持续像正常工作。

为了能让客户端随意收发消息，让程序不受制于阻塞在单线程某个函数中而“僵死”，于是考虑采用**多线程**工作方式，再开一个发送消息线程来专门负责消息发送。这样一来，主进程与线程同时运行：线程负责捕获“发送”，主进程负责捕获“接收”，这样发送和接收就不会互相影响了。

#### 非本机UDP客户端

同理，对于非本机的UDP客户端也采用多线程方式，由于测试环境的限制（windows操作系统通过xshell连接linux服务器），没有两套linux系统机器，于是这个UDP客户端采用***sokit-1.3-win32-enu***或者***SocketMgrV1.1***小工具（在Windows平台下模拟linux套接字工具）模拟多线程UDP客户端。

#### 调试结果



从调试结果来看，UDP客户端发消息1，Tool成功转发至给domain socket客户端；domain socket客户端发消息2，Tool成功转发给UDP客户端。在后面的测试中，实现了UDP客户端随机发，domain socket客户端随机收和UDP客户端随机收，domain socket客户端随机发的功能。

## 系统测试

## 知识点总结与拓展

### Tool端

#### epoll知识点整理

epoll是Linux下多路复用IO接口select/poll的增强版本，当程序中存在大量并发连接时，epoll能选择其中少量活跃连接进行响应和处理，从一定程度上提高了CPU利用率；在获取事件时，它无须遍历整个侦听的描述符集，只要遍历那些已经唤醒并准备好接受服务器下一步命令的描述符集。

epoll操作过程中需要三类接口：

#include <sys/epoll.h>

int epoll\_create(int size);

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout);

##### int epoll\_create(int size);

这个函数是创建一个epoll句柄，创建epoll句柄后，会占用一个fd。size用来告知内核监听的数目有多大。使用完epoll后，必须调用close()关闭，否则可能导致fd被耗尽。

##### int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

这个函数是epoll的事件注册函数。

int epfd：这个参数是epoll\_create的返回值；

int op：这个参数表示动作，它有三个可选项：

* EPOLL\_CTL\_ADD：注册新的fd到epfd中；
* EPOLL\_CTL\_MOD：修改已经注册的fd的监听事件；
* EPOLL\_CTL\_DEL：从epfd中删除一个fd；

int fd：这个参数表示需要监听的fd

struct epoll\_event \*event：这个参数表示告诉内核需要监听什么事件，struct epoll\_event结构如下：

struct epoll\_event

{

\_\_uint32\_t events; /\* Epoll events \*/

epoll\_data\_t data; /\* User data variable \*/

};

events可以是以下几个宏的集合：  
EPOLLIN ：表示对应的文件描述符可以读（包括对端SOCKET正常关闭）；

EPOLLOUT：表示对应的文件描述符可以写；

EPOLLPRI：表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；

EPOLLERR：表示对应的文件描述符发生错误；

EPOLLHUP：表示对应的文件描述符被挂断；

EPOLLET： 将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的。

EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里

##### int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout);

这个函数是epoll等待事件产生的函数，类似于select()调用。

int epfd：是经过epoll\_ctl函数配置后的返回值；

events：参数events用来从内核得到事件的集合；

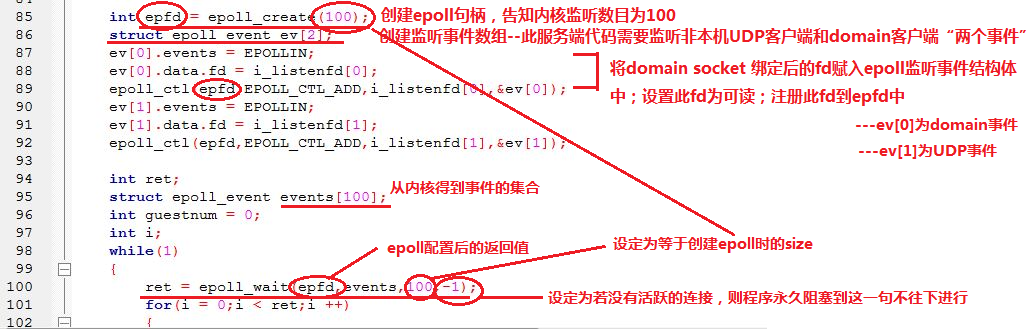
maxevents：告之内核这个events有多大，这个maxevents的值不能大于创建epoll\_create()时的size；

timeout：是超时时间（毫秒，0会立即返回，-1将不确定，也有说法说是永久阻塞）。

int epoll\_wait()函数返回需要处理的事件数目，如返回0表示已超时。

#### Tool端epoll的实现

有了上节的储备知识，就可以在服务端开启epoll功能了，在本次编写的UDP\_domain\_server\_Tool代码中，运用的epoll部分代码如下：



*参考资料：*[*http://www.cnblogs.com/Anker/archive/2013/08/17/3263780.html*](http://www.cnblogs.com/Anker/archive/2013/08/17/3263780.html)

### 2、domain socket端

#### domain socket知识点整理

UNIX Domain Socket是在socket架构上发展起来的**用于同一台主机的进程间通信**（IPC），它不需要经过网络协议栈，不需要打包拆包、计算校验和、维护序号和应答等，只是**将应用层数据从一个进程拷贝到另一个进程**。

UNIX Domain Socket有SOCK\_DGRAM或SOCK\_STREAM两种工作模式，类似于UDP和TCP，但是**面向消息**的UNIX Domain Socket也是**可靠**的，**消息既不会丢失也不会顺序错乱**。

UNIX Domain Socket可用于两个没有亲缘关系的进程，是全双工的，是目前使用最广泛的IPC机制。UNIX Domain socket与网络socket类似，可以与网络socket对比应用。

socket=socket.socket(family, type)

* family参数代表地址家族，比较常用的为AF\_INET或AF\_UNIX。AF\_UNIX用于同一台机器上的进程间通信，AF\_INET对于IPV4协议的TCP和UDP 。
* type参数代表套接字类型，SOCK\_STREAM（流套接字）或者SOCK\_DGRAM（数据报文套接字）。

UNIX Domain socket与网络socket二者编程的不同如下：

address family为AF\_UNIX

因为应用于IPC，所以UNIXDomain socket不需要IP和端口，取而代之的是**文件路径来表示“网络地址”**。这点体现在下面两个方面。

地址格式不同，UNIXDomain socket用结构体sockaddr\_un表示，是一个socket类型的文件在文件系统中的路径，这个socket文件由bind()调用创建，***如果调用bind()时该文件已存在，则bind()错误返回。***

*参考资料：*[*http://blog.csdn.net/guxch/article/details/7041052*](http://blog.csdn.net/guxch/article/details/7041052)

#### domain socket多线程

在本次domain socket客户端多线程代码中，主要运用的函数为多线程创建函数

##### #include <pthread.h>

pthread\_create(pthread\_t \*tidp,const pthread\_attr\_t \*attr, (void\*)(\*start\_rtn)(void\*),void \*arg)

这个函数是线程创建函数，实际上是确定调用该线程函数的入口点，在函数创建以后，就开始运行相关线程函数。其中：

* tidp:线程描述符，由pthread\_t tidp申明后使用；
* attr:线程属性设置，一般使用默认属性，此项设置为NULL；
* \*start\_rtn:线程函数的起始位置，意思是创建完这个线程后，在这个线程里需要执行的子函数，在本次代码中，需要执行的子函数功能是专门负责消息发送；
* arg:传递给\*start\_rtn中线程函数的参数。（注意：正常写子函数时，如果有参数需要传递，这个参数是放在子函数括号里，但是放在线程创建中的子函数，它的参数传递不能放在子函数括号里，否则就变成在线程创建函数里的子函数调用，而不是线程函数了！）

pthread\_create()返回值：成功0，失败-1；

*参考资料：*[*http://blog.csdn.net/youbang321/article/details/7815707*](http://blog.csdn.net/youbang321/article/details/7815707)

下面贴上domain socket客户端多线程相关代码部分截图：

#### domain socket多线程调试代码(部分)



### 3、UDP端

结合上面对domain socket代码端编写储备知识，对于非本机UDP客户端，在代码中体现为：在socket=socket.socket(family, type) 中，address family为AF\_INET，type为SOCK\_DGRAM

对比domain socket，UDP需要IP和端口来代表网络地址，代码中体现为：

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

servaddr.sin\_port = htons(40000);

另外，用于UDP通信的两个重要的函数分别为sendto 和 recvfrom，具体使用参考<http://blog.csdn.net/u011068702/article/details/56680752> 这里不加赘述。

## 代码入库（gitlib）

## 完整代码

### 服务端代码

#include <stdio.h>

#include <stdlib.h>

#include <sys/socket.h>

#include <sys/un.h>

#include <string.h>

#include <unistd.h>

#include <sys/epoll.h>

#include <netinet/in.h>

#include <arpa/inet.h>

int main(int argc,char \*argv[])

{

int i\_listenfd[2] = {0,0}/\*, i\_clientfd = 0\*/;

struct sockaddr\_in servaddr;

struct sockaddr\_un addr\_server;

struct sockaddr\_un addr\_client1;

struct sockaddr\_in addr\_client2;

char psz\_path[32] = "./server\_unixsocket\_file";

int i\_caddr\_len = sizeof(struct sockaddr\_un);

int i\_saddr\_len = 0;

int i\_caddr\_len2 = 0;

char psz\_rbuf[32] = {0};

char psz\_wbuf[32] = "i am server.";

int i\_readlen = 0;

/\*define input rule\*/

if(argc < 4 || argv[0] == "?")

{

printf("Usage: ./server <domain\_addr> <udp\_ip> <udp\_port> \n");

exit(0);

}

//create a UNIX domain stream socket

if ( ( i\_listenfd[0] = socket( AF\_UNIX, SOCK\_DGRAM, 0 ) ) < 0 )

{

perror( "local socket" );

return -1;

}

//create a IPv4 socket

if ( ( i\_listenfd[1] = socket( AF\_INET, SOCK\_DGRAM, 0 ) ) < 0 )

{

perror( "ip socket" );

return -1;

}

//in case it already exists

unlink( psz\_path );

//fill in IPv4 socket address structure

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

servaddr.sin\_port = htons(40000);

addr\_client2.sin\_family = AF\_INET;

addr\_client2.sin\_addr.s\_addr = inet\_addr(argv[2]);

sscanf(argv[3],"%hu" ,&addr\_client2.sin\_port);

addr\_client2.sin\_port = htons(addr\_client2.sin\_port);

//fill in domain socket address structure

memset( &addr\_server, 0, sizeof( addr\_server ) );

addr\_server.sun\_family = AF\_UNIX;

strncpy( addr\_server.sun\_path, psz\_path, sizeof( addr\_server.sun\_path ) - 1 );

memset( &addr\_client1, 0, sizeof( addr\_client1 ) );

addr\_client1.sun\_family = AF\_UNIX;

sscanf(argv[1],"%s",addr\_client1.sun\_path);

//bind the name to the descriptor

i\_saddr\_len = strlen( addr\_server.sun\_path ) + sizeof( addr\_server.sun\_family );

i\_caddr\_len2 = sizeof(servaddr);

if ( bind( i\_listenfd[0], ( struct sockaddr \* )&addr\_server, i\_saddr\_len ) < 0 )

{

perror( "bind" );

return -1;

}

if ( bind( i\_listenfd[1], ( struct sockaddr \* )&servaddr, i\_caddr\_len2 ) < 0 )

{

perror( "bind" );

return -1;

}

int epfd = epoll\_create(100);

struct epoll\_event ev[2];

ev[0].events = EPOLLIN;

ev[0].data.fd = i\_listenfd[0];

epoll\_ctl(epfd,EPOLL\_CTL\_ADD,i\_listenfd[0],&ev[0]);

ev[1].events = EPOLLIN;

ev[1].data.fd = i\_listenfd[1];

epoll\_ctl(epfd,EPOLL\_CTL\_ADD,i\_listenfd[1],&ev[1]);

int ret;

struct epoll\_event events[100];

int guestnum = 0;

int i;

while(1)

{

ret = epoll\_wait(epfd,events,100,-1);

for(i = 0;i < ret;i ++)

{

if((events[i].data.fd == i\_listenfd[0]))

{

i\_readlen = recvfrom( i\_listenfd[0], psz\_rbuf, sizeof( psz\_rbuf ) - 1, 0,

( struct sockaddr \* )&addr\_client1, ( socklen\_t \*)&i\_caddr\_len );

if ( i\_readlen < 0 )

{

perror( "read" );

return -1;

}

psz\_rbuf[i\_readlen] = '\0';

printf( "receive msg:%s\n", psz\_rbuf );

if ( sendto( i\_listenfd[1], psz\_rbuf, strlen( psz\_rbuf ) + 1, 0,

( struct sockaddr \* )&addr\_client2, i\_caddr\_len2 ) < 0 )

{

perror( "write" );

return -1;

}

printf("OK,the msg is send to udp client!\n");

}

else

{

i\_readlen = recvfrom( i\_listenfd[1], psz\_rbuf, sizeof( psz\_rbuf ) - 1, 0,

( struct sockaddr \* )&addr\_client2, ( socklen\_t \*)&i\_caddr\_len2 );

if ( i\_readlen < 0 )

{

perror( "read" );

return -1;

}

psz\_rbuf[i\_readlen] = '\0';

printf( "receive msg:%s\n", psz\_rbuf );

if ( sendto( i\_listenfd[0], psz\_rbuf, strlen( psz\_rbuf ) + 1, 0,

( struct sockaddr \* )&addr\_client1, i\_caddr\_len ) < 0 )

{

perror( "write" );

return -1;

}

printf("OK, msg is send to domain client!\n");

}

}

}

unlink( psz\_path );

return 0;

}

### domain socket客户端代码

#include <stdio.h>

#include <sys/un.h>

#include <sys/socket.h>

#include <string.h>

#include <unistd.h>

#include <stdlib.h>

#include <pthread.h>

#include <netinet/in.h>

#include <errno.h>

#include <sys/time.h>

#include <arpa/inet.h>

#include <sys/select.h>

typedef struct{

int i\_fd ;

char psz\_clientpath[100];

struct sockaddr\_un addr\_server;

}args;

//发送线程，负责消息发送

void \*send\_thread(args \*s\_args)

{ char psz\_clientpath[100];

struct sockaddr\_un addr\_server;

int i\_fd;

i\_fd = s\_args->i\_fd;

memcpy(&psz\_clientpath, s\_args->psz\_clientpath, strlen(s\_args->psz\_clientpath));

addr\_server = s\_args->addr\_server;

struct sockaddr\_un addr;

int i\_addr\_len = 0;

memset( &addr, 0, sizeof( addr ) );

addr.sun\_family = AF\_UNIX;

printf("create pthread success\n");

strncpy( addr.sun\_path, psz\_clientpath, sizeof( addr.sun\_path ) - 1 );

i\_addr\_len = strlen( addr.sun\_path ) + sizeof( addr.sun\_family );

while (1)

{

char psz\_wbuf[32];

scanf("%s",psz\_wbuf);

if ( sendto( i\_fd, psz\_wbuf, strlen( psz\_wbuf ) + 1, 0,

( struct sockaddr \* )&addr\_server, i\_addr\_len ) < 0 )

{

perror( "pthread write" );

return 0;

}

}

return NULL;

}

int main(int argc,char \*argv[])

{

int i\_fd = 0;

struct sockaddr\_un addr;

char psz\_clientpath[32];

strcpy(psz\_clientpath,argv[1]);

char psz\_serverpath[32] = "./server\_unixsocket\_file";

int i\_addr\_len = 0;

char psz\_wbuf[32];

char psz\_rbuf[32] = {0};

int i\_readlen = 0;

int retu;

pthread\_t tid;

if ( ( i\_fd = socket( AF\_UNIX, SOCK\_DGRAM, 0 ) ) < 0 )

{

perror("socket");

return -1;

}

memset( &addr, 0, sizeof( addr ) );

addr.sun\_family = AF\_UNIX;

strncpy( addr.sun\_path, psz\_clientpath, sizeof( addr.sun\_path ) - 1 );

unlink( psz\_clientpath );

i\_addr\_len = strlen( addr.sun\_path ) + sizeof( addr.sun\_family );

if ( bind( i\_fd, ( struct sockaddr \* )&addr, i\_addr\_len ) < 0 )

{

perror("bind");

return -1;

}

//fill socket adress structure with server's address

memset( &addr, 0, sizeof( addr ) );

addr.sun\_family = AF\_UNIX;

strncpy( addr.sun\_path, psz\_serverpath, sizeof( addr.sun\_path ) - 1 );

i\_addr\_len = strlen( addr.sun\_path ) + sizeof( addr.sun\_family );

//创建线程

args s\_args;

s\_args.i\_fd = i\_fd ;

memcpy(s\_args.psz\_clientpath,psz\_clientpath, strlen(psz\_clientpath));

s\_args.addr\_server = addr;

retu = pthread\_create(&tid, NULL, (void \*)send\_thread, (void \*)&s\_args);

printf("retu = %d\n",retu);

puts("create pthread success");

while(1)

{

if ( ( i\_readlen = recvfrom( i\_fd, psz\_rbuf, sizeof( psz\_rbuf ) - 1, 0,

( struct sockaddr \* )&addr, ( socklen\_t \* )&i\_addr\_len ) ) < 0 )

{

perror("main thread write");

return -1;

}

psz\_rbuf[i\_readlen] = '\0';

printf( "receive msg:%s\n", psz\_rbuf );

}

unlink( psz\_clientpath );

return -1;

}

### UDP客户端代码

#include <stdio.h>

#include <sys/un.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <string.h>

#include <unistd.h>

int main(int argc,char \*argv[])

{

int i\_fd = 0;

struct sockaddr\_in addr;

char psz\_serverpath[32] = "./server\_unixsocket\_file";

int i\_addr\_len = sizeof(struct sockaddr\_in);

char psz\_wbuf[32];

char psz\_rbuf[32] = {0};

int i\_readlen = 0;

if ( ( i\_fd = socket( AF\_INET, SOCK\_DGRAM, 0 ) ) < 0 )

{

perror("socket");

return -1;

}

memset( &addr, 0, sizeof( addr ) );

addr.sin\_family = AF\_INET;

addr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

addr.sin\_port = htons(50000);

if ( bind( i\_fd, ( struct sockaddr \* )&addr, i\_addr\_len ) < 0 )

{

perror("bind");

return -1;

}

//fill socket adress structure with server's address

memset( &addr, 0, sizeof( addr ) );

addr.sin\_family = AF\_INET;

addr.sin\_addr.s\_addr = (inet\_addr(argv[1]));

addr.sin\_port = htons(40000);

while(1)

{

scanf("%s",psz\_wbuf);

if ( sendto( i\_fd, psz\_wbuf, strlen( psz\_wbuf ) + 1, 0,

( struct sockaddr \* )&addr, i\_addr\_len ) < 0 )

{

perror( "write" );

return -1;

}

if ( ( i\_readlen = recvfrom( i\_fd, psz\_rbuf, sizeof( psz\_rbuf ) - 1, 0,

( struct sockaddr \* )&addr, ( socklen\_t \* )&i\_addr\_len ) ) < 0 )

{

perror("write");

return -1;

}

psz\_rbuf[i\_readlen] = '\0';

printf( "receive msg:%s\n", psz\_rbuf );

}

return -1;

}