Nat 实验设计报告

中国科学院大学

页

2022 年 5 月 13 日

一、网络地址转换实验

1. 实验内容

网络中的 IP 地址分为公网地址和私网地址两类，当在专用网内部的一些主机本来已经分配到了本地 IP 地址，但又想和因特网上的主机通信（并不需要加密）时，可使用 NAT。。NAT 不仅能解决 IP 地址不足的问题，而且还能够有效地避免来自网络外部的攻击，隐藏并保护网络内部的计算机 nat 设备的主要工作包括：

① 维护私网地址/端口与公网地址/端口的映射关系；

② 对数据包内容进行重写（ Translation），修改 IP 地址/端口等字段，使得数据包在相应网络中有意义。

NAT 的工作场景可以分为私网主机连接到公网服务器（ SNAT）和 私网主机作为服务器（DNAT）两种。在本次实验中，我们需要实现 nat 在这两种情况下不同的功能。我们的测试包括三个部分：

① SNAT 实验：单 nat 下私网主机访问公网服务器；

② DNAT 实验： 公网主机访问私网服务器；

③ 构造一个包含两个 nat 的拓扑，测试主机能否穿过两个 nat 通信。

2. 实验流程

（1）编写实验代码，完成 nat.相关代码实现，本次实验中需要完成其中 get_packet_direction、）do_translation、nat_timeout 、nat_exit 和 parse_config 等函数，make 生成 nat 可执行程序。

（2）完成 SAT 实验：在 n1 上运行 nat 程序，在 h3 上运行 HTTP Server ，在 h1 和 h2 上分别访问 h3。

给定拓扑如图 1：

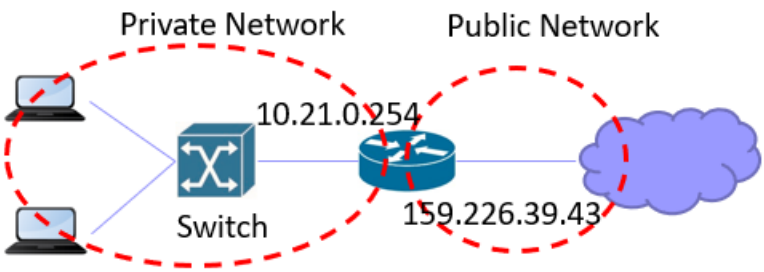


图 1

① 运行给定网络拓扑(nat_topo.py)；

② 在 n1, h1, h2, h3 上运行相应脚本；

③ 在 n1 上运行 nat 程序： n1# ./nat exp1.conf；

④ 在 h3 上运行 HTTP 服务：h3# python ./http_server.py

⑤ 在 h1, h2 上分别访问 h3 的 HTTP 服务

h1# wget http://159.226.39.123:8000

h2# wget http://159.226.39.123:8000

补充测试代码(nat_topo.py)：

```
    n1.cmd('./nat exp1.conf > n1-output.txt 2>&1 &' )
    h3.cmd('python http_server.py > h3-output.txt 2>&1 &')

    for h in (h1, h2):
        h.cmd('wget http://159.226.39.123:8000 > %s-output.txt 2>&1' % h)

    n1.cmd('pkill -SIGTERM nat')
```

（3）完成 DNAT 实验：在 n1 上运行 nat 程序，在 h1 和 h2 上分别运行 HTTP Server，在 h3 上访问 h1 和 h2 ；

① 运行给定网络拓扑(nat_topo.py)；

② 在 n1, h1, h2, h3 上运行相应脚本；

③ 在 n1 上运行 nat 程序： n1# ./nat exp2.conf；

④ 在 h1、h2 上运行 HTTP 服务：h1/h2# python ./http_server.py

⑤ 在 h3 上分别访问 h1、h2 的 HTTP 服务

h3# wget http://159.226.39.43:8000

h3# wget http://159.226.39.43:8001

补充测试代码(nat_topo.py)：

```
    n1.cmd('./nat exp2.conf > n1-output.txt 2>&1 &' )

    h1.cmd('python http_server.py > h1-output.txt 2>&1 &')
    h2.cmd('python http_server.py > h2-output.txt 2>&1 &')

    h3.cmd('wget http://159.226.39.43:8000 > h3-output.txt 2>&1')
    h3.cmd('wget http://159.226.39.43:8001 > h3-output.txt 2>&1')

    n1.cmd('pkill -SIGTERM nat')
```

（3）完成 SNAT 实验构造一个包含两个 nat 的拓扑，测试主机能否穿过两个 nat 通信。

手动构造一个包含两个 nat 的拓扑 h1 <-> n1 <-> n2 <-> h2，类比 exp1.conf 完成 SNAT 和 DNAT 的规则转换 exp3_1.config 和 exp3_2c=.config。

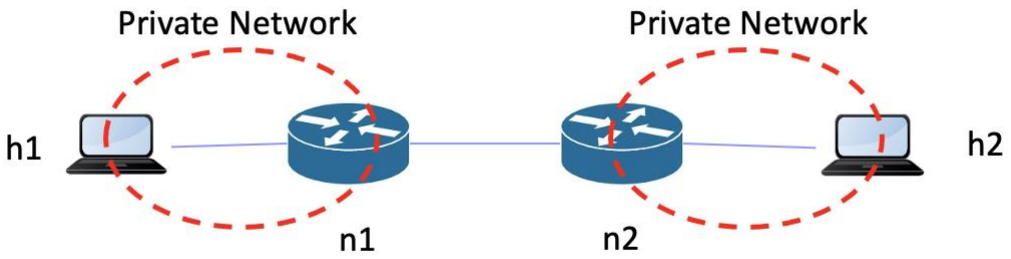


图 2

节点 n1 作为 SNAT， n2 作为 DNAT，主机 h2 提供 HTTP 服务，主机 h1 穿过两个 nat 连接到 h2 并获取相应页面

拓扑：

```
class NATTopo(Topo):
    def build(self):
        h1 = self.addHost('h1')
        h2 = self.addHost('h2')
        n1 = self.addHost('n1')
        n2 = self.addHost('n2')

        self.addLink(h1, n1)
        self.addLink(n2, h2)
```

```
        self.addLink(n1, n2)
```

测试代码：

```
    n1.cmd('./nat exp3_1.conf &' )
    n2.cmd('./nat exp3_2.conf &' )
    h2.cmd('python http_server.py > h2-output.txt 2>&1 &')

    h1.cmd('wget http://159.226.39.123:8000')

    n1.cmd('pkill -SIGTERM nat')
    n2.cmd('pkill -SIGTERM nat')
```

3. 实验设计

（1）确定 pkt 方向，比较 packet 的地址端口和匹配的路由表项对应的端口 即可判定其方向. 1. 当源地址为内部地址，且目的地址为外部地址时，方向为 DIR_OUT；2. 当源地址为外部地址，且目的地址为 external_iface 地址时，方向为 DIR_I。

```
static int get_packet_direction(char *packet)
{
    //fprintf(stdout, "TODO: determine the direction of this packet.\n");
    struct iphdr *ihr = packet_to_ip_hdr(packet);
    rt_entry_t *sentry = longest_prefix_match(ntohl(ihr->saddr));
    rt_entry_t *dentry = longest_prefix_match(ntohl(ihr->daddr));
    if(sentry->iface==nat.internal_iface &&
dentry->iface==nat.external_iface)
        return DIR_OUT;
    if(sentry->iface==nat.external_iface &&
ntohl(ihr->daddr)==nat.external_iface->ip)
        return DIR_IN;
    return DIR_INVALID;
}
```

（2）pkt 的重写，完成对合法数据包的处理，进行(internal_ip, internal_port) <-> (external_ip, external_port)之间的转换，填充映射表。具体说来，若该数据包在 NAT 中有对应连接映射，根据方向重写, 发包。若该数据包暂无连接，尝试添加连接。

若该数据包的方向为 DIR_OUT，为该 TCP 连接的第一个数据包（请求连接数据包），NAT 中没有对应连接映射 (SNAT)；该数据包的方向为 DIR_IN，为该 TCP 连接的第一个数据包，NAT 中没有对应连接映射，但有对应处理规则 (DNAT)。确定映射关系时可以先用(rmt_ip, rmt_port) 定位到一组映射结构（链表），再根据数据包方向，决定用(rmt_ip, rmt_port) + (int_ip, int_port) 还是(rmt_ip, rmt_port) +(ext_ip, ext_port) 来确定唯一的映射结构，然后添加映射后更新检验和将数据包从相应端口发出。如果无法添加，则丢弃当前数据包，发送

ICMP 主机不可达。处理过程如下图：

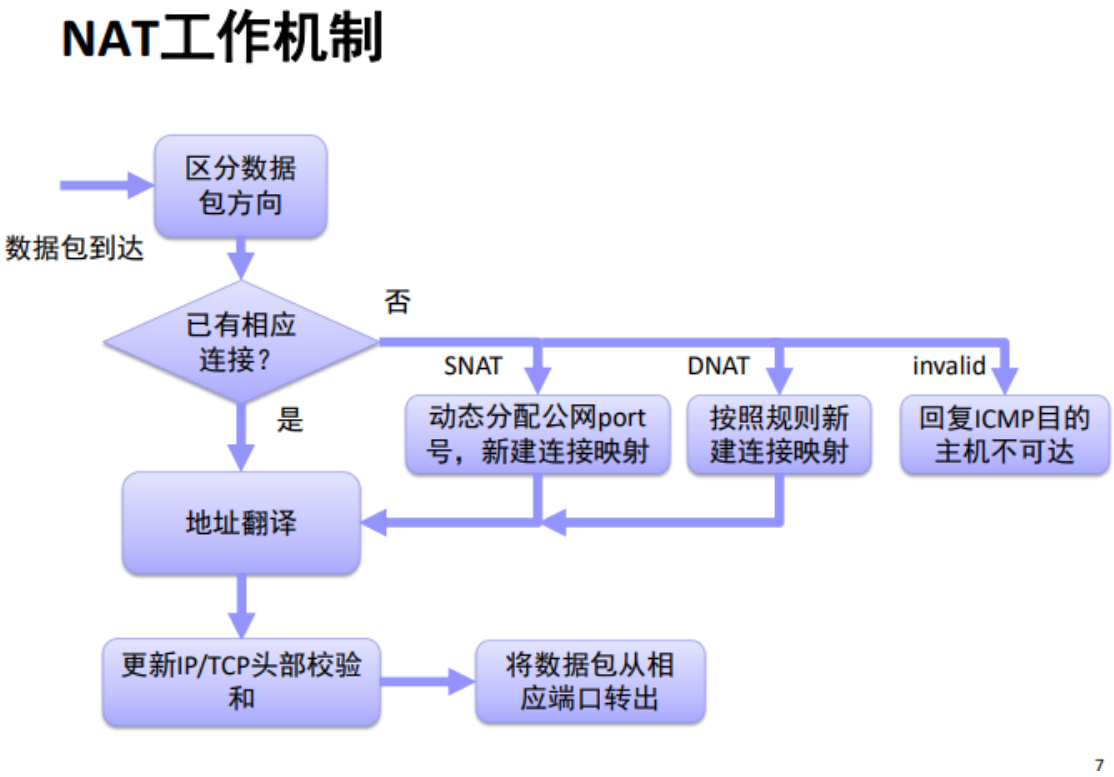


图 3

```
void do_translation(iface_info_t *iface, char *packet, int len, int dir)
{
    // fprintf(stdout, "TODO: do translation for this packet.\n");
    pthread_mutex_lock(&nat.lock);

    struct iphdr *ihr = packet_to_ip_hdr(packet);
    struct tcphdr *thr = packet_to_tcp_hdr(packet);
    u32 rm_ip = dir == DIR_IN ? ntohl(ihr->saddr) : ntohl(ihr->daddr);
    u16 rm_port = dir == DIR_IN ? ntohs(thr->sport) : ntohs(thr->dport);
    u32 temp_ip = dir == DIR_IN ? ntohl(ihr->daddr) : ntohl(ihr->saddr);
    u16 temp_port = dir == DIR_IN ? ntohs(thr->dport) : ntohs(thr->sport);
    u8 id = hash8((char *)&rm_ip, 4);
    int exist = 0, found = 0;
    struct nat_mapping *entry = NULL;
    if (dir == DIR_OUT)
    {
        list_for_each_entry(entry, &nat.nat_mapping_list[id], list)
        {
            if (entry->remote_ip == rm_ip && entry->remote_port == rm_port
&& entry->internal_ip == temp_ip && entry->internal_port == temp_port)
            {
                exist = 1;
                break;
```

```
            }
        }
        if (!exist)
        {
            entry = malloc(sizeof(struct nat_mapping));
            entry->remote_ip = rm_ip;
            entry->remote_port = rm_port;
            entry->internal_ip = temp_ip;
            entry->internal_port = temp_port;
            entry->external_ip = nat.external_iface->ip;
            for (int i = NAT_PORT_MIN; i < NAT_PORT_MAX; i++)
            {
                if (!nat.assigned_ports[i])
                {
                    nat.assigned_ports[i] = 1;
                    entry->external_port = i;
                    found = 1;
                    break;
                }
            }
            if (!found)
            {
                free(packet);
                free(entry);
                return;
            }
            list_add_tail(&entry->list, &nat.nat_mapping_list[id]);
        }
        ihr->saddr = htonl(entry->external_ip);
        thr->sport = htons(entry->external_port);
        entry->conn.internal_fin = thr->flags & TCP_FIN;
        entry->conn.internal_seq_end = tcp_seq_end(ihr, thr);
        entry->conn.internal_ack = thr->flags & TCP_ACK;
    }
    else
    { // DIR_IN
        list_for_each_entry(entry, &nat.nat_mapping_list[id], list)
        {
            if (entry->remote_ip == rm_ip && entry->remote_port == rm_port
&& entry->external_ip == temp_ip && entry->external_port == temp_port)
            {
                exist = 1;
                break;
            }
```

```
        }
        if (!exist)
        {
            struct dnat_rule *rule = NULL;
            list_for_each_entry(rule, &nat.rules, list)
            {
                if (rule->external_ip == temp_ip && rule->external_port ==
temp_port)
                {
                    entry = malloc(sizeof(struct nat_mapping));
                    entry->remote_ip = rm_ip;
                    entry->remote_port = rm_port;
                    entry->external_ip = temp_ip;
                    entry->external_port = temp_port;
                    entry->internal_ip = rule->internal_ip;
                    entry->internal_port = rule->internal_port;
                    list_add_tail(&entry->list,
&nat.nat_mapping_list[id]);
                    found = 1;
                    break;
                }
            }
            if (!found)
            {
                free(packet);
                free(entry);
                return;
            }
        }
        ihr->daddr = htonl(entry->internal_ip);
        thr->dport = htons(entry->internal_port);
        entry->conn.external_fin = thr->flags & TCP_FIN;
        entry->conn.external_seq_end = tcp_seq_end(ihr, thr);
        entry->conn.external_ack = thr->flags & TCP_ACK;
    }
    entry->update_time = time(NULL);
    thr->checksum = tcp_checksum(ihr, thr);
    ihr->checksum = ip_checksum(ihr);

    pthread_mutex_unlock(&nat.lock);

    ip_send_packet(packet, len);
}
```

（3）NAT 老化：对认为已经结束的连接进行老化操作：

①双方都已发送 FIN 且回复相应 ACK 的连接，一方发送 RST 包的连接，可以直接回收端口号以及相关内存空间。

②双方已经超过 60 秒未传输数据的连接，认为其已经传输结束，可以回收端口号以及相关内存空间。

```
void *nat_timeout()
{
    while (1) {
        // fprintf(stdout, "TODO: sweep finished flows periodically.\n");
        sleep(1);
        pthread_mutex_lock(&nat.lock);
        struct nat_mapping *entry, *q;
        for (int i = 0; i < HASH_8BITS; i++)
        {
            list_for_each_entry_safe(entry, q, &nat.nat_mapping_list[i],
list)
            {
                if (is_flow_finished(&entry->conn) || time(NULL) -
entry->update_time >= TCP_ESTABLISHED_TIMEOUT)
                {
                    list_delete_entry(&entry->list);
                    nat.assigned_ports[entry->external_port] = 0;
                    free(entry);
                }
            }
        }
        pthread_mutex_unlock(&nat.lock);
    }
    return NULL;
}
```

（4）根据 config文件中读取的每一行字符串，分别配置 external-iface ，internal-iface 和 DNAT Rules。

```
int parse_config(const char *filename)
{
    // fprintf(stdout, "TODO: parse config file, including i-iface, e-iface
(and dnat-rules if existing).\n");
    FILE *fd = fopen(filename, "r");
    if (!fd)
```

```
        return -1;
    char buf[MAX_LINE];
    for (int i = 0; i < 2; i++)
    {
        fgets(buf, MAX_LINE, fd);
        char *face = strtok(buf, " ");
        face = strtok(NULL, "\n");
        iface_info_t *iface = if_name_to_iface(face);
        if (!iface)
        {
            log(ERROR, "get iface error");
            exit(0);
        }
        if (i == 0)
            nat.internal_iface = iface;
        else
            nat.external_iface = iface;
    }

    int index[10], c;
    while (1)
    {
        while ((c = fgetc(fd)) != ':' && c != EOF)
            ;
        if (c == EOF)
            return 0;
        if (fscanf(fd, " %d.%d.%d.%d:%d -> %d.%d.%d.%d:%d", index, index +
1, index + 2, index + 3, index + 4, index + 5, index + 6, index + 7, index
+ 8, index + 9) == 10)
        {
            unsigned ip1 = (((((index[0] << 8) + index[1]) << 8) + index[2])
<< 8) + index[3];
            unsigned port1 = index[4];
            unsigned ip2 = (((((index[5] << 8) + index[6]) << 8) + index[7])
<< 8) + index[8];
            unsigned port2 = index[9];
            struct dnat_rule *rule = malloc(sizeof(struct dnat_rule));
            rule->external_ip = ip1;
            rule->internal_ip = ip2;
            rule->external_port = (u16)port1;
            rule->internal_port = (u16)port2;
            list_add_tail(&rule->list, &nat.rules);
        }
    }
```

```
    return 0;
}
```

4. 实验结果

(1)SNAT 实验：

h1->h3:

```
<!doctype html>
<html>
    <head> <meta charset="utf-8">
        <title>Network IP Address</title>
    </head>
    <body>
            My IP is: 159.226.39.123
            Remote IP is: 159.226.39.43
        </body>
</html>
```

h2->h3:

```
<!doctype html>
<html>
    <head> <meta charset="utf-8">
        <title>Network IP Address</title>
    </head>
    <body>
            My IP is: 159.226.39.123
            Remote IP is: 159.226.39.43
        </body>
</html>
```

结果显示，从 h3 角度来看，h1 和 h2 的 ip 一致为 159.226.39.43 说明 NAT 转换成功。

(2)DNAT 实验

h3->h1

```
<!doctype html>
<html>
    <head> <meta charset="utf-8">
        <title>Network IP Address</title>
    </head>
    <body>
            My IP is: 10.21.0.1
```

```
            Remote IP is: 159.226.39.123
        </body>
</html>
```

h3->h2

```
<!doctype html>
<html>
    <head> <meta charset="utf-8">
        <title>Network IP Address</title>
    </head>
    <body>
            My IP is: 10.21.0.2
            Remote IP is: 159.226.39.123
        </body>
</html>
```

h1 发回 IP 为 10.21.0.1，h1 发回 IP 为 10.21.0.2，它们发回到 h3 自己没有经过 nat 转换的私网 ip，结果符合预期。

（3）双 NAT 穿透实验

```
<!doctype html>
<html>
    <head> <meta charset="utf-8">
        <title>Network IP Address</title>
    </head>
    <body>
            My IP is: 10.21.0.2
            Remote IP is: 159.226.39.43
        </body>
</html>
```

h2 正常返回页面，h2ip 为私网 ip10.21.0.2，h2 认为的 h1 地址为公网 ip159.226.39.43 符合实验预期结果。

5. 思考题

1. 实验中的 NAT 系统可以很容易实现支持 UDP 协议，现实网络中 NAT 还需要对 ICMP 进行地址翻译，请调研说明 NAT 系统如何支持 ICMP 协议

答：

ICMP 协议的报文如下：

类型（Type）	代码（Code）	检验和（Checksum）
标识符（Identifier）	序列号（Sequence number）	

源IP	源端口	目的IP	目的端口
h1 IP	Type + Code	h2 IP	Identifer

首先 ICMP 报文可以分为两类，询问报文、差错报告报文。

（1）

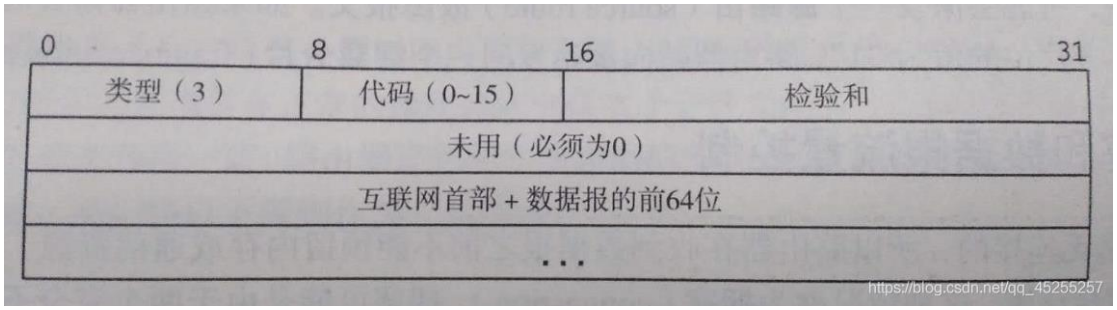
对于询问报文，其头部的最后 4 字节，前 2 字节为 identifier，后 2 字节为 sequence。Identifier 为主机标识，每台主机有唯一固定的标识，且各主机的标识不同。Sequence 为序列号，每个 ICMP包都有不同的序列号。因此可以使用 Identifier 或者 Identifier+Sequence 作为标识符，代替端口号。使用<IP, Identifier>来建立唯一映射。假设主机结点 h1 要 ping 结点 h2，h1 发送 ICMP报文， h1会根据报文头部中包含的类型信息 Type 和代码信息 Code 生成源端口号，发出 ICMP 询问报文时，NAT 以其 Identifier 作为标识，并分配 NAT 的一个端口与之建立映射，添加相应的 nat表项：

内网IP	内网端口	协议	外网IP	外网端口
h1 IP	Type + Code	ICMP	h1 public_IP	IDENTIFIER（随机分配端口号）

在服务器收到 ICMP 请求后，生成 ICMP 响应报文，响应报文中的（Type+Code）会作为源端口，IDENTIFIER 作为目的端口。

源IP	源端口	目的IP	目的端口
h2 IP	Type + Code	h1 public_IP	IDENTIFIER

但根据报文格式它将 ICMP 头部的 Identifier 与 Sequence 全部抹为 0。



当差错报文回到 NAT 时，它发现 Identifier 字段全 0，无法根据 Identifier 查找映射表。一种解决方案是 NAT 此时需要检查 ICMP 报文数据段。根据 ICMP 协议规则，当差错报文产生时，出错 IP 数据包的 IP 头部以及接下来的 8 字节会填入数据部分。如果出错数据包是 ICMP 询问报文，IP 头部接下来的 8 字节正好是ＩＣＭＰ头部，可以从其中找到原本的 Identifier 与 Sequence，根据该字段查找映射表就能像上面那样找到内网主机。如果出错数据包是 TCP、UDP 数据包，那么可以从那 8 字节中提取到 NAT 发出时的源端口，根据端口号查找映射表找到内网主机。

2．给定一个有公网地址的服务器和两个处于不同内网的主机，如何让两个内网主机建立 TCP 连接并进行数据传输。（提示：不需要 DNAT 机制）。

答：

这种情况是要两个处于不同 nat网络下的结点建立直接连接，可以称为 nat 穿透。NAT 穿透，简单地讲就是要让处于不同 NAT 网络下的两个节点(Peer)建立直接连接，只知道自己的内网地址是肯定不行的，他们需要知道对方的公网 IP 和端口，然后双方互相向对方发送数据包，从而建立起连接。整个流程可以看做两个关键步骤：

1.发现自己的公网 IP 和 Port；

2.将自己的 IP 和 Port 共享给对方

其中，第二步，我们可以简单地通过一个第三方服务器来交换双方的 IP 和 Port，但是第一步就比较困难，我们不妨根据不同类型的 NAT 的特点，分别看看在不同的 NAT 类型下，怎样才能拿到一个可供通讯的公网 IP 和 Port。

基本思路如下：

1.给定具有公网 IP 和 Port 的服务(Server 1)

2.(Client)发送一个数据包给这个公网服务(Server1)

3.(Server1)通过解析 IP 协议包，就能得知(Client)的公网地(eAddr:ePort) ，服务器收到主机 1 和主机 2 的连接后，知道 1 与 2 的公网地址和 NAT 分配给它们的端口号.

4.(Server1)将该公网地址(eAddr:ePort)回传给 Client1 和 2

5.这样两个不同的节点 Client1 和 Client2 通过第三方服务器交换公网地址(eAddr1:ePort1)(eAddr2:ePort2)以及对方的 NAT 分配的端口号从而避开了 DNAT 的解析。

6.自由地进行通讯

参考资料：

【1】ICMP 报文如何通过 NAT 来地址转换。

【2】差错报文 ICMP。