A Complementary Approach for Transparent NAT Connectivity

Lucas Clemente Vella, Lásaro Camargos, and Pedro Frosi Rosa
Computing Faculty
Federal University of Uberlândia
Minas Gerais, Brazil
Email: lvella@comp.ufu.br, {lasaro, frosi}@facom.ufu.br

***Abstract*—NAT has been responsible for the survival of IPv4 and in essence should not be left out in IPv6. NAT are virtually transparent to client-server applications that generally do not require special configuration to work properly. However, P2P applications are responsible for generating about half of Internet traffic and require special settings on home routers to support outside connections. This paper presents a complement to the UPnP IGD protocol, including changes in the core of the Linux operating system, to make NAT traversal transparent to home and small office users in the use of P2P applications or in providing services to the outside world. Our approach overcomes some of the major limitations of NAT solutions, by extending existing standard behaviours. In the proposed solution, current applications need no changes once the transparency is provided through the improvement made in the network related system calls. Tests using a reference implementation and network applications supports the feasibility of the approach.**

***Keywords*-Network Address Translation; NAT traversal; UPnP; home networks.**

I. Introduction

Network Address Translation (NAT) is a commonly used tool to bridge Local Area Networks (LAN) to the Internet, effectively allowing multiple clients to share one valid Internet link. In the usual setup, there is a single public IP address, which can be reached from outside the LAN, and multiple private IP addresses used by the local clients to communicate among themselves. When a local client tries to reach an Internet host, the device in the role of Internet gateway (usually, a small router) performs the necessary address translations, so the message is transparently forwarded via the single public address.

For the network usage pattern of conventional client applications, where the client always initiates the communication with a server, NAT is transparent. Once the first contact is made from inside the LAN, the Internet gateway is able to automatically handle the responses from the server, turning the translation step invisible to most ordinary TCP/IP clients.

While restricted to servers and datacenters in the dawn of the Internet, programs who wait for incoming connections are increasingly more frequent to the users of NAT, namely the home and office Internet users, specially with the great popularity of Peer-to-Peer (P2P) software. To these programs, NAT is not totally transparent and requires explicit network configuration in the gateway to be able to receive external requests.

In order to let P2P and server applications receive connection from the Internet in the presence of NAT, a technique known as NAT traversal must be implemented. The burden of implementing NAT traversal, however, is placed either on the user, that must have the expertise to configure his equipment, or in the software developer, who must support the protocol to configure the router, increasing the development cost.

We argue that NAT transparency should be taken one step further, and the role of traversing NAT should be pushed inside the network stack, being performed by the operating system. In this paper we present a proof of concept extension to the Linux operating system to provide transparent NAT traversal through the standard network API.

In Section II some related work is reviewed. In Section III the proposed integration of NAT traversal with the network stack is explained. In Section IV the reference implementation is detailed along with its protocols. Section V enumerates the test cases for the implementation and its results. Finally, Section VI draws some conclusions and proposes directions on which this work might be improved.

II. Background

With the usage of NAT in home and office gateways, services provided internally are not readily accessible from the outside. This problem has spawn a number of solutions, that targets various levels of abstraction in the network stack.

A. Other Home Network Solutions

The HomeDNS [1] approach works on the level of resource names, with emphasis on HTTP services, which are common for multimedia streaming applications. It provides a dynamic Domain Name Service (DNS) solution that is able to reference, from the external network, the multiple services available inside the home network, which in turn are used to build the URLs for the HTTP requests. This work concerns itself with augmenting to the Internet the reach of HTTP services targeted at the LAN. Differently, ours addresses connectivity issues of any TCP or UDP applications already targeted to the Internet.

Next to HomeDNS, the solution presented in [2] provides means to expose local services of home networks in remote

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-183-0

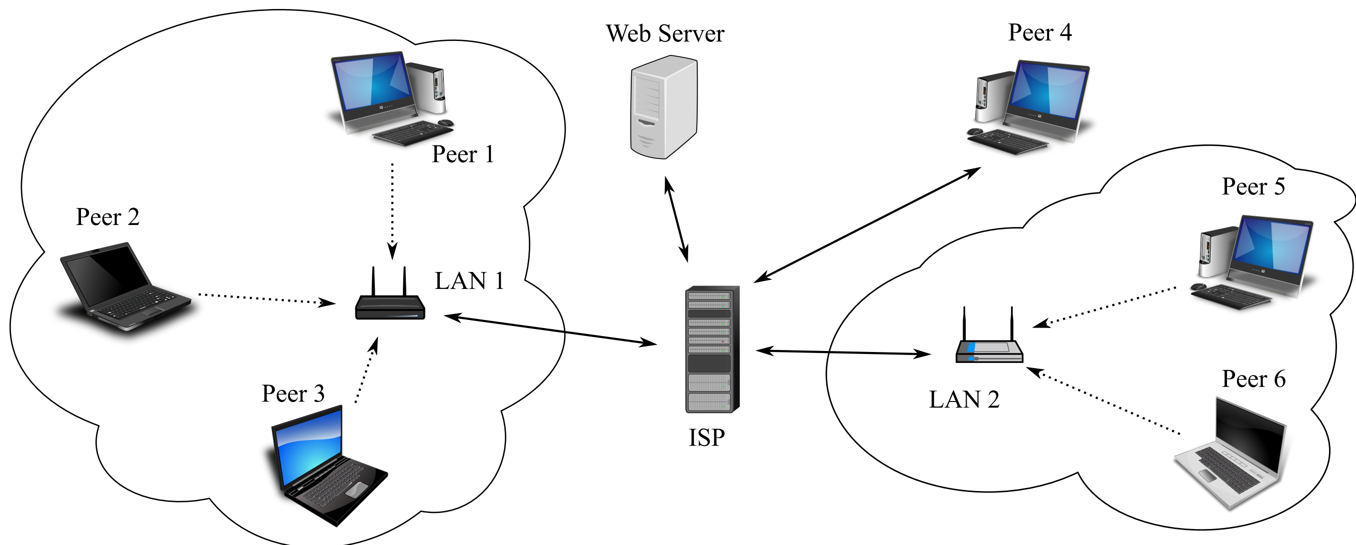


Figure 1. Reachability among peers behind NAT

guest networks. The work is specially focused on UPnP media servers and services targeted to the inside network, which access is intentionally restricted by the local network and requires tight control on its exposure. The difference of our solution it that ours is targeted to services that *should* be externally exposed, but are not due to the network topology.

In [3], some requirements of home gateways are identified. The work tries to address the issues found at home networks by designing a new home gateway, that is, a replacement for the gateways that we currently find in the market. It would provide the means to access internal services from the Internet, but would not dive into details on how it should be done with existing applications behind NAT, except that saying that UPnP IGD protocol could be, in the future, used to perform NAT traversal. Our approach is more pragmatic in that we solve only one problem, namely, NAT traversal, and do so without pushing for new equipment.

B. Internet Protocol v.6 (IPv6)

It has become a common practice among Internet Service Providers (ISP) to assign a single IP addresses to small customers, such as households and small offices. While the practice is somewhat justified by the IPv4 address exhaustion problem, it is possible that even with the eventual widespread adoption of IPv6 [4] (which eliminates the exhaustion problem) the ISP may still assign single addresses to their Small Office/Home Office (SOHO) customers. The Brazilian Internet Steering Committee (CGI.br) states, in a website dedicated to IPv6 adoption, that this is an acceptable practice [5]. In such a scenario, if multiple clients are to connect to the Internet, then there must be a way to share one valid Internet link with these clients; the Network Address Translation has become the *de facto* standard for doing so.

If NAT is still to be useful in an IPv6 world, then NAT drawbacks will persist, and the solution presented in this work will still be relevant, even without the address exhaustion problem.

C. NAT Transparency

NAT is an abstraction. One network element, the Internet gateway, is aware of the abstraction and hides the address translation complexity behind the standard interface of UDP/TCP and IP. Network applications unaware of NAT are functional as long as they do not need to expose any service to the external network. Upon this need, the effect of being behind NAT is felt.

Consider the Fig. 1, depicting a directed graph whose reachability from one node to another means the ability of this node to initiate a communication with the other via the Internet. While Peer 1 needs are nothing but to access the Web Server, the NAT taking place on LAN 1 router will not be felt. There is no problem, either, if Peer 2 wants to download a file via some P2P application from Peer 4, because, since it is a initiative from Peer 2, it will be able to open the needed TCP connection to Peer 4. The problem occurs on the opposite. By its own initiative, Peer 4 will not be able to establish the TCP connection of the P2P application to Peer 2, because the farthest Peer 4 can address is the LAN 1 router. Worse, Peers 1, 2 and 3 are invisible to Peer 5 and 6 (and *vice-versa*) on a P2P environment. Same problem if Peer 3 tries to join a game session hosted by Peer 6, because it has nowhere to send the first UDP packet that would let it into the game, once Peer 6 IP address is masqueraded by LAN 2 router.

To counter these problems, the router must be configured to follow-up TCP connection attempts or unknown UDP packets to specific hosts and ports inside the network, and

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-183-0

this host is the one running the application responsible to deal with the request. One should note that this is not simply IP routing, since the destination address of the IP datagram is the one of the gateway itself (which is public), not the one of the application's host (which is private).

Listening applications are those that either expect for the incoming of TCP connections or the first contact from the remote UDP peers. The need of some listening applications to be externally reachable started to weaken the NAT abstraction. The widespread use of NAT made it a concern to users and developers of those applications. It is now common to find applications implementing protocols to configure automatically gateway equipments on regarding NAT. It is also common to find advanced P2P users aware of the issue and experienced in manual NAT traversing setup.

The current scenario is that there are NAT-aware users running NAT-aware applications on top of *unaware* operating systems communicating through an Internet gateway implementing the NAT technique; whose design goal is to be invisible. As stated by NAT's RFC:

> "Basic Network Address Translation or Basic NAT is a method by which IP addresses are mapped from one group to another, transparent to end users." [6]

D. UPnP and NAT Traversing

Listening applications' developers found in the protocol commonly known as *UPnP* the means to hide the complexity of NAT traversing from their users. UPnP, which stands for Universal Plug and Play, is a set of protocols for discovery and automatic configuration of home networks. Initially developed by Microsoft [7], it is now maintained by the industry consortium named UPnP Forum [8].

UPnP protocols are built on top of HTTP and its UDP version, HTTPU, where its messages are XML based. As such, UPnP protocols are application level protocols, with relatively high overhead and complexity. This design choice renders unpractical the implementation of the protocols in low-level software, like operating system kernels, because the software stack providing those base technologies are often unavailable at this level.

Among the provided protocols, there is the UPnP Internet Gateway Device Protocol (UPnP IGD), whose goal is to control and configure small network gateways. Despite numerous security flaws in many and popular implementations [7], [9], [10], it became the most well supported NAT traversing mechanism by applications and routers. The competing NAT-PMP [11] protocol, despite being much simpler, is young and still does not have the availability of UPnP IGD among off-the-shelf devices.

UPnP IGD Protocol plays an important role in SOHO local networks, because it is the protocol being simply referred as UPnP by the listening applications implementing it, rendering it one of the most common NAT traversal techniques available.

The term UPnP is also commonly used to refer to a functionality present in networked multimedia applications. This usage of the term is a shorthand for UPnP Audio/Video, which is another set of protocols developed by the same UPnP initiative targeted to multimedia streaming, but is otherwise unrelated to the UPnP IGD, which is the one relevant to this work.

When using UPnP, the listening applications inside the network shares the same TCP or UDP address space, allocated in the gateway. Thus, if one application exposes one TCP port to the Internet, another LAN application willing to listen on the Internet must choose another port. This usually does not results in a port scarcity issue, since UPnP is meant to be used in small home and office networks. The 16 bit address of a port is often enough to serve all the Internet applications of these small networks.

The same port can not be shared between applications, as it is done with port multiplexing by some NAT devices. Port multiplexing uses extra previously known information, such as source port and address, to demultiplex an incoming message. When listening to the new connections, there is no such previous information available when an unknown packet arrives, thus the only mean to identify the intended receiver inside the network is the port.

Not only UPnP IGD, the *de facto* standard technique employed as NAT traversal, but also NAT-PMP and other techniques have the drawback of needing support in a per-application basis, a cost paid by the developer. Not all applications have the UPnP feature, especially the legacy ones. Developers may lack the resources to implement the feature in a project, but even if they do, it is an extra functional requirement to be taken into account.

III. The Proposed Approach

In order to make NAT transparent to listening applications, they must be able to use the bare TCP/IP interface of the operating system to wait for contact from the outside network, in the same way client applications may just connect. There should be no extra cost in the development of listening application related specifically to NAT traversal. Users and developers of client applications need not to worry if the host is behind a NAT, neither should listening applications' developers and users.

To achieve this goal, operating systems must be aware of the NAT issue. Since they are already responsible for interfacing with the *sockets* API, the one used by the applications to reach the TCP/IP functionality, it has all the means to automatically manage the gateway in place of the application or, in worse cases, the user.

A. Connection and Disconnection

Upon a `bind()` system call made on a TCP or UDP socket, the operating system may use the same protocols that applications explicitly use to forward the ports on the

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-183-0

gateway to themselves. Since UPnP or other NAT traversal technique is to be implemented by the operating system, this burden is then taken from the application. In the same way the operating system abstracts away the complexities of TCP/IP, it shall also take care of any NAT traversal employed.

When the execution flow is returned to the application by `bind()`, the operating system shall have already attempted to forward the requested port on the router. The return value is dependant on the outcome of this attempt. In case the port is being used by another host, port forwarding operation fails and the operating system must also fail the system call. *This way* the application can perform its default behavior in case of port already in use.

Upon the closing of the socket, either explicit or by process exit, the operating system must automatically remove the port association in the gateway. Unlike the creation of a port forwarding, the removal operation shall be performed asynchronously. Since port forwarding removal can not affect the outcome of the `close()` or `exit()` operation, there is no need to synchronize them.

Applications that automatically forward ports may fail to cleanup their associations on the gateway when no longer needed. It may be so either in case of application crash or because of bad implementation of the port forwarding protocol. A beneficial side effect of our approach is that, since the port forwarding is automatically managed by the network infrastructure, the needed cleanup is performed as long as the system is running, even if the application crashes.

B. Security

The proposed automatic management of NAT traversal targets end-user applications. Due to security concerns, it is important that system daemons and servers which require fine administrative control, such as FTP, HTTP, Telnet and SSH are *not* automatically exposed to the external network. For this reason, associations made by processes on UDP or TCP ports bellow 1024 should be filtered and not configured on the router. The services previously mentioned are by default bound to these ports, and privileges given by the system administrator are needed to use them. Since no common user's application shall use the privileged ports, the activity on them is out of our scope.

Applications may also choose what IP address available in the system to use when binding a socket. As a placeholder meaning *any address available*, an application may use the fake address 0.0.0.0 (aliased as `INADDR_ANY` in POSIX systems). Applications that choose to bind to specific interfaces usually know their intended peers and hold fine control of the network topology, often being manually configured on what IP interfaces to use. Upon this case, we consider that association not generic enough to be automatically forwarded by the gateway, even if the specified address is the route to the gateway. Internet applications do not try to restrict their reachability. If they are to be seen in the Internet, their logical choice of IP interface is *any* (or 0.0.0.0). Should a program or user try to control the connectivity by choosing what interface to use, then we shall not take this control by automatically exposing it on the external network.

One may question that, being the task of opening ports on the router automatic, the system would be more vulnerable to viruses and malicious software. With our approach implemented, a virus would be able to expose the system to the external network, when otherwise the system would only be exposed to the internal network. This is not indeed the case, and a virus might well find its route through a NAT in the same way a legitimate application could do, simply by implementing the same protocol that we use, with no different clearance.

IV. The Implementation

Our reference implementation consists of an extension to the Linux kernel, together with an ancillary user space daemon to handle the gateway configuring protocol [12], resembling a microkernel architecture where system functions are performed by isolated special processes. It only affects applications using sockets API to access to either UDP or TCP on top of IPv4.

This is fundamentally different from other UPnP IGD solutions for GNU/Linux, such as LinuxIGD [13], PseudoICSD [14] and MiniUPnPd [15], in a way that these packages are just plain userspace applications that implements the server part of the protocol, *i.e.* they are used to turn a GNU/Linux NAT router into a UPnP enabled gateway. The focus here is to implement the client side operated by the kernel, and UPnP server implementation is out of scope.

We choose Linux because of its popularity and its source code availability that allows us to do the kind of low-level modification needed. The protocol we use to traverse NAT is the UPnP IGD Protocol, because it is the most well supported by small routers.

A. Changes to Kernel

Inside the kernel, every call to the POSIX system call `bind()` performed on a TCP/UDP IPv4 socket is intercepted. The calls made to privileged ports or to specific IP interfaces are ignored by the automatic port forwarding mechanism. Otherwise, packets to the given port arriving at the Internet gateway must be forwarded to our host. The kernel delivers the bind request to the helper user space daemon responsible for setting up the forwarding. The calling process is put into a sleep state while awaiting the answer from the user space daemon. When the answer arrives, the process is awaken and deals with it. The `bind()` system call may then resume, failing or succeeding in according to the answer received.

To avoid a race condition, the port is preallocated internally before the control is given to the daemon. Otherwise, at least one scenario could lead to an inconsistent state. Consider it: process A tries to bind to address 0.0.0.0 on TCP port

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-183-0

6881, and is put to sleep while awaits the answer from the daemon, then process B tries to bind to address 127.0.0.1 on TCP port 6881. If the TCP port 6881 was not preallocated to process A, the bind on process B succeeds before process A receives its answer from userspace. When A receives the answer, the `bind()` call will no longer be able to succeed as the port is already in use, but it would have already been successfully configured on the gateway for a process that can no longer answer on it.

Linux provides a number of different ways of communicating between kernel space and user space. In order to pass the bind request on to the daemon, we choose to use the Netlink protocol. Netlink is a Linux specific protocol on top of sockets API and network stack, meaning the applications can use it through the usual socket related system calls. It is not a true network protocol, processes may use it only to communicate with the kernel or other processes in the same host.

Netlink was chosen for the sake of simplicity. It is very easy to add a custom protocol on top of Netlink, providing a well definite interface for user level programs, as well as for kernel code. For our purpose, we defined a new protocol called `NETLINK_NAT_PASS`.

Other kernel ⇔ userspace communication methods are not as fit for our purpose as it is Netlink. For instance: we can not use system calls because they are unidirectional, and unlike Netlink, requests can not be sent from kernel to userspace as required by our architecture. Also, unlike *procfs* [16], *sysfs* [17] and other similar file based interfaces, Netlink require no changes on the filesystem, since it have its own namespace.

The processes awaiting for the daemon are placed in a linked list, that is traversed when an answer is issued by the daemon. There is no explicit guarantee that the first made request will be the first answered by the daemon, but that is the likely scenario, with no indications on how it could be otherwise. Since process are queued in the list in the same order they are sent to the daemon, when an answer arrives it will probably be referent to the first process in the list, making the search practically constant.

B. User Space Daemon

Because of the complexity of UPnP IGD Protocol, being a high-level application protocol on top of web services and HTTP, we choose to use it from user space instead of directly inside the kernel. The daemon we called *Natbinder* is responsible for controling the gateway via UPnP IGD. To build this daemon we used the UPnP IGD Protocol implementation from the *MiniUPnPc* routines library [15].

Upon startup, the daemon searches the network for some UPnP enabled Internet Gateway Devices and gets its local host private IP address. This address is used to construct the UPnP requests.

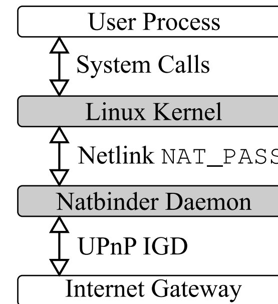


Figure 2. Logical communication stack

After verifying that it is able to reach the gateway, the daemon registers itself on the broadcast channel 0 of the `NETLINK_NAT_PASS` protocol with a Netlink socket. In this channel it will listen for kernel messages regarding IPv4 binding activities of the processes.

On a `bind()` attempt by a process, the action *AddPortMapping* is issued to the gateway. Among the responses specified by the UPnP standard [18] we may receive, two are of particular interest: code 0, meaning success and code 718 (*ConflictInMappingEntry*), meaning that the port requested by the process was already in use by another host. In those cases, the answers given to the kernel are, respectively, to proceed successfully or to fail the bind.

In case we receive a different answer, it is treated as an exceptional condition, which the system is unprepared to handle and unable to further help the binding process or the user. In this case, the message sent back to kernel is that the daemon ignored the bind request. The practical effect is the same as processed successfully, since forbidding the process to use the port will do no good in this case. The exceptional condition is logged by the daemon for manual investigation of the system administrator.

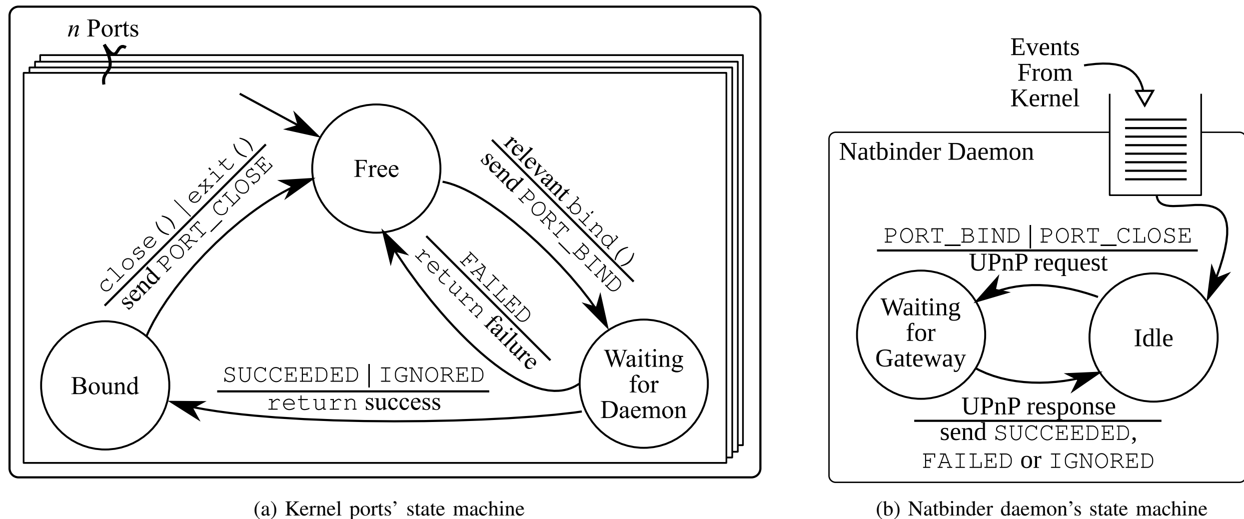
C. The Protocol

The definition of a new protocol on top of Netlink was fairly simple, being a matter of picking a free protocol number in the `netlink.h` header file and aliasing the name `NETLINK_NAT_PASS` to this number. Thus, most of the work on creating the protocol lies in defining its vocabulary and the behavioural interaction between kernel and daemon.

Messages of `NETLINK_NAT_PASS` can be split into two categories: one can be either *request* or *response*. The two *request* type messages, which are always sent by the kernel, are `PORT_BIND` and `PORT_CLOSE`. Each message takes four parameters: a sequence number, that will identify the request within the kernel; the IP address of the request (must be 0.0.0.0 to be relevant); the requested port number (greater than 1023 to be relevant) and the transport protocol used (either TCP or UDP).

The *response* type messages are always issued by the daemon to the kernel in response to a *request* message. They

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-183-0



(a) Kernel ports' state machine

(b) Natbinder daemon's state machine

Figure 3. Protocols' behavioural *automata*

can be either SUCCEEDED, FAILED or IGNORED. Each *response* takes as parameter the sequence number of the *request* that originated it, so the kernel can match each received *response* with a pending *request*.

The whole communication dynamics can be seen as a layered architecture, as shown in Fig. 2. In this perspective, the top layer are the user processes, which are served by the kernel. The kernel provides services to the user processes via system calls, and in turn is served by the Natbinder daemon, layered beneath it. In this layer our newly defined protocol NETLINK_NAT_PASS is used as the interface the daemon uses to provide the services to the kernel. In the lowest layer, the Internet gateway serves the daemon via the UPnP IGD protocol.

Inside the kernel, there is a three-state *automaton* for every port (see Fig. 3a). This *automaton* reacts from system calls events relevant to port forwarding (as discussed in Subsection III-B), and requests services from the daemon. The bind() event triggers a PORT_BIND request for the daemon to perform the port forwarding, and the *automaton* is hold on state "Waiting for Daemon" until receiveis a *response* message (or upon timeout). When terminating a port usage with PORT_CLOSE, the *automaton* waits for no response, resuming immediately.

On the daemon, the requests from the kernel are serialized by the Netlink protocol, being handled sequentially. As show in Fig. 3b, for every *request* message arriving, there is one request made via UPnP to the gateway. Upon each response received, a corresponding *response* type message is issued back to the kernel, using the same sequence number given in the *request*.

D. Error Handling

It is very important that an application do not hang too long while waiting for the bind() to conclude. Since the kernel have no control on the status of processes listening on the broadcast channel of NETLINK_NAT_PASS, two precautions are taken inside the kernel. First, before issuing the PORT_BIND request, kernel checks if there are any listeners on the broadcast channel. If there are none, the ordinary procedure of port binding is resumed.

As a system daemon, Natbinder is expected to run on system startup, but if for some reason it is not running, maybe because of a bug or because it was explicitly shutdown by the user, that check on kernel side will ensure bind() calls will be served normally (*i.e.* without the automatic NAT traversal feature).

There may also be the case the PORT_BIND request was already issued by the kernel, but the daemon stopped answering due to a bug or network error. To avoid letting the user process blocked indefinitely, inside the kernel there is a timeout of half second on waiting for the response. After that, bind() will resume as if it had received the IGNORE message as response.

Having the daemon up and running, we cannot fully trust on the gateway reliability. Configurations made by Natbinder on the gateway are volatile, so if the user simply manually restart a modem working as the Internet gateway, all NAT configurations previously performed are lost. Also, UPnP IGD implementation on devices might have its own issues. For instance, the gateway used during the development of this work had a maximum limit of 32 ports simultaneously forwarded via UPnP IDG. Port mapping requests after this limit would fail with an unknown error code, what will generate an IGNORED response to the kernel.

To counter this kind of problems, concerning the gateway reliability, the daemon holds the ideal state of all port mappings managed by itself. Periodically, it checks the gateway state and compares it with its own ideal state. If

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-183-0

they are divergent, the daemon tries to make the necessary changes to match the gateway state with the ideal state. This way, if a port mapping was not possible because of an unknown error, ideally it should have been mapped, so the ideal state will hold this port mapping. Upon the periodical check on the gateway, this pending port mapping will be tried again. Considering the previously mentioned case where the gateway was restarted, the daemon will find no port mapping entries on the gateway, so it will try to register all of them.

V. Use Cases

To test the approach some existing applications that would benefit from it were chosen to build test case scenarios. The first test was performed with the P2P application rTorrent [19], a BitTorrent client working over TCP. The second test was performed with the game OpenArena [20], whose netwoking multiplayer is done via UDP. Both of the previously mentioned application are not prepared to handle NAT automatically. The third test was performed with Transmission [21], another BitTorrent client which is able to perform automatic port forwarding.

In the first test, with rTorrent, the program was configured to use ports in the range from 10000 to 10009, and five instances of it were executed. The odd port numbers of this range were already taken in the gateway by another host in the network, so only the 5 remaining even ports were available to be used externally. All five instances were able to execute and properly bind to each one of the remaining ports. All five were able to receive incoming connections from the Internet. As expected, if a sixth instance is executed, it is unable to find a suitable port and exits with the error message: *"Could not open/bind port for listening: Address already in use."*

There is a mechanism inside rTorrent, similarly to other applications, including OpenArena, that was designed to find an available port on what to operate. The approach implemented in this work was conceived so not to disrupt such behavior. In this case, some ports where already taken in the gateway, but since they were presented to the application as ports already used by the TCP stack, its own port finding mechanism was able to devise an usable port.

The second test was performed by running the standalone server of OpenArena, which by default wait for players on UDP port 27960, and then opening the client and creating another multiplayer game rood, which will use the next available UDP port: 27961. Both ports were correctly and automatically exposed to the Internet, and both removed when the application closed. External clients were able to connect.

The Transmission test was performed by executing it with our automatic port forwarding mechanism disabled, then the program was killed, simulating a crash. Since it implemented the UPnP protocol, it was able to automatically forward its port on the gateway, but when it was killed, it was not given time to cleanup, so its entry persisted on the gateway. With the automatic port forwarding, the port was forwarded all the same, despite the redundant work performed both by the application and our daemon, but when it was killed, the daemon still cleaned it up on the gateway.

VI. Conclusion and Future Works

In this work, we presented a new approach to NAT traversal, including its architecture and reference implementation. The architectural choices are closely related to the technologies used. Were we using a less common but simpler protocol like NAT-PMP, our architecture would also be simpler.

It is hard to measure the real benefit of our approach, being it subjective when concerning user experience and useful to software developers mostly after its widespread adoption. In any case, as a future work we expect to survey the benefits of its usage with a group of volunteering users.

To reach a public of users and probable volunteers, our implementation will be integrated with Ubuntu, a popular GNU/Linux distribution. At first via a third party package maintained in Ubuntu's Personal Package Archives (PPA). Eventually, depending on community acceptance, into mainline Ubuntu.

As a Linux kernel modification, the implementation shall be submitted for inclusion into the official kernel distribution. Among other factors, the inclusion will be subject on the community acceptance of the concept idea, and the code stability asserted by the early testers.

Using the reference implementation, any OS based on Linux could easily support the approach, even Android, which is a mobile OS but still subject to the common networks behind NAT.

To make our implementation more user friendly, we plan to include a Graphical User Interface (GUI) for the daemon integrated with the system shell. Being a separated application, each system could have its own GUI that provides the better integration with its environment. In Ubuntu, such GUI would be and extension of the NetworkManager application, providing seamless desktop integration. Through this GUI, the user would be able to monitor the status of the automatically forwarded ports on the router.

The approach is not limited to our architecture or specific implementation, and could be done by vendors or third party software providers of other platforms and operating systems, including Windows, MacOS, Playstation, *&c*. The approach could also be implemented over other protocols, like IPv6, should NAT prove to still be useful with it.

In matters of network transparency in the operating systems, network interfaces are virtual enough abstraction. On Linux, besides the real Ethernet devices with associated IP addresses, there are virtual interfaces for loopback, VPN's, PPP, tunnels, *&c*. In a future work, the approach proposed here could be generalised as another virtual network interface,

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-183-0

managed by a NAT client network driver, which would have as address the shared public IP. A bind to it would be automatically forwarded. An interface like this seems more naturally fit.

Acknowledgment

This research was sponsored in part by a grant from CAPES. L. C. V. would like to thank Leonardo de Sá Alt and Rodrigo Queiroz Saramago for their invaluable help in understanding the inner workings of the Linux kernel.

References

[1] P. Belimpasakis, A. Saaranen, and R. Walsh, "Home dns: Experiences with seamless remote access to home services," in *World of Wireless, Mobile and Multimedia Networks, 2007. WoWMoM 2007. IEEE International Symposium on a*, June 2007, pp. 1 –8.

[2] A. Haber, J. De Mier, and F. Reichert, "Virtualization of remote devices and services in residential networks," in *Next Generation Mobile Applications, Services and Technologies, 2009. NGMAST '09. Third International Conference on*, Sept. 2009, pp. 182 –186.

[3] V. Pankakoski, "Experimental design for a next generation residential gateway," Master's thesis, Aalto University, School of Science and Technology, 2010, retrieved: Dec., 2011. [Online]. Available: http://lib.tkk.fi/Dipl/2010/urn100389.pdf

[4] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," RFC 2460 (Draft Standard), Internet Engineering Task Force, Dec. 1998, updated by RFCs 5095, 5722, 5871.

[5] Ipv6.br faq. Núcleo de Informação e Coordenação do Ponto BR. Retrieved: Dec., 2011. [Online]. Available: http://www.ipv6.br/IPV6/MenuIPV6FAQ#Que_tamanho_de_bloco_IPv6_devo_f

[6] P. Srisuresh and K. Egevang, "Traditional IP Network Address Translator (Traditional NAT)," RFC 3022 (Informational), Internet Engineering Task Force, Jan. 2001.

[7] A. Hemel, "Universal plug and play: Dead simple or simply deadly?" in *System Administration Network Engineering, 2006*, May 2006.

[8] Upnp forum. Retrieved: Dec., 2011. [Online]. Available: http://upnp.org/

[9] J. Blokhuis, "Universal plug and play vulnerabilities in eventing," University of Amsterdam, Tech. Rep., 2009.

[10] Upnp hacks: Hacking universal plug and play. Retrieved: Dec., 2011. [Online]. Available: http://www.upnp-hacks.org/

[11] S. Cheshire, M. Krochmal, and K. Sekar, "NAT Port Mapping Protocol (NAT-PMP)," Internet-Draft, Internet Engineering Task Force, 2008, retrieved: Dec., 2011. [Online]. Available: http://tools.ietf.org/id/draft-cheshire-nat-pmp-03.txt

[12] Natbinder. Retrieved: Dec., 2011. [Online]. Available: http://www.gitorious.org/natbinder

[13] Linux upnp internet gateway device. Retrieved: Dec., 2011. [Online]. Available: http://linux-igd.sourceforge.net/

[14] Pseudo ics daemon. Retrieved: Dec., 2011. [Online]. Available: http://pseudoicsd.sourceforge.net/

[15] Miniupnp project homepage. Retrieved: Dec., 2011. [Online]. Available: http://miniupnp.free.fr/

[16] *The `/proc` filesystem*, Documentation/filesystems/proc.txt, contained in Linux source code distribution.

[17] *sysfs – The filesystem for exporting kernel objects*, Documentation/filesystems/sysfs.txt, contained in Linux source code distribution.

[18] *UPnP IGD WANIPConnection*, UPnP Forum Std., Rev. 2.0, Sept. 2010, retrieved: Dec., 2011. [Online]. Available: http://upnp.org/specs/gw/igd2/

[19] The libtorrent and rtorrent project. Retrieved: Dec., 2011. [Online]. Available: http://libtorrent.rakshasa.no/

[20] Openarena. Retrieved: Dec., 2011. [Online]. Available: http://www.openarena.ws/

[21] Transmission. Retrieved: Dec., 2011. [Online]. Available: http://www.transmissionbt.com/

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-183-0