

•  
•  
•  
•  
•  
•  
•

Network Design & Evaluation  
Professor Peter Steenkiste, PhD  
Report by  
Mark Alexander Friedgan  
Bryan Meyer

# Network Design and Evaluation

## Network Address Translation



*Final Report*

December 12, 2000

## Goals

We set out to build a configurable Network Address Translator supporting Dynamic Load Balancing for requests coming to an internal network. There are many uses for such a system. For example, if a web content provider wants to have a highly responsive web server, one option would be for him or her to purchase a high-end web server, which would end up costing around \$100,000. However, for a more affordable solution a load-balancing network address translator (NAT) would be the ideal choice. Such a NAT would allow the creation of a web-farm that has one external IP address for browsers to access. The NAT then takes all requests coming in to that IP address and spreads them across the web servers located in a web farm, internal to the network (hidden from the WAN). Such a setup would allow the content provider to purchase ten \$1500 web servers instead of one \$100,000 server, yet the content provider would offer a more reliable service due to the machine redundancy. Most likely, such a system would be able to serve more hits than the high-end server as well.

The idea for the dynamic load balancing NAT is simple. You present a single IP address to the outside world, placing the NAT behind that IP. You use the NAT to hide an entire network behind. Then, you can manage the NAT via a program known as the "NAT Administrator", which can run either on the router itself or remotely on a Linux machine. The automatic load balancing would use the same protocol as the administrator, but it would be a module loaded into the Apache web server. Based on web server loads, the software would set entries in the NAT table in such a way as to allow or disallow packets to be routed to the server. To deal with server failure one would need to add functionality to the administrator to have it poll servers in order to keep the NAT tables current.

The idea of using a network address translator to do load balancing is not new, it was described in RFC 2391. There are many implementations of NATs ranging from hacked kernels to small, specialized boxes for DSL / cable-modem connections to ones written for large CISCO routers. We are not aware of anyone commercially using a NAT to do load balancing, yet we cannot establish a good reason why not. The router experiences a minimal impact on performance, because the checksum modifications can be pre-calculated, enabling one to serve highly available information with a minimal tradeoff in processing cost.

Unfortunately, due to the loss of a group member and problems with the development environment (e.g. `ip_NAT_modify` was not implemented) we switched to a simplified implementation. Our new goal was then to build a Network Address Translator with basic translation between incoming and outgoing requests. Our current NAT system can be used to connect two subnets together. The NAT System takes packet requests made from an internal subnet, i.e. having a source address (SA) of 192.168.1.169, and translates them (mangles the IP header) to the 192.168.2.xxx domain so that they have a SA of 192.168.2.169. Such a translation allows machines on the external (192.168.2.xxx) subnet to recognize the requests as having come from valid machines on their subnet. These machines will then respond to the requests. The responses are sent back to our NAT system, which then translates their destination address in their IP header back from 192.168.2.169 to 192.168.1.169: the packets in turn reach the original requestor of the information.

We manage the NAT system with a C++ program called NAT\_Admin that runs directly on the IXP 1200. (The Intel IXP 1200 is the programmable router we used to implement the NAT system.) NAT\_Admin allows the network administrator to specify how each "imaginary" IP address on an internal subnet maps to a "real" IP address on an external subnet (and vice-versa).

Figures 1 & 2 illustrate our initial and final system architectures. The next section provides an overview of our architecture.

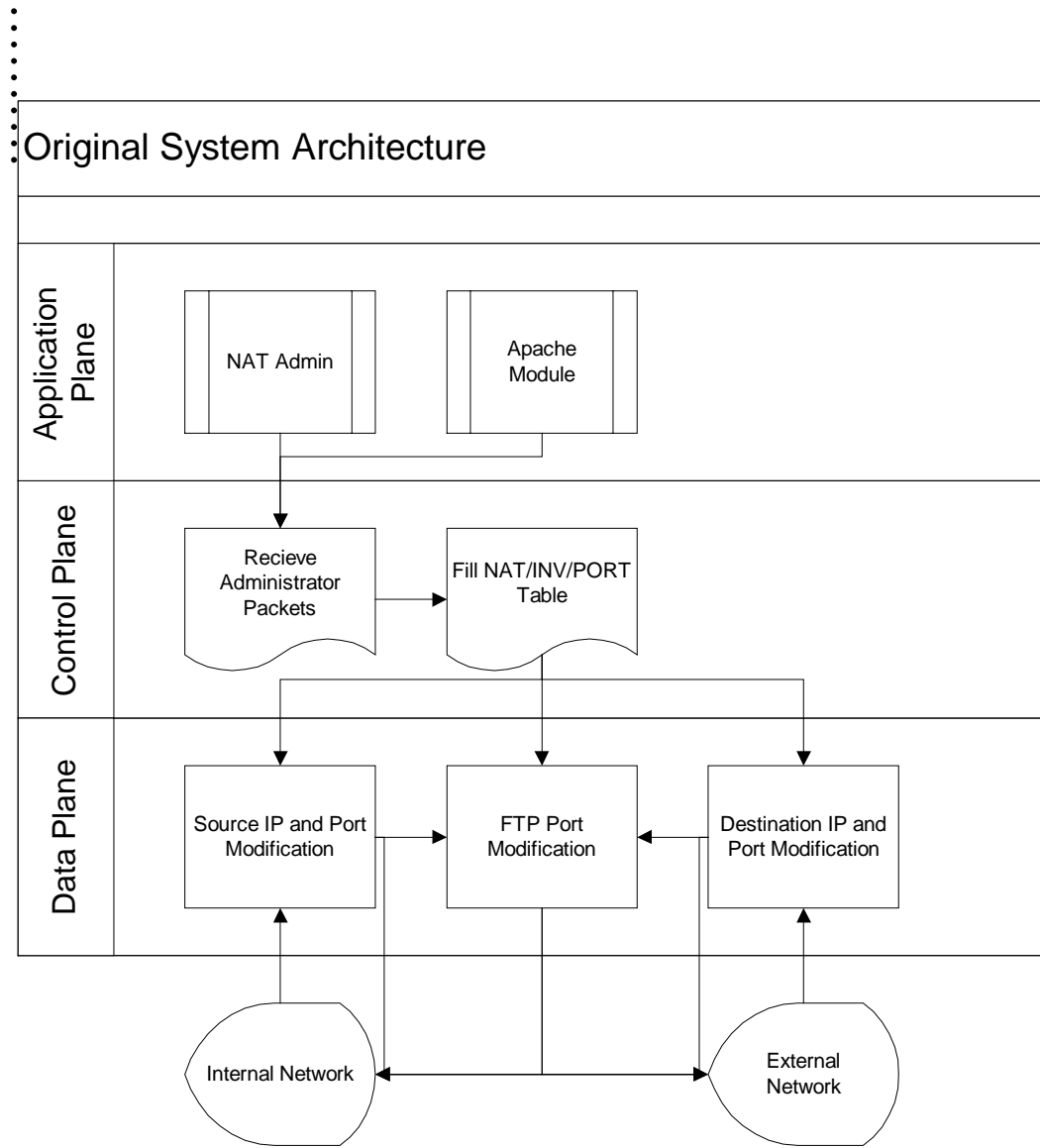


Figure 1

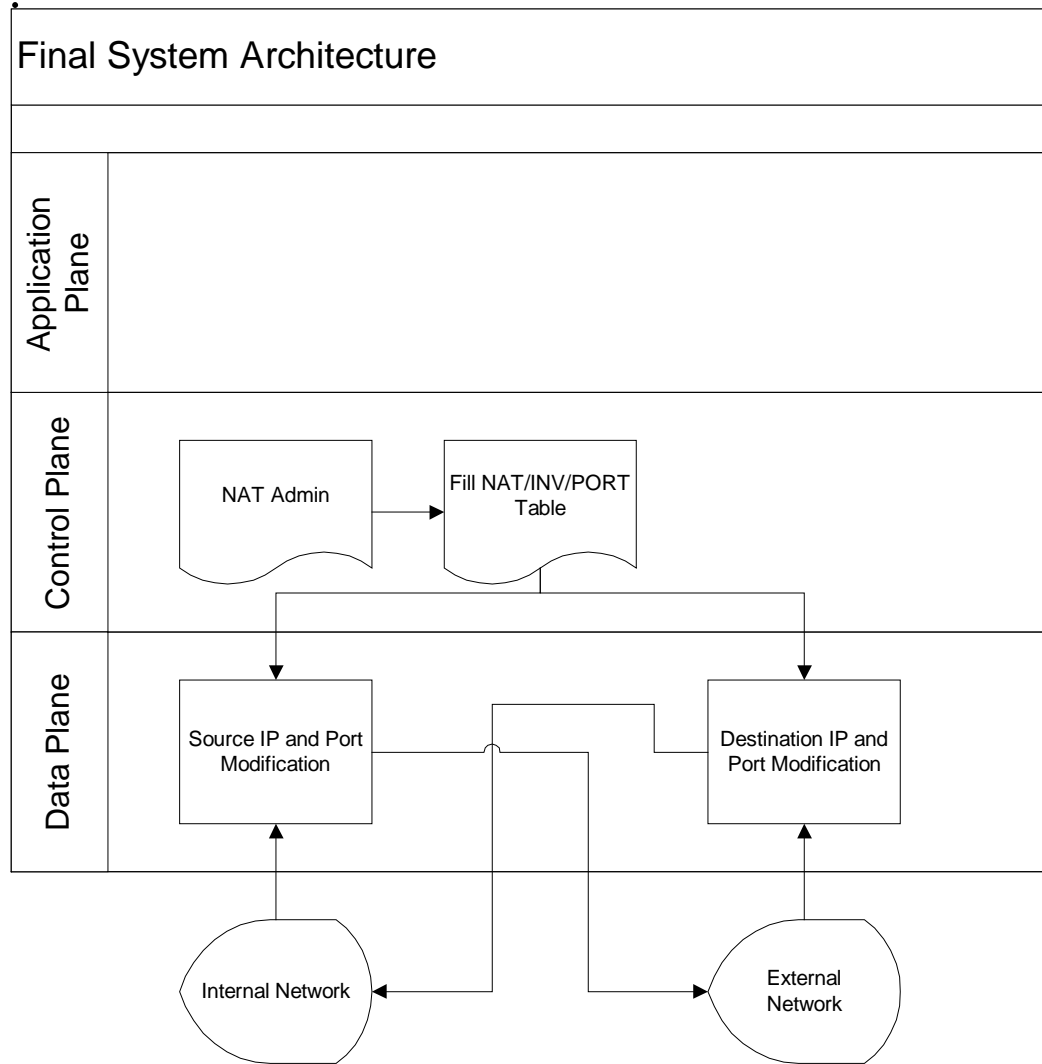


Figure 2

•  
•  
•  
•  
•  
•  
•  
•

## Architecture

General IXP 1200 systems can be abstracted into three functional planes, the Data Plane, the Control Plane, and the Application Plane. For the NAT system, we only require two of the functional planes, the Control Plane and the Data Plane.

### Data Plane

The purpose of the Data Plane is to handle the IP translation of each packet that comes into the IXP 1200 box. The data plane looks at the IP header of each packet originating from the internal network to determine to what IP address the source address should be changed. Similarly, the data plane also looks at all packets originating from the external network to decide to what IP address the destination address should be changed. The Data Plane performs this basic packet modification based on the NAT table and the INV(erse) table. We have additional code written to change the TCP ports using a PORT table, however time did not permit for the full implementation and testing of that part of the system. Using information from the NAT and PORT tables, the `ip_nat_modify` macro mangles packets coming in from the internal network. Similarly, the `ip_nat_modify_back` macro uses the INV table to change packets coming in from the outside port. The checksums are corrected appropriately using pre-calculated checksum differences stored in the tables. This pre-calculation allows the modification to be extremely fast. It takes less than 50 cycles per packet with 3 extra memory accesses on the way out and 7 on the way in. A lot of extraneous calculations have to be done because the packets are not well aligned in memory and thus IP addresses span words, which requires extra retrieval and bit manipulations.

### Control Plane

The Control Plane consists of a program for administrating the NAT and inverse NAT tables. Its purpose is to set up the SRAM in the IXP 1200 to contain the NAT and INV tables to allow the data plane to quickly translate IP addresses back and forth between the internal and external network. To achieve this, we hash each internal source IP address to a location in the NAT table and place the internal IP address (`int_IP`), the external or translated IP address (`ext_ip`), and the checksum difference (`int_IP - ext_IP = checksum_d` in one's complement) into each NAT table entry. Similarly, we set up an inverse table where each entry is hashed by the external IP address (`ext_IP`). The fields in the INV table are as follows, the `ext_IP`, `ext_PORT`, `int_IP`, `int_PORT`, `checksum_d`, and `tcp_d`. Note that we actually do not currently utilize the `ext_PORT`, `int_PORT`, or `tcp_d` entries, but have them there for future improvements to the system. It is important to note that due to the simplification we have made in the way we deal with ports, we are hashing the INV table by the external IP address rather than the external port as is the correct way. It is important to note that we pre-calculate and store the checksum difference for the IP address change. This allows the calculations done in the Data Plane to be minimal thus reducing the impact on system performance.

The Control Plane program receives input from a command line interface provided by VxWorks through the Tornado2 Development Environment. The user is provided with a menu of choices to add entries, delete entries and clear all of the tables in SRAM. The entries in SRAM lie in the 0x2100+ area, which is not utilized by any reference design code.

## Detailed Design and Implementation

Taking our abstract description of the NAT and converting it into IXP 1200 microcode and VX-works c-code took careful planning and some trial and error. In this section, we provide an overview of the detailed design phase of our project and in depth looks at the functions and macros we implemented

### Detailed Design

The process for coming up with our detailed design required doing a lot of research. We used a lot of sources including RFC descriptions, NAT papers, teaching assistants, IXP 1200 documentation, professors, and peers. After having done sufficient research, we came up with the following designs for the Data and Control planes.

#### Data Plane Design

The difficult part of designing the Data Plane (microcode) was figuring out where to place our code. It took a good deal of pouring through code, before we decided to place all of our IP header modification code in the **rec\_lmatch.uc** file. We decided this because all receive threads running on the IXP microprocessors use **rec\_lmatch.uc** to receive new packets and decide what to do with them; that property makes it the natural choice for adding code that will be modifying the packets as they are received. Most of the auxiliary macros we defined (except for endian macros) were placed inside of the **ip.uc** macro file, as they dealt with IP header manipulation. We placed the other two endian macros inside of **endian.uc**.

#### Control Plane Components

The basic job of the Control Plane in our design was to set up the tables utilized by the modification macros. In order to do this we had to find an area of SRAM that was unused in the reference design and designate parts of it as tables. Each table had 32 entries and thus did not span a very large amount of space. The control plane program had to be able to obtain the information from the user, hash the info and store it in the tables for the microcode to read.

### Implementation

Once we determined where to put the code, the next step was going ahead and implementing it. Part of the code was in a new assembly language (microcode), so there was a steep learning curve associated with the implementation phase. However, by the time we finished, we understood much more about how the IXP 1200 architecture actually works. Along with the microcode, we had VxWorks c-code that was needed for the NAT administrator running on the ARM processor. In the rest of this section, we describe the important macros and functions we wrote that allowed the NAT to work.

•  
•  
•  
•  
•  
•  
•  
•  
•  
•  
•

### Data Plane Macros

The data plane microcode used several helper macros to perform the task of mangling IP addresses from and back to their original states. Following, we describe the most important ones.

#### ip\_nat\_modify

This macro utilizes the information in the NAT table to change the source IP of outgoing packets. It is modeled after the reference design's **ip\_modify** macro and is placed after it in order to avoid having it's modifications overridden. It is given the new IP address and the new checksum as arguments to place into the packet. The checksum and new IP address reside across words three and four of the 8-word set looked at by this macro. Prior to calling this macro the old checksum is retrieved and added to the checksum difference to produce the new checksum. Figure 5 graphically depicts the setup of the IP header.

Version (4 bits)	IHL (4 bits)	Type of Service (8 bits)	Total Length (16 bits)	
Identification (16 bits)			Flags (3 bits)	Fragment Offset (13 bits)
Time to Live (8 bits)		Protocol (8 bits)	Header Checksum (16 bits)	
Source Address (32 bits)				
Destination Address (32 bits)				
Options and Padding (multiples of 32 bits)				

Figure 3

NAT Table				
Hash Index 5-bits	Internal IP 32-bits	External IP 32-bits	Checksum Difference 32-bits (16 used)	Padding 32-bits
0				
...				
32				

Figure 4

•  
•  
•  
•  
•  
•  
•  
•

## Network Address Translation

### **ip\_nat\_modify\_back**

This macro works in a similar manner to the one above except for the fact that the destination IP address which needs to be changed by this macro does not reside fully in the 8 words set loaded from SRAM. Therefore this macro also retrieves the destination IP address's second half as well as the rest of that word from SRAM and writes that into the correct area of SDRAM.

INV Table							
Hash Index	External IP	Internal IP	External Port	Internal Port	Checksum Difference	TCP Checksum Difference	Padding
5-bits	32-bits	32-bits	32-bits	32-bits	32-bits	32-bits	64-bits
0							
...							
32							

Figure 5

### **ip\_nat\_tcp\_modify**

This macro was not implemented due to lack of time and resources. It would have modified the source or destination TCP ports and the TCP checksum so that TCP communication would be possible.

### **Code**

Calls to **ip\_nat\_modify** are made in the microcode running on the internal ports and calls to **ip\_nat\_modify\_back** are made on the external port. These calls are made inside *rec\_lmatch.uc* after the longest route has been matched it is known that the packet will be traveling outside of the network.

### **Control Plane Functions**

The control plane functionality is encompassed in the **nat\_admin** program; its primary functions are described below.

#### **int get\_checksum\_diff(int old\_ip, int new\_ip)**

This function, given two IP addresses returns the difference between them being part of the checksum. This is calculated as follows. Each IP address is cut into two 16-bit parts; those parts are added together with one's complement addition; the old IP address checksum is then negated and added to the new IP address checksum, this sum is then once again negated. This produces the correct checksum difference. For calculating the actual checksum difference we needed some insight into how to obtain a new checksum. Figure 6 shows the equations we used to determine how to obtain the checksum difference.





## Experiment 2 – Mid-level NAT functionality

The second part of the testing was for performance. For this, we simply used round trip ping delays. Specifically, we recorded average ping delays (1) between two computers on a network, (2) between two computers talking through the IXP box running the code provided (no added functionality), and (3) between two computers with our NAT code running on the box. We achieved results as listed in the table.

System Setup	Round Trip ping Delay Seen
Ping between two computers on the LAN	0 to 1 ms
Ping between two computers through a bare-bones IXP 1200	1 ms
Ping between two computers through an IXP 1200 running the NAT code	1-2 ms

Figure 7

## Experiment 3 – Low-level NAT functionality

For testing at a low level, we planned on running experiments that would test the scalability of our NAT. Specifically, we would have liked to set up a number of Linux box pairs sending ping packets back and forth and observed sluggishness in the system. Such an experiment would have indicated to us if our NAT were scalable. Unfortunately, we did not have access to enough machines to be able to perform such a benchmark. However, we believe that our system would have performed well even with a large load, seeing as how the overhead of our code (both in terms of added round-trip delay and added processing on the micro-engines) was very small.

## Conclusion

Overall, we learned many things about networking from our project. We learned a lot about how the IXP 1200 works both at a high level view as well as at a low level. For the high level part, we have a clear understanding of how the data plane and the control plane fit in with the process of receiving packets, processing them, and then sending them out. At a low level, we now have a good understanding of how to write microcode.

As for lessons learned, we discovered that to get code to work, it is best to test it in small steps. As you get one segment of code working, you can move on to the next. This allows you to easily diagnose bugs, and makes the overall process of debugging go much more smoothly. If we had better tutorials for the IXP 1200, we would have had an easier time implementing it. There are some other improvements that could be made to the course that would make it more interesting in that the groups would be able to get more done. For example, it would have been nice if we had had the Peth driver issues resolved at the beginning of the semester. In addition, having working hardware demos to play with earlier on would make the transition to testing our simulations occur earlier, providing us more time for development and debugging. Finally, not having the VxWorks development program, Tornado, installed until much later in the semester hindered our ability to fully merge our data plane and control plane until very late in the semester. Overall, we feel that this was a beta semester, using beta code –future semesters will have access to a stable code base and probably will not experience as many of the hardships that we experienced.