network4dev Documentation

Release 1.0

network4dev

Oct 10, 2018

Main Page

Network4Dev is a community-driven initiative to explain networking infrastructure concepts & challenges to a wider audience (for example, but not only, devs & sysadmins).

CHAPTER 1

Introduction

Welcome to Network4Dev! Network4Dev is a community-driven, open source networking site. The site was founded by Daniel Dib in 2018 and all work behind the scenes is done by Cristian Sirbu. At the time of this writing `2018-08-19`, the two maintainers of the GitHub repository are Cristian and Daniel.

As infrastructure is moving towards more elastic, composable and automated setups, infrastructure engineers must have a broader set of skills while still remaining relevant within their speciality. This means that networking people must understand compute, storage, applications, systems and coding at a level where they can interact with the other teams. Conversely, it means that people working mainly with systems and apps must have some understanding of networking. While there are plenty of sites and courses for networking people to learn about automation, there are a lot less resources for people working with systems and apps to learn about networking. The goal of N4D is to fill this gap.

In order to write for people that don't have a strict networking background, the articles will be written to be short, concise and to the point. The articles will not be deep dives but instead focus on fundamentals and the **WHY** of networking.

By making this site open source and community driven, more people can contribute into making this site living and more frequently updated than a site where only one or a couple of people are contributing.

In order to contribute to this site, read the *Contributing* guide.

Please note that all contributions to this site will be reviewed and edited by the maintainers for quality, accuracy and clarity both in terms of language and technical level in order to maintain our goals.

CHAPTER 2

Audience

The target audience for N4D is mainly people working as developers or system administrators. However, this site will be useful for anyone that wants to learn more about networking, especially junior networking people. The reason for this being that this site aims to explain networking concepts in a clear and concise manner, although not in depth.

CHAPTER 3

Contributing

This site was created to be open for contributions from the open-source community. This is achieved by commiting files to Github in `.rst` format.

In order to contribute to this site, the following is needed:

- Github account
- Github client
- Text editor
- (optional) Local install of Sphinx

By leveraging Github, multiple people can write documents for the site without having to use a platform like Wordpress using many different accounts.

3.1 Architectures

N4D chapters are based on architectures. At the start these are:

- Cloud
- Datacenter
- Design
- Networking
- Security
- Services

These are quite self-explanatory but the following adds more context to each architecture:

Cloud - Focus on networking in the public cloud, such as AWS, Azure and GCP.

Datacenter - Focus on on-premises datacenter technologies and design.

Design - Focus on networking design principles such as scalability, fault domains and more.

Networking - Focus on "standard" networking such as LAN and WAN.

Security - Focus on security such as firewalls, IPSec and more.

Services - Focus on networking related services such as DNS and DHCP.

3.2 N4D structure

N4D, using Read the Docs, thus using Sphinx as the backend, has the following structure setup:

- `docs` - All the documents go here
- `_static` - All static content, such as images, go here

In order to keep things sorted, there are also subdirectories based on different architectures, for the `docs` directory these are:

- Cloud (`docs/cloud`)
- Datacenter (`docs/datacenter`)
- Design (`docs/design`)
- Networking (`docs/networking`)
- Security (`docs/security`)
- Services (`docs/services`)

The `_static` directory also has the same layout:

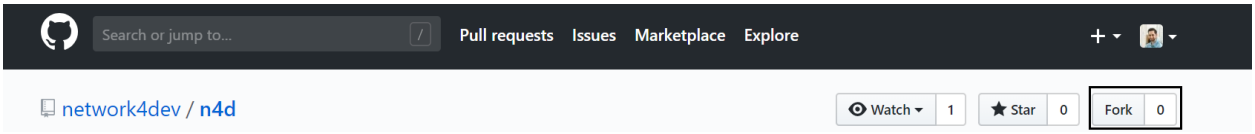
- Cloud (`docs/_static/cloud`)
- Datacenter (`docs/_static/datacenter`)
- Design (`docs/_static/design`)
- Networking (`docs/_static/networking`)
- Security (`docs/_static/security`)
- Services (`docs/_static/services`)

3.3 Workflow

This site uses a forking workflow, meaning that each contributor forks the official repository to create their own server-side copy of the project. This allows the maintainers to keep write access for the maintainers and instead approving incoming Pull Requests.

3.4 Setting up the environment

The first step to contributing is to create a fork of the N4D repository. This is done by clicking the `fork` icon in the top right corner.



When this is done, clone the forked repo to a local directory.

```
daniel@demo:~/n4d-fork$ git clone https://github.com/ddib-ccde/n4d.git
Cloning into 'n4d'...
remote: Counting objects: 22, done.
remote: Compressing objects: 100% (16/16), done.
remote: Total 22 (delta 2), reused 18 (delta 1), pack-reused 0
Unpacking objects: 100% (22/22), done.
```

When using the forking workflow, the upstream repository needs to be added.

`git remote add upstream https://github.com/network4dev/n4d.git`

The command `git remote -v` can be used to verify the origin and upstream repo.

```
daniel@demo:~/n4d-fork/n4d$ git remote -v
origin  https://github.com/ddib-ccde/n4d.git (fetch)
origin  https://github.com/ddib-ccde/n4d.git (push)
upstream        https://github.com/network4dev/n4d.git (fetch)
upstream        https://github.com/network4dev/n4d.git (push)
```

3.5 Contributing text

Before committing any text, it's recommended to make sure that your local copy of the repo is up to date. This is accomplished by fetching from the upstream master and merging it with your local copy.

If you are already synched, it will look like in the demo below.

```
daniel@demo:~/n4d-fork/n4d$ git fetch upstream
From https://github.com/network4dev/n4d
* [new branch]      master     -> upstream/master
daniel@demo:~/n4d-fork/n4d$ git checkout master
Already on 'master'
Your branch is up to date with 'origin/master'.
daniel@demo:~/n4d-fork/n4d$ git merge upstream/master
Already up to date.
```

Before working on a document, create a branch and name it appropriately, such as `osi-model-text` with `git checkout -b osi-model-text`. Using your text editor, edit the file and then commit the changes `git commit -a -m "Wrote about the OSI model`. Then the commit is pushed to origin, using `git push origin osi-model-text`. When pushing to Github, you will need to supply your credentials.

The demo below shows an update to the scrollbar for N4D:

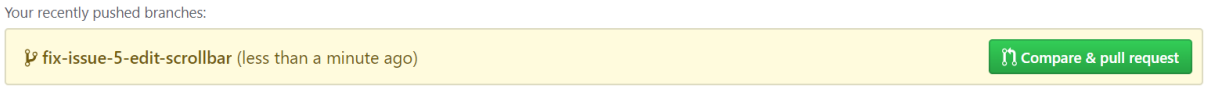
```
daniel@demo:~/n4d-fork/n4d$ git checkout -b fix-issue-5-edit-scrollbar
Switched to a new branch 'fix-issue-5-edit-scrollbar'
daniel@demo:~/n4d-fork/n4d$ git commit -a -m "Updated index.rst to set caption to␣
↪main page"
[fix-issue-5-edit-scrollbar d5ac8bf] Updated index.rst to set caption to main page
1 file changed, 1 insertion(+), 1 deletion(-)
daniel@demo:~/n4d-fork/n4d$ git push origin fix-issue-5-edit-scrollbar
```

(continues on next page)

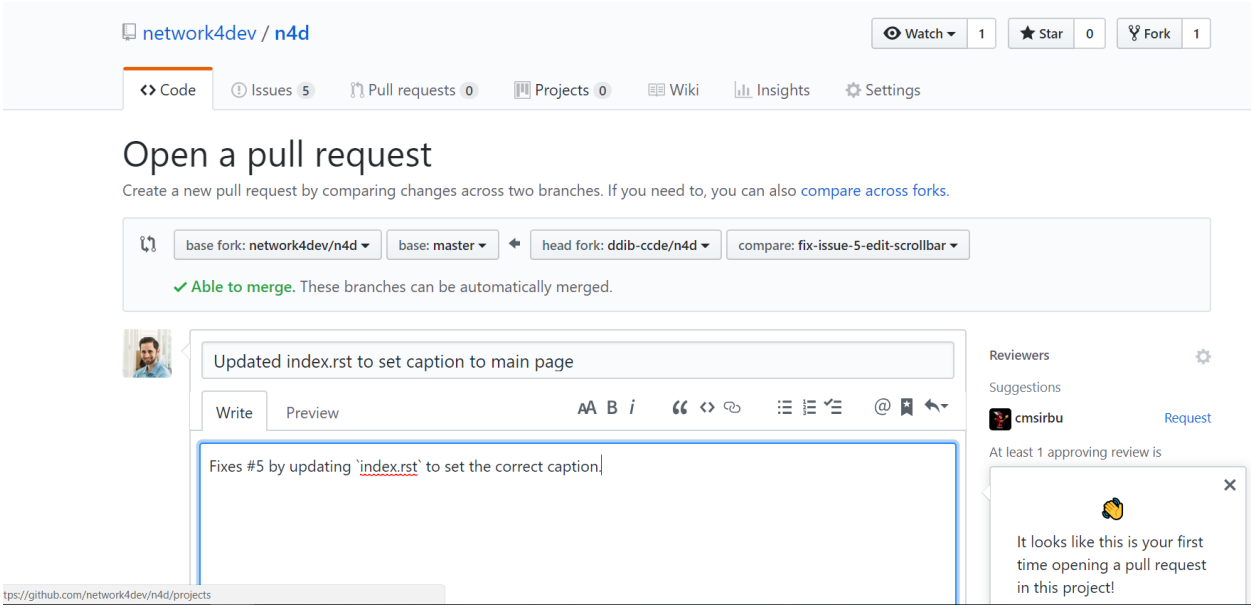
(continued from previous page)

```
Username for 'https://github.com': ddib-ccde
Password for 'https://ddib-ccde@github.com':
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 324 bytes | 324.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/ddib-ccde/n4d.git
* [new branch]      fix-issue-5-edit-scrollbar -> fix-issue-5-edit-scrollbar
```

Once the commit has been pushed to your forked repo at Github, you can submit a PR to upstream. By navigating to your forked repo at Github, Github will see that you have pushed a new branch, in order to create a PR from the branch, click on `compare & pull request`.

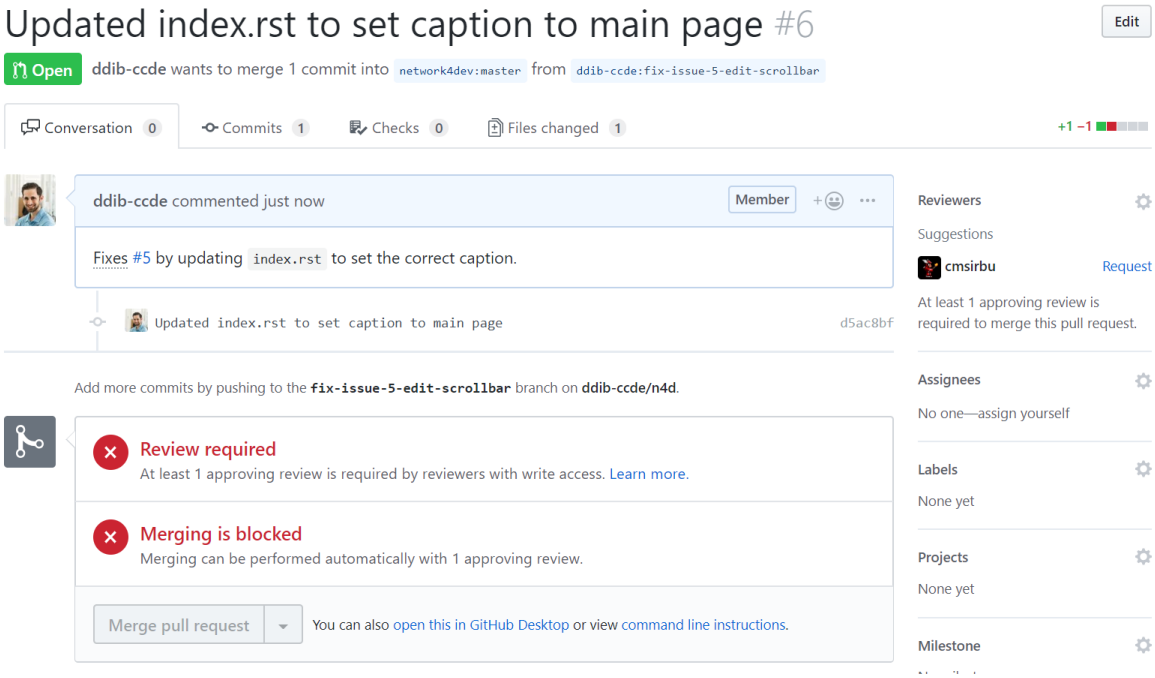


When this is done, it will look like the picture below.



If this PR is a fix to an issue, this can be indicated in the text, as it is in this example with "Fixes #5". Issues can be referenced by putting the hash sign followed by the number of the issue.

When the PR has been opened, it will look like the picture below.



The PR will then have to be approved by one of the maintainers before it will be merged to upstream.

3.6 Brief intro to reStructuredText

reStructuredText (reST) is the default plaintext markup language used by Sphinx. People familiar with Markdown will recognize some of the syntax.

First in every `.rst` document is the header. This is created by using equal sign above and below the text, like below:

```
==================
This is a heading
==================
```

3.6.1 Inline markup

- one asterisk: *text* for italics
- two asterisks: **text** for boldface
- backquotes: ‘‘text‘‘ for code samples

Below is a sample of what it looks like:

- *italics*
- **boldface**
- `code sample`

3.6.2 Hyperlinks

To put a hyperlink in the text, use the following syntax:

‘Link text <https://domain.invalid/>‘_

3.6.3 Images

Images are referenced with the image directive, like below:

```
.. image:: static_/my_image.png
```

Remember to put a blank line before and after this directive.

3.6.4 Code samples

It is supported to highlight different code examples, for example output from console with:

```
.. code-block:: console
```

It looks like the output below:

```
daniel@demo:~/n4d-fork/n4d/docs$ ls
audience.rst  changelog.rst  contributing.rst  design  infrastructure.rst  networking␣
↪ services
_build        cloud          datacenter        images  introduction.rst    security  ␣
↪ _static
```

It is also possible to highlight programming languages such as Python.

For a more complete guide, refer to reStructuredText Primer

To test your syntax, it’s possible to use Online reStructuredText editor

If unsure of the syntax, check one of the existing files in the `docs` directory.

3.7 Local Sphinx build

It is also possible to install Sphinx locally to test out your changes, this process is described here

Note that you should clone your forked repo if you want to be able to push changes upstream.

3.8 Issues

If there is an issue with the site, an article or if you need support. Open an issue by clicking `New issue`.

CHAPTER 4

Infrastructure

This site uses Read the Docs which is a platform commonly used for technical documentation. Read the Docs is using Sphinx to create the documents. The source files are kept in a Github repository leveraging webhooks to push any updates to Read the Docs.

By leveraging Github for the source files, this project can be contributed to from multiple parties and when a PR has been approved, the update will automatically be pushed to Read the Docs.

CHAPTER 5

Changelog

v1.0 - Work in progress

CHAPTER 6

General Networking

This part of the site is about general networking, i.e. networking technologies that are general enough to be used in many parts of the network.

6.1 OSI Model

6.2 Physical

6.3 Ethernet

6.3.1 Link Aggregation

Link aggregation is a general term to describe using multiple Ethernet interfaces simultaneously. There are several terms for this, including bonding, teaming, EtherChannel, and more. However, there are ongoing debates as to what each term means specifically, hence the reason there are many different terms to describe the same overall concept. Link aggregation has two primary goals: increased throughput and increased redundancy.

From a network infrastructure context, link aggregation normally means to combine two or more physical Ethernet interfaces with identical properties (such as link speed and duplex) into a single logical interface. For example, two physical 10 Gbps Ethernet links can be combined into a single 20 Gbps logical link. However, the data is not actually transmitted at 20 Gbps.

The most common analogy for link aggregation is adding more lanes to a highway, but the speed limit stays the same. In other words, with multiple aggregated 10 Gbps interfaces, a data transfer in a single session will not go faster than 10 Gbps, but multiple sessions can be distributed over each of the links, which enables an aggregate throughput of more than an individual link.

Why is link aggregation needed?

Network switches build internal tables to correspond MAC addresses to interfaces so network traffic can be appropriately forwarded. Within the switch, this is referred to as the Content Addressable Memory (CAM) table. When you connect a device such as a PC or a server to a port on a switch, the switch sees the MAC address of the device's network interface and records in the CAM table that it is associated with that particular port. When devices communicate with each other, the switch knows which interfaces to forward the traffic between based on the CAM table.

Individual MAC addresses can be associated to only a single interface in the CAM table, though each interface can have multiple MAC addresses associated. A single IP address is typically associated with a single MAC address (with some exceptions). The Address Resolution Protocol (ARP) runs on both the host and the host's default gateway to map IP addresses to MAC addresses. When link aggregation is used, a MAC address is assigned to the aggregated logical interface. When an IP address is configured on the logical link, ARP resolves the IP address to the MAC address of the virtual link. This allows the multiple physical interfaces to be used simultaneously with the single IP address.

How is link aggregation different than NIC teaming?

Link aggregation requires coordination between the network infrastructure and the device containing multiple physical links with either static configuration or a negotiation protocol such as Link Aggregation Control Protocol (LACP). NIC teaming does not require coordination with the network infrastructure and is entirely host-dependent. With link aggregation, both the host and the network infrastructure share the same view of the local network. With NIC teaming, the network is unaware of the configuration on the host.

For example, VMware's ESXi operating system supports both link aggregation and NIC teaming. When link aggregation with LACP is configured, both the network and ESXi host share the same view of the network with the logical link. By default, ESXi uses NIC teaming. Virtual machines have virtual network interfaces that each have their own associated MAC addresses. When the VM is powered on, the virtual network interfaces are associated with a network interface on the ESXi host. When LACP is used, the VM MAC address becomes associated with the LACP logical link. When the default NIC teaming is used, the VM MAC address becomes associated with one of the ESXi host's physical network interfaces. When a second VM is powered on, its VM MAC address is associated with a different physical network interface on the ESXi host. This facilitates rudimentary network load distribution across multiple physical interfaces on the ESXi host without involving the network infrastructure.

Both LACP and NIC teaming support "active/active" and "active/standby" configurations. In "active/active" mode, all links are used simultaneously. In "active/standby" mode, one or more physical links are intentionally disabled. If one of the active links fails, the standby link becomes active and takes over for the failed link.

Why would I use link aggregation instead of NIC teaming?

Using VMware ESXi as an example, NIC teaming maps each virtual network interface (vNIC) onto a single host physical network interface. The mapping does not change unless the VM is powered off or migrated to another host, or if the physical link on the host goes down. This means network traffic to and from the VM's vNIC will always traverse the same physical interface and is limited to the speed of that interface.

With link aggregation using LACP, the same VM vNIC is mapped to the ESXi host's logical aggregated interface. The same rules still apply where a single network session (also known as a flow) is still limited to the speed of an individual physical link, but multiple flows can take advantage of the multiple links simultaneously, thereby increasing the overall throughput.

For example, with NIC teaming, if a VM is acting as a web server, it will use the same physical link when serving content to multiple clients. With link aggregation, the same VM will use multiple physical interfaces across the single logical interface to serve content to clients.

The choice of using link aggregation or NIC teaming is partly dependent on the network demands of your workload. If redundancy is your primary concern without regard for throughput, such as for workloads that do not generate large

volumes of network traffic, NIC teaming may be the best and simplest option. However, workloads that deal with high volumes of network traffic are typically served better by using link aggregation, which provides both redundancy and higher overall network throughput across multiple sessions.

What happens if I configure link aggregation on the host, but not the network?

Unlike NIC teaming, link aggregation requires coordination with the network infrastructure. The two most common methods are static configuration, and LACP (originally IEEE 802.3ad, now part of IEEE 802.1AX). With LACP, the host and the network switch communicate with each other to make sure each physical link is part of the same link bundle (otherwise known as a Link Aggregation Group or LAG). Any links that are determined to not be a part of the bundle are disabled to prevent forwarding loops and other issues.

For example, assume two different LAGs are configured on a switch and LACP is used. Two different servers are connected with multiple links, one server per LAG. If the links are accidentally misconnected so that each server connected to both LAGs, LACP will detect this and disable the links that do not belong to the proper group. Static configuration has no way to detect this problem, and is therefore not recommended unless both the server and the network switch do not support LACP (which is extremely rare).

Remember that in a network switch, a single MAC address must correspond with a single entry in its CAM table, which represents the interface the MAC address is associated with. If the server is configured to use link aggregation but the network is not, MAC addresses associated with the server appear on multiple switch interfaces simultaneously. Since a MAC address can only be on a single switch interface, the MAC address is disassociated from one interface and re-associated with another as the MAC address is seen from the connected server.

This disassociation and re-association happens over and over again between interfaces on the switch. This is known as "MAC flapping". Different network equipment handles this situation differently. Typically, network interfaces associated with MAC flapping are disabled, either permanently or for a short period of time. Log messages on the switch are also typically generated.

Here is an example of the error message generated on a switch running Cisco's NX-OS:

```
switch1 %L2FM-2-L2FM_MAC_FLAP_DISABLE_LEARN_N3K: Loops detected in the
network for mac 1234.5678.90ab among ports Eth1/1 and Eth1/2 vlan 111 -
Disabling dynamic learning notifications for a period between 120 and 240
seconds on vlan 111

switch1 %L2FM-2-L2FM_MAC_FLAP_RE_ENABLE_LEARN_N3K: Re-enabling dynamic
learning on vlan 111
```

In this case, Cisco's NX-OS handles the issue by disabling dynamic MAC learning for a period of time for the entire VLAN. This means that during this "quiet" period, no MAC address changes will be registered for that VLAN. If a new device comes online in the VLAN, or an existing device changes ports, the change will not be registered in the switch and the device will not be able to communicate with the network until MAC learning is re-enabled. This will happen over and over again until the issue is corrected. The issue can be corrected by shutting down the misconfigured links or correctly configuring link aggregation between the switch and the server.

What is MC-LAG?

Traditional link aggregation involves multiple connections to a single switch. Multi-Chassis Link Aggregation Groups aim to further increase redundancy levels by connecting a single device, such as a server, to multiple switches while still presenting a single logical interface to the server. This introduces device-level redundancy along with link-level redundancy. Currently, all MC-LAG solutions are proprietary to the networking vendor, and require both switches to be running the same network operating system.

For example, Cisco's NX-OS uses an MC-LAG technology called "Virtual Port Channel", or vPC. Two switches running NX-OS are configured to recognize each other and present a single unified LACP-based LAG to the downstream

device, such as a server. The server believes it is connected to a single upstream switch. The two switches coordinate with each other to handle the redundancy and prevent loops and MAC flapping.

6.4 Discovery Protocols - CDP and LLDP

6.5 Switching

6.6 IPv4

6.6.1 NAT44

Network Address Translation is a technique used to change, or translate, the source or destination IPv4 address. This is not an exclusive "or" as both addresses can be translated. This document is iterative in that each use case is described and demonstrated sequentially and in small batches.

Definitions

Master these terms before continuing:

- **inside**: The network *behind* the NAT point, often where source addresses are translated from private to public.
- **outside**: The network *in front of* the NAT point. Often packets in this network hae publicly routable source IP addresses.
- **state**: The information stored in the NAT device about each translation. NAT state is typically stored in a table containing pre and post NAT addressing, and sometimes layer-4 port/protocol information as well.
- **pool**: A range of addresses which can be used as outputs from the NAT process. Addresses in a pool are dynamic resources which are allocated on-demand when hosts traverse a NAT point and require different IPs.
- **ALG**: Application Level Gateway is a form of payload inspection that looks for embedded IP addresses inside the packet's payload, which traditional NAT devices cannot see. ALGs are NAT enhancements on a per-application basis (yuck) to translate these embedded addresses. Applications such as active FTP and SIP require ALGs.
- **unsolicited traffic**: In the context of NAT, this refers to traffic originating from the outside network and destined for some device on the inside network. It is called "unsolicited" because the inside host did not initiate the communication.

In Cisco-speak, some additional terms are defined. These seem confusing at first, but are a blessing in disguise because they are unambiguous.

- **Inside local:** IP on the *inside* network from the perspective of other hosts on the *inside* network. This is often a private RFC 1918 IP address and is seen by other hosts on the inside as such.
- **Inside global:** IP on the *inside* network from the perspective of hosts on the *outside* network. This is often a publicly routable address which represents the inside host to outside hosts.
- **Outside global:** IP on the *outside* network from the perspective of other hosts on the *outside* network. This is often a publicly routable address and is seen by other hosts on the outside as such.
- **Outside local**: IP on the *outside* network from the perspective of hosts on the *inside* network. This is usually the same as the outside global since the all hosts, whether inside or outside, view outside hosts as having a single, public IP address (but not always).

Notes

Things to keep in mind:

- All demonstrations assume Cisco Express Forwarding (CEF) has been disabled which allows IP packet debugging (routing lookups) to be shown. This is not supported on newer Cisco routers, so older devices running 12.4T are used in these demonstrations.
- This document *describes* NAT, but does not *prescribe* NAT. NAT designs are almost never the best overall solutions and should be considered short-term, tactical options. Software should avoid using duplicate and/or hardcoded IPs, then relying on NAT later to fix the blunder.
- Mastery of the NAT order of operations is required for any developer building an application that may traverse a NAT device. This affects which addresses are revealed (and when), impacts DNS, impacts packet payloads (ie, embedded IPs), and more. The basic *inside source* NAT order of operations is described below.
    1. Packet arrives on inside
    2. Router performs route lookup to destination
    3. Router translates source address
    4. Reply arrives on outside
    5. Route translates source address back
    6. Router performs route lookup to destination

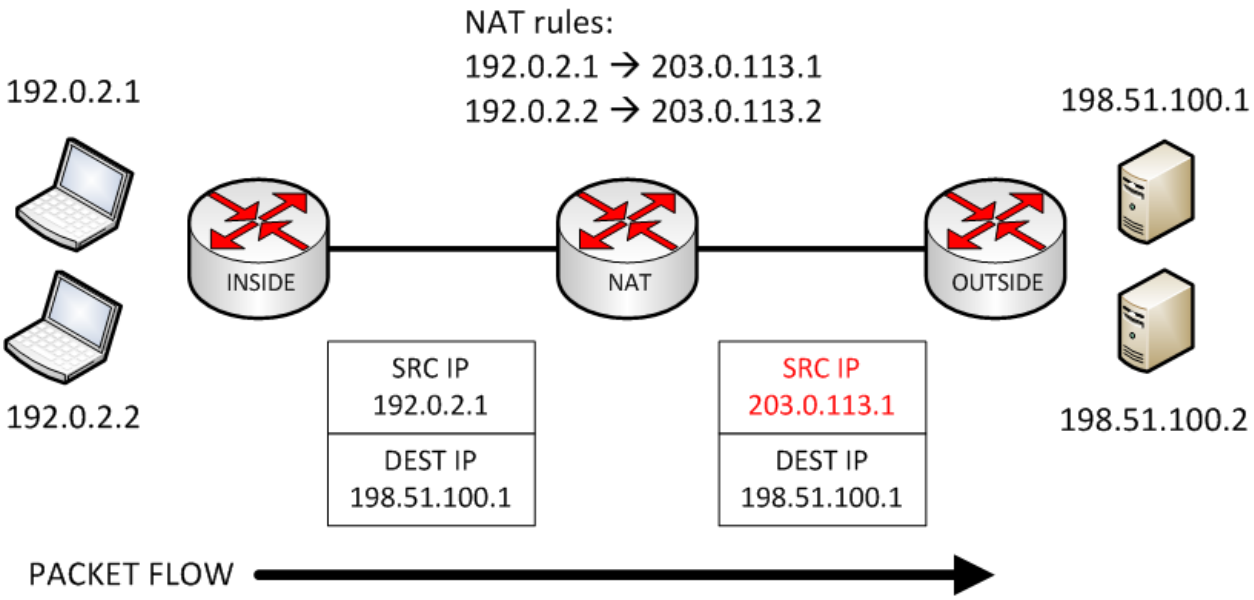
Static One-to-one (1:1) Inside Source NAT

The simplest type of NAT is 1:1 NAT where a single source IP is mapped to another. This technique is useful in the following cases:

- Inside host should not or cannot share a source IP with other hosts
- Custom IP protocols are used (not TCP, UDP, or ICMP)
- Outside-to-inside unsolicited traffic flow is required
- Address traceability is required, typically for security audits

When traffic traverses NAT, only the source IP changes. In this example, inside hosts `192.0.2.1` and `192.0.2.2` are mapped to `203.0.113.1` and `203.0.113.1`, respectively. Cisco IOS syntax is below:

```
# ip nat inside source static (inside_local) (inside_global)
ip nat inside source static 192.0.2.1 203.0.113.1
ip nat inside source static 192.0.2.2 203.0.113.2
```



Enabling NAT (`debug ip nat`) and IP packet (`debug ip packet`) debug reveals how this technology works. Follow the order of operations:

1. Initial packet arrives from `192.0.2.1` to `198.51.100.1` (not in debug)
2. NAT device performs routing lookup towards `198.51.100.1`
3. NAT device translates source from `192.0.2.1` to `203.0.113.1`
4. NAT device forwards packet to `198.51.100.1` in the outside network
5. Return packet arrives from `198.51.100.42` to `203.0.113.1` (not in debug)
6. NAT device translates destination from `203.0.113.1` to `192.0.2.1`
7. NAT device perform routing lookup towards `192.0.2.1`
8. NAT device forwards packet to `192.0.2.1` in the inside network

Some information has been hand-edited from the debug for clarity:

```
IP: tableid=0, s=192.0.2.1, d=198.51.100.1, routed via RIB
NAT: s=192.0.2.1->203.0.113.1, d=198.51.100.1
IP: s=203.0.113.1, d=198.51.100.1, g=203.0.113.253, len 100, forward
NAT*: s=198.51.100.1, d=203.0.113.1->192.0.2.1
IP: tableid=0, s=198.51.100.1, d=192.0.2.1, routed via RIB
IP: s=198.51.100.1, d=192.0.2.1, g=192.0.2.253, len 100, forward
```

See if you can work through the following example on your own, explaining each step, for the second host `192.0.2.2`. Say each step aloud or hand-write it on paper. Mastering the NAT order of operations is essential:

```
IP: tableid=0, s=192.0.2.2, d=198.51.100.2, routed via RIB
NAT: s=192.0.2.2->203.0.113.2, d=198.51.100.2
IP: s=203.0.113.2, d=198.51.100.2, g=203.0.113.253, len 100, forward
NAT*: s=198.51.100.2, d=203.0.113.2->192.0.2.2
IP: tableid=0, s=198.51.100.2, d=192.0.2.2, routed via RIB
IP: s=198.51.100.2, d=192.0.2.2, g=192.0.2.253, len 100, forward
```

The main drawbacks of this solution are scalability and manageability. A network with 10,000 hosts would require 10,000 dedicated addresses with no sharing, which limits scale. Managing all these NAT statements, made massively

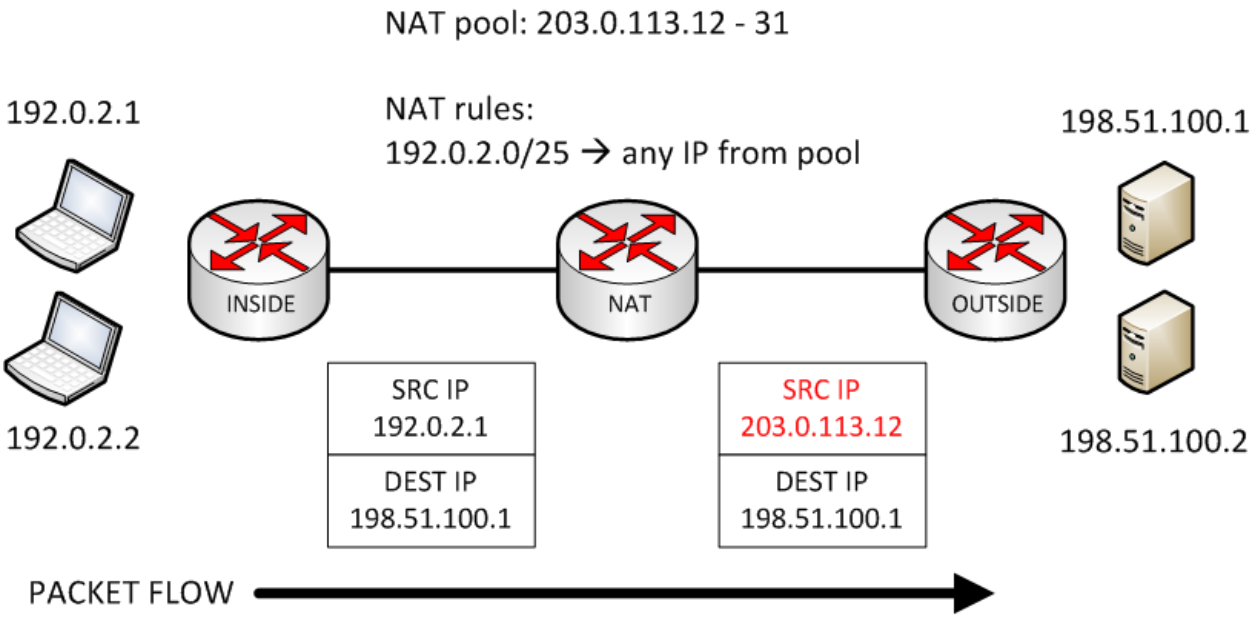
simpler with modern network automation, is still burdensome as someone must still maintain the mapping source of truth.

Dynamic One-to-one (1:1) Inside Source NAT

Rather than identify every inside global address and its corresponding mapping to an inside local address, one can use a NAT pool to define a collection of addresses for dynamic allocation when traffic traverses the NAT. It has the following benefits: * Arbitrary pool size coupled with arbitrary inside host list * Easy management and configuration * Dynamically-allocated state

When traffic traverses NAT, only the source IP changes. In this example, inside hosts within `192.0.2.0/25` are dynamically mapped to an available address in the pool `203.0.113.0/27`. Note that the two network masks do not need to match, and that the ACL can match any inside local address. Cisco IOS syntax is below:

```
# ACL defines the inside local addresses (pre NAT source)
ip access-list standard ACL_INSIDE_SOURCES
 permit 192.0.2.0 0.0.0.127

# Pool defines the inside global addresses (post NAT source)
ip nat pool POOL_203 203.0.113.12 203.0.113.31 prefix-length 27

# Binds the inside local list to the inside global pool for translation
ip nat inside source list ACL_INSIDE_SOURCES pool POOL_203
```



The output from NAT and IP packet debugging shows an identical flow from the previous section. The process has not changed, but the manner in which inside local addresses are mapped to inside global addresses is fully dynamic:

```
IP: tableid=0, s=192.0.2.1, d=198.51.100.1, routed via RIB
NAT: s=192.0.2.1->203.0.113.12, d=198.51.100.1
IP: s=203.0.113.12, d=198.51.100.1, g=203.0.113.253, len 100, forward
NAT: s=198.51.100.1, d=203.0.113.12->192.0.2.1
IP: tableid=0, s=198.51.100.1, d=192.0.2.1, routed via RIB
IP: s=198.51.100.1, d=192.0.2.1, g=192.0.2.253, len 100, forward
```

Like the static 1:1 solution, this solution still requires a large number of publicly routable (post NAT) addresses. Though the administrator does not have to manage the addresses individually, simply obtaining public IPv4 addresses

is a challenge given its impending exhaustion. The solution can optionally allow outside-to-inside traffic but only after the NAT state has already been created. The solution does not provide traceability between inside local and inside global *unless* the administrator specifically captures the NAT state table through shell commands, logging, or some other means. the

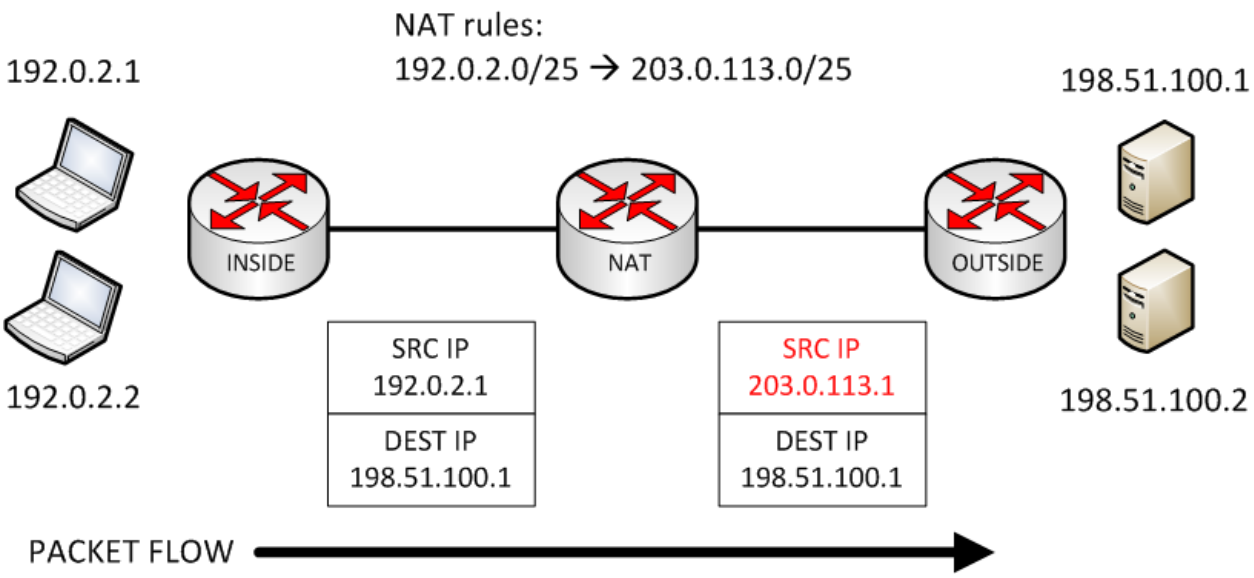
Static One-to-one (1:1) Inside Network NAT

This option is a hybrid of the previous static and dynamic 1:1 NAT techniques. The solution requires one inside local prefix and one inside global prefix, both of the same prefix length, to be mapped at once. The solution can be repeated for multiple inside local/global prefix pairs.

It has the best of both worlds: * Inside host should not or cannot share a source IP with other hosts * Custom IP protocols are used (not TCP, UDP, or ICMP) * Outside-to-inside unsolicited traffic flow is required * Address traceability is required, typically for security audits * Easy management and configuration * Dynamically-allocated state

When traffic traverses NAT, only the source IP changes. In this example, inside hosts within `192.0.2.0/25` are statically mapped to their matching address in the pool `203.0.113.0/25`. The network masks *must* match. Cisco IOS syntax is below:

```
# ip nat inside source static network (inside_local) (inside_global) (pfx_len)
ip nat inside source static network 192.0.2.0 203.0.113.0 /25
```



The output from NAT and IP packet debugging shows an identical flow from the previous section:

```
IP: tableid=0, s=192.0.2.1, d=198.51.100.1, routed via RIB
NAT: s=192.0.2.1->203.0.113.1, d=198.51.100.1
IP: s=203.0.113.1, d=198.51.100.1, g=203.0.113.253, len 100, forward
NAT: s=198.51.100.1, d=203.0.113.1->192.0.2.1
IP: tableid=0, s=198.51.100.1, d=192.0.2.1, routed via RIB
IP: s=198.51.100.1, d=192.0.2.1, g=192.0.2.253, len 100, forward

IP: tableid=0, s=192.0.2.2, d=198.51.100.2, routed via RIB
NAT: s=192.0.2.2->203.0.113.2, d=198.51.100.2
IP: s=203.0.113.2, d=198.51.100.2, g=203.0.113.253, len 100, forward
NAT: s=198.51.100.2, d=203.0.113.2->192.0.2.2
```

(continues on next page)

(continued from previous page)

```
IP: tableid=0, s=198.51.100.2, d=192.0.2.2, routed via RIB
IP: s=198.51.100.2, d=192.0.2.2, g=192.0.2.253, len 100, forward
```

Static Many-to-one (N:1) Inside Source NAT (Port Forwarding)

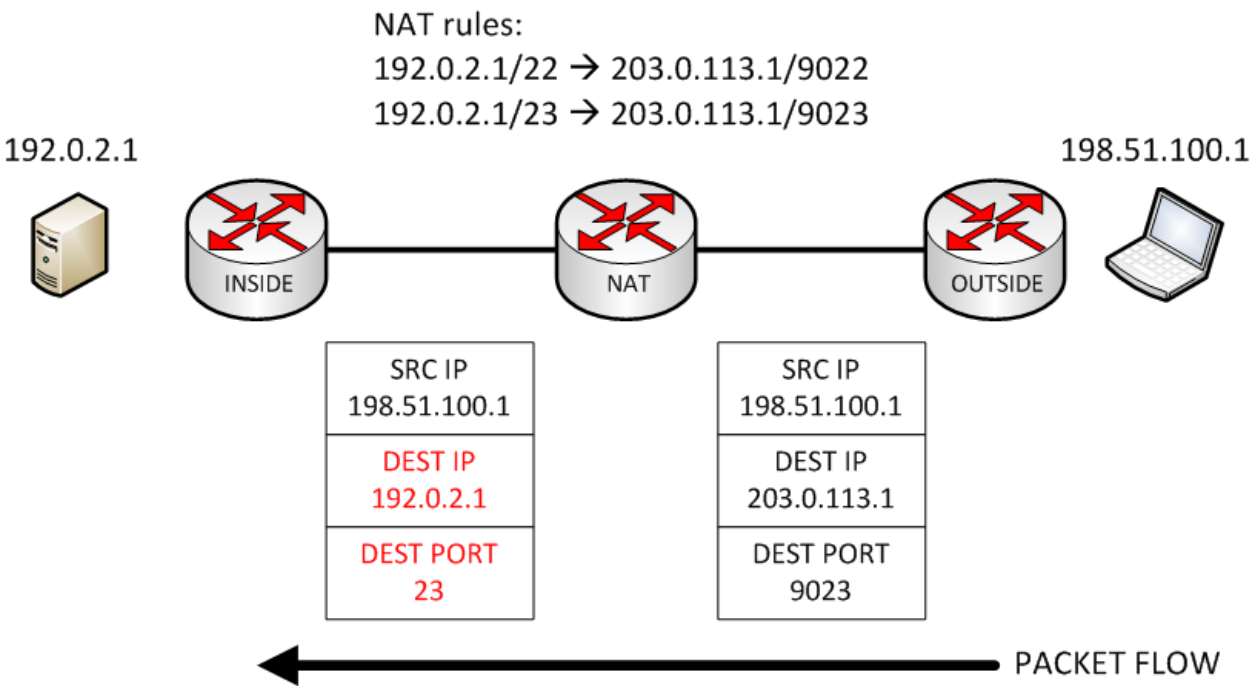
This technique is used to provide overloaded outside-to-inside access using TCP or UDP ports. It's particularly useful for reaching devices typically hidden behind NAT that need to receive unsolicited traffic.

In these examples, telnet (TCP 23) and SSH (TCP 22) access is needed from the outside network towards `192.0.2.1`. To reach this inside local address, outside clients will target `203.0.113.1` using ports 9022 for SSH and 9023 for telnet, respectively. Cisco IOS syntax is below:

```
ip nat inside source static tcp 192.0.2.1 22 203.0.113.1 9022
ip nat inside source static tcp 192.0.2.1 23 203.0.113.1 9023
```

Generating traffic to test this solution requires more than a simple `ping`. From the outside, a user telnets to `203.0.113.1` on port 9023 which the NAT device "forwards" to `192.0.2.1` on port 23. That is how this feature earned the name "port forwarding". The inside local device sees the session coming from `198.51.100.1`, the outside local address, which in this example is untranslated:

```
R3#telnet 203.0.113.1 9023
Trying 203.0.113.1, 9023 ... Open
Username: nick
Password: nick

R1#who
    Line       User       Host(s)    Idle       Location
   0 con 0                 idle       00:00:47
* 98 vty 0     nick       idle       00:00:00 198.51.100.1
```



Enabling NAT (`debug ip nat`) and IP packet (`debug ip packet`) debug reveals how this technology works.

The NAT process is similar but starts from the outside.

1. Initial packet arrives from `198.51.100.1` to `192.0.2.1` (not in debug)
2. NAT device translates destination port from 9023 to 23
3. NAT device translates destination from `203.0.113.1` to `192.0.2.1`
4. NAT device perform routing lookup towards `192.0.2.1`
5. NAT device forwards packet to `192.0.2.1` in the inside network
6. Return packet arrives from `192.0.2.1` to `198.51.100.1` (not in debug)
7. NAT device performs routing lookup towards `198.51.100.1`
8. NAT device translates source port from 23 to 9023
9. NAT device translates source from `192.0.2.1` to `203.0.113.1`
10. NAT device forwards packet to `198.51.100.1` in the outside network

Device output:

```
NAT: TCP s=37189, d=9023->23
NAT: s=198.51.100.1, d=203.0.113.1->192.0.2.1
IP: tableid=0, s=198.51.100.1, d=192.0.2.1, routed via RIB
IP: s=198.51.100.1, d=192.0.2.1, g=192.0.2.253, len 42, forward
IP: tableid=0, s=192.0.2.1, d=198.51.100.1, routed via RIB
NAT: TCP s=23->9023, d=37189
NAT: s=192.0.2.1->203.0.113.1, d=198.51.100.1
IP: s=203.0.113.1, d=198.51.100.1, g=203.0.113.253, len 42, forward
```

The next example is almost identical except uses SSH. Use this opportunity to test your understanding by following the debug output and reciting the NAT order of operations:

```
R3#ssh -l nick -p 9022 203.0.113.1

Password: nick

R1#who
    Line       User       Host(s)              Idle       Location
   0 con 0                 idle                 00:02:23
* 98 vty 0     nick       idle                 00:00:00 198.51.100.1

  Interface    User               Mode         Idle     Peer Address

NAT: TCP s=20534, d=9022->22
NAT: s=198.51.100.1, d=203.0.113.1->192.0.2.1 [21542]
IP: tableid=0, s=198.51.100.1, d=192.0.2.1, routed via RIB
IP: s=198.51.100.1, d=192.0.2.1, g=192.0.2.253, len 40, forward
IP: tableid=0, s=192.0.2.1, d=198.51.100.1, routed via RIB
NAT: TCP s=22->9022, d=20534
NAT: s=192.0.2.1->203.0.113.1, d=198.51.100.1
IP: s=203.0.113.1, d=198.51.100.1, g=203.0.113.253, len 268, forward
```

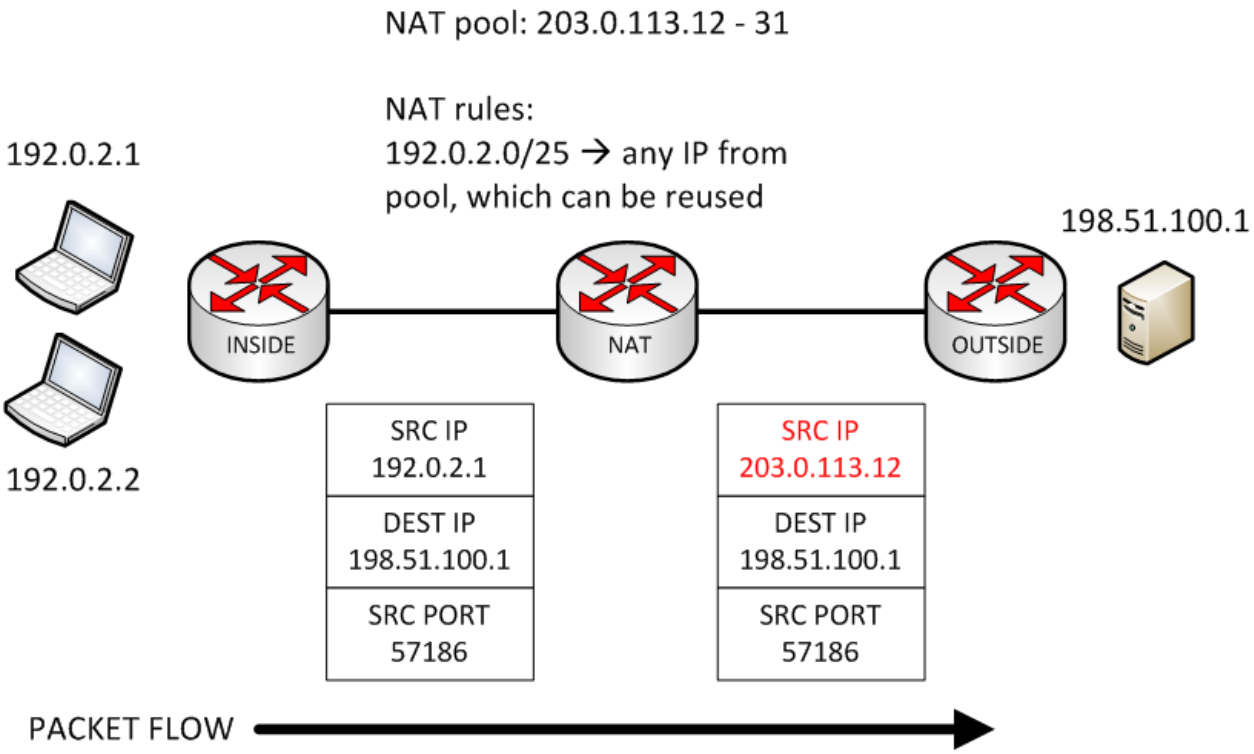
Dynamic Many-to-one (N:1) Inside Source NAT

This type of NAT is the most commonly deployed. Almost every consumer Internet connection has a LAN network for wired/wireless access. All hosts on this segment are translated to a single source IP address by using layer-4 source

port overloading. The changed source port serves as the demultiplexer to translate return traffic back to the proper source IP address.

This solution is also called NAT overload, Port Address Translation (PAT), or Network Address/Port Translation (NAPT). The solution can consume a NAT pool (range) but can reuse inside global addresses in the pool across many inside local addresses:

```
# ACL defines the inside local addresses (pre NAT source)
ip access-list standard ACL_INSIDE_SOURCES
 permit 192.0.2.0 0.0.0.127

# Pool defines the inside global addresses (post NAT source)
ip nat pool POOL_203 203.0.113.12 203.0.113.31 prefix-length 27

# Binds the inside local list to the inside global pool for translation
# Addresses allocated from the pool can be re-used
ip nat inside source list ACL_INSIDE_SOURCES pool POOL_203 overload
```



In order to see this solution, `debug ip nat detailed` is used to more explicitly show the inside and outside packets and their layer-4 ports. The first example uses telnet from `192.0.2.1` to `198.51.100.1`.

1. Initial packet arrives from `192.0.2.1` to `198.51.100.1` (not in debug)
2. NAT device performs routing lookup towards `198.51.100.1`
3. NAT device identifies inside translation using source port 57186.
4. NAT device translates source from `192.0.2.1` to `203.0.113.12`
5. NAT device forwards packet to `198.51.100.1` in the outside network
6. Return packet arrives from `198.51.100.42` to `203.0.113.1` (not in debug)
7. NAT device identifies outside translation using destination port 57186.

8. NAT device translates destination from `203.0.113.12` to `192.0.2.1`
9. NAT device perform routing lookup towards `192.0.2.1`
10. NAT device forwards packet to `192.0.2.1` in the inside network

Device output below explains how this works. Note that TCP port 23 is irrelevant for this type of NAT because only the source port matters:

```
IP: tableid=0, s=192.0.2.1, d=198.51.100.1, routed via RIB
NAT: i: tcp (192.0.2.1, 57186) -> (198.51.100.1, 23)
NAT: s=192.0.2.1->203.0.113.12, d=198.51.100.1
IP: s=203.0.113.12, d=198.51.100.1, g=203.0.113.253, len 42, forward
NAT: o: tcp (198.51.100.1, 23) -> (203.0.113.12, 57186)
NAT: s=198.51.100.1, d=203.0.113.12->192.0.2.1
IP: tableid=0, s=198.51.100.1, d=192.0.2.1, routed via RIB
IP: s=198.51.100.1, d=192.0.2.1, g=192.0.2.253, len 42, forward
```

The power of the solution is illustrated in the output below. A new source, `192.0.2.2` also initiates a telnet session to `198.51.100.1` and is translated to the same inside global address `203.0.113.12` except has a source port of 55943. Try to follow the order of operations:

```
IP: tableid=0, s=192.0.2.2, d=198.51.100.1, routed via RIB
NAT: i: tcp (192.0.2.2, 55943) -> (198.51.100.1, 23)
NAT: s=192.0.2.2->203.0.113.12, d=198.51.100.1
IP: s=203.0.113.12, d=198.51.100.1, g=203.0.113.253, len 42, forward
NAT: o: tcp (198.51.100.1, 23) -> (203.0.113.12, 55943)
NAT: s=198.51.100.1, d=203.0.113.12->192.0.2.2
IP: tableid=0, s=198.51.100.1, d=192.0.2.2, routed via RIB
IP: s=198.51.100.1, d=192.0.2.2, g=192.0.2.253, len 42, forward
```

The solution, while very widely used, has many drawbacks:

- Many hosts use a common IP address; removes end-to-end addressing
- Applications that require source ports to remain unchanged may not work. The NAT would have to retain source ports, which assumes inside local devices never use the same source port for inside-to-outside flows.
- It only works TCP and UDP traditionally, but most implementations also support ICMP. Protocols like GRE, IPv6-in-IPv4, and L2TP do not work.

Twice NAT

This NAT technique is by far the most complex. It is generally limited to in environments where there are overlapping IPs between two hosts that must communicate directly. This can occur between RFC 1918 addressing when organizations undergo mergers and acquisitions.

In this example, an inside host `192.0.2.1` needs to communicate to an outside host `198.51.100.1`. Both of these problems exist:

1. There is already a host with IP `198.51.100.1` on the inside network
2. There is already a host with IP `192.0.2.1` on the outside network

To make this work, each host must see the other as some other address. That is, both the inside source and outside source must be concurrently translated, hence the name “twice NAT”. This should not be confused with “double NAT” which is discussed in the CGN section.

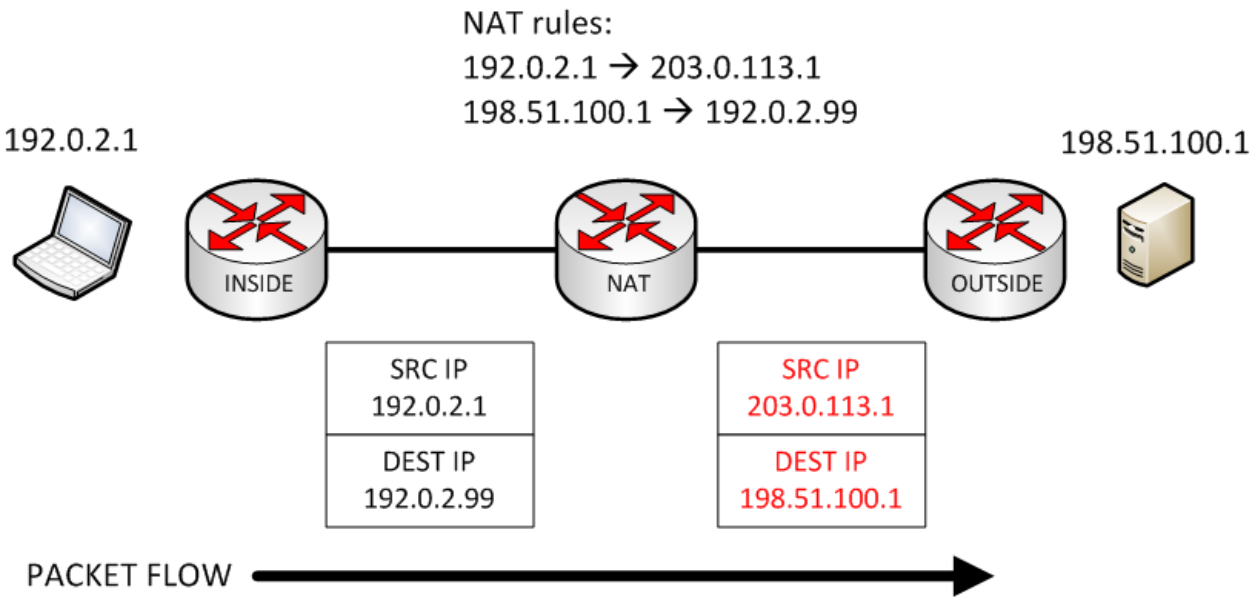
Twice NAT is where all four of the NAT address types are different:

- inside local: `192.0.2.1`

- inside global: `203.0.113.1`
- outside global: `198.51.100.1`
- outside local: `192.0.2.99`

Cisco IOS example syntax:

```
# ip nat inside source static (inside_local) (inside_global)
ip nat inside source static 192.0.2.1 203.0.113.1

# ip nat outside source static (outside_global) (outside_local)
ip nat outside source static 198.51.100.1 192.0.2.99 add-route
```



The order of operations is similar to previous inside source NAT examples, except that following every source translation is a destination translation.

1. Initial packet arrives from `192.0.2.1` to `192.0.2.99` (not in debug)
2. NAT device performs routing lookup towards `192.0.2.99`
3. NAT device translates source from `192.0.2.1` to `203.0.113.1`
4. NAT device translates destination from `192.0.2.99` to `198.51.100.1`
5. NAT device forwards packet to `198.51.100.1` in the outside network
6. Return packet arrives from `198.51.100.1` to `203.0.113.1` (not in debug)
7. NAT device translates source from `198.51.100.1` to `192.0.2.99`
8. NAT device translates destination from `203.0.113.1` to `192.0.2.1`
9. NAT device perform routing lookup towards `192.0.2.1`
10. NAT device forwards packet to `192.0.2.1` in the inside network

Device output showing the order of operations is below:

```
IP: tableid=0, s=192.0.2.1, d=192.0.2.99, routed via RIB
NAT: s=192.0.2.1->203.0.113.1, d=192.0.2.99 [40]
NAT: s=203.0.113.1, d=192.0.2.99->198.51.100.1 [40]
```

(continues on next page)

(continued from previous page)

```
IP: s=203.0.113.1, d=198.51.100.1, g=203.0.113.253, len 100, forward
NAT*: s=198.51.100.1->192.0.2.99, d=203.0.113.1 [40]
NAT*: s=192.0.2.99, d=203.0.113.1->192.0.2.1 [40]
IP: tableid=0, s=192.0.2.99, d=192.0.2.1, routed via RIB
IP: s=192.0.2.99, d=192.0.2.1, g=192.0.2.253, len 100, forward
```

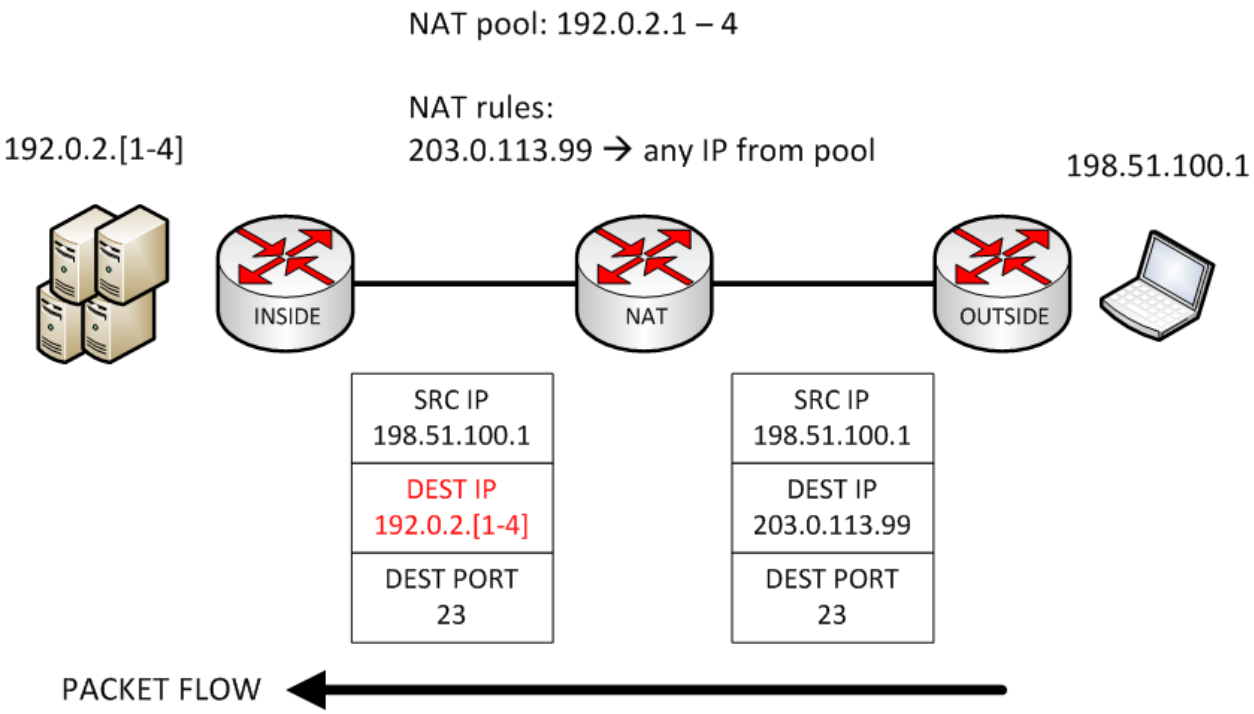
NAT as a crude load balancer

This use case is uncommonly used but does, at a basic level, represent how a server load balancer might work. It combines the logic of port forwarding (unsolicited outside-to-inside access) with a rotary NAT pool to create a dynamic solution whereby external clients can access internal hosts in round-robin fashion, typically for load balancing.

Cisco IOS syntax example:

```
# Define a list of virtual IPs for outside-to-inside access
ip access-list standard ACL_VIRTUAL_IP
 permit 203.0.113.99

# Define the inside local "servers" in the pool
ip nat pool POOL_192 192.0.2.1 192.0.2.4 prefix-length 29 type rotary

# Bind the virtual IP list to the server pool
ip nat inside destination list ACL_VIRTUAL_IP pool POOL_192
```

Each time the outside host with IP `198.51.100.1` telnets to the virtual IP address used to represent the server pool of `203.0.113.99`, the NAT device selects a different inside local address for the destination of the connection. The range of inside IP addresses goes from `192.0.2.1` to `192.0.2.4`, which are actual server IP addresses behind the NAT.



The debug is shown below with RIB/packet forwarding output excluded for clarity as it reveals nothing new. Each block of output is from a separate telnet session, and represents a single keystroke:

```
# Choose server 192.0.2.1
NAT*: s=198.51.100.1, d=203.0.113.99->192.0.2.1
NAT*: s=192.0.2.1->203.0.113.99, d=198.51.100.1

# Choose server 192.0.2.2
NAT*: s=198.51.100.1, d=203.0.113.99->192.0.2.2
NAT*: s=192.0.2.2->203.0.113.99, d=198.51.100.1

# Choose server 192.0.2.3
NAT*: s=198.51.100.1, d=203.0.113.99->192.0.2.3
NAT*: s=192.0.2.3->203.0.113.99, d=198.51.100.1

# Choose server 192.0.2.4
NAT*: s=198.51.100.1, d=203.0.113.99->192.0.2.4
NAT*: s=192.0.2.4->203.0.113.99, d=198.51.100.1
```

Carrier Grade NAT (CGN)

This NAT technique does not introduce any new technology as it usually refers to simply chaining PAT solutions (dynamic inside source NAT) in series to create an aggregated, more overloaded design. Consider the home network of a typical consumer that has perhaps 5 IP-enabled devices on the network concurrently. Each device may have 50 connections for a total of 250 flows. A single IP address could theoretically support more than 65,000 flows using PAT given the expanse of the layer-4 port range. Aggregating flows at every higher point in a hierarchical NAT design helps achieve greater NAT efficiency/density. Thus, given ~250 flows per home, a single CGN could perform NAT aggregation services for ~260 homes to result in ~65,000 ports utilized from a single address.

The purpose of CGN has always been to slow to exhaustion of IPv4 addresses, buying time for a proper IPv6 deployment. The vast majority of network engineers acknowledge that CGN is not a permanent solution.

CGN is also known as Large Scale NAT (LSN), LSN444, NAT444, hierarchical NAT, and double NAT. Do not confuse CGN with "twice NAT" which is typically a single NAT point where the source *and* destination are both translated.

Cisco IOS sample syntax on the home router (first NAT point):

```
# ACL defines the inside local addresses (pre NAT source)
ip access-list standard ACL_INSIDE_SOURCES
 permit 192.0.2.0 0.0.0.127

# Use the single outside IP (often a DHCP address) to translate inside hosts
ip nat inside source list ACL_INSIDE_SOURCES interface FastEthernet0/1 overload
```

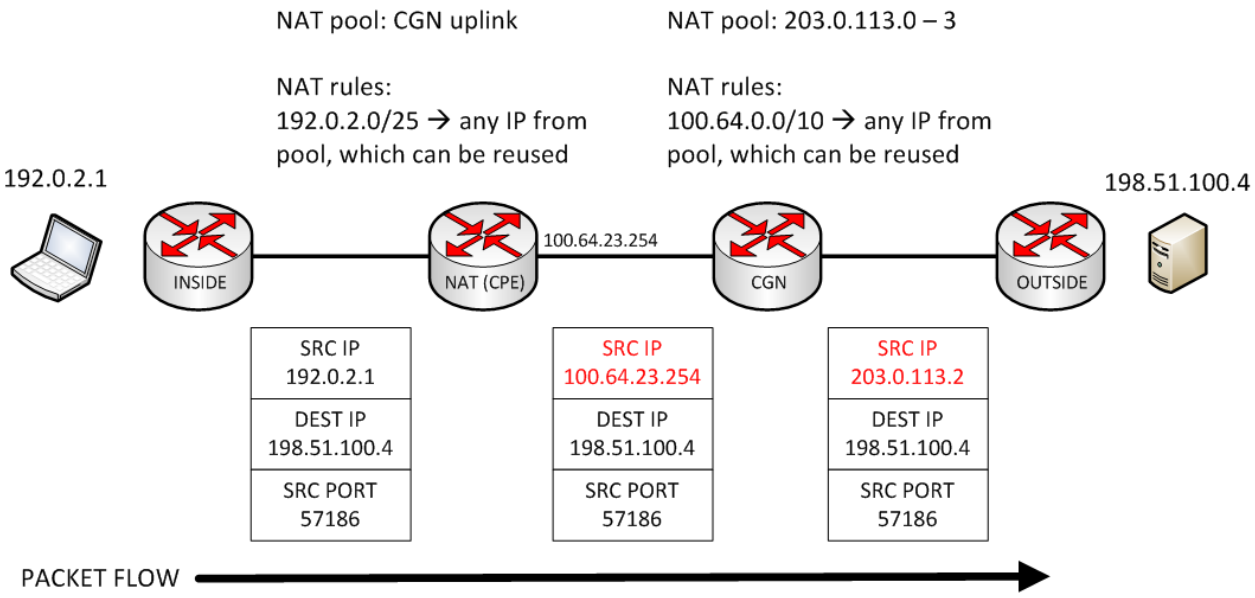
Cisco IOS sample syntax on the CGN router (second NAT point):

```
# Contains the inside global addresses (CGN range) from the home routers
ip access-list standard ACL_CONSUMER_ROUTERS
 permit 100.64.0.0 0.63.255.255

# Create NAT pool, typically a small number of public IPs for the CGN domain
ip nat pool POOL_CGN 203.0.113.0 203.0.113.3 prefix-length 30

# Bind the home routers using CGN 100.64.0.0/10 space to the CGN pool
ip nat inside source list ACL_CONSUMER_ROUTERS pool POOL_CGN overload
```

Given that no new technology is introduced, the debug below can be summarized in 4 main steps. The debugs are broken up to help visualize the NAT actions across multiple devices along the packet's journey upstream and downstream.

1. Inside-to-outside NAT at home router

2. Inside-to-outside NAT at CGN router
3. Outside-to-inside NAT at CGN router
4. Outside-to-inside NAT at home router



Device output:

```
# Inside-to-outside NAT at home router
IP: tableid=0, s=192.0.2.1, d=198.51.100.4, routed via RIB
NAT: s=192.0.2.1->100.64.23.254, d=198.51.100.4
IP: s=100.64.23.254, d=198.51.100.4, g=100.64.23.253, len 100, forward

# Inside-to-outside NAT at CGN router
IP: tableid=0, s=100.64.23.254, d=198.51.100.4, routed via RIB
NAT: s=100.64.23.254->203.0.113.2, d=198.51.100.4
IP: s=203.0.113.2, d=198.51.100.4, g=203.0.113.254, len 100, forward

# Outside-to-inside NAT at CGN router
NAT*: s=198.51.100.4, d=203.0.113.2->100.64.23.254
IP: tableid=0, s=198.51.100.4, d=100.64.23.254, routed via RIB
IP: s=198.51.100.4, d=100.64.23.254, g=100.64.23.254, len 100, forward

# Outside-to-inside NAT at home router
NAT*: s=198.51.100.4, d=100.64.23.254->192.0.2.1
IP: tableid=0, s=198.51.100.4, d=192.0.2.1, routed via RIB
IP: s=198.51.100.4, d=192.0.2.1, g=192.0.2.253, len 100, forward
```

Like most "quick fix" solutions, CGN has serious drawbacks:

- Massive capital investment for specialized hardware to perform high-density NAT with good performance and reliability.
- Burdensome regulatory requirements on NAT logging. If ~260 households all share a single public IP, criminal activity tracking becomes more difficult. Carriers are often required to retain NAT logging, along with consumer router IP addressing bindings, to identify where malicious activity originates. This comes with additional capital and operating expense.
- ALG support is already hard with NAT. With CGN, almost impossible.

- No possibility of outside-to-inside traffic, even with port forwarding, for customers.
- No possibility of peer-to-peer applications working.
- Restricted ports per consumer (250 in the theoretical example above)

In closing, developers should build applications that can use IPv6, totally obviating the complex and costly workarounds need to get them working across CGN in many cases. Applications that can be dual-stacked from the beginning provide an easy migration path to IPv6.

NAT as a security tool

This is a topic of much debate, and there are two sides to the coin.

NAT provides the following security advantages:

- No reachability: In one-to-many NAT scenarios where unsolicited outside-to-inside flows are technically impossible, the inside hosts are insulated from direct external attacks.
- Obfuscation: The original IP address of the inside host is never revealed to the outside world, making targeted attacks more difficult.
- Automatic firewalling: As a stateful device, only outside-to-inside flows already in the state table are allowed, and the inside-to-outside NAT state is created based on an ACL, much like a firewall.

Many of these security advantages are easily defeated:

- Relatively simple attacks like phishing and social engineering cause clients to initiate connections to the outside. Such attacks are easy to stage and more effective than outside-in frontal assaults.
- Most applications don't use their IP addresses as identification. A web application might have usernames or a digital certificate. The IP address itself is mostly irrelevant and not worth protecting. It's entirely irrelevant when clients receive IP addresses dynamically (via DHCP).
- A NAT device drops outside-in flows due to lack of state, but does not provide any protection against spoofed, replayed, or otherwise bogus packets that piggy-back on existing flows. Security appliances do. The only similarity between NATs and firewalls is that they maintain state. They should not be categorized in any other way.

In closing, do not rely on NAT as a real security technology. It's roughly equivalent to closing your blinds in your home. It adds marginal security value but there are clearly better alternatives. Like blinds on windows, NAT may conceal and slow down some attacks, but should never be confused for a legitimate security component in the design.

Section author: Nick Russo <njrusmc@gmail.com>

6.7 IPv6

6.8 Routing

6.9 Ping and Traceroute

6.10 Troubleshooting

CHAPTER 7

Services

This part of the site is about services such as DNS, DHCP and more.

7.1 Further Reading

CHAPTER 8

Network Design

This part of the site is about network design, i.e. network design principles around high availability, resiliency, modularity, scalability and more.

8.1 Further Reading

CHAPTER 9

Cloud Networking

This part of the site is about cloud networking, e.g. AWS, Azure, GCP or other companies leveraging cloud computing.

9.1 AWS

Amazon Web Services (AWS) is at the time of this writing (`2018-09-04`) the largest provider of cloud services in the world. AWS offers a breadth of services in both compute, storage, databases, serverless, CDN and much more. The purpose of this section is to describe the networking services in AWS and an overview of AWS networking.

9.1.1 Networking in AWS Overview

Networking in public cloud is quite different from an on-premises network. In order to build a multi-tenant, scalable, fault tolerant and high speed network, AWS has designed the network quite differently than a "traditional" network. The most notable difference is:

- No support for multicast
- No support for broadcast

What this means is that routing protocols using multicast are not supported without some form of tunneling but more importantly this means that because broadcast is not supported, ARP requests are not flooded in the network. Without ARP, how can a host learn the MAC address of another host? This requires some magic. Which is performed by the hypervisor. When the hypervisor sees an ARP request, it will proxy the ARP response back to the host. More about the specifics of networking in AWS later.

9.1.2 AWS Fundamentals

Instance

With AWS terminology, a virtual machine (VM) is called an instance.

Availability Zone

An Availability Zone (AZ) is one or several datacenters in the same geographical area sharing infrastructure such as power and internet transports.

Region

A region is a collection of AZs. The number of regions is always growing and currently there are regions in North America, Europe, Asia, and South America.

All regions consist of at least two AZs, some three and some even four. To provide high availability, applications should be deployed in more than one AZ.

Some of AWS' services are regional and some are global.

VPC

Virtual Private Cloud (VPC) is one of the foundational building blocks of AWS. VPC is a virtual datacenter deployed in AWS' multi-tenant cloud. By default, all instances deployed in AWS, are deployed to a default VPC. The default VPC is designed to help organizations start quickly. For that reason, the instances will by default get a public IP assigned, and have a route to the internet by default. This is not true for customer created VPCs.

These are actions that are performed in the default VPC:

- Creates a VPC with a /16 CIDR block (172.31.0.0/16)
- Creates a /20 subnet in each AZ
- Creates an internet gateway and attaches it to the VPC
- Creates a route table with a default route (0.0.0.0/0) pointing to the IGW
- Creates a default security group (SG) and associates it with instances in the VPC
- Creates a Network ACL (NACL) and associates it with the subnets in the VPC
- Associates the default DHCP options set for the AWS account with the default VPC

The default security group has the following logic:

- Allows all inbound traffic from other instances associated with the same SG
- Allows all outbound traffic

The default NACL allows all traffic both inbound and outbound.

Subnet

When a VPC is created, it must have a subnet in order to be able to address the instances. For the default VPC, 172.31.0.0/16 is used. For customer created VPCs, the user will assign a subnet ranging from a /16 up to a /28.

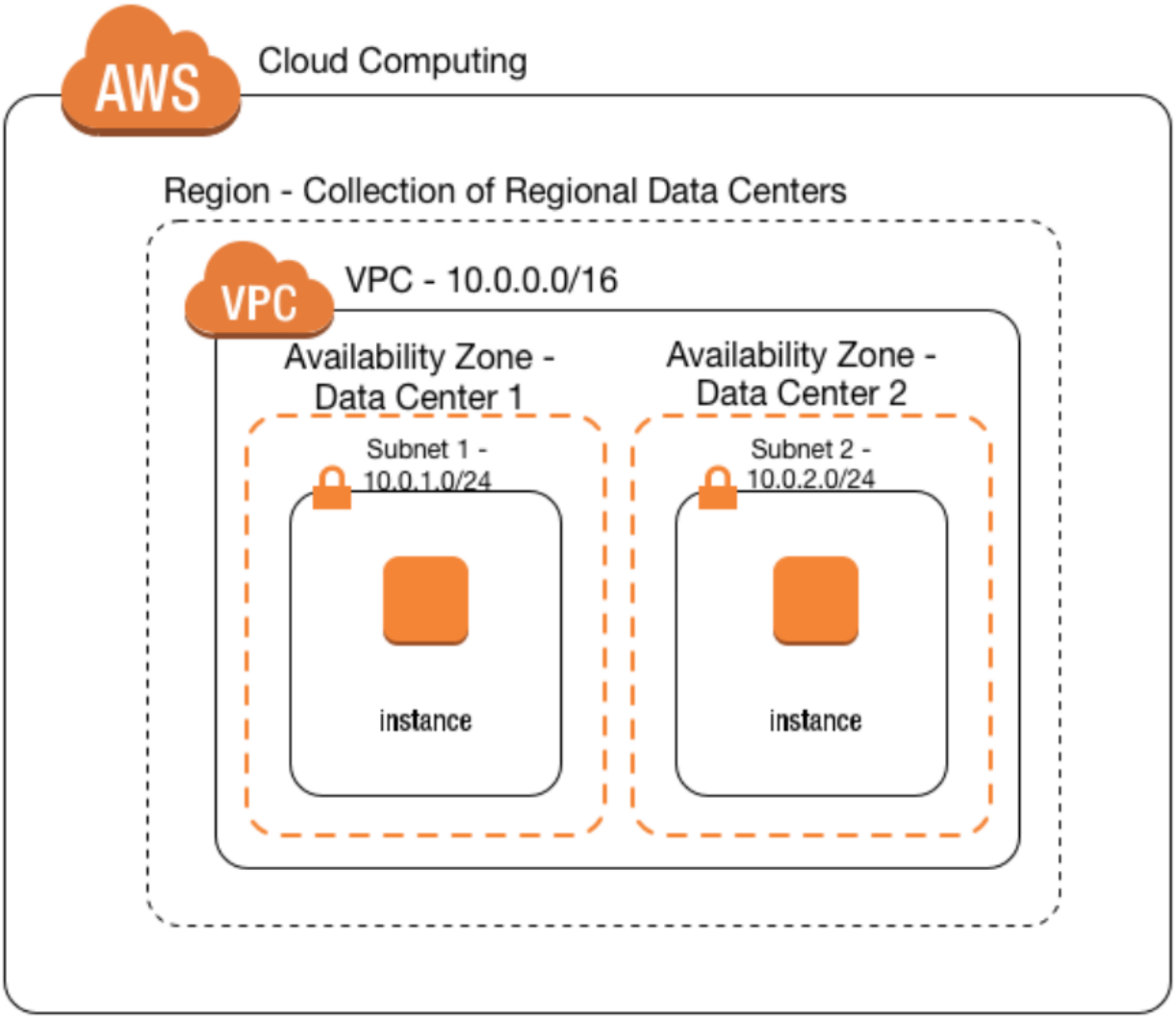
Subnets are always specific to an AZ, but there can be several subnets in an AZ.

Note that some addresses in a subnet are reserved for use by Amazon. For example, in the subnet 192.168.0.0/24, the following addresses would be reserved:

- 192.168.0.0 - Network address
- 192.168.0.1 - Reserved by AWS for VPC router
- 192.168.0.2 - Reserved by AWS for DNS server

- 192.168.0.3 - Reserved by Aws for future use
- 192.168.0.255 - Broadcast address

So what does a VPC look like? This image from Amazon's blog shows the VPC construct:



Route Table

Every subnet needs a route table, by default a subnet gets assigned to a default route table. It is also possible to create custom route tables. A route table can be associated with multiple subnets but every subnet can only belong to one route table.

It is common practice to have private and public subnets and using different route tables for each. The public subnet would then have a public IP or elastic IP and be reachable from the internet while the private subnet wouldn't be, at least not without first passing an NAT gateway or instance.

Note that routes always point to some form of gateway or endpoint, such as an internet gateway (IGW), NAT gateway, virtual gateway (VGW), EC2 instance, VPC peer or VPC endpoint and not towards an IP address directly.

It's important to know that forwarding can't be changed for local traffic, that is traffic inside of the VPC. It's not possible to override this behavior with a longer route so routing in a VPC is not strictly longest prefix matching (LPM).

Security Group

A security group is similar to a stateful ACL or a basic form of firewall, albeit distributed. The SG is applied to the instance directly and keeps track of TCP, UDP and ICMP connection status. Rules are white list only, consisting of ACL entries, and contain an implicity deny, meaning that anything not permitted, is denied. A SG can reference another SG, for example a database SG allowing any instance in the application SG to send traffic inbound on the port that the database service is exposing.

Note that an instance can be associated with several SGs, and that all rules would be evaluated before permitting or denying a flow. It's possible to change the SG applied to an instance after it has been launched.

Network ACL

The Network ACL (NACL) is a stateless ACL, applied at the subnet level. Logically, any traffic going to an instance, will be evaluated in the NACL first, before being evaluated by the SG. NACLs are ordered and can have both permit and deny statements. It's only possible to reference CIDR ranges and not instances or other ACLs.

For example, to deny all SSH traffic into a subnet, a NACL could block TCP port 22 inbound instead of putting rules in each SG.

Elastic Network Interface

The Elastic Network Interface (ENI) is a virtual network interface card (NIC). An instance can have multiple ENIs and an ENI can be moved to another instance within the same subnet. It's also possible to have multiple addresses assigned to an ENI. An ENI has a dynamically assigned private IP and optionally a public one as well. While the primary ENI assigned to an instance has dynamically allocated IP addresses, with a customly created ENI, there's flexibility to auto-assign or statically assign IP addresses within the given subnet. An instance primary ENI may also have one or more statically provided secondary addresses through the console or API.

Public IP Address

Public IP addresses are addresses that are reachable from the internet. AWS uses two types of public IP addresses:

- Dynamically assigned public IP addresses
- Elastic IP addresses

If the instance does not need to have a persistent public IP, it can use the public IP (dynamically assigned). This IP is however released if the instance terminates. For instances needing a persistent IP, use the Elastic IP instead. Note that this IP comes with a cost when not assigned to an instance.

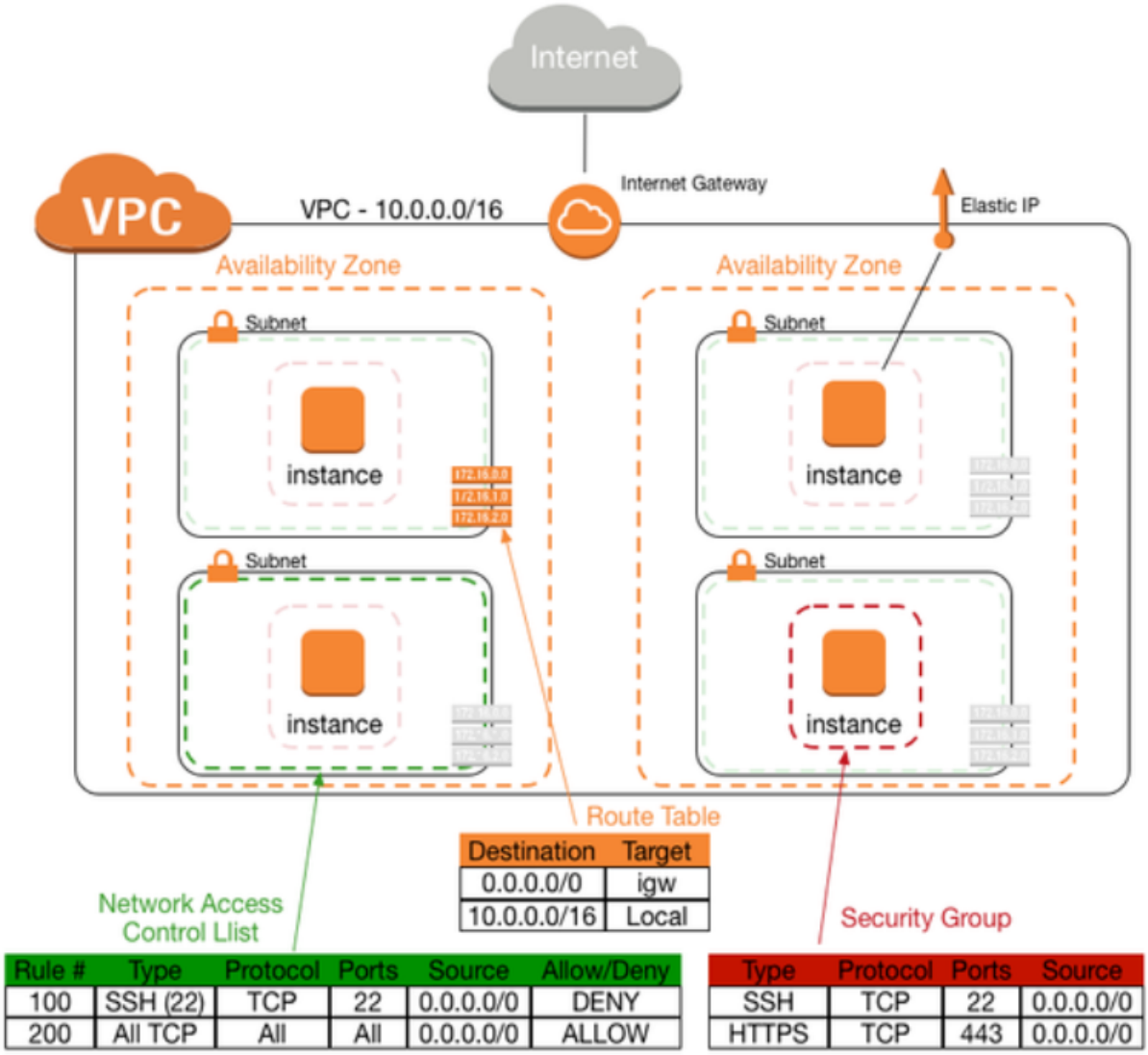
Elastic IP

The Elastic IP is not dynamically assigned but rather statically assigned to instances that need a persistent IP address. It can be associated with another instance if the first one is terminated.

Having a persistent IP can simplify writing ACLs, DNS entries and having the IP address whitelisted in other systems. Note that there is always a private IP assigned to the instance, this is the one that is seen when looking at the OS level. This holds true for both dynamically assigned addresses and the Elastic IP. The private IP address remains the same unless the EIP is moved to another instance.

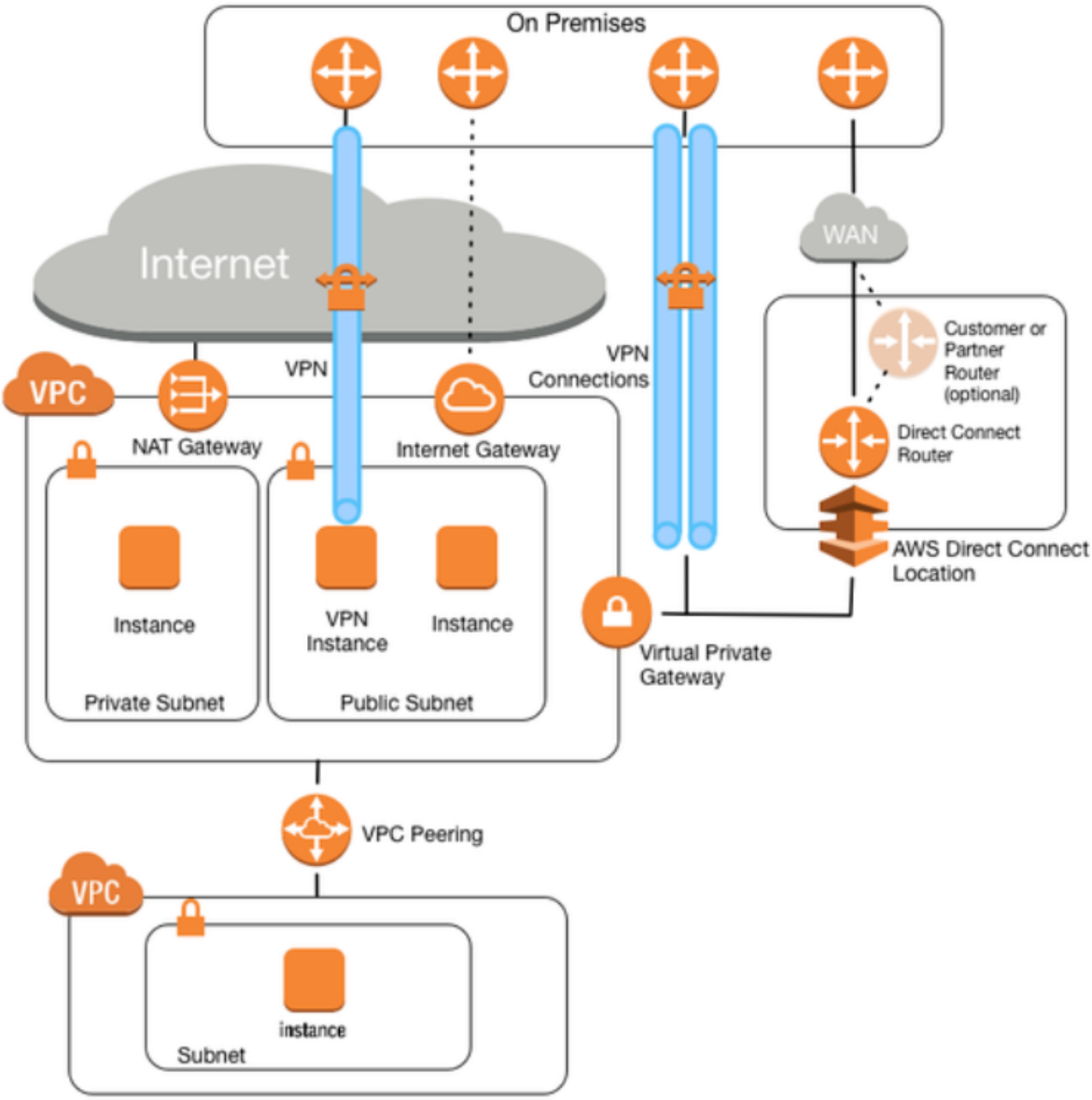
How can the instance have a private IP but still be publically accessible? NAT. AWS will perform 1:1 NAT so that the instance can be reached from the outside.

The VPC then, with added details, looks like below, once again the picture from Amazon's blog.



9.1.3 External Connectivity to the VPC

There are multiple ways to connect to a VPC but keep in mind that a VPC does not do transitive routing. This picture, from Amazon's blog, shows the different options of connecting to the VPC.



Internet Gateway

The internet gateway (IGW) provides access to the internet. This device is provided by Amazon, is highly available and is used by pointing a default route at the IGW. The IGW is a regional service.

NAT Gateway

The NAT gateway is a service provided per AZ, used for translating private IP addresses into public ones. This is mainly used for instances that need to reach the internet, for example for downloading software updates, but that should not be reachable from the outside. The devices having ther private IP addresses being translated, will share a single Elastic IP.

As noted above, because it is a service provided per AZ, if multiple AZs are leveraged, use one NAT gateway per AZ.

Virtual Private Gateway

The virtual private gateway (VGW) is a regional service, highly available, that terminates two AWS services:

- Virtual Private Networks (VPNs)
- AWS Direct Connect

The VGW terminates IPSec tunnels and can either use static routing or Border Gateway Protocl (BGP) for dynamic routing. It's also possible to propagate routes learned dynamically into the VPC.

Direct Connect circuits point to the VGW which is the entry point into the VPC.

Virtual Private Network

VPNs are provided through the VGW, as noted above or by having an instance terminate the VPN. For example by using AMIs from networking vendors such as Cisco, Palo Alto and others. A VGW can never setup a tunnel to another VGW. This is because a VGW is always only a responder, never the initiator.

Only 100 routes are supported to be added into the VPC through the use of VPNs.

Direct Connect (DX)

AWS Direct Connect provides the ability to establish a dedicated network connection from sites such as data centers, offices, or colocation enviroments to AWS. It provides a more consistent network experience than internet-based connections at bandwidths ranging from 50 Mbps to 10 Gbps on a single connection. Direct Connect (DX) has the following requirements :

- 802.1Q
- BGP
- BFD (Optional)

Virtual Interfaces (VIFs)

A VIF is a configuration consisting primarily of an 802.1Q VLAN and the options for an associated BGP Session. It contains all the configuration parameters required for both the AWS end of a connection and your end of the connection AWS Direct connect support two types of VIFs:

- Public VIFs
- Private VIFs

1. Public VIFs: Public Virtual interfaces enable your network to reach all of the AWS public IP addresses for the AWS region with which your AWS Direct Connect connection is associated. Public VIFs are typically used to enable direct network access to services that are not reachable via a private IP address within your own VPC. These include Amazon S3, Amazon DynamoDB and Amazon SQS.

2. Private VIFs: Private Virtual Interfaces enable your network to reach resources that have been provisioned within your VPC via their private IP address. A Private VIF is associated with the VGW for your VPC to enable this connectivity. Private VIFs are used to enable direct network access to services that are reachable via an IP address within your own VPC. These include Amazon EC2, Amazon RDS and Amazon Redshift.

802.1Q Virtual Local Area Networks (VLANs)

802.1Q is an Ethernet standard that enables Virtual Local Area Networks (VLANs) on an Ethernet network, it uses the addition of a VLAN tag to the header of an Ethernet frame to define membership of a particular VLAN.

Link Aggregation Groups (LAGs)

A LAG is a logical interface that uses the LACP (Link Aggregation Control Protocol) to aggregate multiple 1 Gbps or 10 Gbps connections at a single AWS Direct Connect location, allowing you to treat them as a single, managed connection.

Direct Connect Gateway

A Direct Connect gateway enables you to combine private VIFs with multiple VGWs in local or in the remote regions. You can use this feature to establish connectivity from an AWS Direct Connect location in one geographical zone to an AWS region in a different geographical zone.

Bidirectional Forwarding Detection (BFD)

Bidirectional forwarding detection (BFD) is a network fault detection protocol that provides fast failure detection times, which facilitates faster re-convergence for dynamic routing protocols. It is a mechanism used to support fast failover of connections in the event of a failure in the forwarding path between two routers. If a failover occurs, then BFD notifies the associated routing prorocols to recalculate available routes.

Border Gateway Protocol

Border Gateway Prorocol (BGP) is a routing prorocol used to exchange network routing and reachability information, either within the same AS (iBGP) or a different autonomous system (eBGP).

BGP Best Path Selection (VGW)

The best path algorithm in AWS follows this logic:

1. Local routes to the VPC
2. Longest prefix match first
3. Static route table entries to:
   - IGW
   - VGW
   - Network interface
   - Instance ID
   - VPC peering
   - NAT gateway
   - VPC endpoint
4. VPN routes
   (a) Prefer Direct Connect BGP routes

(b) VPN static routes

(c) BGP Routes from VPN (not direct connect)

For BGP routes coming from VPN, the following priority applies:

1. Shortest AS-path
2. Lowest origin (IGP over EGP)

9.1.4 Other Interesting Information

Virtual Router

There is a "virtual" router attached to the VPC. In reality, this is a distributed service provided by both HW and a hypervisor, of course.

DNS in a VPC

DNS is handled by the VPC by default. Every instance is assigned a DNS name with its IP address, automatically managed by AWS. It's possible to use Route53 to setup a private zone as well or using an instance hosting the DNS service.

Be mindful that DNS service is only available to DNS clients within the same VPC. Although the "plus two" address (e.g. 172.31.0.2) for each VPC is reserved for DNS service by default, for simplicity Amazon also reserves the IP address *169.254.169.253* which can be used by any DNS client to reach the DNS service endpoint configured for its local VPC.

Routing and Switching

- AWS only supports unicast traffic, no multicast or broadcast. As described previously, ARP responses are proxied by the hypervisor
- Traffic between subnets in a VPC is allowed, unless restricted by a security group, NACL or filtered at the host
- Traffic between two instances in the same VPC is always local and can't be overridden with more specific routes
- Traffic that is not sourced by or destined to an instance is dropped, unless disabling the source/destination check
- Transitive routing is not supported, meaning that VPC A can't communicate with VPC C by going through VPC B
- Jumbo frames are supported inside of a VPC but traffic leaving a VPC always uses a Maximum Transmission Unit (MTU) of 1500 bytes

Section author: Daniel Dib

9.2 Azure

Microsoft Azure is a relative newcomer in cloud computing compared to AWS. Over the past couple of years Microsoft has done some incredible work with the Azure platform and open source community to become one of the leading cloud providers.

The services they offer include compute, storage, database, serverless, CDN and much more. The purpose of this section is to describe the networking services in Azure and an overview of Azure networking.

9.2.1 Networking in Azure Overview

Networking in public cloud is quite different from an on-premises network. In order to build a multi-tenant, scalable, fault tolerant and high speed network, Azure has designed the network quite differently than a "traditional" network. The most notable difference is:

- No support for multicast
- No support for broadcast

What this means is that routing protocols using multicast are not supported without some form of tunneling but more importantly this means that because broadcast is not supported, ARP requests are not flooded in the network. Without ARP, how can a host learn the MAC address of another host? This requires some magic. Which is performed by the hypervisor. When the hypervisor sees an ARP request, it will proxy the ARP response back to the host. More about the specifics of networking in Azure later.

9.2.2 Azure Fundamentals

Geopolitical Region

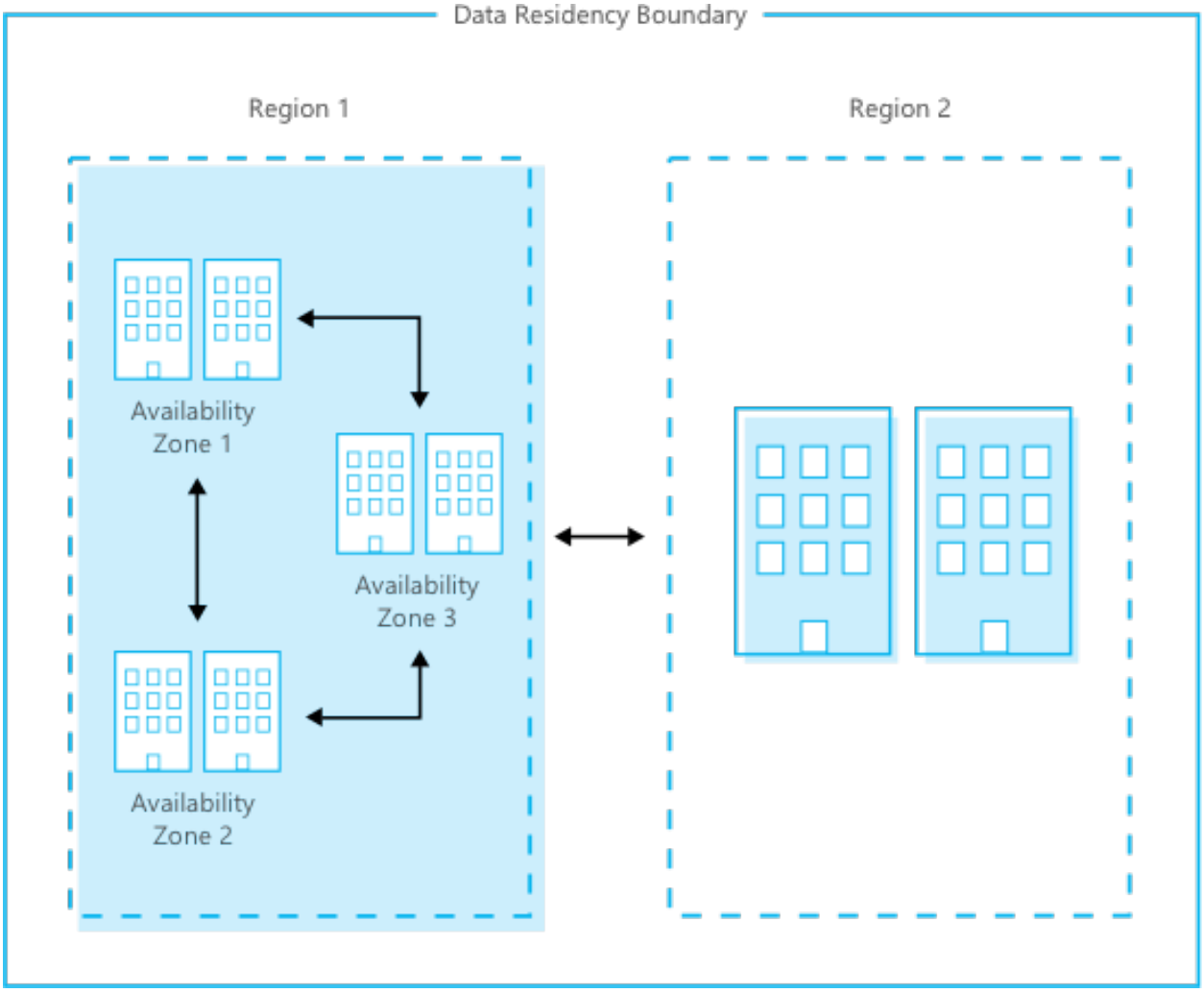
A Geopolitical region is a collection of Azure regions which can be reached from a standard ExpressRoute.

https://docs.microsoft.com/en-us/azure/expressroute/expressroute-locations has more details around this.

Regions & Availability Zones

A Region in Azure is one or several datacenters in the same geographical area sharing infrastructure such as power and internet transports.

Within a region there are multiple Availability Zones. Each Availability Zone is a separate physical location, for example a separate data centre room, or another building with very low latency to the other availability zones within the region.



https://azure.microsoft.com/en-gb/global-infrastructure/regions/

Resource Group (RG)

A resource group is a logical grouping of infrastructure resources. For example a Production RG may include:

- Production VNet
- Storage account(s) related to VMs
- VM Servers
- Azure Traffic Manager DNZ zones

VNet

A VNet is an IP Prefix allocated for use in Azure.

You can have multiple VNets within a subscription/resource group.

VNets are isolated instances by default, if you have multiple VNets within a region or globally they will be unable to communicate with each other.

VNet Peering, VPNs, or ExpressRoute can allow this communication.

Example of VNet assignments might be:

VNet	Assignment
10.0.0.0/16	Western Europe (Prod)
10.0.16.0/16	North Europe (DR)

Subnet

A subnet is an IP Prefix allocation within a VNet. Multiple Subnets are generally assigned within a VNet. Depending on the application deployment model you might want to assign a VNet for Management, Apps, Databases, etc…

Note: This is just an example and not a recommended subnet layout.

Subnet	Usage
10.0.0.0/24	GatewaySubnet
10.0.1.0/24	Management
10.0.2.0/24	Apps
10.0.3.0/24	Database

Network Security Group (NSG)

A NSG is a stateful firewall in Azure. They can be associated either with a specific VM, or with an entire subnet. See https://docs.microsoft.com/en-us/azure/virtual-network/security-overview for more information.

Network Virtual Appliance (NVA)

A Network Virtual Appliance (NVA) is the name for using a traditional network vendor VM in Azure.

For example you can deploy VMs from vendors such as Juniper VSRX firewalls, Palo Alto firewalls, Cisco CSR routers and integrate them into your standard networking toolset.

There is nothing special about these VMs, many people deploy Linux VMs to manage routing and firewalling.

User Defined Routes (UDR)

UDR is a routing table within Azure. It allows static routes to be defined and system Routes overwritten. UDRs are assigned at a subnet level, and you can assign the same UDR to multiple subnets. Generally you wouldn't use a UDR unless you're using a Network Virtual Appliance (NVA)

Section author: Daniel Rowe <n4d@danrowe.email>

CHAPTER 10

Datacenter Networking

This part of the site is about datacenter networking, i.e. networking technologies used in on-premises datacenters.

10.1 Further Reading

CHAPTER 11

Network Security

This part of the site is about network security such as firewalls, IDS/IPS, VPNs and more.

11.1 Further Reading