



# **NETWORKING**

**Professor Ken Birman**  
**CS4414 Lecture 20**

# IDEA MAP FOR TODAY

**The Internet, IP addresses and port numbers.  
Packets, routing, firewalls, tunnels, network address translation**

**TCP basics, SSL security**

**Socket API, Google GRPC**

**http versus https, VPNs, VPC**

# INTERNET BASICS

The internet is like a computerized postal system.

Any process can set up a “mailbox” (a **socket**), and post an address on it (**bind** an IP address and port number).

The address can then be registered for use by programs anywhere on the net... with limitations

# IP ADDRESSES



Mailroom at 11 Riverside Drive, NYC

IPv4 is 32-bits (but only 28 are useable). IPv6 doubles this.

- There are actually several types of addresses (classes)
- In CS4414 we won't dive into why, or what the others are for

Each address has an associated “port number” because one machine could have many processes using the network. Like a mailbox in an apartment building.

- The IP address is used to route to the computer. Like a street address.
- The port number is used to figure out which process gets the packet.

# ADDRESS DIRECTORY: DNS

The domain name service (DNS) map server names (www.cornell.edu) to IP addresses (128.253.173.247).

The DNS is operated by for-profit companies. They sell domain addresses, and server owners pay for listings.

Fancy web sites, like Netflix.com, have ways to route you to a data center somewhere near you, for speed.

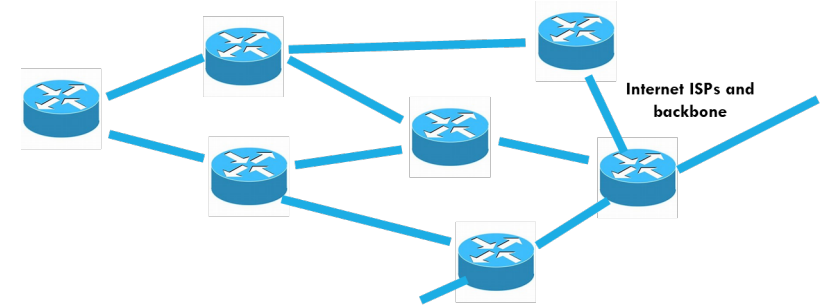
# A SINGLE MACHINE CAN HOST MANY NETWORK SERVICES

To distinguish them, we use a “port number”

This is like an apartment number in a big building.

Some port numbers are standard, like 80 (for web services).  
Others are allocated on demand and look like random integers

# ROUTERS AND LINKS



The Internet is composed of high-speed network links between routers and switches.

- A **router** takes an incoming packet and looks up the routing rule for sending it to the specified destination.
- Then it forwards the message out on the corresponding link.
- **Switches** are seen in clusters or racks of computers. Unlike a router, which can adapt the route over time, a switch uses fixed routing.

Packets have some maximum size, like 8KB. A “message” from process to process would often be far larger and will be sent as a series of packets.

# MAXIMUM PACKET SIZE VARIES

In the wide-area Internet, 1400 bytes for historical reasons. By now this feels too small, but it isn't easy to change...

In a local area network, 8KB is more typical.

In a datacenter you can switch to “fat packets” like 64KB or sometimes, even larger.

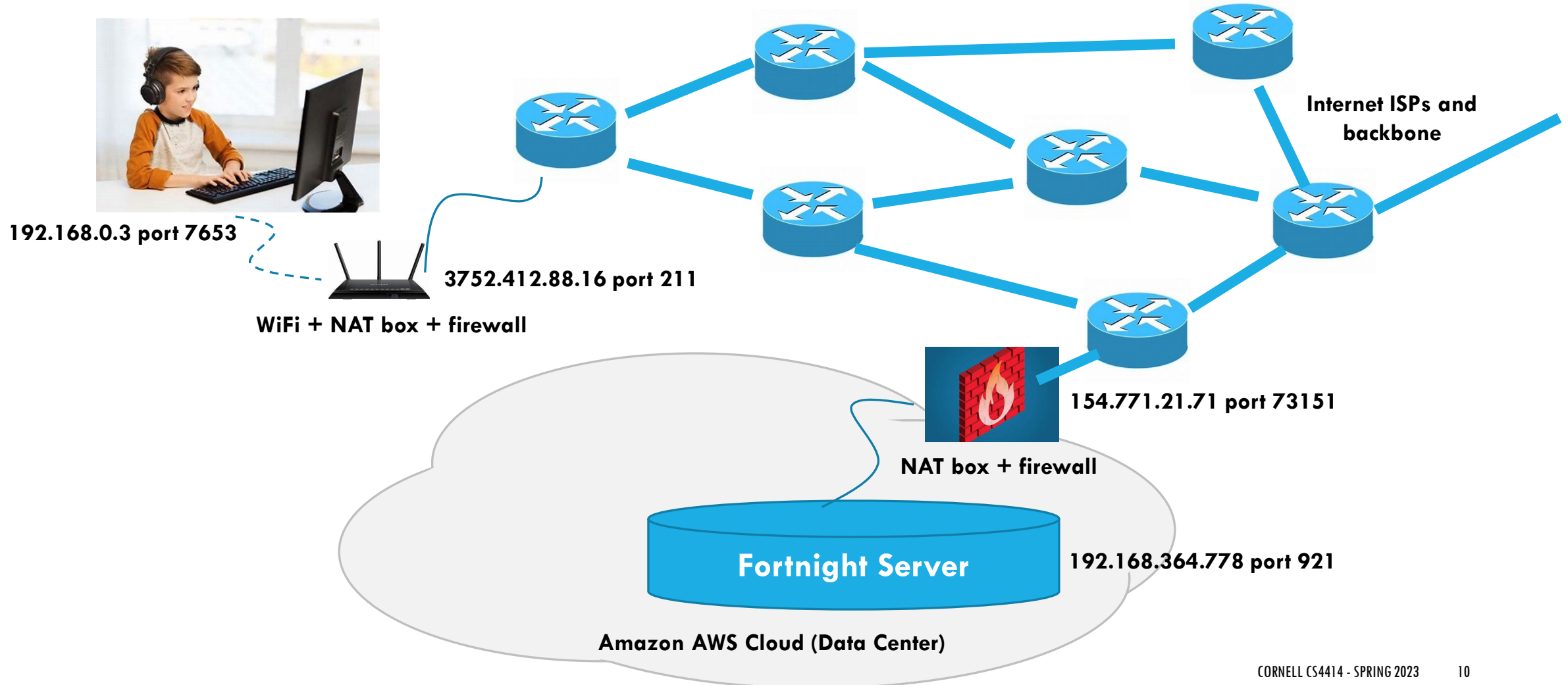


# NETWORK ADDRESS TRANSLATION

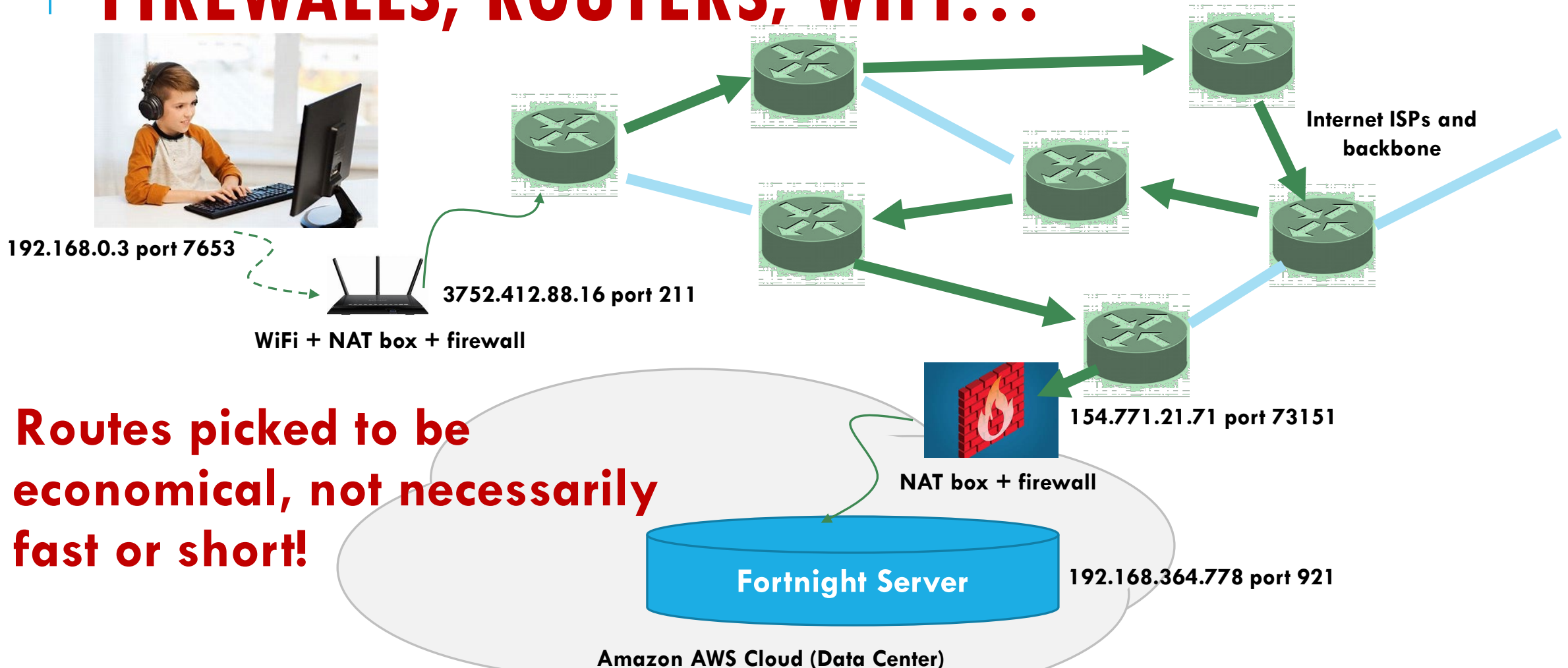
The “space” of addresses is much too small to cover the entire globe, so it evolved into a world of overlapping regions that use their own addresses. Like when a wifi router uses 192.168.0.xxx

Network address translators dynamically modify the (ip-address, port-no) information in packet headers to implement this (they might also block some packets: “firewall”)

# NETWORK ADDRESS TRANSLATION, FIREWALLS, ROUTERS, WIFI...

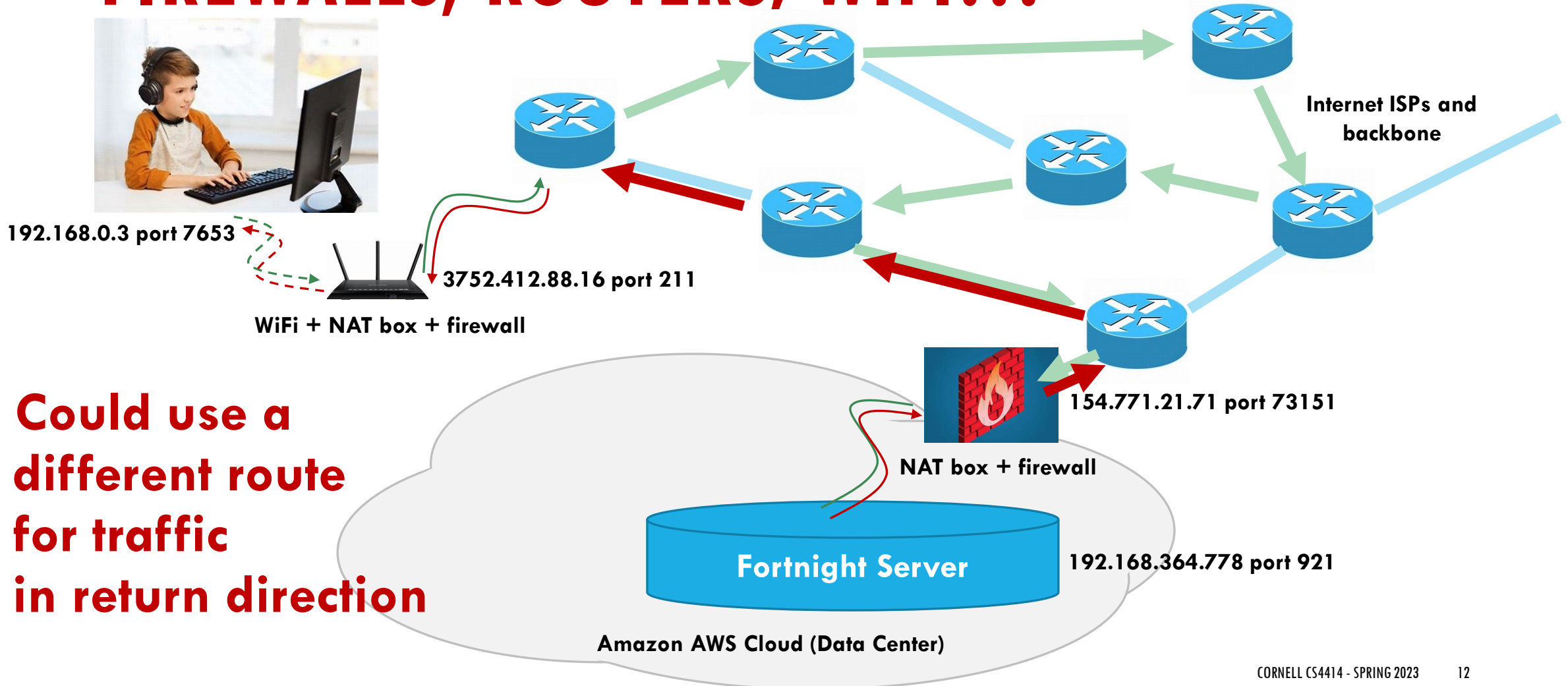


# NETWORK ADDRESS TRANSLATION, FIREWALLS, ROUTERS, WIFI...



**Routes picked to be economical, not necessarily fast or short!**

# NETWORK ADDRESS TRANSLATION, FIREWALLS, ROUTERS, WIFI...



**Could use a different route for traffic in return direction**

# TUNNELS



Trainload of IP packets enters a tunnel....

Sometimes it is convenient to send data through some domain without that domain “seeing” the packet headers.

A tunnel is used in such cases. A connection is made, but then packets are sent through it as “pure data”.

On the far side, they exit the tunnel and get routed “normally”

# A CRAZY PATCHWORK! BUT IT WORKS...

Internet routers are *fast* and speed of light is quite fast too.

... Should you care that your data went via Delaware?

... For the ISP, this route could be cheaper to operate, or use faster links and routers. The physically shortest path may be slower!

At each stage, a packet is routed to whatever router is next in the route for the particular IP address is happens to carry at that stage.

# MISTAKES DO HAPPEN, BUT RARELY

Internet routers use “routing tables” that tell them where to send packets. They adapt to route around outages.

Sometimes, mistakes can happen, but this is rare.

Goal of routing: Pick the most cost-effective, reliable, highest performing path available.

# MISTAKES DO HAPPEN, BUT RARELY

The Switch

## How the Chinese Internet ended up in Cheyenne, Wyoming



The building on Pioneer Ave. that houses Sophidea, the company that received a deluge of Chinese Internet traffic Tuesday. (Google Streetview)

An ISP operator (somewhere very remote from China) accidentally miscoded one digit of a “fixed” route.

This disrupted global routing.

Traffic intended for a centralized router in Shenzhen was “redirected” to a small cafe in Wyoming, near Yellowstone Park...

... *all the traffic, for much of China*



# WHAT DID THE CAFE DO WITH CHINA'S INTERNET TRAFFIC?

They were “dropped” (discarded, silently)

In fact, this is a feature of the Internet!

The Internet works like a network of highways and bridges, but a bridge can just toss cars off if a traffic jam ever forms!



# WHY DROP PACKETS? END-TO-END PRINCIPLE

In the early days of the Internet a debate arose: should the Internet be reliable, or is it ok to drop packets?

The “end to end” principle was this: The Internet itself doesn’t need to be reliable, as long as it is blindingly fast and *mostly* reliable. This ultimately “won” over other options.

In fact, failures are rare. But overload is common.

# TCP STREAMS EMBODY THE END-TO-END RULE

Like a pipe, TCP sends a stream of bytes from A to B, with no losses or corruption or out-of-order data.

TCP has two “end points”. The “end to end” principle makes these endpoints responsible for reliability and security.

TCP uses a *protocol* to achieve them: A scripted exchange of messages with a format that TCP imposes.

# DATA LIVES INSIDE THESE TCP PACKETS

TCP has a header of its own, with information used by the end-to-end protocol operated by the TCP module in the Linux kernel.

The message you send is chopped into segments and carried as data within these TCP packets.

On arrival, TCP will deliver the data, but you won't see the TCP headers: it removes them in the kernel protocol stack.

# TCP STARTS WITH A SPECIAL THREE-WAY HANDSHAKE



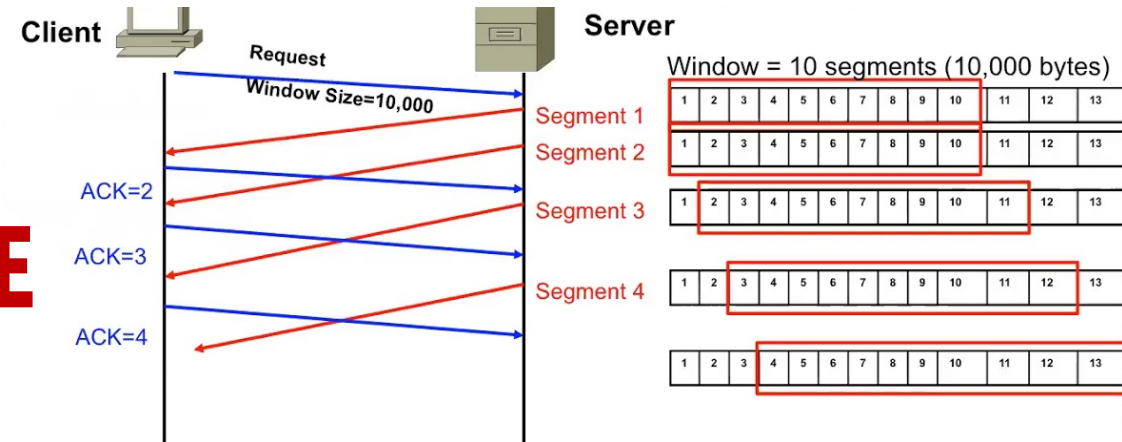
TCP has some very basic security built in: the initial connection involves a “three way handshake”

1. Hello B. I am A, trying to connect to you. [SEQ=1 234]
2. A, I would love to connect. [SEQ=9876]
3. B, this is A again. Success! [SEQ=1 234,9876]

SSL is a standard for creating stronger security on TCP connections. It additionally establishes a session “key” used to encrypt all data.

# TCP SLIDING WINDOW: USED AFTER HANDSHAKE

Mile high summary:

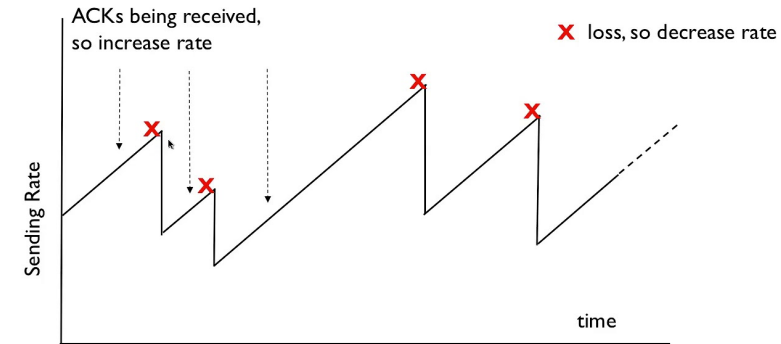


The TCP sliding window. Animation available [here](#)

When TCP sends data, the sender numbers the packets. The receiver acknowledges receipt (“ack”) or if it notices a gap, asks for a retransmission (“nack”).

This is done using a form of bounded buffer: the sliding window.

# TCP ALSO DOES RATE CONTROL



TCP has a scheme for varying the transmission rate to match the speed of the connection.

TCP steadily speeds up until packet drops occur: some router got overloaded. Then it slows down drastically... then speeds up.

Called an *additive increase, multiplicative decrease* scheme.



NAT box + firewall

# LIMITATIONS

All systems have firewalls of various kinds, and some also have additional features (such as “network address translation”, and “VPN tunneling”).

Those are programmed to block unwanted Internet traffic.

For example, my wifi router gives computers in my home IP addresses like 192.168.0.11. These are not visible outside my home.



# ASYMMETRIC CONNECTIVITY!

A process on my computer, in my home, can connect *to* Netflix.com or Amazon.com or Azure.com.

Once the session is established, messages can flow both ways: the wifi router opens a “tunnel” that allows them through.

Yet that same Azure.com server wouldn't be able to initiate a connection to my computer: traffic would be blocked!

# HOW DO WE MAKE THESE CONNECTIONS?

Consider a client on my machine, perhaps a web browser.

The server is on some other machine, perhaps at Netflix.com

- In fact, Netflix.com is **hosted** by Amazon (rents machines on their cloud)
- So if you talk to a Netflix.com server, your TCP connection will actually be to a machine in an Amazon AWS data center.
- This is an aspect of modern cloud computing: “rent don’t own”.

# SOCKET OPERATIONS: SERVER

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);  
// logic to initialize serv_addr struct not shown!  
bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));  
listen(sockfd, 5);  
int connfd = accept(sockfd);  
int bytesread = receive(connfd, buffer, nbytes);  
send(connfd, buffer, nbytes);
```

# SOCKET OPERA

Create a connection endpoint. This will be used for a TCP connection, but more initialization is required.

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);  
// logic to initialize serv_addr struct not shown!  
bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));  
listen(sockfd, 5);  
int connfd = accept(sockfd);  
int bytesread = receive(connfd, buffer, nbytes);  
send(connfd, buffer, nbytes);
```

# SOCKET OPERATIONS: SERVER

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);  
// logic to initialize serv_addr struct not shown!  
bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));  
listen(sockfd, 5);  
int connfd = accept(sockfd, NULL, NULL);  
int bytesread = receive(connfd, buffer, nbytes);  
send(connfd, buffer, nbytes);
```

Associate an IP address and port number with it. These come from "gethostbyname"

# SOCKET OPERATIONS: SERVER

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);  
// logic  
bind(sockfd, &serv_addr, sizeof(serv_addr));  
listen(sockfd, 5);  
connfd = accept(sockfd);  
int bytesread = receive(connfd, buffer, nbytes);  
send(connfd, buffer, nbytes);
```


Tell Linux this server will accept up to five simultaneous client connections via the TCP protocol.

# SOCKET OPERATIONS: SERVER

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);  
// logic to initialize serv_addr struct not shown!  
bind(sockfd, (struct sockaddr_in *) &serv_addr, sizeof(serv_addr));  
listen(sockfd, 5);  
int connfd = accept(sockfd, (struct sockaddr_in *) &client_addr, &client_addr_len);  
int bytesread = receive(connfd, buffer, nbytes);  
send(connfd, buffer, nbytes);
```


Accept returns the file descriptor of an established connection, and now the server can read bytes from it

# SOCKET OPERATIONS: SERVER

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);  
// logic to initialize serv_addr struct not shown!  
bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));  
listen(sockfd, 5);  
int connfd = accept(sockfd, );  
int bytesread = receive(connfd, buffer, nbytes);  
send(connfd, buffer, nbytes);
```



# SOCKET OPERATIONS: SERVER

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);  
// logic to initialize serv_addr struct not shown!  
bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));  
listen(sockfd, 5);  
int connfd = accept(sockfd);  
int byt  Write some bytes  
send(connfd, buffer, nbytes);
```

# SOCKET OPERATIONS: CLIENT

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

... client does not need “bind” or “listen” or “accept”!

```
int connfd = connect(sockfd, (struct sockaddr *) &serv_addr,  
                    sizeof(serv_addr));
```

```
send(connfd, buffer, nbytes);
```

```
int bytesread = receive(connfd, buffer, nbytes);
```

# HOW DID THE CLIENT GET THE SERVER'S IP ADDRESS?

It uses **gethostbyname**, then has to fill in the address struct.

It gets its own IP address by calling **gethostbyname** on **localhost**.

- The server's port number is from a table of standard port numbers.
- Linux can also assign a port number. Your server can use this, then check the address and port number it received, then publish it in a file or on a web page for the client to find.

# OK... A IS NOW CONNECTED TO B!

A TCP stream is just a stream of bytes.

To make a request with arguments:

- A builds a header identifying the requested operation and serializes the arguments into a byte array.
- B reads the header first, allowing it to learn how much data to read.
- The request-id will be used later to pair the result with a waiting thread.

# GOOGLE GRPC: A POPULAR PACKAGING OF THESE MECHANISMS

A library that can be used from C++, Java, Python, etc. Runs over TCP.

Fairly easy to use, but we won't get into the details in CS4414.

GRPC allows a server to associate an object with a connection. The client can call methods in a type-checked way.

# GRPC OVERHEADS?

GRPC has a fast serialization method, much better than CORBA (from Lecture 18). When using a secure TCP session (with SSL), there is a small extra encryption/decryption delay.

## **Network delays are the main cost:**

- From Ken's lake cottage in Trumansburg to Cornell: 1.5ms (routed via Syracuse).
- Between two machines in a data center, they are as low as 50-100us.
- On the public Internet, bandwidth of 10MB is excellent.
- But inside a data center, rates can reach 10 GB: 1000x faster!.

# GRPC EXAMPLE: HELLO WORLD

```
// The greeting service definition is used by both the
// Client and the server. It contains virtual methods
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

```
class GreeterServiceImpl final : public Greeter::Service {
  Status SayHello(ServerContext* context, const HelloRequest*
                  request, HelloReply* reply) override {
    std::string prefix("Hello to my favorite client!");
    reply->set_message(prefix + request->name());
    return Status::OK;
  }
};
```

# NOTE ABOUT THIS EXAMPLE

What I showed you is incomplete!

GRPC requires a specific installation (it isn't hard).

Compiling a client or server also requires various include files, and a special “initialization” must be called from main.



# WOULD A WEB BROWSER USE GRPC?

The web has its own encoding for messages.

This HTML encoding is less efficient than the one GRPC uses. It works well and is universal but is slow to compute and “bulky”.

- Every message is encoded as a web page, and every reply
- Data is printed in ascii text format.
- Uses layers of standards: SOAP on HTML on XML... SSL would add a layer of encryption to this.

... **SO**

You launch your web browser and type in Netflix.com

The first step is to look up the IP address for Netflix.com. This is done by calling `gethostbyname`, which uses the DNS service.

You get an IP address for Netflix.com (in fact, the IP address of a nearby AWS data center: Netflix is hosted by Amazon).

# NEXT STEPS

Your browser initiates an http connection (insecure: https uses SSL, but http isn't encrypted in any way).

AWS selects some machine within the Amazon cloud with the Netflix web server process already running on it.

A three-way handshake is performed.

# NEXT STEPS



**Your Netflix Cookie!**

Your web browser first sends Netflix a “cookie” with your user information. It uses the old web services model, so the cookie is actually encoded as a small web page.

Next Netflix sends back an initial welcome web page. The browser reads this back as a byte stream, deserializes it, and renders the page on your screen.

# YOU SELECT A MOVIE...

Netflix recommends Wednesday.

You click the tile... it is associated with a URL link to the movie.

The browser sends an HTTP request to Netflix to “open” the URL.  
Netflix streams the movie back.



**“For the record, I don’t believe that I’m better than everyone else. Just that I’m better than you.”**

# WOULD THE BROWSER BE FASTER ON GRPC?

HTML is not a suitable encoding for a photo or video!

But it has built-in special features for sending photos and videos. The browser makes an extra TCP connection, and the server streams the data in the format the client's browser requests.

The server even resizes the images for the client's browser size.

# OTHER OPTIONS

Client and server sound like A and B are very different kinds of programs, but in fact any GRPC process can play both roles.

So we can build applications in which there are multiple processes that cooperate in various ways, using message passing to share information.

As an example, A and B could “replicate” some kind of data captured from the real world by A: A would relay it to B.

# WHY REPLICATE?

**Fault tolerance:** in a cloud-scale system, we may see crashes.

**Performance:** A and B could share the work for some task.  
Netflix serves popular movies from many servers.

**Coordination:** A might have some role, such as “be an air traffic controller for flight Delta 121” and B could track the actions in order to be smarter about ATC for other flights.



# OTHER COMMUNICATION OPTIONS

Some systems use a “message bus” or “message queue”. We saw one in the ATC example in Lecture 18.

These use a publish-subscribe model.

- We define a set of topics, and programs can subscribe to them.
- If a publisher publishes to some topic, upcalls occur in subscribers.

# THESE WORK EVEN IN A SINGLE MACHINE!

You don't need two computers to create a GRPC or message bus application.

The tools work in an identical way on one computer (if the firewall is configured to allow it)

This is popular because it makes it so much easier to develop applications, and to port them from place to place.

# NETWORK SECURITY

We touched upon TCP security via SSL: a special version of a handshake that assigns a cryptographic key to each session.

HTTPS uses SSL automatically.

But what other forms of security are available?

# VPN AND VPC

Many of us have access to special computers at Cornell.

These aren't available for public use, so they live inside `cs.cornell.edu` and aren't even visible from outside Cornell.

With a “virtual private network” you can step inside the Cornell firewall from home, as if you were in Gates Hall!

# HOW A VPN WORKS.

Cornell operates a special VPN router. You need yournetid to log into it. They use a product called Cisco AnyConnect.

Once you log in, a TTP SSL connection will exist from your machine to the VPN router.

IP addresses in the Cornell domain, or DNS requests, are automatically sent over this encrypted link.

# VIRTUALLY PRIVATE *CLOUD*

VPC is similar in concept, for people who use cloud computing.

Suppose your organization uses 10 machines on Azure or AWS.

Those 10 can be isolated from other cloud users: a VPC creates the illusion of a network composed purely of your machines, plus services the cloud operator provides (like the global file system)

# WITH VPC, ALL TRAFFIC IS ENCRYPTED

This way even if some intruder is watching the network, and even if your own applications use GRPC without SSL enabled, your messages are still encrypted.

It happens automatically. AWS and Azure both have special hardware accelerators to do the encryption and decryption.

In fact, with VPC they even encrypt files, using the same technique!

# SUMMARY OF LECTURE 20

When applications will run on multiple machines, we use networking to link the processes.

This occurs over TCP, often with SSL encryption. Encryption is not the default but is very common.

Google GRPC is one common library for building networked applications with a C++ object-oriented style of code.