Lecture 22

Today
- APIs + RPCs
- NAT

Announcements
- TCP due on Tuesday
- HW5 out after today
- Guest lecture on Tuesday!

Lecture 22: APIs + RPCs; NAT

Warmup: How do you define a protocol? Describe it to someone who hasn't taken 1680.

ARP TCP UDP SNOWCAST

WIFI IP RIP HTTP

SSH TLS WS QUIC

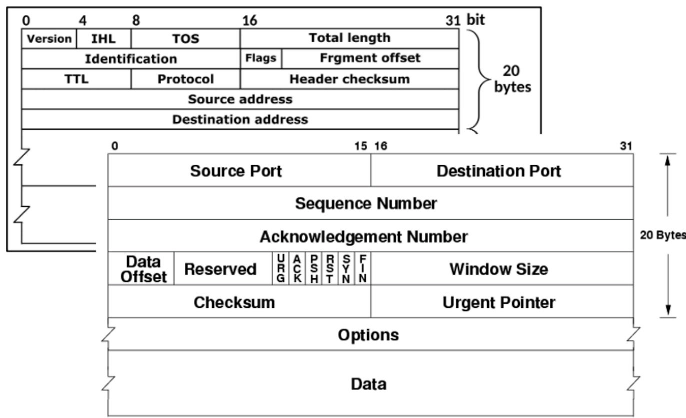
TELNET DNS DHCP BGP

... ICMP IS-IS OSPF

Set of rules for communicating with other systems
= > What kinds of messages to communicate
=> What format (packet format, etc.)
=> Semantics for messages (what to send, what to respond with)
=> How to deal with errors, timeouts, retries, etc.

There are some common themes in all of these...

How do we define a protocol?



Data representation (headers, packet formats)

```
From: draft-ietf-tcpm-rfc793bis-28                      Internet Standard

Internet Engineering Task Force (IETF)                      W. Eddy, Ed.
STD: 7                                                       MTI Systems
Request for Comments: 9293                                   August 2022
Obsoletes: 793, 879, 2873, 6093, 6429, 6528,
           6691
Updates: 1011, 1122, 5961
Category: Standards Track
ISSN: 2070-1721

                 Transmission Control Protocol (TCP)

Abstract

   This document specifies the Transmission Control Protocol (TCP).  TCP
   is an important transport-layer protocol in the Internet protocol
   stack, and it has continuously evolved over decades of use and growth
   of the Internet.  Over this time, a number of changes have been made
```

Semantics (when to send each message, how to handle errors)

You made a custom protocol...

Client to Server Commands

The client sends the server messages called **commands**. There are two commands the client can send the server, in the following format:

```
Hello:
    uint8 commandType = 0;
    uint16 udpPort;

SetStation:
    uint8 commandType = 1;
    uint16 stationNumber;
```

TYPE PORT

A `uint8` is an unsigned 8-bit integer; a `uint16` is an unsigned 16 programs `MUST` use **network byte order**. So, to send a `Hello` co send exactly three bytes to the server: one for the command type

```
// Guessing game example (lecture 3!!)
type struct GuessMessage {
    MessageType uint8
    Number uint16
}

func (m *GuessMessage) Marshal() []byte {
        buf := new(bytes.Buffer)
        err := binary.Write(buf, binary.BigEndian, m.MessageType)
        if err != nil {
                . . .
        }

        err = binary.Write(buf, binary.BigEndian, m.Number)
        if err != nil {
                . . .
        }
        return buf.Bytes()
}
```

All the protocols we've seen so far (Snowcast, IP, TCP, RIP, ...): manually packing bytes into buffers

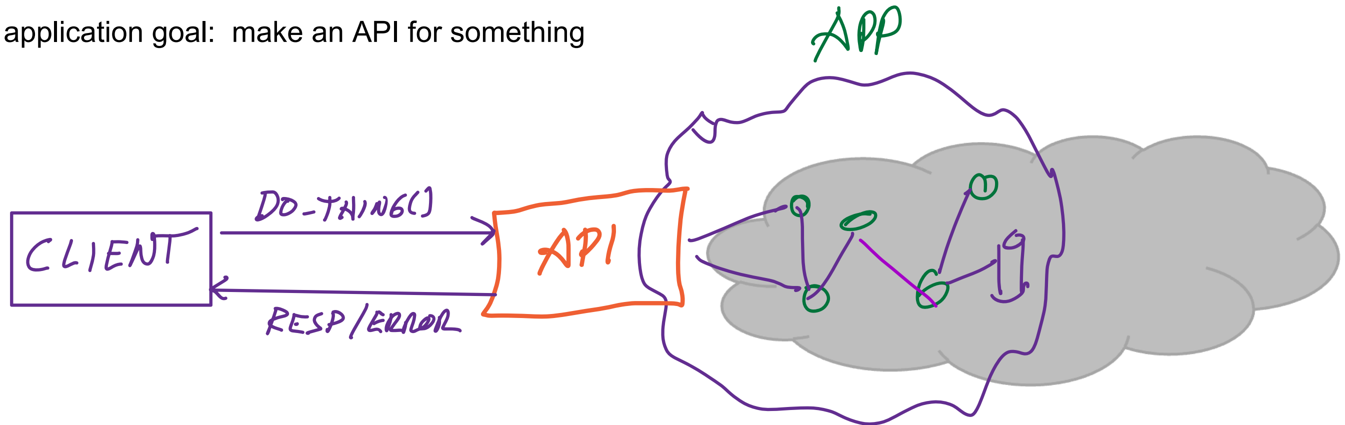
This is useful for learning
- How protocols work under the hood,
- How fundamental internet protocols actually work

But if your job is to build applications, is that what you should be doing?
=> Almost certainly not.

There must be a better way!

How SHOULD you write a protocol, outside of this class?

Typical application goal: make an API for something



Idea: build on tools define an API for components to interact with app
=> Create "endpoints" for where you write code to perform some action
=> Don't want to worry about serialization/error handling
=> Build on existing protocols to handle scaling (proxies, load balancing, etc.)

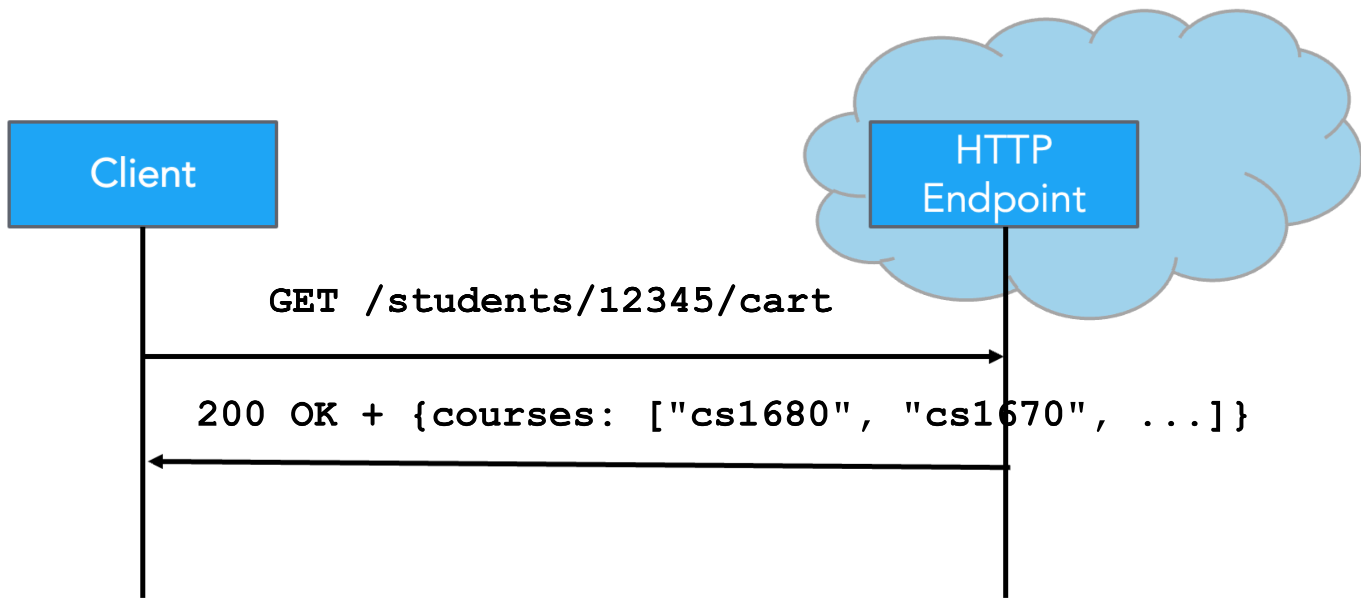
To look at it another way (in case this is more clear)...

What we have: some servers/services that live somewhere in the cloud
=> Might be distributed, might not

Want: end-user to be able to use your app
- Read some concrete object (user, product list, etc.)
- Write/upload/make changes to those objects

=> Maybe we can abstract away the protocol-level stuff in some way? Maybe with tools we already know about?

How to do this with HTTP?



Implementing an API with HTTP

- Endpoints are URLs
- Leverage existing semantics of HTTP
  - Methods: GET (request data), POST (upload data)
  - Status codes: if succeeded/failed
  - Headers/URL parameters: arguments/input data
  - Cookies/headers: arguments/authentication info

- Response comes back in some kind of common format (JSON, XML, ...)
  => "self-describing formats"

=> Lots of frameworks to help build this!

Example: Gradescope

COURSE ROSTER

```
curl -X GET 'https://www.gradescope.com/courses/567871/memberships.csv’
   -H 'User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:109.0)
Gecko/20100101 Firefox/118.0’
   -H 'Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8'
-H 'Accept-Language: en-US,en;q=0.5’
   -H 'Accept-Encoding: gzip, deflate, br’
   -H 'Referer: https://www.gradescope.com/courses/567871/memberships’
   -H 'DNT: 1’
   -H 'Connection: keep-alive’
   -H 'Cookie: remember_me=XXXXXXXXXXXXXXXX; __stripe_mid=XXXXXXXXXXXXX;
signed_token=XXXXXXXXXXXXXXX; _gradescope_session=XXXXXX[. . .]XXXXXXXX;
__stripe_sid=XXXXXXXXXXX’
   -H 'Upgrade-Insecure-Requests: 1’
   -H 'Sec-Fetch-Dest: document’
   -H 'Sec-Fetch-Mode: navigate’
   -H 'Sec-Fetch-Site: same-origin’
   -H 'Sec-Fetch-User: ?1'
```

COOKIE FOR AUTH

Example: Github's REST API

GitHub Docs | Version: Free, Pro, & Team | Search GitHub Docs | Sign up

REST API / Repositories / Repositories

Path parameters

`org` string Required

The organization name. The name is not case sensitive.

"ARGUMENTS"

Query parameters

`type` string

Specifies the types of repositories you want returned.

Default: `all`

Can be one of: `all`, `public`, `private`, `forks`, `sources`, `member`

`sort` string

The property to sort the results by.

Default: `created`

Can be one of: `created`, `updated`, `pushed`, `full_name`

`direction` string

The order to sort by. Default: `asc` when using `full_name`, otherwise `desc`.

Can be one of: `asc`, `desc`

`per_page` integer

The number of results per page (max 100). For more information, see "Using

Code samples for "List organization repositories"

Request example

URL

GET `/orgs/{org}/repos`

cURL | JavaScript | GitHub CLI

```
curl -L \
  -H "Accept: application/vnd.github+json" \
  -H "Authorization: Bearer <YOUR-TOKEN>" \
  -H "X-GitHub-Api-Version: 2022-11-28" \
  https://api.github.com/orgs/ORG/repos
```

Response

Example response | Response schema

WHAT RESPONSE LOOKS LIKE

Status: 200


```
[
  {
    "id": 1296269,
    "node_id": "MDEwOlJlcG9zaXRvcnkxMjk2MjY5",
    "name": "Hello-World",
    "full_name": "octocat/Hello-World",
    "owner": {
      "login": "octocat",
      "id": 1,
```

Request

```
curl -X GET -H "Accept: application/vnd.github+json" \
            -H "Authorization: Bearer <API-TOKEN>" \
            -H "X-GitHub-Api-Version: 2022-11-28" \
            https://api.github.com/orgs/brown-cs1680-s26/repos
```

Request headers: includes API token to identify client

Response

```
< HTTP/2 200
< date: Thu, 16 Apr 2026 12:56:29 GMT
< content-type: application/json; charset=utf-8
< content-length: 7538
< cache-control: private, max-age=60, s-maxage=60
< vary: Accept, Authorization, Cookie, X-GitHub-OTP,Accept-Encoding, Accept,
X-Requested-With
< etag: "4f4429cabb2d17aa65ed4f16713a5eb9ea1f3f67961ecaedfbd8cbba8ae9c2c5"
[ . . .]
< server: github.com
< x-github-request-id: 9DB6:374132:50D521D:148A738A:69E0DC7C
[
  {
    "id": 1136395562,
    "node_id": "R_kgDOQ7wFKg",
    "name": "some-repo-name",
    "full_name": "brown-cs1680-s26/some-other-repo-name",
    "private": true,
    "owner": {
      "login": "brown-cs1680-s26",
      "id": 255217704,
      "node_id": "O_kgDODzZQKA",
      "avatar_url": "https://avatars.githubusercontent.com/u/255217704?v=4",
      "gravatar_id": "",
      "url": "https://api.github.com/users/brown-cs1680-s26",
      "html_url": "https://github.com/brown-cs1680-s26",
      "followers_url": "https://api.github.com/users/brown-cs1680-s26/followers",
      "following_url":
"https://api.github.com/users/brown-cs1680-s26/following{/other_user}",
      "gists_url": "https://api.github.com/users/brown-cs1680-s26/gists{/gist_id}",
      "starred_url":
"https://api.github.com/users/brown-cs1680-s26/starred{/owner}{/repo}",
      "subscriptions_url":
"https://api.github.com/users/brown-cs1680-s26/subscriptions",
      "organizations_url": "https://api.github.com/users/brown-cs1680-s26/orgs",
      "repos_url": "https://api.github.com/users/brown-cs1680-s26/repos",
      "events_url": "https://api.github.com/users/brown-cs1680-s26/events{/privacy}",
      "received_events_url":
"https://api.github.com/users/brown-cs1680-s26/received_events",
      "type": "Organization",
      "user_view_type": "public",
      "site_admin": false
    },
```

HEADERS

Can leverage HTTP features like caching!

Response includes other API endpoints

APIs for the web: REST
REpresentational State Transfer
"Architectural style" for building things on the web

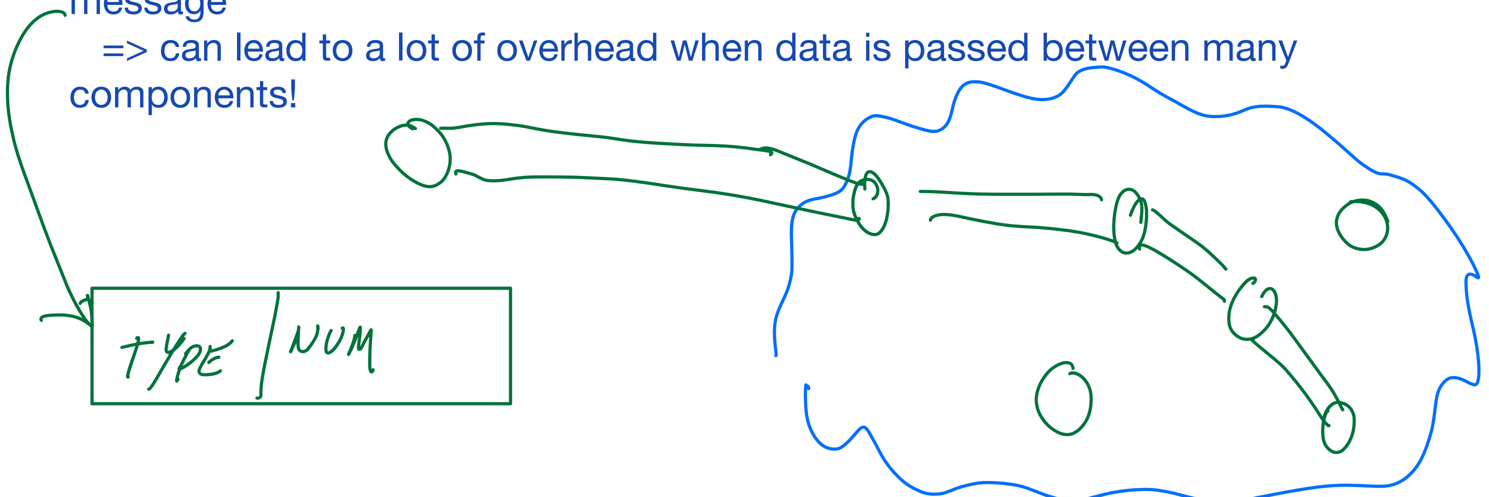
Key properties
- Use URLs to specify resources you want: interact via URLs, responses contain other URLs
- Leverages properties of HTTP
    - Responses indicate how they should be cached
    - Stateless: server doesn't need to remember state about client between requests

Why is this useful?
- HTTP is ubiquitous!
    => Really easy to interact with API
        ... from a browser
        ... from basically any language
        ... even from shell scripts

- Lots of existing tools to scale (caching, proxies, load balancers, etc)

Why use JSON/etc vs. a binary encoding (manually packing byte arrays)?
- Easier to interpret in most languages
- Easy to extend

However, serializing JSON is expensive: compare to parsing a Snowcast message
=> can lead to a lot of overhead when data is passed between many components!

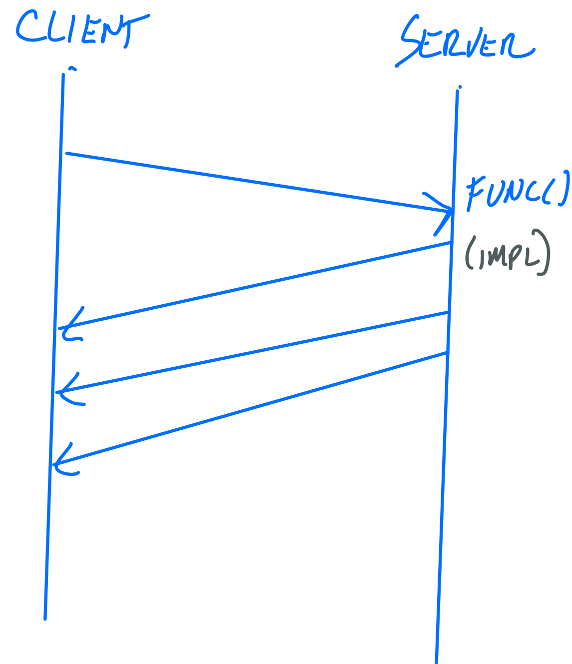


If every node needs to serialize and deserialize, this adds up!

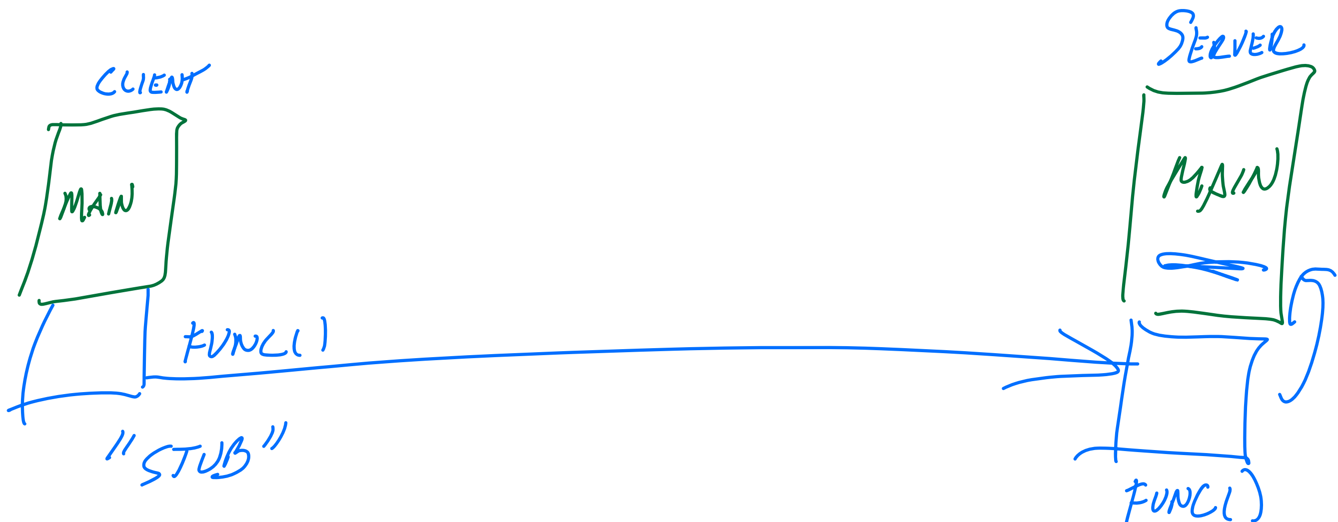
Remote Procedure Call (RPC)
Make a function call happen over the network

Provides
- A way to define messages
- Semantics for how the operate (sync/async, one-to-many)
- A way to define the message format

=> Together, these let you define a protocol in some way



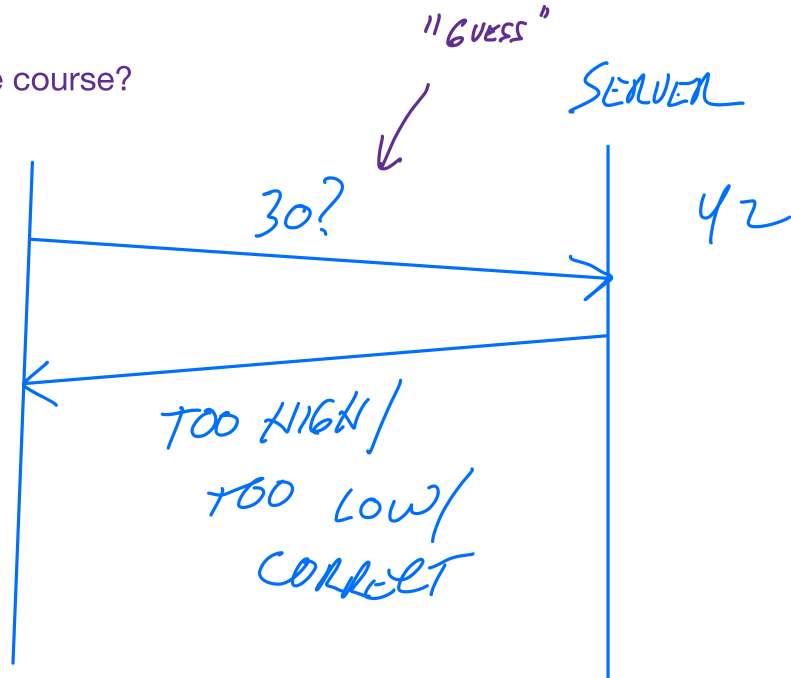
How to use
1. Write some kind of definition for the protocol
2. Framework generates a "stub" with functions you can call
   => Think: library, header files, etc.
   => Kind of like a stencil: it handles the protocol stuff, you write the logic



Some examples: gRPC (made by Google, open-source), Apache Thrift, JSON RPC, NFS (network file system)

Example: gRPC

Remember our "guessing game" protocol from the beginning of the course? Let's define it in gRPC



In RPC, we often define the protocol in an Interface Description Language (IDL), which is a custom language for this purpose. gRPC's IDL is called protobuf (short for "protocol buffers") another tool for automating serialization. Both projects were initially developed by Google, but are now open-source and widely used in and outside of Google.

```
service GuessingGame {
    rpc MakeGuess(Guess) returns (GuessResult) {}
}

enum GuessStatus {
    GUESS_NONE = 0;
    GUESS_TOO_LOW = 1;
    GUESS_TOO_HIGH = 2;
    GUESS_CORRECT = 3;
}

message Guess {
    int32 number = 1;
}

message GuessResult {
    GuessStatus result = 1;
}
```

Specify RPCs this protocols supports
=> These are the endpoints, or "functions" that can be called on the server

Define what messages look like
(Protobuf has a lot of features for this: can support optional fields, list types, extensibility, ... see documentation for more!)

ORDER IN MESSAGE

What happens now?
gRPC has "backends" for many languages. Can use IDL to generate code that implements the protocol

=> Think: header file, struct definitions, function stubs

=> Leverages gRPC library to send messages (built on HTTP/2, includes TLS for security, has logic for timeouts, etc.)

See the demo (and linked code example) for a running example, and to see what the code looks like. Note how our implementation is much simpler than the original guessing game from lecture 2!!

Takeaways: why should you care?

=> Unless you really want to optimize your protocol, use an IDL

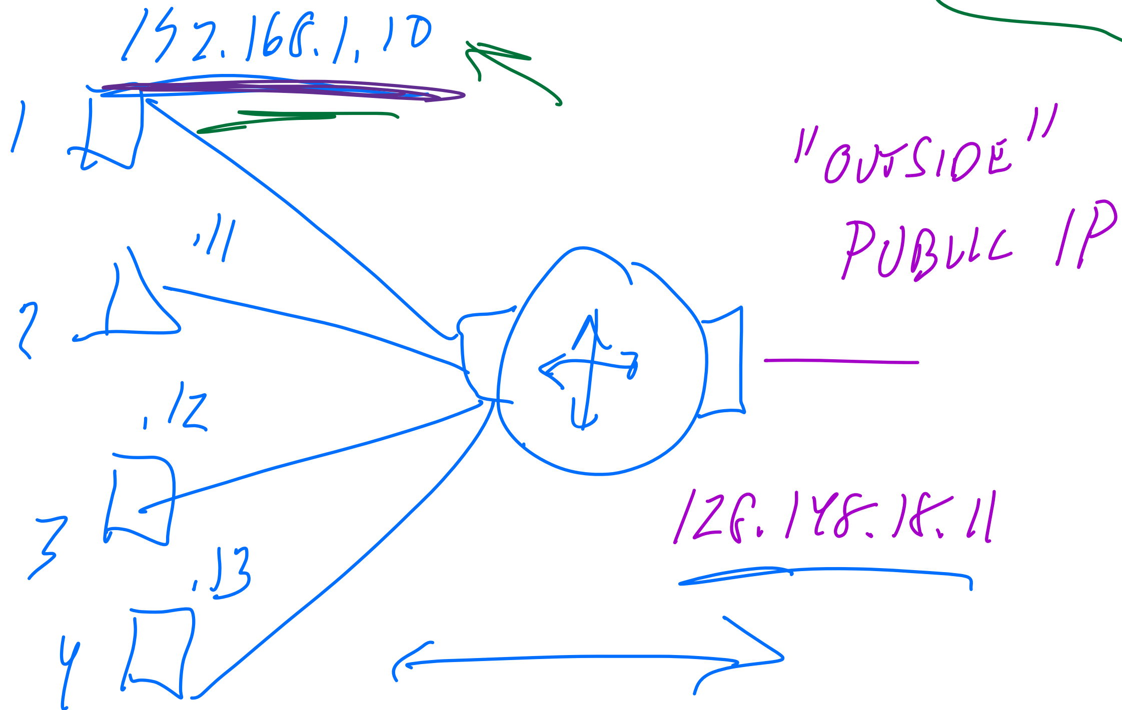
=> Parsing code is easy to get (slightly) wrong, hard to make fast

=> Only want to do this once, can leverage tools to help!

(See extra notes at the end for more on comparisons between REST APIs and gRPC, which we didn't discuss this year)

Network Address Translation (NAT)

Problem: you get just one IP from your ISP
=> Need to share IP among many devices in the same network!
=> (Also: multiple IPs on same system for internal use (e.g., Docker)



"INSIDE NETWORK"

192.168.1.0/24

When on a network with NAT:
- You get a private IP
• Router does some stuff to share one public IP with other devices

=> When traffic leaves network, IP always appears as public IP

Private IP ranges (RFC 1918)

- 127.0.0.0/8: "Loopback address" -- always for current host
- 10.0.0.0/8
- 192.168.0.0/16
- 172.16.0.0/12 ← DOCKER

Network Address Translation (NAT)

Problem: you get just one IP from your ISP
=> Need to share IP among many devices in the same network!
=> (Also: multiple IPs on same system for internal use (e.g., Docker)

When on a network with NAT: - You get a private IP
• Router does some stuff to share one public IP with other devices
=> When traffic leaves network, IP always appears as public IP

Private IP ranges (RFC 1918)

- 127.0.0.0/8: "Loopback address" -- always for current host
- 10.0.0.0/8 ←
- 192.168.0.0/16
- 172.16.0.0/12 ← DOCKER

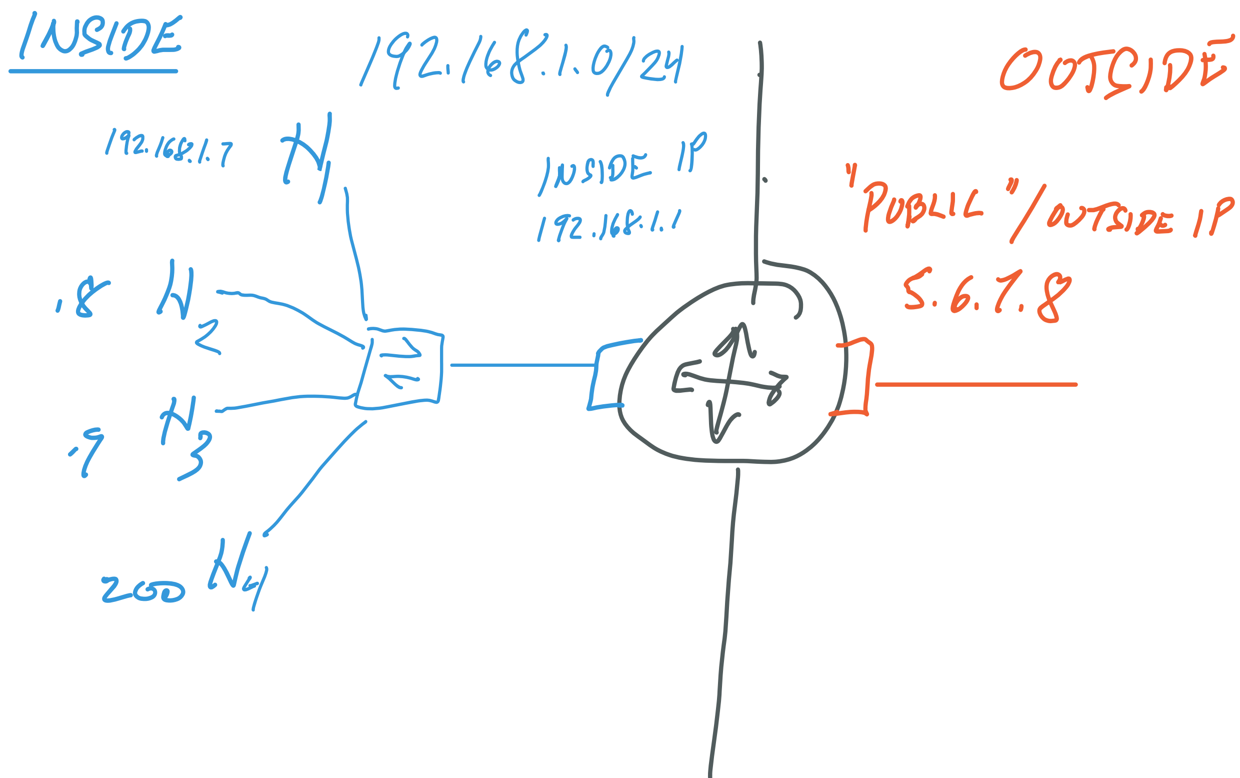
Network Address Translation (NAT): in class, we saw how if we hosted our webserver on our laptop, we couldn't connect to it from the wider Internet. Why does this happen?

We noticed there was something weird going on with IP addresses:
- Our IP on Nick's laptop: something like 10.3.128.54
- Our IP as seen by external website: 128.148.11.2 (IP registered to Brown, 128.148.x.x

What's going on, why are these different?????

The answer is due to this one weird trick.... NAT

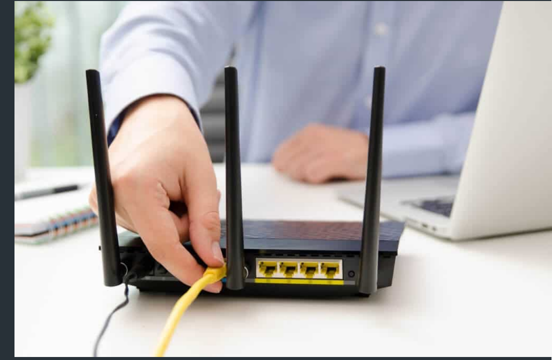
How NAT works (the gist, details next)
=> Routers have one or more "public" or "outside" IP addresses, which are "real" IPs that are routable on the public Internet
=> Devices on the "inside" network have IPs in a "private" address range
=> Router translates (modifies) packets from "outside" to "inside" address



Why?? NAT was developed in the 1990s when the Internet was first beginning to run out of IPv4 addresses (and before IPv6 had much adoption). It has persisted since then, and has become ubiquitous in how the Internet works for end users.

It's also kind of a hack...

Where it gets weird…



You get just one IP from your ISP…

=> Need to share IP among many devices on the same network!

Solution: Create a "private" IP range used within local network

=> Routers need to do extra work to share public IP among many private IPs

=> Network Address Translation (NAT)
(A form of connection multiplexing)

Private IPs (RFC1918)

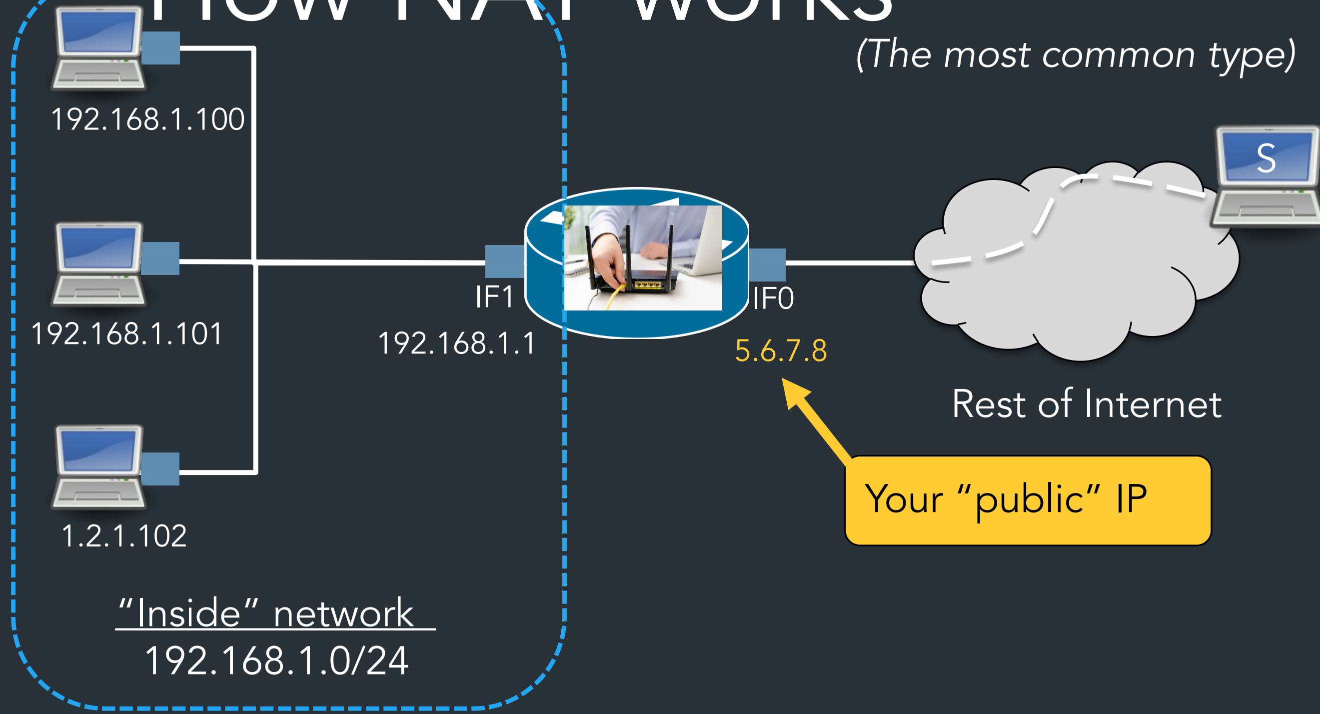
IP ranges reserved for "private" networks:

Prefix	Use
127.0.0.0/8	"Loopback" address—always for current host
10.0.0.0/8	Reserved for private internal networks (RFC1918)
192.168.0.0/16	Reserved for private internal networks (RFC1918)
172.16.0.0/12	Reserved for private internal networks (RFC1918)

(Handwritten annotations: EXAMPLES — 10.0.0.0/8 ← BIG NETWORKS (BROWN WIFI); 192.168.0.0/16 ← MOST HOME NETWORKS; 172.16.0.0/12 ← DOCKER)

- Many networks will use these blocks internally
- These IPs should never be routed over the Internet!
  - What would happen if they were?

How NAT works



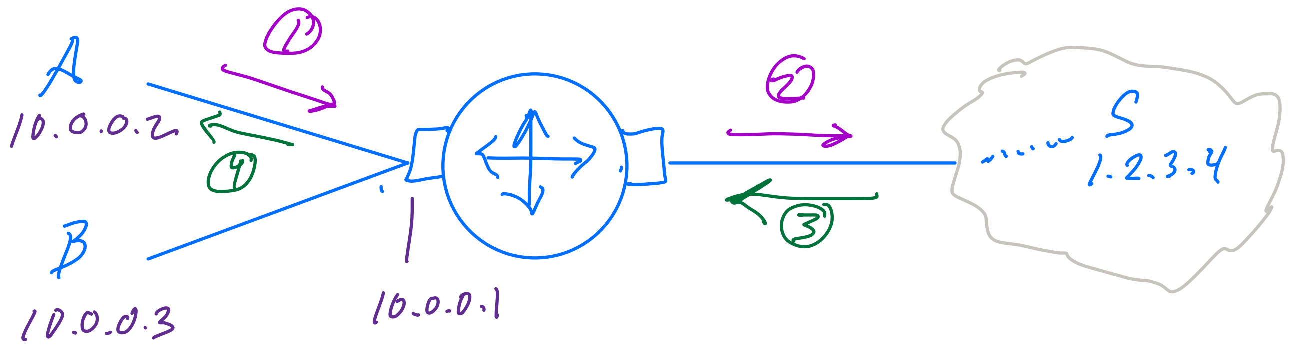
Goal: Share one IP among many hosts on a private network

Router *translates (modifies) packets from "inside" to use "outside" address*

=> Router needs to remember connection state

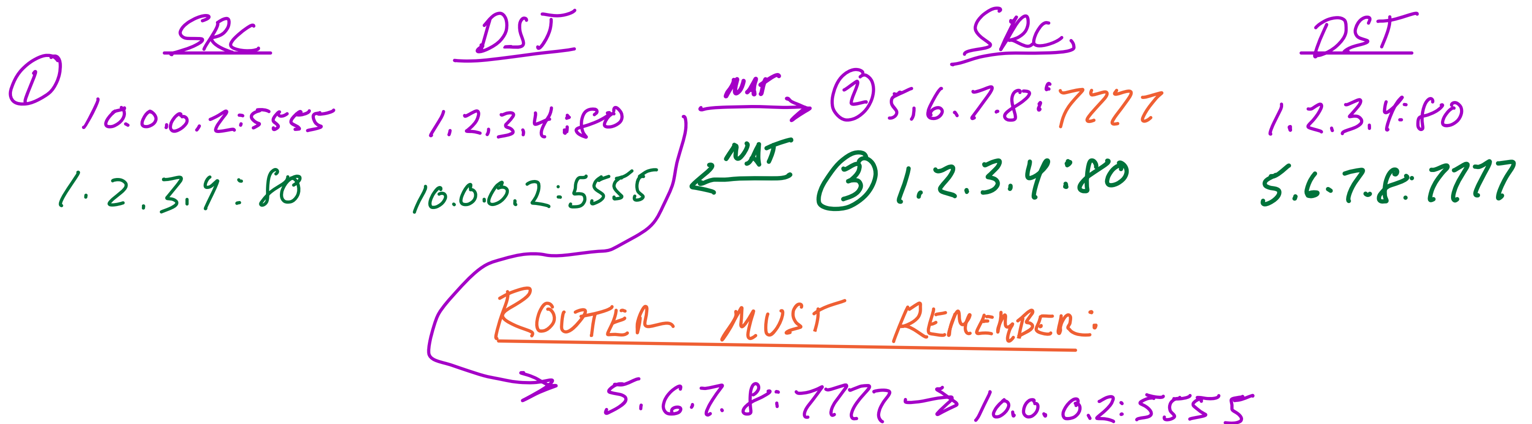
=> **Router makes some (sketchy) assumptions about traffic**

NAT translation: an example



INSIDE: 10.0.0.0/24

Suppose A wants to connect to S on port 80:



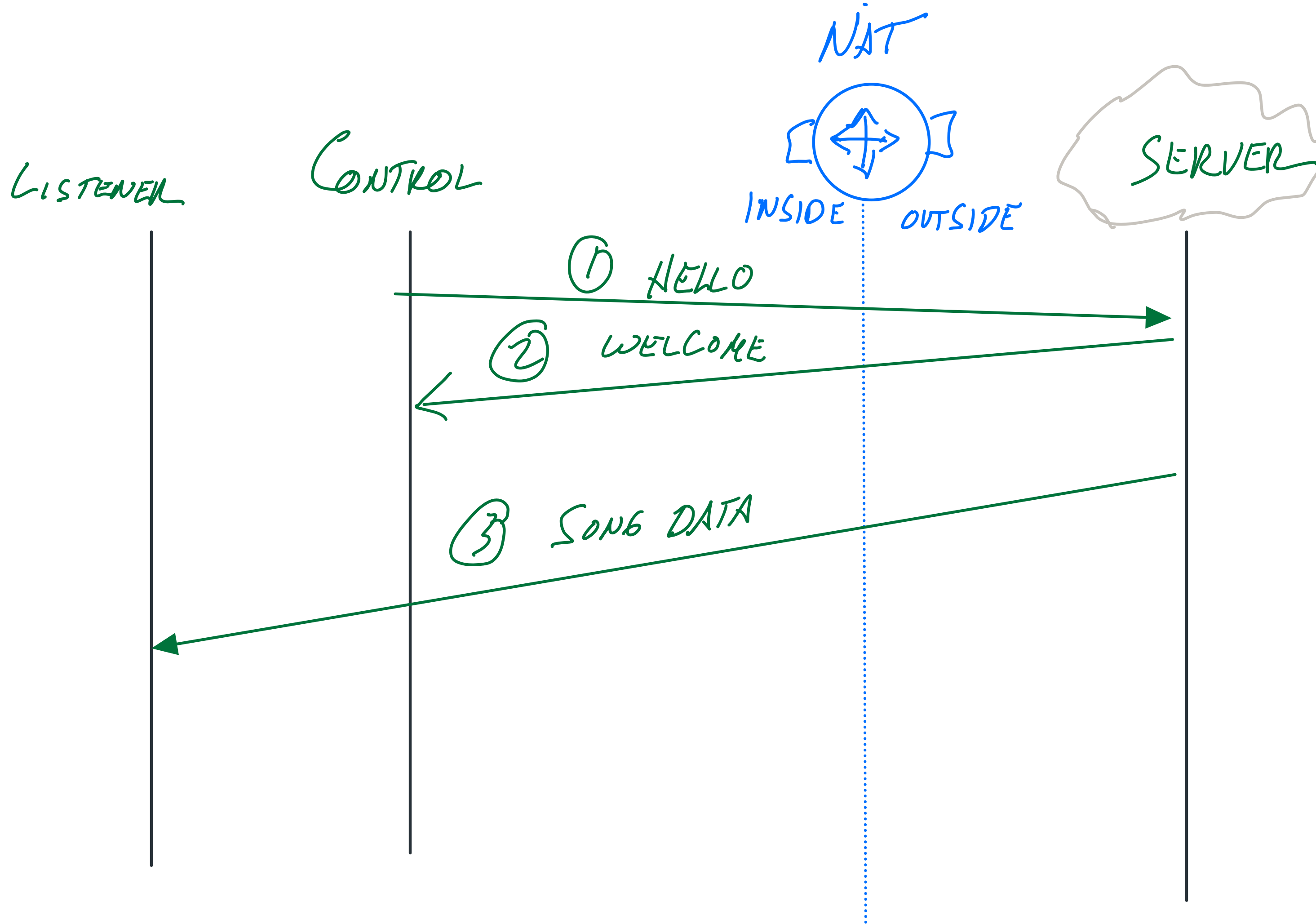
Key points

-- Router needs to keep track of state to remember how to "translate" packets

-- Can run out of possible translations (number of ports, space in table)
=> Big NATs (eg. Brown) have multiple outside IPs to increase number of possible translations

-- Router needs to figure out when connections start and end, so it can clean up table
=> Kinda works for TCP (protocol has well-defined start and end), sketchy for UDP (need to make assumptions based on timing)

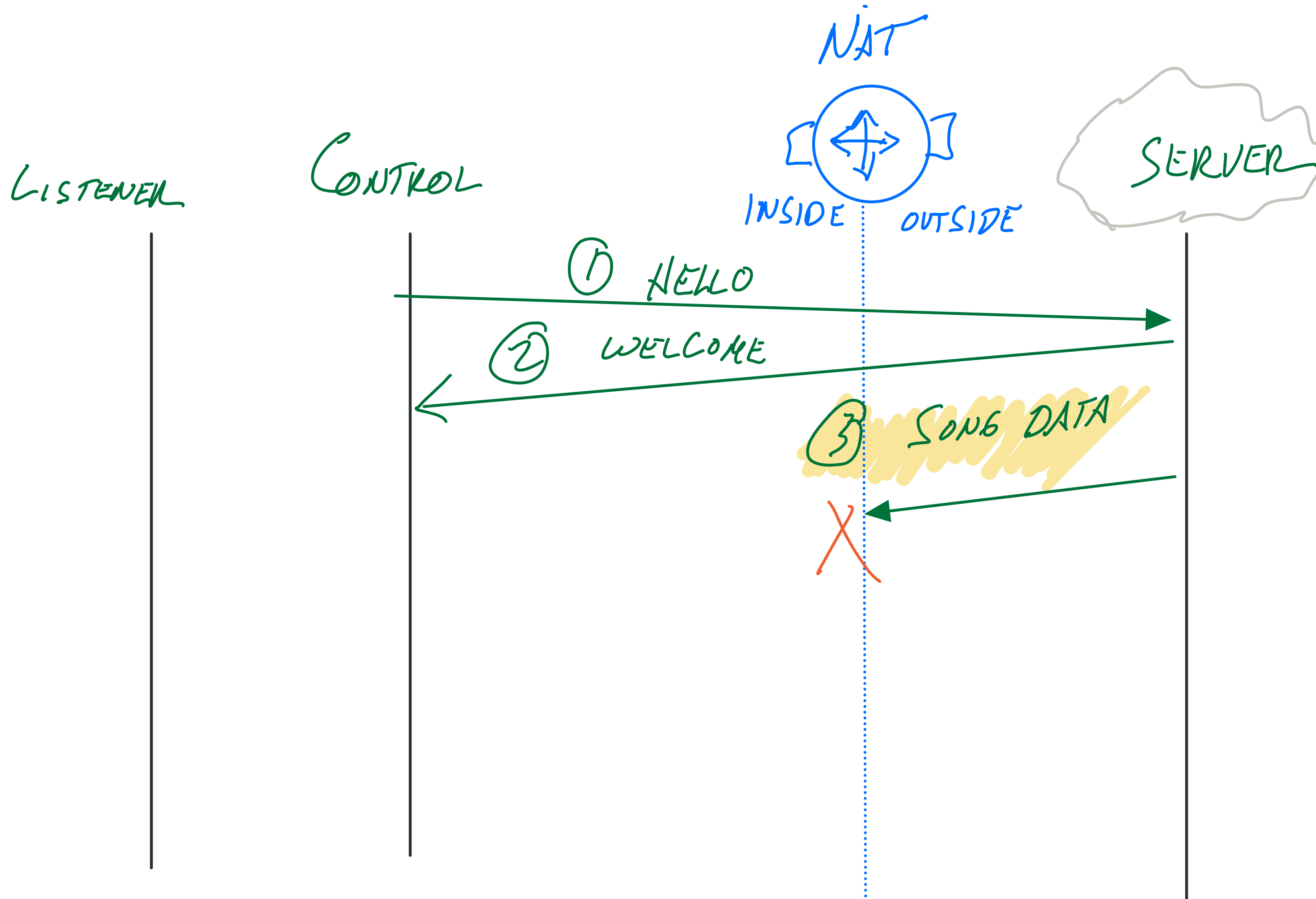
NAT vs. SNOWCAST



Which of these steps will not work if the client is behind a NAT????

(More notes on NAT we'll discuss later....)

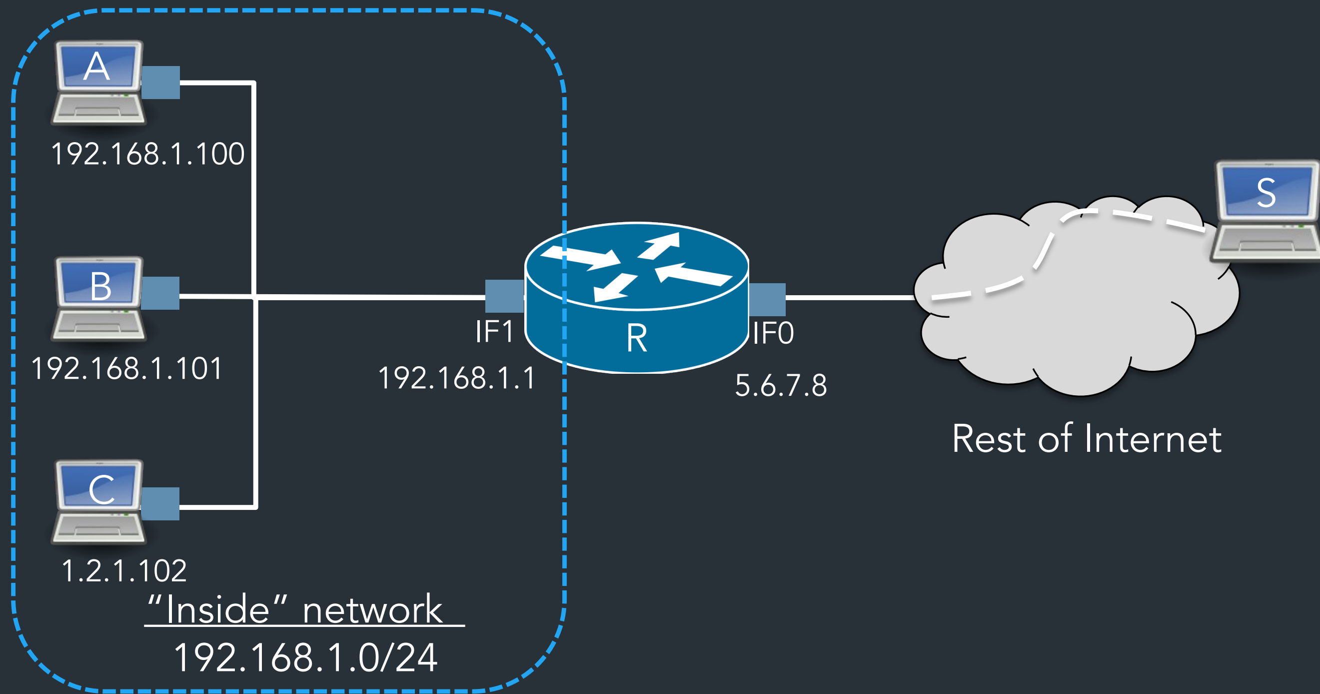
NAT vs. SNOWCAST



Which of these steps will not work if the client is behind a NAT????

=> Server won't be able to send data to the listener! When the server sends a packet to the listener client, there's no translation rule for the listener's port, so the packet will get dropped!

=> In general, an outside host can't send packets to an inside host unless the inside host has made a connection first



What happens when outside host S wants to connect to inside host A?

Can't do it (at least without special setup)!

⇒ By default, R only knows how to translate packets for connections originating from INSIDE the network

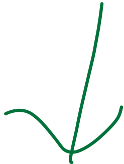
⇒ Breaks end to end connectivity!!!

Comparison: gRPC vs. REST API workflow

GRPC / RPC

PROTO. FILE (IDL)

LIBRARY



USE STUBS TO CALL FUNCTIONS

REST API

DEFINE SERVER ENDPOINT

/ORG/REPOS

CLIENT MAKES HTTP REQUEST

JSON.

RPC

Define messages/formats in IDL (eg. protobuf)

RPC tools generate library code for your language

Write programs that use stubs to call functions

APP NEEDS TO DO LESS SERIALIZATION.

More expensive software dependency (RPC library)
=> Often when app developer has more control over both ends

Message format is a lot smaller

REST

REST APIS
- Define endpoints for applications to access
- Publish API specification for URLs
- Make HTTP requests, receive response in e.g., JSON

MORE UNSTRUCTURED/
MORE FLEXIBLE
FORMAT

Can leverage existing tools for HTTP, caching, etc.