CSCI-1680
APIs

Nick DeMarinis

Based partly on lecture notes by Rodrigo Fonseca, Scott Shenker and John Jannotti

Administrivia

Final project is online

- Group registration form: due tomorrow (11/29) by 5pm EST
- Brief proposal: due Friday 12/1, no late days permitted!!
  - We will review all of these over the weekend

- HW4 (probably last HW): out this week, due next week

- TCP grading: end of this week, early next week
  - Look for email today/tomorrow

Project examples

- Make your own iterative DNS resolver
- Build a simple HTTP server
- Make your own web API for something
- Implement Snowcast, etc. using RPCs (more next week)
- Extend your IP/TCP in some way…

These are only a few ideas!

HTTP request: a way to fetch (**GET**) or send (**POST**) some object

- Doesn't need to be a web page
- Doesn't need to be from a browser

⇒ Generic way to ask the server to do something => an API over the network!

How do programs communicate?

Need a protocol! We've seen lots of examples….
IP, TCP, ICMP, RIP, OSPF, BGP, DNS, HTTP, Snowcast …

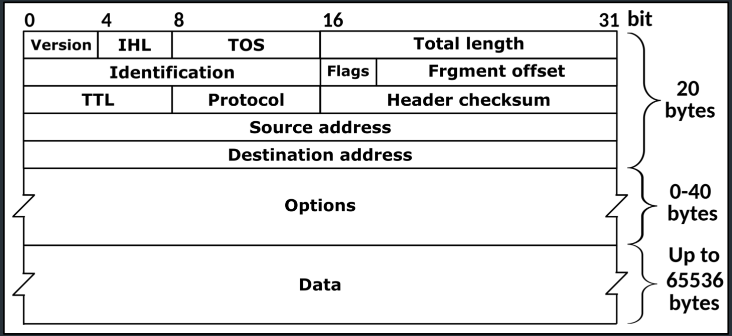
L3, L4.

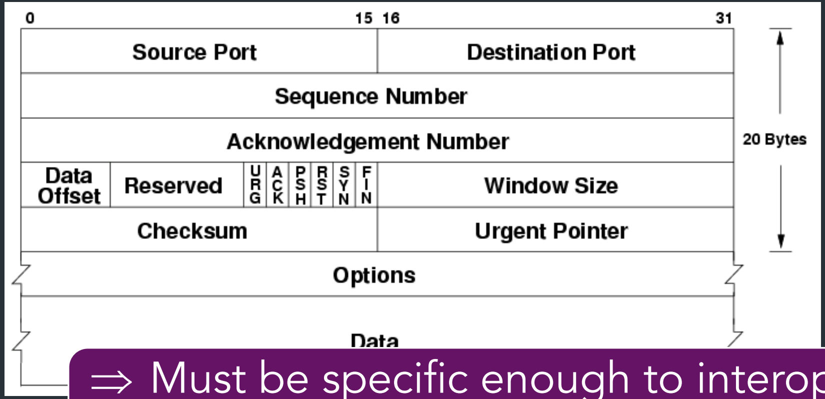
APPS

- PACKET STRUCTURE
- STATE MACHINE
- IMPL AGNOSTIC/INTEROPERABILITY
- INTERFACE.

Requirements for protocols

Data representation (headers, packet formats)





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

```
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
```

```
Network Working Group                                       D. Waitzman
Request for Comments: 1149                                      BBN STC
                                                           1 April 1990


   A Standard for the Transmission of IP Datagrams on Avian Carriers

Status of this Memo

   This memo describes an experimental method for the encapsulation of
   IP datagrams in avian carriers.  This specification is primarily
                                                    an experimental, not
                                                    no is unlimited.

                                                    oughput, and low
```

⇒ Must be specific enough to interoperate
(support multiple architectures, byte orders, languages, locales …)

When 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;
```

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

```
// 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 {
                . . .
        }
```

All the protocols you've made so far (+IP, TCP, RIP, …):
manually packing bytes into buffers

All the protocols you’ve been writing so far: manually loading 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 this what you should be doing?

Almost certainly not.

How SHOULD you write a protocol outside this class?

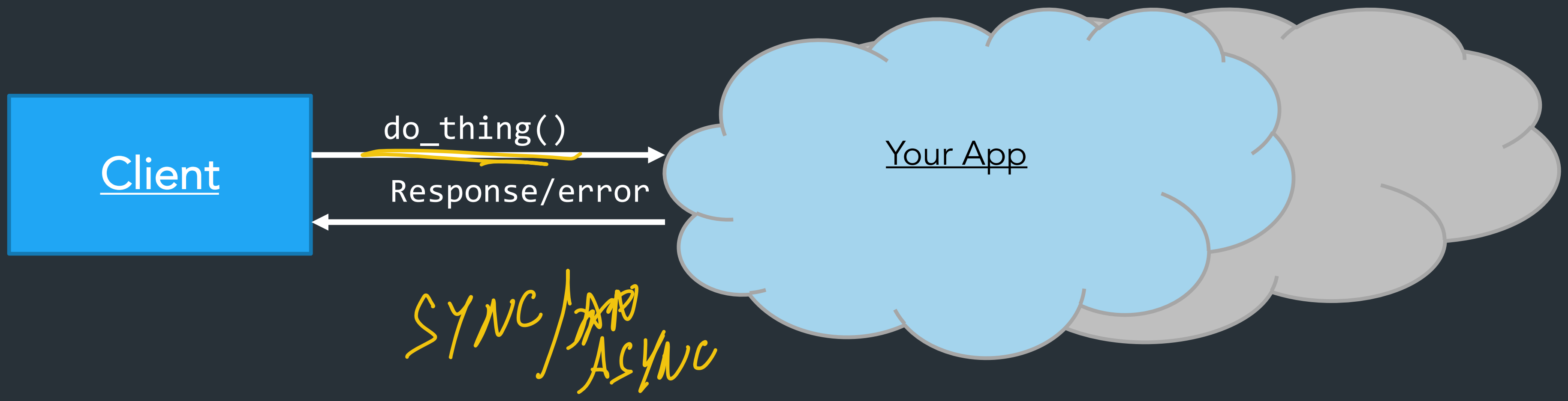
And why?

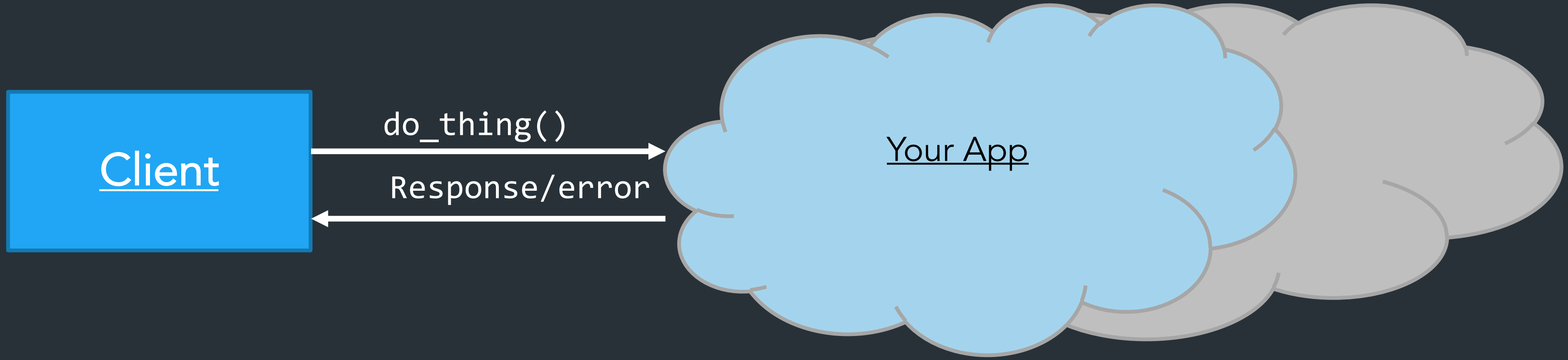
** At least, how to start thinking about it*

Typical application goal: make an API for something

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

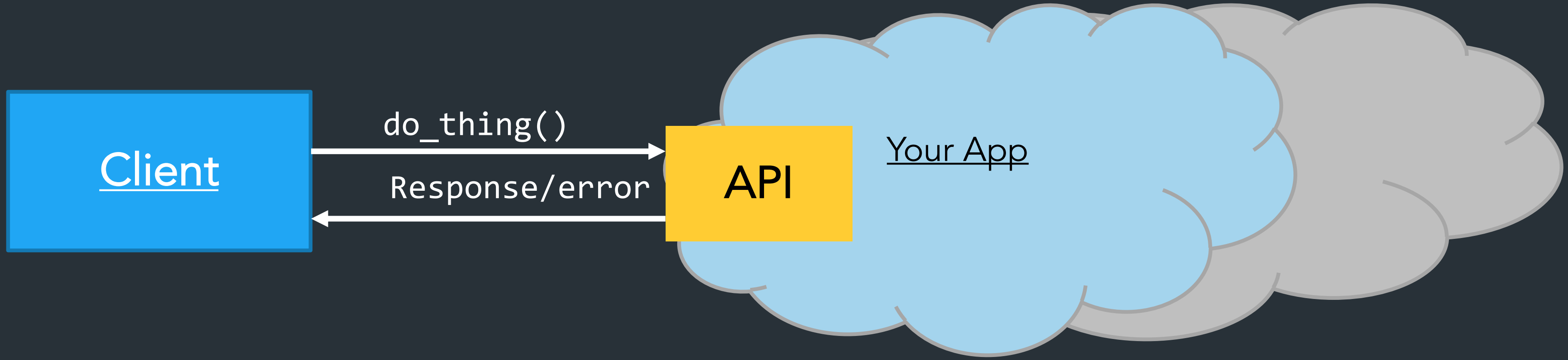
Want: end-user to be able to use your app
- Read data
- Write/upload data





Challenges/Requirements

- Heterogeneous devices (desktop/mobile, different OSes)
- Application will change
- Number of user devices will scale
- Number of services/services will scale too!



Would like to have a generic API for interacting with application services

=> Flexible to changes

=> Easy to scale

Why doesn't this work?

– ERROR-PRONE W/ CHANGES

– DON'T WANT TO CARE ABOUT FORMAT

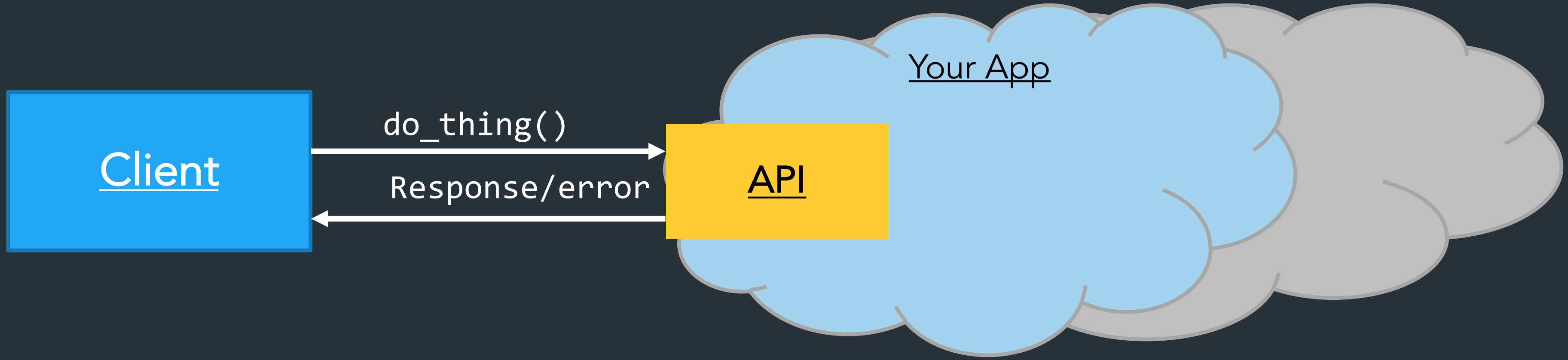
Client to Server Commands

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;
```

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

```
// 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()
}
```

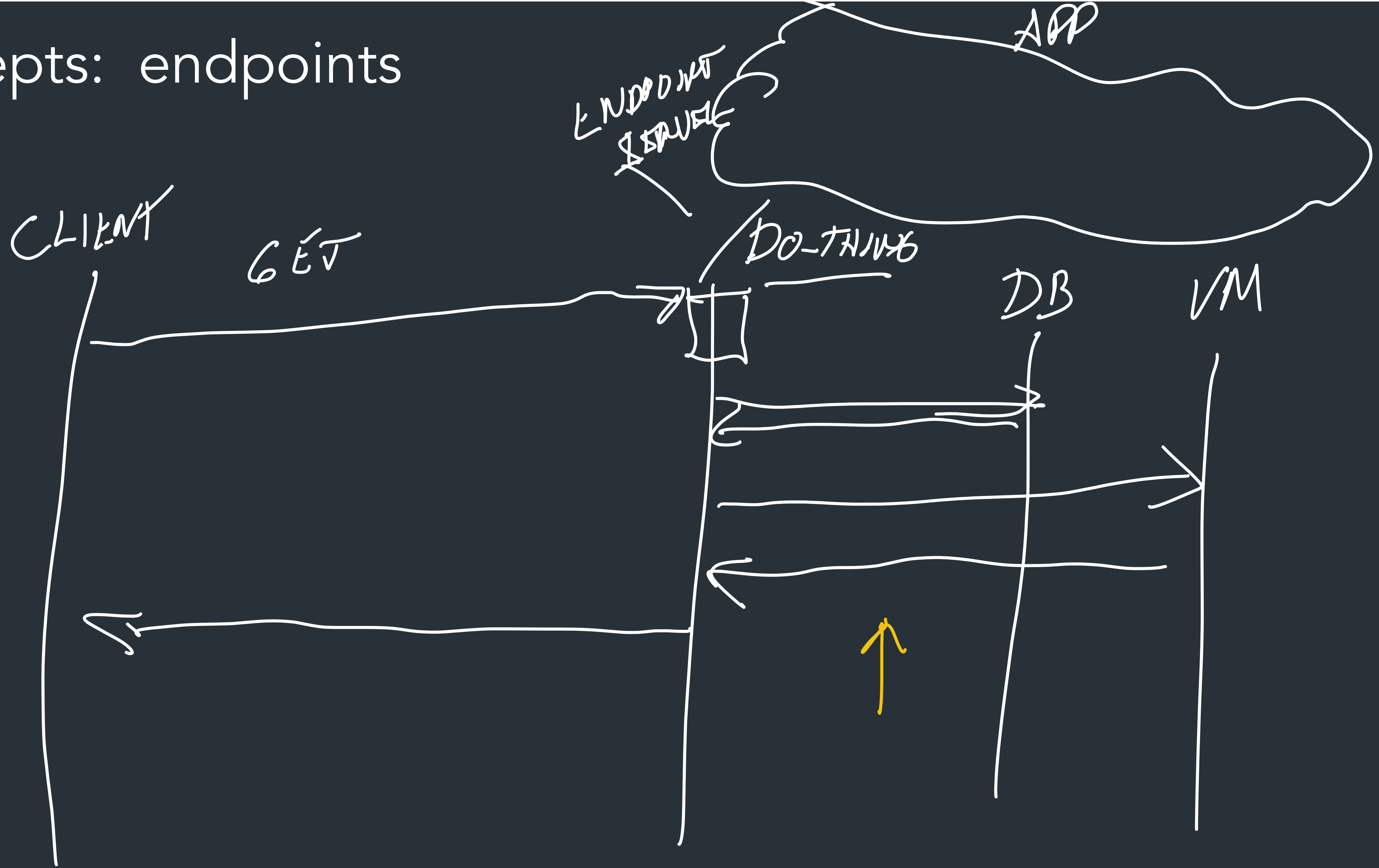


Usually, build on existing tools that can define the API for you
=> Creates endpoints where you write code to perform actions

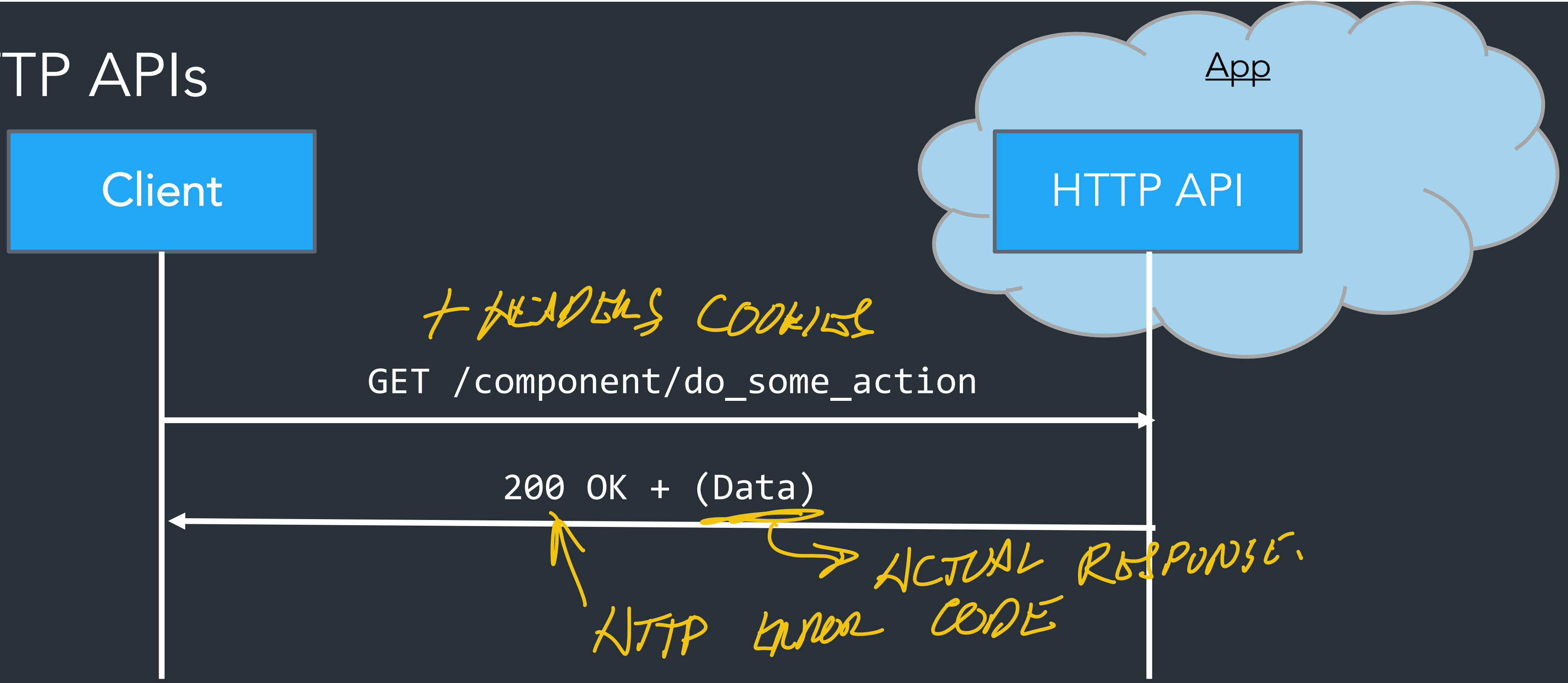
=> Don't need to worry about serializing/deserializing messages

=> Build on existing protocols to handle scaling
(eg. HTTP proxies, load balancing, caching, etc.)

Concepts: endpoints



HTTP APIs



- Endpoints at various URLs
- Usually: Request data with GET, upload with POST
- Client authenticates/passes inputs data with headers, cookies
- Response normally JSON, XML, or other self-describing format

TERMINAL WAY TO DO WEB REQUESTS

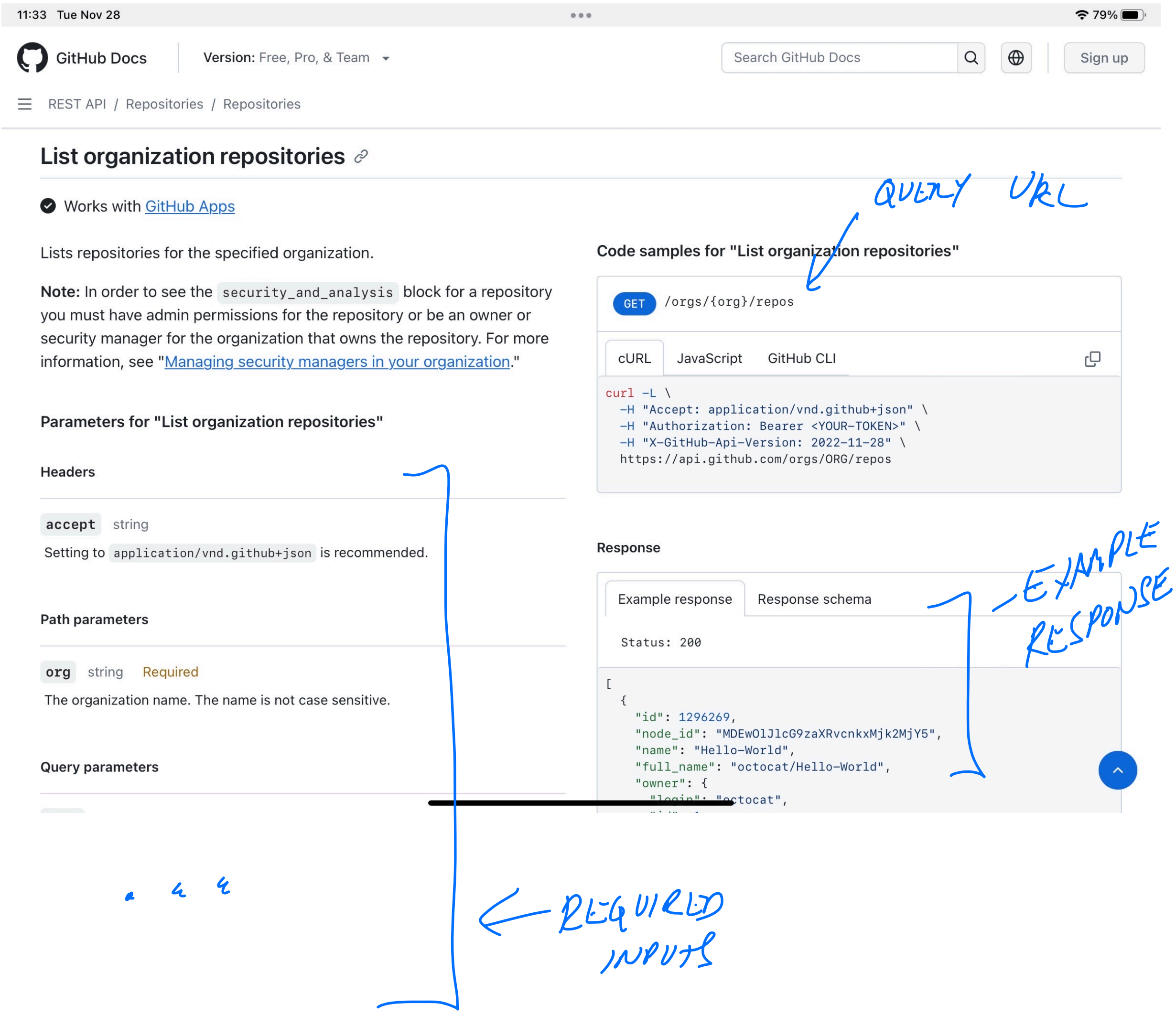
```
curl -X GET 'https://www.gradescope.com/courses/567871/memberships.csv’
  -H 'User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:109.0) Gecko/20100101
  Firefox/118.0’
  -H 'Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8'
  -H 'Accept-Language: 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'
```

HEADERS

Example: docs for Github's REST API

Here's one method for listing the repositories in a github org

For more: https://docs.github.com/en/rest



11:33 Tue Nov 28 ••• 79%

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

REST API / Repositories / Repositories

List organization repositories

Works with GitHub Apps

Lists repositories for the specified organization.

Note: In order to see the `security_and_analysis` block for a repository you must have admin permissions for the repository or be an owner or security manager for the organization that owns the repository. For more information, see "Managing security managers in your organization."

Parameters for "List organization repositories"

Headers

`accept` string

Setting to `application/vnd.github+json` is recommended.

Path parameters

`org` string Required

The organization name. The name is not case sensitive.

Query parameters

Code samples for "List organization repositories"

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

Status: 200

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

Example: Github public API

```
$ curl https://api.github.com/users/ndemarinis
{
  "login": "ndemarinis",
  "id": 1191319,
  "node_id": "MDQ6VXNlcjExOTEzMTk=",
  "avatar_url": "https://avatars.githubusercontent.com/u/1191319?v=4",
  "gravatar_id": "",
  "url": "https://api.github.com/users/ndemarinis",
  "type": "User",
  "site_admin": false,
  "name": "Nick DeMarinis",
  "blog": "https://vty.sh",
  "twitter_username": null,
  "public_repos": 10,
 . . .
}
```

Why is this useful?

- HTTP is ubiquitous
- Lots of existing tools to scale HTTP
  - Cookies etc. for user authentication
  - Proxies/load balancers

Why use JSON/etc vs. a binary encoding?

```
// Here's an example JSON response from the Github API when querying for info
// about a repo (eg. GET https://api.github.com/repositories/org/something
// Q:  Why bother using JSON when we could use a binary format?  A binary format
// would use so much less space!
//  - If we had a binary format, both sides would need to
//     know how the data is organized
//       => This is a "Self-describing format"  (eg. JSON, YAML, XML, ...)
//          - need a lot less info up front on each device using it
// - Human-readable
// - Easy to use by web tools (JSON works well with Javascript)
// - Can leverage web caching, proxies, load-balancers, etc.
//
[
  {
    "id": 1296269,
    "node_id": "MDEwOlJlcG9zaXRvcnkxMjk2MjY5",
    "name": "Hello-World",
    "full_name": "octocat/Hello-World",
    "owner": {
      "login": "octocat",
      "id": 1,
      "node_id": "MDQ6VXNlcjE=",
      "avatar_url": "https://github.com/images/error/octocat_happy.gif",
      "gravatar_id": "",
      "url": "https://api.github.com/users/octocat",
      "html_url": "https://github.com/octocat",
      "followers_url": "https://api.github.com/users/octocat/followers",
      "following_url":
"https://api.github.com/users/octocat/following{/other_user}",
      "gists_url": "https://api.github.com/users/octocat/gists{/gist_id}",
      "starred_url":
"https://api.github.com/users/octocat/starred{/owner}{/repo}",
      "subscriptions_url": "https://api.github.com/users/octocat/subscriptions",
      "organizations_url": "https://api.github.com/users/octocat/orgs",
      "repos_url": "https://api.github.com/users/octocat/repos",
      "events_url": "https://api.github.com/users/octocat/events{/privacy}",
      "received_events_url":
"https://api.github.com/users/octocat/received_events",
      "type": "User",
      "site_admin": false
    },
    "private": false,
    "html_url": "https://github.com/octocat/Hello-World",
```

Note: these are other API endpoints!
=> a form of indirection, refers to other places we can query for even more info!

```
    "labels_url":
"https://api.github.com/repos/octocat/Hello-World/labels{/name}",
    "languages_url":
"https://api.github.com/repos/octocat/Hello-World/languages",
    "merges_url": "https://api.github.com/repos/octocat/Hello-World/merges",
    "milestones_url":
"https://api.github.com/repos/octocat/Hello-World/milestones{/number}",
    "notifications_url":
"https://api.github.com/repos/octocat/Hello-World/notifications{?since,all,partic
ipating}",
    "pulls_url":
"https://api.github.com/repos/octocat/Hello-World/pulls{/number}",
    "releases_url":
"https://api.github.com/repos/octocat/Hello-World/releases{/id}",
    "ssh_url": "git@github.com:octocat/Hello-World.git",
    "stargazers_url":
"https://api.github.com/repos/octocat/Hello-World/stargazers",
    "statuses_url":
"https://api.github.com/repos/octocat/Hello-World/statuses/{sha}",
    "subscribers_url":
"https://api.github.com/repos/octocat/Hello-World/subscribers",
    "subscription_url":
"https://api.github.com/repos/octocat/Hello-World/subscription",
    "tags_url": "https://api.github.com/repos/octocat/Hello-World/tags",
    "teams_url": "https://api.github.com/repos/octocat/Hello-World/teams",
    "trees_url":
"https://api.github.com/repos/octocat/Hello-World/git/trees{/sha}",
    "clone_url": "https://github.com/octocat/Hello-World.git",
    "mirror_url": "git:git.example.com/octocat/Hello-World",
    "hooks_url": "https://api.github.com/repos/octocat/Hello-World/hooks",
    "svn_url": "https://svn.github.com/octocat/Hello-World",
    "homepage": "https://github.com",
    "language": null,
    "forks_count": 9,
    "stargazers_count": 80,
    "watchers_count": 80,
    "size": 108,
    "default_branch": "master",
    "open_issues_count": 0,
    "is_template": false,
    "topics": [
      "octocat",
      "atom",
```


NOT JUST STRINGS!

```
      "electron",
      "api"
    ],
    "has_issues": true,
    "has_projects": true,
    "has_wiki": true,
    "has_pages": false,
    "has_downloads": true,
    "has_discussions": false,
    "archived": false,
    "disabled": false,
    "visibility": "public",
    "pushed_at": "2011-01-26T19:06:43Z",
    "created_at": "2011-01-26T19:01:12Z",
    "updated_at": "2011-01-26T19:14:43Z",
    "permissions": {
      "admin": false,
      "push": false,
      "pull": true
    },
    "security_and_analysis": {
      "advanced_security": {
        "status": "enabled"
      },
      "secret_scanning": {
        "status": "enabled"
      },
      "secret_scanning_push_protection": {
        "status": "disabled"
      }
    }
  }
]
```


TIMESTAMP MAY STILL NEED EXTRA WORK TO DESERIALIZE THIS

What if you need more flexibility?

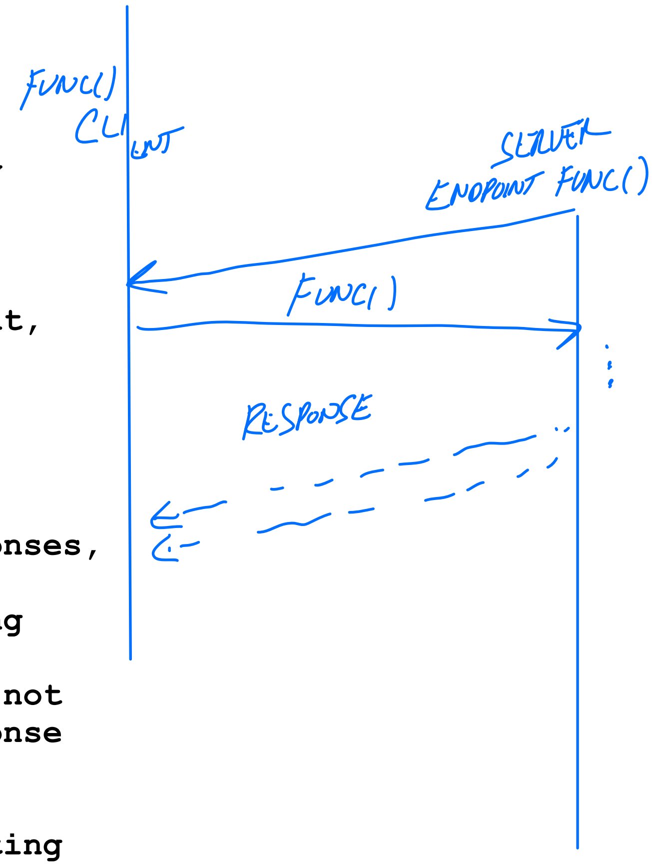
RPC (Remote procedure call)
- Basically, make a function call happen over the network in some way

— Defining how the data should be formatted
  => Could be a custom binary format, could still be JSON, …

- Semantics for messages
  - What happens when there's an error? (timeout, retry, etc.)
  - one request => multiple responses, or vice versa
  - Could be blocking/non-blocking

=> More flexibility vs. HTTP since not constrained to HTTP's request/response semantics

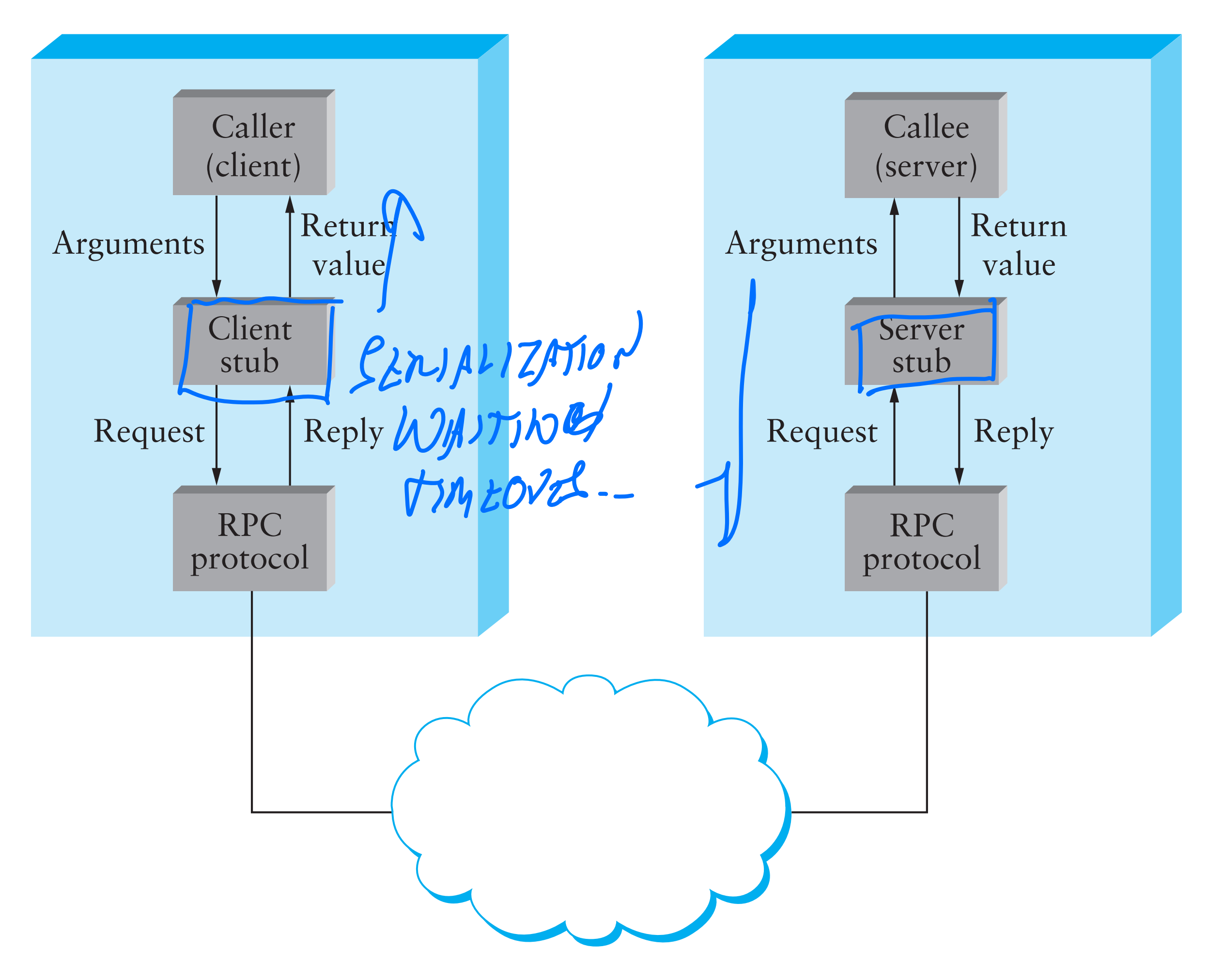
Imagine Snowcast but after abstracting away abstracting away the logic for when to send and how to wait for snowcast message, handle timeouts



Lots of examples of RPC frameworks:
- RPC (Network file system (NFS)
- gRPC: Google's RPC framework
- Apache thrift
- Java remote methods

In general, an RPC framework provides the following:
- A way to define messages you want to send/receive
- Semantics for how they work (sync/async, one-to-many, etc)
- A way to describe the data format

=> Provides library to use in your implementation:
 => Client: generates "stubs" where you can call functions in your code, serializes request + args, which go to network
 =>


Caller
(client)
Arguments
Return
value
Client
stub
Request
Reply
RPC
protocol
Callee
(server)
Arguments
Return
value
Server
stub
Request
Reply
RPC
protocol
SERIALIZATION
WAITING
TIMEOUT...

Stub Functions

- Local stub functions at client and server give appearance of a local function call
- client stub
  - marshalls parameters -> sends to server -> waits
  - unmarshalls results -> returns to client
- server stub
  - creates socket/ports and accepts connections
  - receives message from client stub -> unmarshalls parameters -> calls server function
  - marshalls results -> sends results to client stub

Some examples

- gRPC
- Apache Thrift
- JSON-RPC
- XML-RPC, SOAP
- . . .

Alternative to self-describing data (JSON, XML, YAML, etc.) is to pre-define the schema for the data in a way that the framework can use

<u>IDL (Interface Description Language)</u>:
  => Specify precisely what you want the data format to look like
   => Framework generates code that does the serialization (think: header files, class/struct defs)

Gives you: basic integer types, arrays, maps, enums, string, etc.

IDL provides:
  => Serialization for these basic types, code generation to stitch this together to serialize data structures

Example: gRPC

- IDL-based, defined by Google
  - Protocol Buffers as IDL
- User specifies services, calls
  - Single and streaming calls
  - Support for timeouts, cancellations, etc
- Transport: based on HTTP/2

```
service HelloService {
  rpc SayHello (HelloRequest)
  returns (HelloResponse);
}

message HelloRequest {
  string greeting = 1;
}
message HelloResponse {
  string reply = 1;
}
```

Input

IDL:

gRPC

- Generates stubs in many languages
  - C/C++, C#, Node.js, PHP, Ruby, Python, Go, Java
  - These are interoperable
- Transport is http/2

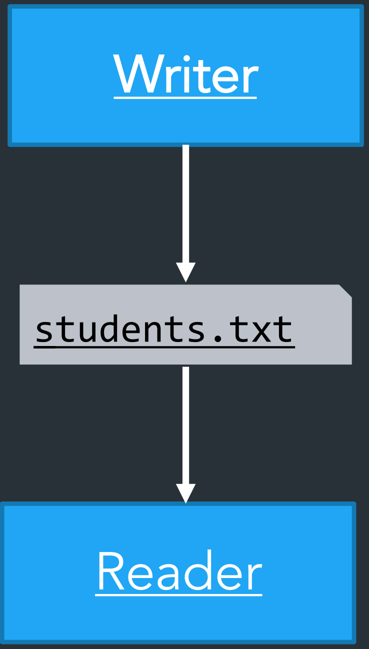
Protocol Buffers

(EG PROTOBUF)

- Defined by Google, released to the public
  - Widely used internally and externally
  - Supports common types, service definitions
  - Natively generates C++/Java/Python/Go code
    - Over 20 other supported by third parties
  - Efficient binary encoding, readable text encoding
- Performance
  - 3 to 10 times smaller than XML
  - 20 to 100 times faster to process

Protocol Buffers Example (for a file)

```
message Student {
  required String name = 1;
  required int32 credits = 2;
}
```




```
Student s;
s.set_name(“Jane”);
s.set_credits(20);
fstream output(“students.txt” , ios:out | ios:binary );
s.SerializeToOstream(&output);
```

```
Student s;
fstream input(“students.txt” , ios:in | ios:binary
  );
s.ParseFromIstream();
```

Conclusions

- Unless you *really* want to optimize your protocol for performance, use an IDL
- Parsing code is easy to get (slightly) wrong, hard to make fast—only want to do this once!
- Which one should you use?

EXTRA CONTENT
IF YOU WANT
TO READ FURTHER!

Which data types?

- Basic types
  - Integers, floating point, characters
  - Some issues: endianness (ntohs, htons), character encoding, IEEE 754
- Flat types
  - Strings, structures, arrays
  - Some issues: packing of structures, order, variable length
- Complex types
  - Pointers! Must flatten, or serialize data structures

protobuf: Binary Encoding

- Variable-length integers
  - 7 bits out of 8 to encode integers
  - Msb: more bits to come
  - Multi-byte integers: least significant group first
- Signed integers: zig-zag encoding, then varint
  - 0:0, -1:1, 1:2, -2:3, 2:4, …
  - Advantage: smaller when encoded with varint
- General:
  - Field number, field type (tag), value
- Strings:
  - Varint length, unicode representation

Apache Thrift

- Originally developed by Facebook
- Used heavily internally
- Supports (at least): C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, Smalltalk, and Ocaml
- Types: basic types, list, set, map, exceptions
- Versioning support
- Many encodings (protocols) supported
  - Efficient binary, json encodings