[544] gRPC

Tyler Caraza-Harter

Learning Objectives

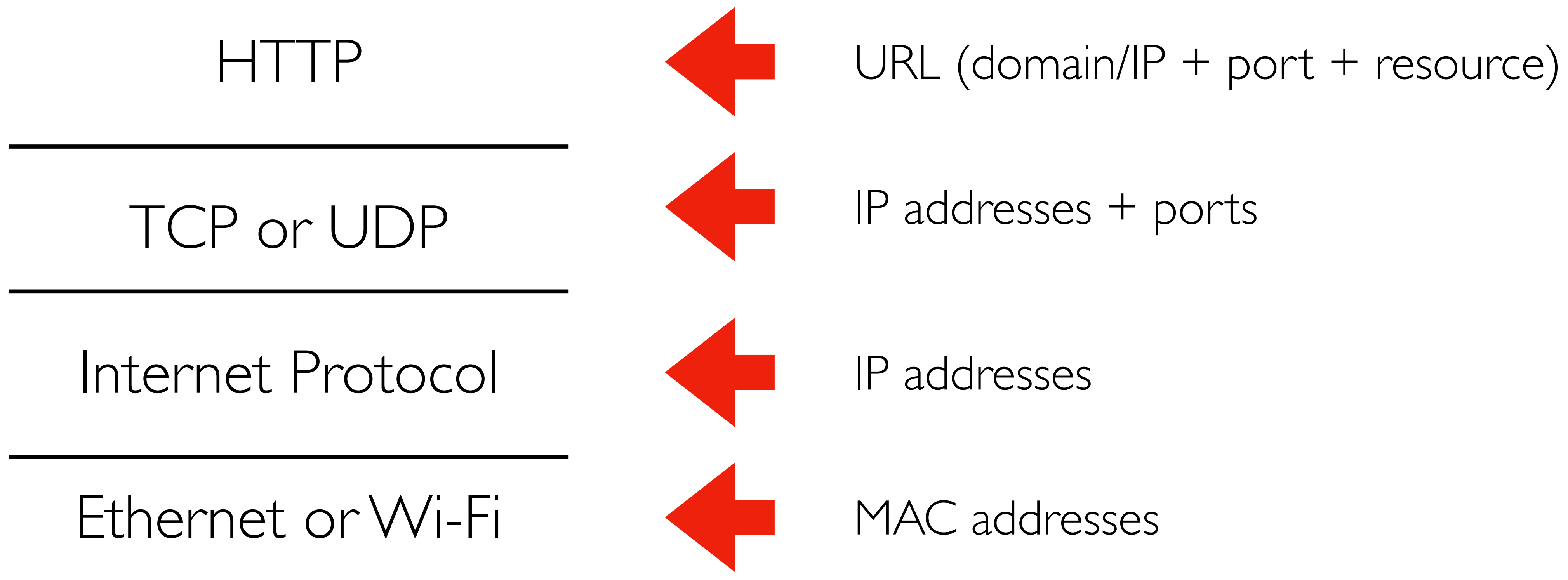
- describe the functionality that HTTP provides (beyond what TCP alone provides)
- call functions remotely via gRPC

Outline

HTTP

gRPC

HTTP (Hypertext Transfer Protocol)



HTTP	URL (domain/IP + port + resource)
TCP or UDP	IP addresses + ports
Internet Protocol	IP addresses
Ethernet or Wi-Fi	MAC addresses

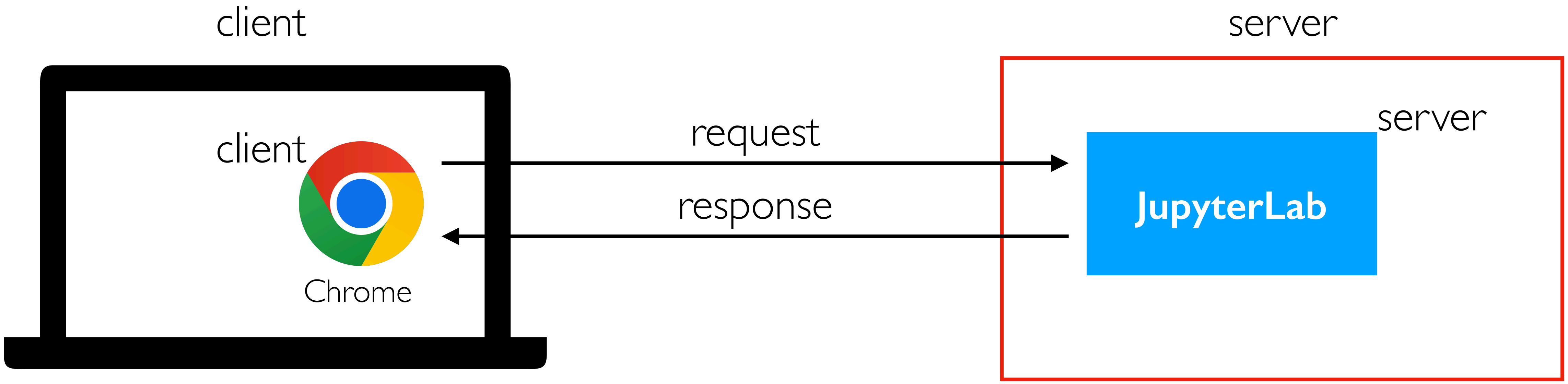
`https://tyler.caraza-harter.com:443/cs544/f24/schedule.html`

domain name `tyler.caraza-harter.com` (mapped to an IP)

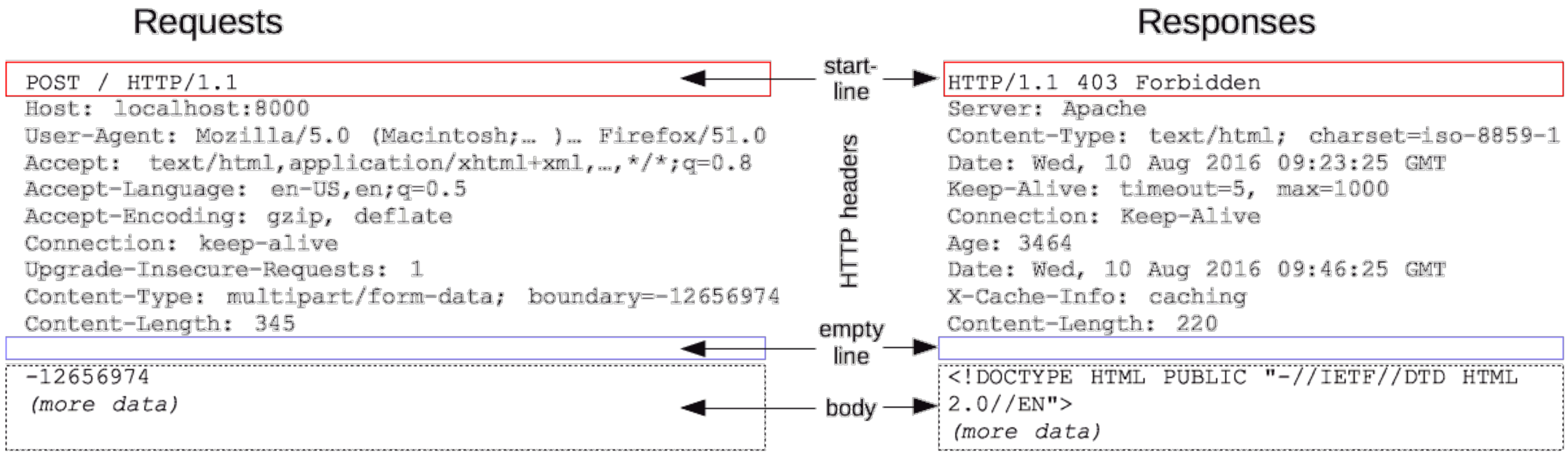
port `443` (443 is default for https)

resource `/cs544/f24/schedule.html`

HTTP Messages Betwen Clients and Servers



Parts: method, resource, status code, headers, body


start-
line
HTTP headers
empty
line
body

Requests

```
POST / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macintosh;… )… Firefox/51.0
Accept: text/html,application/xhtml+xml,…,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Content-Type: multipart/form-data; boundary=-12656974
Content-Length: 345

-12656974
(more data)
```

Responses

```
HTTP/1.1 403 Forbidden
Server: Apache
Content-Type: text/html; charset=iso-8859-1
Date: Wed, 10 Aug 2016 09:23:25 GMT
Keep-Alive: timeout=5, max=1000
Connection: Keep-Alive
Age: 3464
Date: Wed, 10 Aug 2016 09:46:25 GMT
X-Cache-Info: caching
Content-Length: 220

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML
2.0//EN">
(more data)
```

https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages

HTTP Methods (types of messages)

Types of request

- **POST**: create a new resource (request+response have body)
- **PUT**: update a resource (request+response have body, usually)
- **GET**: fetch a resource (response has body)
- **DELETE**: delete a resource
- others...

Canvas REST API example:

`GET https://canvas.wisc.edu/api/v1/conversations`
(see all Canvas conversations in JSON format)

`POST https://canvas.wisc.edu/api/v1/conversations`
(create new Canvas conversation)

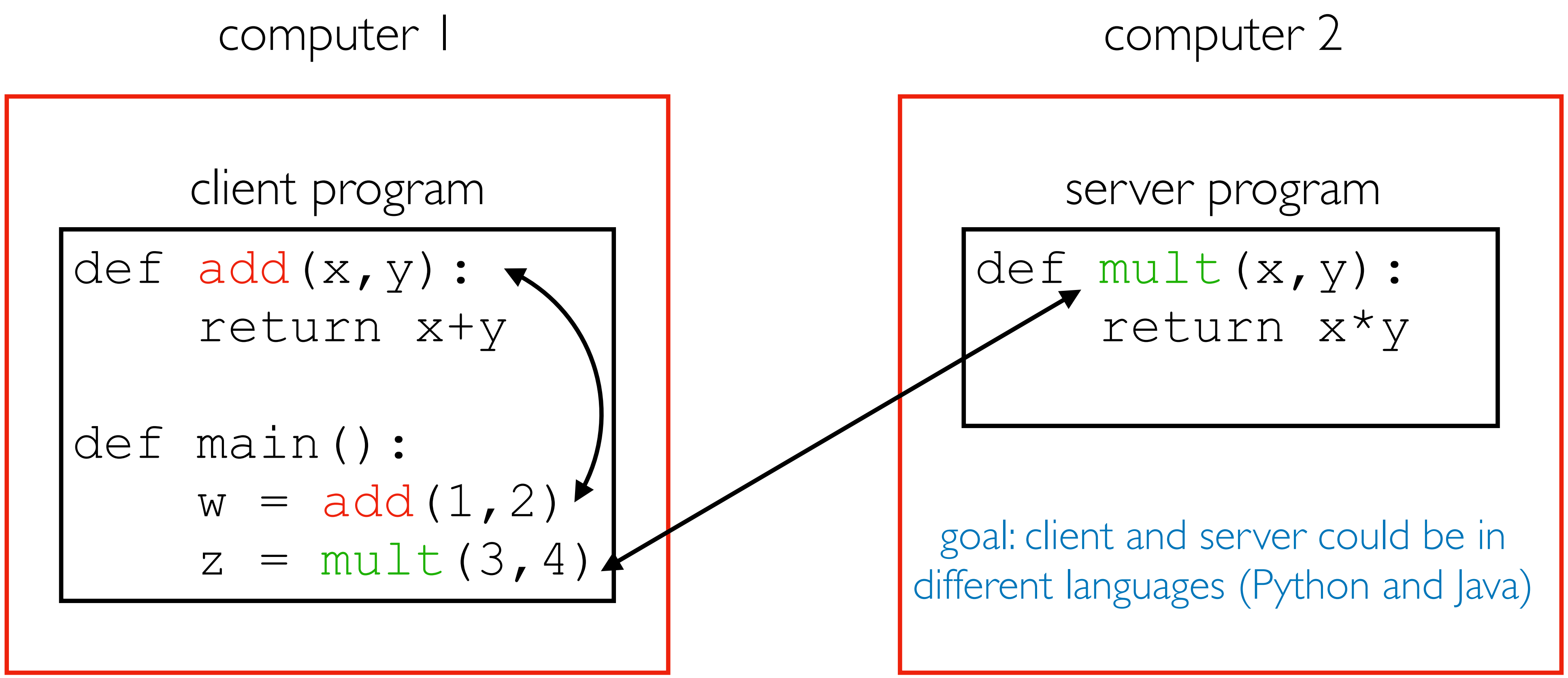
https://canvas.instructure.com/doc/api/conversations.html

Outline

HTTP

gRPC

Remote Procedure Calls (RPCs)



computer 1

client program

```
def add(x,y):
    return x+y

def main():
    w = add(1,2)
    z = mult(3,4)
```

computer 2

server program

```
def mult(x,y):
    return x*y
```

goal: client and server could be in different languages (Python and Java)

procedure = function

- **main** calling **add** is a regular procedure call
- **main** call **mult** is a remote procedure call

There are MANY tools to do RPCs

- Thrift (developed at Meta)
- gRPC (developed at Google) -- this semester

why remote?

- server might have faster hardware
- server might have access to data not directly available to client
- server might be written in different programming language

Example: increase function

```
counts = {
   "A": 123, ...
}

def increase(key, amt):
    counts[key] += amt
    return counts[key]

curr = increase("A", 5)
print(curr) # 128
```

what if we want many programs running on different computers to have access to this dict and the increase function?

Example: increase function

client

```
curr = increase("A", 5)
print(curr) # 128
```

server

```
counts = {
   "A": 123, ...
}

def increase(key, amt):
    counts[key] += amt
    return counts[key]
```

client

...

move counts and increase to a server accessible to many client programs on different computers

Example: increase function

client

```
def increase(key, amt):
    ...code to send






curr = increase("A", 5)
print(curr) # 128
```

computer 1

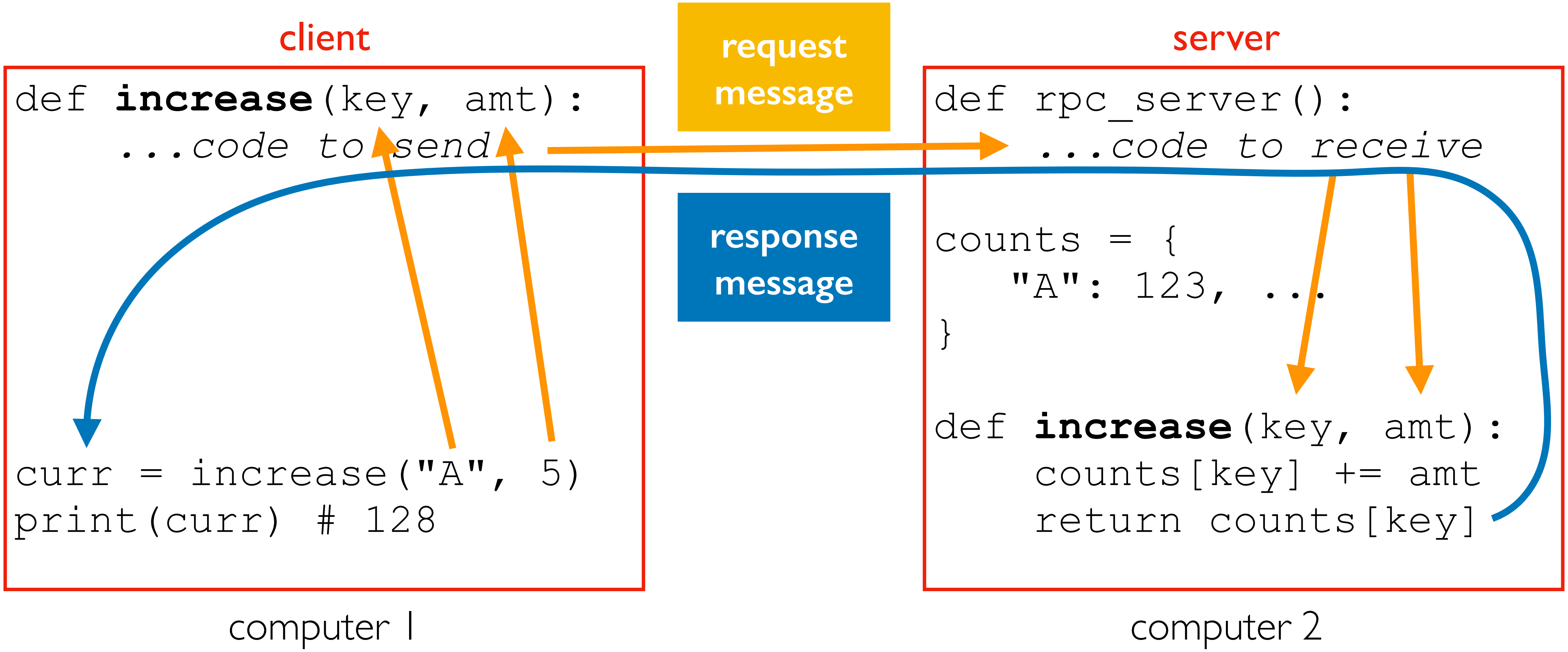
server

```
def rpc_server():
    ...code to receive

counts = {
   "A": 123, ...
}

def increase(key, amt):
    counts[key] += amt
    return counts[key]
```

computer 2

need some extra functions to make calling a remote function *feel* the same as calling a regular one

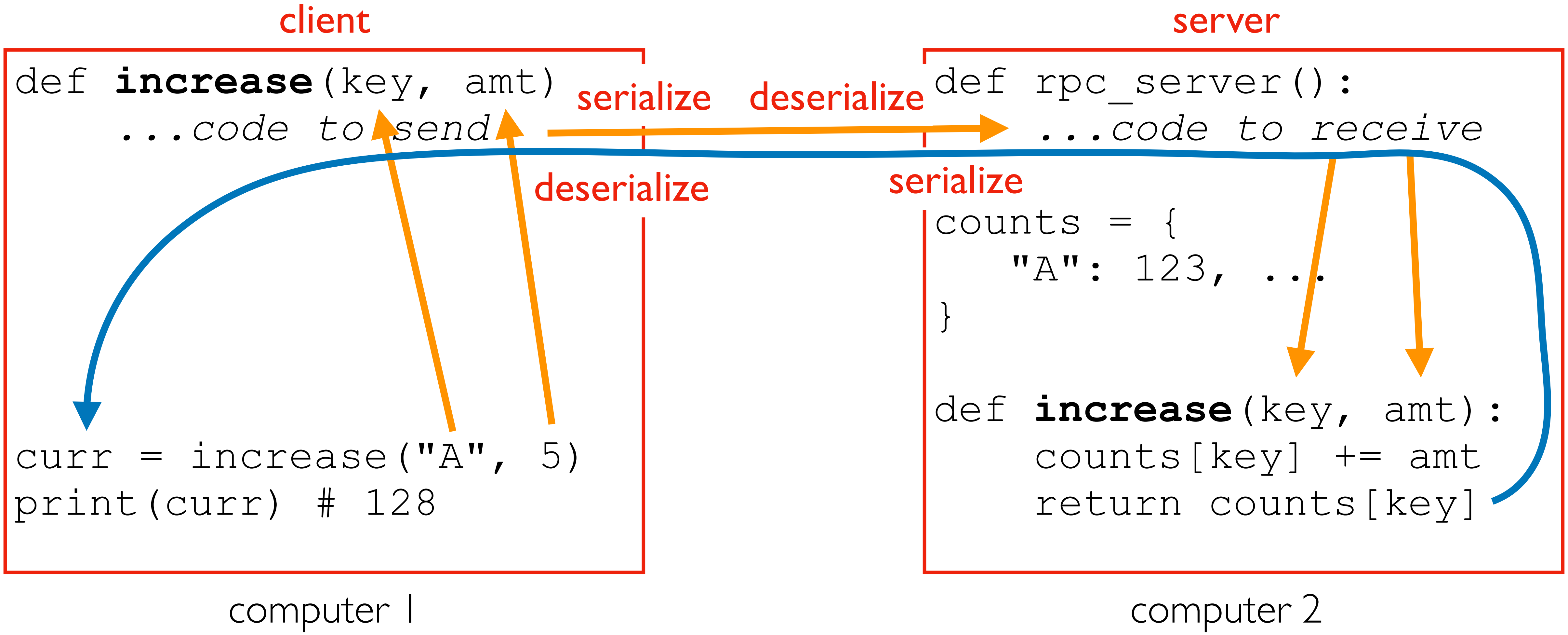
Example: increase function



```
def increase(key, amt):
    ...code to send

curr = increase("A", 5)
print(curr) # 128
```

```
def rpc_server():
    ...code to receive

counts = {
   "A": 123, ...
}

def increase(key, amt):
    counts[key] += amt
    return counts[key]
```

Serialization/Deserialization



client

```
def increase(key, amt)
    ...code to send

curr = increase("A", 5)
print(curr) # 128
```

serialize

deserialize

deserialize

serialize

server

```
def rpc_server():
    ...code to receive

counts = {
   "A": 123, ...
}

def increase(key, amt):
    counts[key] += amt
    return counts[key]
```

computer 1

computer 2

request message

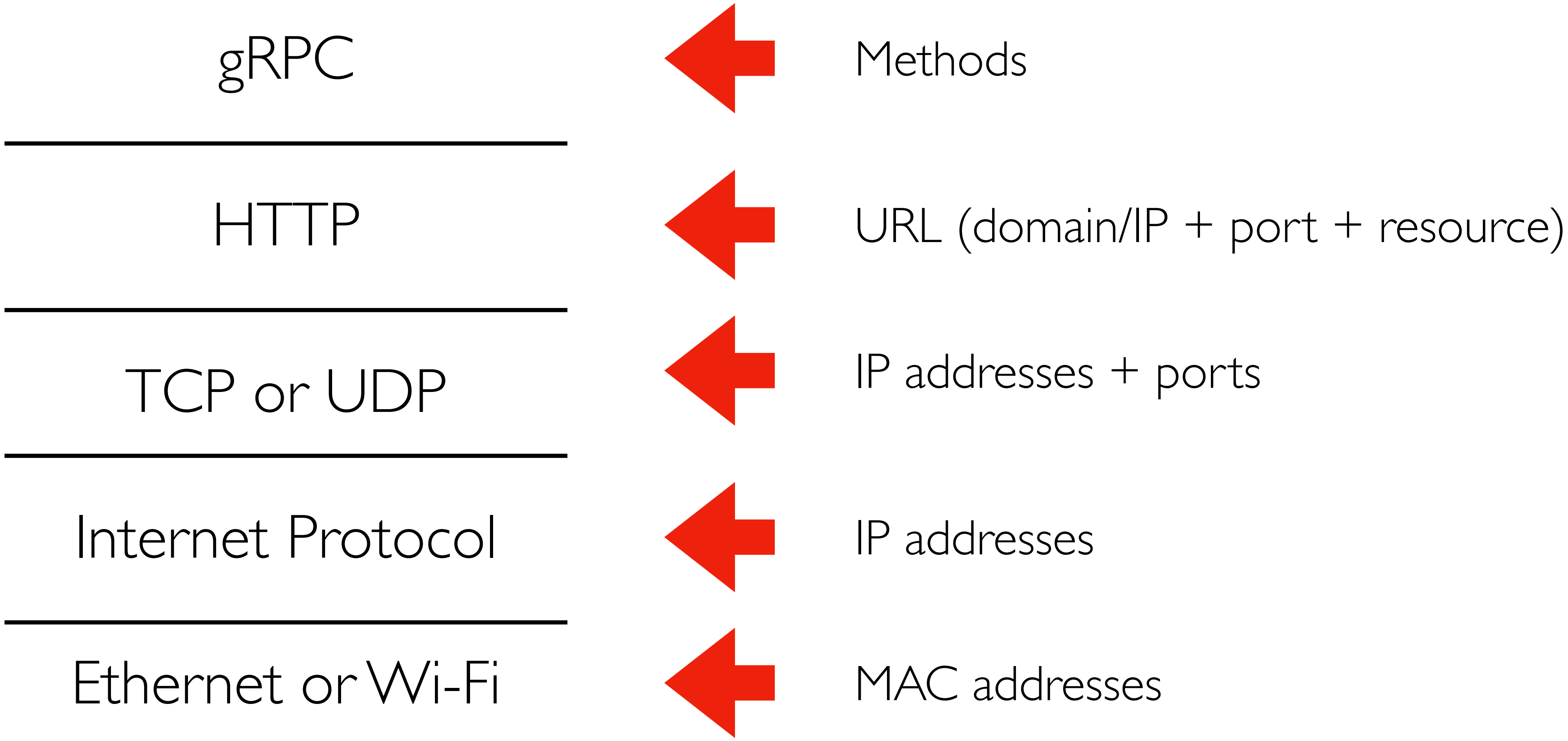
```
args somehow encoded as bytes:
b'{"key": "A"
   "amt": 5}'
```

response message

```
return val as bytes:
b'128'
```

Serialization/deserialization converts to/from bytes. Could be JSON. gRPC uses *protocol buffers*

gRPC builds on HTTP



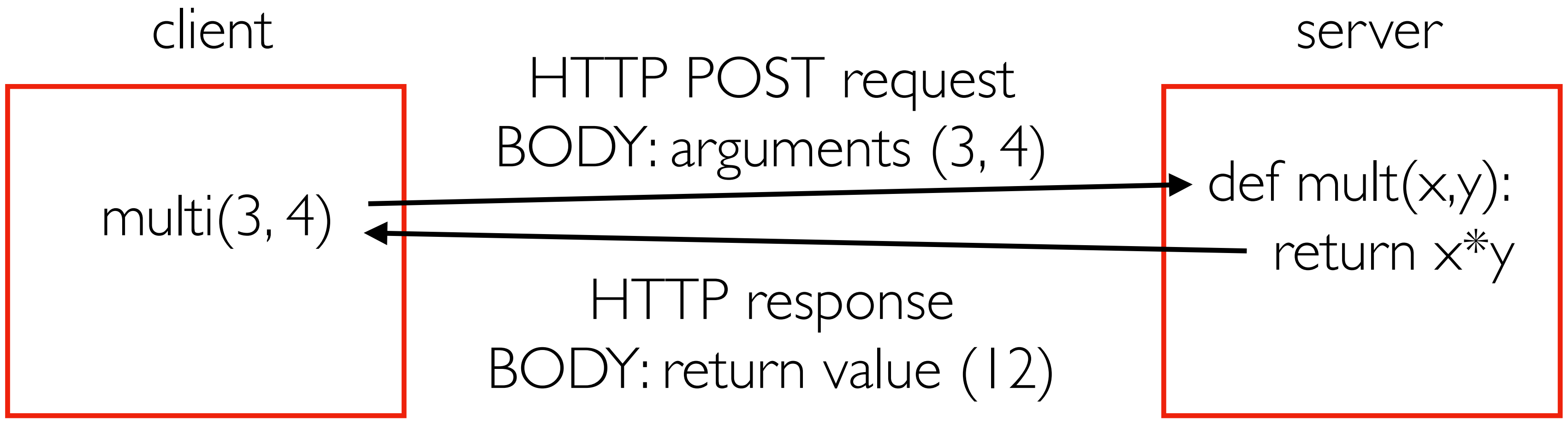
gRPC — Methods

HTTP — URL (domain/IP + port + resource)

TCP or UDP — IP addresses + ports

Internet Protocol — IP addresses

Ethernet or Wi-Fi — MAC addresses



client

multi(3, 4)

HTTP POST request
BODY: arguments (3, 4)

HTTP response
BODY: return value (12)

server

```
def mult(x,y):
    return x*y
```

Serialization/deserialization (Protobufs)

How do we represent arguments and return values as bytes in a request/response body?

Serialization: various types (ints, strs, lists, etc) to **bytes** ("wire format")
Deserialization: **bytes** to various types

Challenge 1: every language has different types and we want cross-languages calls

gRPC uses Google's Protocol Buffers provide a uniform type system across languages.

Challenge 2: different CPUs order bytes differently

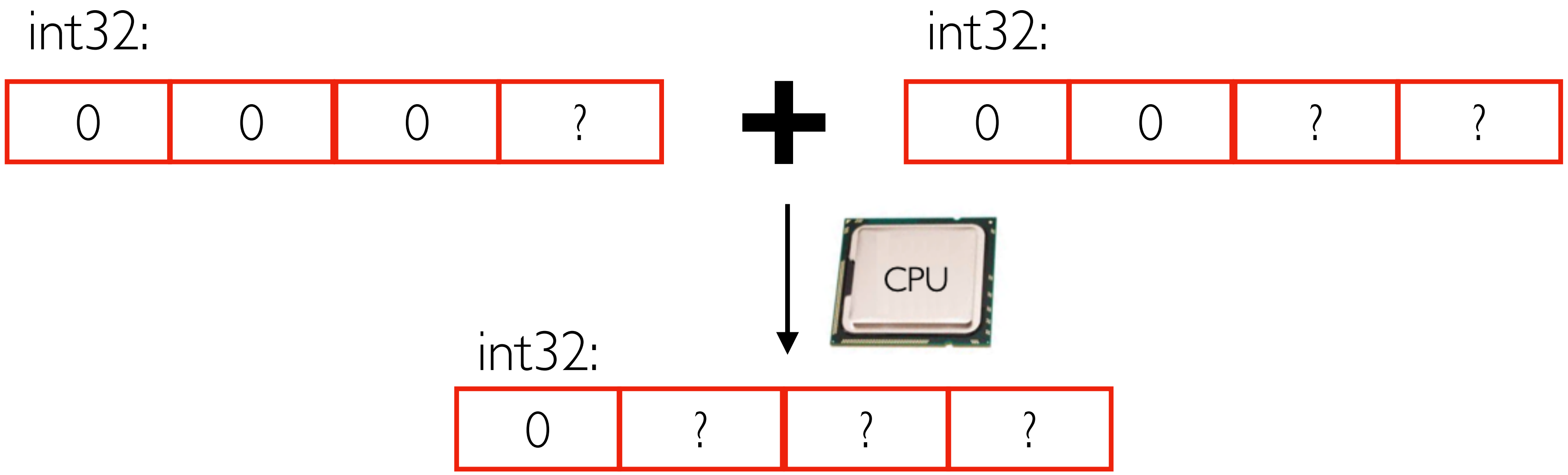
cpu A int32:	byte 1	byte 2	byte 3	byte 4
cpu B int32:	byte 4	byte 3	byte 2	byte 1

Equivalent with digit order: "twelve" is "12" by convention, but people could have chosen "21" to mean "twelve"

.proto	C++	Java	Python
double	double	double	float
float	float	float	float
int32	int32	int	int
int64	int64	long	int
uint32	uint32	int	int
uint64	uint64	long	int
sint32	int32	int	int
sint64	int64	long	int
bool	bool	boolean	bool
string	string	String	str
bytes	string	ByteString	bytes

https://protobuf.dev/programming-guides/proto/

Variable-Length Encoding



int32:

0	0	0	?

+

int32:

0	0	?	?

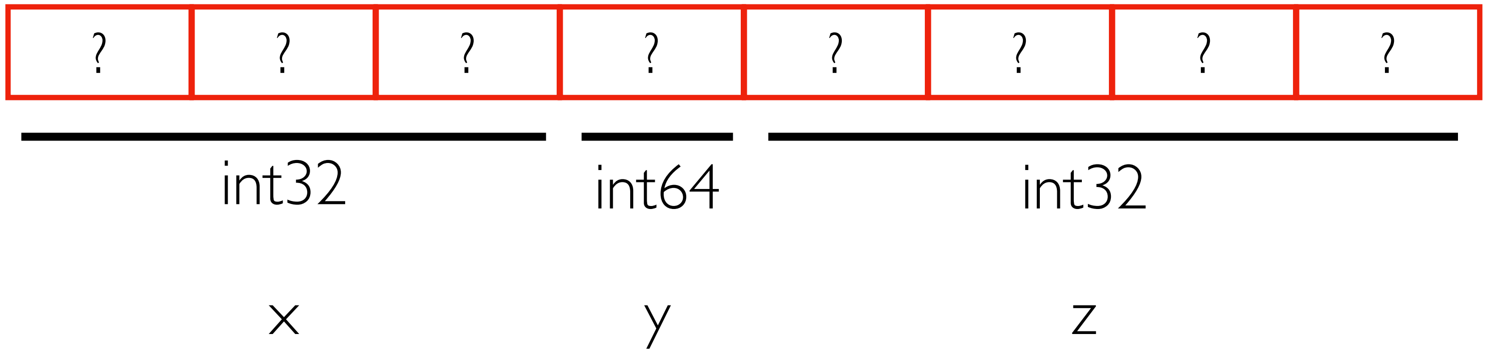
int32:

0	?	?	?

For **computational efficiency**, int32's use 4 bytes during computation. Also helps w/ offsets.

For **space efficiency**, smaller numbers in int32s could user fewer bytes (4 bytes is max). This reduces network traffic.

Example nums in a protobuf:



?	?	?	?	?	?	?	?

Demos...