[544] gRPC

Meenakshi Syamkumar

Learning Objectives

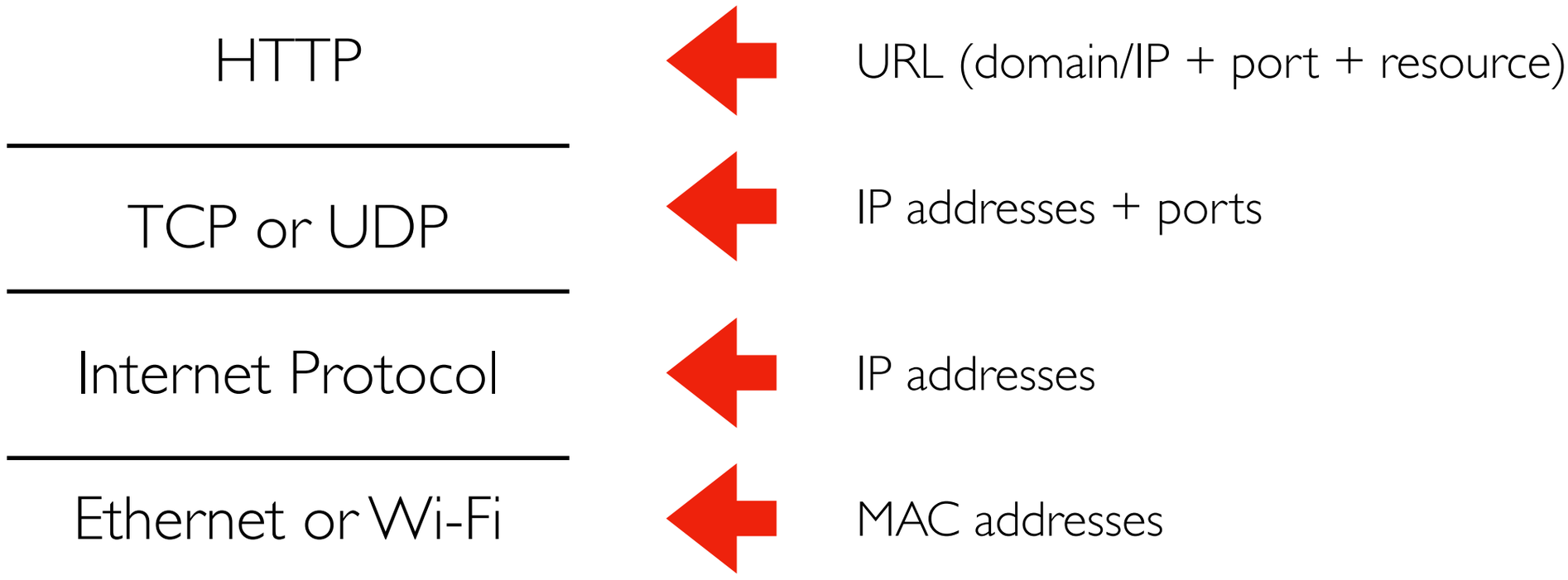
- describe the functionality that HTTP provides (beyond what TCP alone provides)
- call functions remotely via gRPC

Outline

HTTP

gRPC

HTTP (Hypertext Transfer Protocol)



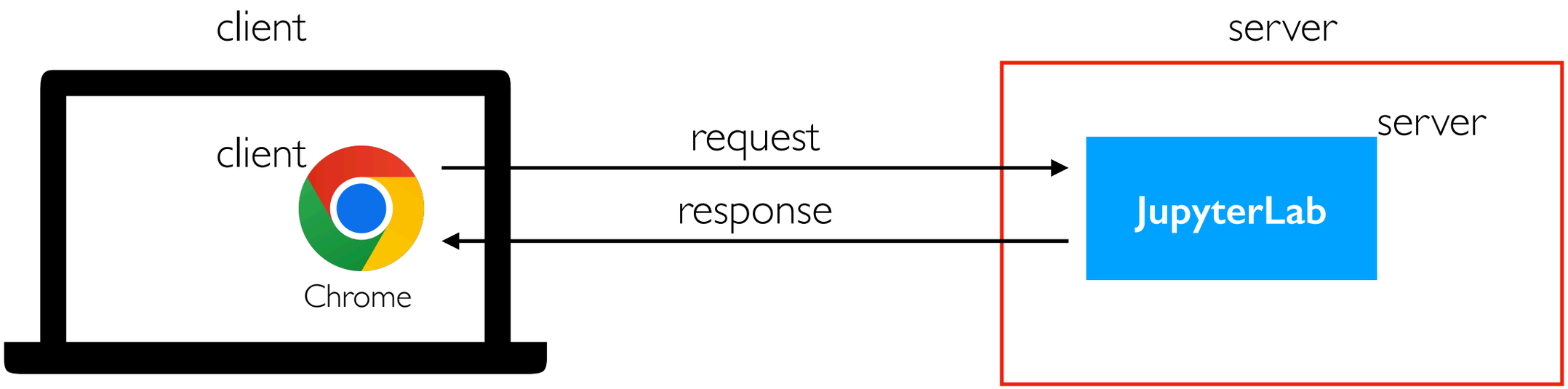
`https://ms.sites.cs.wisc.edu:443/cs544/f25/schedule.html`

domain name
(mapped to an IP)

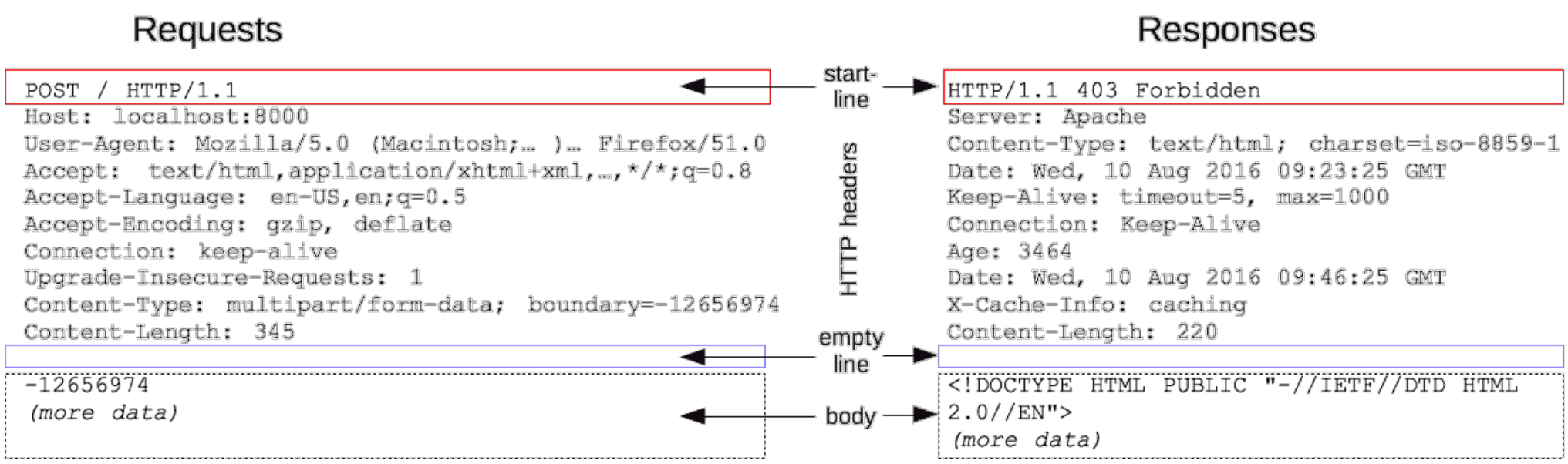
port
(443 is default for https)

resource

HTTP Messages Betwen Clients and Servers



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



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)

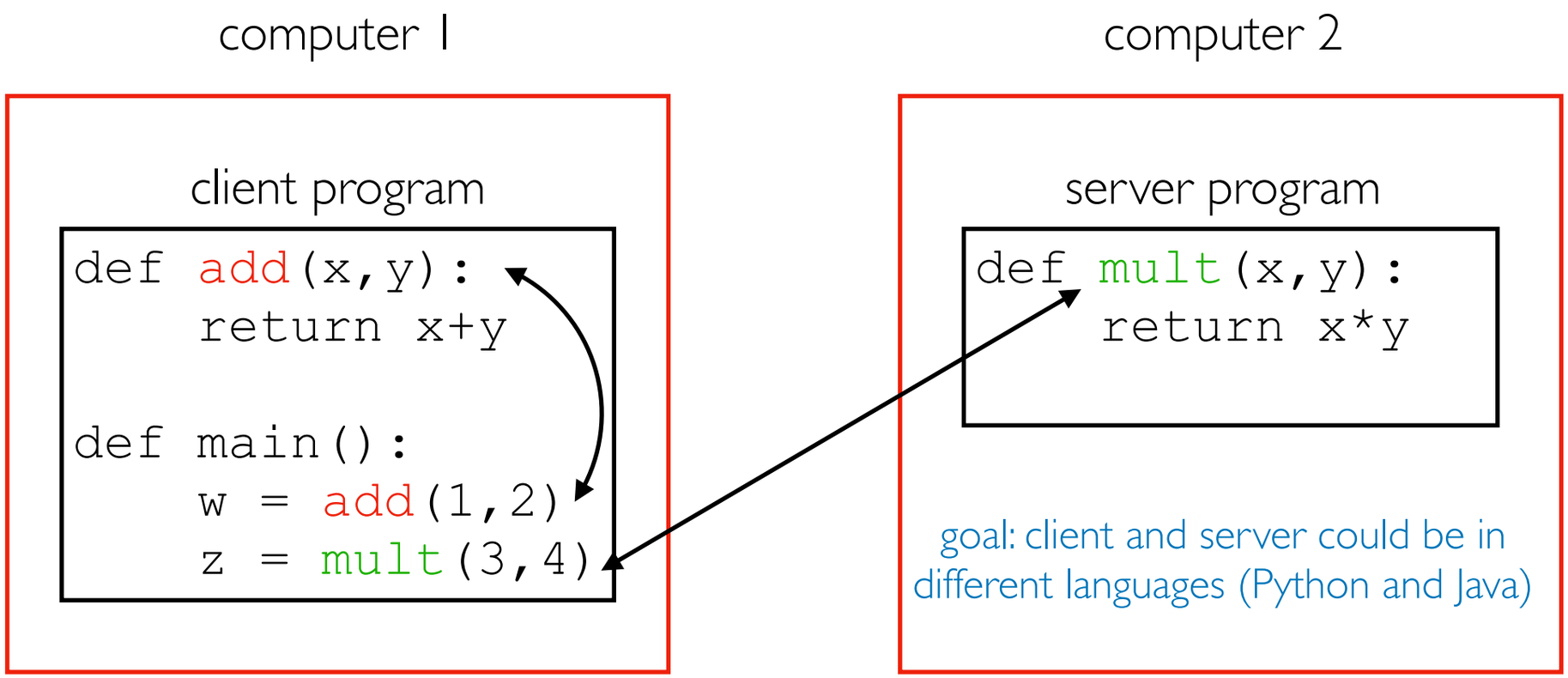
https://canvas.instructure.com/doc/api/conversations.html

Outline

HTTP

gRPC

Remote Procedure Calls (RPCs)



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

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
```

client

...

server

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

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

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

client

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

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

request message

response message

server

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

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

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

computer 1

computer 2

Serialization/Deserialization

client

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

serialize

deserialize

deserialize

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

server

serialize

```
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

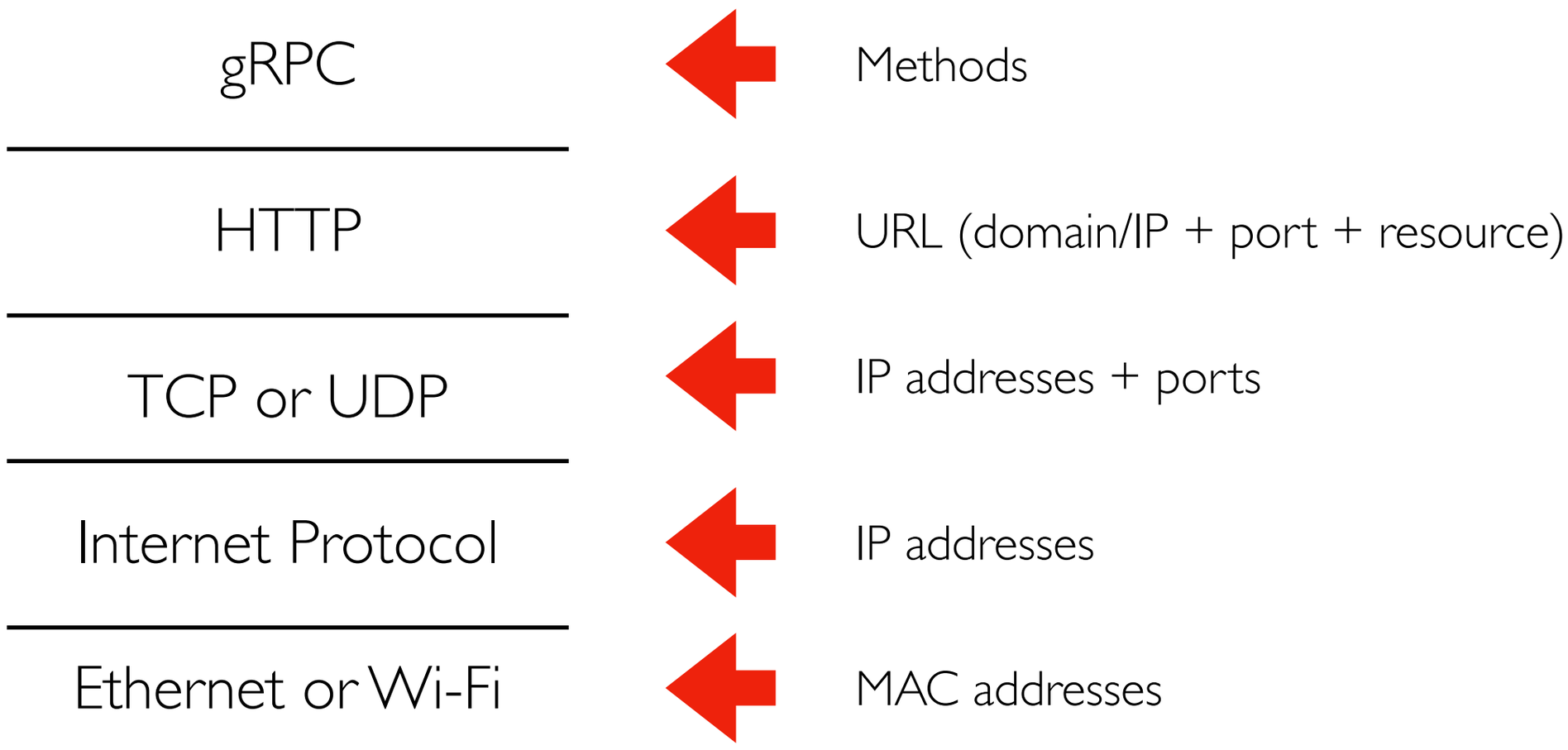
```
args somehow encoded as bytes:
b'{"key": "A"
   "amt": 5}'
```

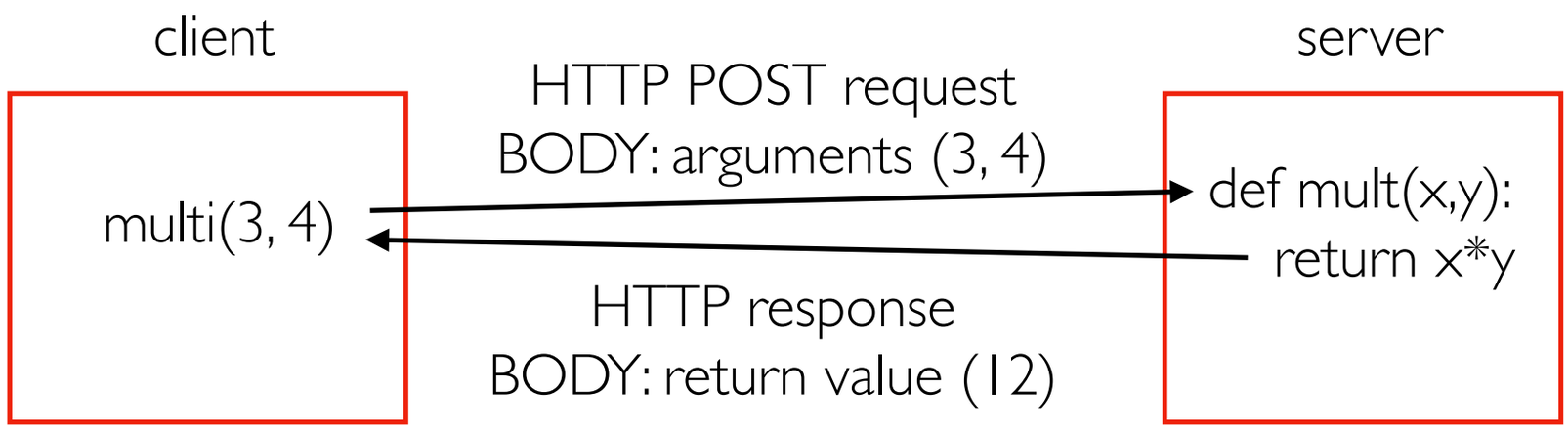
response message

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

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

gRPC builds on HTTP





Serialization/deserialization (Protobufs)

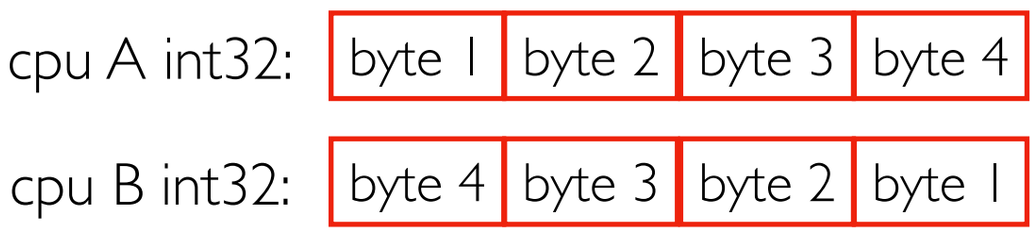
How do we represent arguments and return values as bytes in a request/response body?

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

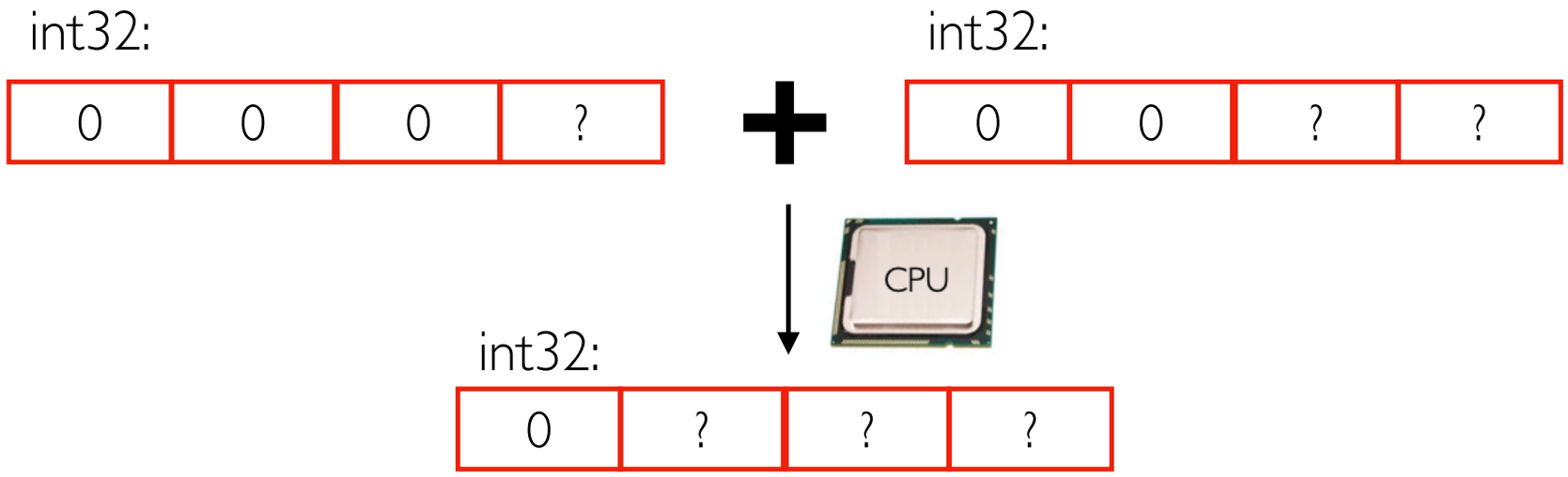


Equivalent with digit order: "twelve" is "12" by convention, but people could have chosen "21" to mean "twelve"

.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/

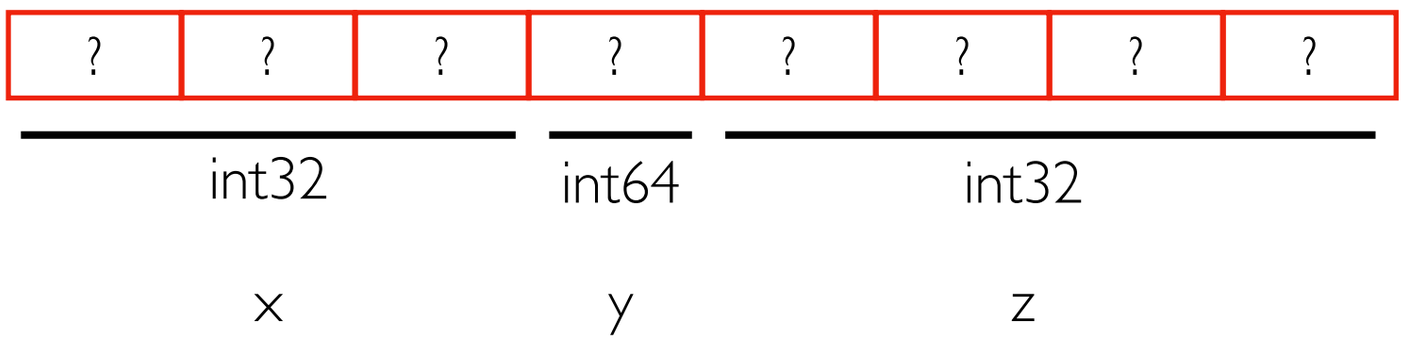
Variable-Length Encoding



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...