Scalability and Availability

Ryan Eberhardt and Julio Ballista
May 20, 2021

Logistics

- Week 8 exercises are due on Tuesday
- Project 2 coming out today, due on the last day of class
  - This is the last assignment! You’re in the home stretch!
- Let us know how we can help!

Reflections

- *Really nice! I find it interesting how you can almost use Rust's error messages as guardrails to bump around and hopefully lead to a correct solution. Obviously this doesn't work in all cases (deadlock), but it's pretty effective :)*
- *I feel like while I understand some of the basic syntax/writing of rust, I don't think I know enough to start a rust project myself from scratch*
  - You might surprise yourself! `cargo new projectname` will create a new project, and then you can add dependencies in Cargo.toml. That's about it!
- *Project 1 was fun! I was very disappointed at how hacky debuggers are on the inside.*
  - Welcome to systems :)
- *On the subject of multithreading, I can see how rust is helpful, but at least for me, deadlock from bad logic is usually what got me as opposed to data races, but I assume that ensuring that no deadlock occurs is something like the halting problem.*
  - Data races are extremely prevalent and hard to avoid in complicated codebases, so this was a major design goal for Rust
  - It's also possible to build languages that give guarantees about deadlock. Extremely relevant for writing smart contracts (e.g. Ethereum), see Oxide as an example

This week

- Moving up a level of abstraction: Discussing safety in the context of systems design
- Today: How do you keep big systems running?
- Tuesday: How do you keep information secure from attackers?
- This could be an entire class. We will just skim the surface and talk about the parts we feel are most important to understand
  - How do you keep big systems running? Take CS 144/244, 245, 244B
  - How do you keep information secure? Take CS 155, 356, 255

Networking in a Nutshell

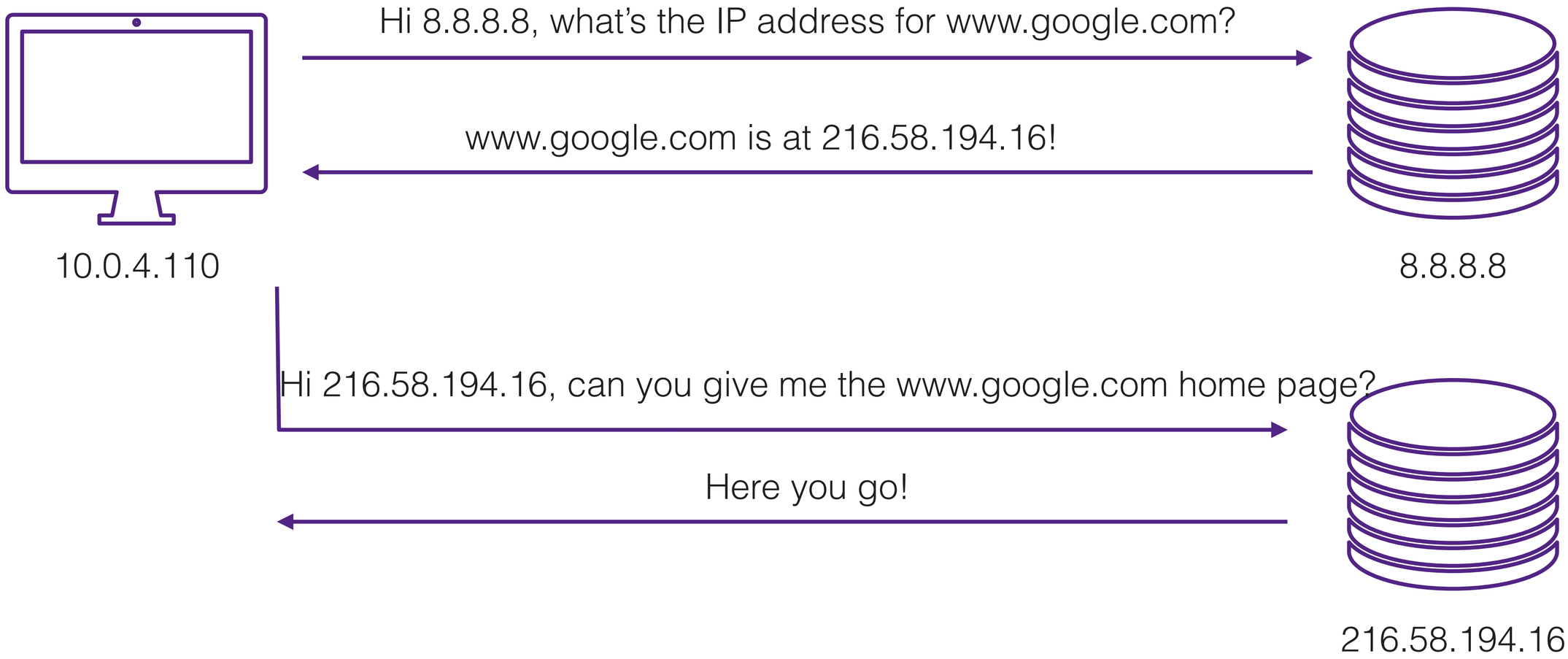
IP addresses

- Every computer on a network has an “IP address” uniquely identifying it on the network
  - An IPv4 address is 4 bytes. Usually written as 4 numbers, 0-255, separated by periods (e.g 192.168.1.230)
- If you want to talk to a computer, you need to know its IP address
- How do you find the IP address? (Too hard to remember!)
  - Your computer is configured with the address of a *DNS server* (can be hardcoded)
  - When you want to reach “www.google.com,” ask the DNS server for the IP address
  - IP address of www.google.com:

    🍌 `dig +noall +answer www.google.com`

    ```
    www.google.com.          204     IN      A       216.58.194.16
    ```

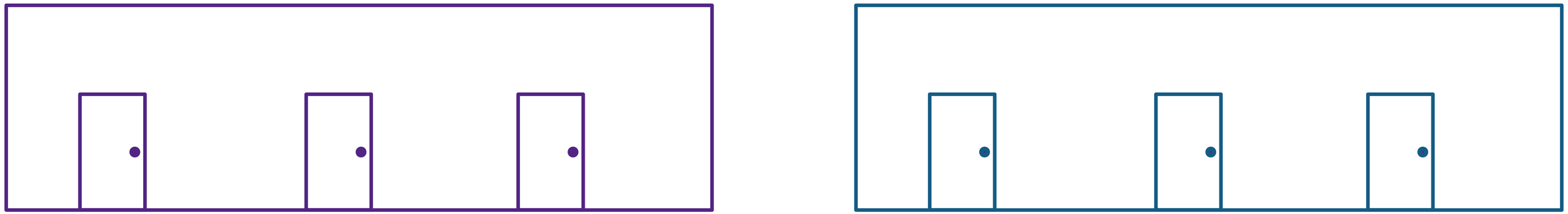
DNS resolution



Understanding port numbers

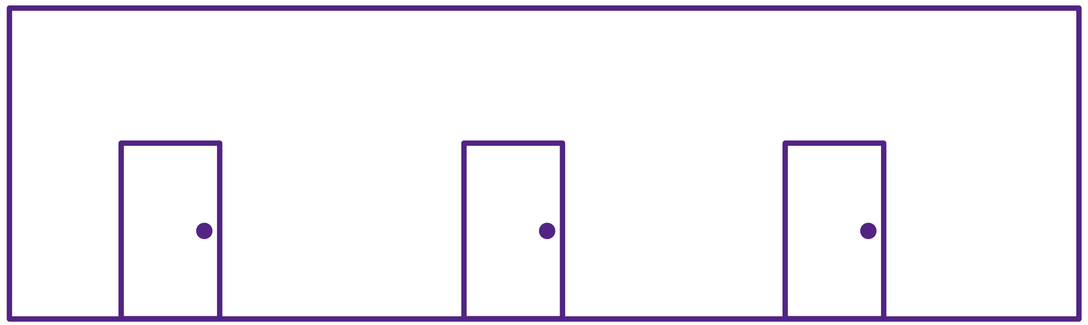
“Host” (computer) = apartment complex

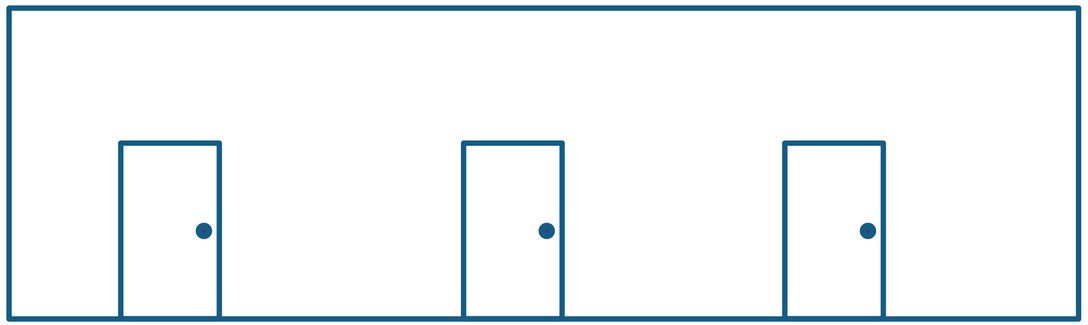
“Host” (computer) = apartment complex



“Host” (computer) = apartment complex

“IP address” = apartment complex address

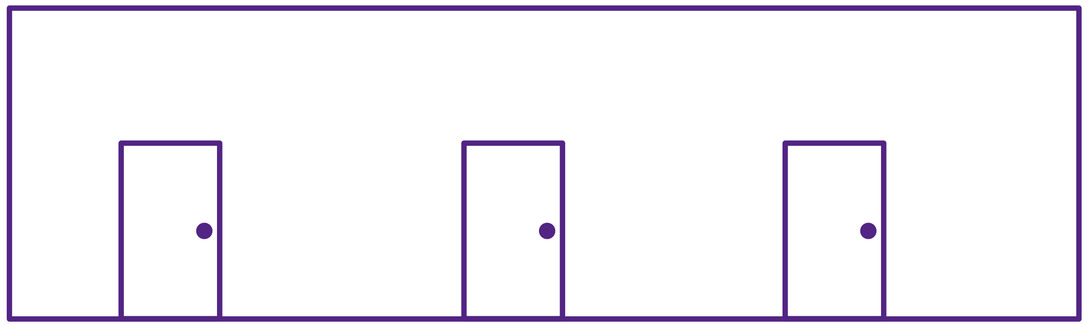




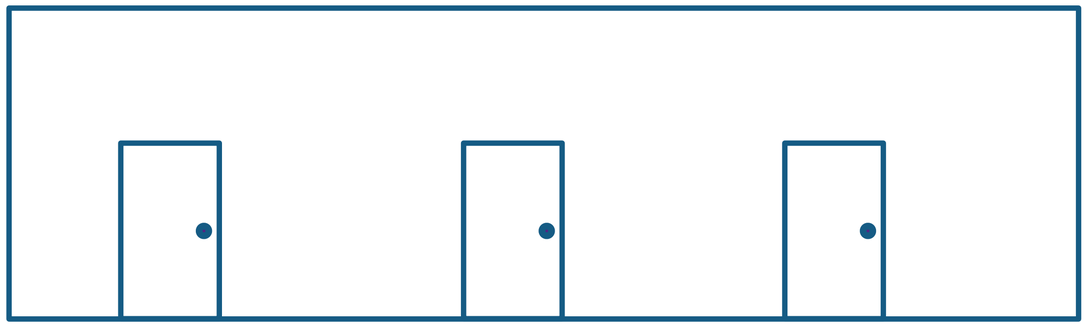
“Host” (computer) = apartment complex

“IP address” = apartment complex address

171.67.215.200



10.0.4.128

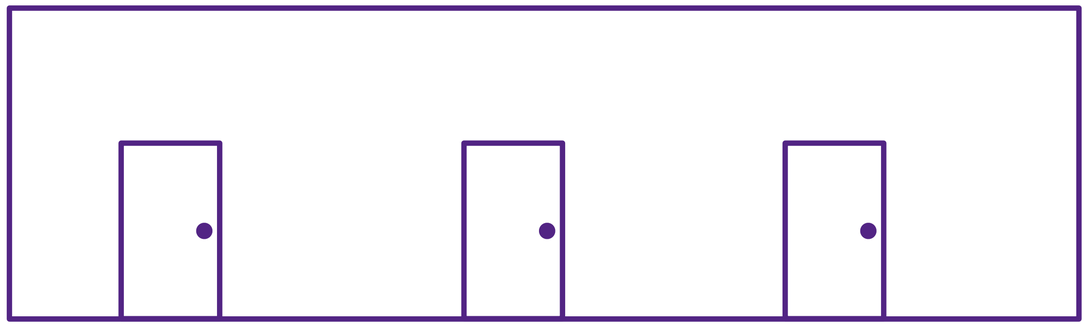


“Host” (computer) = apartment complex

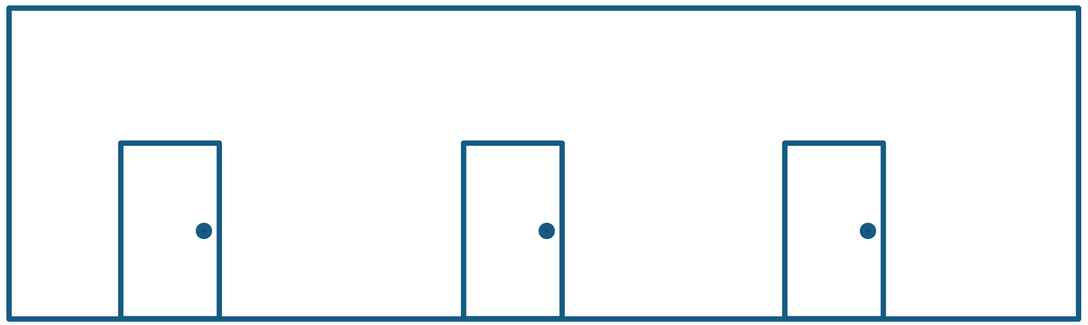
“IP address” = apartment complex address

“Port number” = apartment number

171.67.215.200



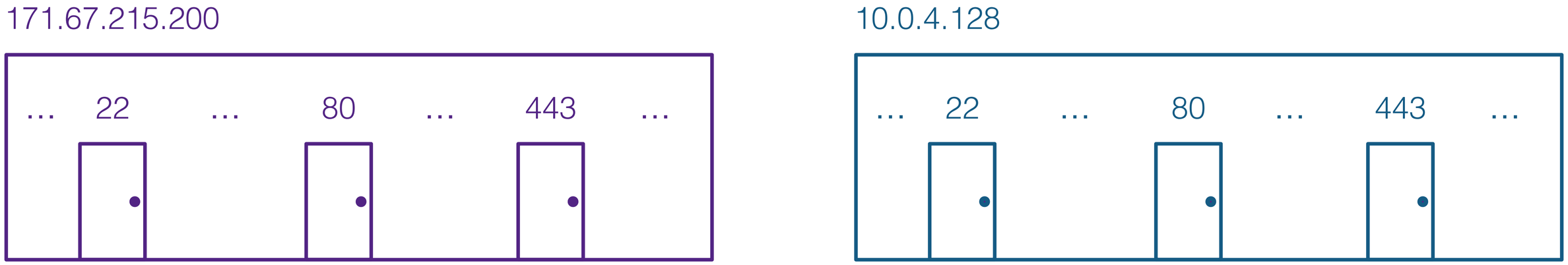
10.0.4.128



“Host” (computer) = apartment complex

“IP address” = apartment complex address

“Port number” = apartment number



Want to go to http://web.stanford.edu?

Use DNS to find web.stanford.edu's IP address: 171.67.215.200

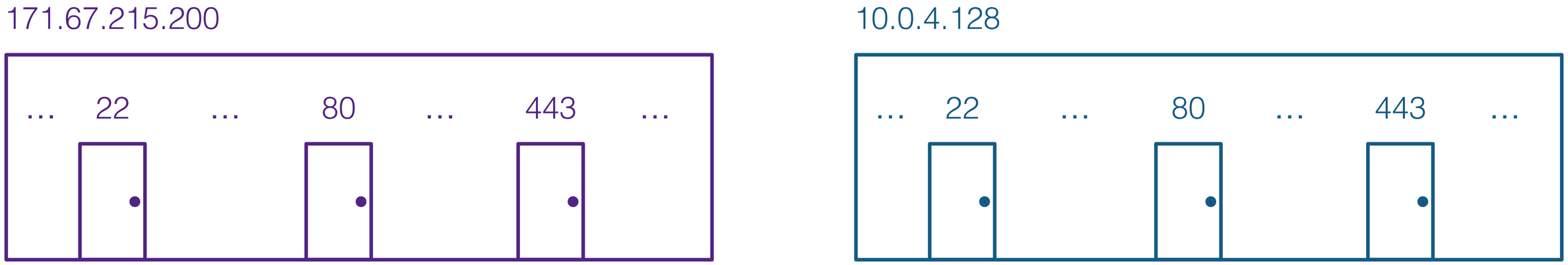
Go to that apartment complex

Knock on the apartment that runs the HTTP service (port 80)

“Host” (computer) = apartment complex

“IP address” = apartment complex address

“Port number” = apartment number



Want to SSH into myth.stanford.edu?

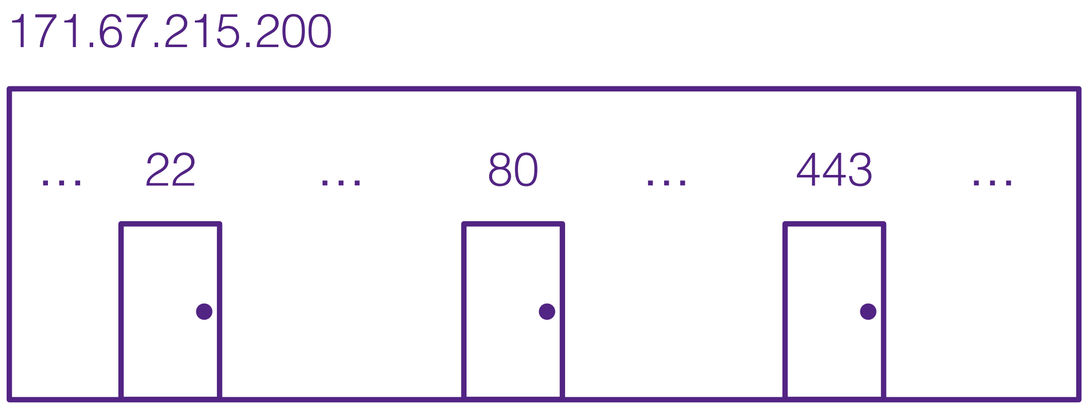
Use DNS to find myth.stanford.edu's IP address: 171.64.15.29

Go to that apartment complex

Knock on the apartment that runs the SSH service (port 22)

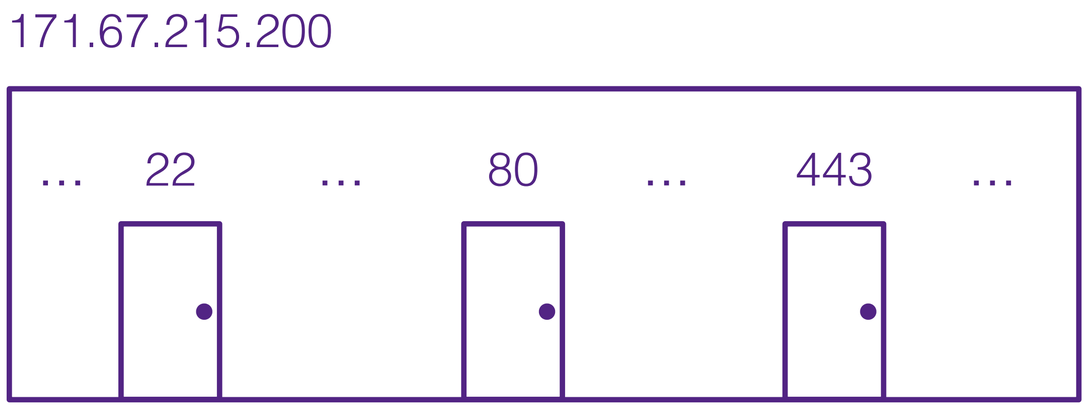
Starting a server

Apartment complex = host

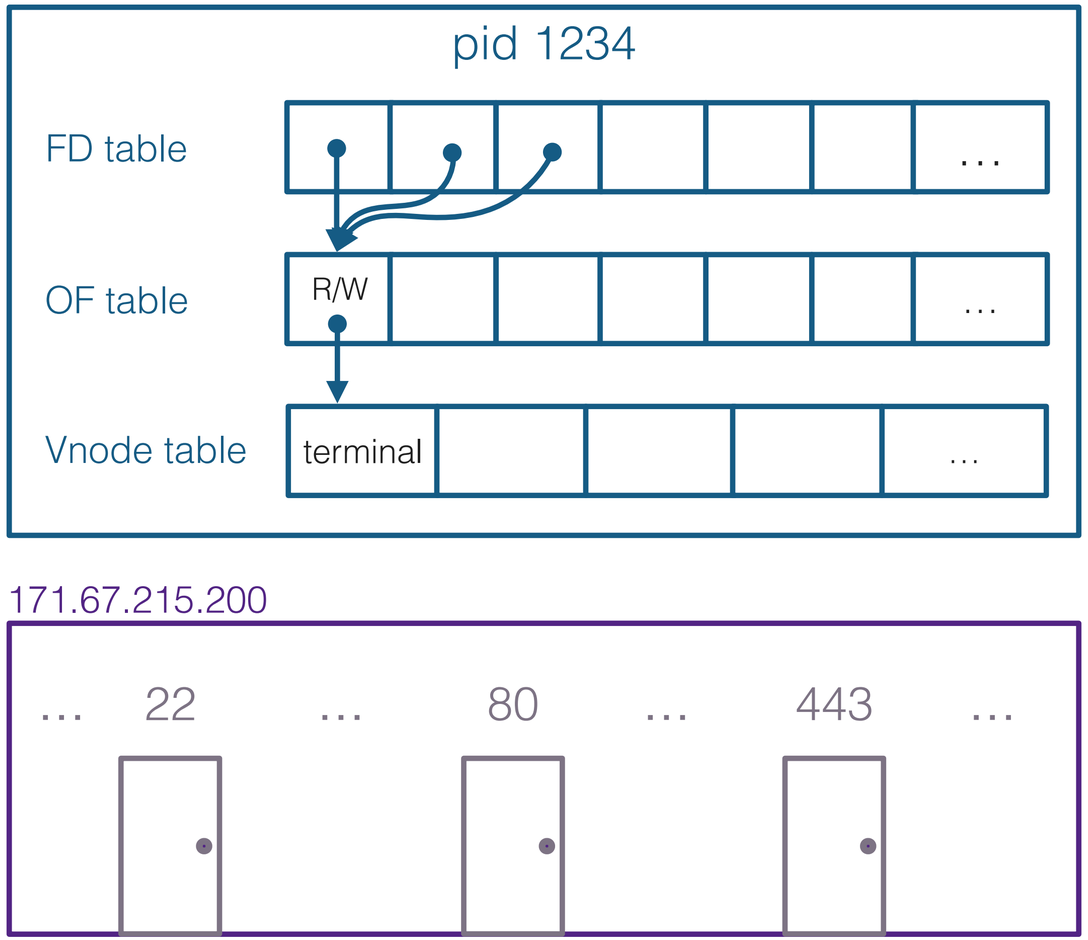


Apartment complex = host

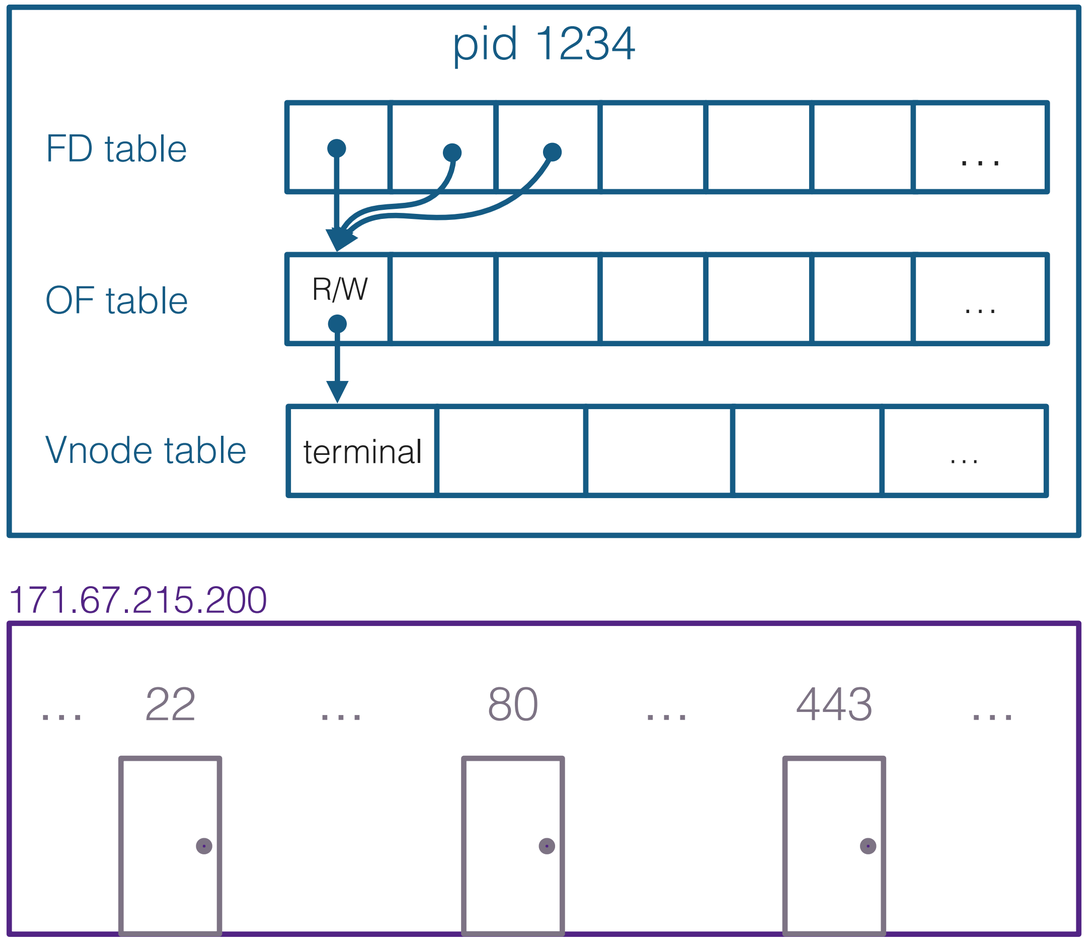
Each host will have some processes running on it



Each host will have some processes running on it

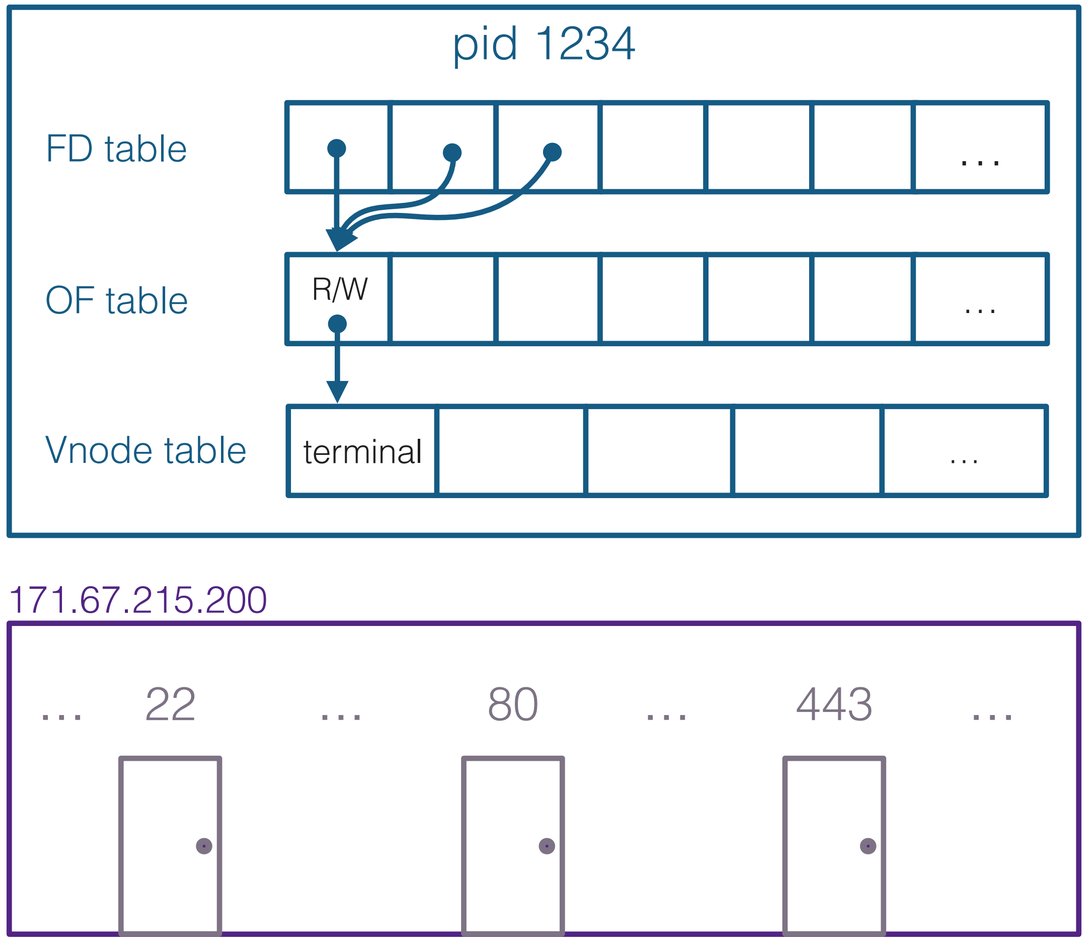


“Binding” to a port:



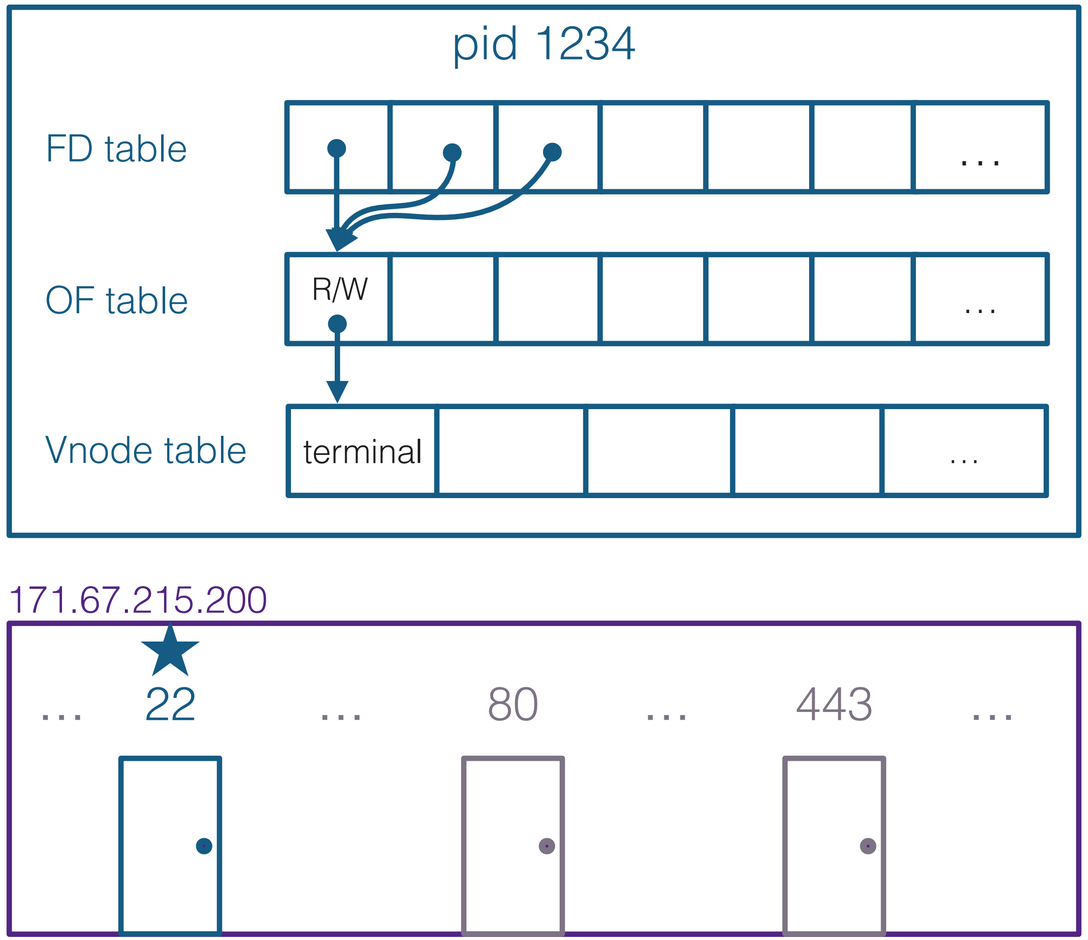
“Binding” to a port:

Process “sets up shop” in an apartment. (Only one process per apartment)



“Binding” to a port:

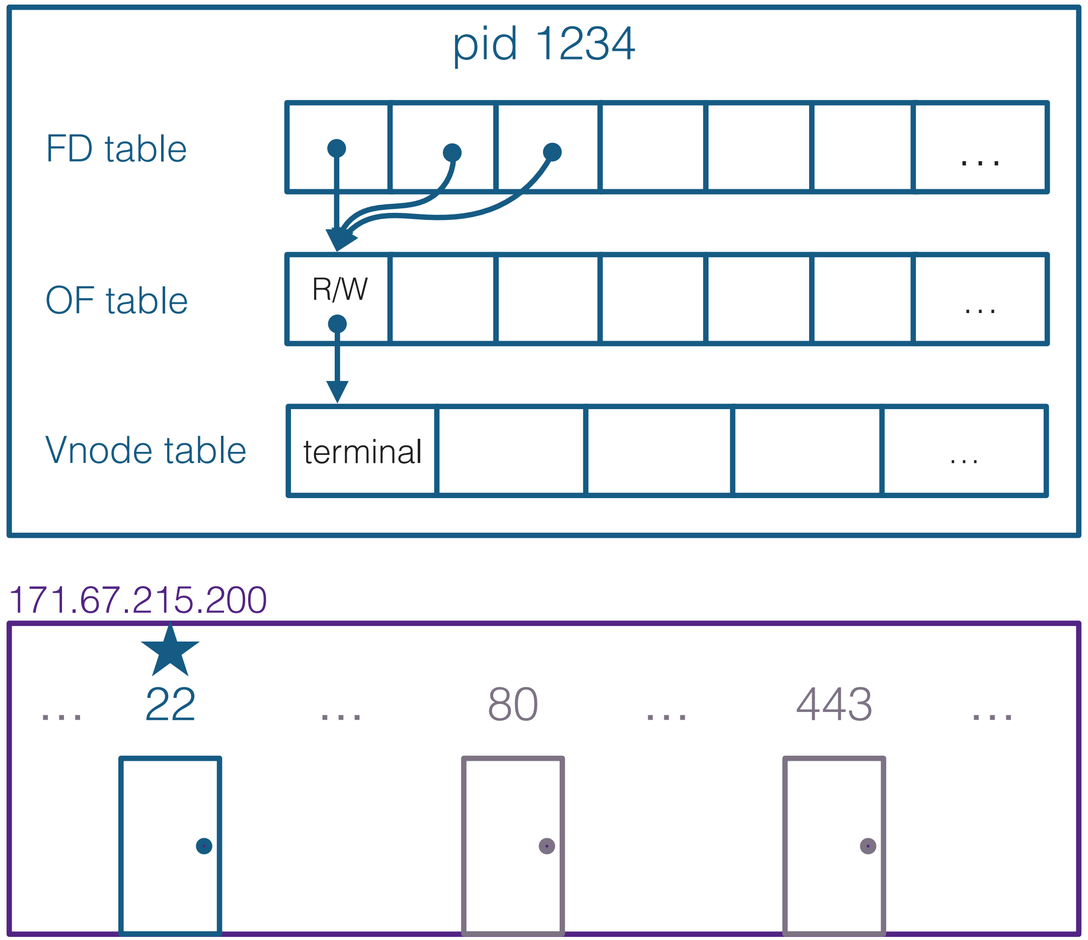
Process “sets up shop” in an apartment. (Only one process per apartment)



“Binding” to a port:

Process “sets up shop” in an apartment. (Only one process per apartment)

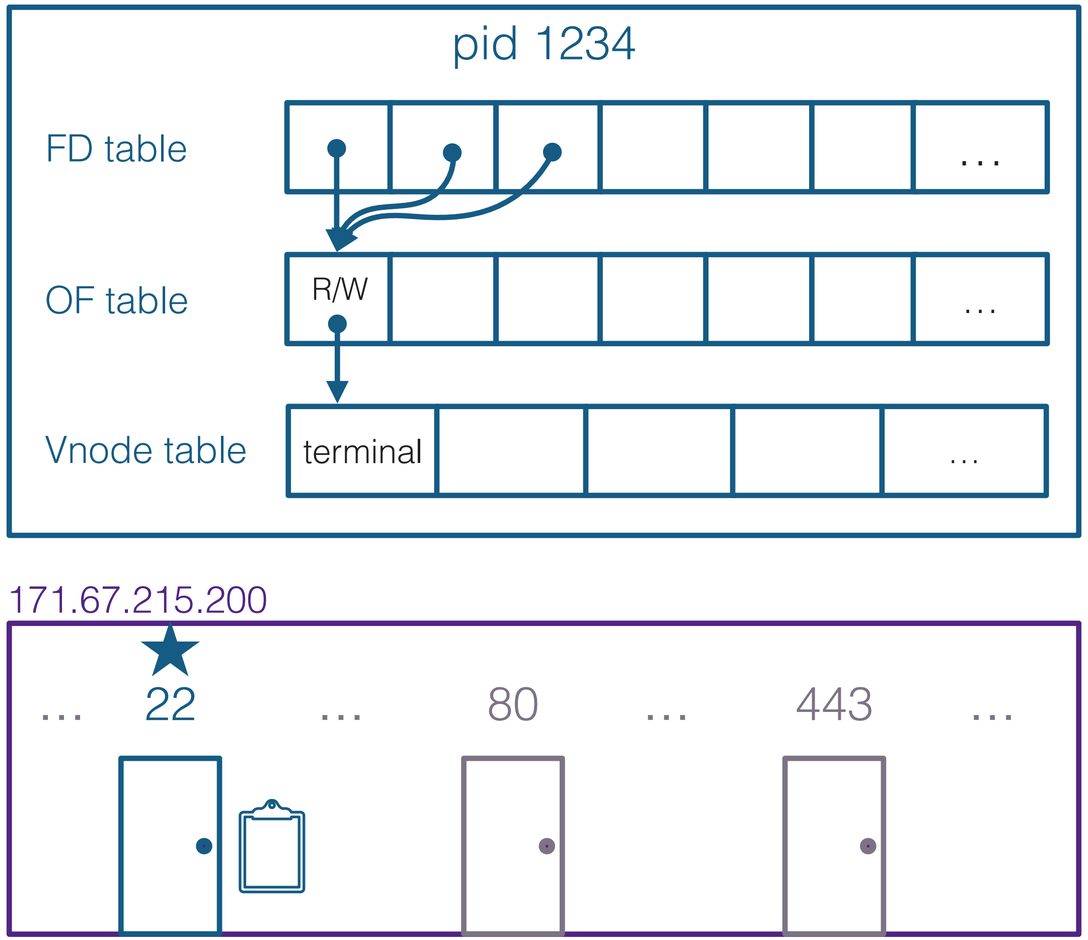
Process installs a “waiting list” outside the apartment



“Binding” to a port:

Process “sets up shop” in an apartment. (Only one process per apartment)

Process installs a “waiting list” outside the apartment

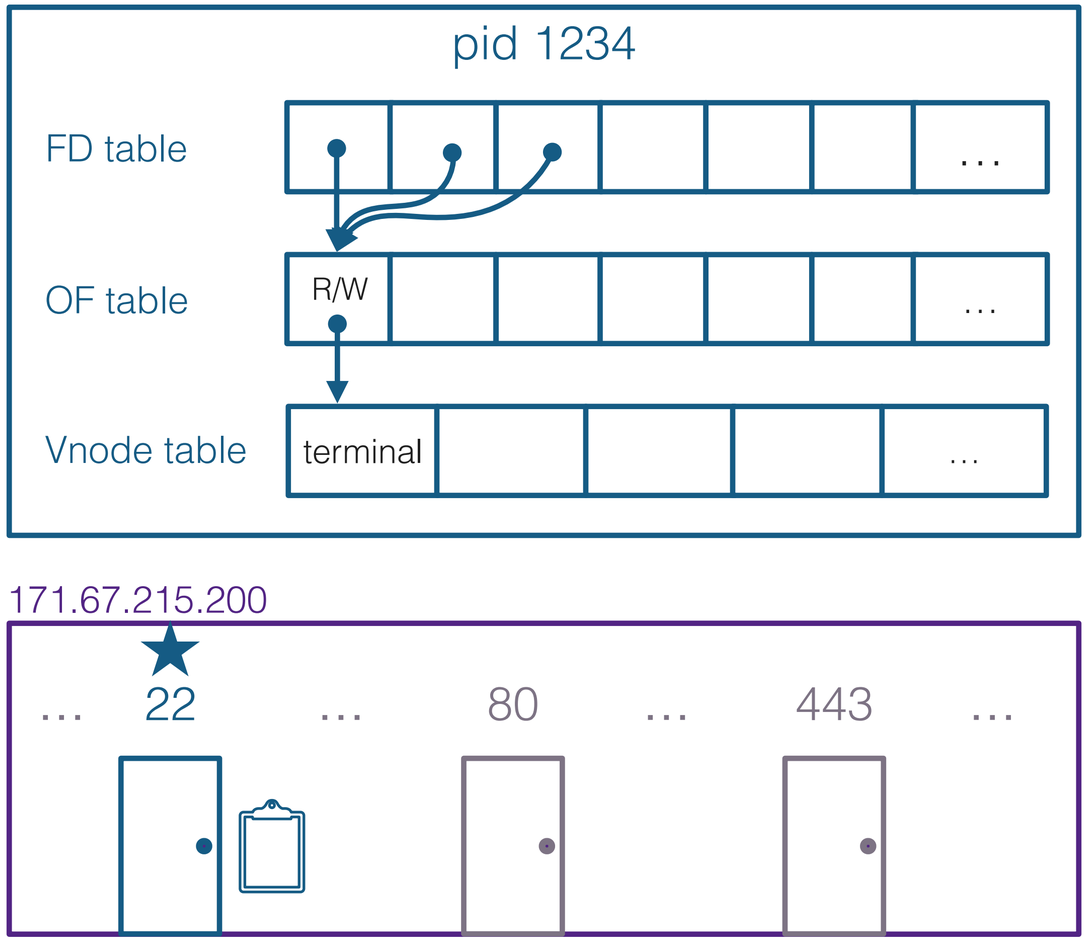


“Binding” to a port:

Process “sets up shop” in an apartment. (Only one process per apartment)

Process installs a “waiting list” outside the apartment

Waiting list is attached to a file descriptor, so the process can see when someone arrives

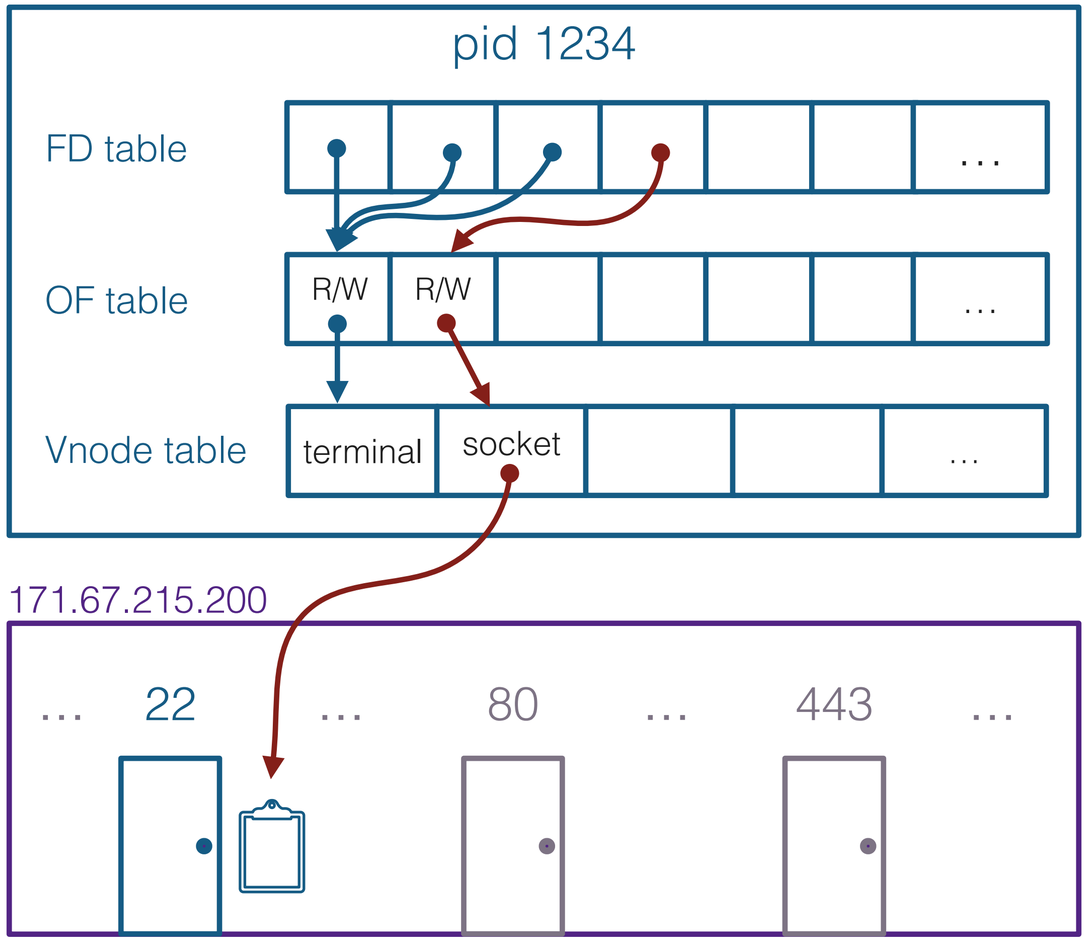


“Binding” to a port:

Process “sets up shop” in an apartment. (Only one process per apartment)

Process installs a “waiting list” outside the apartment

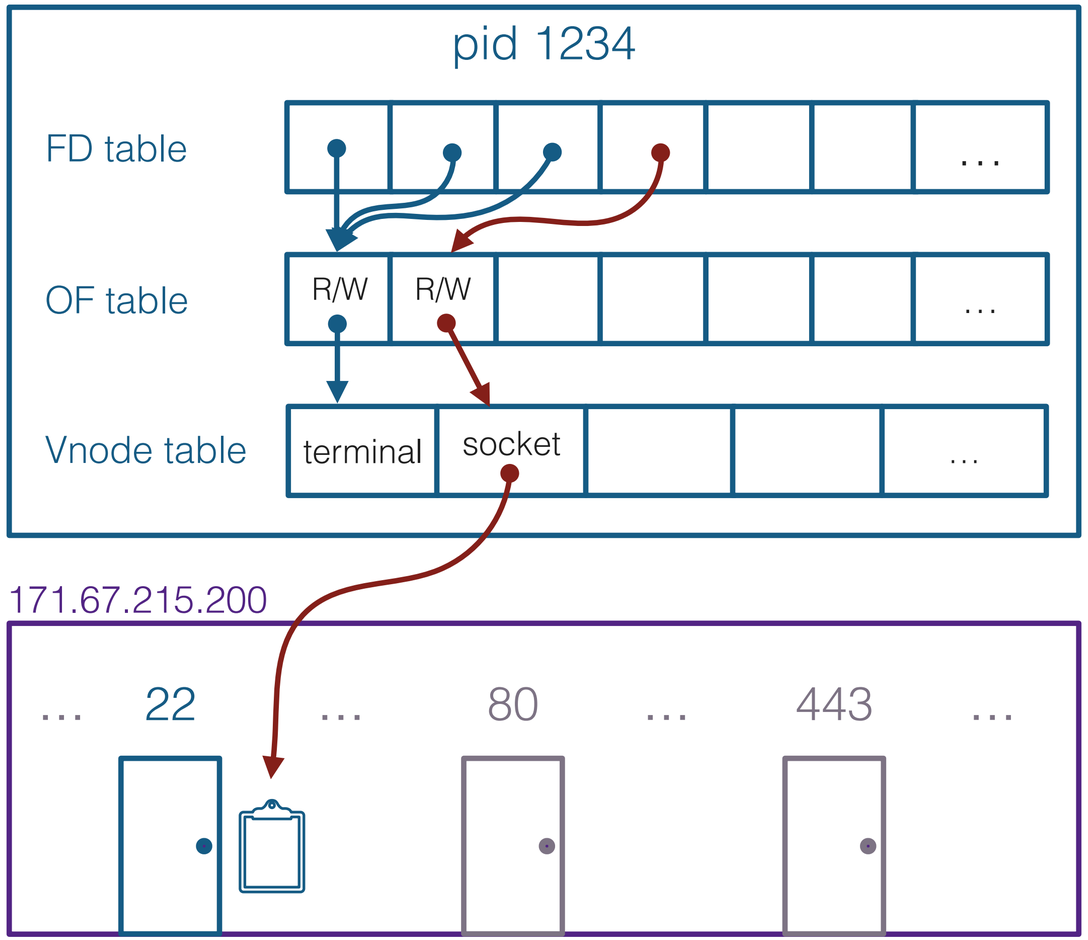
Waiting list is attached to a file descriptor, so the process can see when someone arrives



“Binding” to a port:

Other processes can bind to other ports

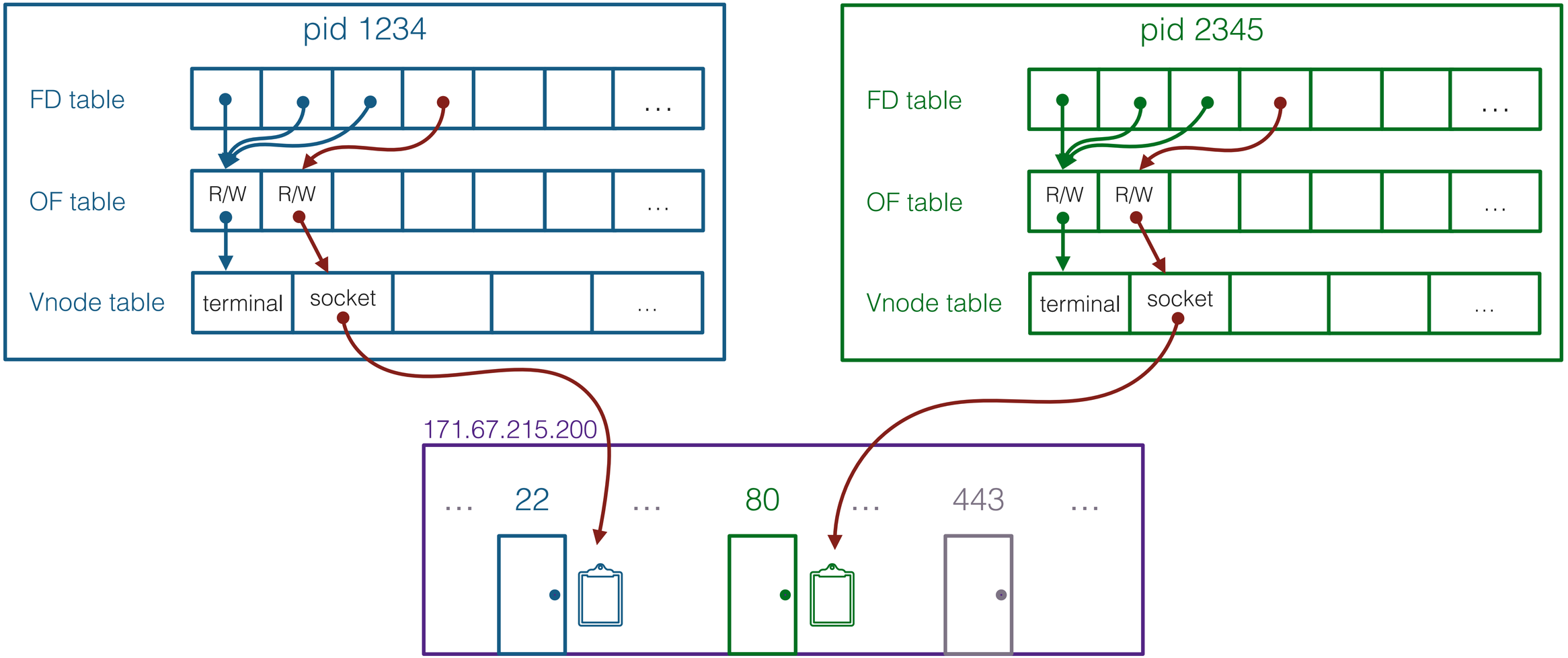
(no two processes can bind to the same port — one application per apartment!)



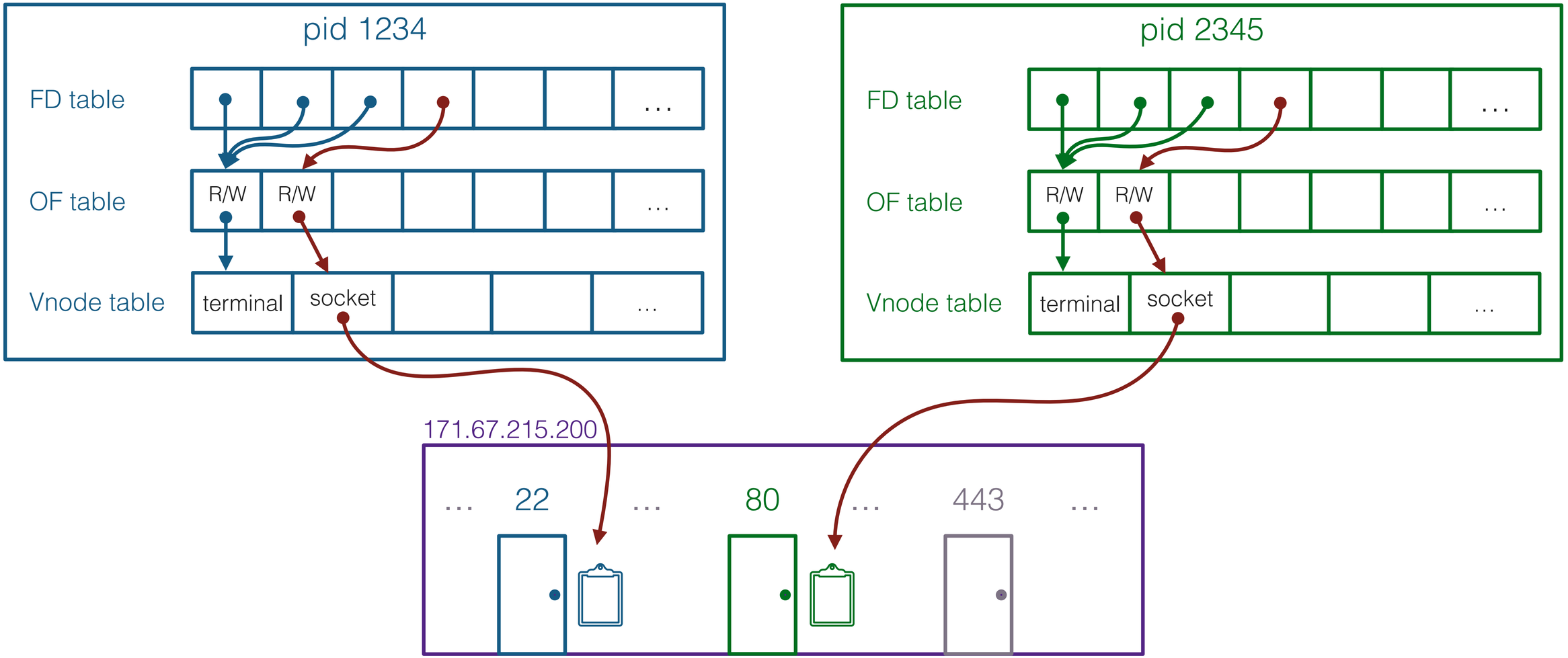
“Binding” to a port:

Other processes can bind to other ports

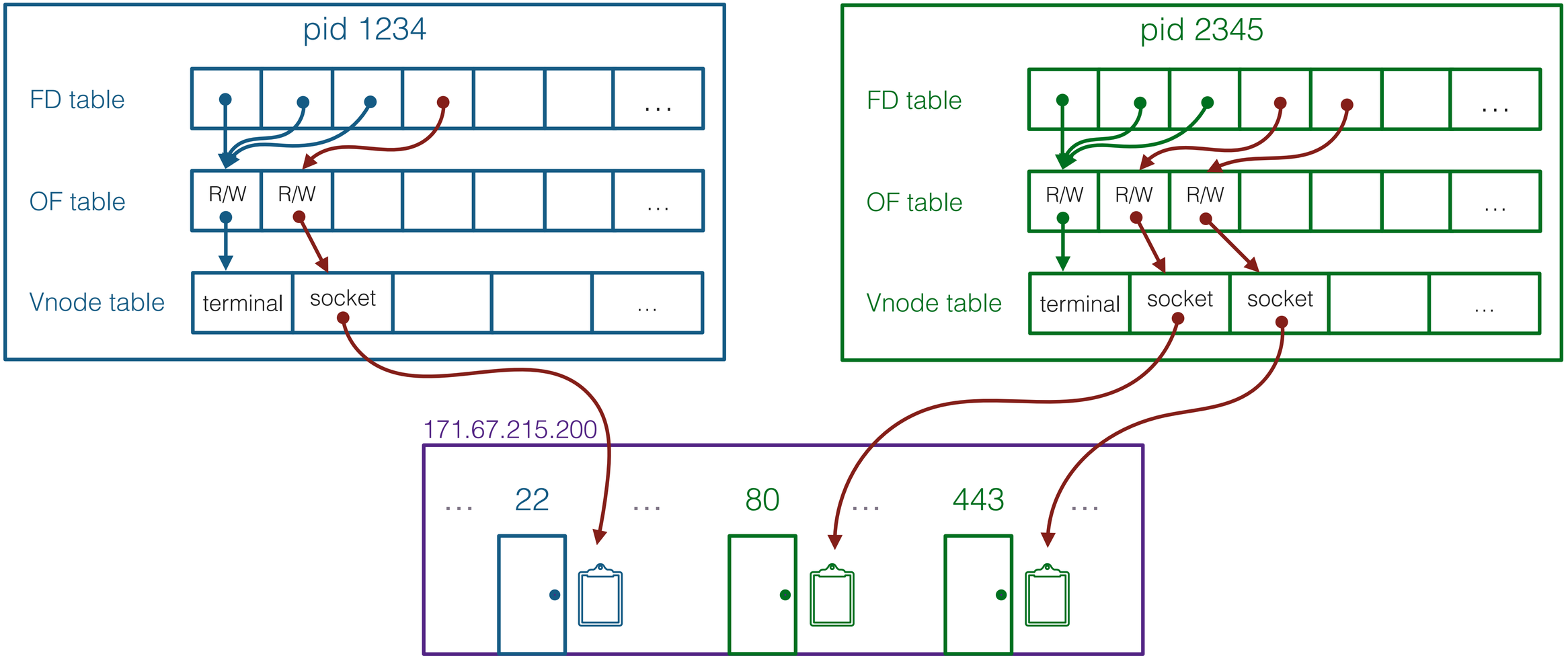
(no two processes can bind to the same port — one application per apartment!)



“Binding” to a port:
A process can bind to multiple ports, if it desires

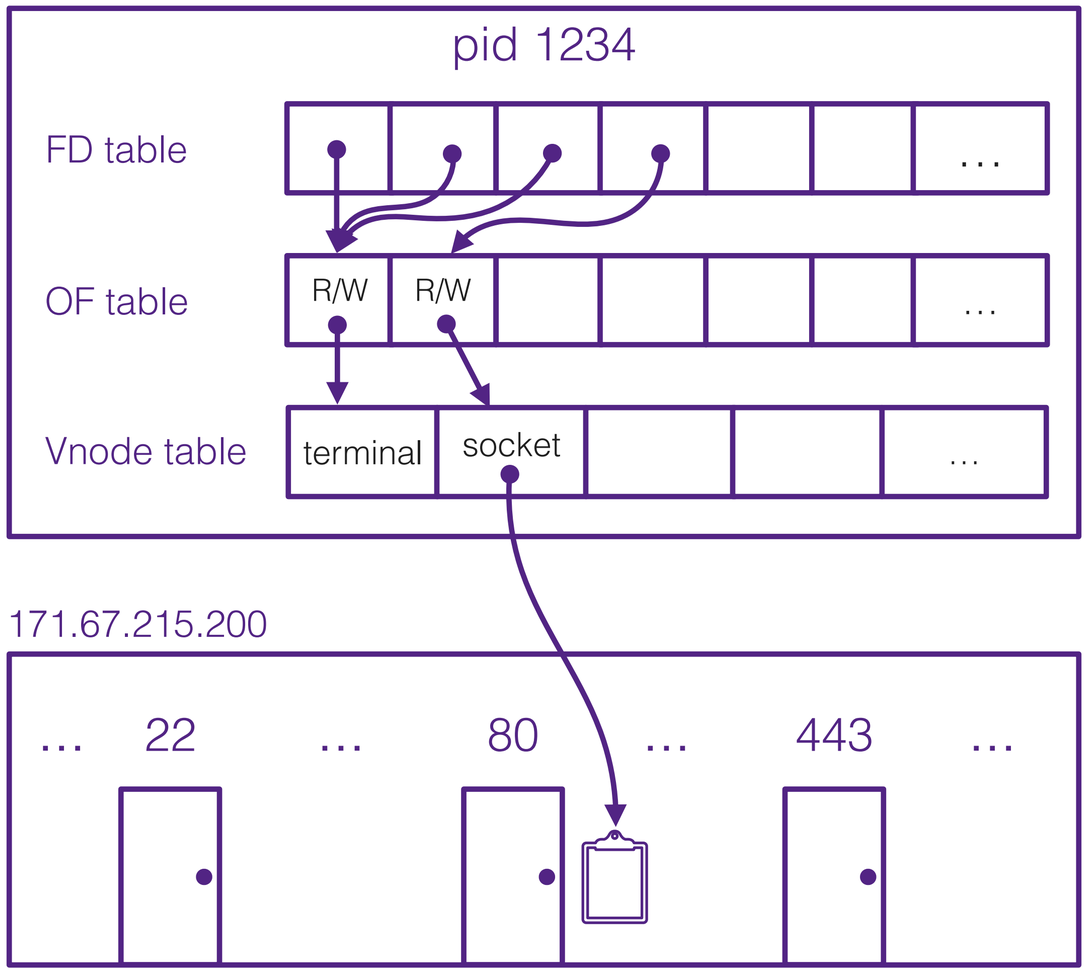


“Binding” to a port:
A process can bind to multiple ports, if it desires

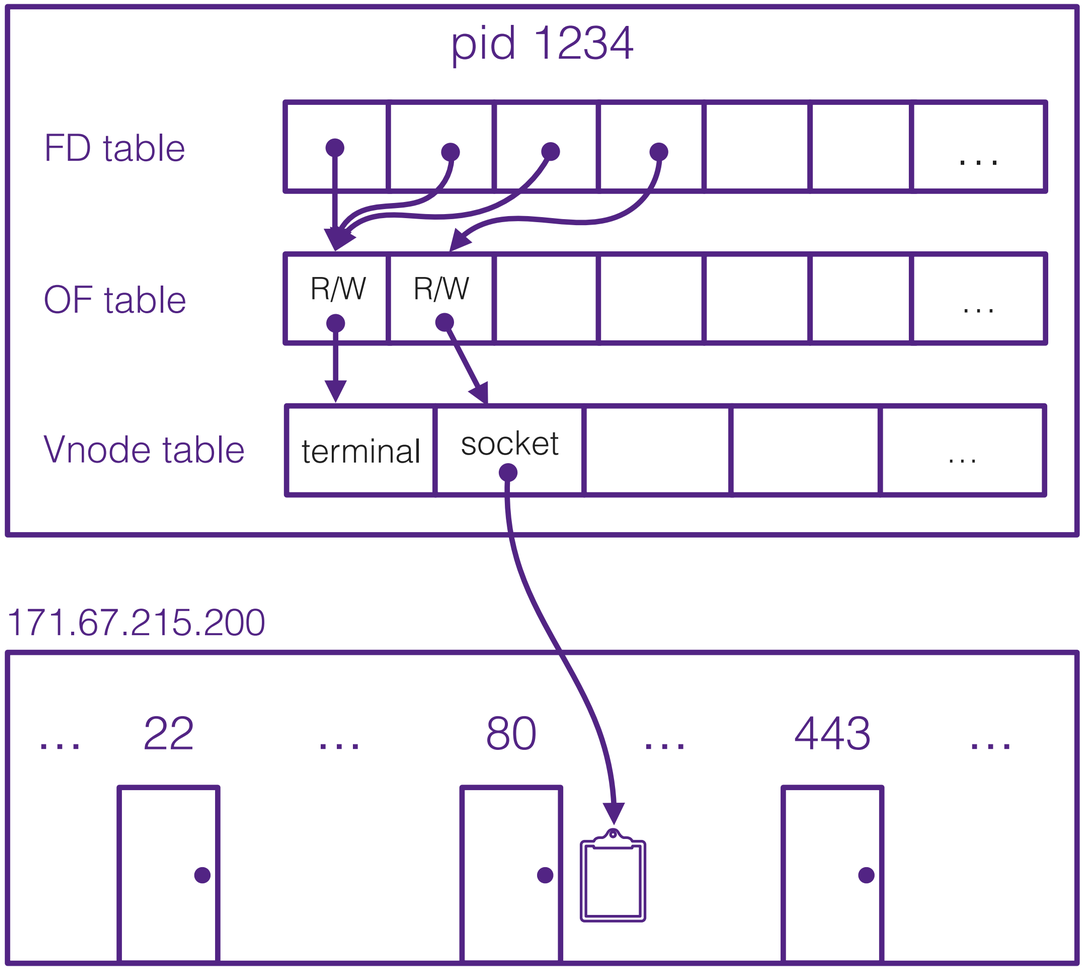


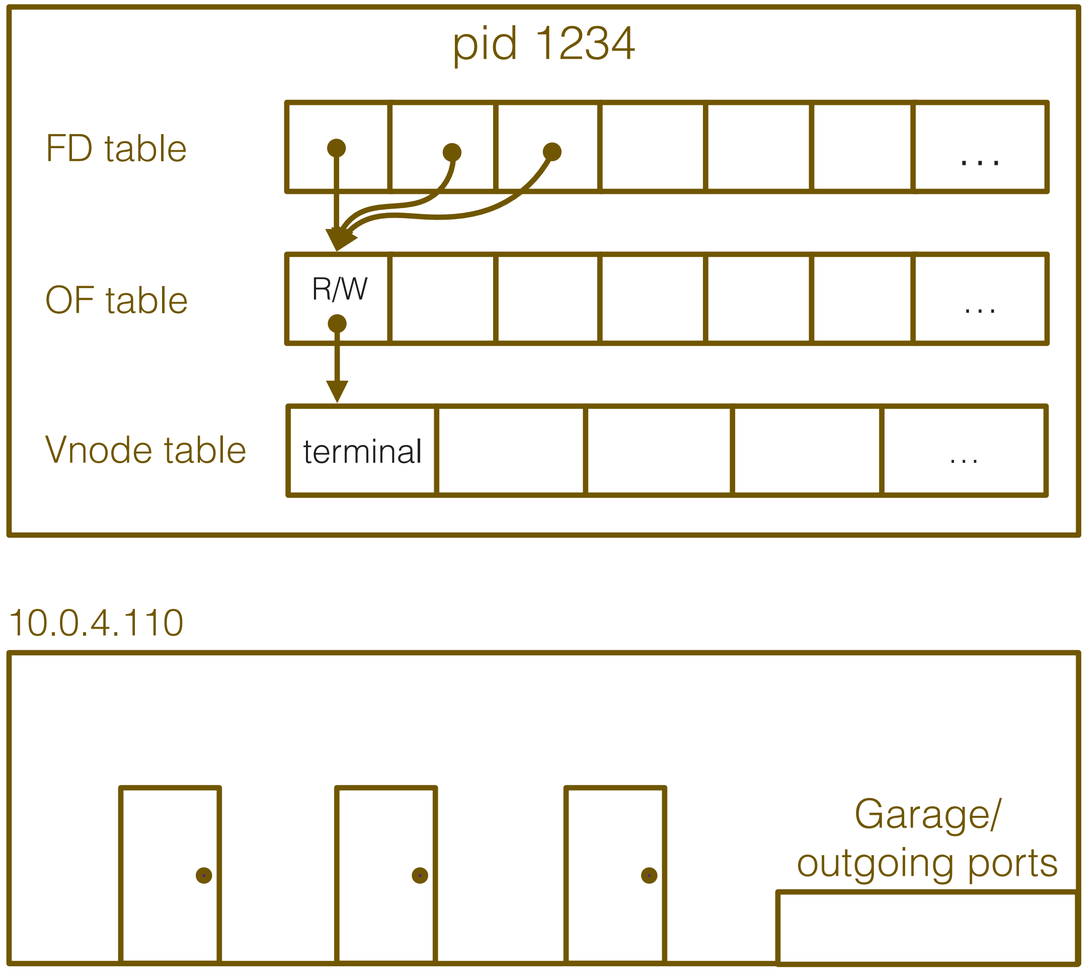
Connecting a client

Say we have a server bound on 171.67.215.200:80

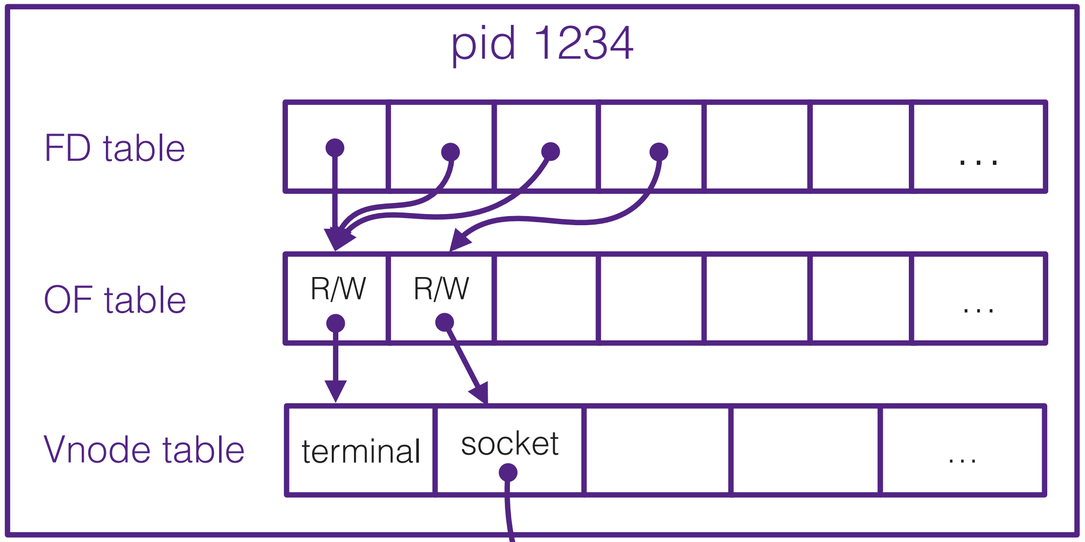


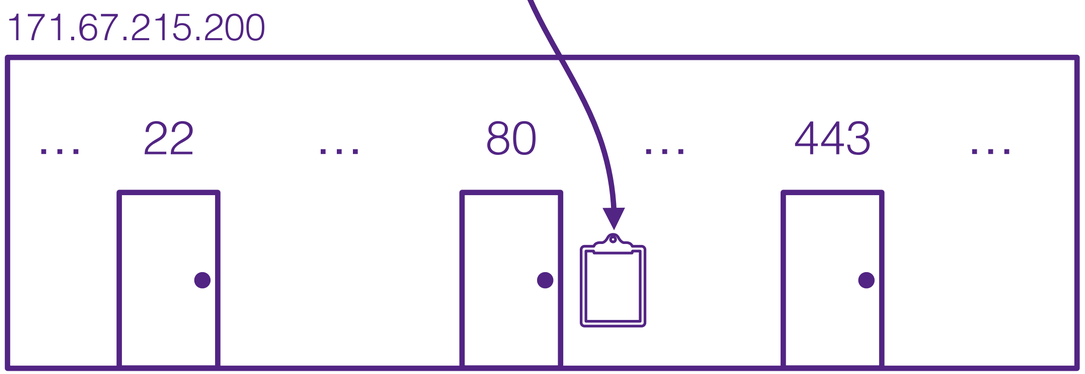
On some other computer, we want to talk to that server

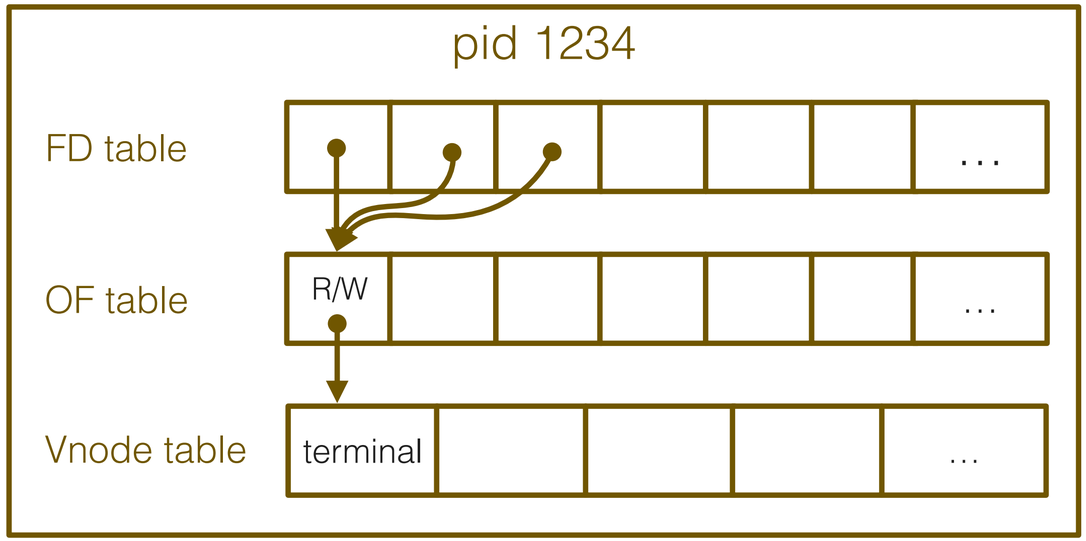


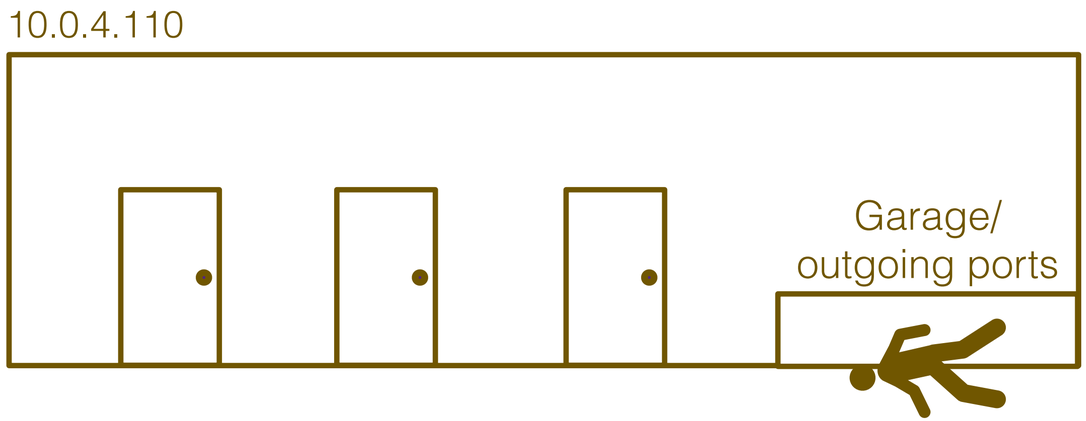


The “client” walks out to try to find 171.67.215.200:80

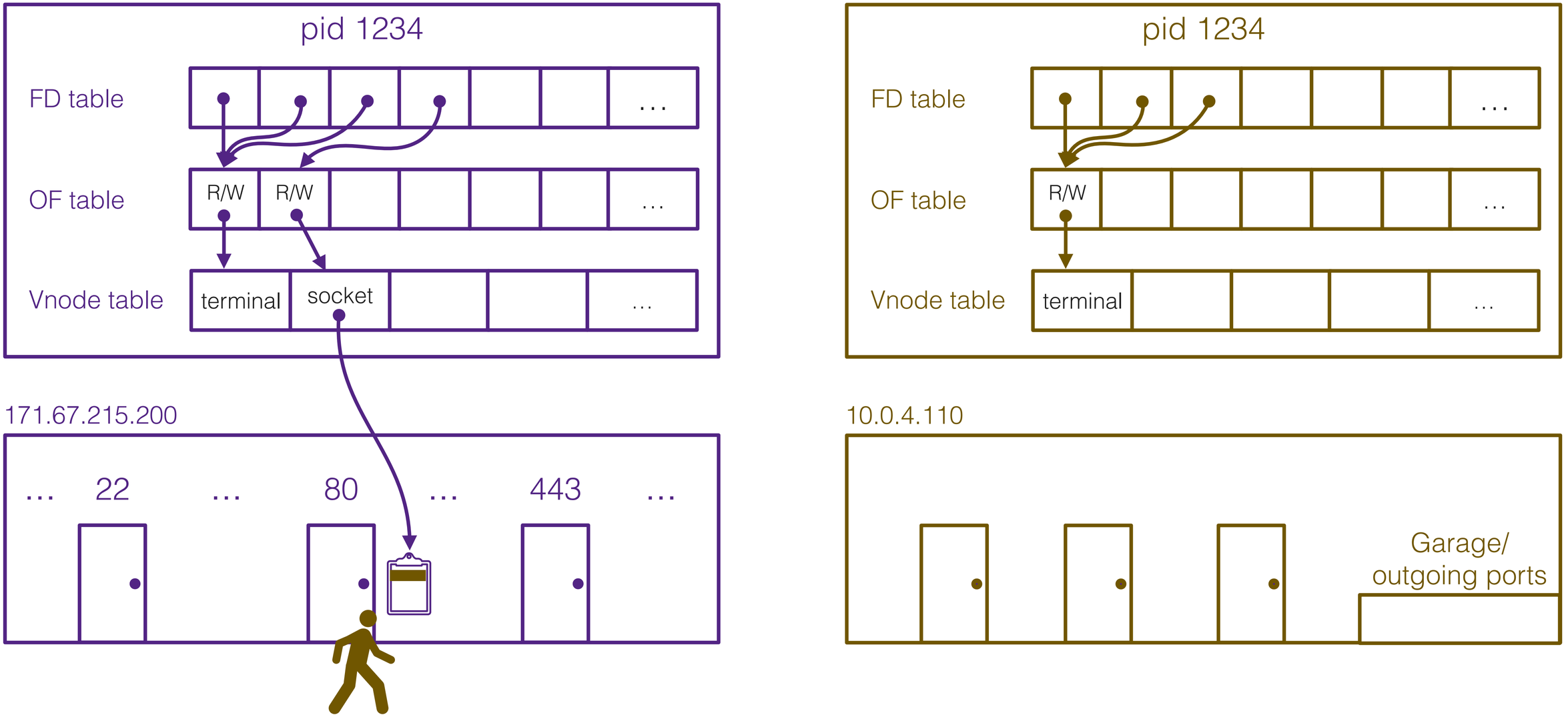




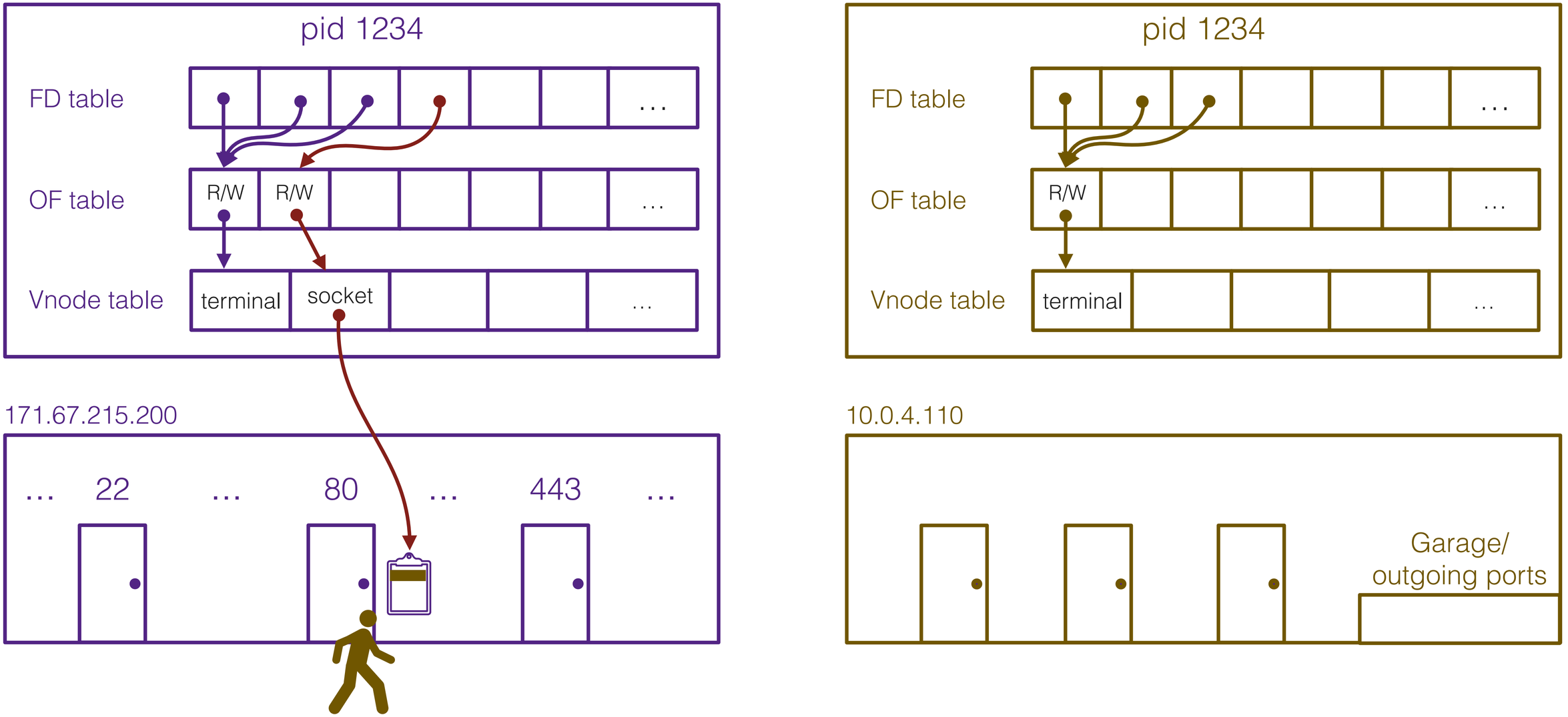




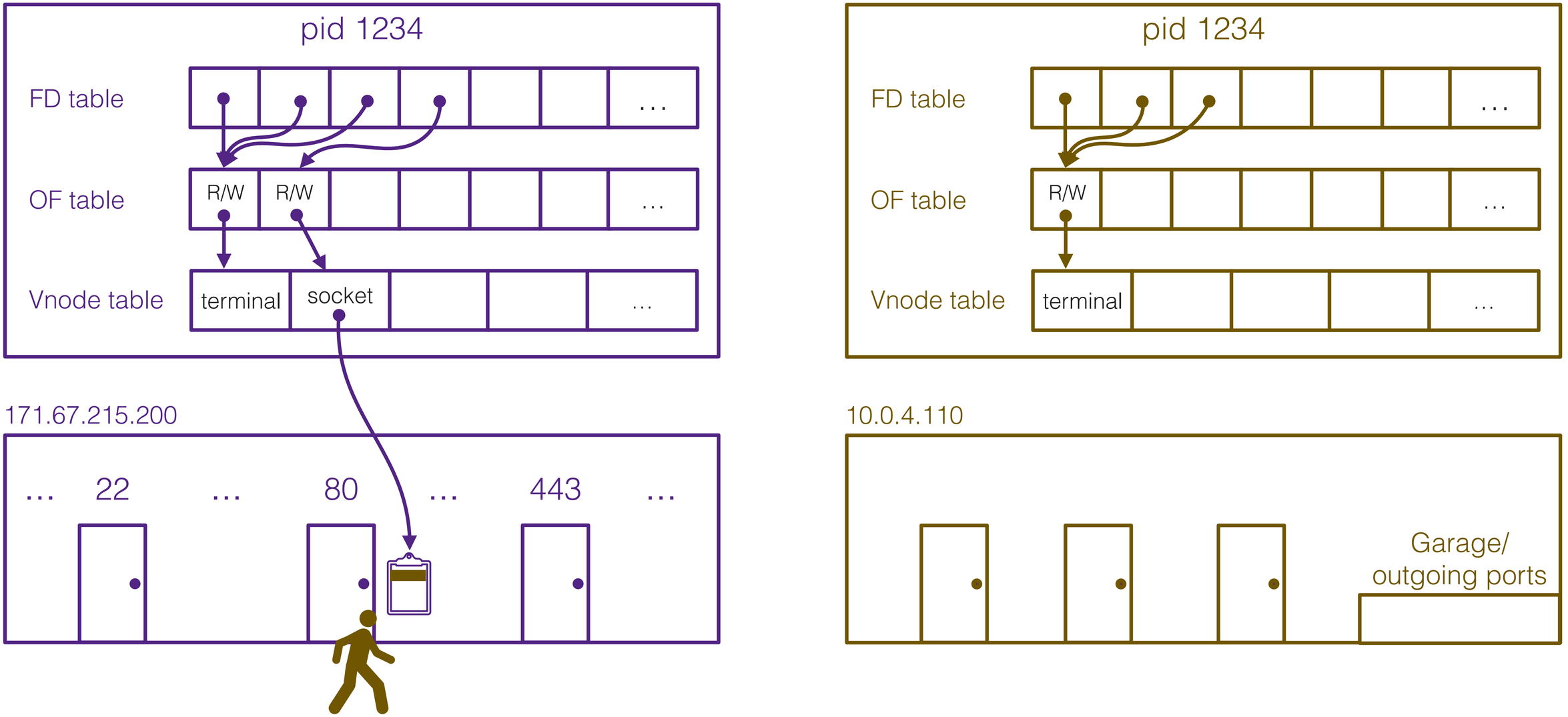
If successful, it adds itself to the waiting list



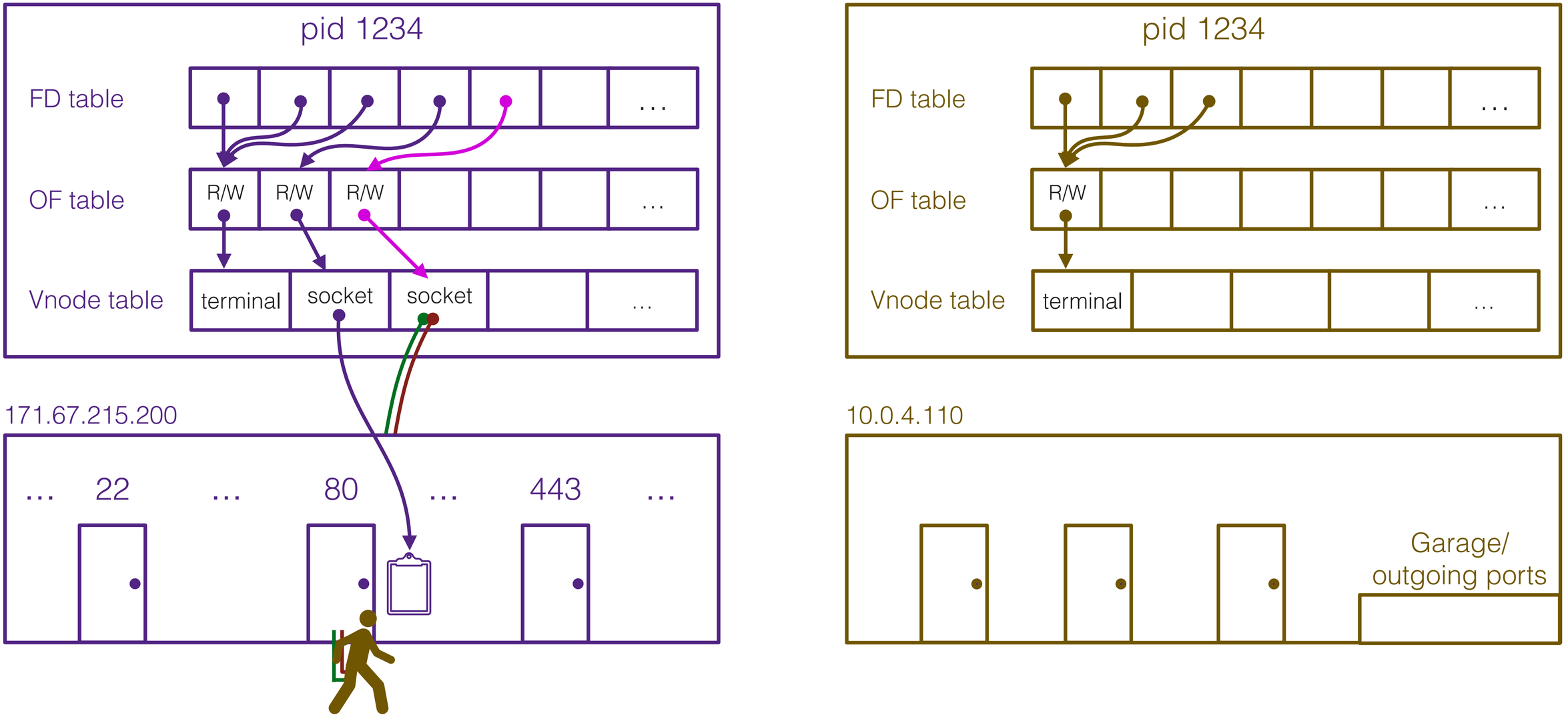
The server sees the client through its waiting list file descriptor



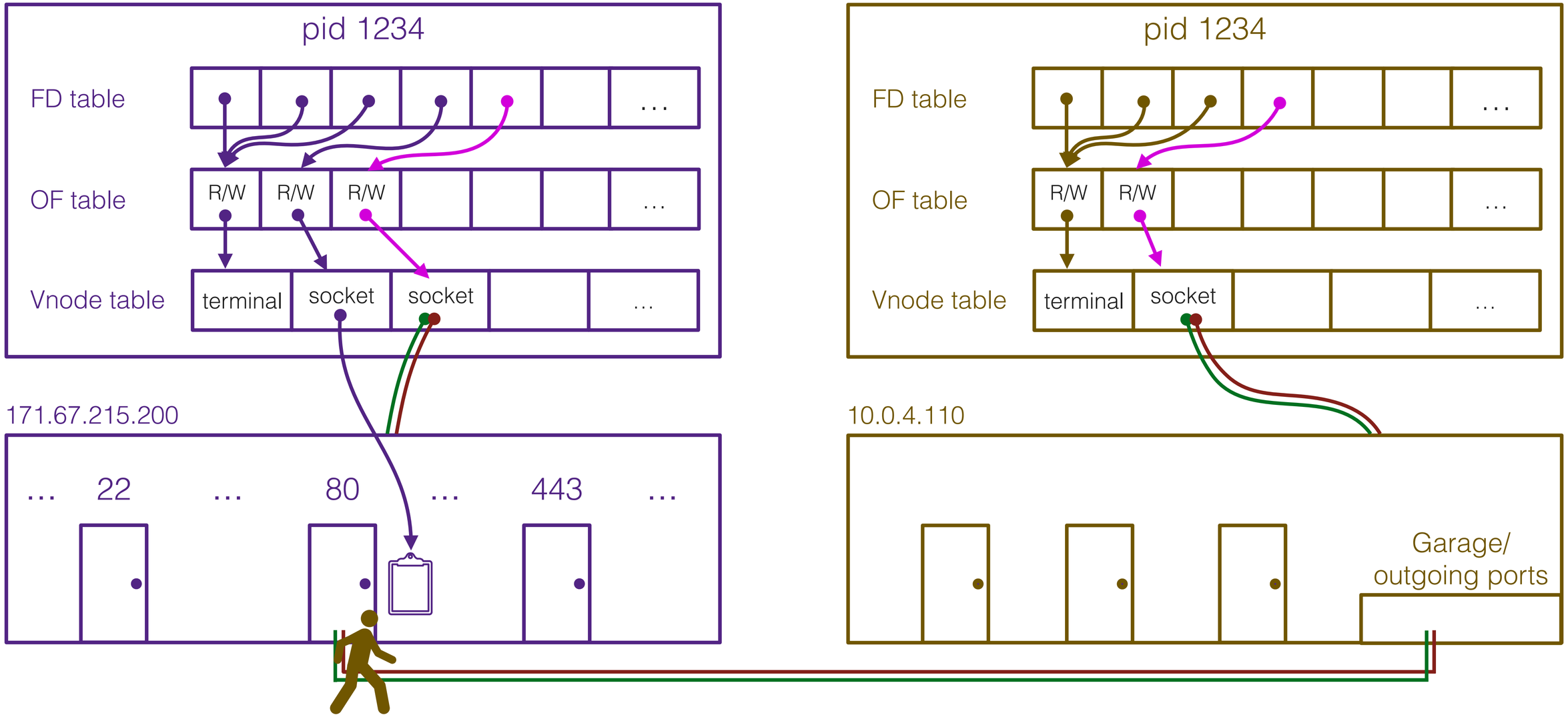
It takes the client off the waiting list and creates a new bidirectional "socket" that it can use to talk directly with the client



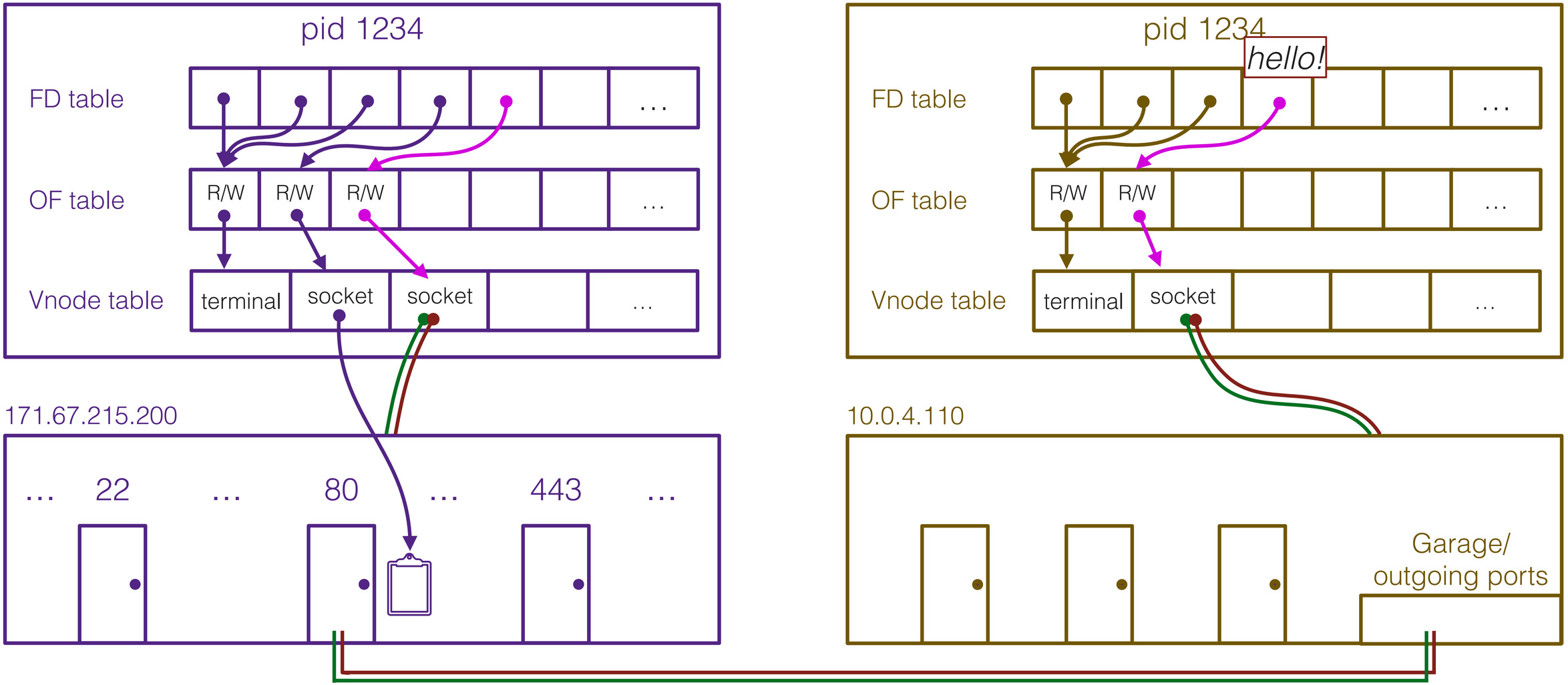
It takes the client off the waiting list and creates a new bidirectional "socket" that it can use to talk directly with the client



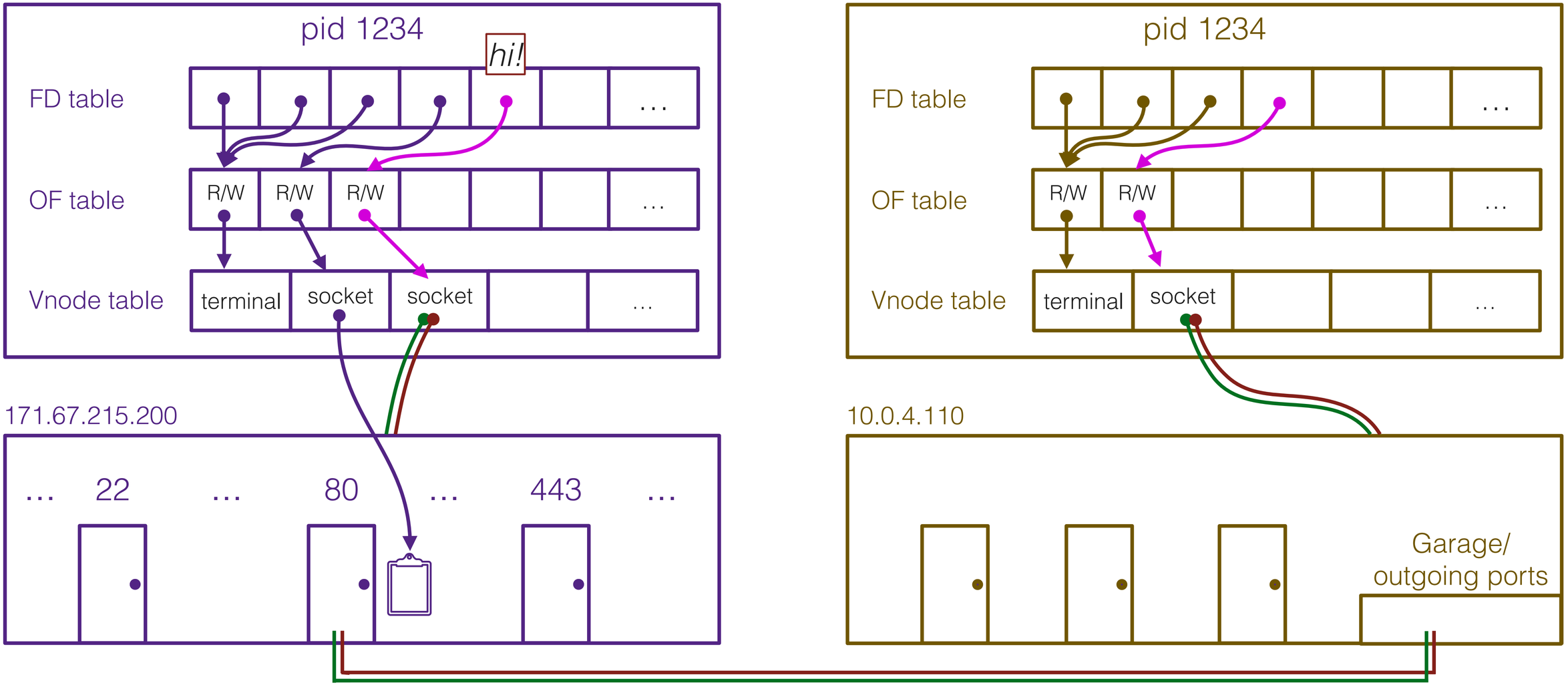
Successful in making a connection, the client also creates a new file descriptor it can use to talk to the server



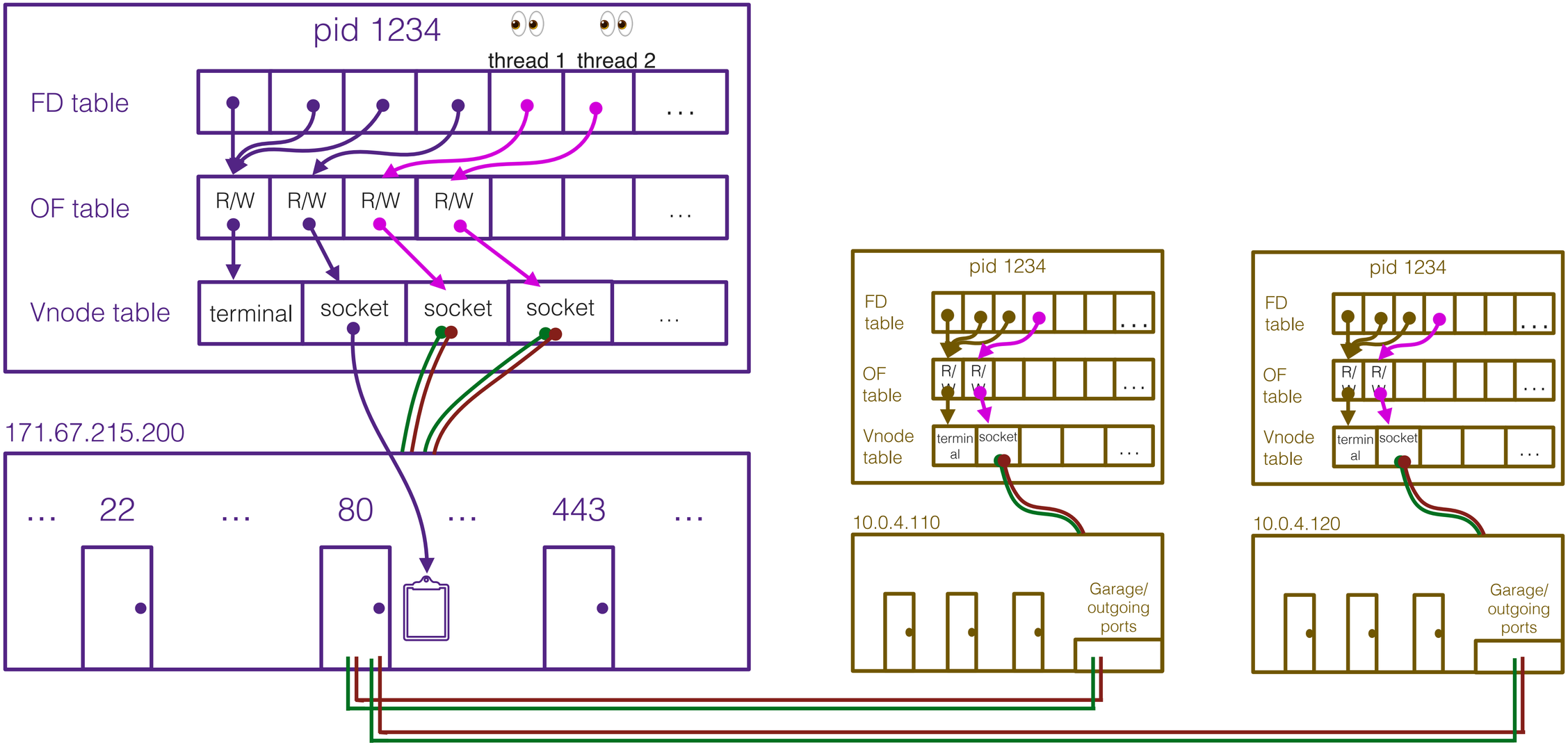
If the client writes to its fd 3, it will be readable on the server's fd 4



Similarly, if the server writes to fd 4, it will be readable on the client's fd 3



The server can talk to multiple clients at the same time, using separate file descriptors (often using a thread facilitate each conversation over each fd)



Scalability and Availability

Properties of networked systems

- Scalability: How well can the system grow as demands increase over time?
  - An unscalable system will not be able to grow to meet demand no matter how much resources you throw at it
- Availability: How well is the system able to stay available and avoid downtime?
  - Becomes increasingly challenging as a system scales
  - If an server is available 99.99% of the time (down only 0.88 hours/year), a system not engineered for fault tolerance relying on 1,000 servers will be available 99.99% ^ 1000 = 90.48% of the time (down 834 hours/year)
- (There are many more properties we will not talk about today)

Simple server setup

- Client looks up server's IP address using DNS
- Client connects to server's IP over the network
- Client and server each create a file descriptor for communication with each other

Simple server setup

- Is it scalable?
- Individual computers aren’t scalable
  - Becomes exponentially more expensive as you try to upgrade performance
  - Much cheaper if we could use two machines with commodity performance than one machine with 2x performance
  - Internet traffic has grown far faster than hardware has increased in power. Hardware can’t keep up even if our wallets could
- Scale out, not up!

Simple server setup

- Is it available?
- Hardly.
  - Server could get overloaded and run out of resources (memory, CPU time, file descriptors, etc)
  - Server could fail (system crashes, hardware fails, dog eats power cable, network outage, etc)

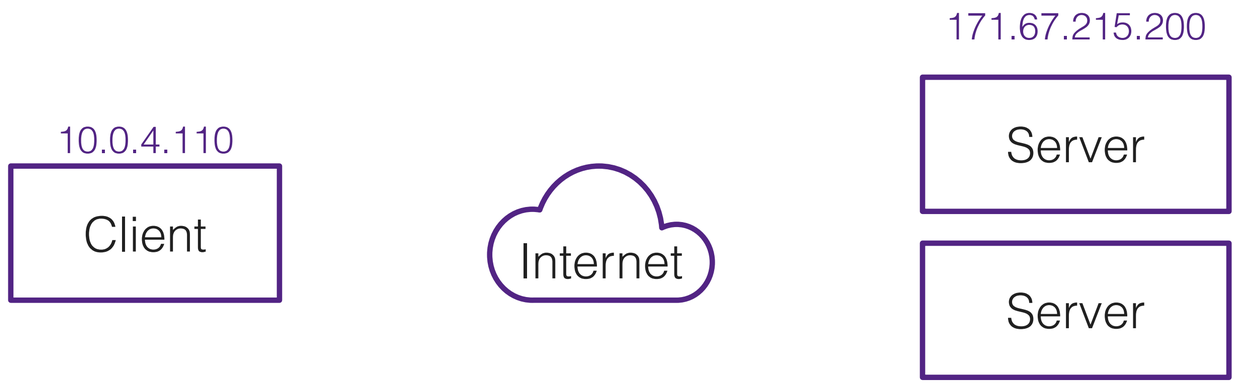
Distributed systems

- We want to distribute a system’s functionality over a large number of servers to achieve scalability and availability
- These servers talk to each other using networking to collaborate on whatever problem we are trying to solve

Scaling out

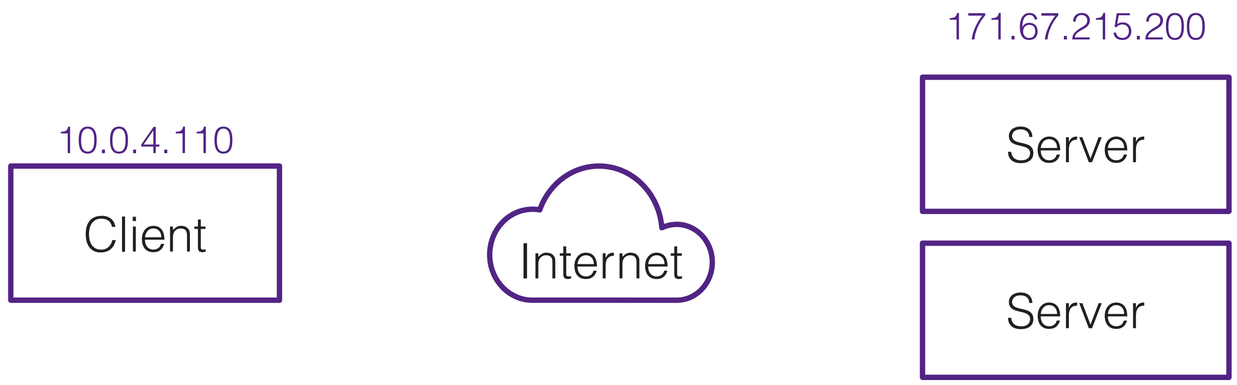
How can we design our system to make use of multiple servers?

Scaling out



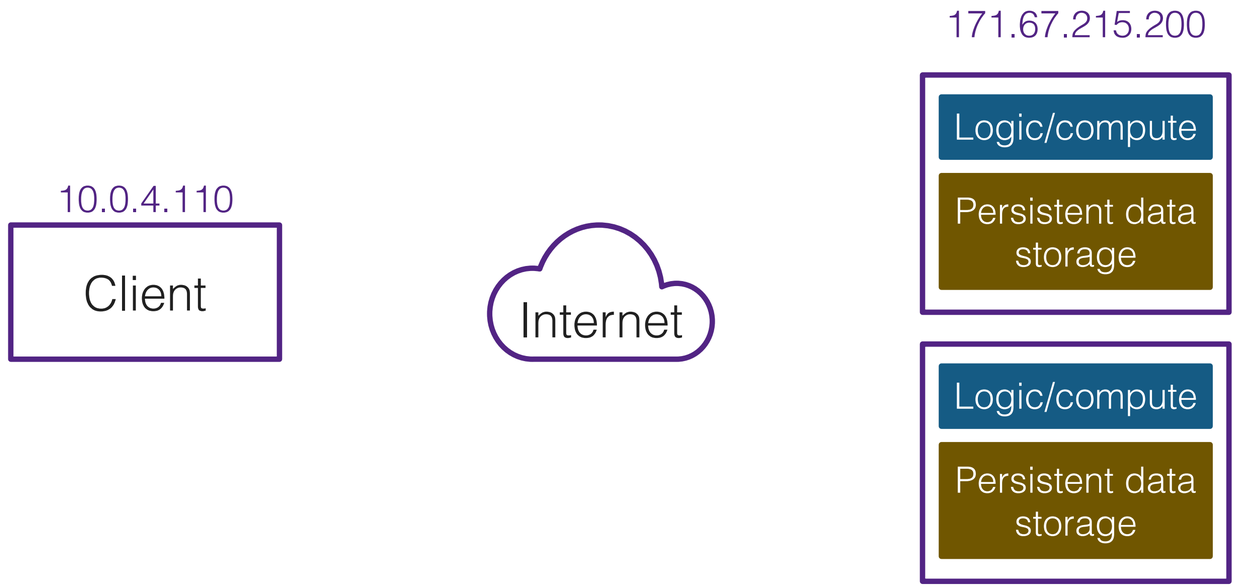
How can we design our system to make use of multiple servers?

Scaling out



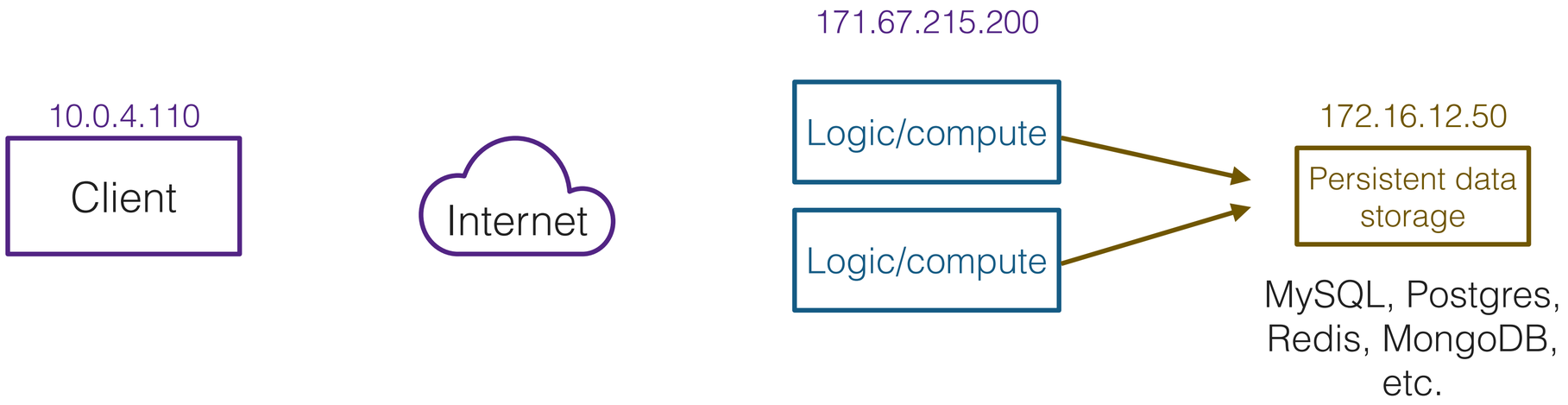
Simply duplicating our current setup won’t work.

Scaling out



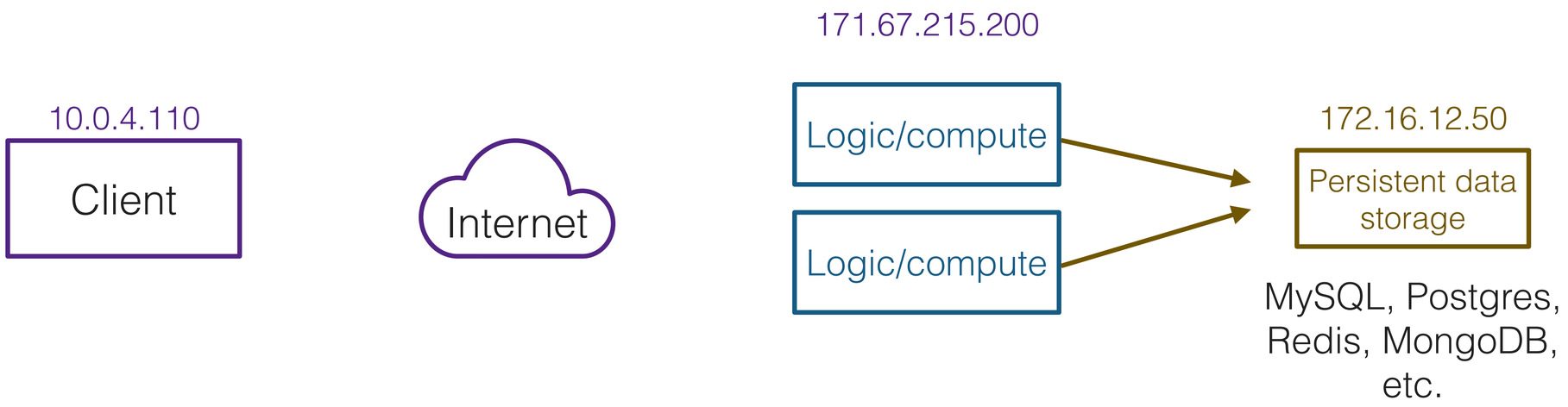
Simply duplicating our current setup won't work.
The duplicate servers would need to synchronize their data storage.
This is a very hard problem that is already solved by databases!

Scaling out



Simply duplicating our current setup won't work.
The duplicate servers would need to synchronize their data storage.
This is a very hard problem that is already solved by databases!

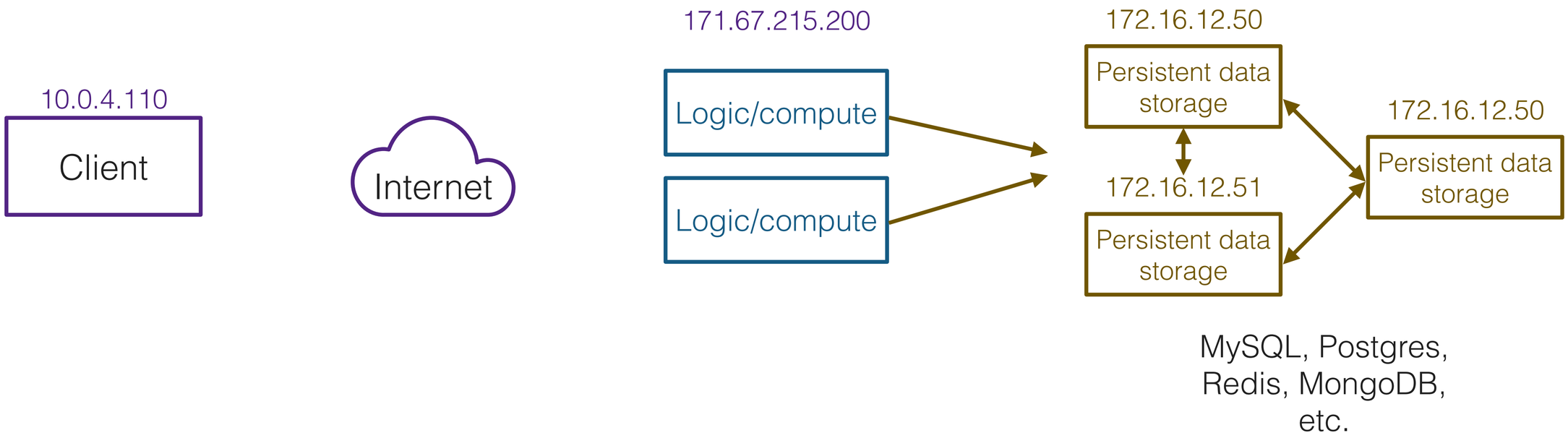
Scaling out



These database systems come with mechanisms to scale to multiple servers for reliability and performance

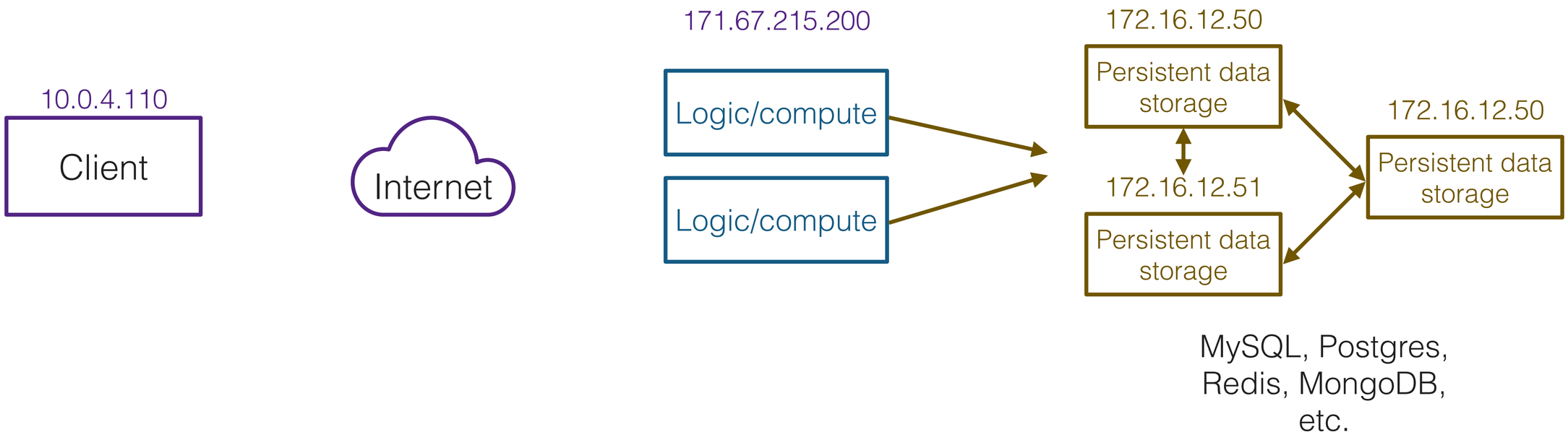
Scaling out

Take CS 245, CS 244B!



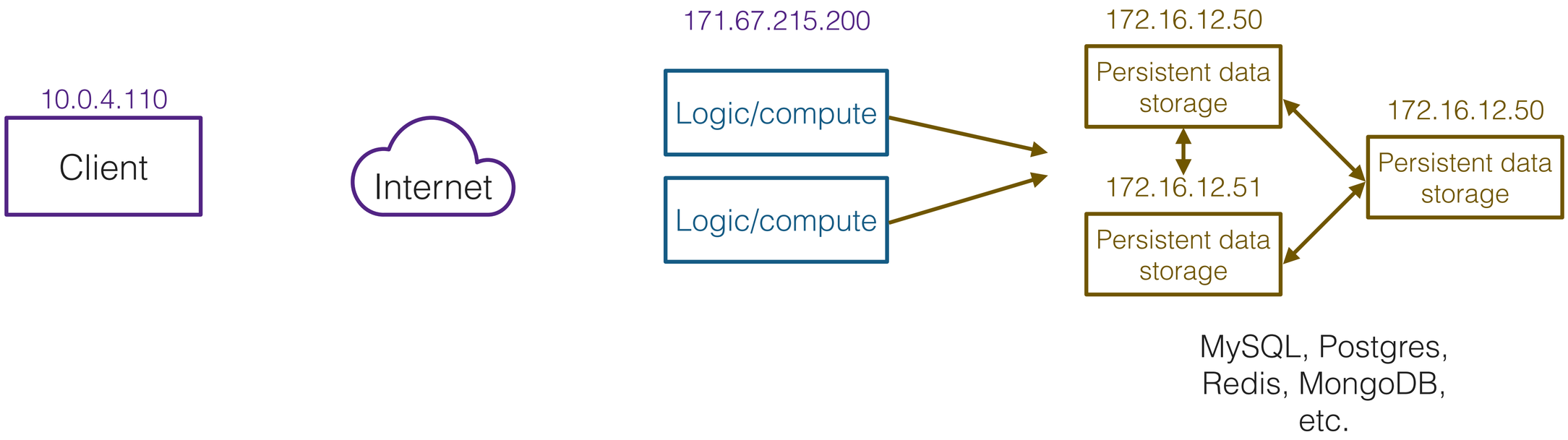
These database systems come with mechanisms to scale to multiple servers for reliability and performance

Scaling out



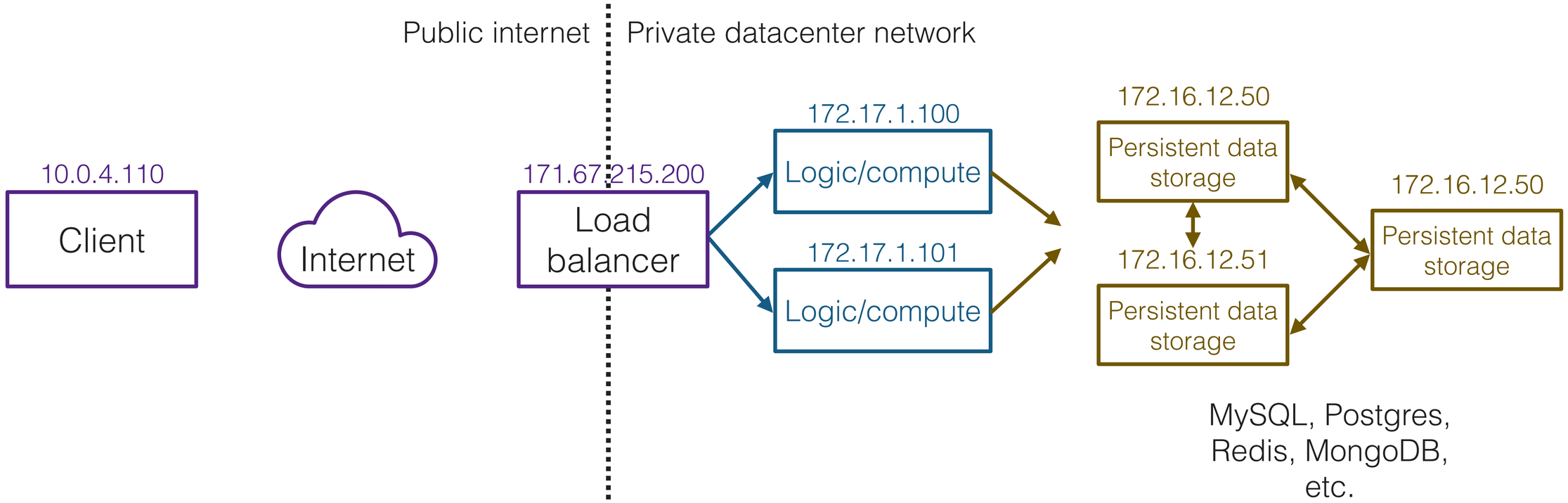
Still have a problem: Multiple servers, but only one IP!

Scaling out



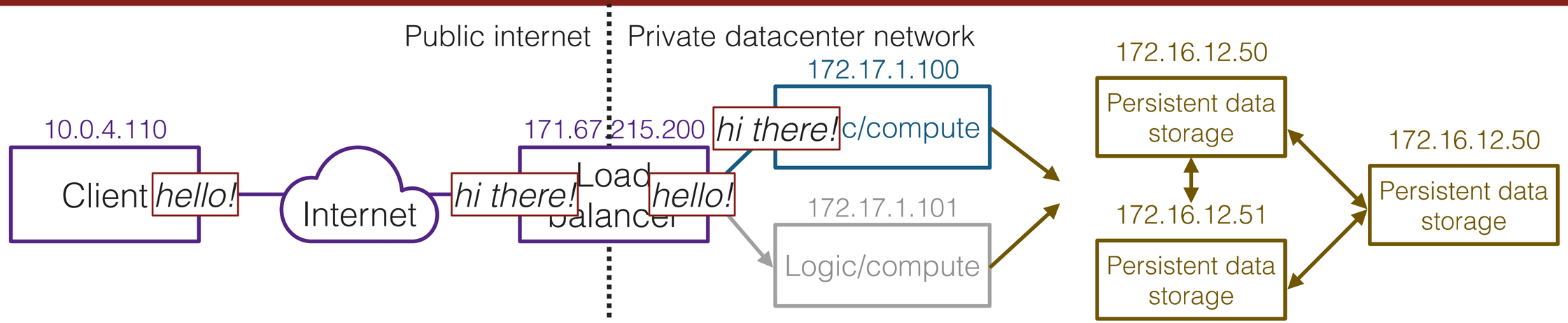
Load balancers: Distribute traffic across compute nodes

Scaling out



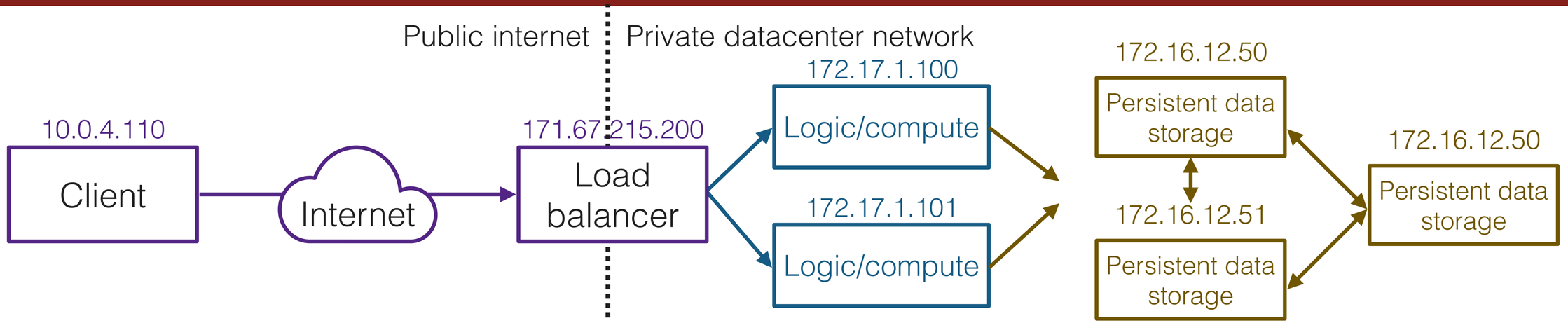
Load balancers: Distribute traffic across compute nodes

Load balancers



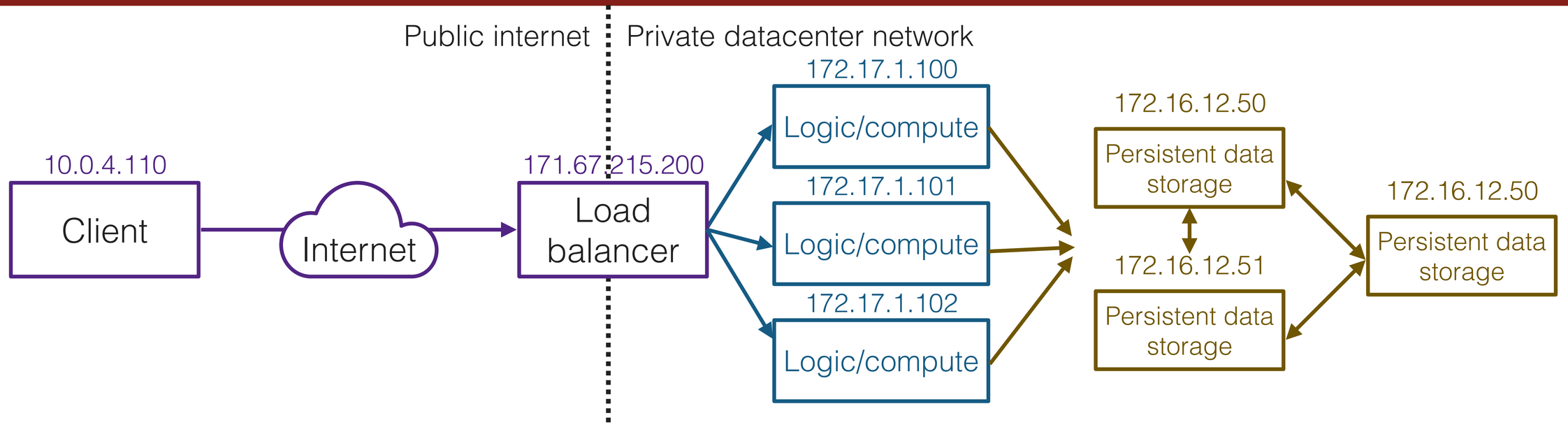
- When a client opens a connection to the load balancer, it selects a compute node and opens a connection to that compute node
  - Any traffic the client sends is relayed to the compute node. Any traffic the compute node sends is proxied back to the client
  - There are a variety of strategies for selecting the compute node (e.g. random selection, picking the one with the lowest load, round-robin, etc)
- The load balancer doesn't do anything else; anything resource-intensive is offloaded to the compute nodes. Consequently, load balancers can handle a large number of concurrent connections

Load balancers



- Scalability: If many clients are connecting, we can add more compute nodes

Load balancers



- Scalability: If many clients are connecting, we can add more compute nodes
- Availability: If one of the compute nodes fails, load balancer will detect that it isn't able to contact that server, and it can stop relaying traffic there
- Client never needs to know that our infrastructure is changing!
- Can we stop here?

Load balance your load balancers!

Load balance your load balancers!

- Systems carrying large amounts of traffic can't rely on a single load balancer
  - YouTube currently accounts for 15% of all internet traffic (source)
  - There's no way a single machine can handle that much traffic passing through it
- A lone load balancer introduces a single point of failure
  - Hardware failures are uncommon, but they do happen
  - Entire-datacenter failures are uncommon, but they do happen
  - Murphy's Law of large-scale systems: anything that can go wrong will go wrong! If you need high availability, you *have* to be prepared for the worst

Possible solution: Round-robin DNS

- DNS can return *multiple* IP addresses for a given hostname, shuffling the order
- Clients will pick the first one, moving down the list if IPs are unreachable
- You can specify multiple load balancers in this list, potentially in different datacenters
- 🍌 `dig +noall +answer reddit.com`

```
reddit.com.             147     IN      A       151.101.193.140
reddit.com.             147     IN      A       151.101.129.140
reddit.com.             147     IN      A       151.101.65.140
reddit.com.             147     IN      A       151.101.1.140
```

- Second time:

🍌 `dig +noall +answer reddit.com`

```
reddit.com.             339     IN      A       151.101.1.140
reddit.com.             339     IN      A       151.101.129.140
reddit.com.             339     IN      A       151.101.193.140
reddit.com.             339     IN      A       151.101.65.140
```

Downsides of DNS load balancing

- Not very intelligent: can’t take into account whether some servers are more overloaded than others
- DNS infrastructure has a lot of caching. It’s hard to consistently rotate the order of IPs if your DNS responses get cached
  - Leads to uneven distribution of load
- If one of the servers fails, DNS will happily continue announcing its IP address
  - Clients will eventually try one of the other IP addresses when they realize the dead server is dead, but this can significantly increase latency to establish a connection

Huge sites, one IP?

- 🍌 `dig +noall +answer www.google.com`

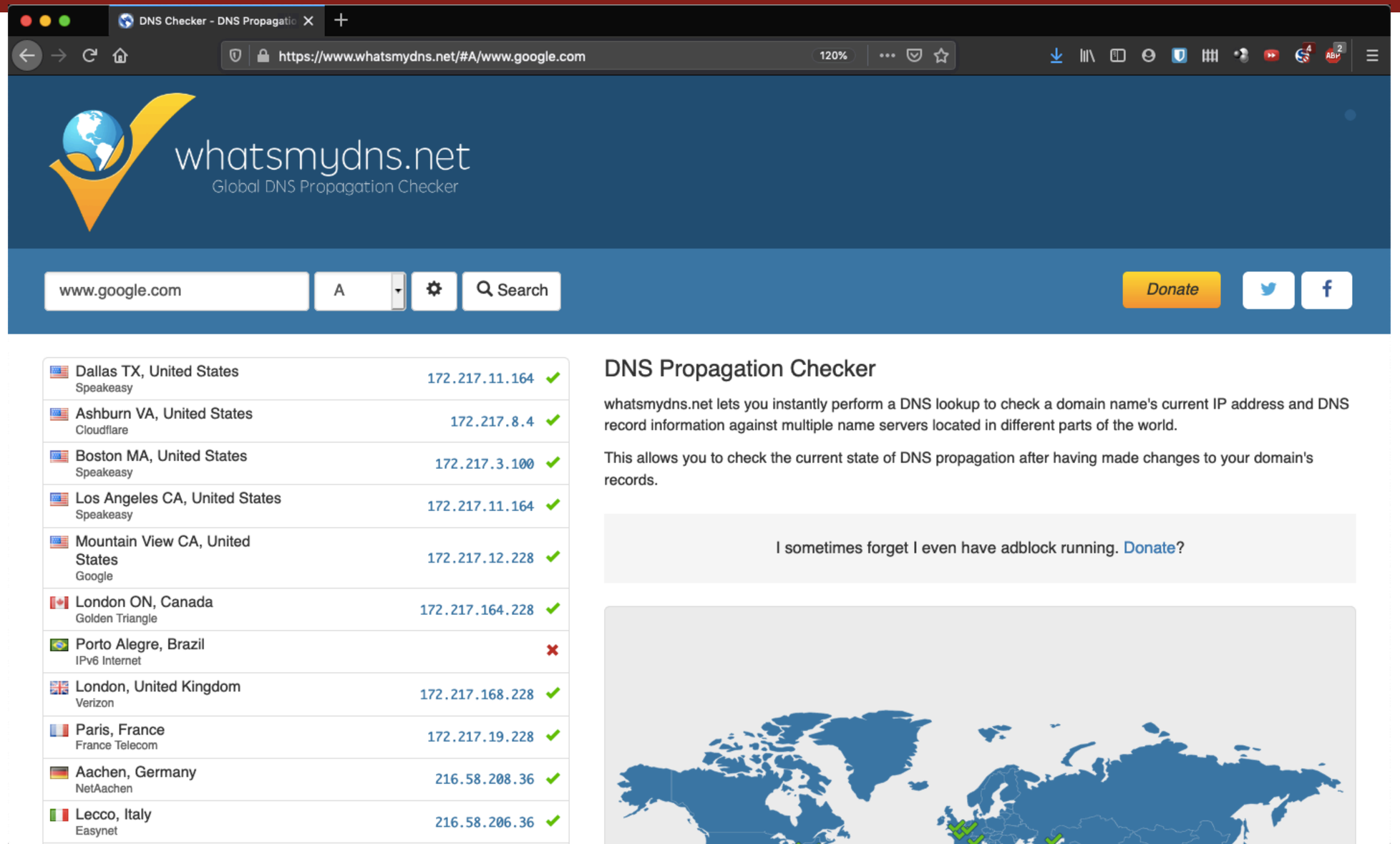
```
www.google.com.                 69      IN      A       216.58.217.196
```

- 🍌 `dig +noall +answer www.facebook.com`

```
www.facebook.com.       4314    IN      CNAME   star-mini.c10r.facebook.com.
star-mini.c10r.facebook.com. 32 IN      A       31.13.70.36
```

- What's going on?

Geographic routing with DNS



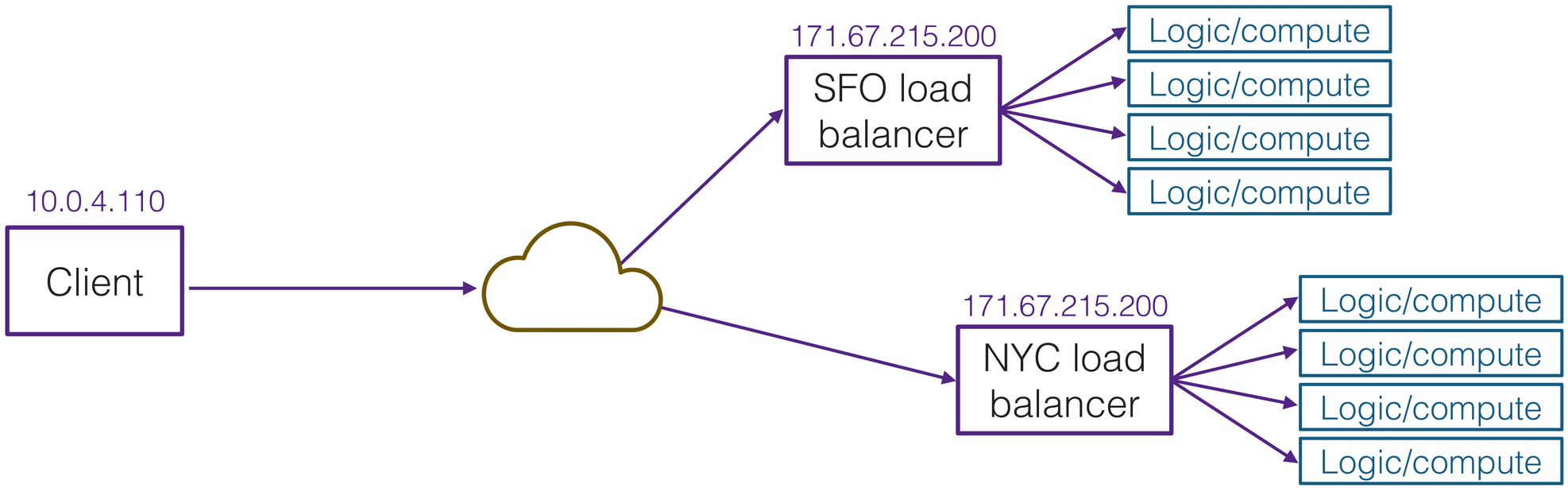
Geographic routing with DNS

Dallas TX, United States Speakeasy	172.217.11.164 ✔
Ashburn VA, United States Cloudflare	172.217.8.4 ✔
Boston MA, United States Speakeasy	172.217.3.100 ✔
Los Angeles CA, United States Speakeasy	172.217.11.164 ✔
Mountain View CA, United States Google	172.217.12.228 ✔
London ON, Canada Golden Triangle	172.217.164.228 ✔
Porto Alegre, Brazil IPv6 Internet	✖
London, United Kingdom Verizon	172.217.168.228 ✔
Paris, France France Telecom	172.217.19.228 ✔
Aachen, Germany NetAachen	216.58.208.36 ✔
Lecco, Italy Easynet	216.58.206.36 ✔
Yeditepe, Turkey Yeditepe University	172.217.169.164 ✔
Astrakhan, Russia Astrakhan Teleco	173.194.73.99 173.194.73.103 173.194.73.104 173.194.73.105 ✔

- DNS servers can respond with the IP for the load balancer that is closest to the client
- Reduces connection latency and helps to distribute traffic
- Doesn’t fix availability… If local datacenter goes down, want to fail over to other datacenters

IP Anycast

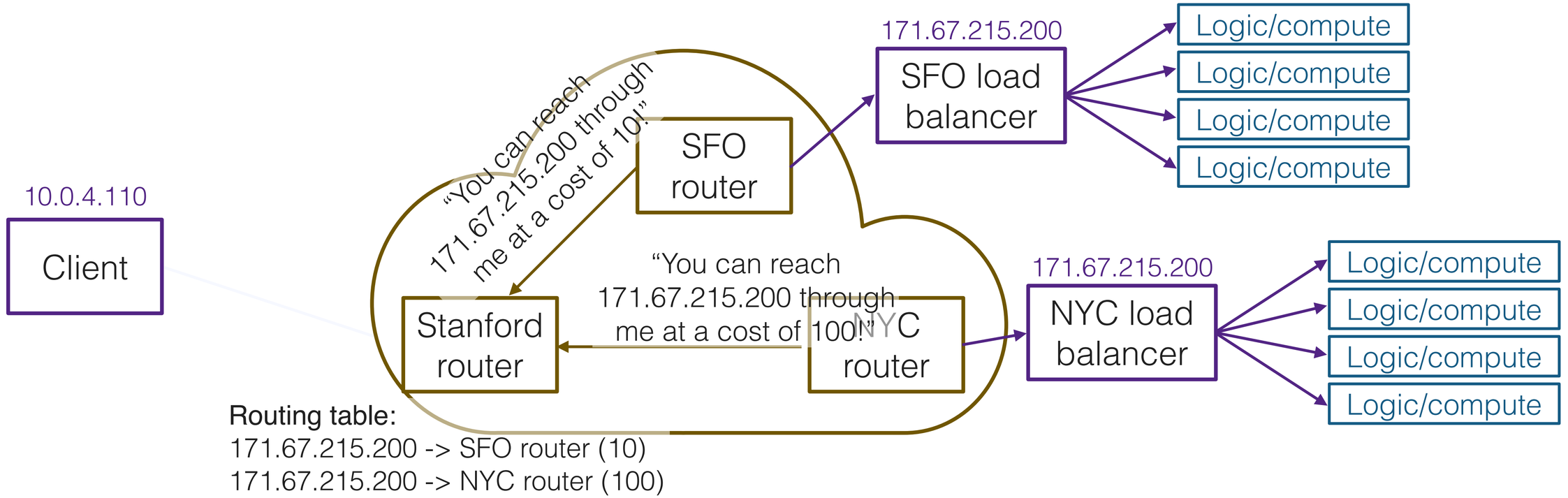
- Though we don't usually think like this, it's possible for a single IP address to correspond to *multiple* computers
- Multiple datacenters can announce to the internet that they "own" a particular IP



Note: a datacenter will almost always have multiple load balancers to distribute load and provide availability.

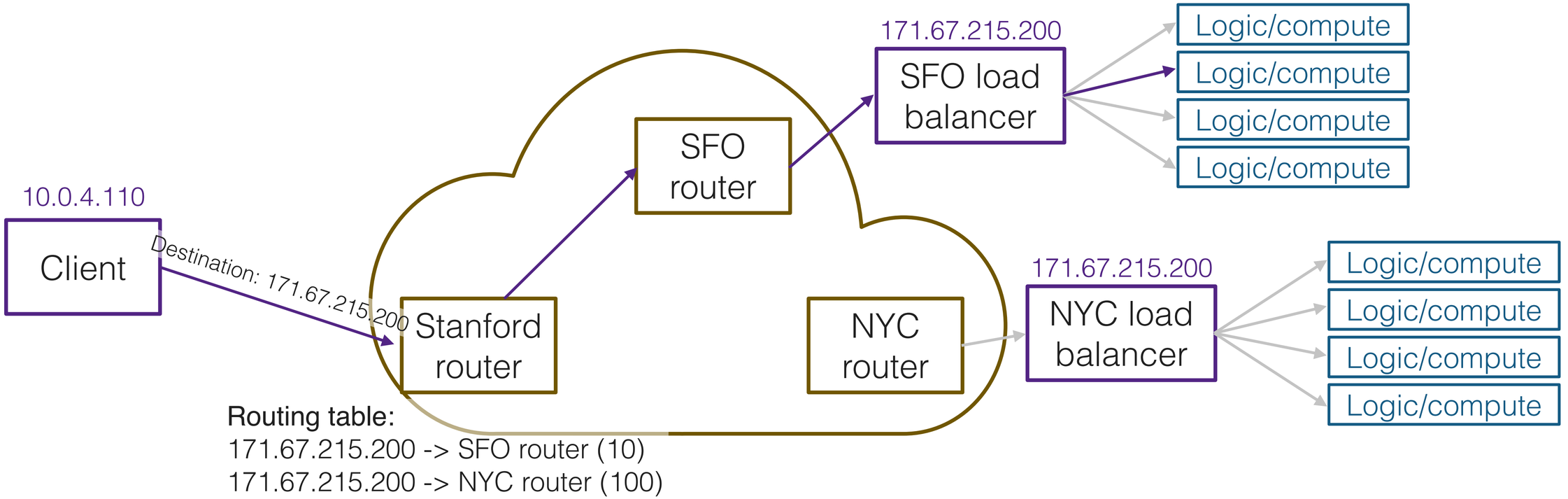
IP Anycast

- Though we don't usually think like this, it's possible for a single IP address to correspond to *multiple* computers
- Multiple datacenters can announce to the internet that they "own" a particular IP



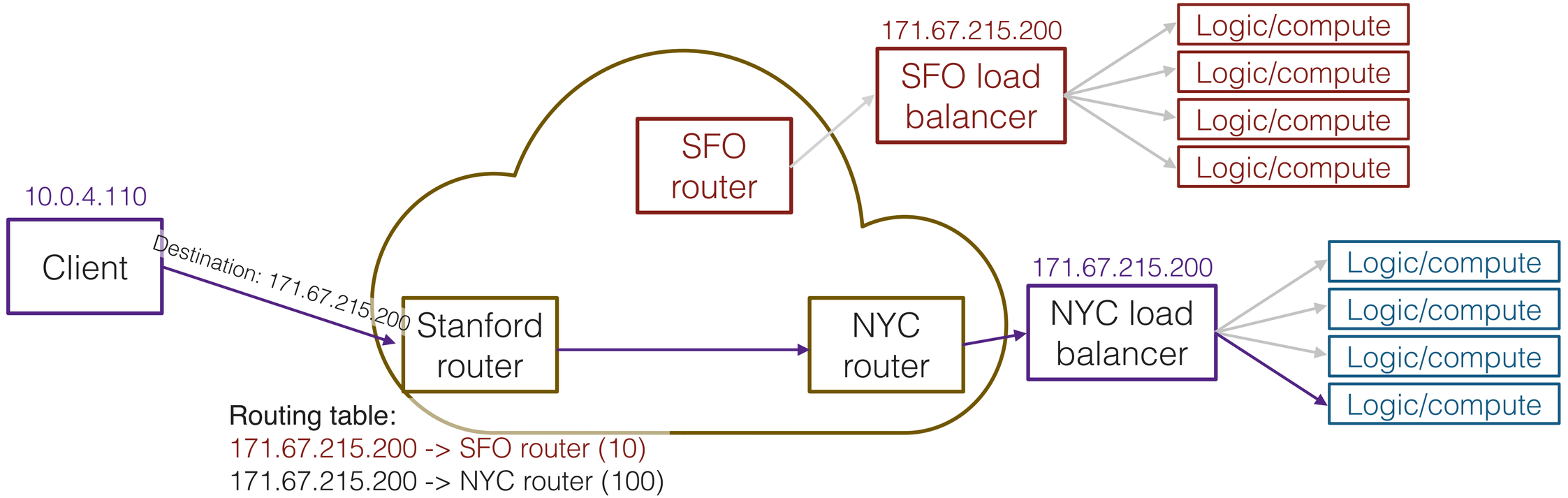
IP Anycast

- Though we don't usually think like this, it's possible for a single IP address to correspond to *multiple* computers
- Multiple datacenters can announce to the internet that they "own" a particular IP
- When a client tries to connect to an IP, they'll use the datacenter that is closest to them
- If one of the datacenters goes down, the internet will notice and reroute traffic



IP Anycast

- Though we don't usually think like this, it's possible for a single IP address to correspond to *multiple* computers
- Multiple datacenters can announce to the internet that they "own" a particular IP
- When a client tries to connect to an IP, they'll use the datacenter that is closest to them
- If one of the datacenters goes down, the internet will notice and reroute traffic



Engineer for failure

Chaos engineering

- To design reliable networked systems, you must assume any part of the system can fail
- But in a complex system, it's hard to predict all failure modes
- Hard to learn how a system will fail until it fails
- Solution? Intentionally induce failure!
  - (in a controlled environment, where we can fix problems quickly, instead of having unexpected disasters at 3am)
- Netflix philosophy of Chaos Engineering: "the discipline of experimenting on a system in order to build confidence in the system's capability to withstand turbulent conditions in production."

Netflix Simian Army

- Chaos Monkey
  - Original tool, intended to simulate a thought experiment: If you were to give a monkey a wrench and let it loose in a datacenter, what would happen?
  - Randomly terminates servers in production, exposing engineers to frequent failures and incentivizing fault-tolerant design
- Chaos Gorilla: Randomly terminates an entire datacenter
- Chaos Kong: Randomly terminates an entire geographic *region*
- Others: Latency Monkey, Doctor Monkey, Janitor Monkey, Conformity Monkey, etc.

More reading

- https://blog.codinghorror.com/working-with-the-chaos-monkey/
  - “Raise your hand if where you work, *someone deployed a daemon or service that randomly kills servers and processes in your server farm*. Now raise your other hand if that person is still employed by your company.

    Who in their right mind would willingly choose to work with a Chaos Monkey?”
- https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116
- http://principlesofchaos.org/