Chapter 2 · Part 2

Socket Programming and HTTP

Building a connection with the socket calls, then the message format two programs agree on.

Yaxiong Xie

Part 1 of 4

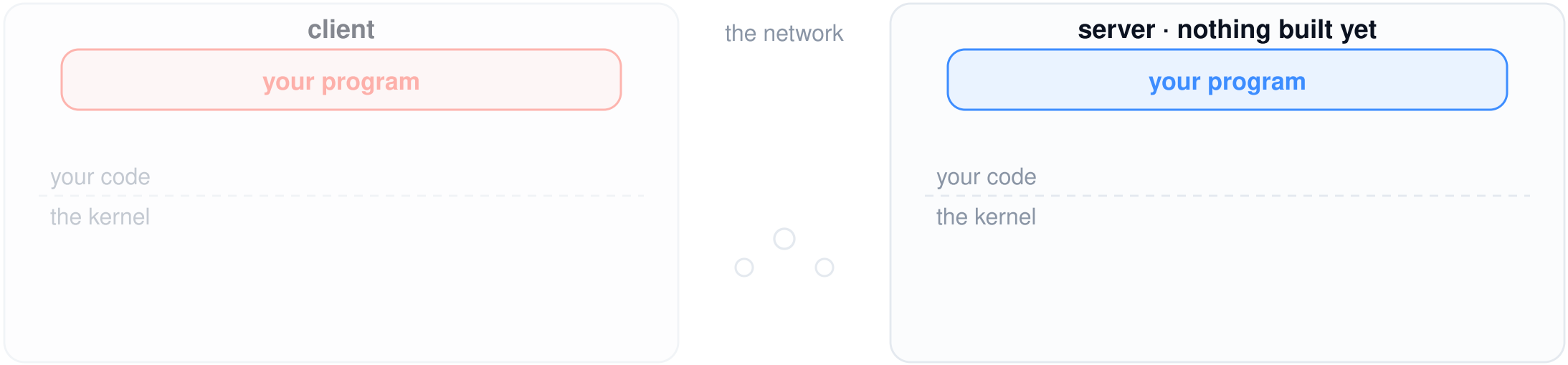
How do two programs build a connection?

01

Socket programming

One picture of two machines. Every slide changes one thing in it and names the call that did it — then a second client arrives and we ask what a connection really is.

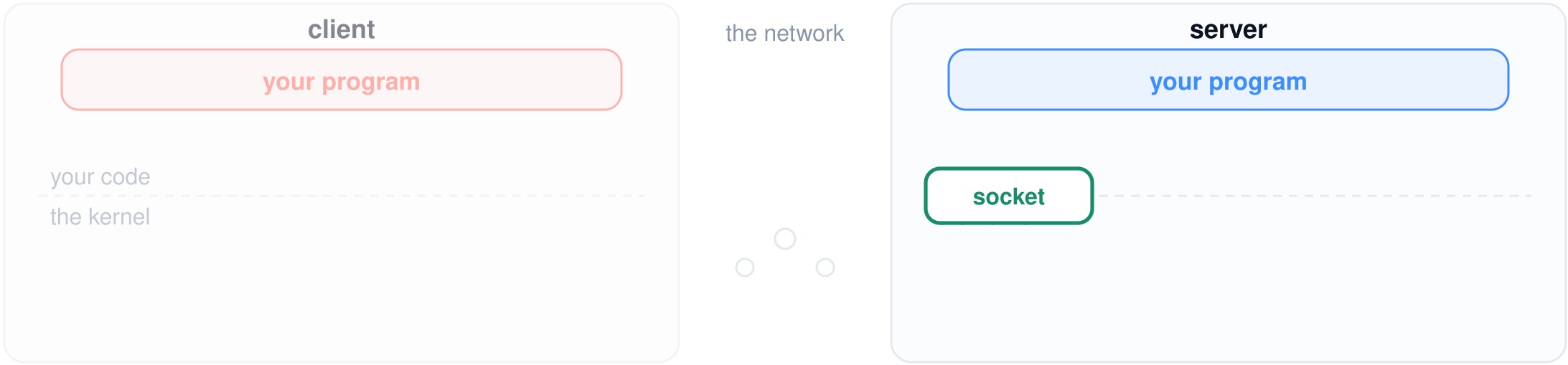
A client can arrive at any moment



If the server is not already listening when the client arrives, the attempt is refused. So the server is what we build first.

The server must be listening before the client runs.

socket() creates the endpoint



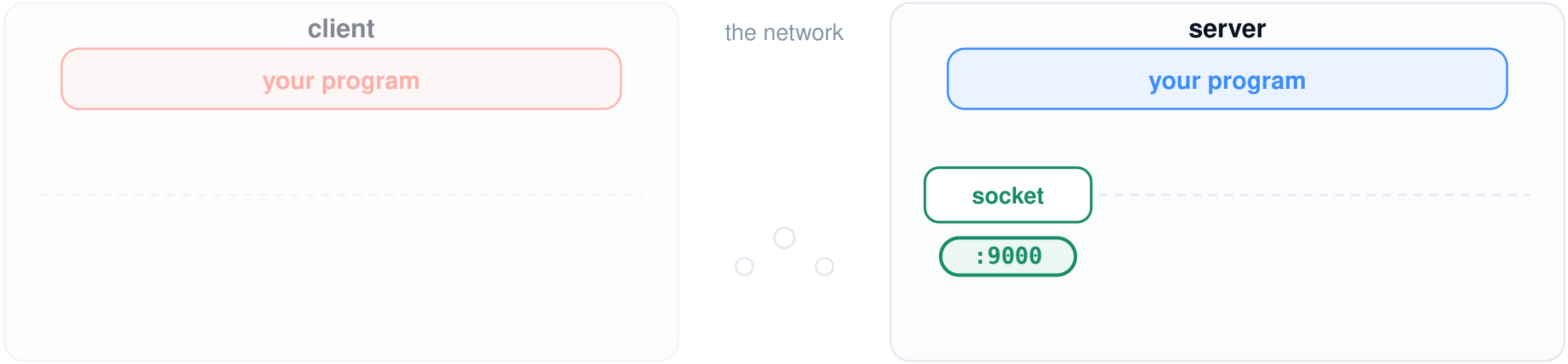

```
srv = socket(AF_INET, SOCK_STREAM)
```

line 3

A TCP endpoint now exists inside the kernel. The call returns its number, and that number is all the program holds.

Nothing has reached the network yet.

bind() gives it an address other machines can name



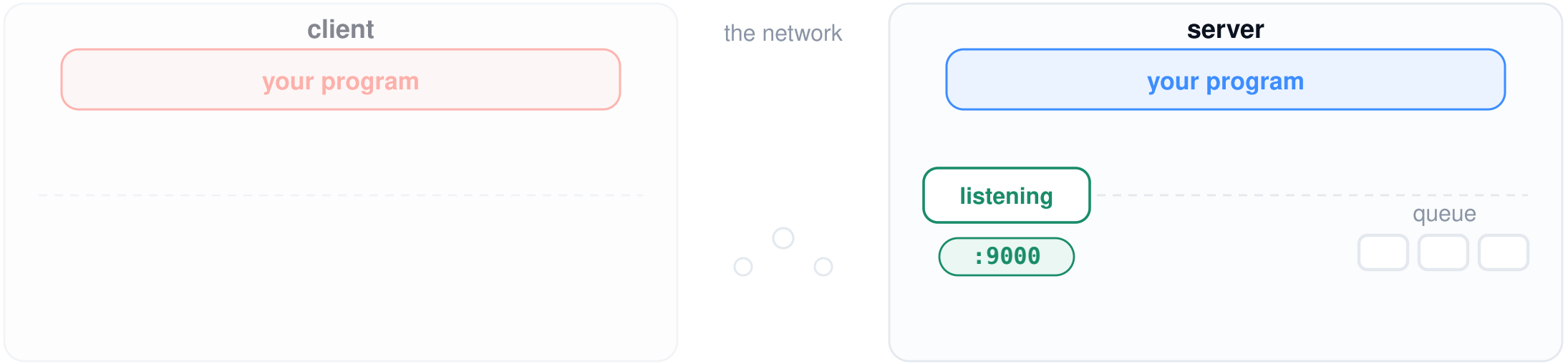

```
srv.bind(("", 9000))
```

line 4

The empty string means every address of this machine; 9000 is the port. A client does not do this — the kernel picks a port for it.

This is the number the client has to know in advance.

listen() adds the queue the kernel will fill

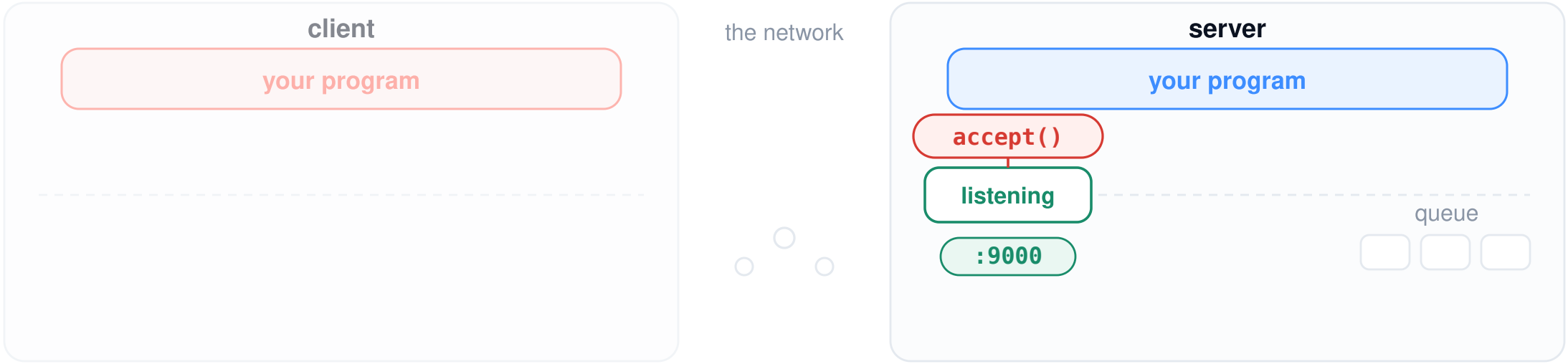


`srv.listen(128)` line 5

128 is the backlog: the largest number of finished connections the kernel will hold in this queue while the program has not yet accepted them. Linux caps it at net.core.somaxconn, 4096 by default.

The kernel now completes handshakes for port 9000 without the program.

accept() waits for the queue to have something in it



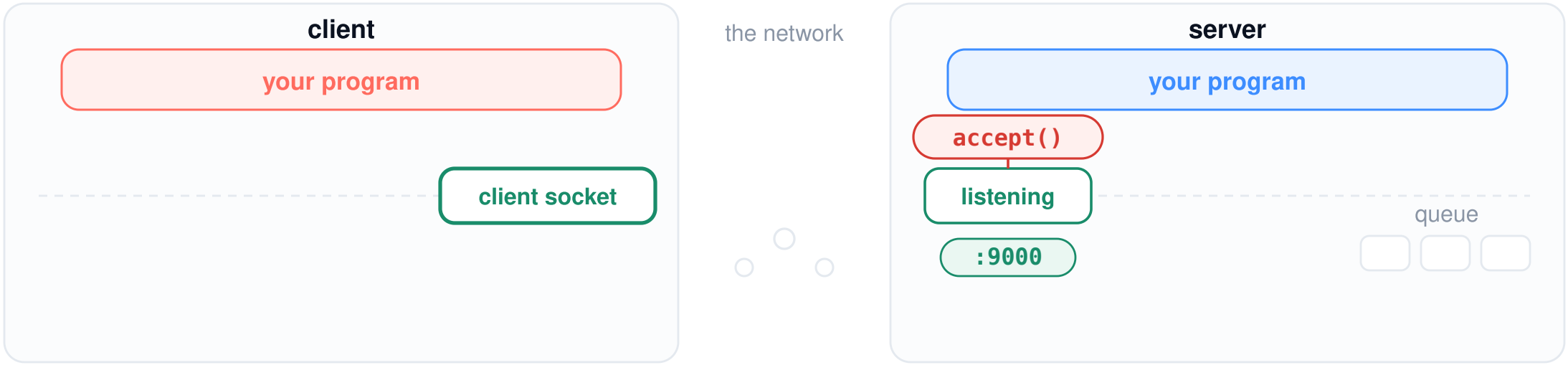

```
conn, peer = srv.accept()
```

line 8

The red pill marks where the program is: inside accept(), on the listening socket. The queue is empty, so the call does not return.

The server is up. Nothing more happens until a client connects.

The client makes a socket with the same call



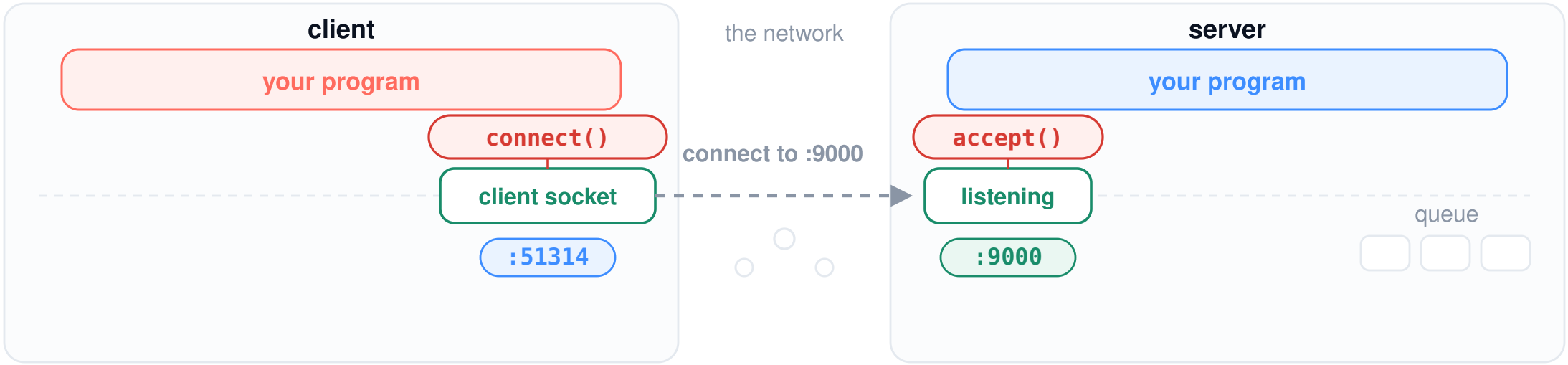

```
cli = socket(AF_INET, SOCK_STREAM)
```

line 3

Identical call, identical result: an endpoint in the kernel and a number. Nothing about it is a "client" yet, and nothing has left the machine.

The next call is what makes this socket a client.

connect() reaches for the port the server bound




```
cli.connect(("203.0.113.7", 9000))
```

line 4

The kernel takes a local port for the client — 51314, nobody chose it — and starts the handshake. connect() waits for it to finish.

Address and port must match exactly what bind() claimed.

SYN and ACK are two bits in the TCP header

the control bits in every TCP header, one bit each

SYN — set on a segment that carries the sender's starting sequence number

ACK — set when the acknowledgement number in this segment is valid: everything before it has arrived

the second segment of the three sets both bits at once

Neither bit carries data. What they do is say how to read the sequence number and acknowledgement number fields in the same header (RFC 9293 § 3.1).

SYN carries a starting sequence number. ACK confirms one.

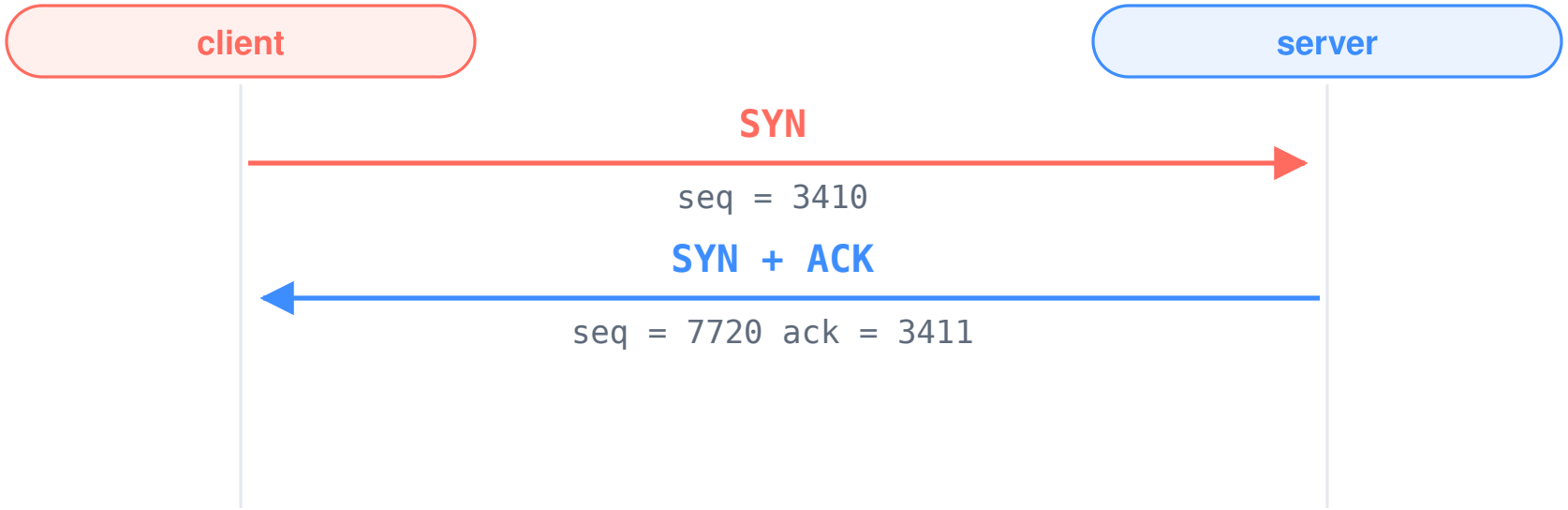
Segment 1: the client sends SYN

neither program has run a line

The client's kernel picks a starting sequence number — 3410 here, chosen at random — and sends a segment with the SYN bit set and no data in it.

The first segment says where the client's numbering starts.

Segment 2: the server answers with both bits set

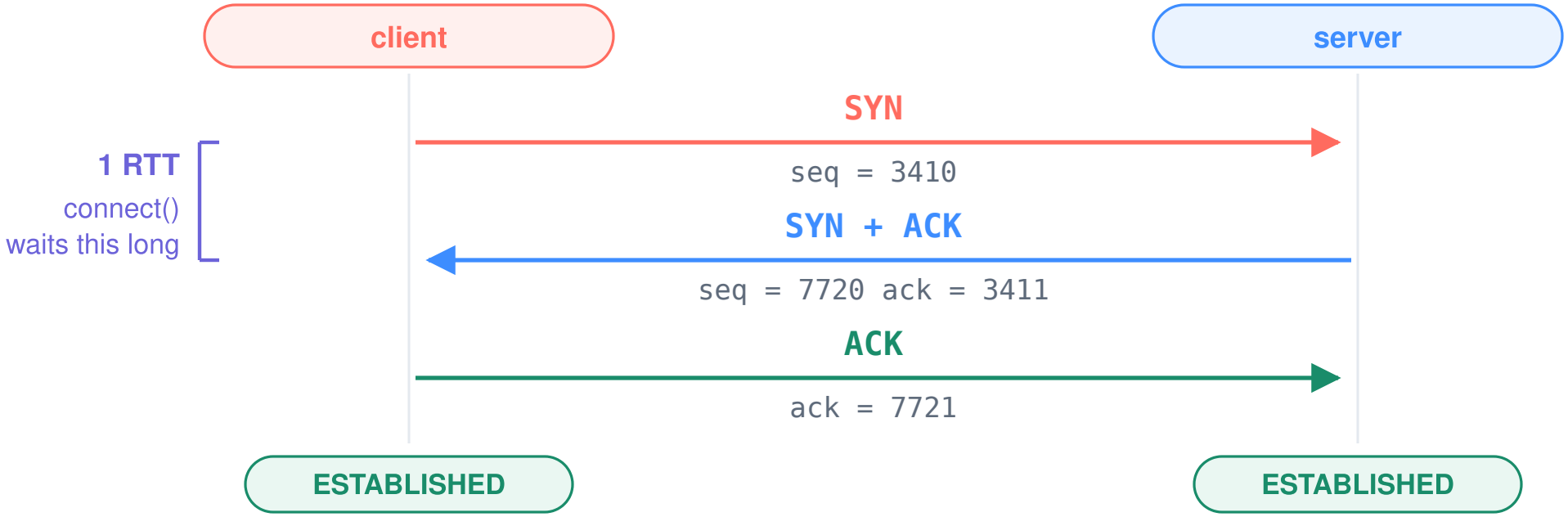


neither program has run a line

The server's kernel picks its own starting number, 7720, and sets SYN for it and ACK for the client's. ack = 3411 because a SYN occupies one sequence number of its own.

After two segments each side knows where the other's numbering starts.

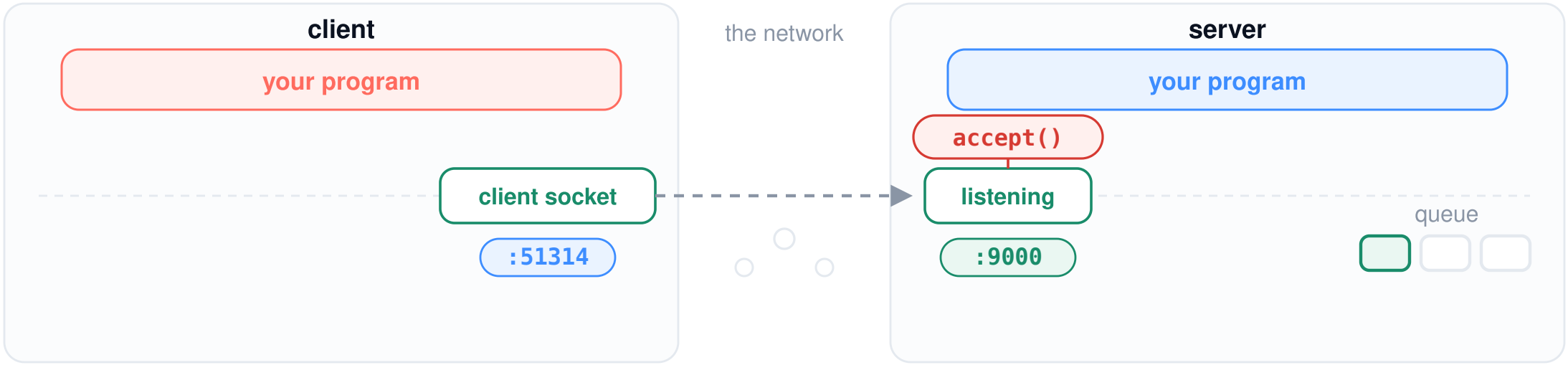
Segment 3: the client acknowledges 7720



The connection is now ESTABLISHED in both kernels. The client waited one round trip: its SYN out and the SYN+ACK back. The third segment costs it nothing.

A TCP connection costs one round trip before it can carry data.

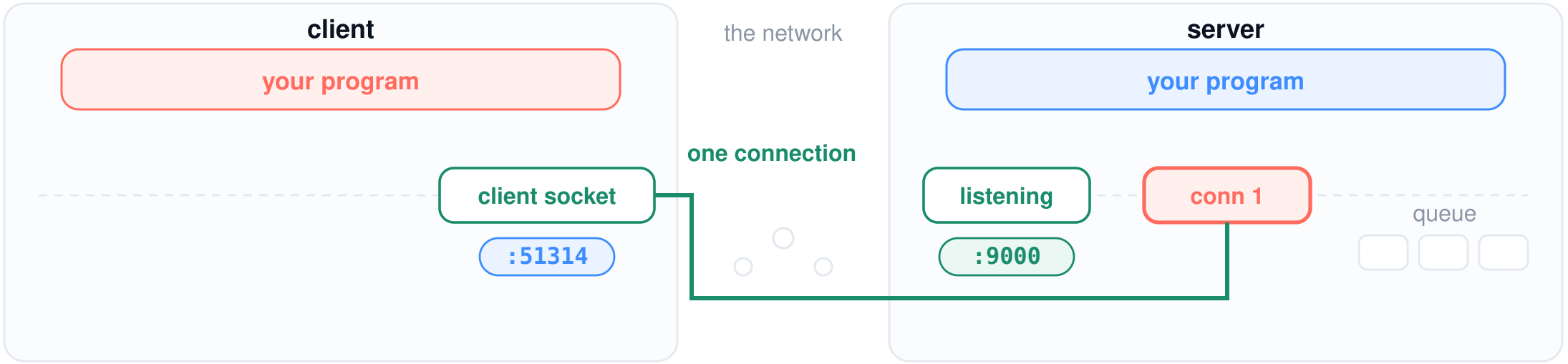
The finished connection lands in the queue



The handshake is done, so the kernel puts the connection in the queue and lets connect() return. The server program has still not run a line.

accept() can finally return.

accept() takes it out and hands back a second socket



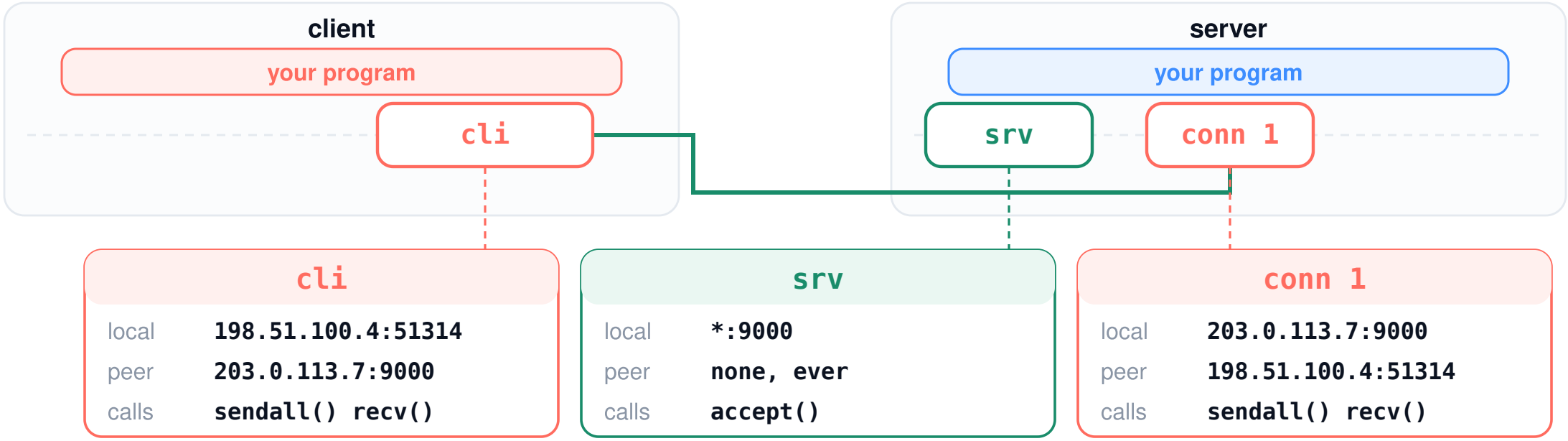

```
conn, peer = srv.accept()
```

line 8 returns

The listening socket is untouched and goes back to waiting. This connection is out of the queue for good, and all the data for this client goes through the new socket.

One listening socket, one connection socket per client.

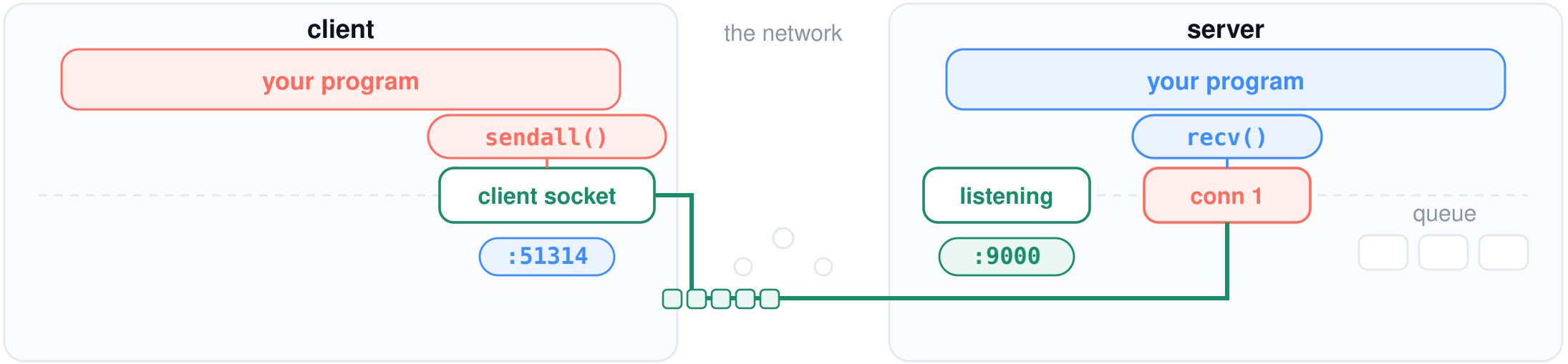
Three sockets, and what each one holds



cli and conn 1 are the two ends of one connection: the same four numbers, swapped. srv has no peer and never gets one — it only hands back new sockets.

srv accepts connections. cli and conn 1 carry data. Same type, different state.

sendall() puts the bytes in, recv() takes them out




```
cli.sendall(b"hello")  →  data = conn.recv(4096)
```

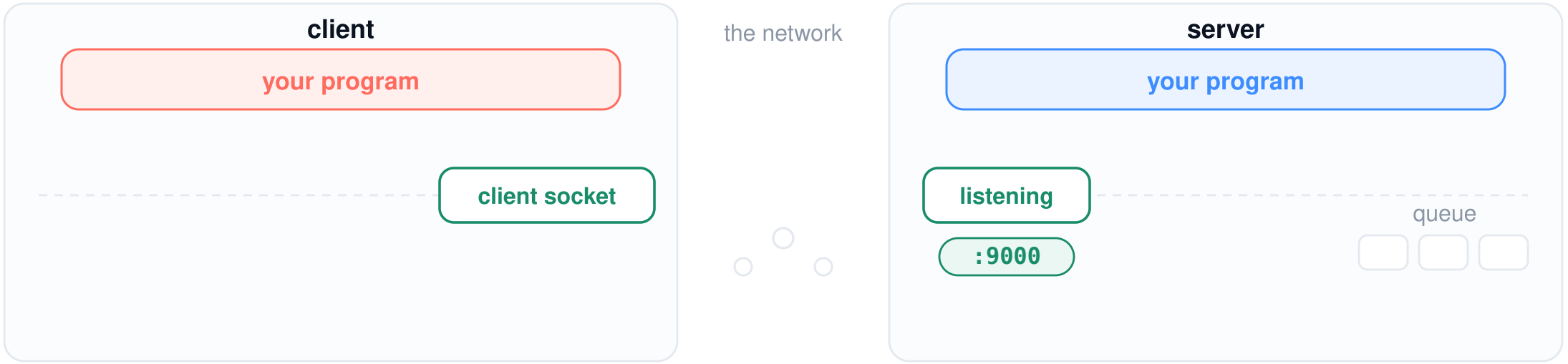
lines 5 and 9

sendall() returns once the kernel has taken the five bytes; it has not waited for the server to read them. recv() returns whatever is in the buffer, at most 4096 bytes, which is not the same as one message.

One call hands bytes to the kernel. One call takes them out at the far end.

1 The server | 2 The client | 3 The exchange | 4 Many clients | 5 Serving them all | 6 What is missing

close() removes the connection socket only



`conn.close()` line 11

conn is gone and the client's next recv() will return zero bytes. srv is unchanged: still listening, still on port 9000.

Closing srv instead of conn shuts the server down.

Each turn of the loop makes a new conn and drops it

once, before the loop

```
srv = socket(AF_INET, SOCK_STREAM)
srv.bind(("", 9000))
srv.listen(128)
```

every turn, one client each

```
while True:
    conn, peer = srv.accept()
    data = conn.recv(4096)
    conn.sendall(data.upper())
    conn.close()
```

	SRV	CONN
after listen()	listening on :9000	—
after accept()	listening on :9000	conn 1 created
recv() and sendall()	listening on :9000	conn 1 carries bytes
after close()	listening on :9000	conn 1 gone
after accept() again	listening on :9000	conn 2 created

Only the right-hand column changes. conn is a different object on every turn, and the number in the picture goes up because accept() hands back a new socket each time.

srv lives as long as the program. conn lives as long as one client.

The whole thing, now that every line has a picture

server

```python
import socket

srv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
srv.bind(("", 9000))
srv.listen(128)

while True:
    conn, peer = srv.accept()
    data = conn.recv(4096)
    conn.sendall(data.upper())
    conn.close()
```

client

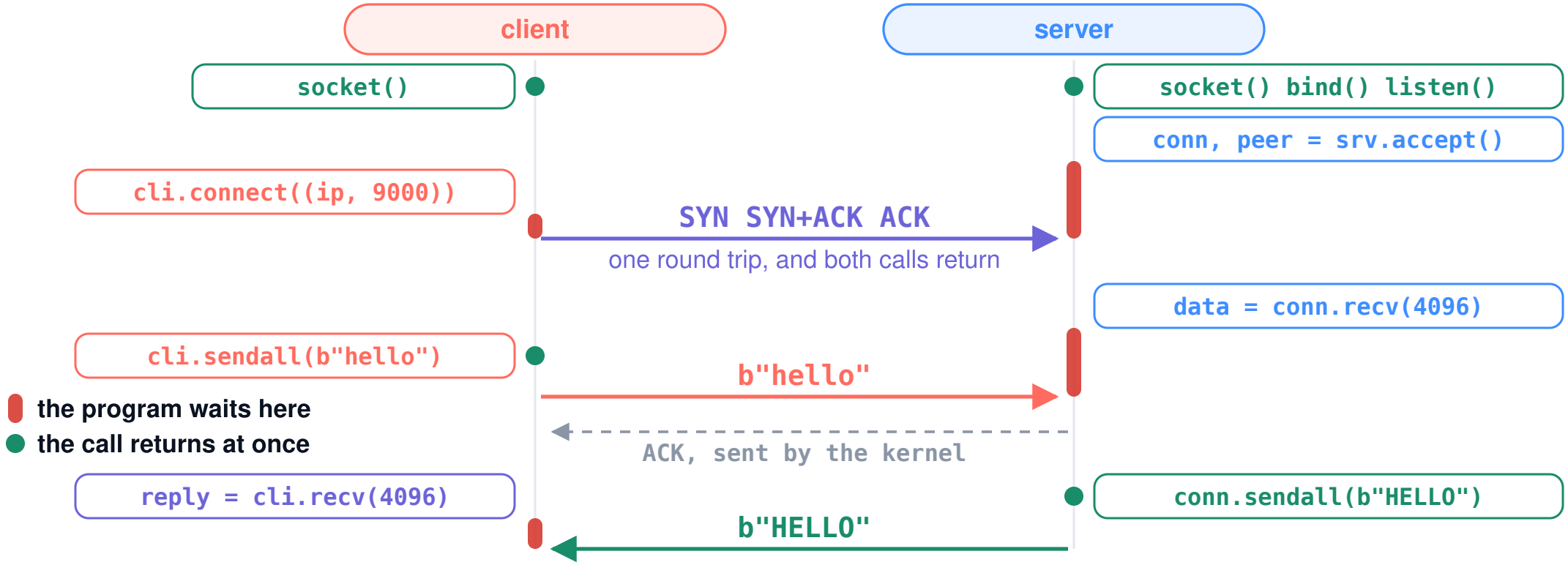
```python
import socket

cli = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
cli.connect(("203.0.113.7", 9000))
cli.sendall(b"hello")
reply = cli.recv(4096)
cli.close()
```

Eleven lines and seven lines. Nothing here has not been on screen already.

This is a working client and a working server.

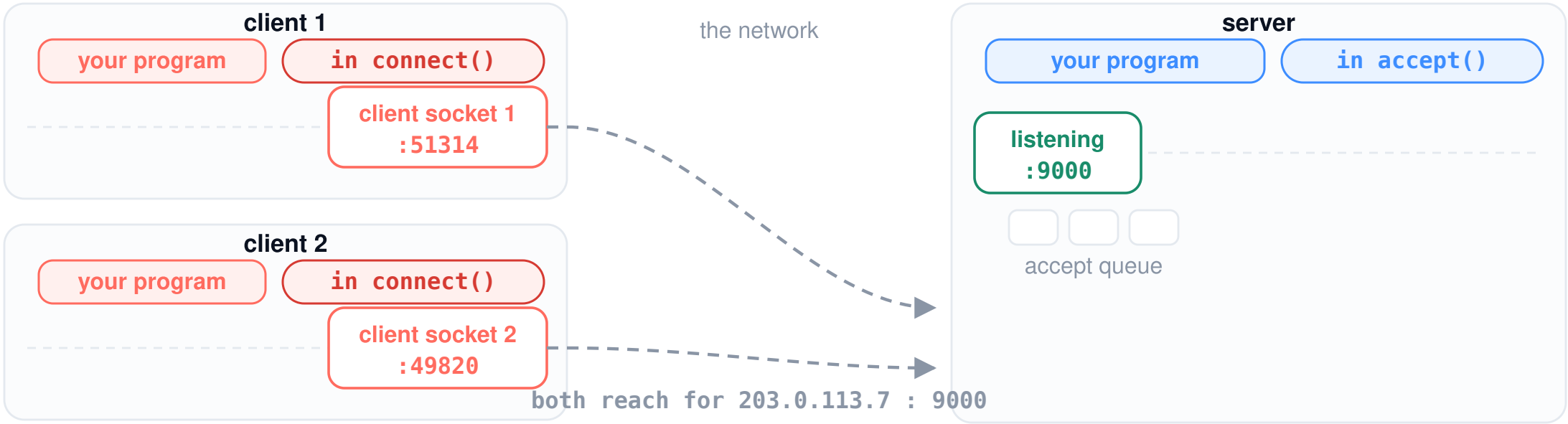
Every call in order, and the three that wait



Assume blocking sockets. Note the server enters recv() before the client sends anything, and that the ACK for the data is the kernel's work, not a line of either program.

accept(), connect() and recv() wait. The other five return at once.

Now two clients want the same server




```
cli.connect(("203.0.113.7", 9000))
```

both clients, line 4

Neither client knows the other exists. Both of them have exactly one address to aim at, and it is the same one.

Two connect() calls arrive at one listening socket.

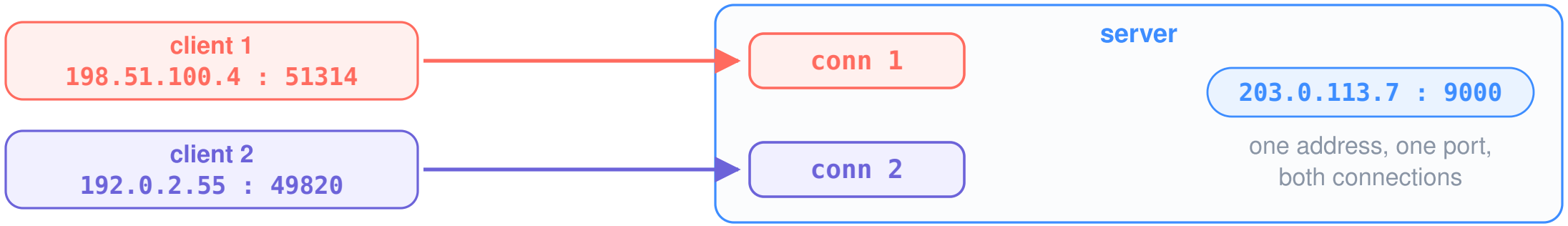
accept() returns one connection socket per client

`conn, peer = srv.accept()` line 8, once for each client

The loop comes back to accept() and calls it again, so a second connection socket appears. Look at the three green numbers: the listening socket and both connections are all on port 9000.

N clients means one listening socket and N connection sockets.

Those two connections, written out



	CLIENT IP	CLIENT PORT	SERVER IP	SERVER PORT
conn 1	198.51.100.4	51314	203.0.113.7	9000
conn 2	192.0.2.55	49820	203.0.113.7	9000

these two differ, connection by connection

these two are the same for every client

This is all there is to a connection's identity. The socket number is private to the program, and the port is shared by both, so neither can be what tells them apart.

conn 1 and conn 2 differ in two of their four numbers.

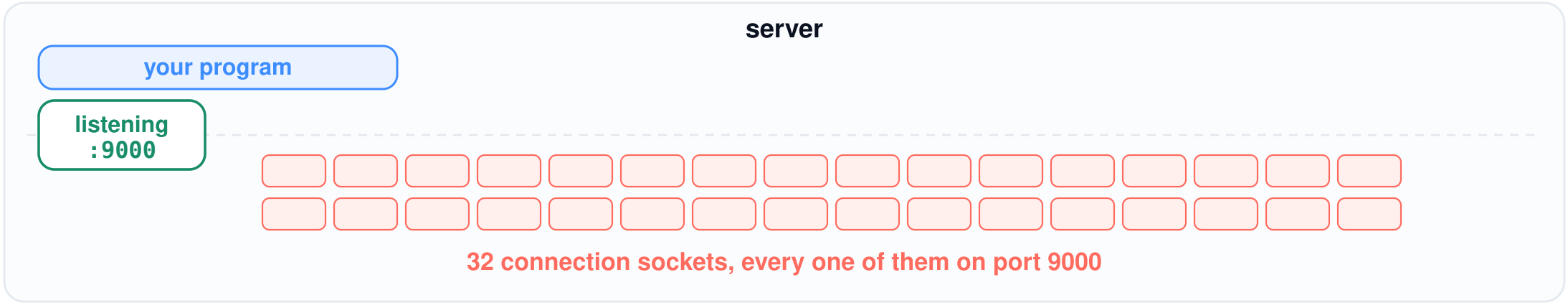
Any number of connections, named the same way

	these two differ, connection by connection		these two are the same for every client	
	CLIENT IP	CLIENT PORT	SERVER IP	SERVER PORT
conn 1	198.51.100.4	51314	203.0.113.7	9000
conn 2	192.0.2.55	49820	203.0.113.7	9000
conn 3	198.51.100.4	51862	203.0.113.7	9000

conn 3 is a third connection, from client 1's own machine: same program, same IP, a second call to connect(). Its client port is what keeps it distinct from conn 1. Every TCP segment carries all four numbers.

However many connections there are, the left-hand pair is what has to differ.

The listening socket is never the limit



what actually bounds the number

- file descriptors
- memory for the buffers
- 65,536 values in a port field

Tens of thousands of open connections on one machine is ordinary. The last box bounds one client talking to one server port — not the server, which sees many clients.

One port and one listening socket carry thousands of connections.

Nobody chose the client's port

the server's port

`bind(("", 9000))`

written into the program, and told to every client

the client's port

`no bind() call at all`

taken by the kernel during connect(), freed at close()

every port number there is, drawn to scale

Run the client twice and the number differs. The kernel holds it only for the life of one connection and then frees it.

The server's port is fixed and published. The client's is temporary.

What those three ranges actually mean

0 – 1023
system, or well-known

WHO PUTS A NAME HERE
IANA, under its strictest process

CAN YOU BIND ONE?
only with privilege — root, or `CAP_NET_BIND_SERVICE`

FOR INSTANCE
80 http **443** https **22** ssh

1024 – 49151
user, or registered

WHO PUTS A NAME HERE
IANA writes one down if you ask it to

CAN YOU BIND ONE?
yes. Any program, no permission, registered or not

FOR INSTANCE
3306 mysql **5432** postgres

49152 – 65535
dynamic, or private

WHO PUTS A NAME HERE
nobody. IANA never assigns one of these

CAN YOU BIND ONE?
yes — but the kernel is handing them out, so expect collisions

FOR INSTANCE
whatever `connect()` was given

And a kernel's ephemeral range is its own setting, not IANA's: Linux defaults to 32768–60999, which reaches down into the registered block.

The registry records names. The only rule the kernel enforces is the first card.

A connection is state in two kernels, not a wire

client kernel

ONE CONNECTION

198.51.100.4:51314 → 203.0.113.7:9000

next byte to send · next byte expected

how much the other side can still take

a send buffer · a receive buffer

a retransmission timer

the network

routers forward packets
and keep none of this

server kernel

THE SAME CONNECTION

198.51.100.4:51314 → 203.0.113.7:9000

next byte to send · next byte expected

how much the other side can still take

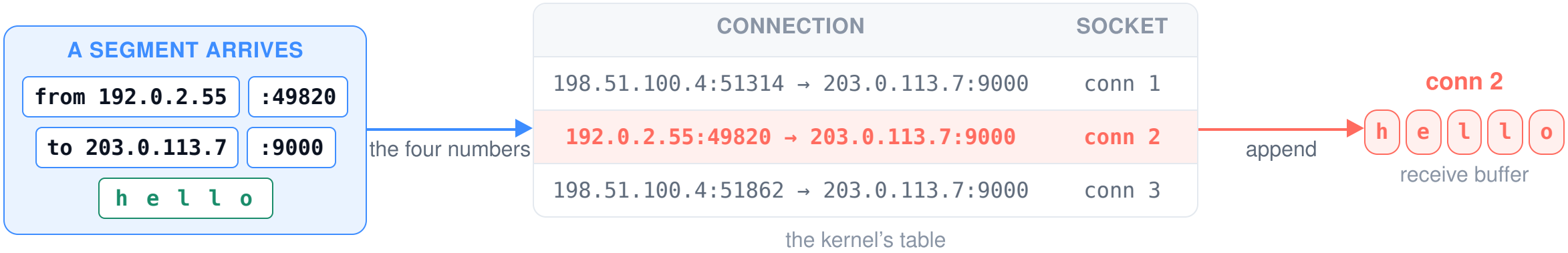
a send buffer · a receive buffer

a retransmission timer

"Established" means both of these exist. A NAT or a stateful firewall in the path keeps a table of its own, but no router has to.

The connection is this state. If either end loses it, the connection ends.

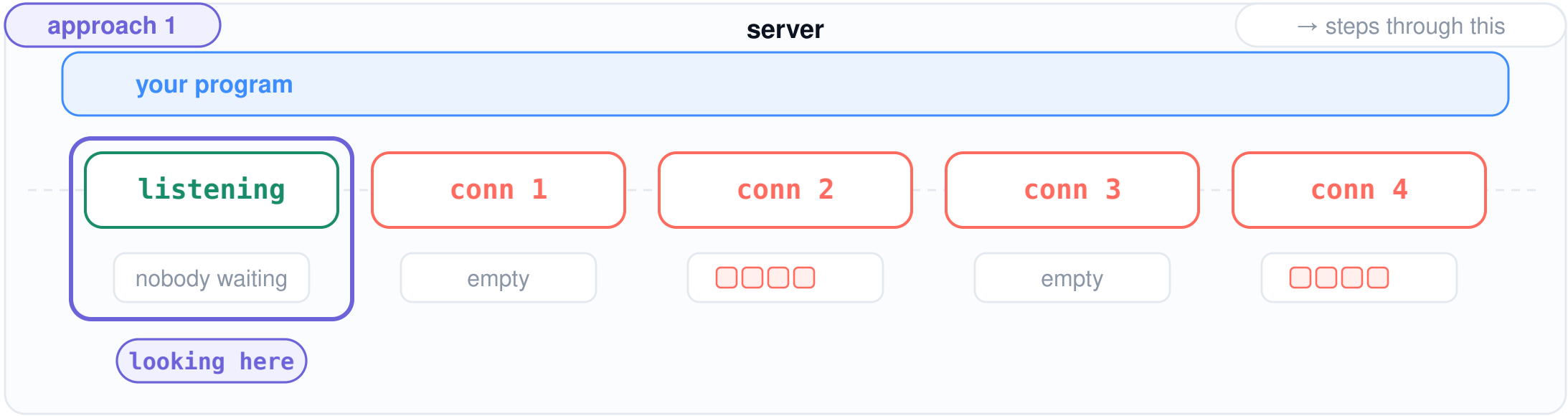
How arriving bytes find the right socket



Every segment gets looked up. The socket number your program holds is a handle onto one row of this table, and recv() drains that row's buffer.

The kernel maps four numbers to one receive buffer. One port therefore serves thousands.

How does one program serve many connections?

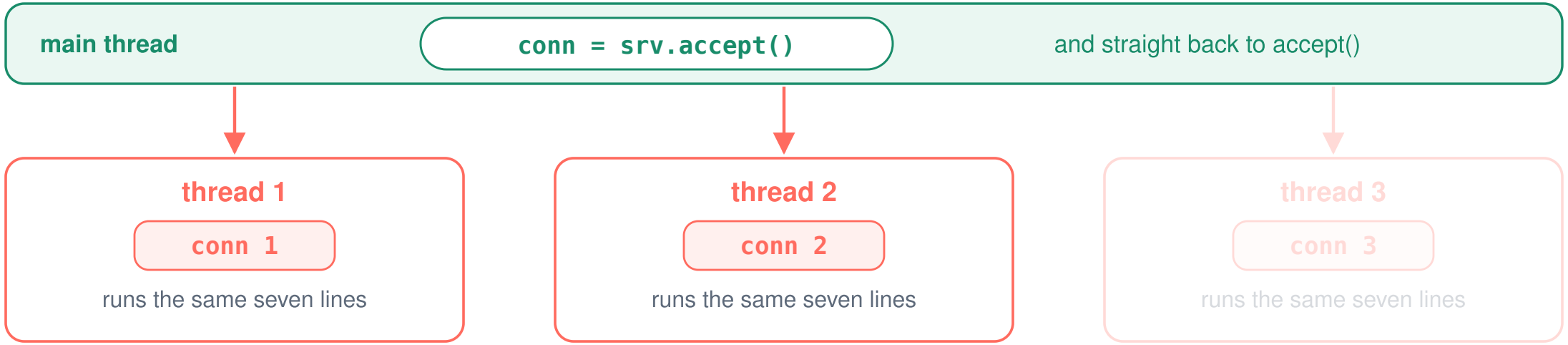


1 / 8 · the listening socket: nobody is waiting to connect

Every socket has to be non-blocking, or a recv() on an empty conn 1 would stop the lap. And taking the bytes out is only half of it: handling the request costs time too, and how much depends on what the request asks for.

A request that needs 40 ms of work delays every other connection by 40 ms.

Approach 2: the main thread only accepts



one accept() returns, one thread is created for it

The main thread never reads or writes a connection. It calls accept(), hands the socket that comes back to a thread of its own, and goes round to accept() again.

One accept() per client, and one thread per client.

Each thread reads and writes its own socket

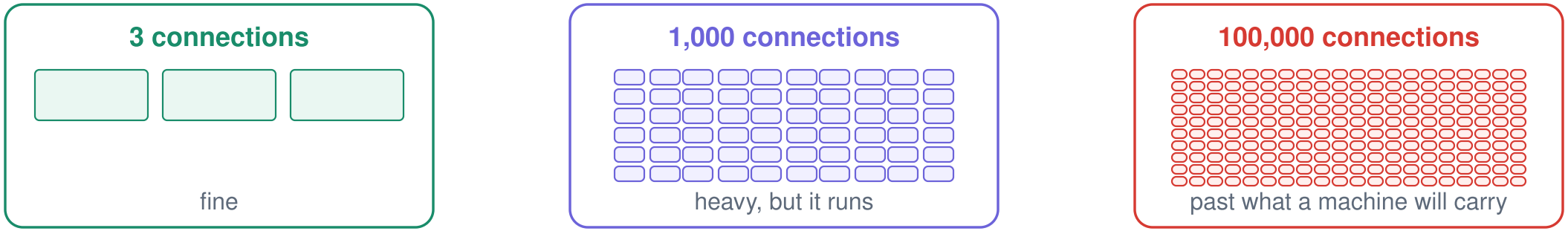
the request: recv() takes it up, and the work uses it up

the reply: different bytes, made by the work, sent back down

The request does not come back out. recv() takes it up, the work consumes it, and what goes down is a reply the work produced — or nothing at all, if the work wrote to a database or a file instead.

Three requests in progress at once, each at its own speed.

What a thread for every connection costs



- a stack of its own for every thread — 8 MB of address space by default on Linux, of which only the pages it touches are real memory
- the scheduler has to hold every one of them and switch between them
- anything two threads share needs a lock

A thread pool caps the number, and then the size of the pool is the ceiling on how many clients can be in progress at once.

The code stays simple. The cost grows with the number of connections.

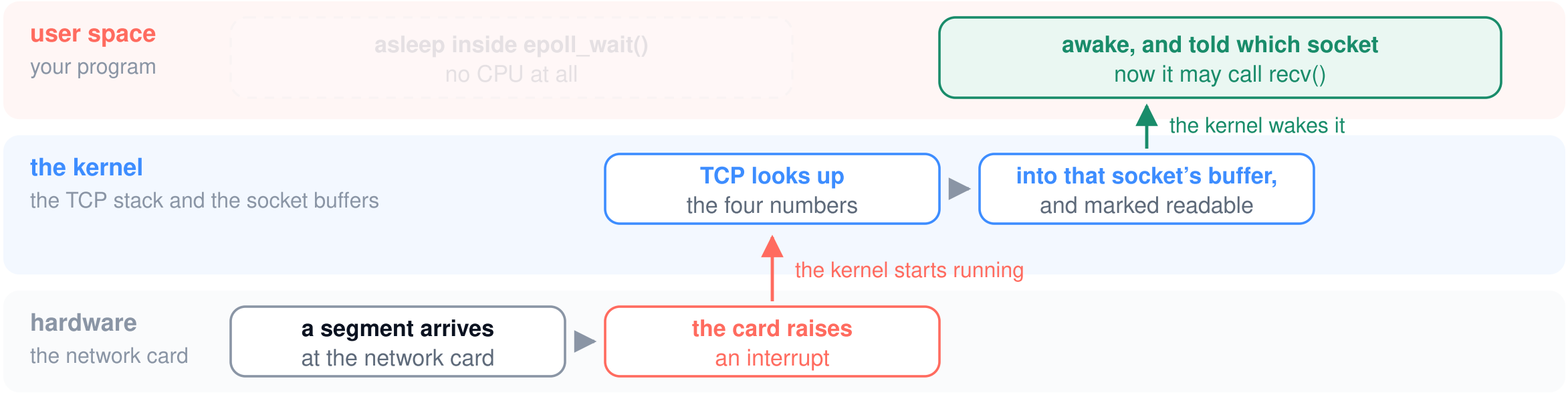
Approach 3: ask the kernel which sockets are ready

select() and poll() do the same job; epoll and kqueue are the ones that stay cheap at thousands

This is approach 1's lap, walked by the kernel instead of by your program: one call instead of one per socket, and it does not return until at least one socket has something. No recv() the thread then makes can block.

One thread serves every ready connection and waits on none of them. This is an event loop.

Hardware, kernel, and where your program is



Nothing in the top band runs while the three lower boxes happen: the program holds no CPU at all. Linux takes one interrupt and then polls the card for a batch of packets rather than one interrupt per packet, but the shape is this.

The interrupt starts it, the kernel does the work, and the program is woken last.

The three approaches, side by side

	1 you walk the list	2 a thread per connection	3 ask the kernel
WHAT WAITS	nothing — the lap never stops	each thread, in its own recv()	one epoll call, on all of them
SYSTEM CALLS PER LAP	one for every socket	one per thread, and it blocks	one, for all of them
WHILE NOTHING ARRIVES	the program keeps asking	every thread is parked	the program is asleep, no CPU
COST PER CONNECTION	one entry in your own list	a thread with its own stack	one entry in a kernel set
HOW SOON IT NOTICES	next time the lap comes round	as soon as the kernel wakes it	as soon as the kernel wakes it
YOU HAVE MET IT IN	teaching code	Apache’s prefork model	nginx, Node.js, Redis

select() and poll() sit between 1 and 3: one call, but the whole set of sockets is handed in and scanned every time, so the work still grows with the count. epoll and kqueue keep the set in the kernel.

socket, bind, listen, accept, connect, send, recv, close — the same eight calls in all three.

TCP handed us bytes, not our messages

Every byte arrived, in order, exactly once — with no record of the three calls.

Marking where a message ends is the application's job.

1 The server | 2 The client | 3 The exchange | 4 Many clients | 5 Serving them all | 6 What is missing

And that was only the transport half

The server still needs

- error handling: a refused connection, a reset, a client that vanishes
- timeouts, so one silent client cannot hold a socket forever
- back-pressure, when a client reads slower than you write
- TLS, if anything on the wire matters

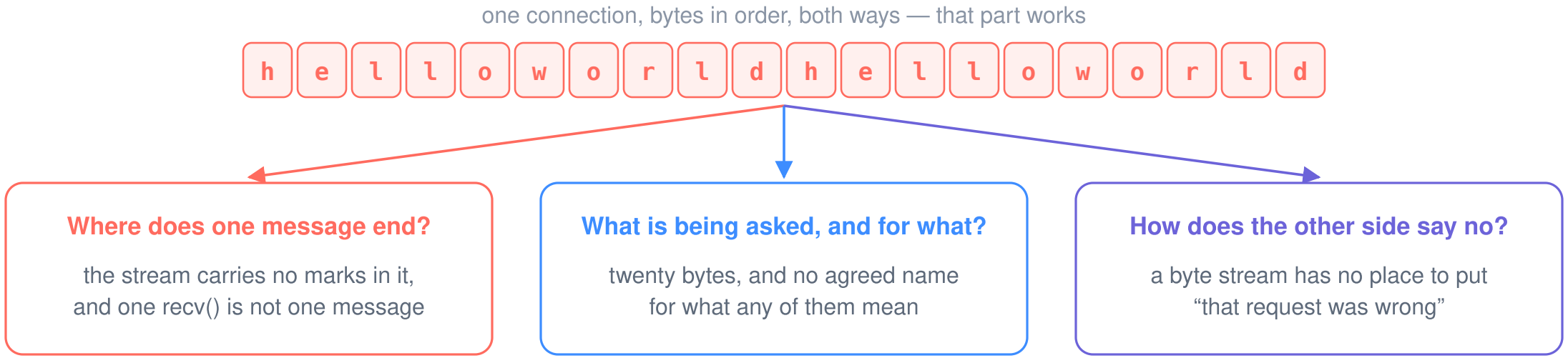
The application still needs

- a message format both ends already know
- a rule for where each message ends
- a way to report that a request was wrong
- a way to tell one client from another across connections

UDP sockets exist too — sendto() and recvfrom(), and no connection at all.

The left column is engineering work. The right column needs a protocol.

Three questions the socket interface does not answer



every pair of programs has to answer these three somehow

None of the three is a transport problem. The bytes arrive in order. What is missing is an agreement about what they mean.

Part 2 is HTTP, the agreement the Web uses.

1 The server | 2 The client | 3 The exchange | 4 Many clients | 5 Serving them all | 6 What is missing