Socket migration for SO_REUSEPORT

Kuniyuki Iwashima

Cloud Support Engineer

Agenda

- What is SO_REUSEPORT?
- When SO_REUSEPORT misbehaves
- Where SO_REUSEPORT misbehaves
- How to make it acceptable

What is SO_REUSEPORT?

SO_REUSEPORT

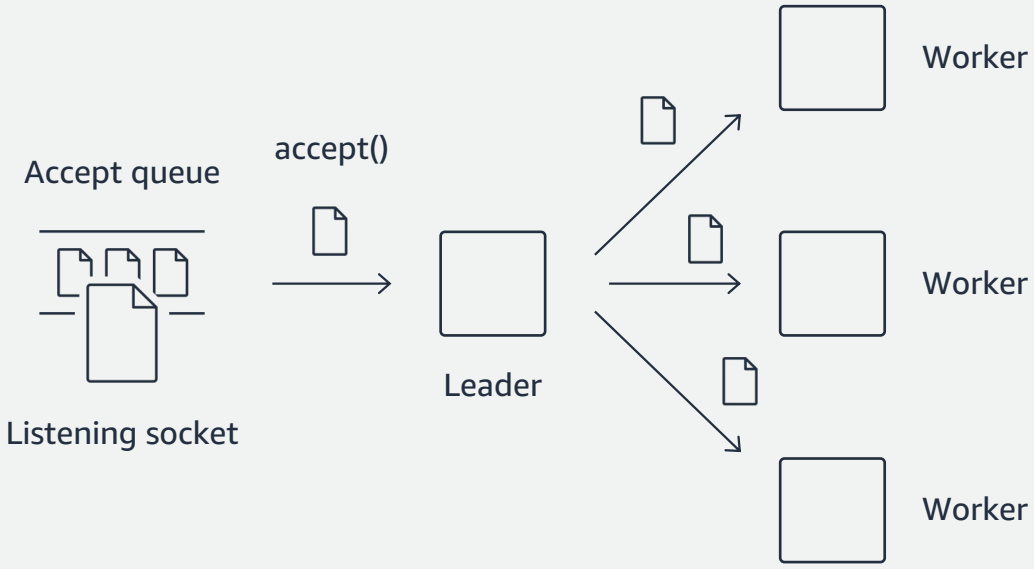
- Added in v3.9 for high-performance servers

Without SO_REUSEPORT

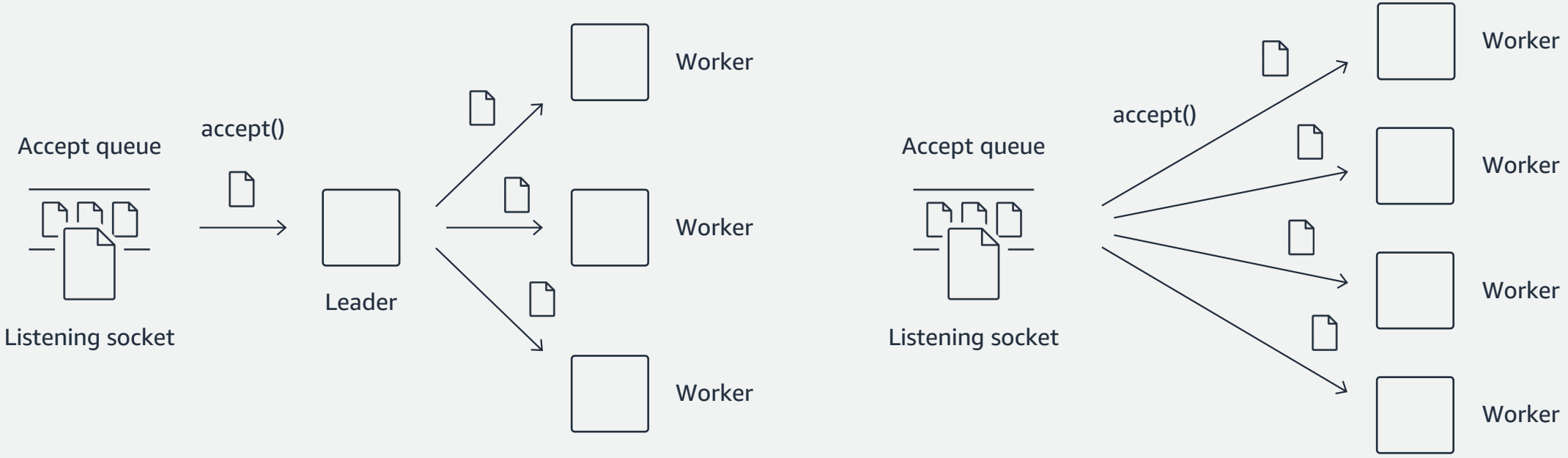
- Only one socket is allowed to listen() on a given TCP port
- bind()/listen() on the same port fail with EADDRINUSE

```
$ sudo python3
>>> from socket import *
>>>
>>> def get_server():
...         sk = socket(AF_INET, SOCK_STREAM, 0)
...         sk.bind(('localhost', 80))
...         sk.listen(32)
...         return sk
...
>>> server_1 = get_server()
>>> server_2 = get_server()
...
OSError: [Errno 98] Address already in use
```

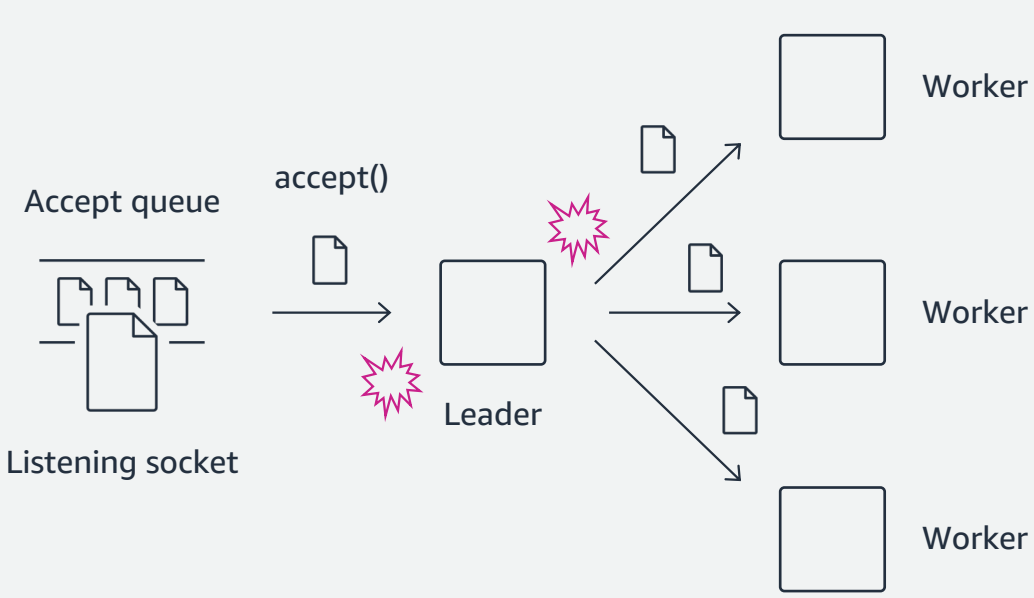
Traditional approaches without SO_REUSEPORT

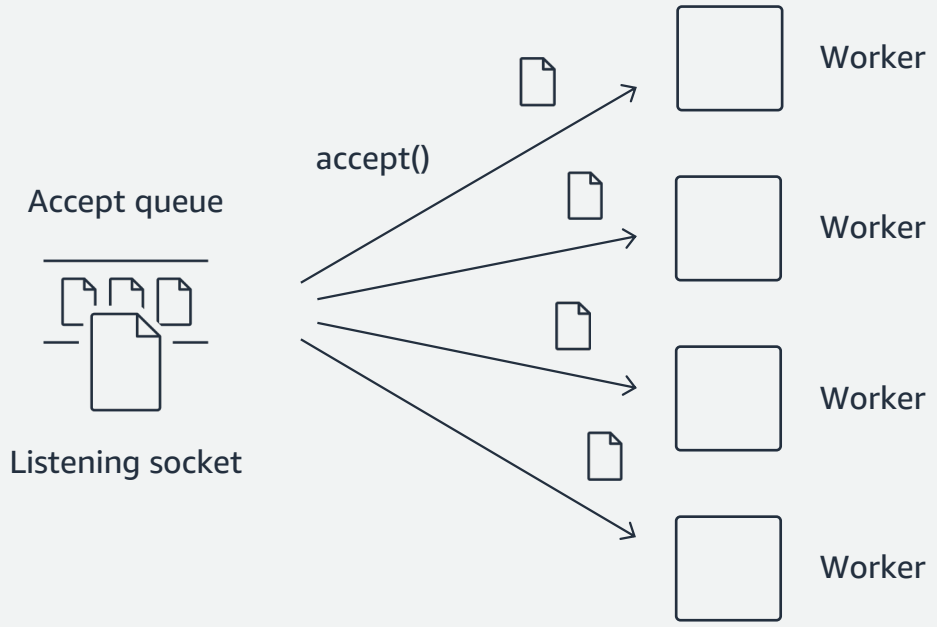


Traditional approaches without SO_REUSEPORT

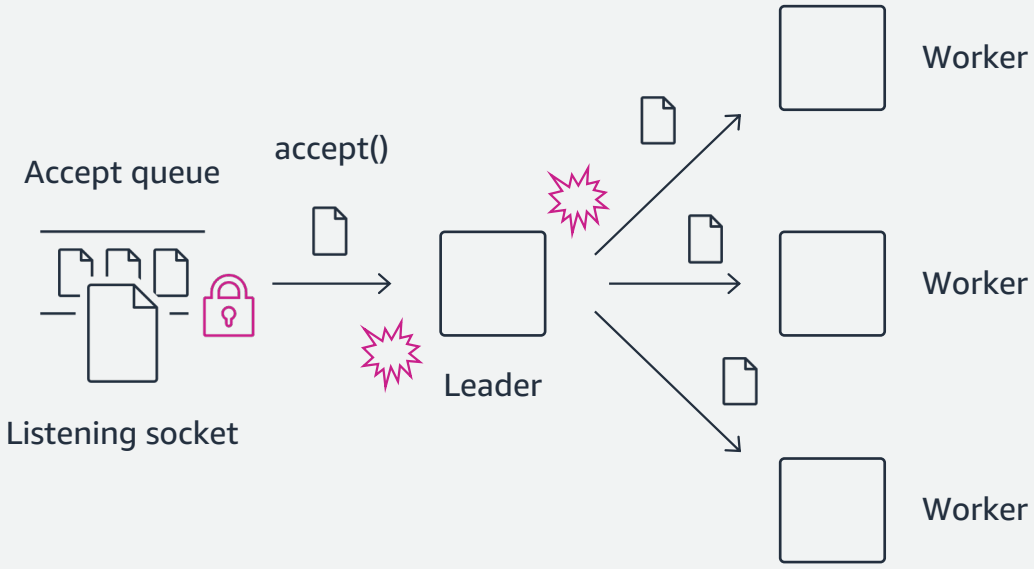


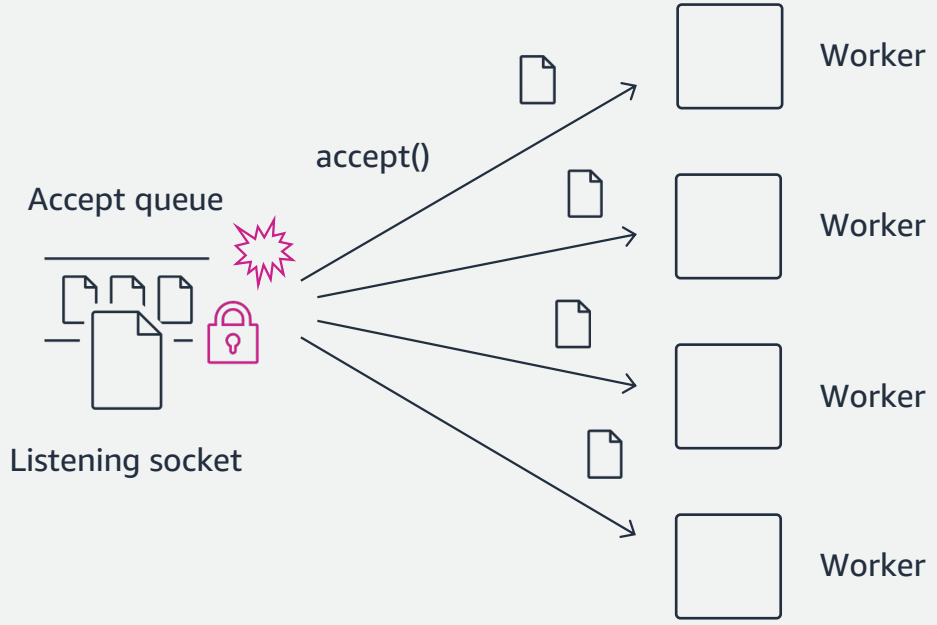
Traditional approaches without SO_REUSEPORT



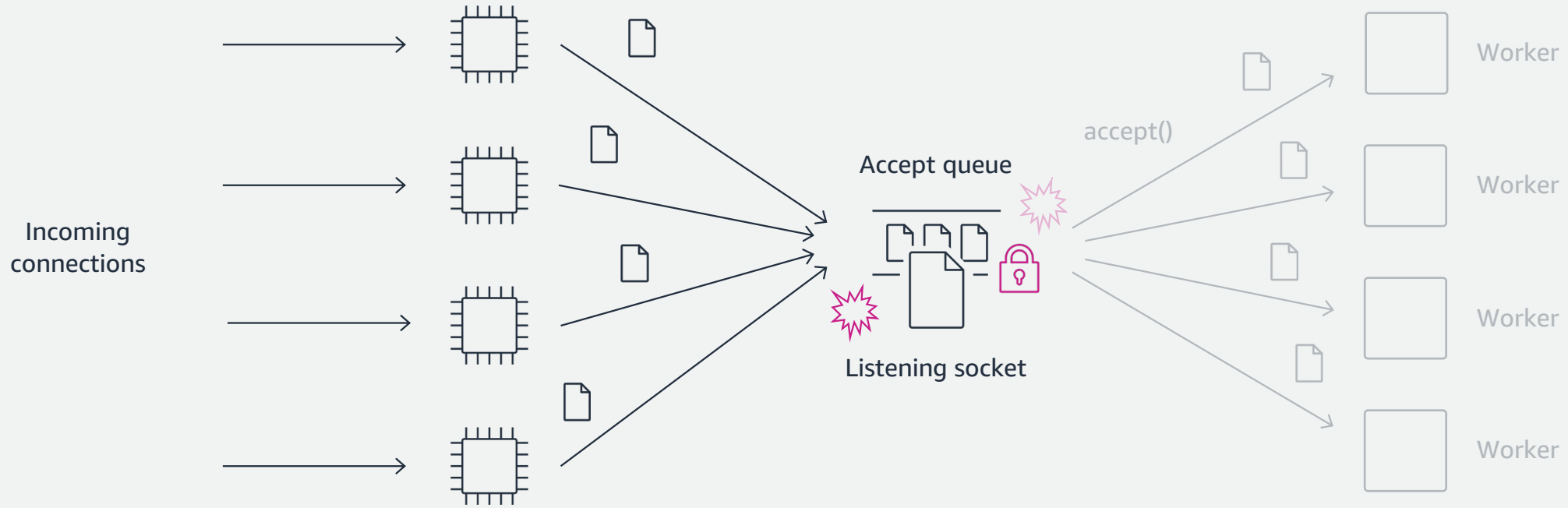


Traditional approaches without SO_REUSEPORT

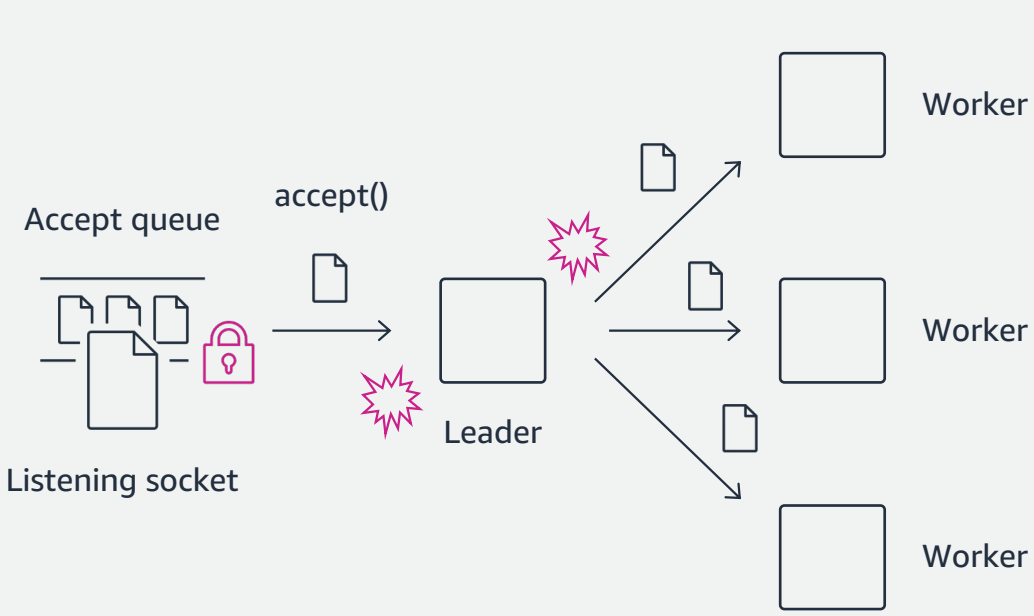


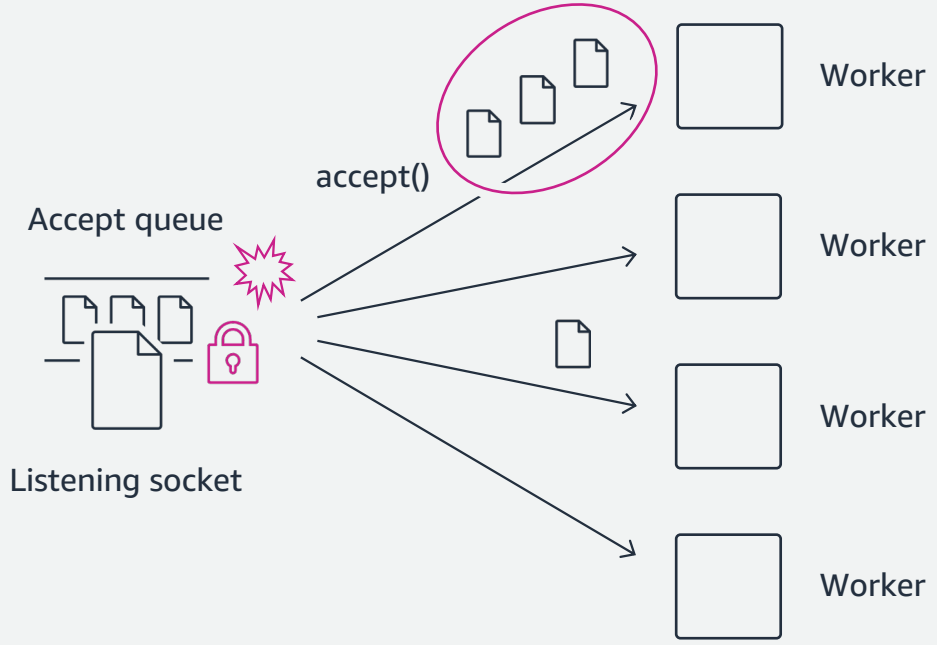


Traditional approaches without SO_REUSEPORT



Traditional approaches without SO_REUSEPORT



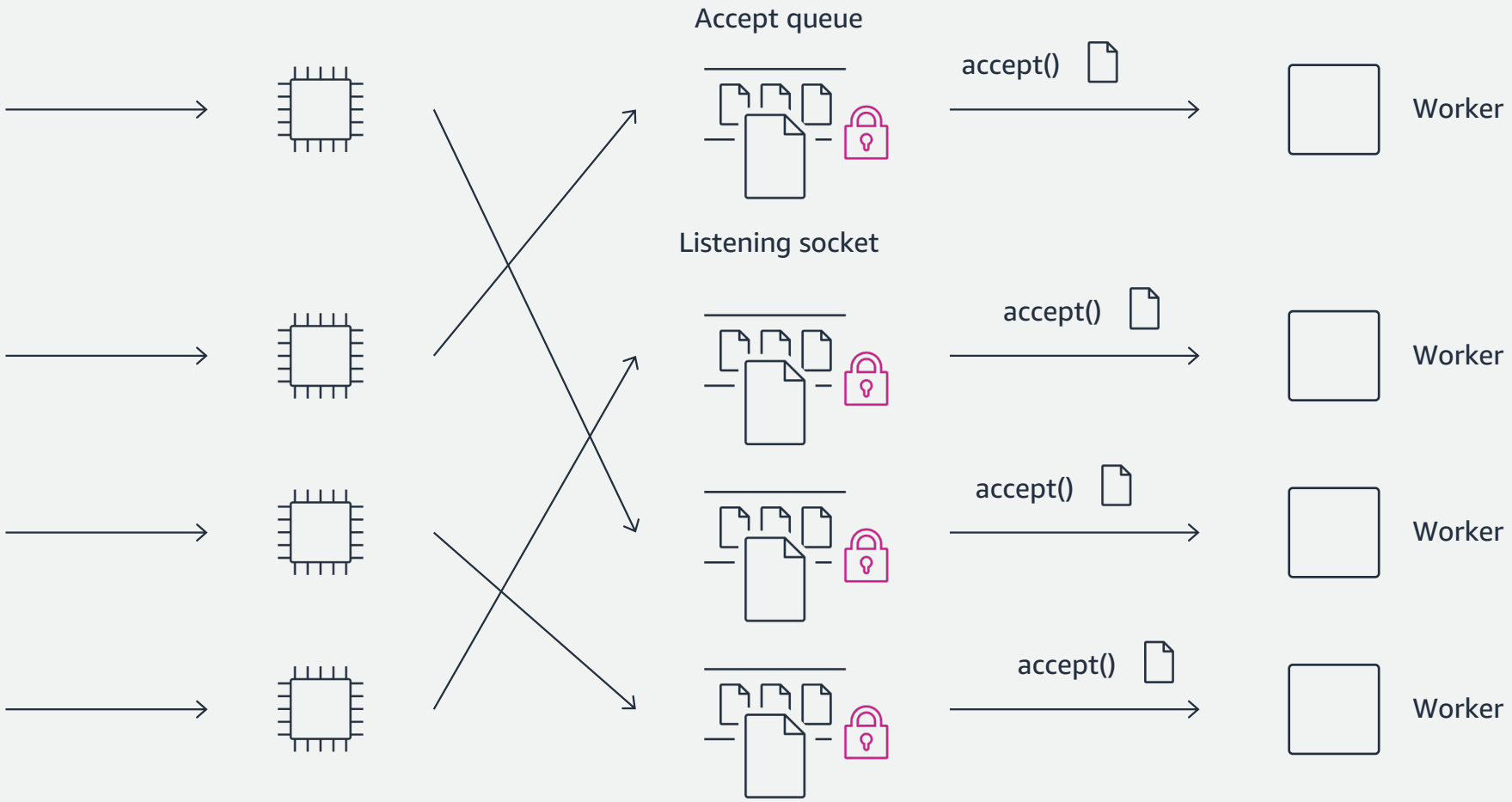


SO_REUSEPORT

- Added in v3.9 for high-performance servers
- Multiple sockets are allowed to listen() on the same port

```
$ sudo python3
>>> from socket import *
>>>
>>> def get_reuseport_server():
...         sk = socket(AF_INET, SOCK_STREAM, 0)
...         sk.setsockopt(SOL_SOCKET, SO_REUSEPORT, 1)
...         sk.bind(('localhost', 80))
...         sk.listen(32)
...         return sk
...
>>> server_1 = get_reuseport_server()
>>> server_2 = get_reuseport_server()
>>> # No error here
```

SO_REUSEPORT



SO_REUSEPORT

- Added in v3.9 for high-performance servers
- Multiple sockets are allowed to listen() on the same port
  - Address the accept() bottleneck
  - Distribute connections almost evenly (randomly)

"The TCP implementation has a problem"

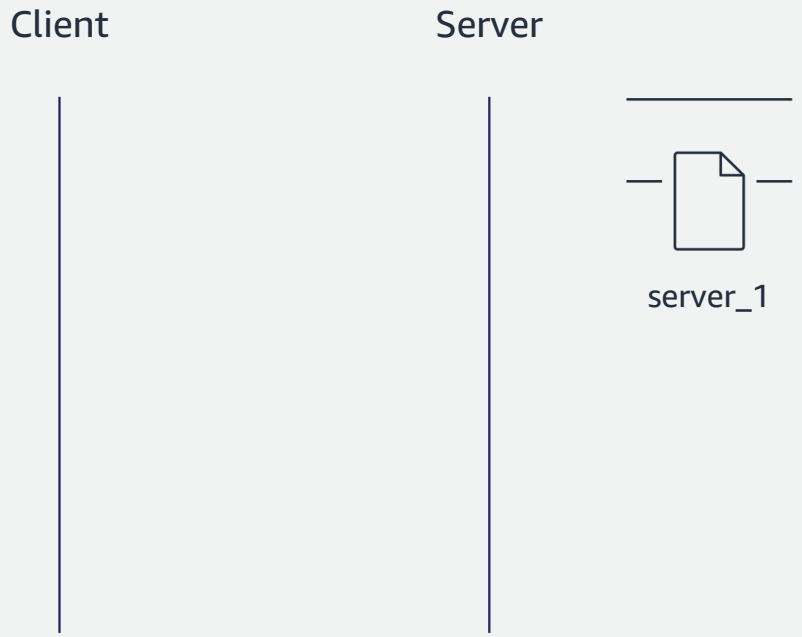
(c617f398edd4)

When SO_REUSEPORT misbehaves

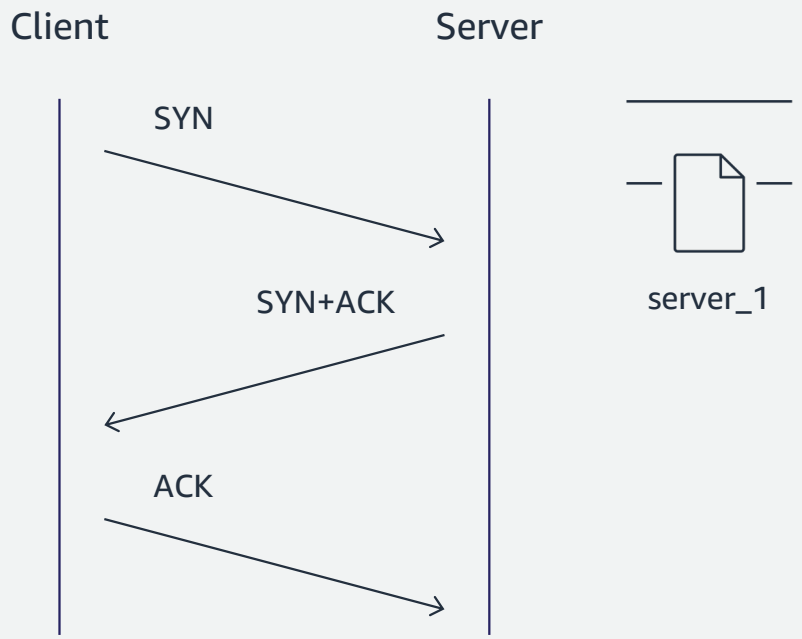
Quiz 1

```
from socket import *

def get_reuseport_server():
    s = socket(AF_INET, SOCK_STREAM, 0)
    s.setsockopt(SOL_SOCKET, SO_REUSEPORT, 1)
    s.bind(("localhost", 80))
    s.listen(32)
    return s

def get_client():
    c = socket(AF_INET, SOCK_STREAM, 0)
    c.connect(("localhost", 80))
    return c

def quiz1():
    server_1 = get_reuseport_server()

    client = get_client()
    client.send(b'Hello World')

    server_2 = get_reuseport_server()

    server_1.close()

    server_2.setblocking(0)
    child, _ = server_2.accept()
    print(child.recv(1024))

if __name__ == '__main__':
    quiz1()
```

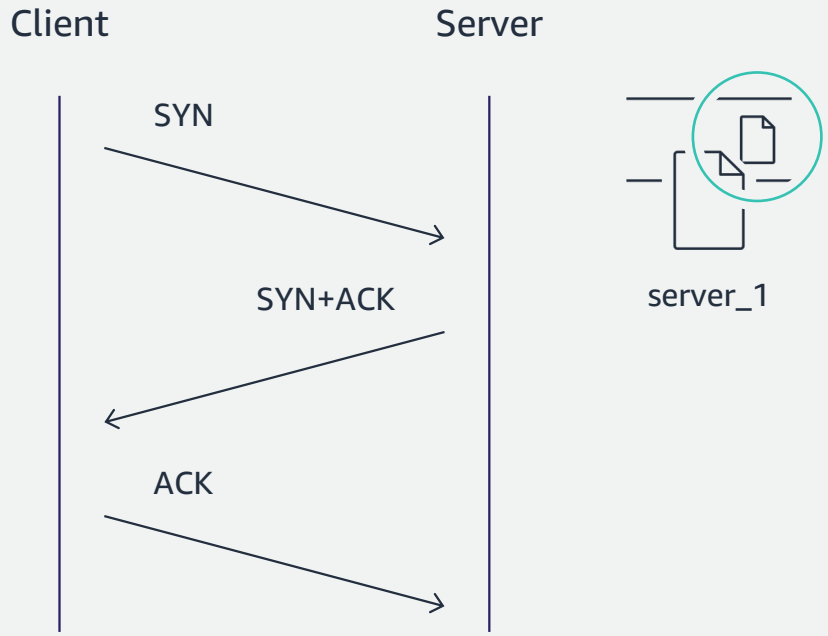
```
from socket import *

def get_reuseport_server():
        s = socket(AF_INET, SOCK_STREAM, 0)
        s.setsockopt(SOL_SOCKET, SO_REUSEPORT, 1)
        s.bind(("localhost", 80))
        s.listen(32)
        return s

def get_client():
        c = socket(AF_INET, SOCK_STREAM, 0)
        c.connect(("localhost", 80))
        return c

def quiz1():
        server_1 = get_reuseport_server()

        client = get_client()
        client.send(b'Hello World')

        server_2 = get_reuseport_server()

        server_1.close()

        server_2.setblocking(0)
        child, _ = server_2.accept()
        print(child.recv(1024))

if __name__ == '__main__':
        quiz1()
```



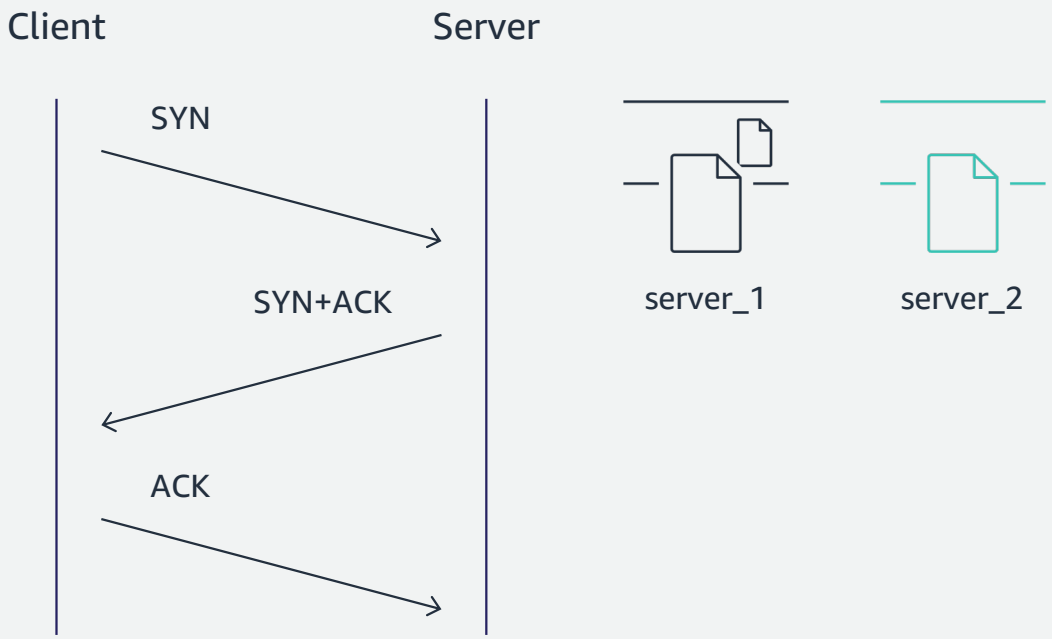
```
from socket import *

def get_reuseport_server():
    s = socket(AF_INET, SOCK_STREAM, 0)
    s.setsockopt(SOL_SOCKET, SO_REUSEPORT, 1)
    s.bind(("localhost", 80))
    s.listen(32)
    return s

def get_client():
    c = socket(AF_INET, SOCK_STREAM, 0)
    c.connect(("localhost", 80))
    return c

def quiz1():
    server_1 = get_reuseport_server()

    client = get_client()
    client.send(b'Hello World')

    server_2 = get_reuseport_server()

    server_1.close()

    server_2.setblocking(0)
    child, _ = server_2.accept()
    print(child.recv(1024))

if __name__ == '__main__':
    quiz1()
```



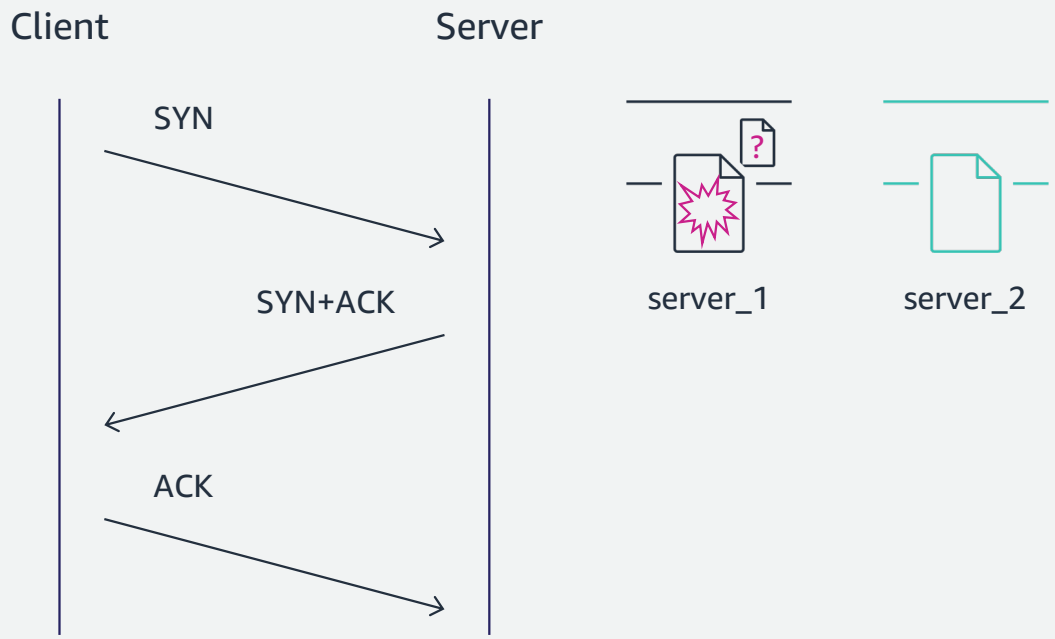
```
from socket import *

def get_reuseport_server():
    s = socket(AF_INET, SOCK_STREAM, 0)
    s.setsockopt(SOL_SOCKET, SO_REUSEPORT, 1)
    s.bind(("localhost", 80))
    s.listen(32)
    return s

def get_client():
    c = socket(AF_INET, SOCK_STREAM, 0)
    c.connect(("localhost", 80))
    return c

def quiz1():
    server_1 = get_reuseport_server()

    client = get_client()
    client.send(b'Hello World')

    server_2 = get_reuseport_server()

    server_1.close()

    server_2.setblocking(0)
    child, _ = server_2.accept()
    print(child.recv(1024))

if __name__ == '__main__':
    quiz1()
```



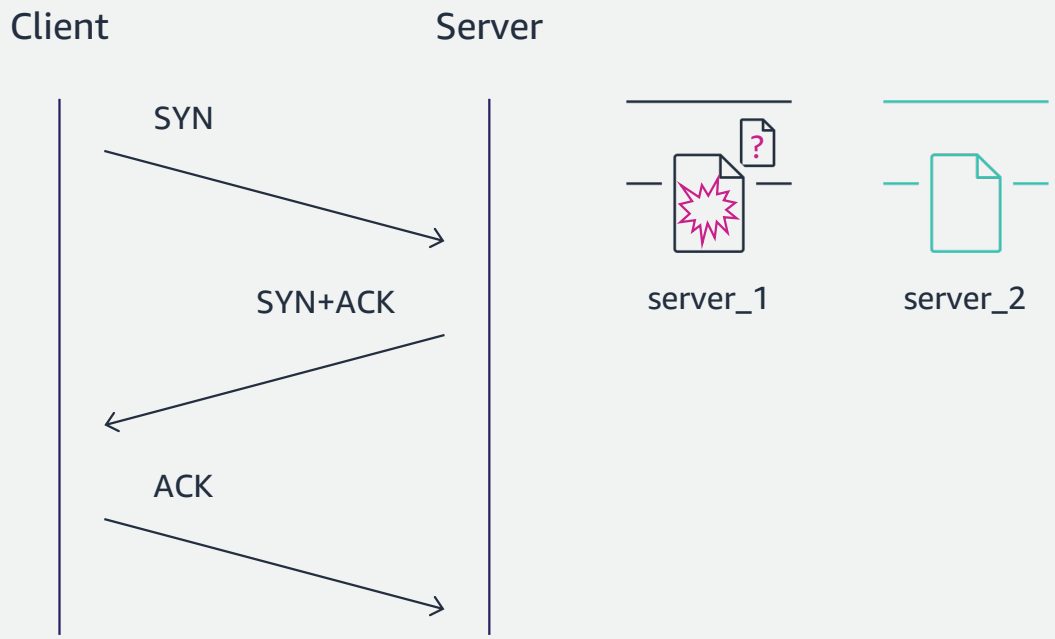
```
from socket import *

def get_reuseport_server():
    s = socket(AF_INET, SOCK_STREAM, 0)
    s.setsockopt(SOL_SOCKET, SO_REUSEPORT, 1)
    s.bind(("localhost", 80))
    s.listen(32)
    return s

def get_client():
    c = socket(AF_INET, SOCK_STREAM, 0)
    c.connect(("localhost", 80))
    return c

def quiz1():
    server_1 = get_reuseport_server()

    client = get_client()
    client.send(b'Hello World')

    server_2 = get_reuseport_server()

    server_1.close()

    server_2.setblocking(0)
    child, _ = server_2.accept()
    print(child.recv(1024))

if __name__ == '__main__':
    quiz1()
```



```
from socket import *

def get_reuseport_server():
    s = socket(AF_INET, SOCK_STREAM, 0)
    s.setsockopt(SOL_SOCKET, SO_REUSEPORT, 1)
    s.bind(("localhost", 80))
    s.listen(32)
    return s

def get_client():
    c = socket(AF_INET, SOCK_STREAM, 0)
    c.connect(("localhost", 80))
    return c

def quiz1():
    server_1 = get_reuseport_server()

    client = get_client()
    client.send(b'Hello World')

    server_2 = get_reuseport_server()

    server_1.close()

    server_2.setblocking(0)
    child, _ = server_2.accept()
    print(child.recv(1024))

if __name__ == '__main__':
    quiz1()
```



```
from socket import *

def get_reuseport_server():
    s = socket(AF_INET, SOCK_STREAM, 0)
    s.setsockopt(SOL_SOCKET, SO_REUSEPORT, 1)
    s.bind(("localhost", 80))
    s.listen(32)
    return s

def get_client():
    c = socket(AF_INET, SOCK_STREAM, 0)
    c.connect(("localhost", 80))
    return c

def quiz1():
    server_1 = get_reuseport_server()

    client = get_client()
    client.send(b'Hello World')

    server_2 = get_reuseport_server()

    server_1.close()

    server_2.setblocking(0)
    child, _ = server_2.accept()
    print(child.recv(1024))

if __name__ == '__main__':
    quiz1()
```



```
from socket import *

def get_reuseport_server():
    s = socket(AF_INET, SOCK_STREAM, 0)
    s.setsockopt(SOL_SOCKET, SO_REUSEPORT, 1)
    s.bind(("localhost", 80))
    s.listen(32)
    return s

def get_client():
    c = socket(AF_INET, SOCK_STREAM, 0)
    c.connect(("localhost", 80))
    return c

def quiz1():
    server_1 = get_reuseport_server()

    client = get_client()
    client.send(b'Hello World')

    server_2 = get_reuseport_server()

    server_1.close()

    server_2.setblocking(0)
    child, _ = server_2.accept()
    print(child.recv(1024))

if __name__ == '__main__':
    quiz1()
```

Non-optimal

```
$ sudo python3 quiz1.py
...
   child, _ = server_2.accept()
...
BlockingIOError: [Errno 11] Resource temporarily unavailable
```

Optimal

```
$ sudo python3 quiz1.py
b'Hello World'
```

```
from socket import *

def get_reuseport_server():
    s = socket(AF_INET, SOCK_STREAM, 0)
    s.setsockopt(SOL_SOCKET, SO_REUSEPORT, 1)
    s.bind(("localhost", 80))
    s.listen(32)
    return s

def get_client():
    c = socket(AF_INET, SOCK_STREAM, 0)
    c.connect(("localhost", 80))
    return c

def quiz1():
    server_1 = get_reuseport_server()

    client = get_client()
    client.send(b'Hello World')

    server_2 = get_reuseport_server()

    server_1.close()

    server_2.setblocking(0)
    child, _ = server_2.accept()
    print(child.recv(1024))

if __name__ == '__main__':
    quiz1()
```

Non-optimal

```
$ sudo python3 quiz1.py
...
   child, _ = server_2.accept()
...
BlockingIOError: [Errno 11] Resource temporarily unavailable
```

```
$ man 3 errno
...
       EAGAIN
              Resource temporarily unavailable
              (may be the same value as EWOULDBLOCK) (POSIX.1)
```

```
$ man 2 accept
...
       EAGAIN or EWOULDBLOCK
              The socket is marked nonblocking and
              no connections are present to be accepted.
```

```
from socket import *

def get_reuseport_server():
    s = socket(AF_INET, SOCK_STREAM, 0)
    s.setsockopt(SOL_SOCKET, SO_REUSEPORT, 1)
    s.bind(("localhost", 80))
    s.listen(32)
    return s

def get_client():
    c = socket(AF_INET, SOCK_STREAM, 0)
    c.connect(("localhost", 80))
    return c

def quiz1():
    server_1 = get_reuseport_server()

    client = get_client()
    client.send(b'Hello World')

    server_2 = get_reuseport_server()

    server_1.close()

    server_2.setblocking(0)
    child, _ = server_2.accept()
    print(child.recv(1024))

if __name__ == '__main__':
    quiz1()
```

Non-optimal

```
$ sudo python3 quiz1.py
...
   child, _ = server_2.accept()
...
BlockingIOError: [Errno 11] Resource temporarily unavailable
```

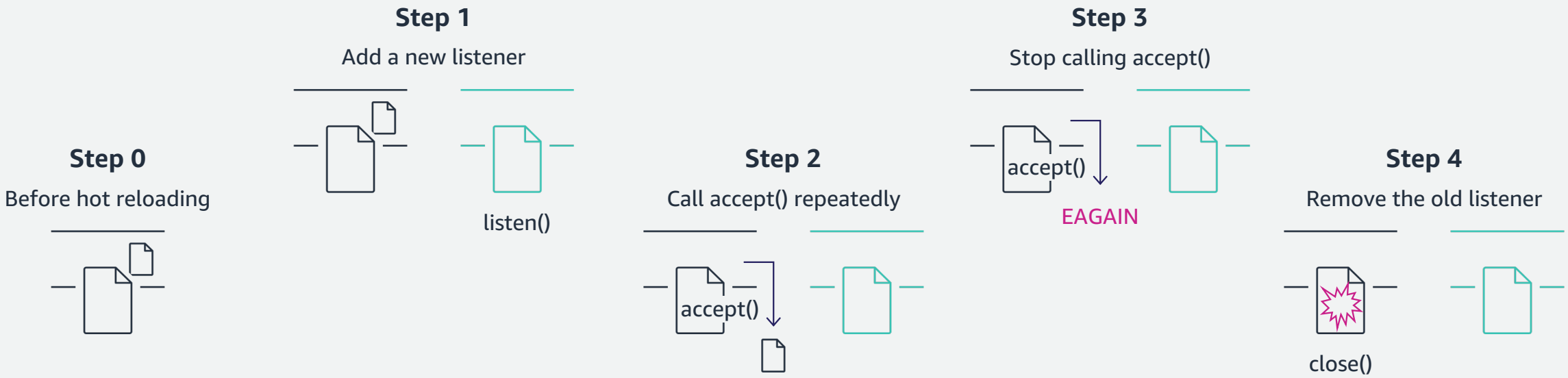
```
$ sudo tcpdump -i lo -t -nn
...
IP 127.0.0.1.39258 > 127.0.0.1.80: Flags [S], seq 428153089, ...
IP 127.0.0.1.80 > 127.0.0.1.39258: Flags [S.], seq 319723959, ...
IP 127.0.0.1.39258 > 127.0.0.1.80: Flags [.], ack 1, ...
IP 127.0.0.1.39258 > 127.0.0.1.80: Flags [P.], seq 1:12, ack 1, ...
IP 127.0.0.1.80 > 127.0.0.1.39258: Flags [.], ack 12, ...
IP 127.0.0.1.80 > 127.0.0.1.39258: Flags [R.], seq 1, ack 12, ...
```

Real world scenario

- While hot reloading servers, connections are lost
- Closing a listener aborts connections in the accept queue

Workaround

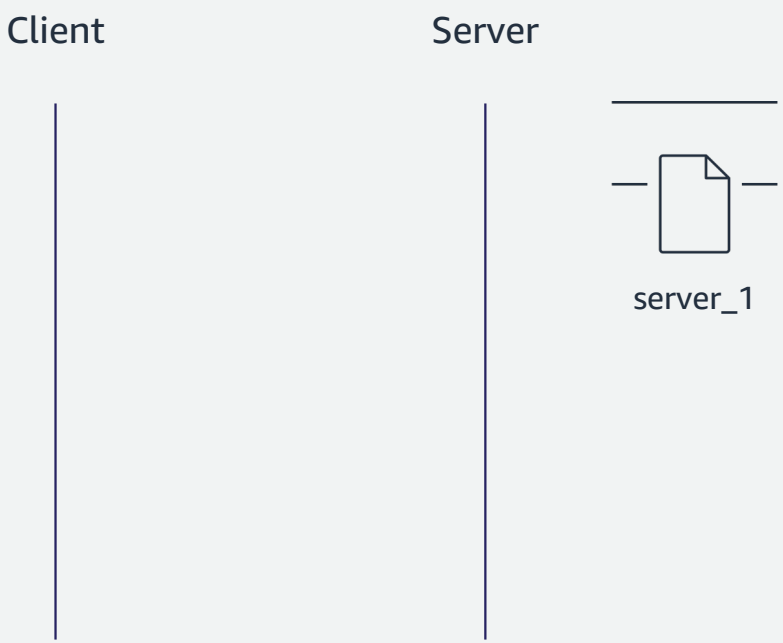
- Implement connection draining before close()
- Call accept() repeatedly until EAGAIN



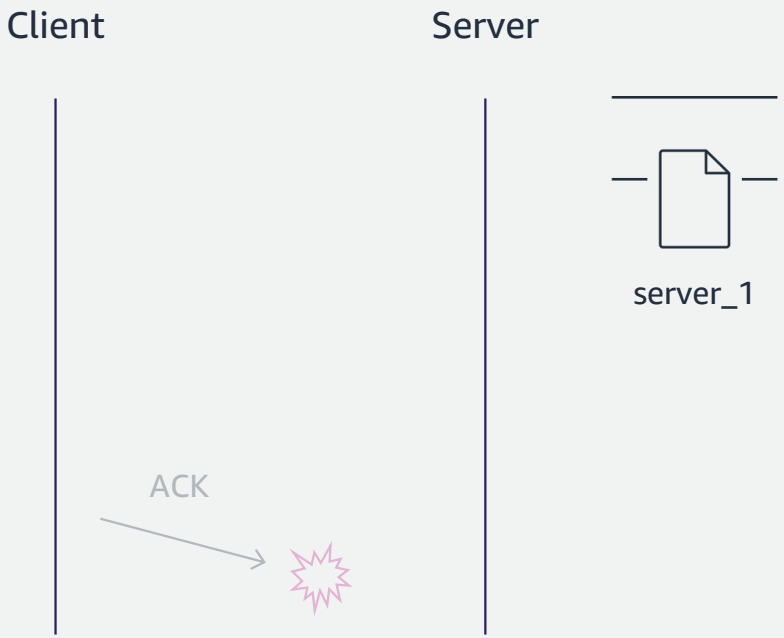
Quiz 2

```
import subprocess, time
from quiz1 import *

def drop_ack(flag=True):
    subprocess.run('iptables -{} INPUT -d 127.0.0.1 -p tcp --dport 80 --tcp-flags SYN,ACK ACK -j DROP'
                   .format('A' if flag else 'D').split(' '))

def quiz2():
    server_1 = get_reuseport_server()

    drop_ack(True)

    client = get_client()
    client.send(b'Hello World')

    server_2 = get_reuseport_server()

    server_1.close()

    drop_ack(False)
    time.sleep(1)

    server_2.setblocking(0)
    child, _ = server_2.accept()
    print(child.recv(1024))

if __name__ == '__main__':
    quiz2()
```

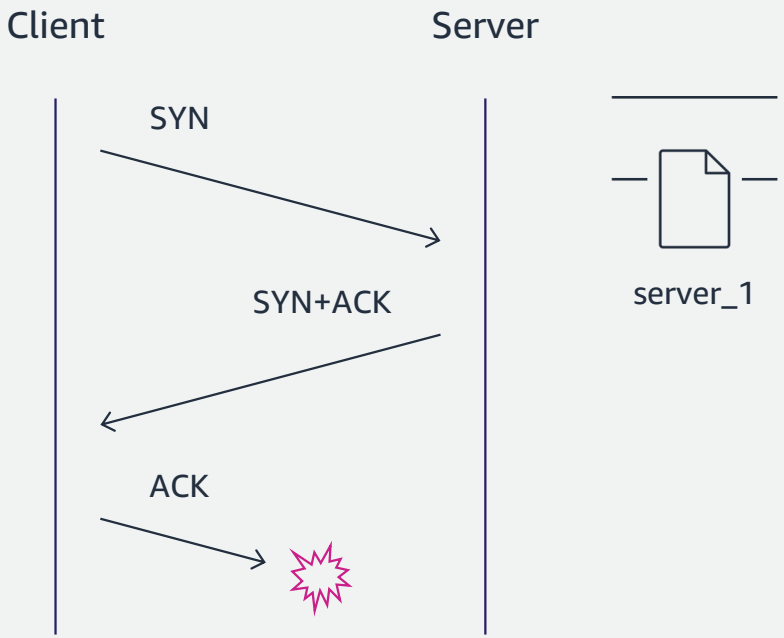
```
import subprocess, time
from quiz1 import *

def drop_ack(flag=True):
    subprocess.run('iptables -{} INPUT -d 127.0.0.1 -p tcp --dport 80 --tcp-flags SYN,ACK ACK -j DROP'
                   .format('A' if flag else 'D').split(' '))

def quiz2():
    server_1 = get_reuseport_server()

    drop_ack(True)

    client = get_client()
    client.send(b'Hello World')

    server_2 = get_reuseport_server()

    server_1.close()

    drop_ack(False)
    time.sleep(1)

    server_2.setblocking(0)
    child, _ = server_2.accept()
    print(child.recv(1024))

if __name__ == '__main__':
    quiz2()
```



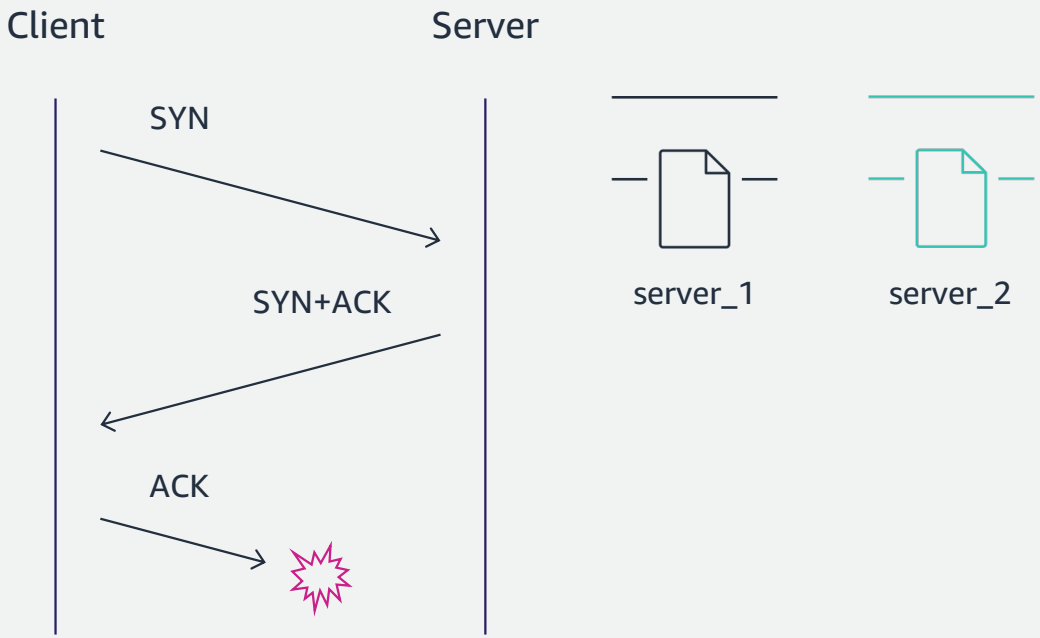
```
import subprocess, time
from quiz1 import *

def drop_ack(flag=True):
    subprocess.run('iptables -{} INPUT -d 127.0.0.1 -p tcp --dport 80 --tcp-flags SYN,ACK ACK -j DROP'
                   .format('A' if flag else 'D').split(' '))

def quiz2():
    server_1 = get_reuseport_server()

    drop_ack(True)

    client = get_client()
    client.send(b'Hello World')

    server_2 = get_reuseport_server()

    server_1.close()

    drop_ack(False)
    time.sleep(1)

    server_2.setblocking(0)
    child, _ = server_2.accept()
    print(child.recv(1024))

if __name__ == '__main__':
    quiz2()
```



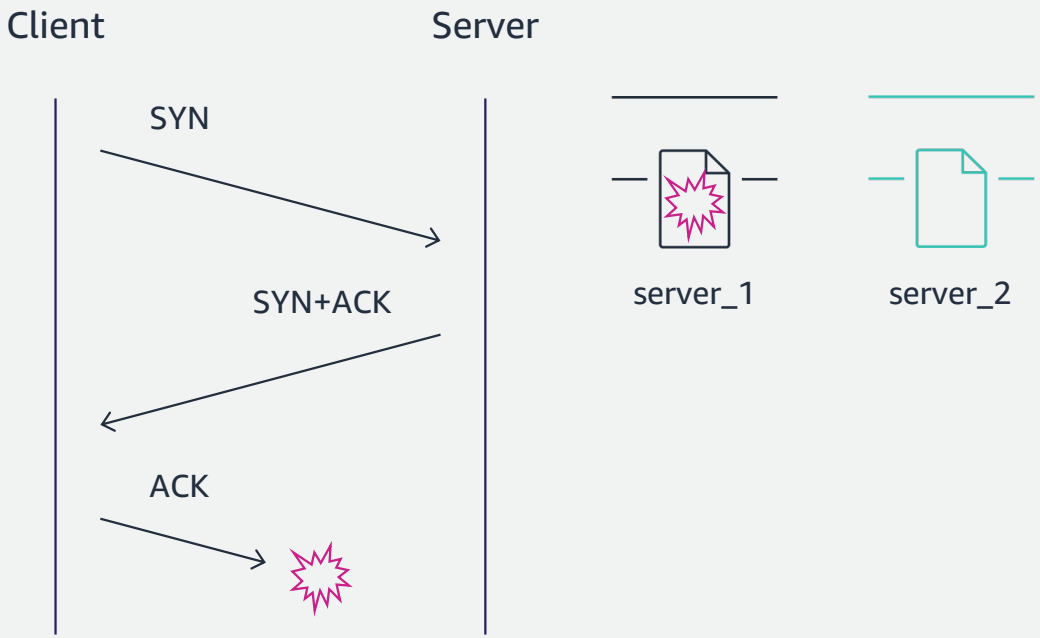
```
import subprocess, time
from quiz1 import *

def drop_ack(flag=True):
    subprocess.run('iptables -{} INPUT -d 127.0.0.1 -p tcp --dport 80 --tcp-flags SYN,ACK ACK -j DROP'
                   .format('A' if flag else 'D').split(' '))

def quiz2():
    server_1 = get_reuseport_server()

    drop_ack(True)

    client = get_client()
    client.send(b'Hello World')

    server_2 = get_reuseport_server()

    server_1.close()

    drop_ack(False)
    time.sleep(1)

    server_2.setblocking(0)
    child, _ = server_2.accept()
    print(child.recv(1024))

if __name__ == '__main__':
    quiz2()
```



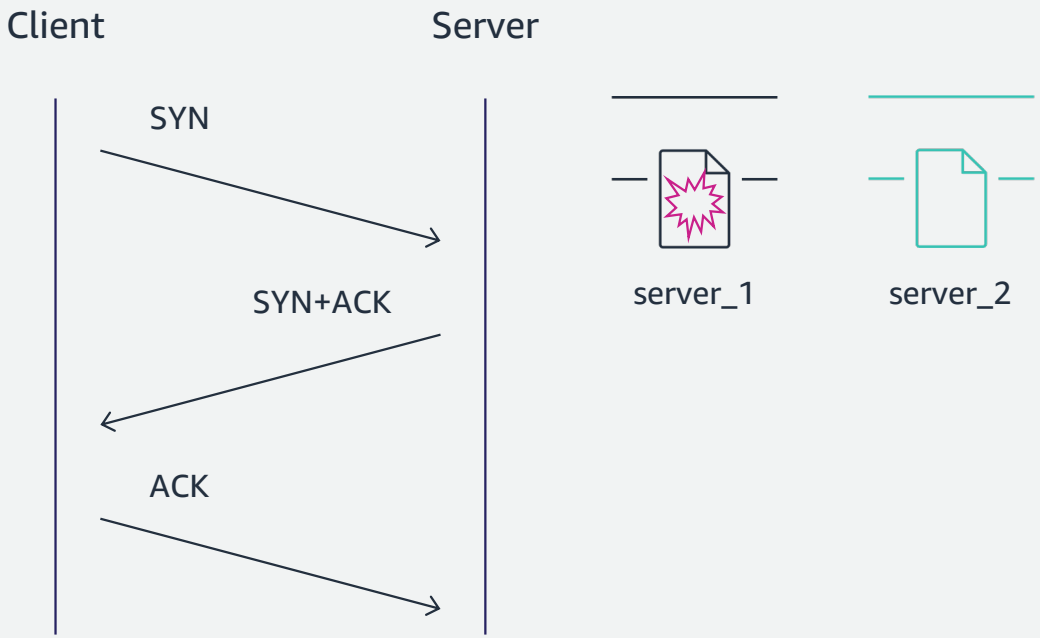
```
import subprocess, time
from quiz1 import *

def drop_ack(flag=True):
    subprocess.run('iptables -{} INPUT -d 127.0.0.1 -p tcp --dport 80 --tcp-flags SYN,ACK ACK -j DROP'
                   .format('A' if flag else 'D').split(' '))

def quiz2():
    server_1 = get_reuseport_server()

    drop_ack(True)

    client = get_client()
    client.send(b'Hello World')

    server_2 = get_reuseport_server()

    server_1.close()

    drop_ack(False)
    time.sleep(1)

    server_2.setblocking(0)
    child, _ = server_2.accept()
    print(child.recv(1024))

if __name__ == '__main__':
    quiz2()
```



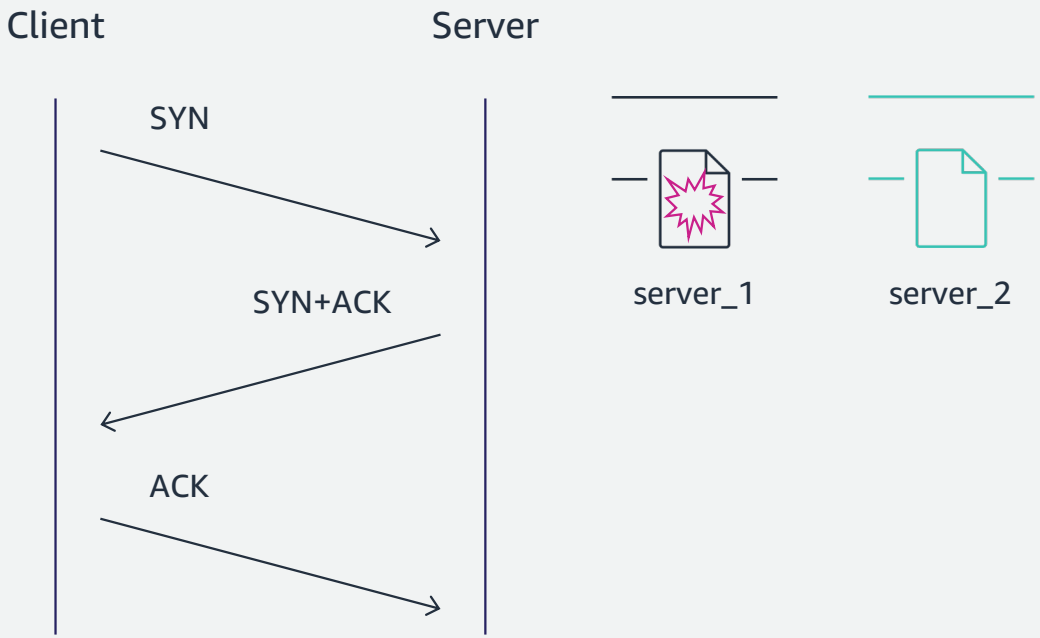
```
import subprocess, time
from quiz1 import *

def drop_ack(flag=True):
    subprocess.run('iptables -{} INPUT -d 127.0.0.1 -p tcp --dport 80 --tcp-flags SYN,ACK ACK -j DROP'
                   .format('A' if flag else 'D').split(' '))

def quiz2():
    server_1 = get_reuseport_server()

    drop_ack(True)

    client = get_client()
    client.send(b'Hello World')

    server_2 = get_reuseport_server()

    server_1.close()

    drop_ack(False)
    time.sleep(1)

    server_2.setblocking(0)
    child, _ = server_2.accept()
    print(child.recv(1024))

if __name__ == '__main__':
    quiz2()
```



```
import subprocess, time
from quiz1 import *

def drop_ack(flag=True):
    subprocess.run('iptables -{} INPUT -d 127.0.0.1 -p tcp --dport 80 --tcp-flags SYN,ACK ACK -j DROP'
                   .format('A' if flag else 'D').split(' '))

def quiz2():
    server_1 = get_reuseport_server()

    drop_ack(True)

    client = get_client()
    client.send(b'Hello World')

    server_2 = get_reuseport_server()

    server_1.close()

    drop_ack(False)
    time.sleep(1)

    server_2.setblocking(0)
    child, _ = server_2.accept()
    print(child.recv(1024))

if __name__ == '__main__':
    quiz2()
```



```
import subprocess, time
from quiz1 import *

def drop_ack(flag=True):
    subprocess.run('iptables -{} INPUT -d 127.0.0.1 -p tcp --dport 80 --tcp-flags SYN,ACK ACK -j DROP'
                   .format('A' if flag else 'D').split(' '))

def quiz2():
    server_1 = get_reuseport_server()

    drop_ack(True)

    client = get_client()
    client.send(b'Hello World')

    server_2 = get_reuseport_server()

    server_1.close()

    drop_ack(False)
    time.sleep(1)

    server_2.setblocking(0)
    child, _ = server_2.accept()
    print(child.recv(1024))

if __name__ == '__main__':
    quiz2()
```



```
import subprocess, time
from quiz1 import *

def drop_ack(flag=True):
    subprocess.run('iptables -{} INPUT -d 127.0.0.1 -p tcp --dport 80 --tcp-flags SYN,ACK ACK -j DROP'
                   .format('A' if flag else 'D').split(' '))

def quiz2():
    server_1 = get_reuseport_server()

    drop_ack(True)

    client = get_client()
    client.send(b'Hello World')

    server_2 = get_reuseport_server()

    server_1.close()

    drop_ack(False)
    time.sleep(1)

    server_2.setblocking(0)
    child, _ = server_2.accept()
    print(child.recv(1024))

if __name__ == '__main__':
    quiz2()
```

Non-optimal

```
$ sudo python3 quiz2.py
...
   child, _ = server_2.accept()
...
BlockingIOError: [Errno 11] Resource temporarily unavailable
```

Optimal

```
$ sudo python3 quiz2.py
b'Hello World'
```

```
import subprocess, time
from quiz1 import *

def drop_ack(flag=True):
    subprocess.run('iptables -{} INPUT -d 127.0.0.1 -p tcp --dport 80 --tcp-flags SYN,ACK ACK -j DROP'
                   .format('A' if flag else 'D').split(' '))

def quiz2():
    server_1 = get_reuseport_server()

    drop_ack(True)

    client = get_client()
    client.send(b'Hello World')

    server_2 = get_reuseport_server()

    server_1.close()

    drop_ack(False)
    time.sleep(1)

    server_2.setblocking(0)
    child, _ = server_2.accept()
    print(child.recv(1024))

if __name__ == '__main__':
    quiz2()
```

Non-optimal

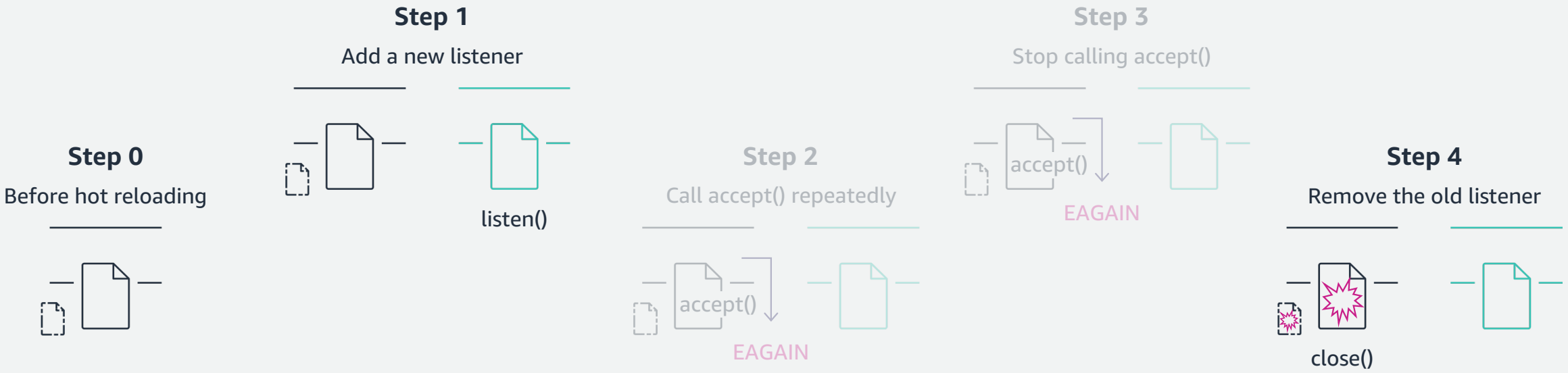
```
$ sudo python3 quiz2.py
...
   child, _ = server_2.accept()
...
BlockingIOError: [Errno 11] Resource temporarily unavailable


$ sudo tcpdump -i lo -t -nn
...
IP 127.0.0.1.39260 > 127.0.0.1.80: Flags [S], seq 599362899, ...
IP 127.0.0.1.80 > 127.0.0.1.39260: Flags [S.], seq 2100710622, ...
IP 127.0.0.1.39260 > 127.0.0.1.80: Flags [.], ack 1, ...
IP 127.0.0.1.39260 > 127.0.0.1.80: Flags [P.], seq 1:12, ack 1, ...
IP 127.0.0.1.39260 > 127.0.0.1.80: Flags [P.], seq 1:12, ack 1, ...
IP 127.0.0.1.80 > 127.0.0.1.39260: Flags [R], seq 2100710623, ...
```

Real world scenario

- While hot reloading servers, in-flight requests are lost
- Closing a listener aborts immature connections during 3-way handshake

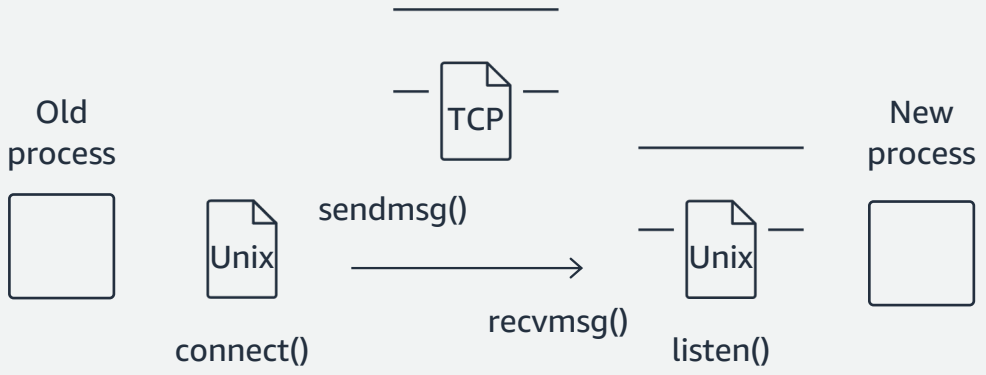
Workaround?

- Connection draining does not work
  - EAGAIN does not mean there are no in-flight requests
  - Even in-flight requests are tied to a listener but invisible to user space



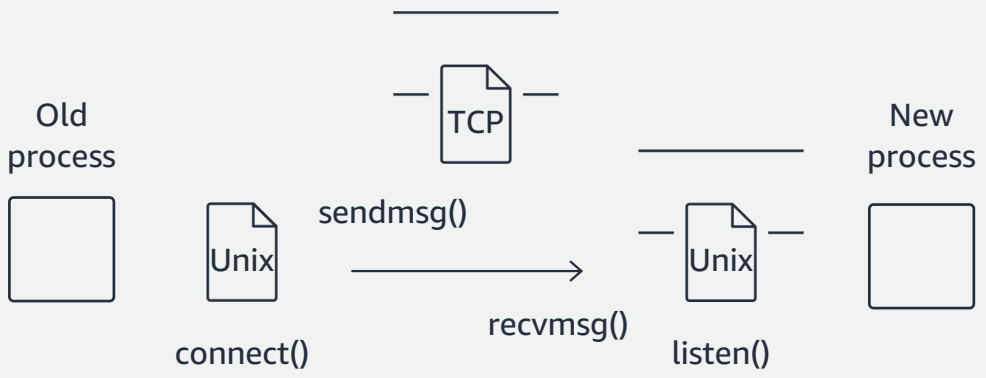
Workaround 1 - FD Passing

- Pass a TCP listener via AF_UNIX sockets with SCM_RIGHTS
  - Need not close() listeners
- Need not drain connections
  - Apply new settings faster



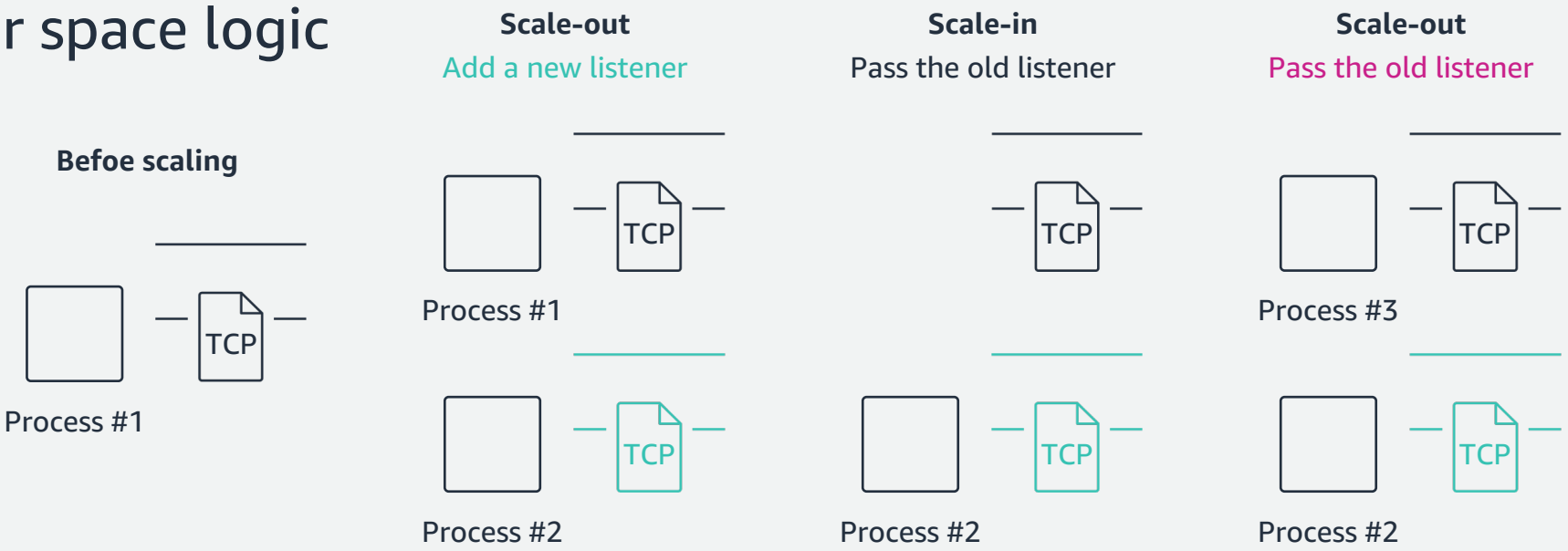
Workaround 1 - FD Passing

- Cannot scale-out/in easily
  - Need interactions between processes
  - Cannot close()
  - Complicated user space logic



Workaround 1 - FD Passing

- Cannot scale-out/in easily
  - Need interactions between processes
  - Cannot close()
  - Complicated user space logic

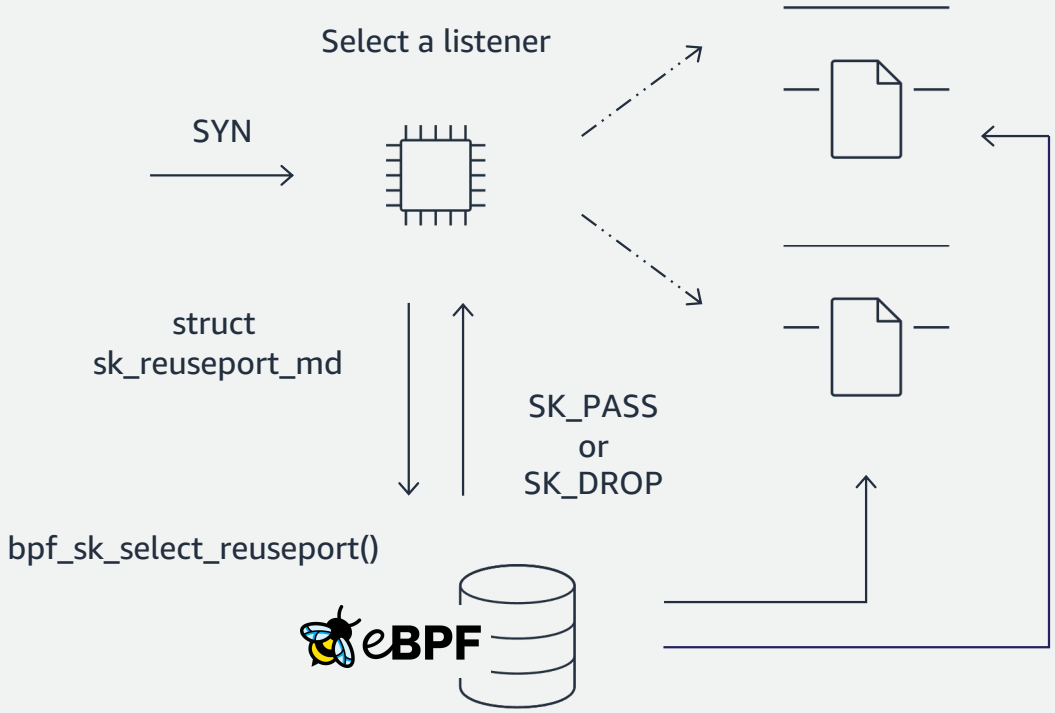


Workaround 2 - Connection draining with BPF

- BPF feature added for SO_REUSEPORT in v4.19 (9d6f417714c3)
  - BPF_MAP_TYPE_REUSEPORT_SOCKARRAY
  - BPF_PROG_TYPE_SK_REUSEPORT

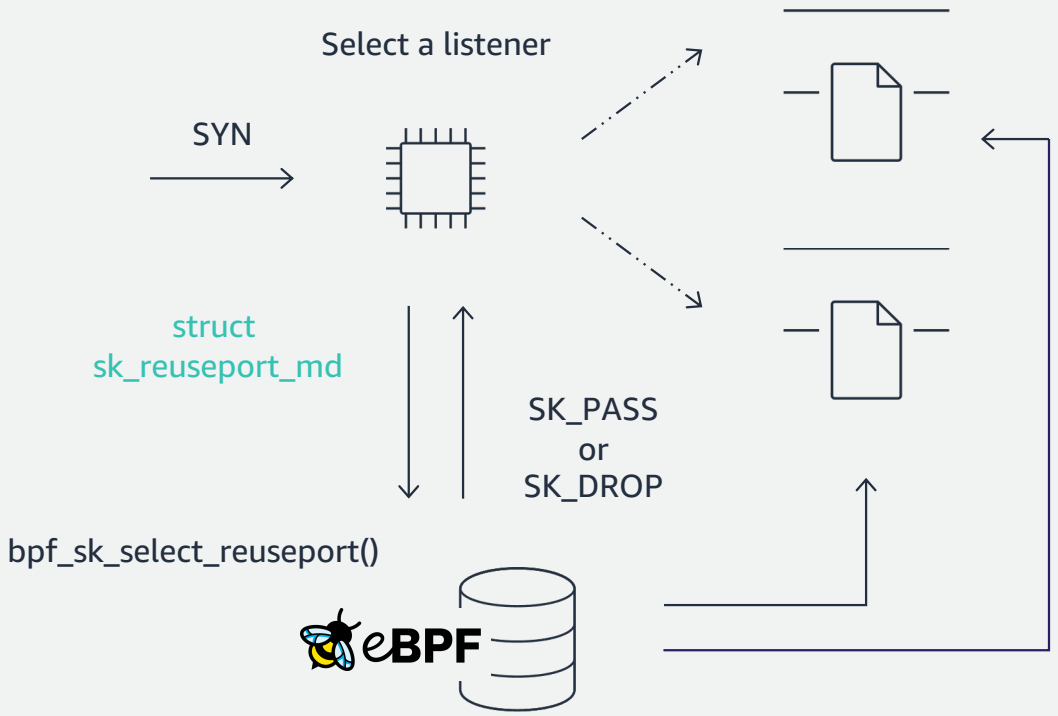
Workaround 2 - Connection draining with BPF

- BPF_MAP_TYPE_REUSEPORT_SOCKARRAY
  - Contain listeners on the same port
- BPF_PROG_TYPE_SK_REUSEPORT
  - Executed when receiving a SYN packet
  - Decide which listener in the map handles the connection

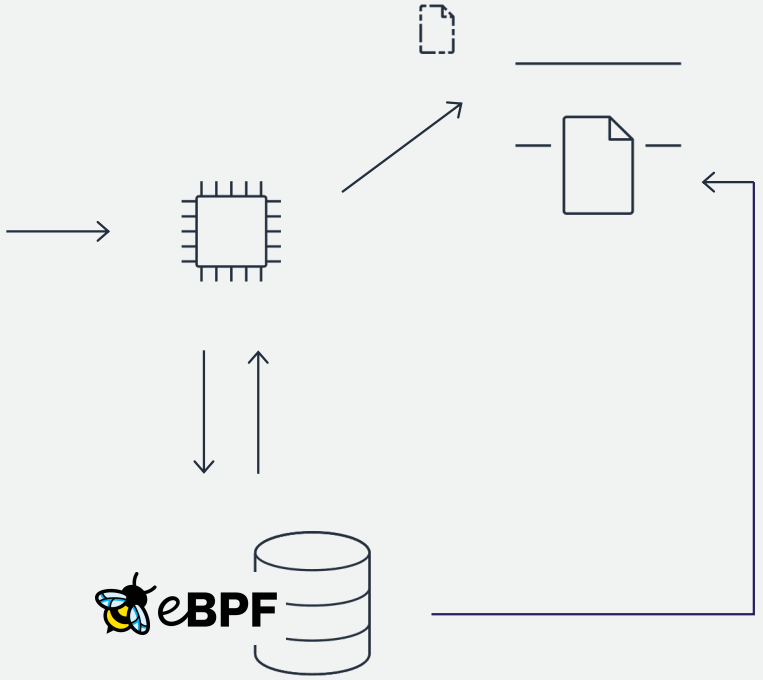


Workaround 2 - Connection draining with BPF

```
struct sk_reuseport_md {
        /*
         * Start of directly accessible data. It begins from
         * the tcp/udp header.
         */
        __bpf_md_ptr(void *, data);

        /* End of directly accessible data
         */
        __bpf_md_ptr(void *, data_end);

        __u32 len;              /* Total length of packet
                                 * (starting from the tcp/udp header).
                                 */
        __u32 eth_protocol;
        __u32 ip_protocol;      /* IP protocol. e.g. IPPROTO_TCP, IPPROTO_UDP */
        __u32 bind_inany;       /* Is sock bound to an INANY address? */
        __u32 hash;             /* A hash of the packet 4 tuples */
};
```

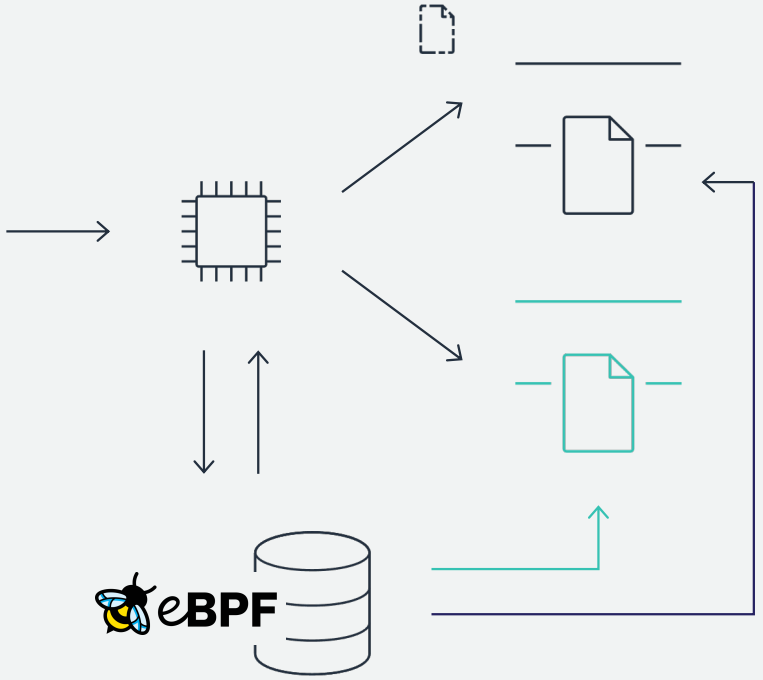


Workaround 2 - Connection draining with BPF



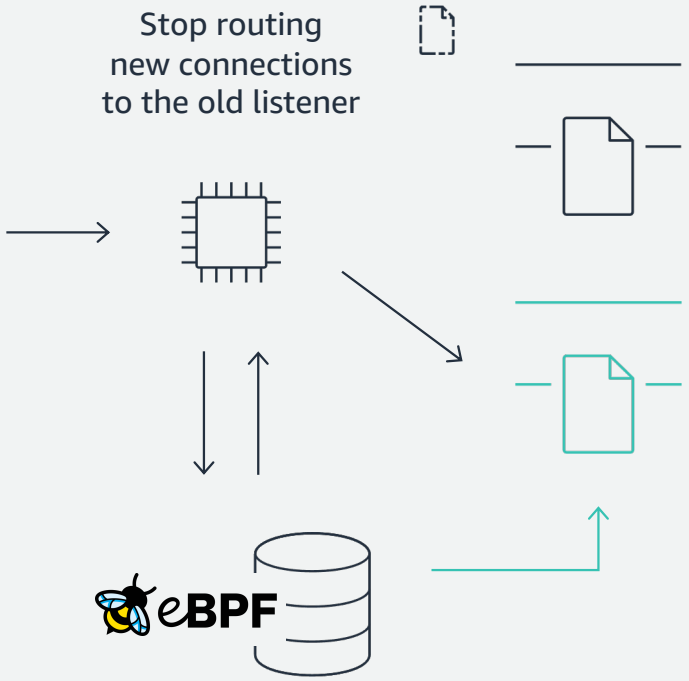
Workaround 2 - Connection draining with BPF

1. Add a new listener to the map



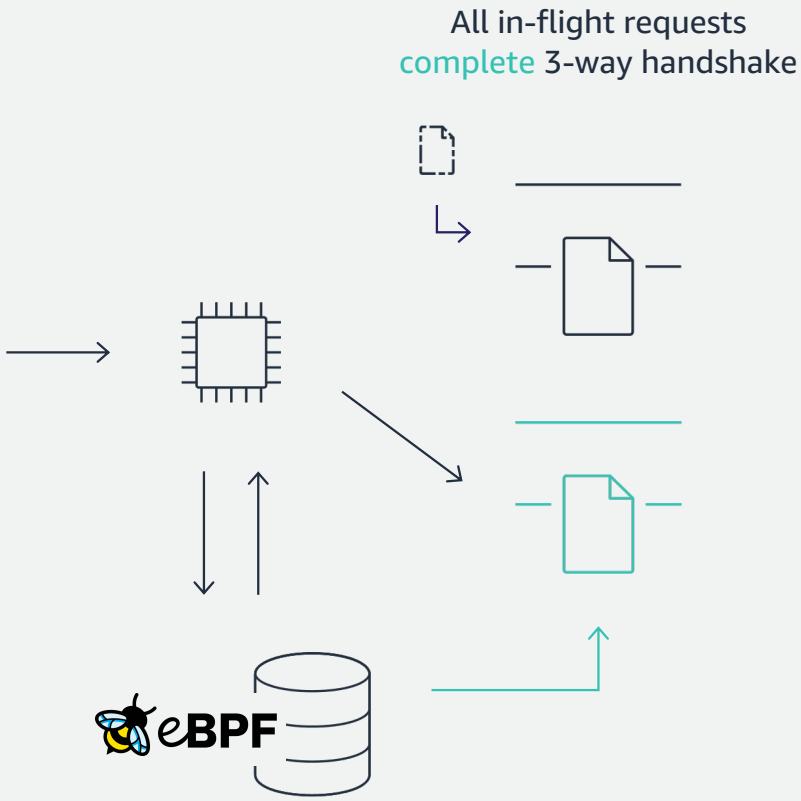
Workaround 2 - Connection draining with BPF

1. Add a new listener to the map
2. Remove the old listener from the map



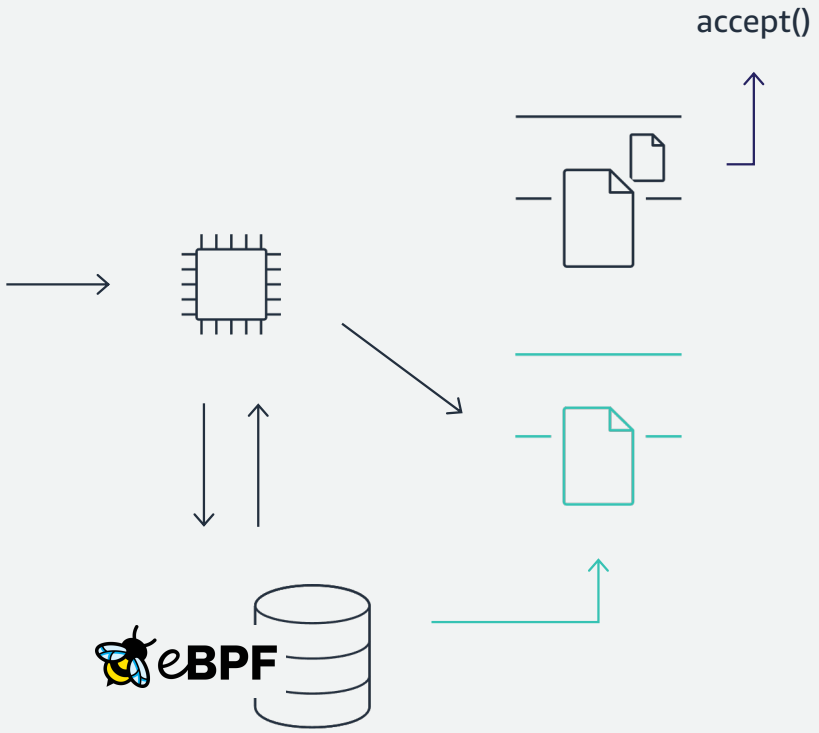
Workaround 2 - Connection draining with BPF

1. Add a new listener to the map
2. Remove the old listener from the map
3. Wait for the SYN+ACK timer to expire
   - net.ipv4.tcp_synack_retries = 5 (default)
   - 1 + 2 + 4 + 8 + 16 = 31s



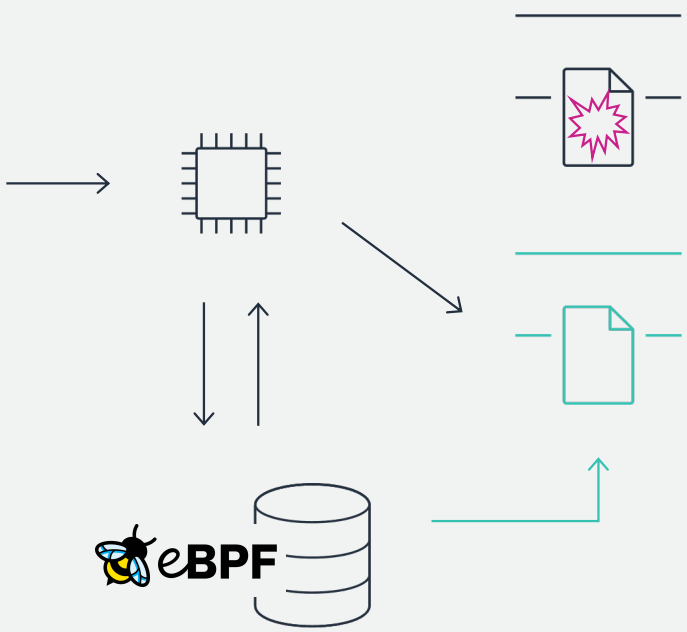
Workaround 2 - Connection draining with BPF

1. Add a new listener to the map
2. Remove the old listener from the map
3. Wait for the SYN+ACK timer to expire
   - net.ipv4.tcp_synack_retries = 5 (default)
   - 1 + 2 + 4 + 8 + 16 = 31s
4. accept() until -EAGAIN



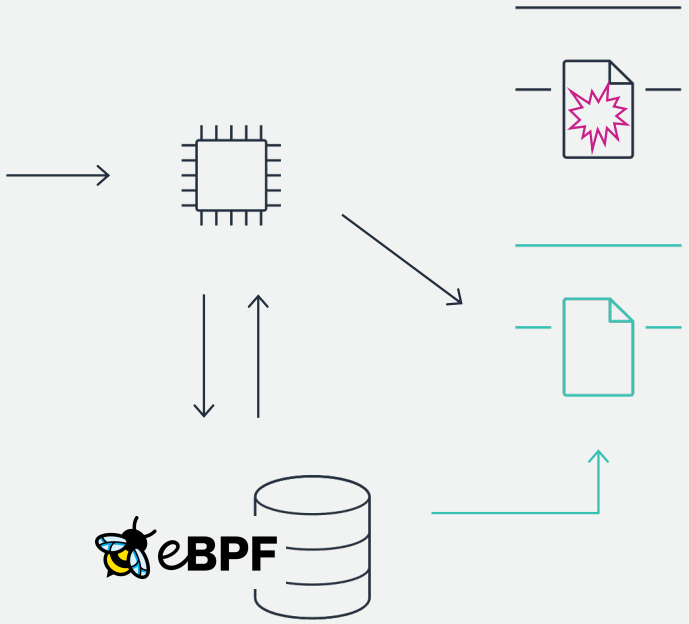
Workaround 2 - Connection draining with BPF

1. Add a new listener to the map
2. Remove the old listener from the map
3. Wait for the SYN+ACK timer to expire
   - net.ipv4.tcp_synack_retries = 5 (default)
   - 1 + 2 + 4 + 8 + 16 = 31s
4. accept() until -EAGAIN
5. close() the old listener



Workaround 2 - Connection draining with BPF

- Scale-out/in easily
  - Removing itself from the map stops routing new connections
  - Each process can work independently
- Need connection draining
  - Processes with old settings remain longer
  - Unsafe in terms of security



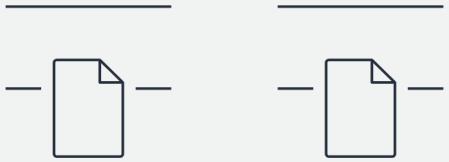
Is the behaviour acceptable?

- If only one socket is listening, connection failures are acceptable
- If multiple sockets are listening … ?

Where SO_REUSEPORT misbehaves

Where SO_REUSEPORT should work

Create multiple sockets

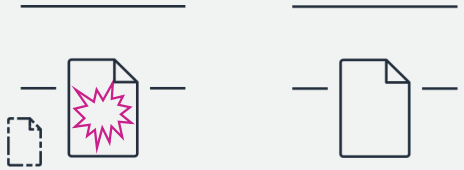


socket(), bind(), listen()

socket(), bind(), listen()

Quiz 2

Process 3-way handshake



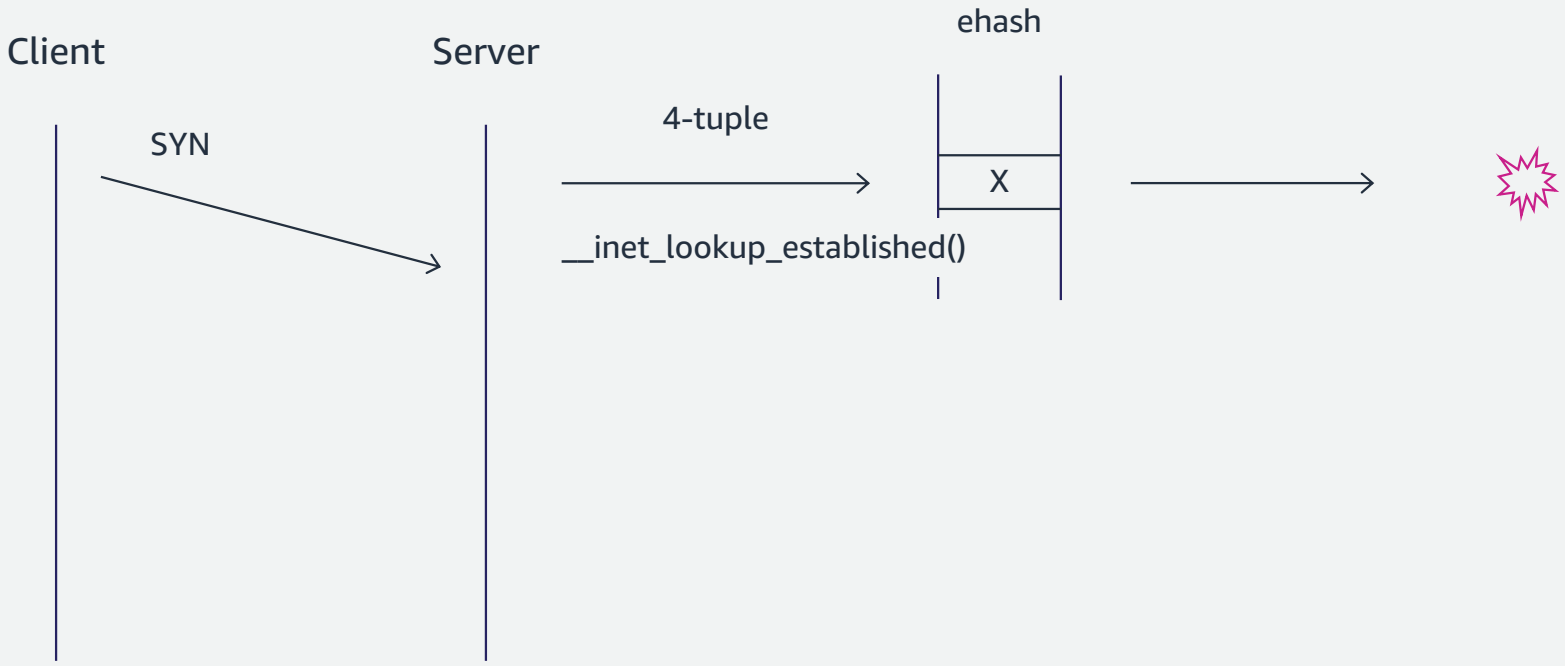
Quiz 1

Close a listener

close()

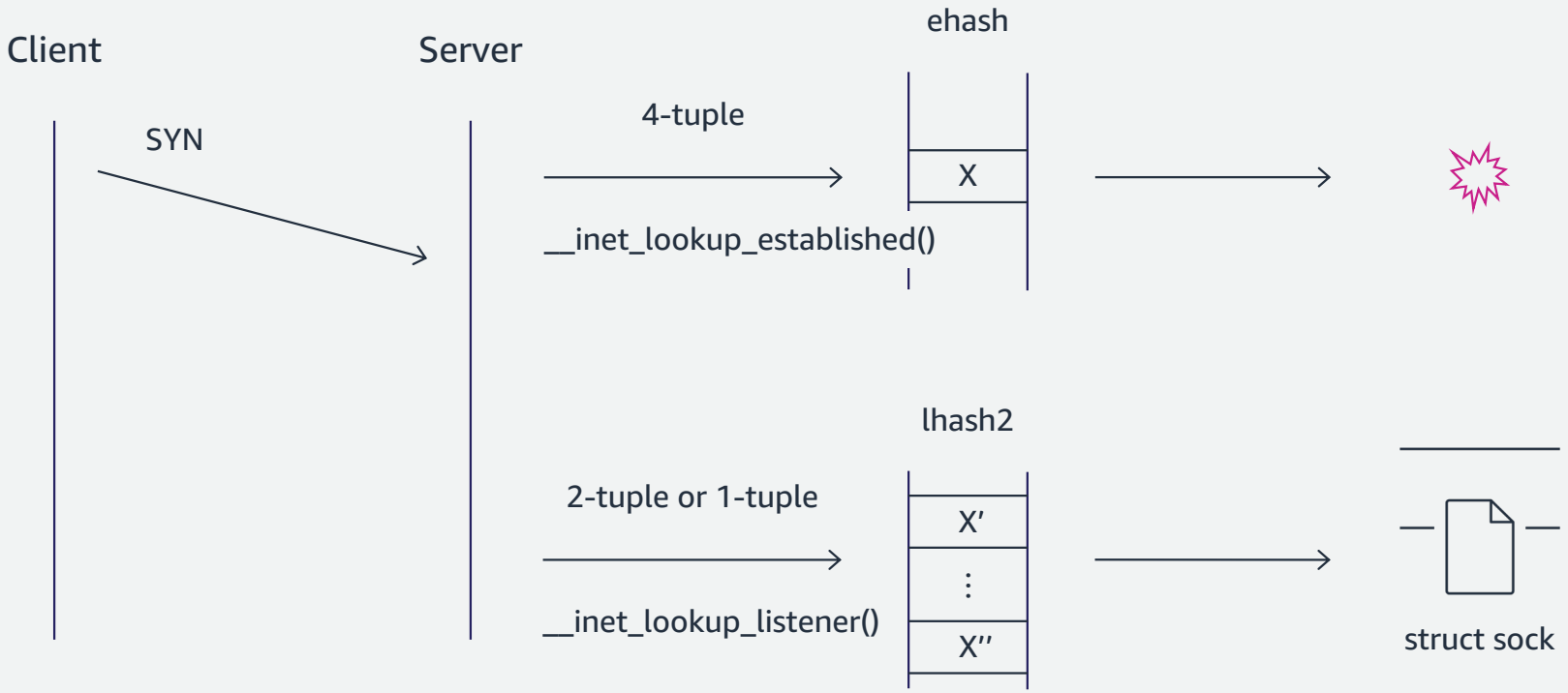
3-way handshake - SYN

1. Look up a listener



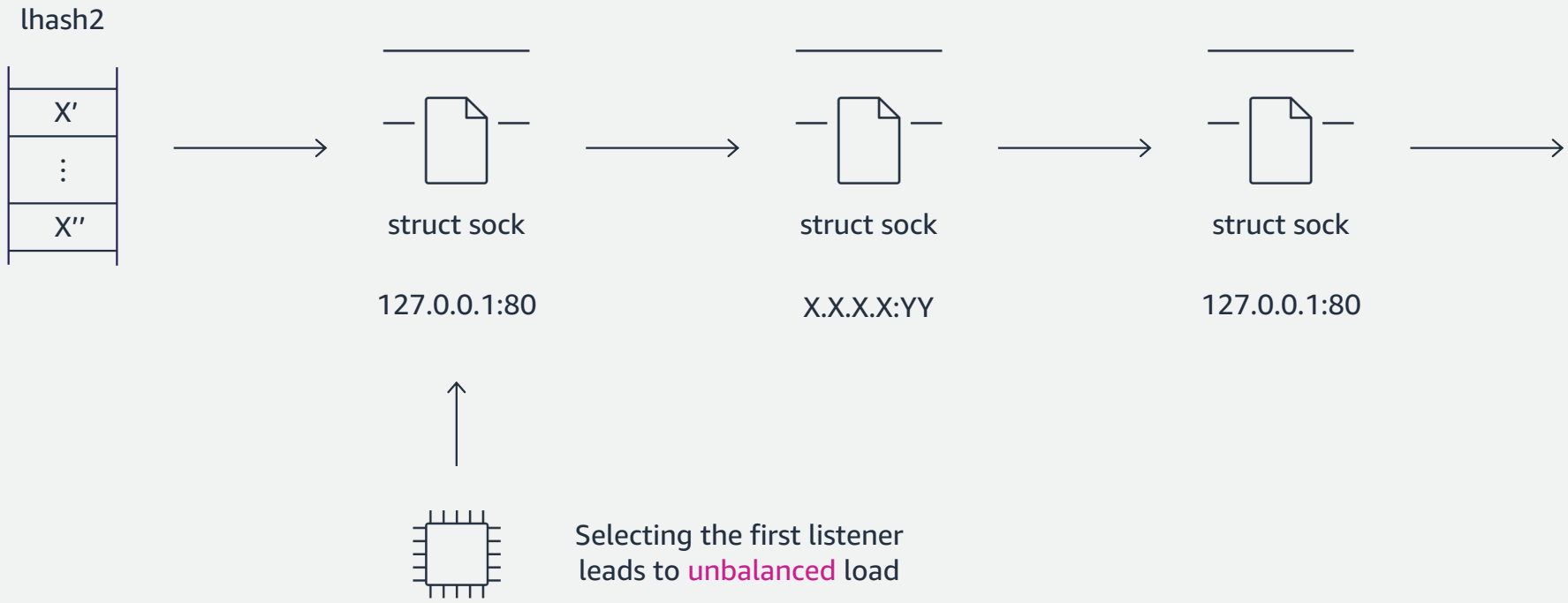
3-way handshake - SYN

1. Look up a listener



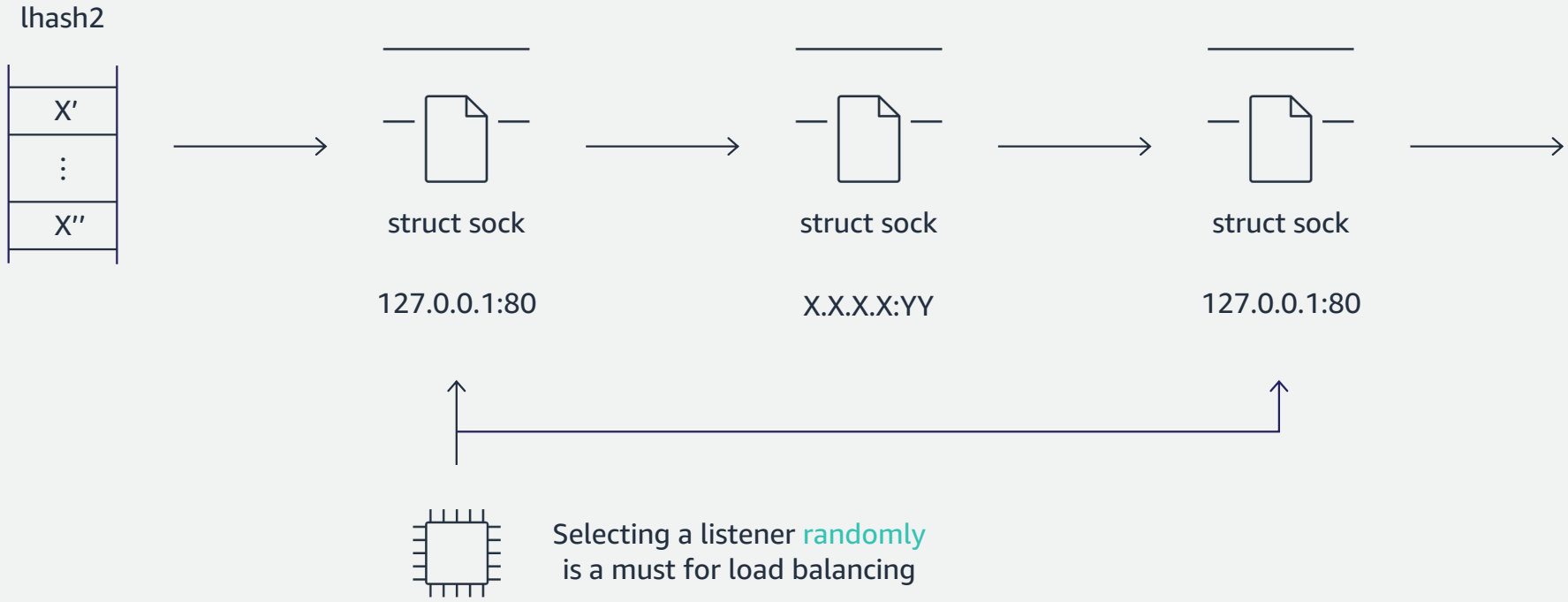
3-way handshake - SYN

1. Look up a listener



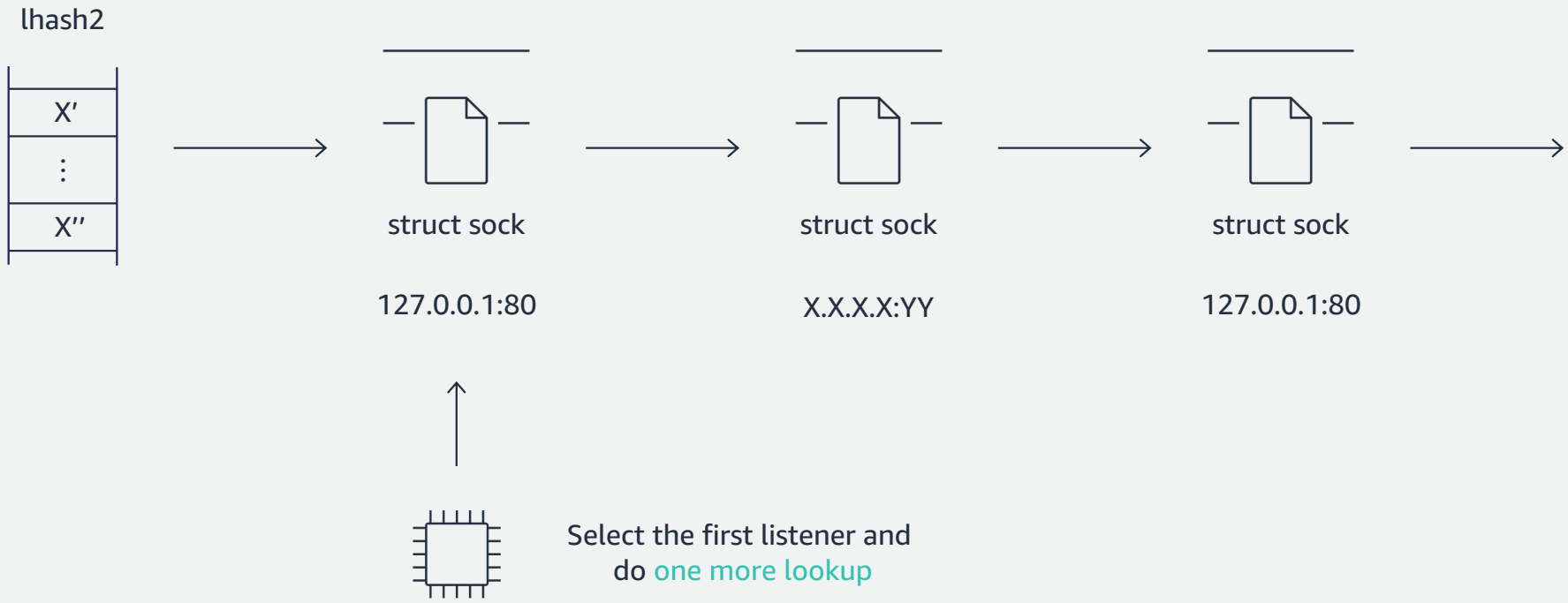
3-way handshake - SYN

1. Look up a listener



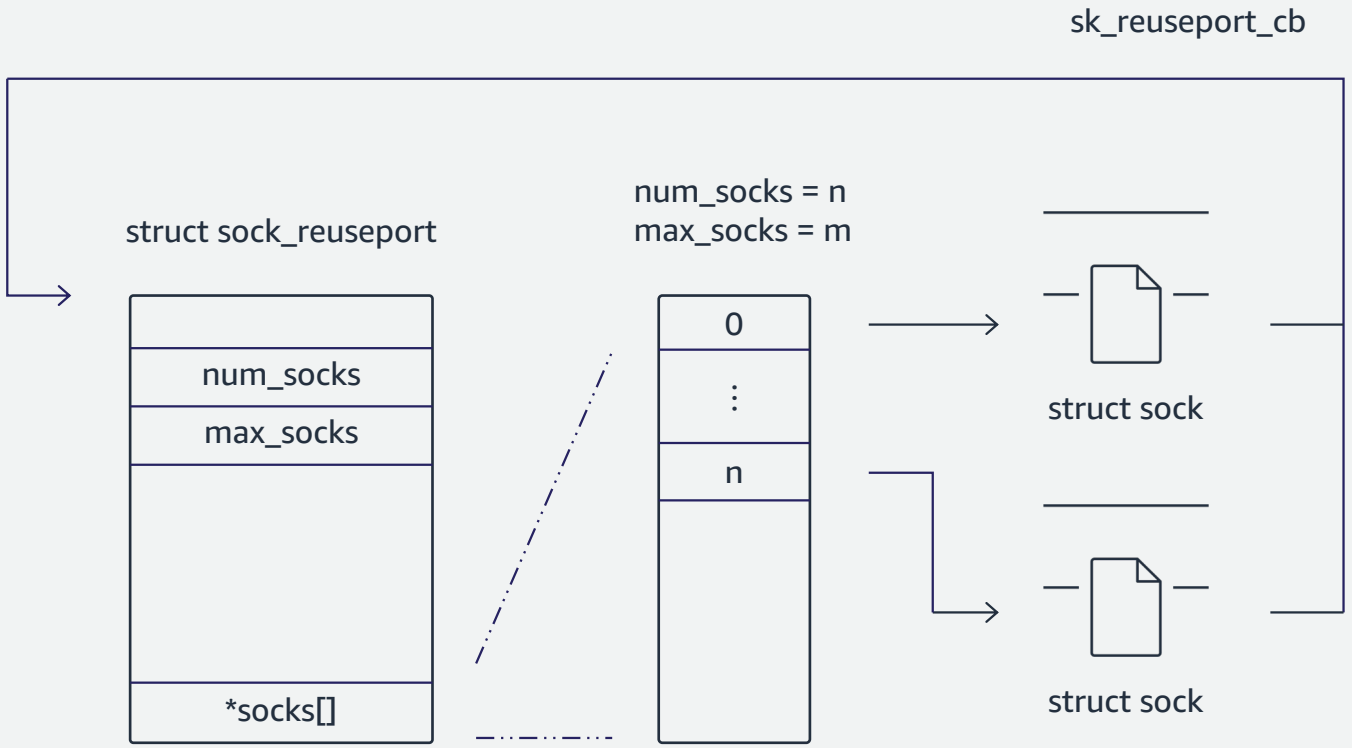
3-way handshake - SYN

1. Look up a listener



3-way handshake - SYN

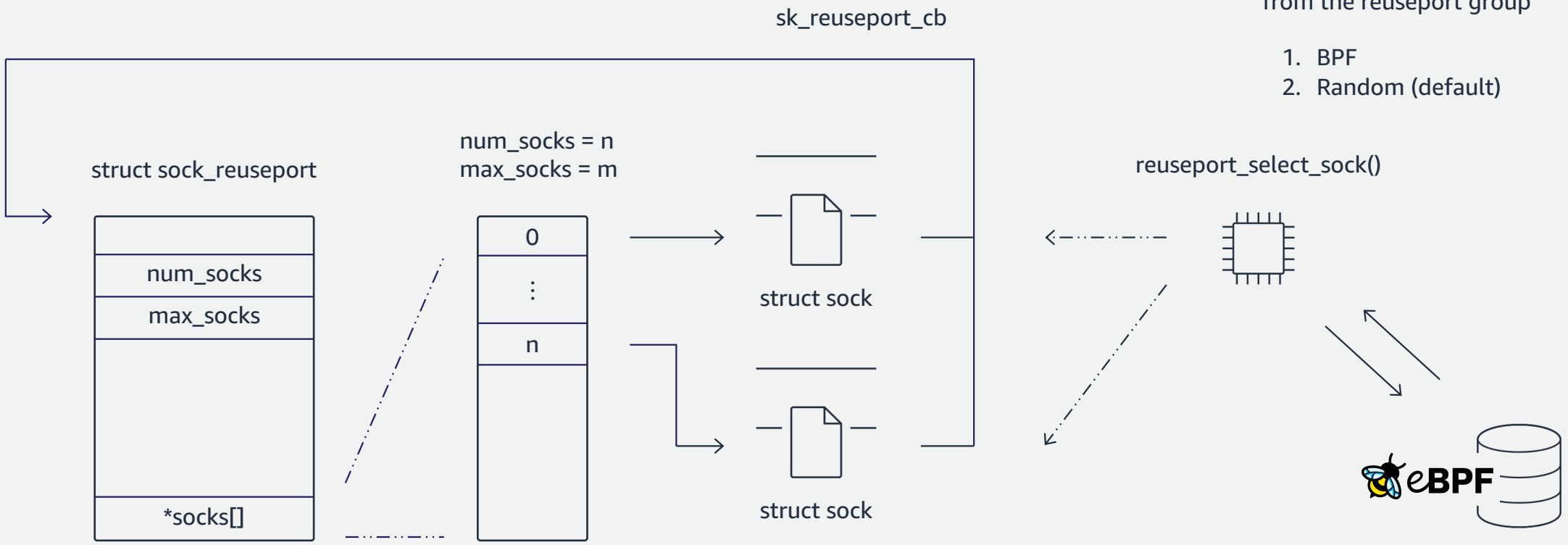
1. Look up a listener



A reuseport group is shared between listeners on the same port

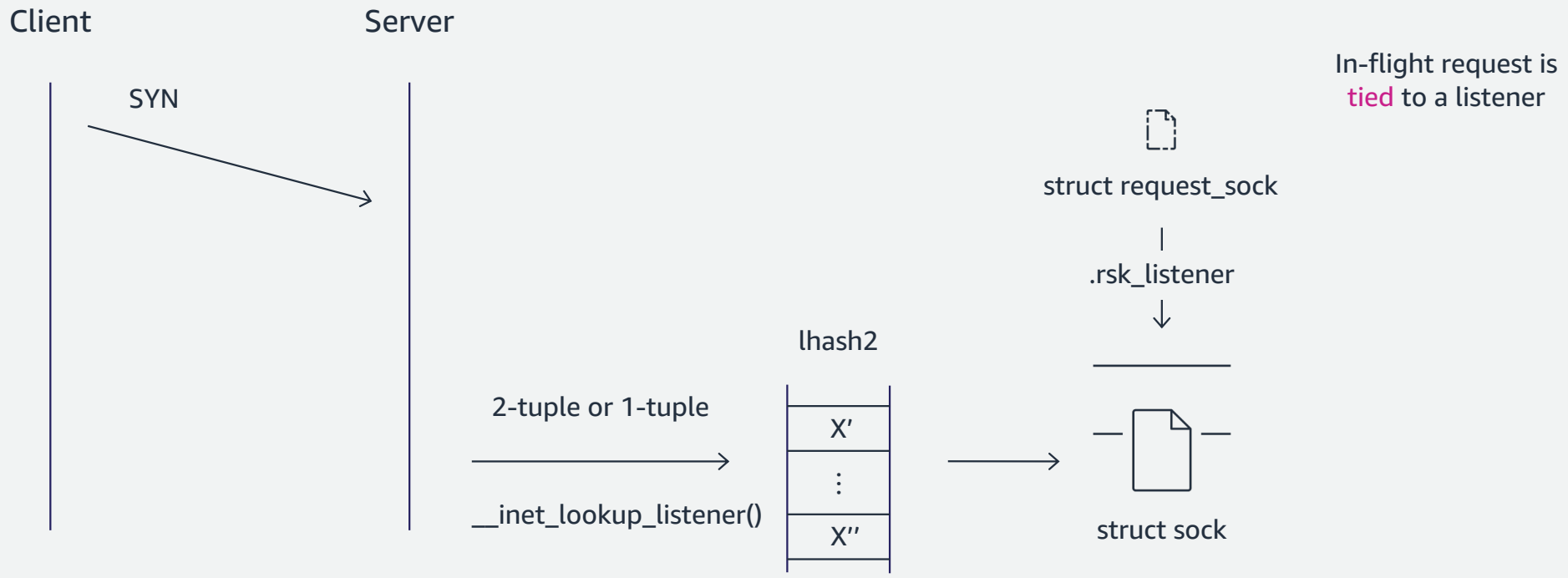
3-way handshake - SYN

1. Look up a listener



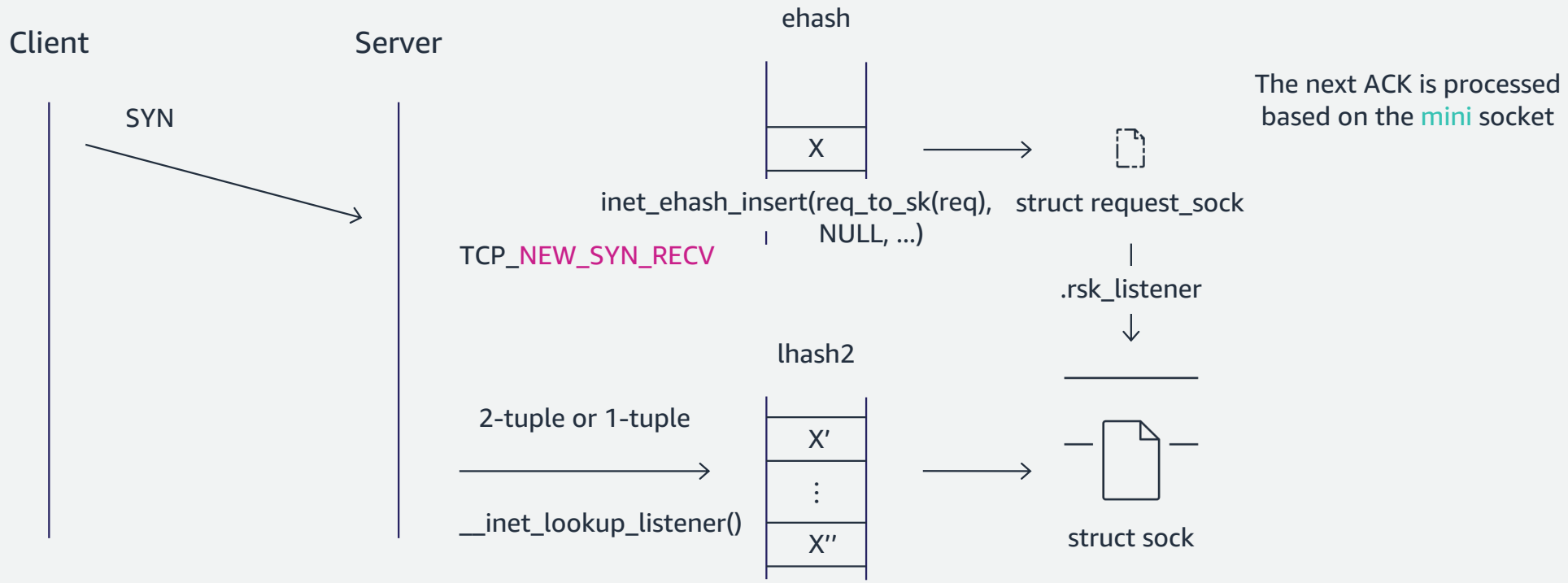
3-way handshake - SYN

2. Create a mini socket



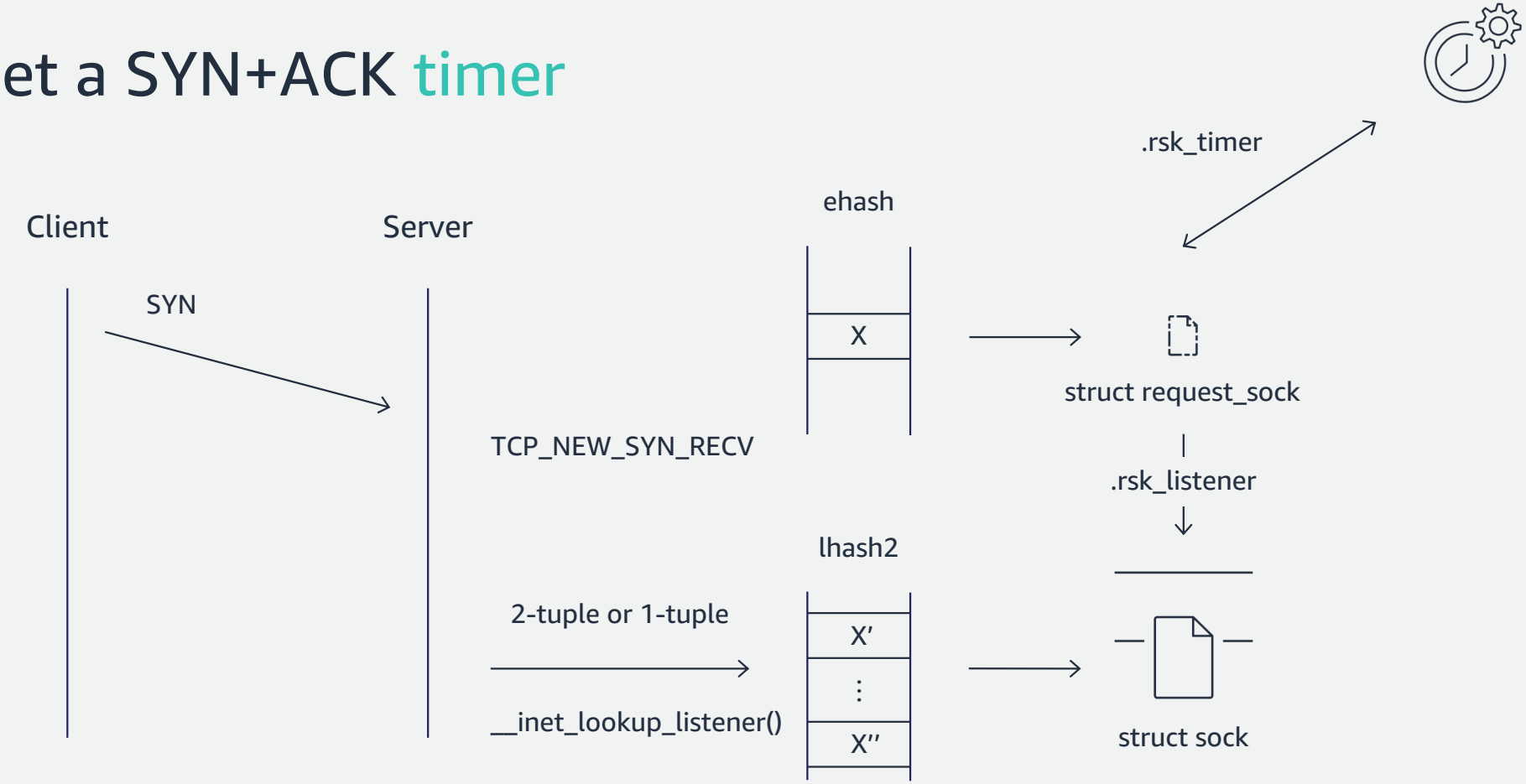
3-way handshake - SYN

3. Put the mini socket into ehash



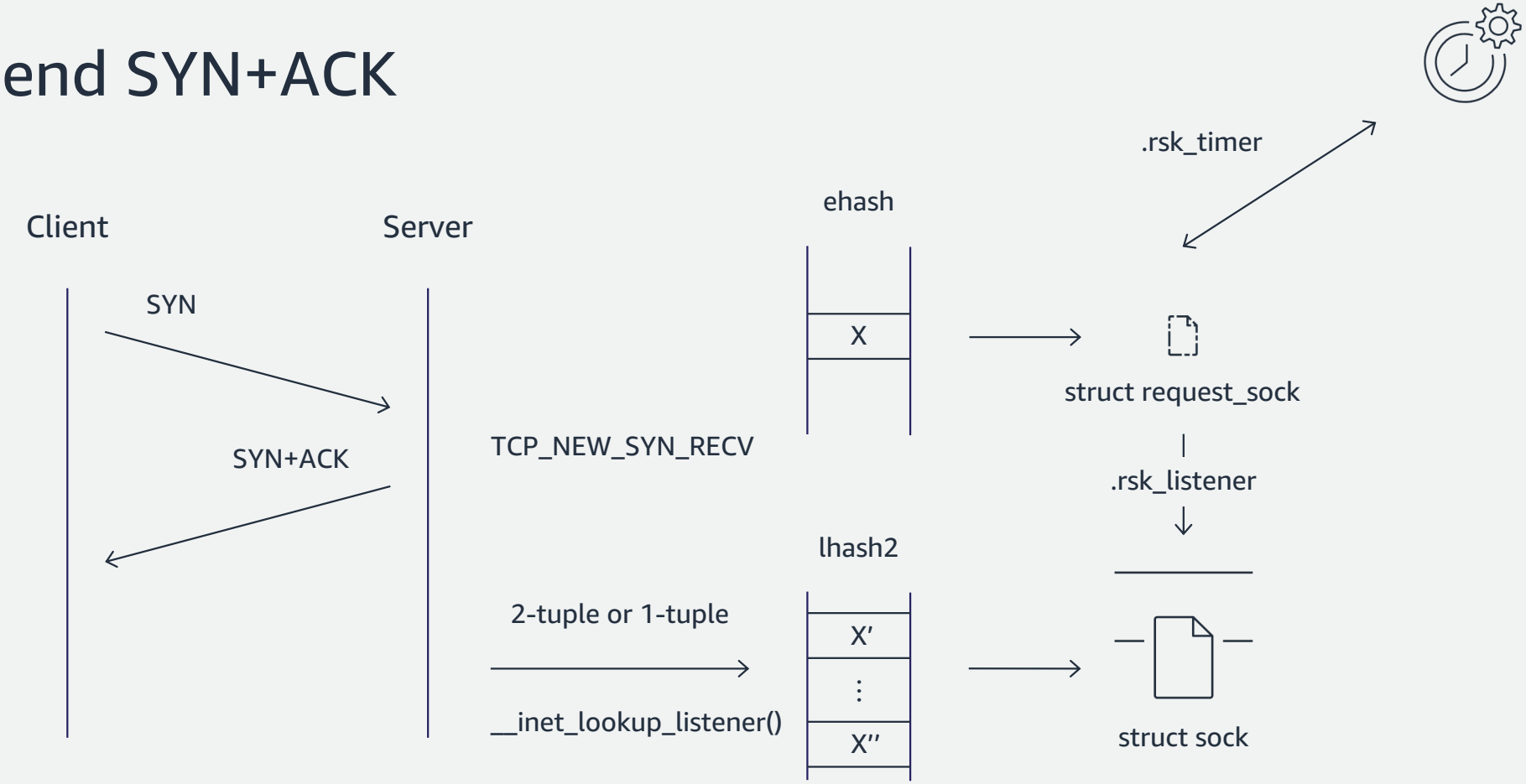
3-way handshake - SYN

4. Set a SYN+ACK timer



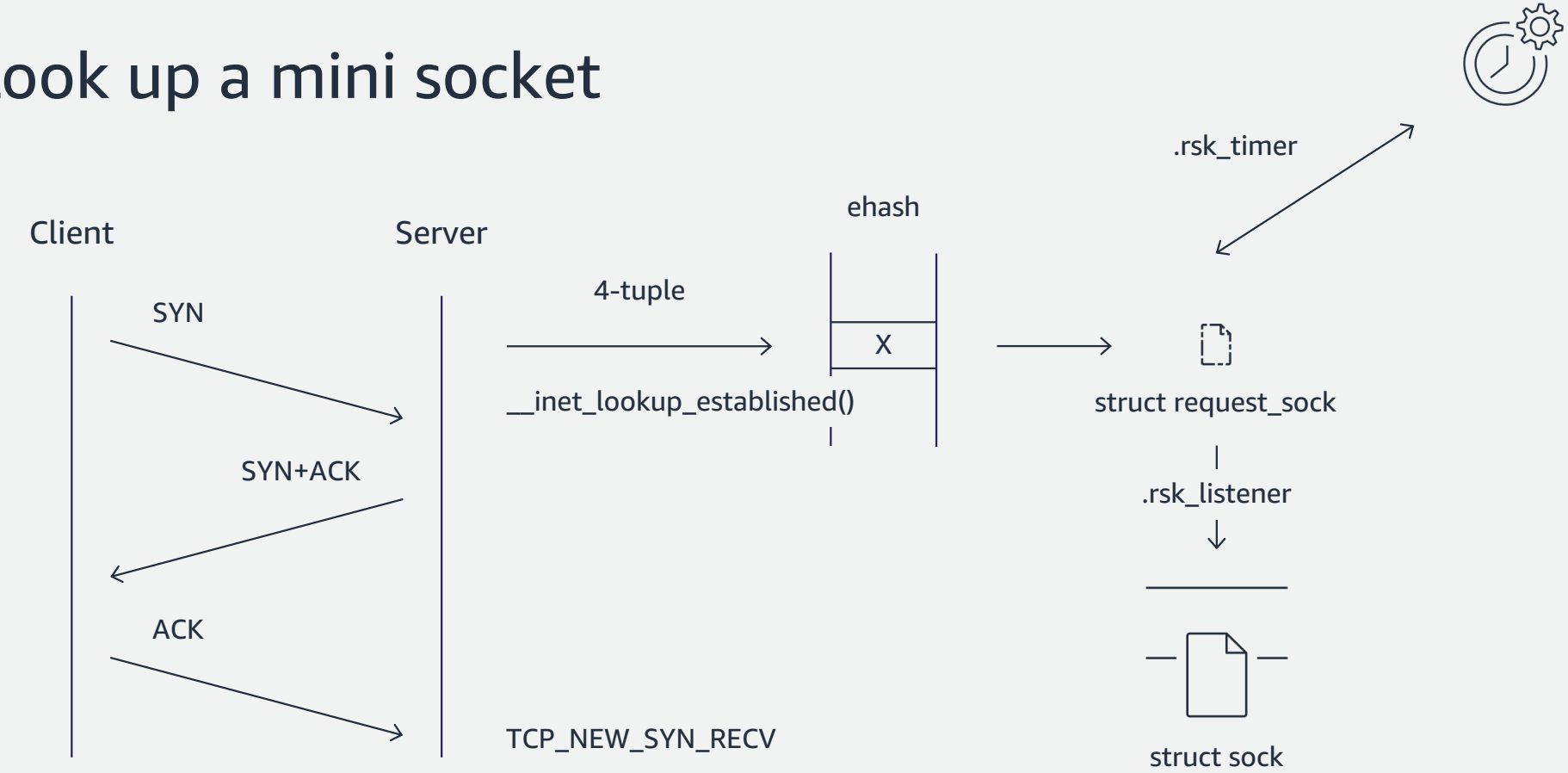
3-way handshake - SYN

5. Send SYN+ACK



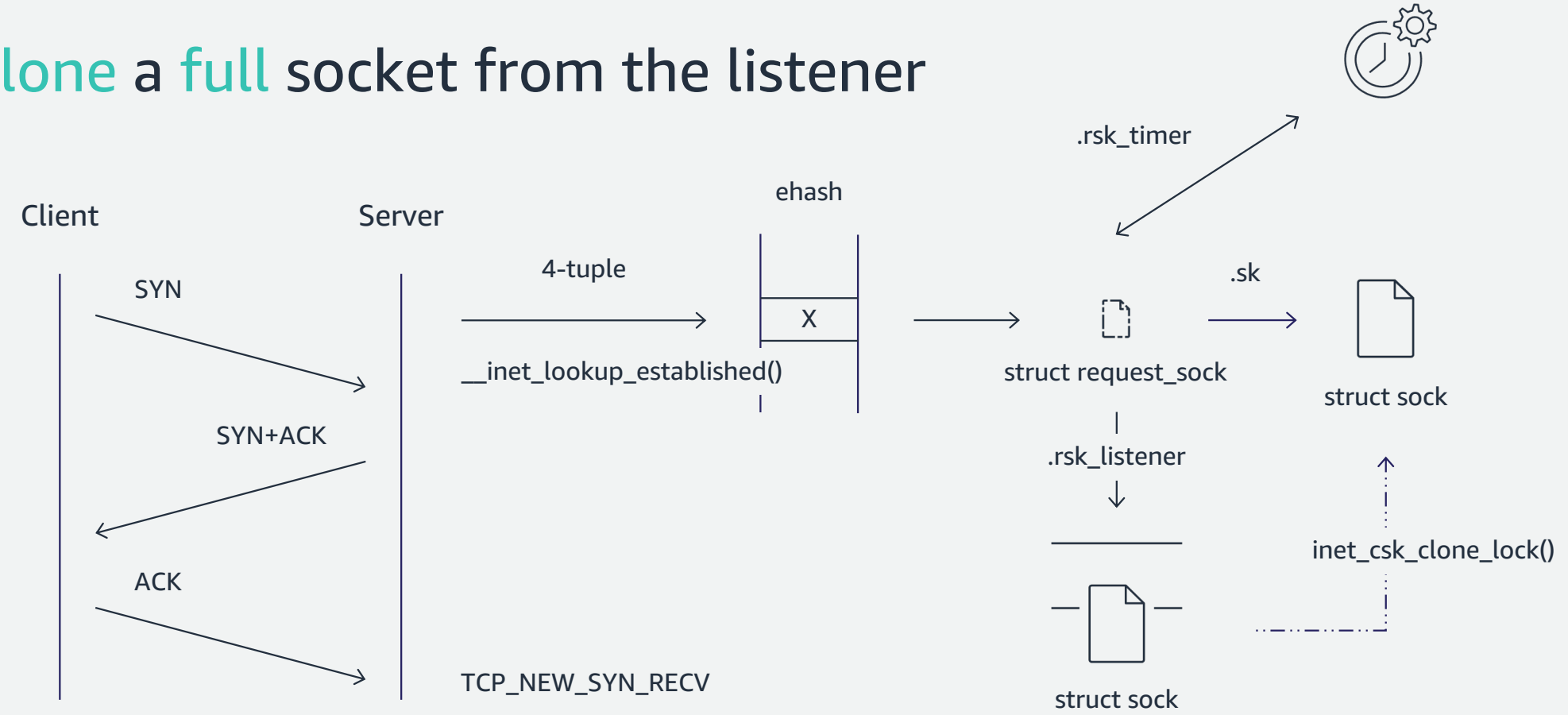
3-way handshake - ACK

1. Look up a mini socket



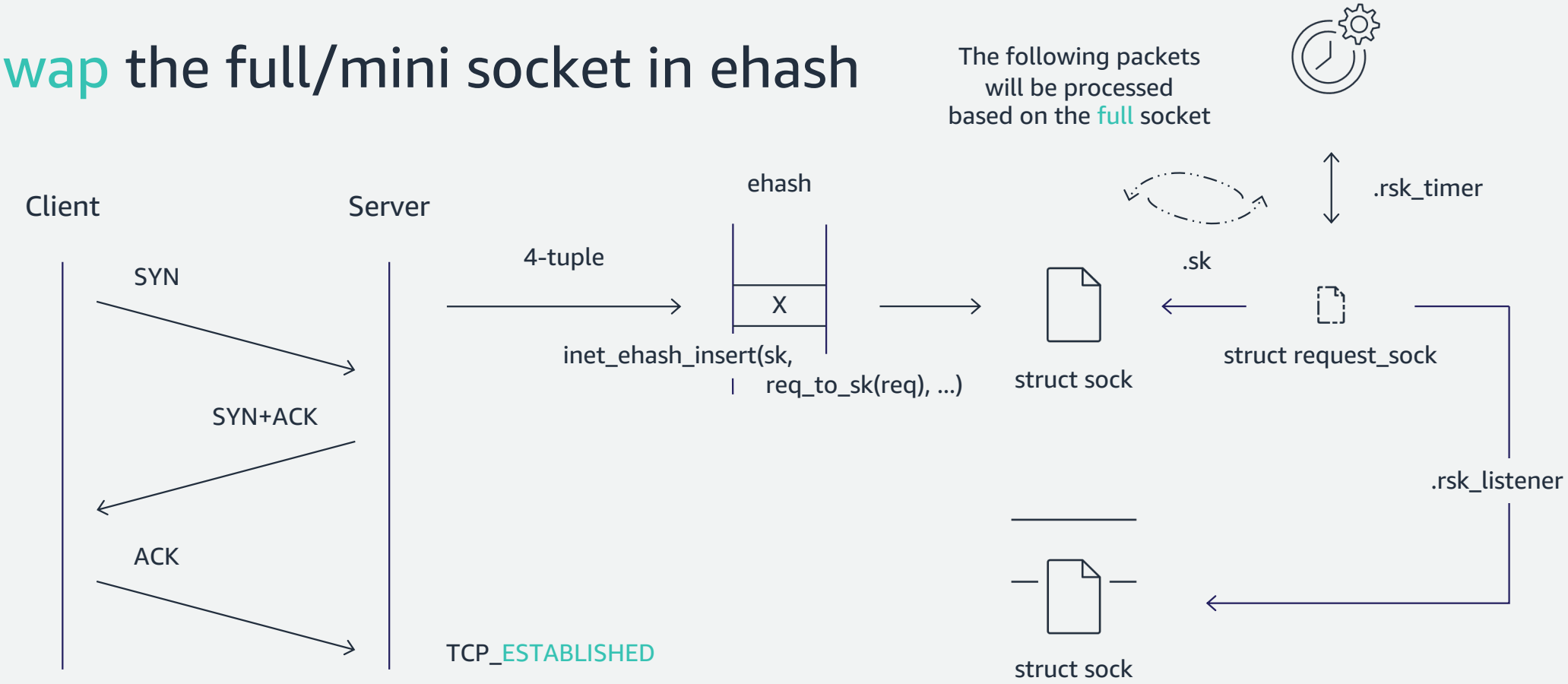
3-way handshake - ACK

2. Clone a full socket from the listener



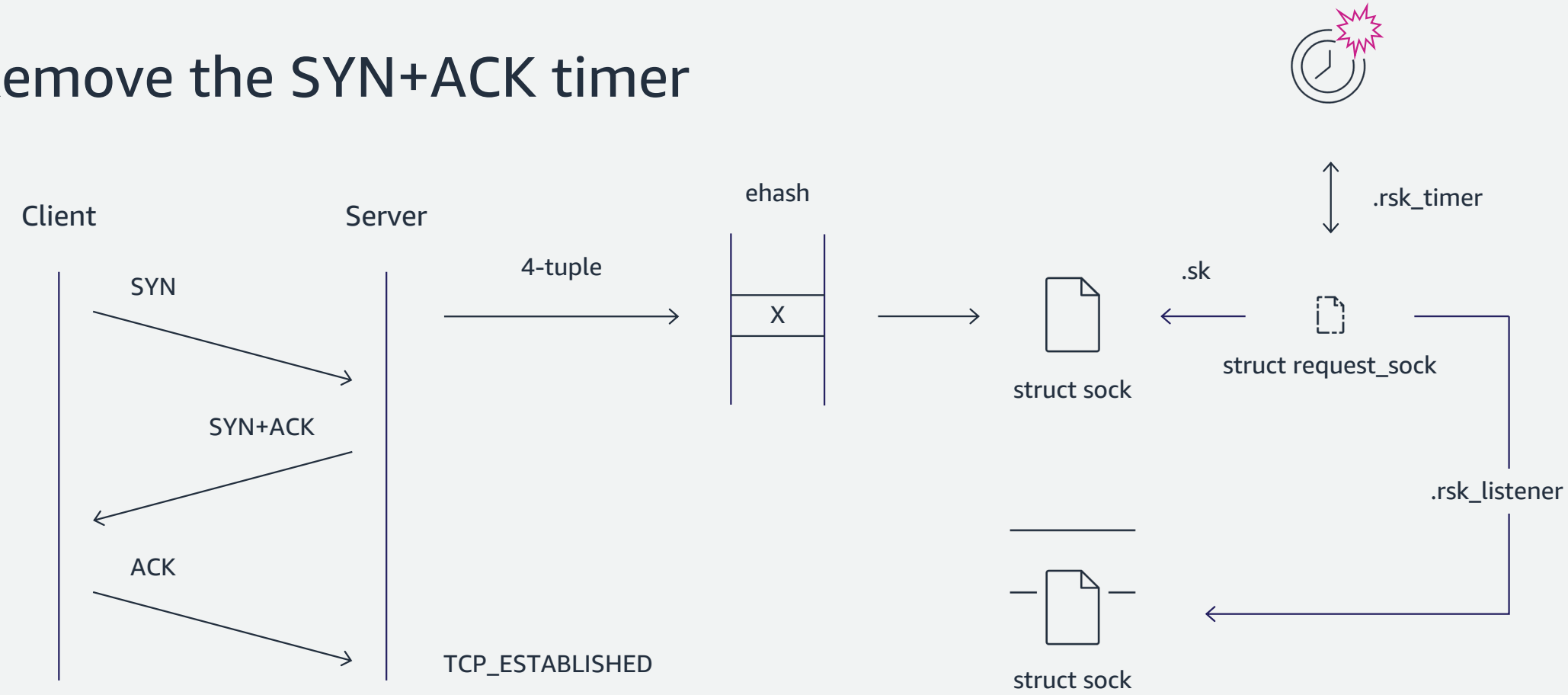
3-way handshake - ACK

3. Swap the full/mini socket in ehash



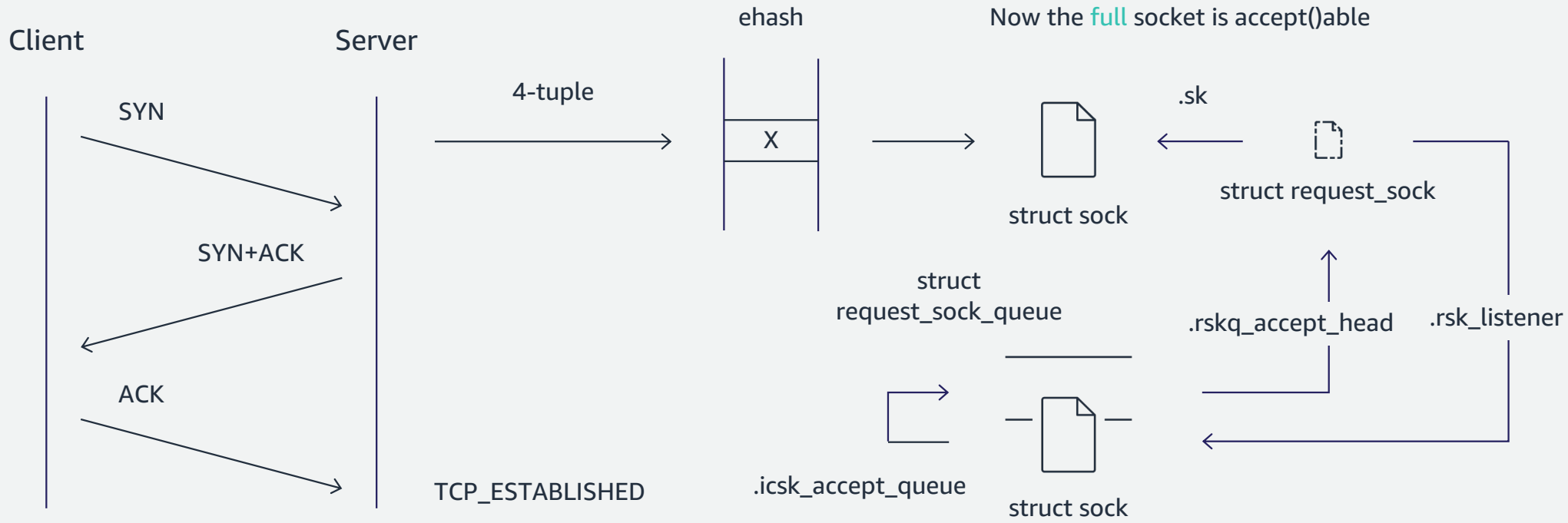
3-way handshake - ACK

4. Remove the SYN+ACK timer



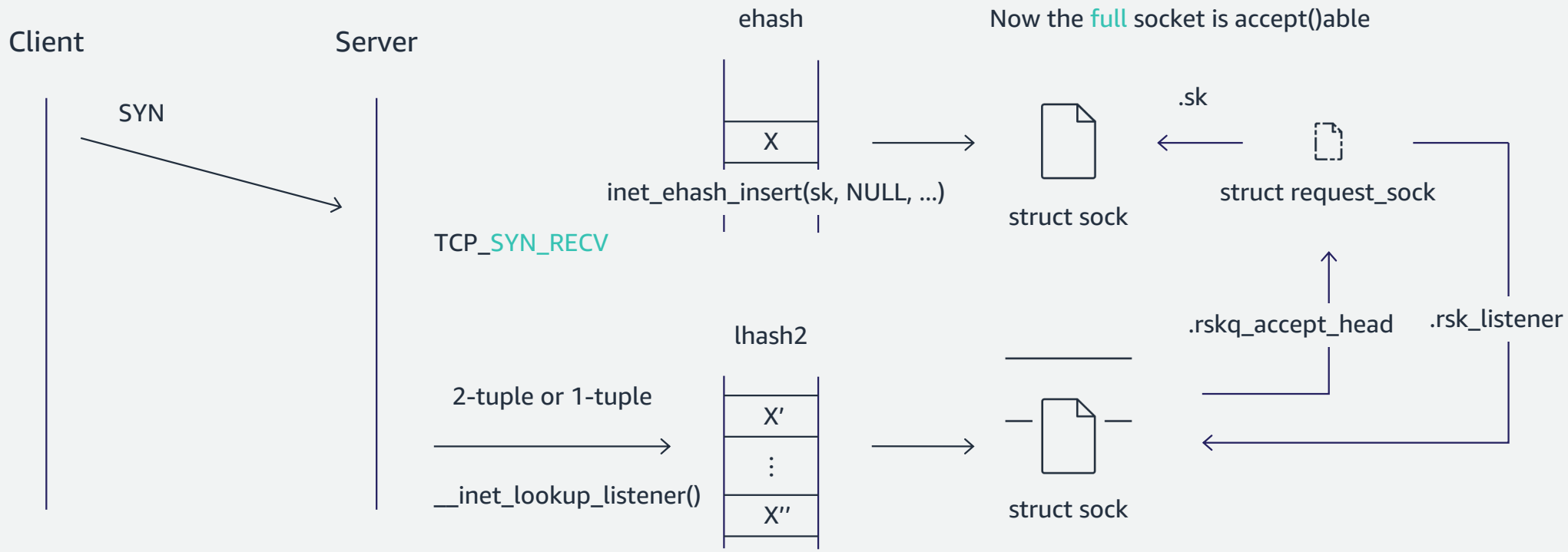
3-way handshake - ACK

5. Put the mini socket into the listener's accept queue

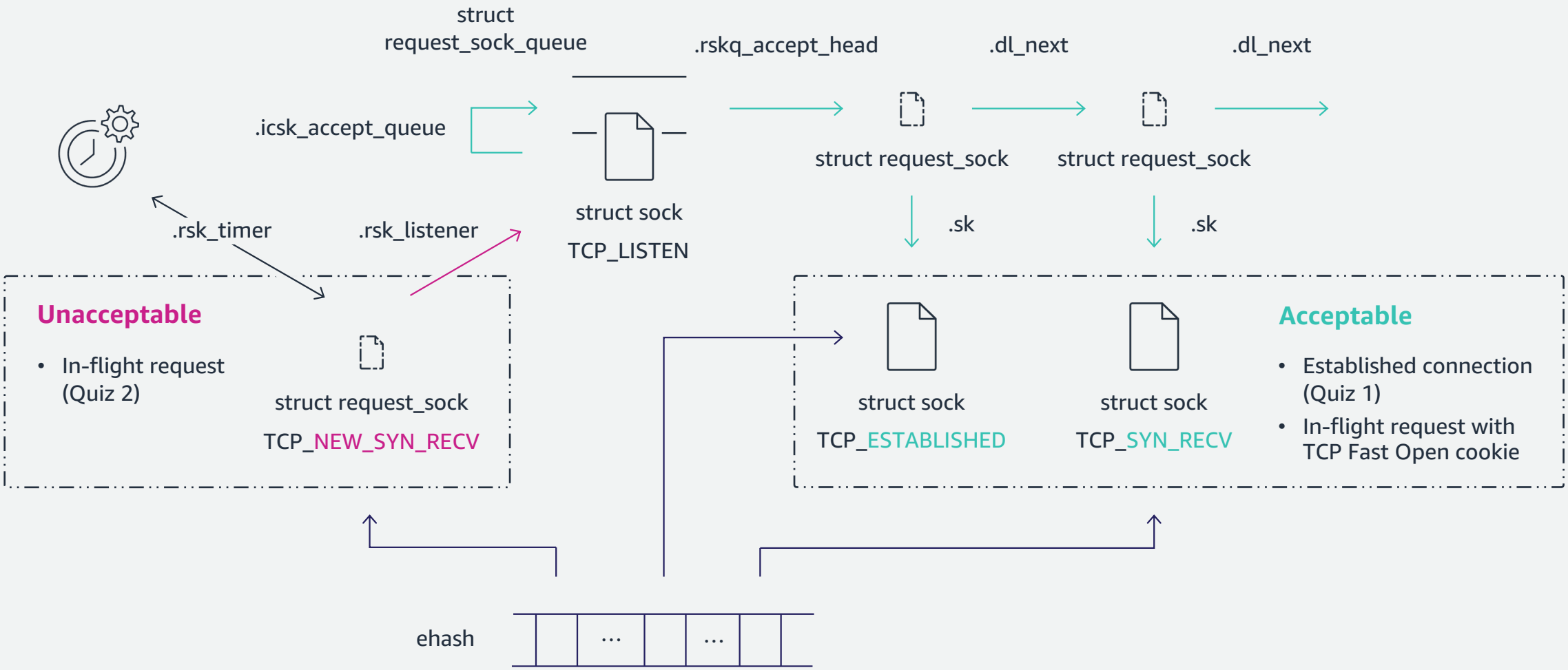


3-way handshake - SYN with TCP Fast Open cookie

All the previous steps are done at once before sending SYN+ACK

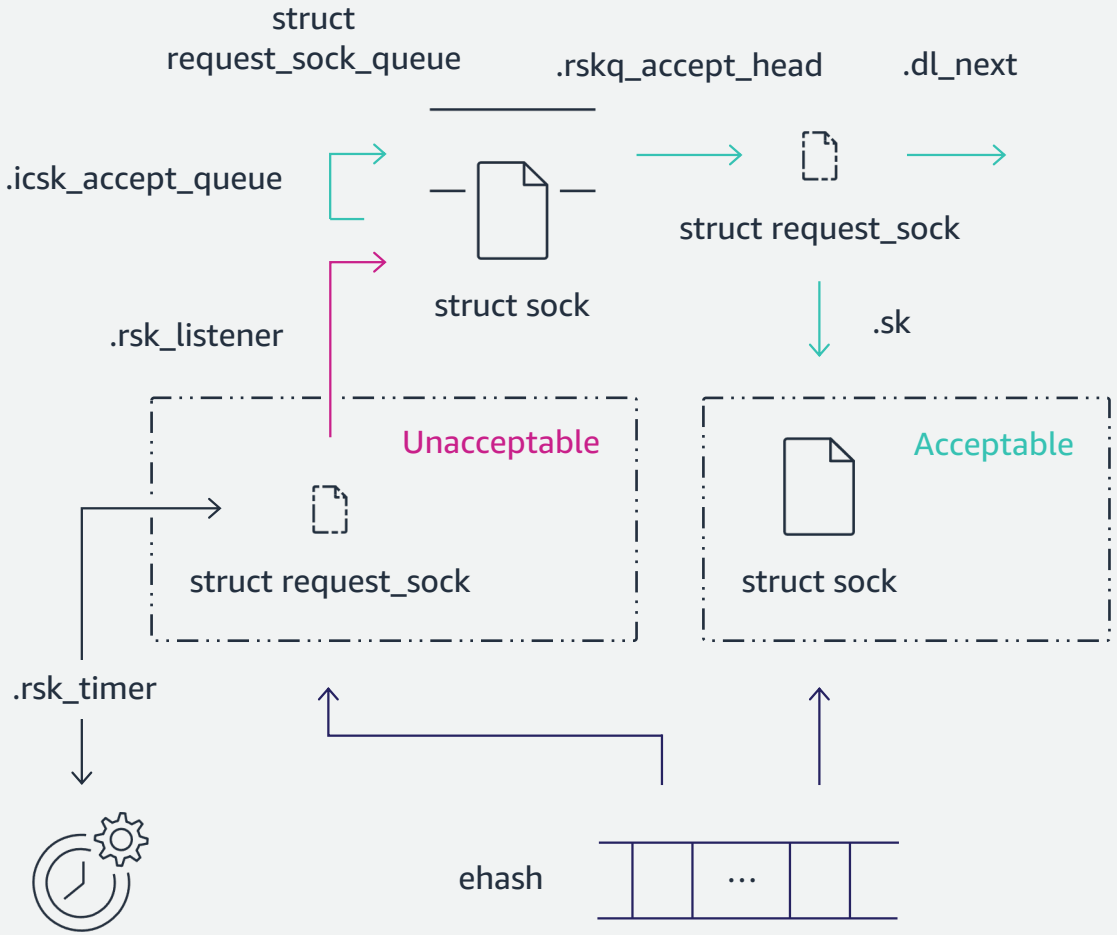


Connection types



Connection types

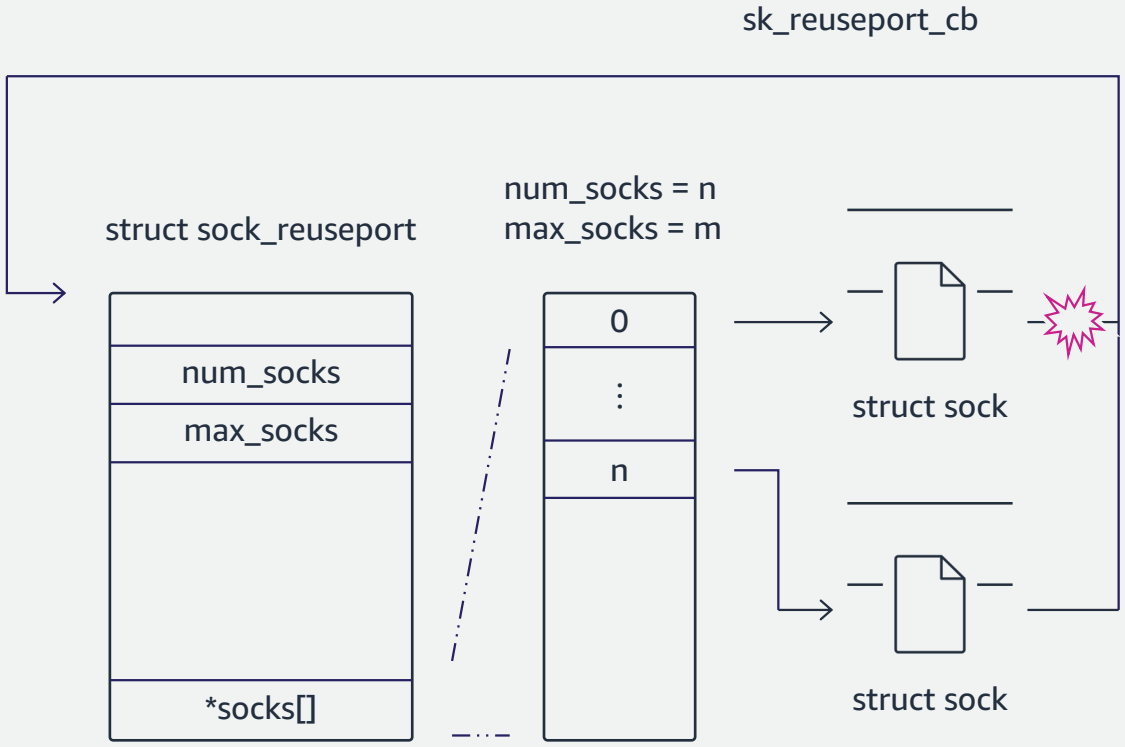
- Acceptable
  - Can be referred from a listener
  - Aborted during close()
- Unacceptable
  - Cannot be referred from a listener
  - Aborted after close()



During close()

1. Remove the listener from the reuseport group

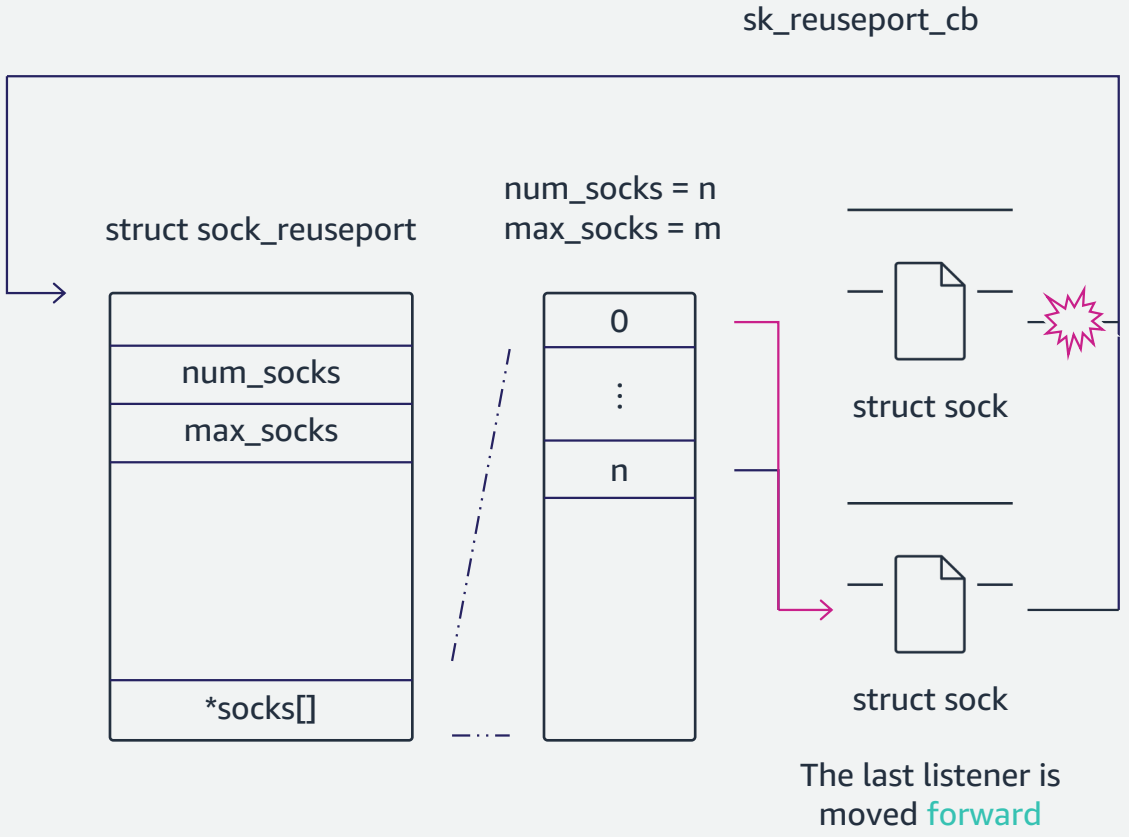
```
void reuseport_detach_sock(struct sock *sk)
{
        struct sock_reuseport *reuse;
        int i;
...
        reuse = rcu_dereference_protected(sk->sk_reuseport_cb,
                                          lockdep_is_held(&reuseport_lock));
...
        rcu_assign_pointer(sk->sk_reuseport_cb, NULL);
...
        for (i = 0; i < reuse->num_socks; i++) {
                if (reuse->socks[i] == sk) {
                        reuse->socks[i] = reuse->socks[reuse->num_socks - 1];
                        reuse->num_socks--;
                        if (reuse->num_socks == 0)
                                call_rcu(&reuse->rcu, reuseport_free_rcu);
                        break;
                }
        }
...
}
```



During close()

1. Remove the listener from the reuseport group

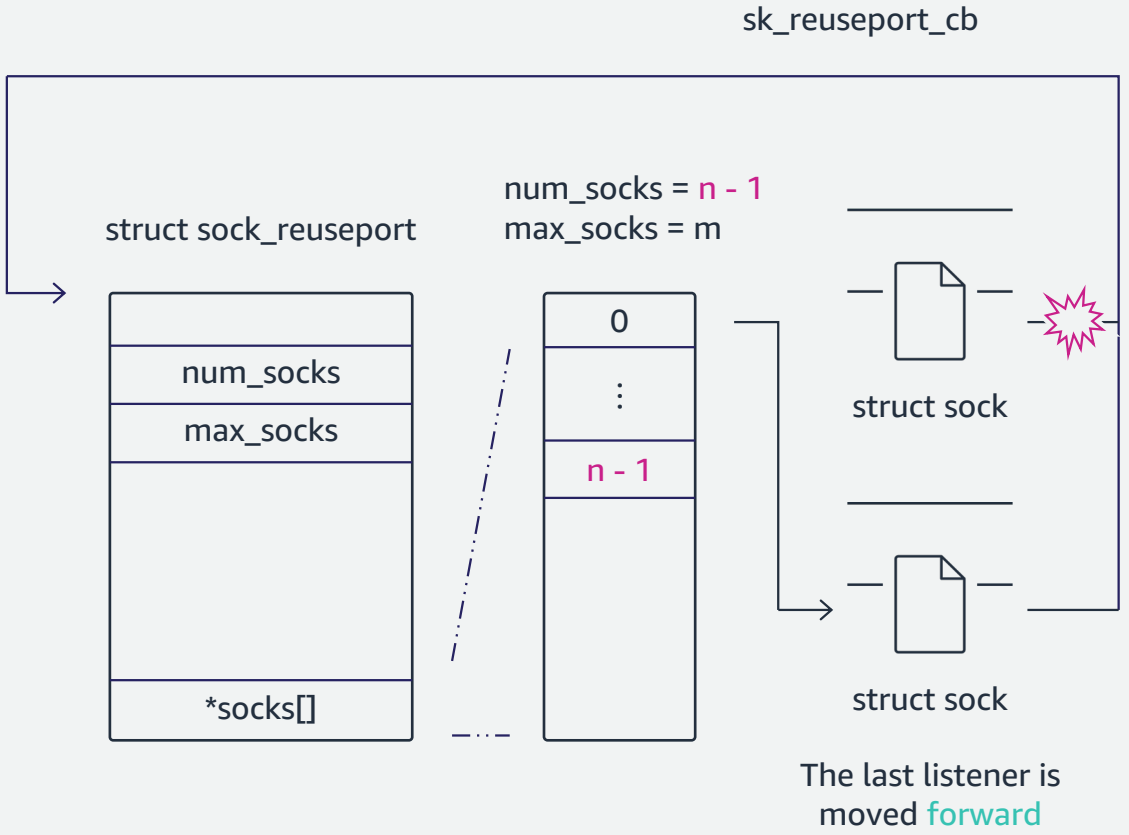
```
void reuseport_detach_sock(struct sock *sk)
{
        struct sock_reuseport *reuse;
        int i;
...
        reuse = rcu_dereference_protected(sk->sk_reuseport_cb,
                                          lockdep_is_held(&reuseport_lock));
...
        rcu_assign_pointer(sk->sk_reuseport_cb, NULL);
...
        for (i = 0; i < reuse->num_socks; i++) {
                if (reuse->socks[i] == sk) {
                        reuse->socks[i] = reuse->socks[reuse->num_socks - 1];
                        reuse->num_socks--;
                        if (reuse->num_socks == 0)
                                call_rcu(&reuse->rcu, reuseport_free_rcu);
                        break;
                }
        }
...
}
```



During close()

1. Remove the listener from the reuseport group

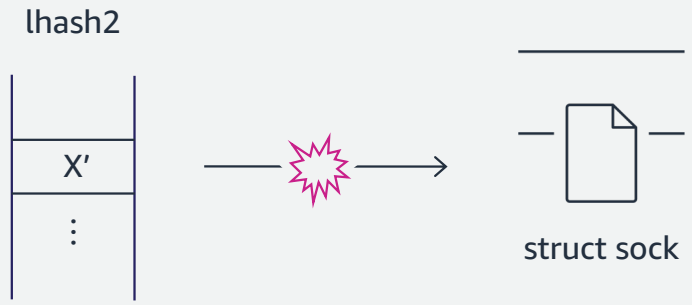
```
void reuseport_detach_sock(struct sock *sk)
{
        struct sock_reuseport *reuse;
        int i;
...
        reuse = rcu_dereference_protected(sk->sk_reuseport_cb,
                                          lockdep_is_held(&reuseport_lock));
...
        rcu_assign_pointer(sk->sk_reuseport_cb, NULL);
...
        for (i = 0; i < reuse->num_socks; i++) {
                if (reuse->socks[i] == sk) {
                        reuse->socks[i] = reuse->socks[reuse->num_socks - 1];
                        reuse->num_socks--;
                        if (reuse->num_socks == 0)
                                call_rcu(&reuse->rcu, reuseport_free_rcu);
                        break;
                }
        }
...
}
```



During close()

2. Remove the listener from lhash2

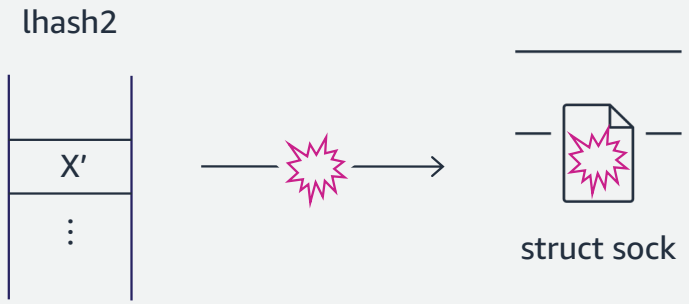
```
void inet_unhash(struct sock *sk)
{
...
        if (rcu_access_pointer(sk->sk_reuseport_cb))
                reuseport_detach_sock(sk);
        if (ilb) {
                inet_unhash2(hashinfo, sk);
...
        }
...
}

static void inet_unhash2(struct inet_hashinfo *h, struct sock *sk)
{
...
        ilb2 = inet_lhash2_bucket_sk(h, sk);
...
        hlist_del_init_rcu(&inet_csk(sk)->icsk_listen_portaddr_node);
...
}
```



During close()

3. Set TCP_CLOSE to the listener's state

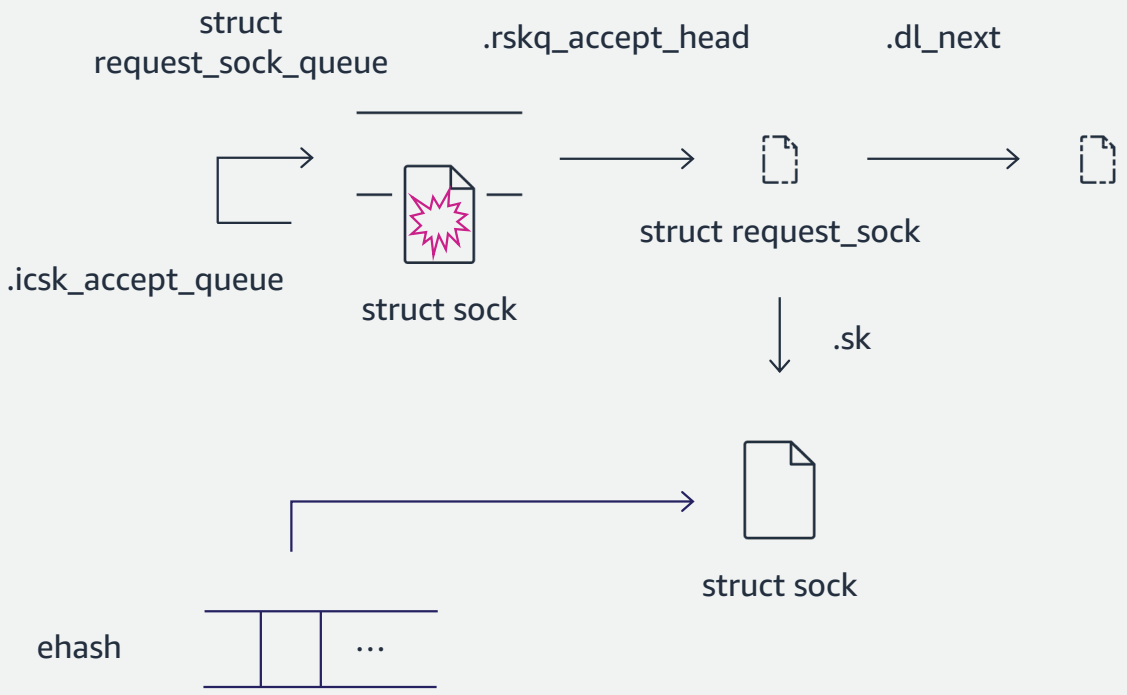
```
void tcp_set_state(struct sock *sk, int state)
{
        int oldstate = sk->sk_state;
...
        switch (state) {
...
        case TCP_CLOSE:
...
                sk->sk_prot->unhash(sk);
...
        }

        /* Change state AFTER socket is unhashed to avoid closed
         * socket sitting in hash tables.
         */
        inet_sk_state_store(sk, state);
}
```



During close()

4. Free the mini/full sockets one by one

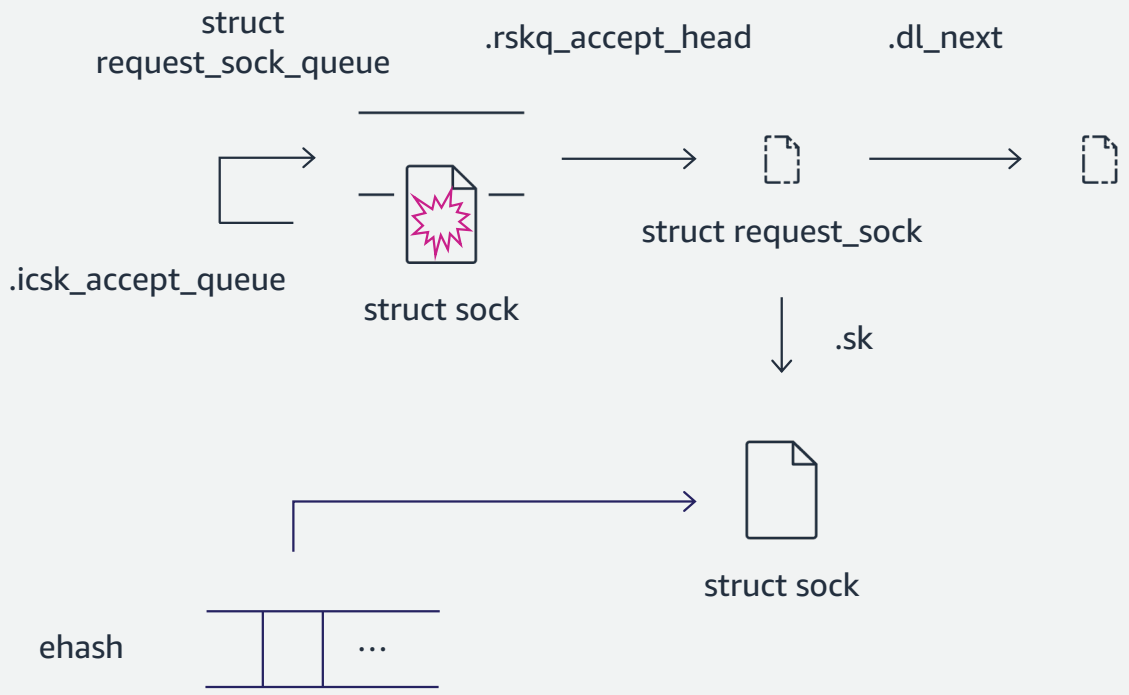
```
void __tcp_close(struct sock *sk, long timeout)
{
...
        if (sk->sk_state == TCP_LISTEN) {
                tcp_set_state(sk, TCP_CLOSE);

                /* Special case. */
                inet_csk_listen_stop(sk);
...
        }
....
}
```



During close()

4. Free the mini/full sockets one by one

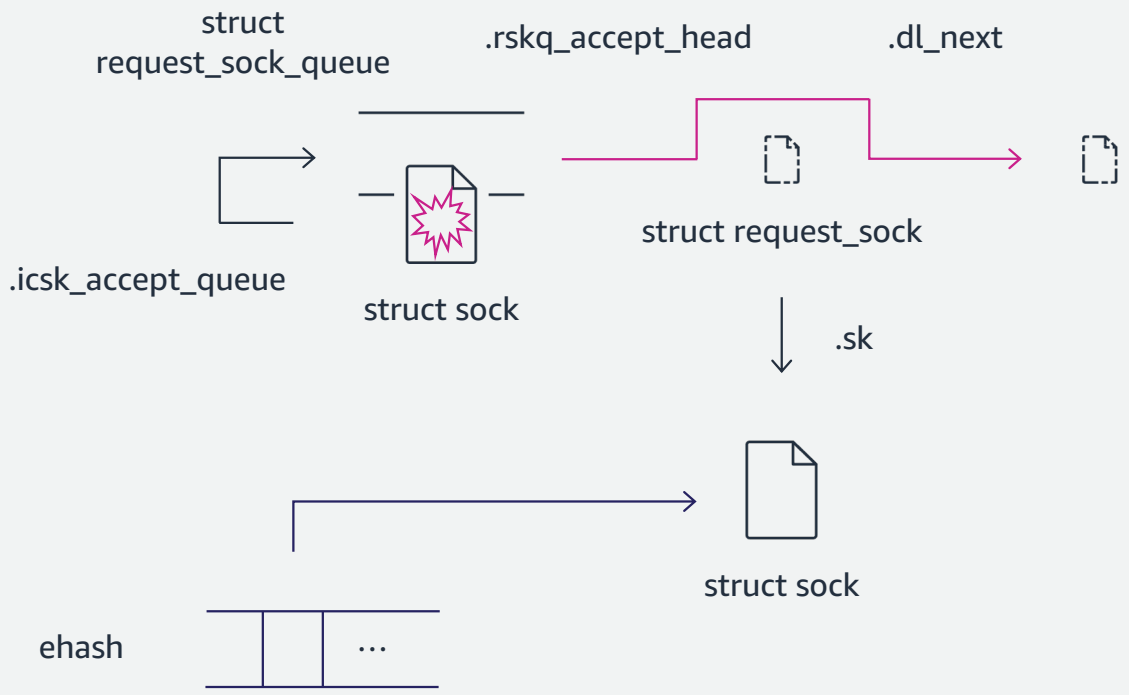
```
void inet_csk_listen_stop(struct sock *sk)
{
...
        while ((req = reqsk_queue_remove(queue, sk)) != NULL) {
                struct sock *child = req->sk;
...
                sock_hold(child);
...
                inet_child_forget(sk, req, child);
                reqsk_put(req);
...
                sock_put(child);
...
        }
...
}
```



During close()

4. Free the mini/full sockets one by one

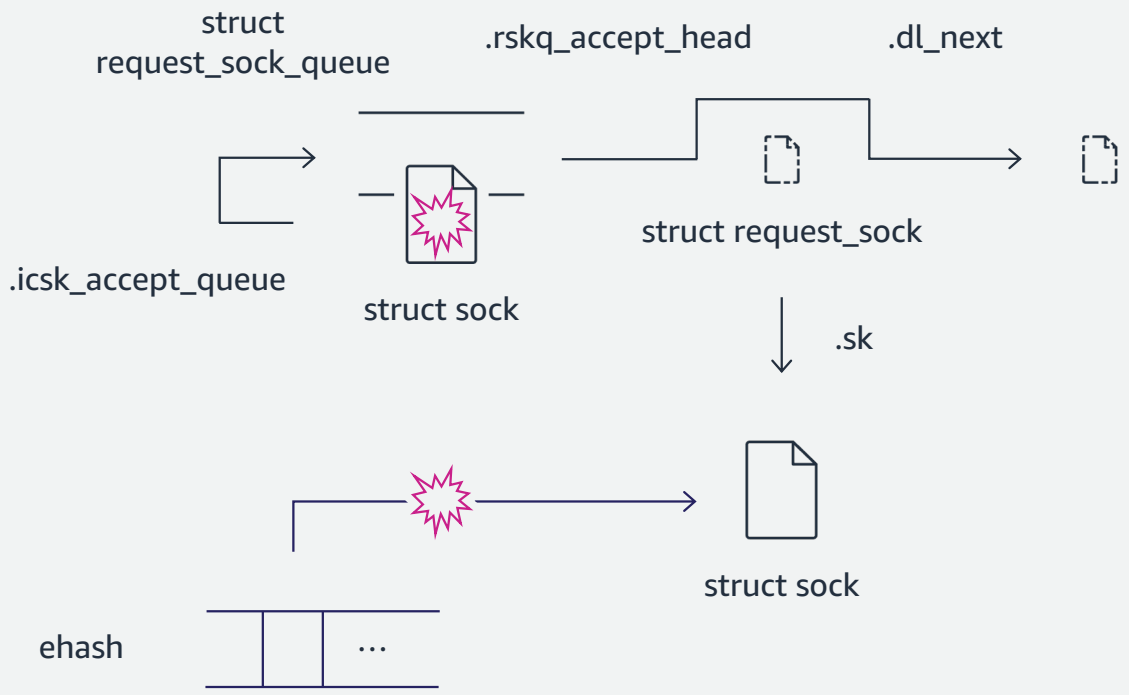
```
void inet_csk_listen_stop(struct sock *sk)
{
...
        while ((req = reqsk_queue_remove(queue, sk)) != NULL) {
                struct sock *child = req->sk;
...
                sock_hold(child);
...
                inet_child_forget(sk, req, child);
                reqsk_put(req);
...
                sock_put(child);
...
        }
...
}
```



During close()

4. Free the mini/full sockets one by one

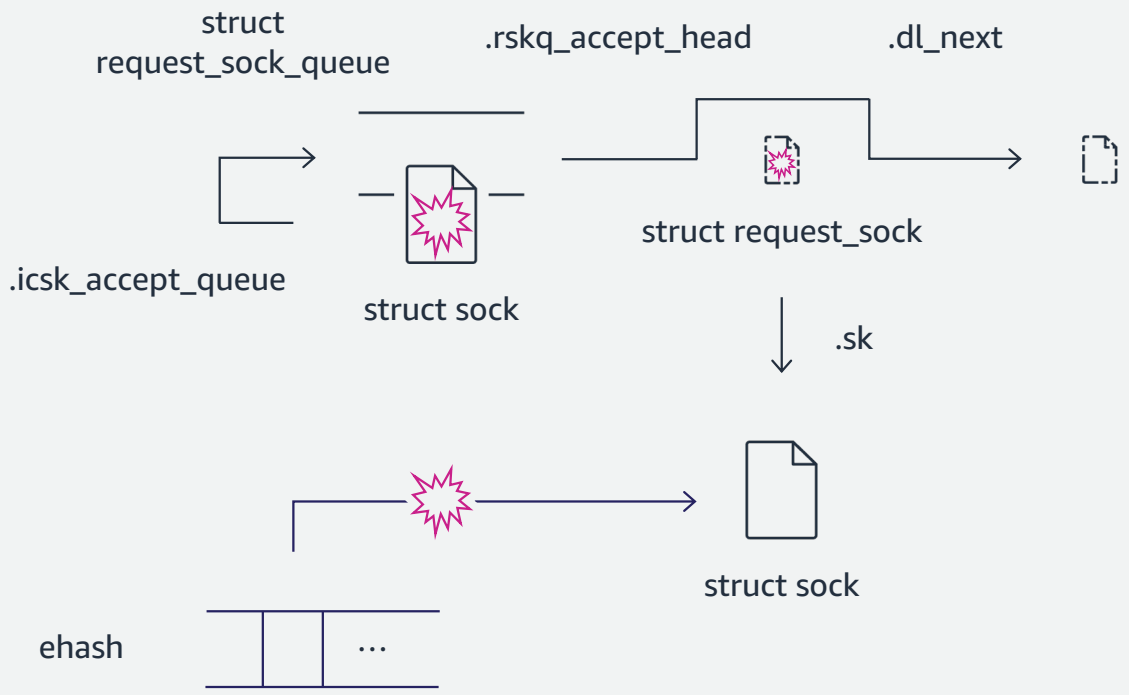
```
void inet_csk_listen_stop(struct sock *sk)
{
...
        while ((req = reqsk_queue_remove(queue, sk)) != NULL) {
                struct sock *child = req->sk;
...
                sock_hold(child);
...
                inet_child_forget(sk, req, child);
                reqsk_put(req);
...
                sock_put(child);
...
        }
...
}
```



During close()

4. Free the mini/full sockets one by one

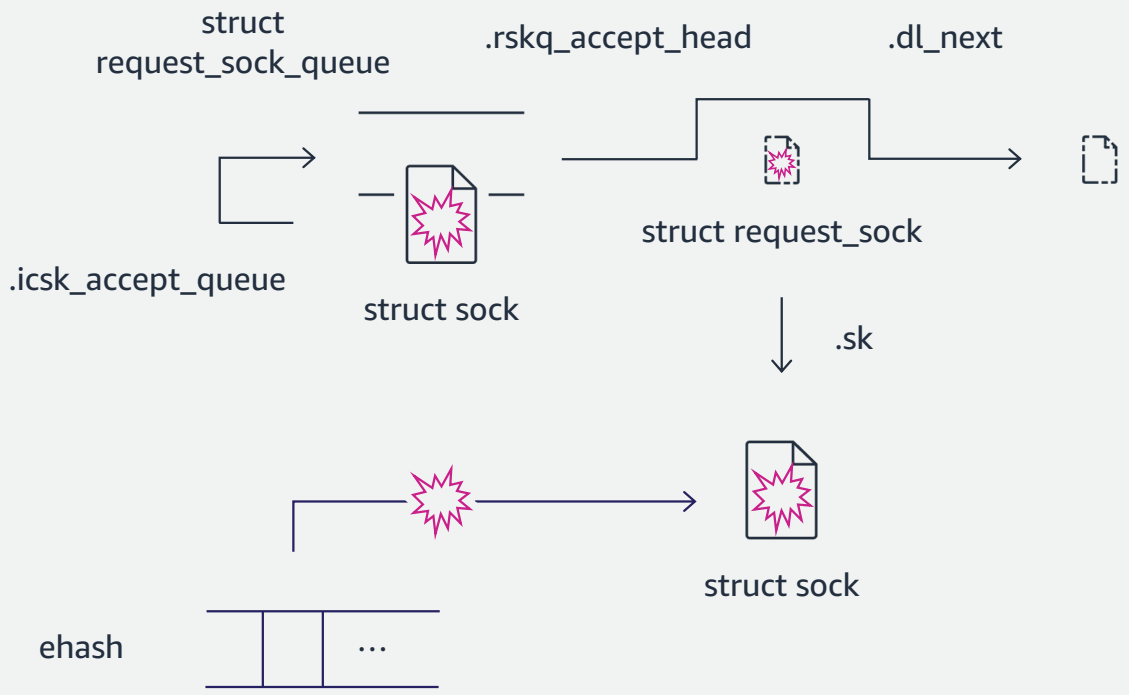
```
void inet_csk_listen_stop(struct sock *sk)
{
...
        while ((req = reqsk_queue_remove(queue, sk)) != NULL) {
                struct sock *child = req->sk;
...
                sock_hold(child);
...
                inet_child_forget(sk, req, child);
                reqsk_put(req);
...
                sock_put(child);
...
        }
...
}
```



During close()

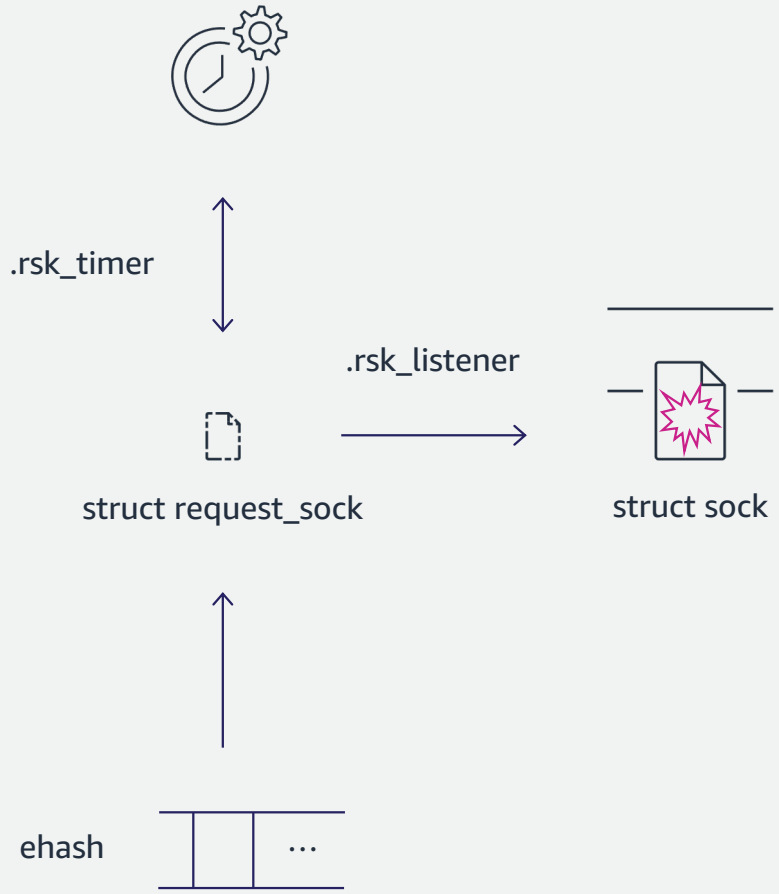
4. Free the mini/full sockets one by one

```
void inet_csk_listen_stop(struct sock *sk)
{
...
        while ((req = reqsk_queue_remove(queue, sk)) != NULL) {
                struct sock *child = req->sk;
...
                sock_hold(child);
...
                inet_child_forget(sk, req, child);
                reqsk_put(req);
...
                sock_put(child);
...
        }
...
}
```



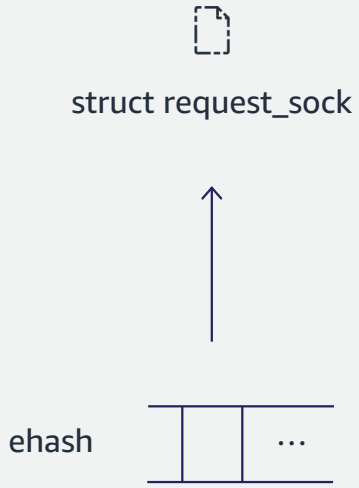
After close()

- The mini socket is freed when
  - Receiving ACK
  - Retransmitting SYN+ACK



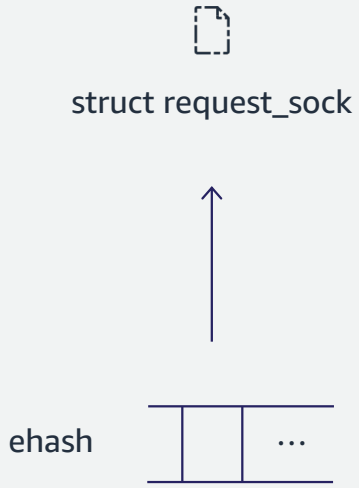
After close() - Receiving ACK

```
int tcp_v4_rcv(struct sk_buff *skb)
{
...
lookup:
        sk = __inet_lookup_skb(&tcp_hashinfo, skb, __tcp_hdrlen(th), th->source,
                               th->dest, sdif, &refcounted);
...
        if (sk->sk_state == TCP_NEW_SYN_RECV) {
                struct request_sock *req = inet_reqsk(sk);
...
                sk = req->rsk_listener;
...
                if (unlikely(sk->sk_state != TCP_LISTEN)) {
                        inet_csk_reqsk_queue_drop_and_put(sk, req);
                        goto lookup;
                }
...
        }
...
}
```



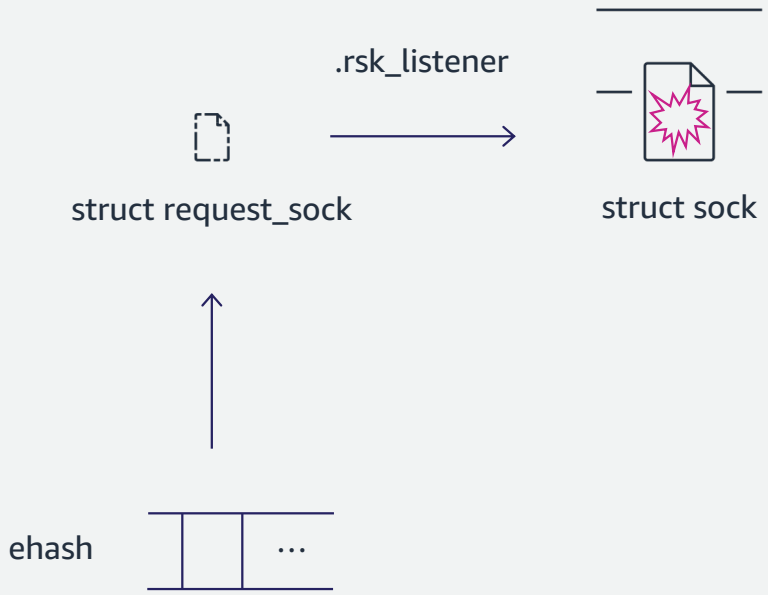
After close() - Receiving ACK

```
int tcp_v4_rcv(struct sk_buff *skb)
{
...
lookup:
        sk = __inet_lookup_skb(&tcp_hashinfo, skb, __tcp_hdrlen(th), th->source,
                               th->dest, sdif, &refcounted);
...
        if (sk->sk_state == TCP_NEW_SYN_RECV) {
                struct request_sock *req = inet_reqsk(sk);
...
                sk = req->rsk_listener;
...
                if (unlikely(sk->sk_state != TCP_LISTEN)) {
                        inet_csk_reqsk_queue_drop_and_put(sk, req);
                        goto lookup;
                }
...
        }
...
}
```



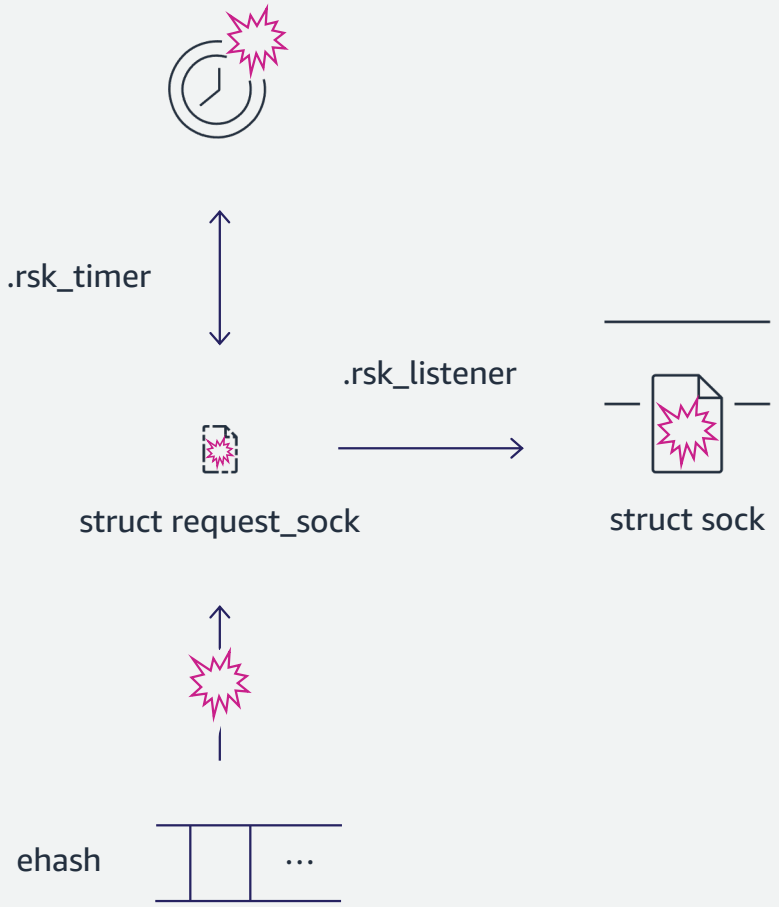
After close() - Receiving ACK

```
int tcp_v4_rcv(struct sk_buff *skb)
{
...
lookup:
        sk = __inet_lookup_skb(&tcp_hashinfo, skb, __tcp_hdrlen(th), th->source,
                               th->dest, sdif, &refcounted);
...
        if (sk->sk_state == TCP_NEW_SYN_RECV) {
                struct request_sock *req = inet_reqsk(sk);
...
                sk = req->rsk_listener;
...
                if (unlikely(sk->sk_state != TCP_LISTEN)) {
                        inet_csk_reqsk_queue_drop_and_put(sk, req);
                        goto lookup;
                }
...
        }
...
}
```



After close() - Receiving ACK

```
int tcp_v4_rcv(struct sk_buff *skb)
{
...
lookup:
        sk = __inet_lookup_skb(&tcp_hashinfo, skb, __tcp_hdrlen(th), th->source,
                               th->dest, sdif, &refcounted);
...
        if (sk->sk_state == TCP_NEW_SYN_RECV) {
                struct request_sock *req = inet_reqsk(sk);
...
                sk = req->rsk_listener;
...
                if (unlikely(sk->sk_state != TCP_LISTEN)) {
                        inet_csk_reqsk_queue_drop_and_put(sk, req);
                        goto lookup;
                }
...
        }
...
}
```

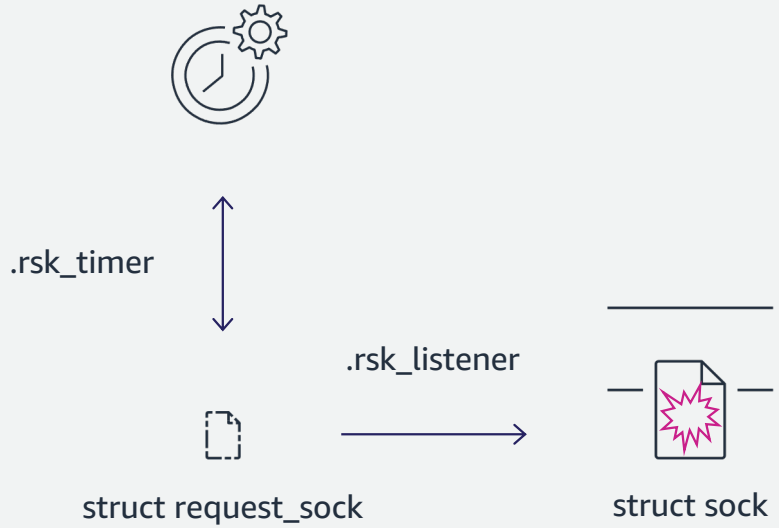


After close() - Retransmitting SYN+ACK

```
static void reqsk_timer_handler(struct timer_list *t)
{
        struct request_sock *req = from_timer(req, t, rsk_timer);
        struct sock *sk_listener = req->rsk_listener;
...
        if (inet_sk_state_load(sk_listener) != TCP_LISTEN)
                goto drop;
...
drop:
        inet_csk_reqsk_queue_drop_and_put(sk_listener, req);
}
```

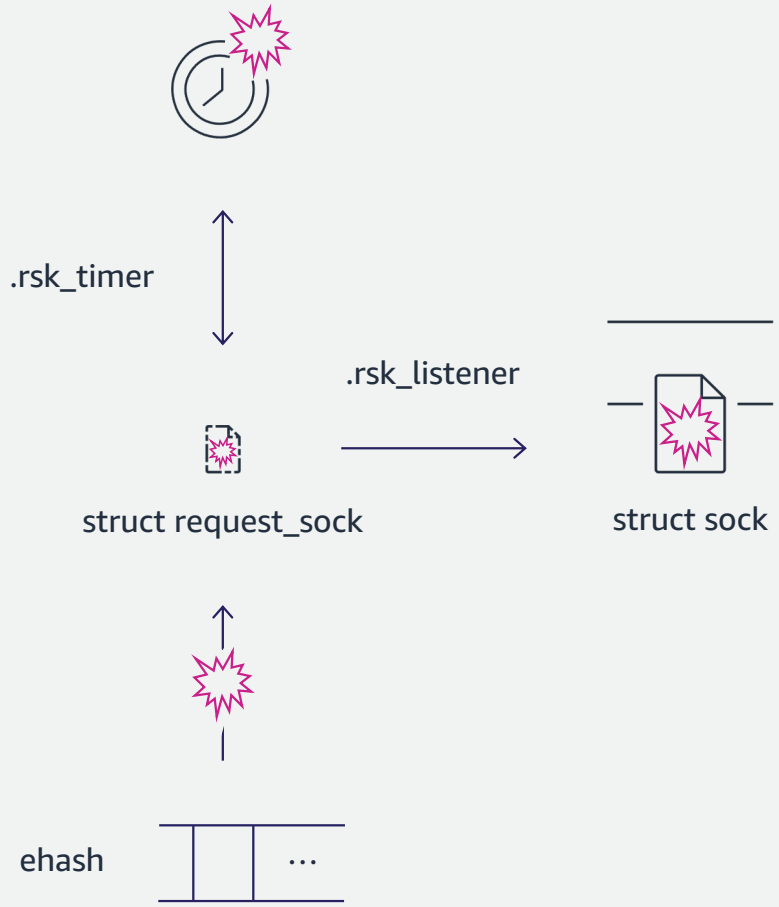
After close() - Retransmitting SYN+ACK

```
static void reqsk_timer_handler(struct timer_list *t)
{
        struct request_sock *req = from_timer(req, t, rsk_timer);
        struct sock *sk_listener = req->rsk_listener;
...
        if (inet_sk_state_load(sk_listener) != TCP_LISTEN)
                goto drop;
...
drop:
        inet_csk_reqsk_queue_drop_and_put(sk_listener, req);
}
```



After close() - Retransmitting SYN+ACK

```
static void reqsk_timer_handler(struct timer_list *t)
{
        struct request_sock *req = from_timer(req, t, rsk_timer);
        struct sock *sk_listener = req->rsk_listener;
...
        if (inet_sk_state_load(sk_listener) != TCP_LISTEN)
                goto drop;
...
drop:
        inet_csk_reqsk_queue_drop_and_put(sk_listener, req);
}
```



Where SO_REUSEPORT works

- SO_REUSEPORT works
  - Around the socket layer with the reuseport group
  - When selecting a listener for a mini socket
- Other behaviour is the same with the no SO_REUSEPORT case
  - Simplify the implementation
  - Connection failures are reasonable, but unacceptable

How to make it acceptable

Socket migration

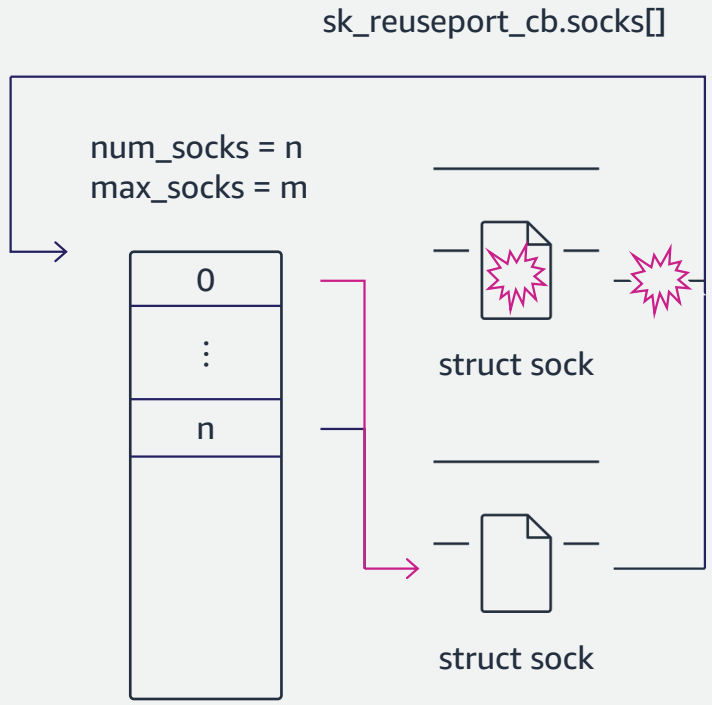
- The mini/full sockets are freed up at
  - inet_csk_listen_stop()
  - tcp_v4_rcv()
  - reqsk_timer_handler()
- Check if another socket is listening on the port
  - If not, abort the mini/full sockets
  - If exists, migrate the mini/full sockets to another listener

How to find another listener on the port

- Lookup in the reuseport group?
  - reuseport_detach_sock()
  - reuseport_grow()

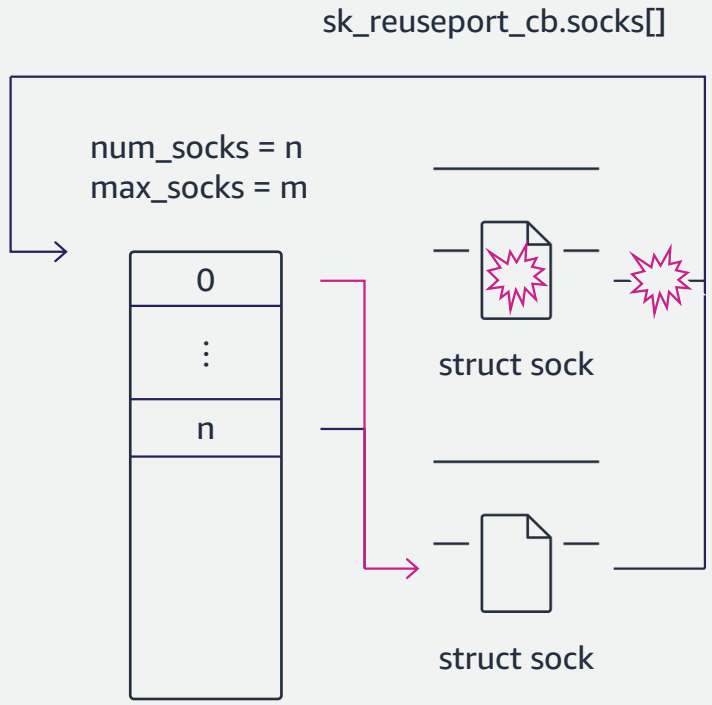
reuseport_detach_sock()

- Called just after close()
  - sk_reuseport_cb will be NULL in all the connection failure paths



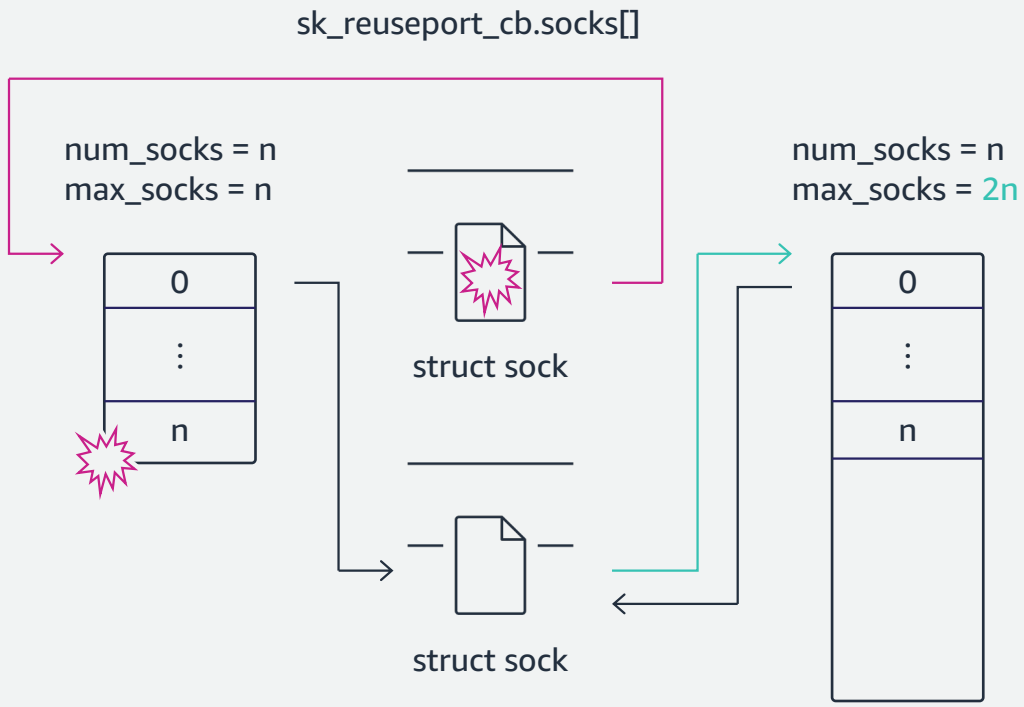
reuseport_detach_sock()

- Called just after close()
  - sk_reuseport_cb will be NULL in all the connection failure paths
- Called just before freeing a listener
  - Called in sk_destruct()
  - After all of its mini sockets are freed
  - Setting NULL can be delayed



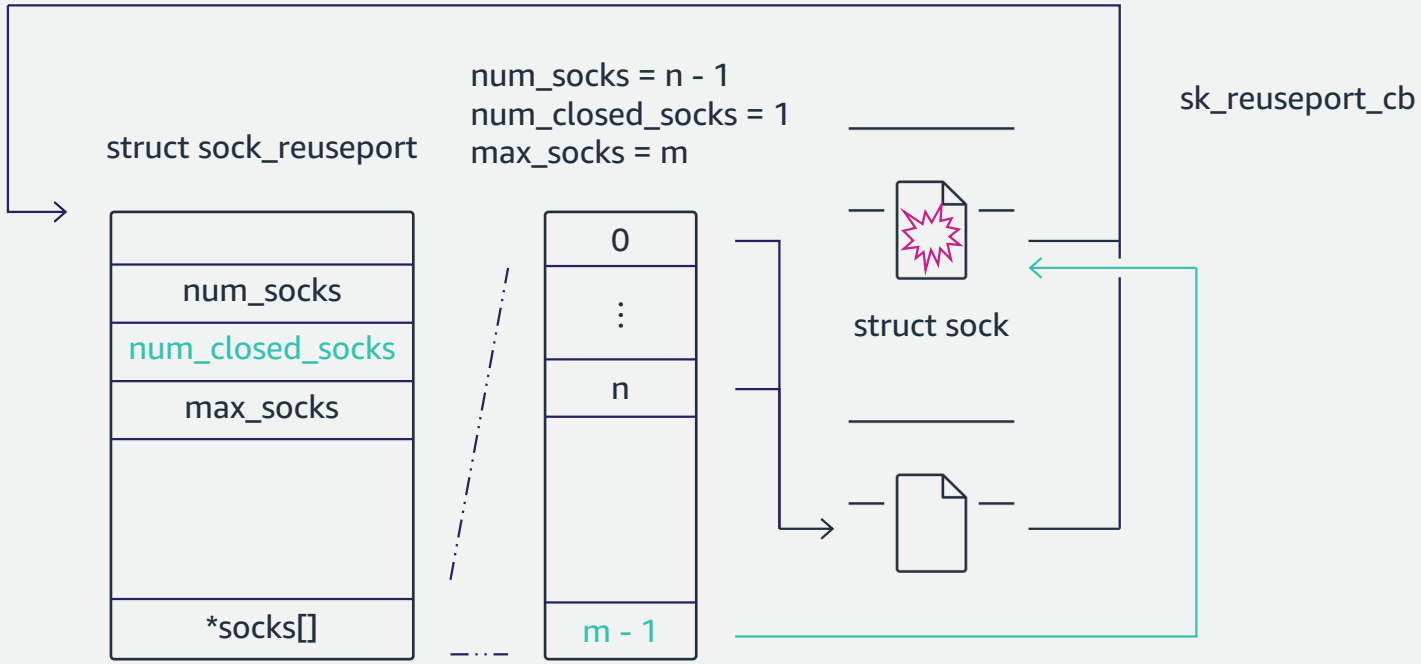
reuseport_grow()

- Called when the number of listeners overflows max_socks
- Allocate a new reuseport group and free the old one
  - The old sk_reuseport_cb can be stale
  - The closed listener must stay in the group to be copied to the new one



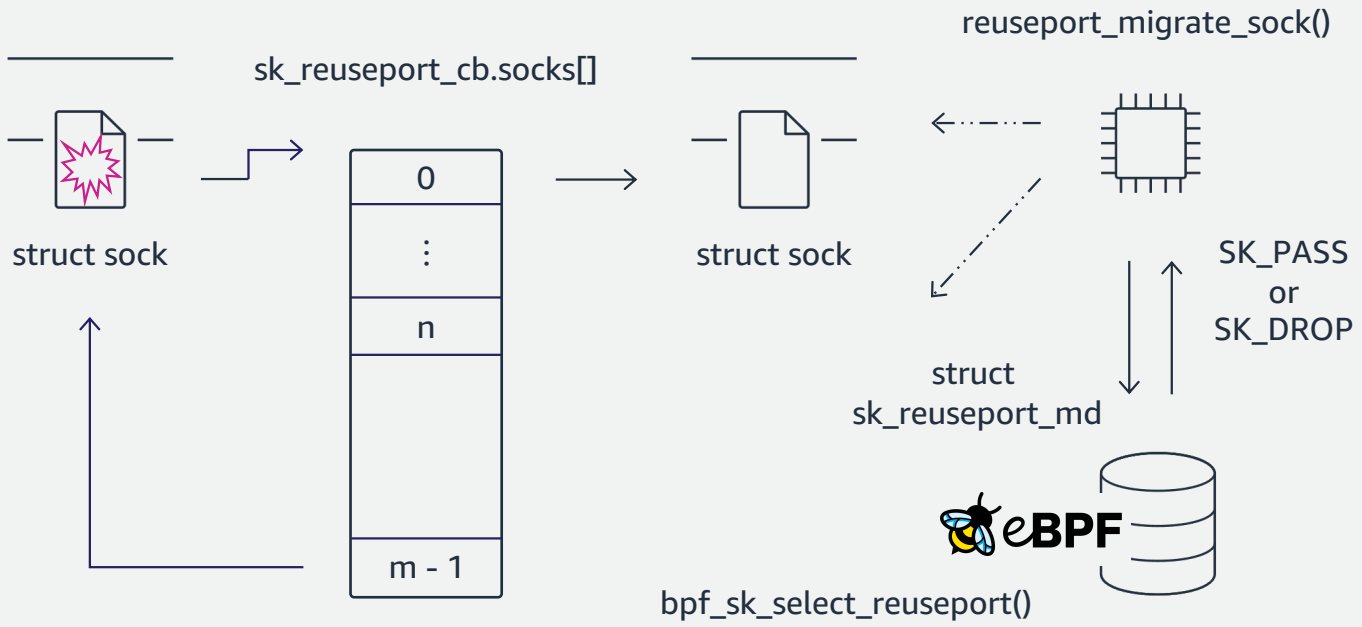
reuseport_stop_listen_sock()

- Replace the first reuseport_detach_sock() call
- Move the closed listener backward in socks[]



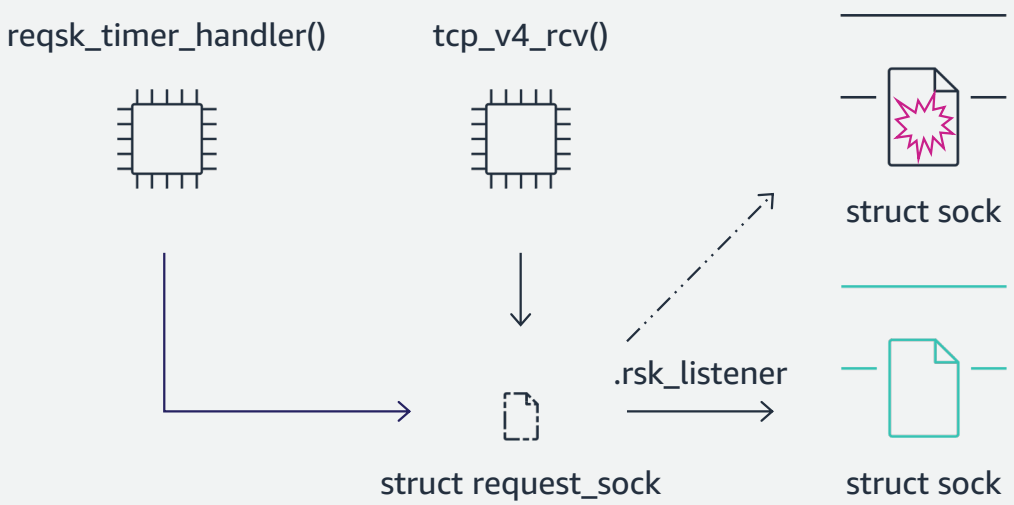
How to find another listener on the port

- reuseport_migrate_sock()
  - Called while/after closing a listener
  - Select a new listener from the reuseport group
    1. By BPF
    2. By random (default)



How to migrate sockets

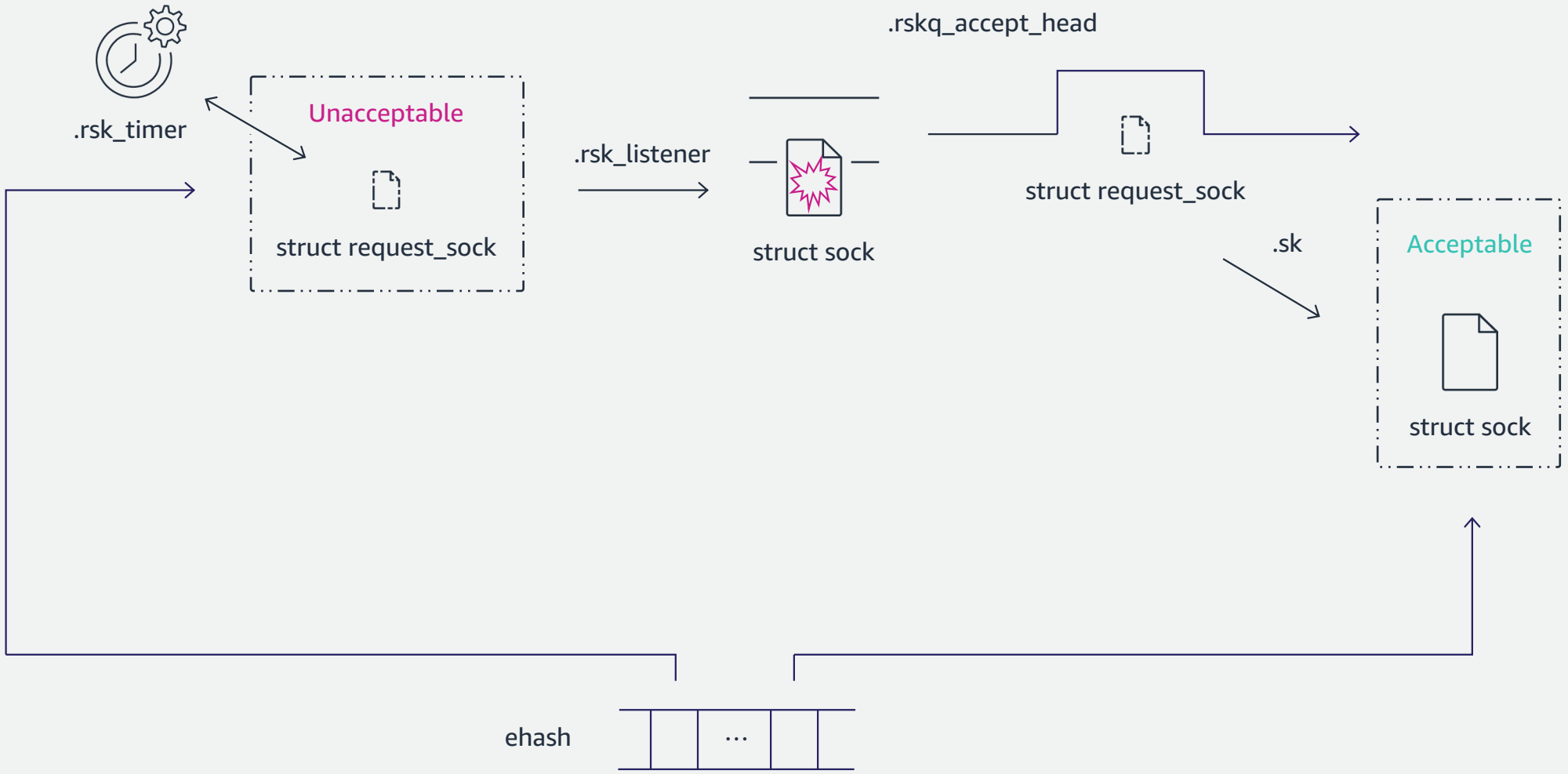
- Rewrite rsk_listener directly?
- Another CPU may be referring to the mini socket in
  - tcp_v4_rcv()
  - reqsk_timer_handler()



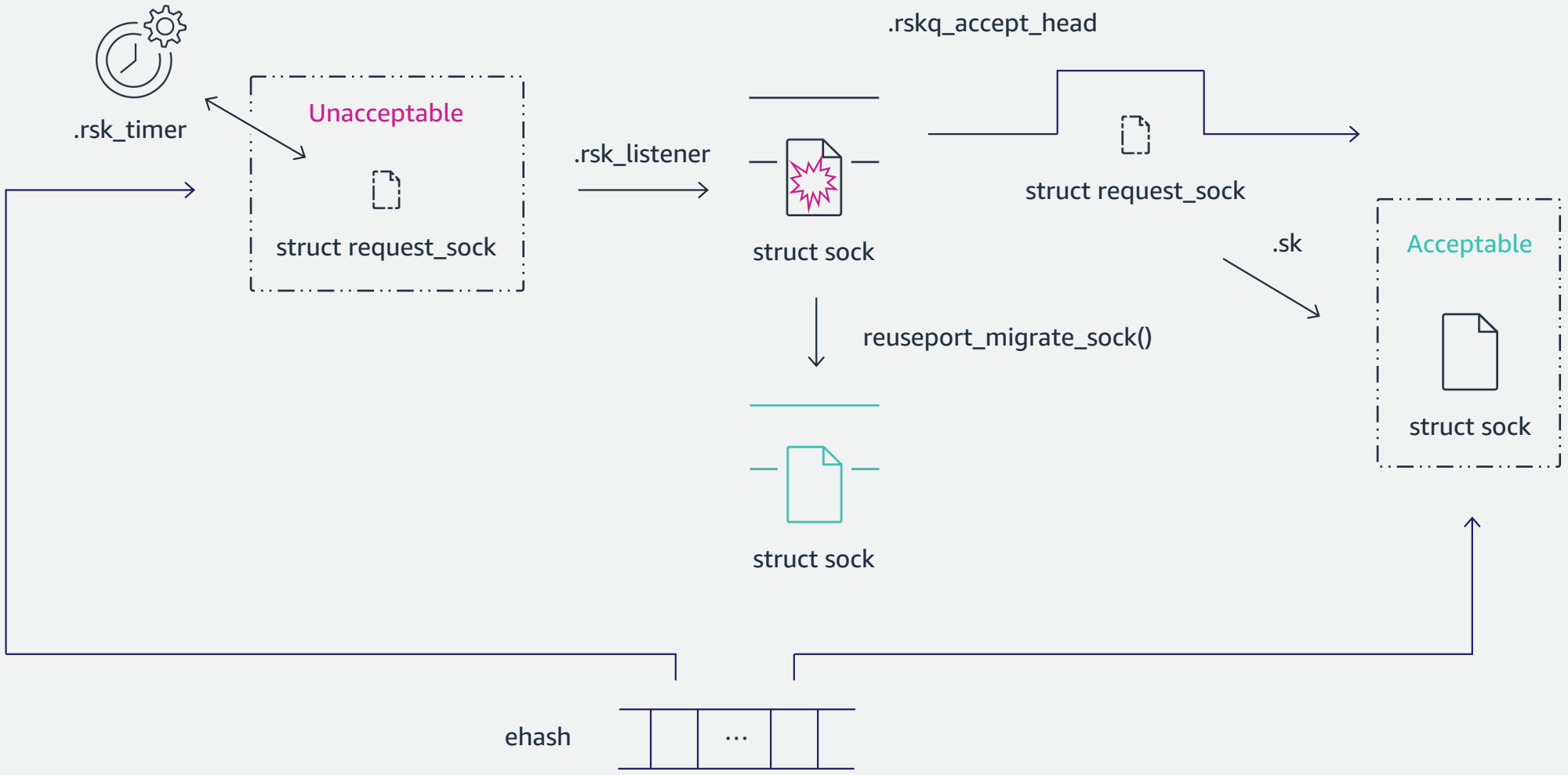
How to migrate sockets

- Clone request_sock and rewrite rsk_listener
- Put the cloned mini socket into
  - The new listener’s accept queue
  - ehash

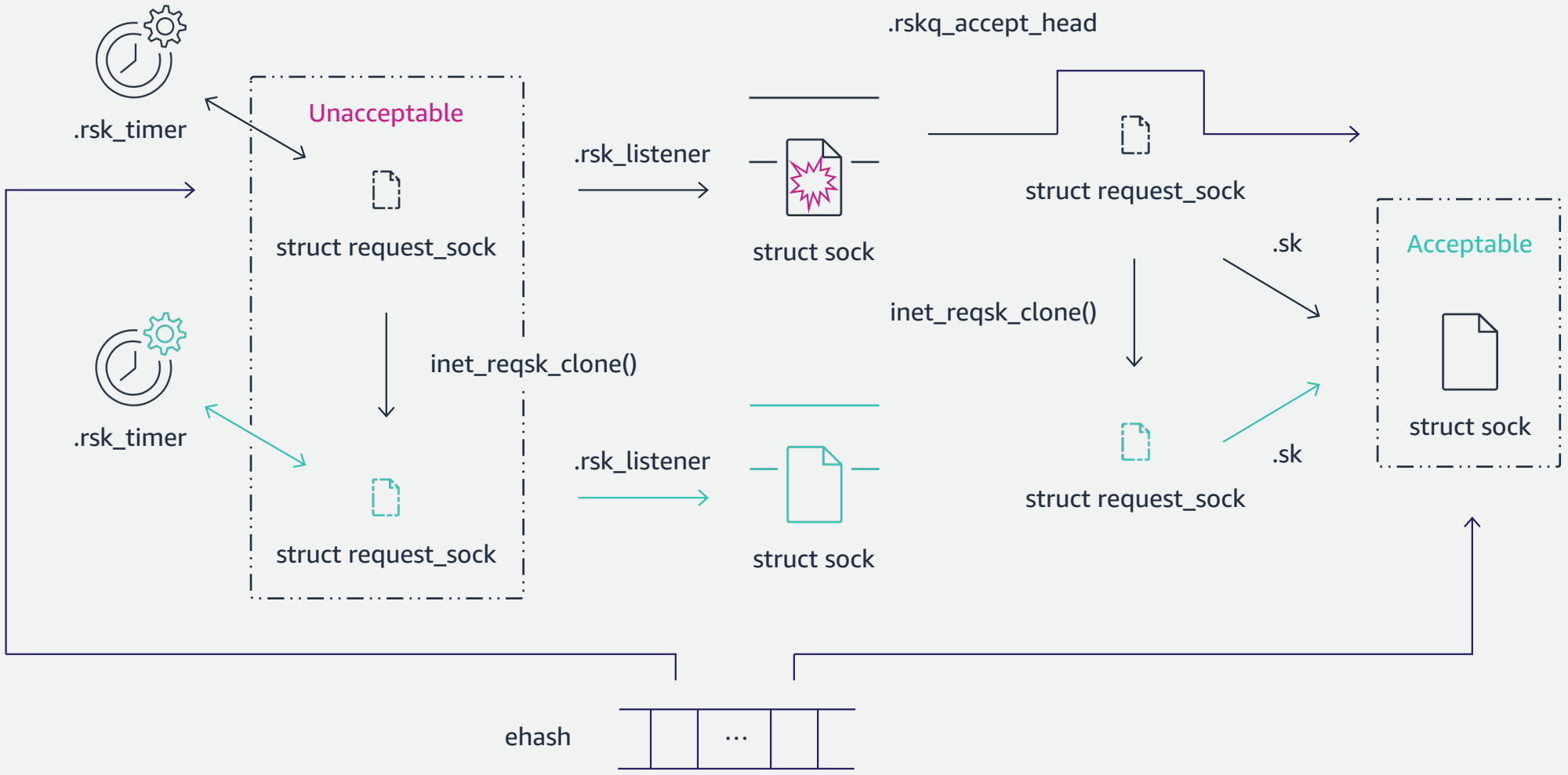
Migrate request_sock



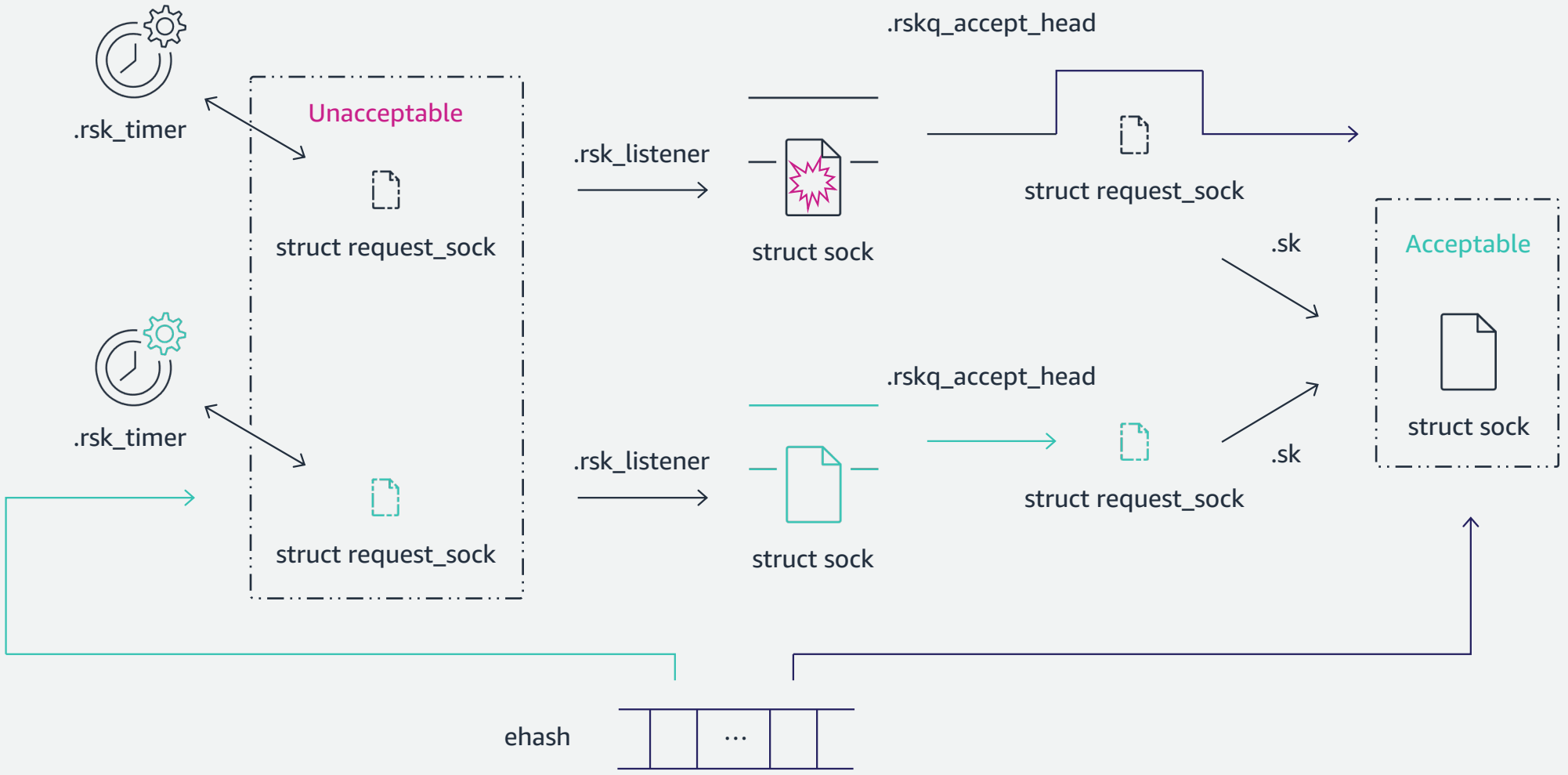
Migrate request_sock



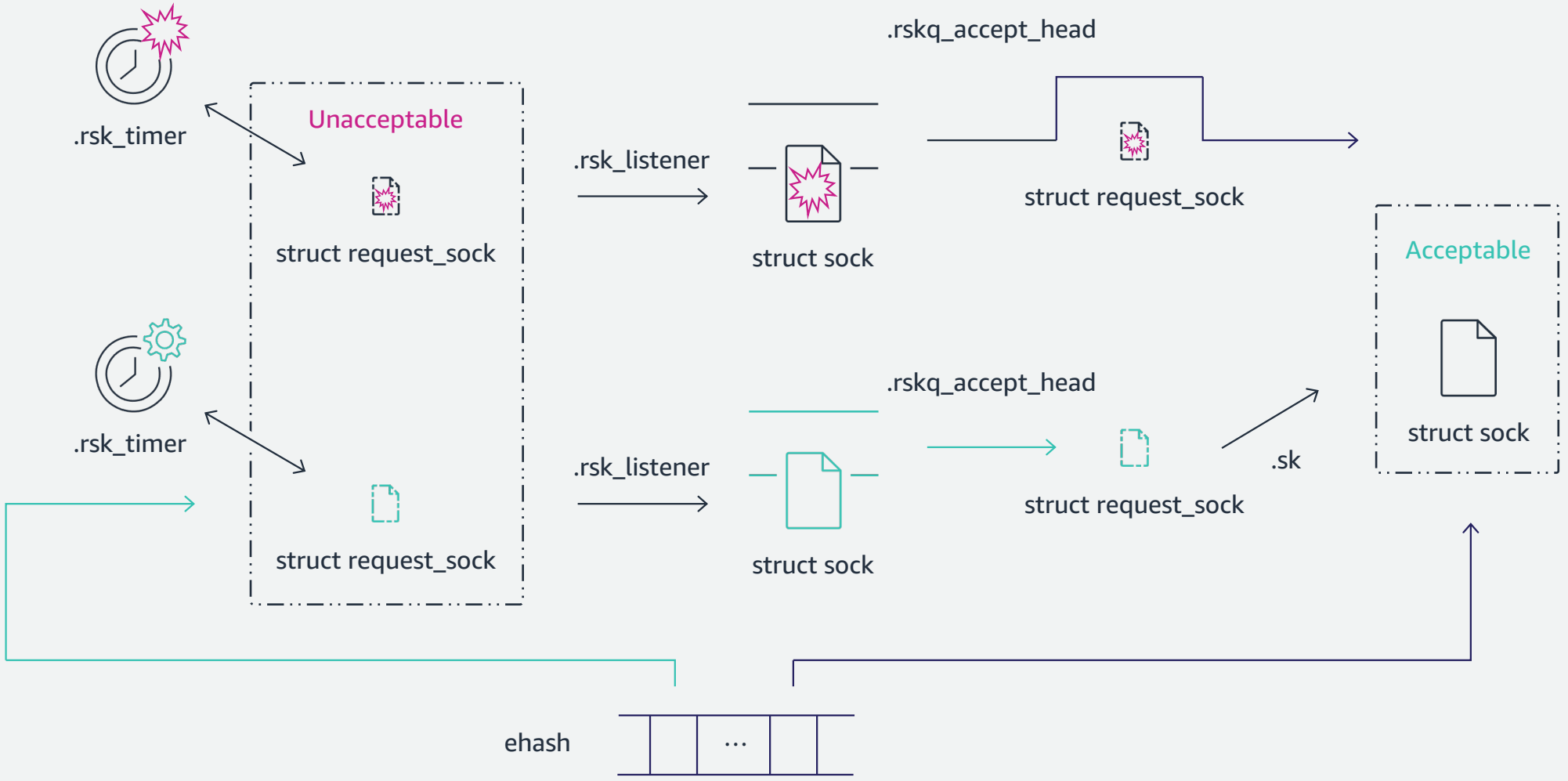
Migrate request_sock



Migrate request_sock



Migrate request_sock



Quiz with Socket migration

```
from socket import *

def get_reuseport_server():
        s = socket(AF_INET, SOCK_STREAM, 0)
        s.setsockopt(SOL_SOCKET, SO_REUSEPORT, 1)
        s.bind(("localhost", 80))
        s.listen(32)
        return s

def get_client():
        c = socket(AF_INET, SOCK_STREAM, 0)
        c.connect(("localhost", 80))
        return c

def quiz1():
        server_1 = get_reuseport_server()

        client = get_client()
        client.send(b'Hello World')

        server_2 = get_reuseport_server()

        server_1.close()

        server_2.setblocking(0)
        child, _ = server_2.accept()
        print(child.recv(1024))

if __name__ == '__main__':
        quiz1()
```

Optimal

```
$ sudo sysctl -w net.ipv4.tcp_migrate_req=1
net.ipv4.tcp_migrate_req = 1

$ sudo python3 quiz1.py
b'Hello World’
```

```
import subprocess, time
from quiz1 import *

def drop_ack(flag=True):
    subprocess.run('iptables -{} INPUT -d 127.0.0.1 -p tcp --dport 80 --tcp-flags SYN,ACK ACK -j DROP'
                   .format('A' if flag else 'D').split(' '))

def quiz2():
    server_1 = get_reuseport_server()

    drop_ack(True)

    client = get_client()
    client.send(b'Hello World')

    server_2 = get_reuseport_server()

    server_1.close()

    drop_ack(False)
    time.sleep(1)

    server_2.setblocking(0)
    child, _ = server_2.accept()
    print(child.recv(1024))

if __name__ == '__main__':
    quiz2()
```

Optimal

```
$ sudo sysctl -w net.ipv4.tcp_migrate_req=1
net.ipv4.tcp_migrate_req = 1

$ sudo python3 quiz2.py
b'Hello World'
```

Why disabled by default?

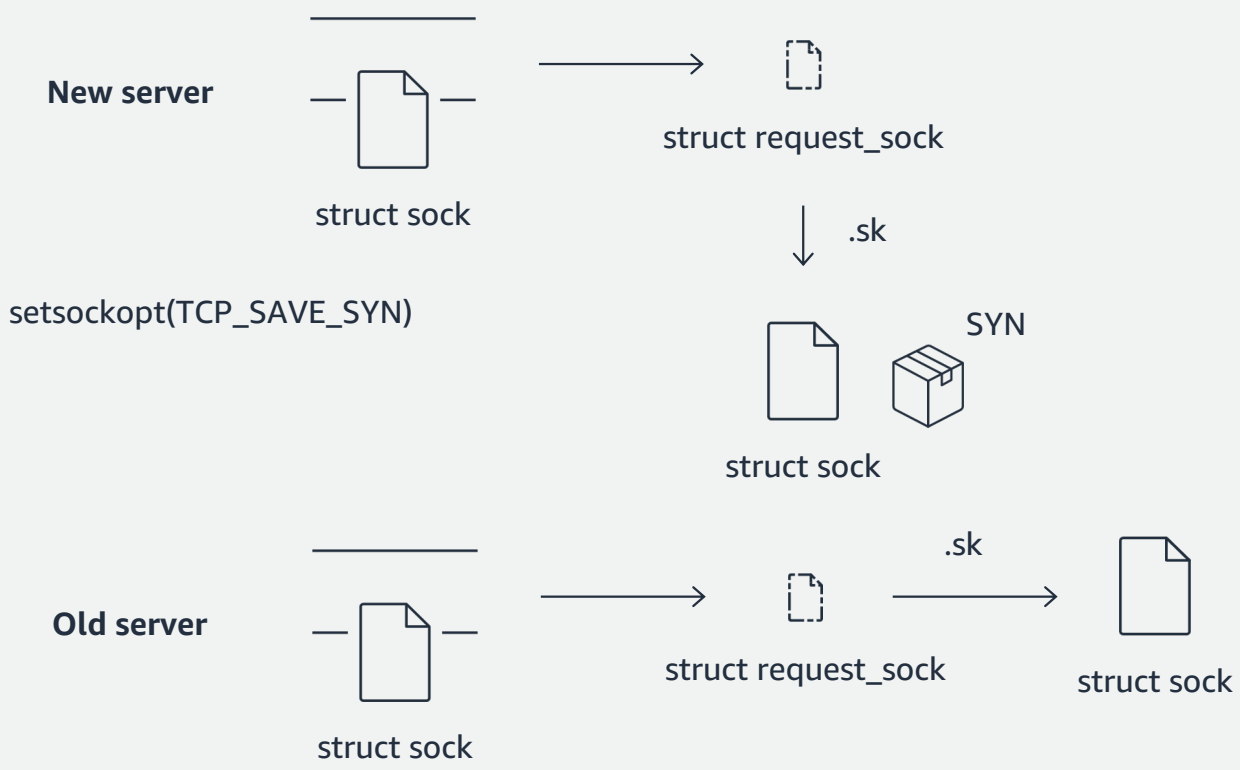
```
$ sudo sysctl net.ipv4.tcp_migrate_req
net.ipv4.tcp_migrate_req = 0

$ sudo sysctl -w net.ipv4.tcp_migrate_req=1
net.ipv4.tcp_migrate_req = 1

$ sudo python3 quiz1.py
b'Hello World’

$ sudo python3 quiz2.py
b'Hello World'
```

Why disabled by default?

- Different sockets may listen() on the same port

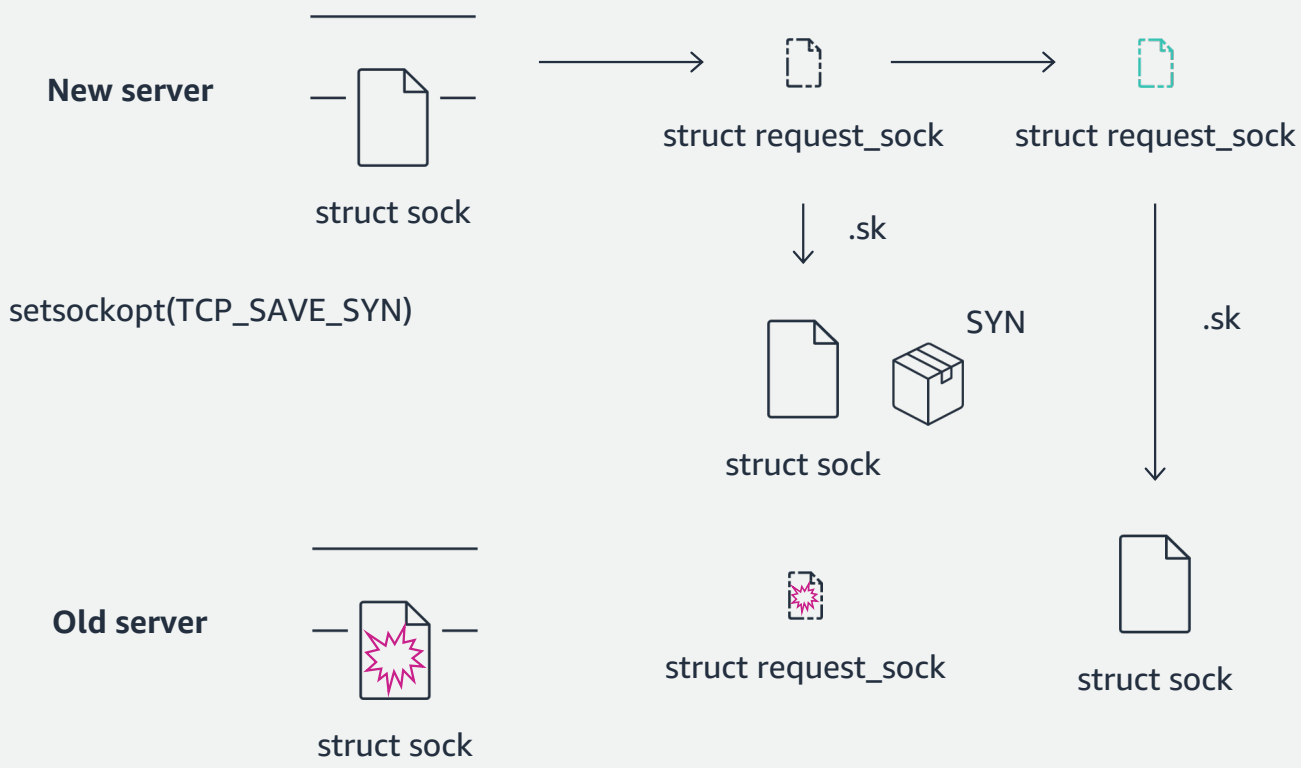
Why disabled by default?

- Different sockets may listen() on the same port
  - e.g. TCP_SAVE_SYN



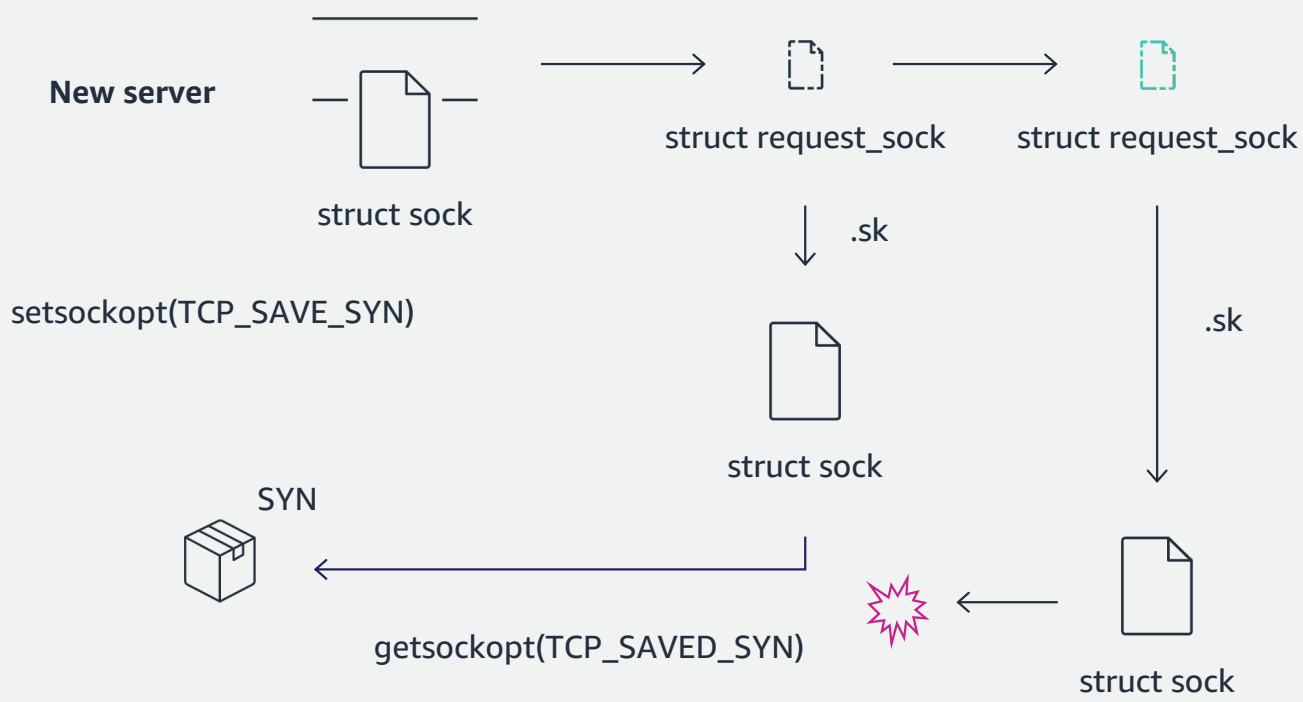
Why disabled by default?

- Different sockets may listen() on the same port
  - e.g. TCP_SAVE_SYN



Why disabled by default?

- Different sockets may listen() on the same port
  - e.g. TCP_SAVE_SYN



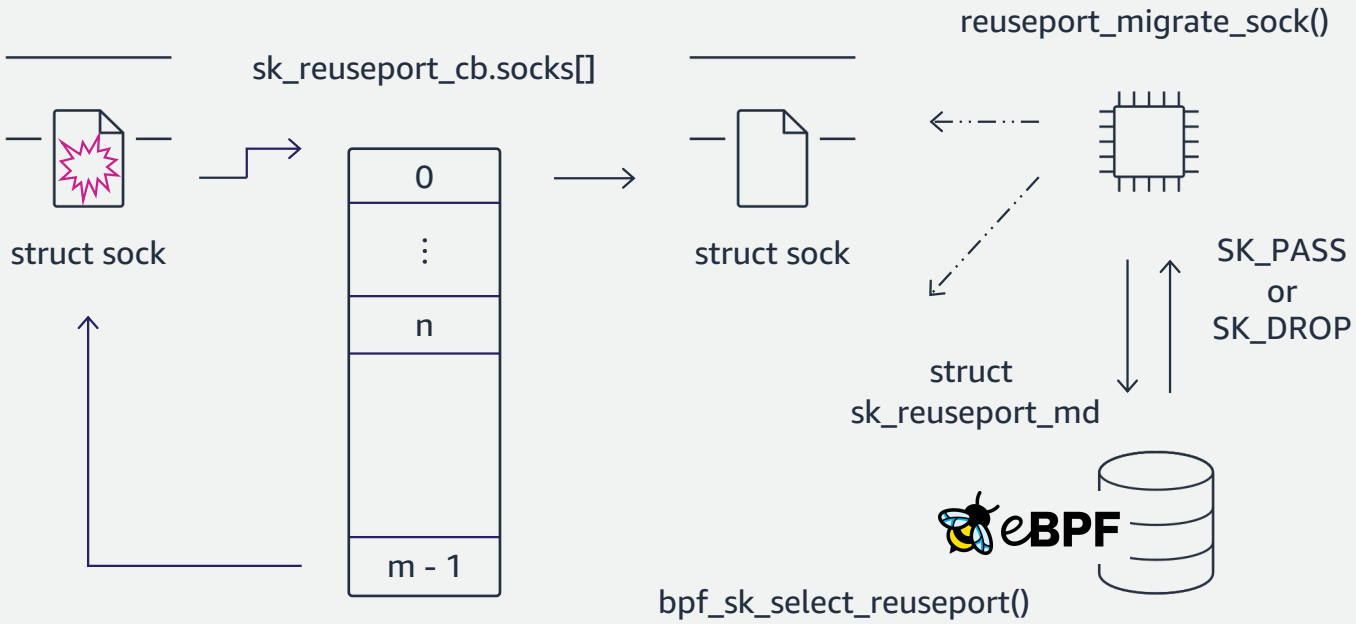
Why disabled by default?

- Different sockets may listen() on the same port
  - e.g. TCP_SAVE_SYN
- BPF may define routing policy for SYN

BPF_SK_REUSEPORT_SELECT_OR_MIGRATE

- New attach type for BPF_PROG_TYPE_SK_REUSEPORT
  - SEC("sk_reuseport/migrate")
- Executed when
  - Receiving a SYN packet

    and
  - Migrating a mini socket



BPF_SK_REUSEPORT_SELECT_OR_MIGRATE

- New attach type for BPF_PROG_TYPE_SK_REUSEPORT
  - SEC("sk_reuseport/migrate")
- Executed when
  - Receiving a SYN packet

    and
  - Migrating a mini socket

```
struct sk_reuseport_md {
        __bpf_md_ptr(void *, data);
        __bpf_md_ptr(void *, data_end);

        __u32 len;              /* Total length of packet
                                 * (starting from the tcp/udp header).
                                 */
        __u32 eth_protocol;
        __u32 ip_protocol;      /* IP protocol. e.g. IPPROTO_TCP, IPPROTO_UDP */
        __u32 bind_inany;       /* Is sock bound to an INANY address? */
        __u32 hash;             /* A hash of the packet 4 tuples */

        __bpf_md_ptr(struct bpf_sock *, sk);            /* closing listener */
        __bpf_md_ptr(struct bpf_sock *, migrating_sk);  /* mini/full socket */
};
```

Which to use, sysctl or BPF?

- BPF_SK_REUSEPORT_SELECT_OR_MIGRATE
  - Listeners have different settings at the setsockopt() level
  - BPF program is attached to route SYN packets based on some rules
- net.ipv4.tcp_migrate_req

Conclusion

- Closing a listener aborts two kinds of connections
  - Acceptable : ESTABLISHED, SYN_RECBV
  - Unacceptable : NEW_SYN_RECV
- Socket migration feature is available from 5.14 (1f26622b791b)
  - Prevent connection failures when other listeners exist on the same port
  - Make hot-reloading easier and faster

Thank you!

Q&A