UDP and Network Address Translation

Networked Systems 3
Lecture 14

Lecture Outline

- The UDP protocol and datagram sockets
- Impact of Network Address Translation

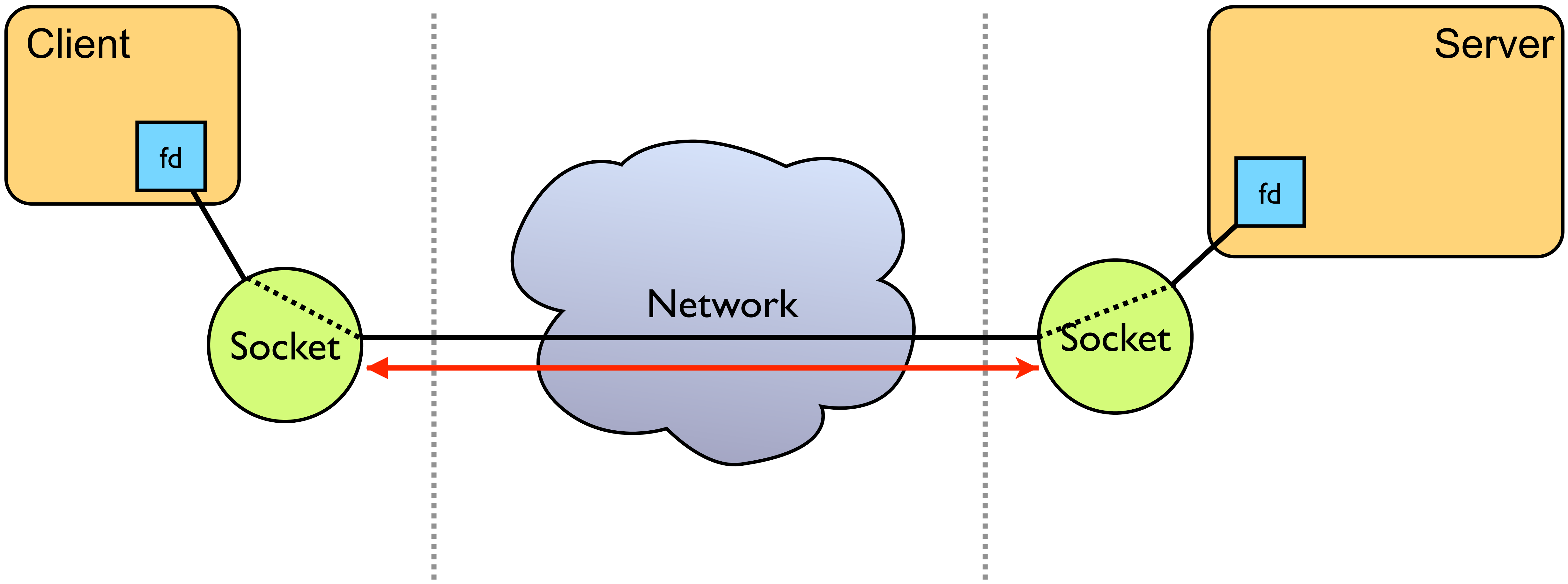
Using UDP Datagrams

- UDP provides an unreliable datagram service, identifying applications via a 16 bit port number
  - UDP ports are separate from TCP ports
  - Often used peer-to-peer (e.g., for VoIP), so both peers must `bind()` to a known port
  - Create via `socket()` as usual, but specify `SOCK_DGRAM` as the socket type:

```
int    fd;
...
fd = socket(AF_INET, SOCK_DGRAM, 0);
```

  - No need to `connect()` or `accept()`, since no connections in UDP

Using UDP Datagrams




```
int fd = socket(...)

bind(fd, ..., ...)

sendto(fd, data, datalen, addr, addrlen)

recvfrom(fd, buffer, buflen, flags, addr, addrlen)

close(fd)
```

Sending UDP Datagrams

The `sendto()` call sends a single datagram. Each call to `sendto()` can send to a different address, even though they use the same socket.

```
int                   fd;
char                  buffer[...];
int                   buflen = sizeof(buffer);
struct sockaddr_in    addr;
...
if (sendto(fd, buffer, buflen, (struct sockaddr *) addr, sizeof(addr)) < 0) {
    // Error...
}
```

Alternatively, `connect()` to an address, then use `write()` to send the data. There is no connection made at the UDP layer, the `connect()` call only sets the destination address for future packets.

Receiving UDP Datagrams

The `read()` call may be used to read a single datagram, but doesn't provide the source address of the datagram. Most code uses `recvfrom()` instead – this fills in the source address of the received datagram:

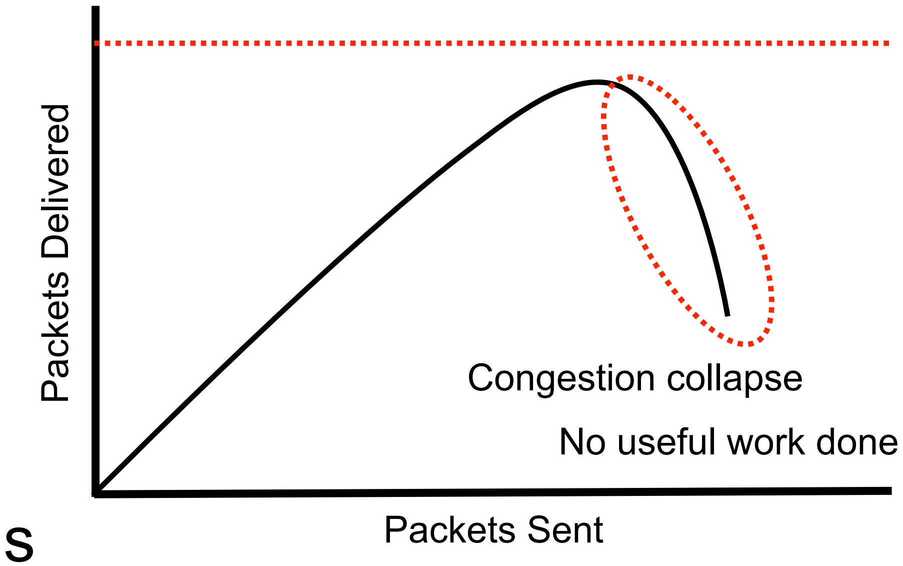
```
int                  fd;
char                 buffer[...];
int                  buflen = sizeof(buffer);
struct sockaddr      addr;
socklen_t            addr_len = sizeof(addr);
int                  rlen;
...
rlen = recvfrom(fd, buffer, buflen, 0, &addr, &addrlen);
if (rlen < 0) {
    // Error...
}
```

UDP Framing and Reliability

- Unlike TCP, each UDP datagram is sent as exactly one IP packet (which may be fragmented in IPv4)
  - Each `read()` corresponds to a single `write()`
- But, transmission is unreliable: packets may be lost, delayed, reordered, or duplicated in transit
  - The application is responsible for correcting the order, detecting duplicates, and repairing loss – if necessary
  - Generally requires the sender to include some form of sequence number in each packet sent

UDP Guidelines

- Need to implement congestion control in applications
  - To avoid congestion collapse of the network
  - Should be approximately fair to TCP
  - RFC 3448 provides one algorithm for doing this



- Need to provide sequencing, reliability, and timing in applications
  - Sequence numbers and acknowledgements
  - Retransmission and/or forward error correction
  - Timing recovery
- UDP programming guidelines: RFC 5405
  - https://tools.ietf.org/html/rfc5405

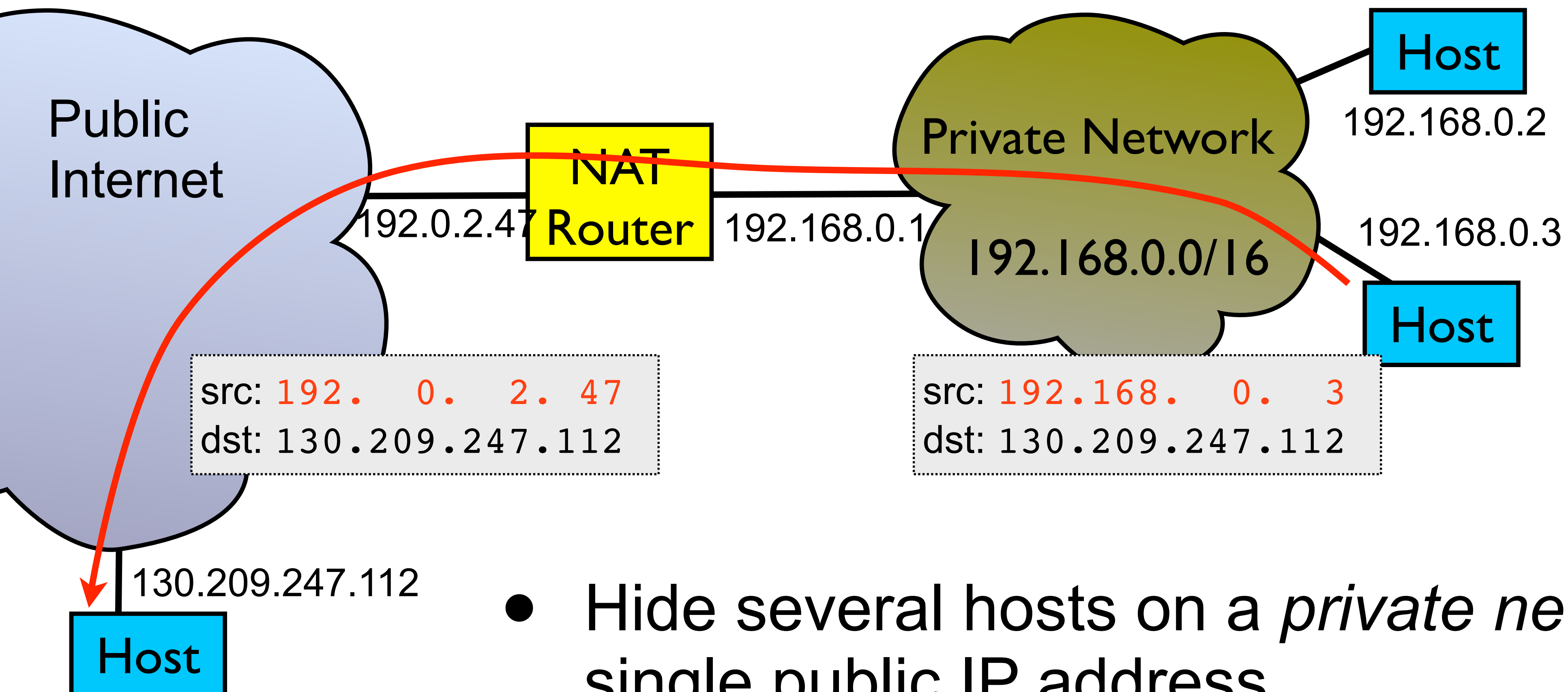
Network Address Translation

- IPv4 address space is exhausted → lecture 9
- IPv6 is the long-term solution

- There is a widely deployed work-around: NAT (network address translation)

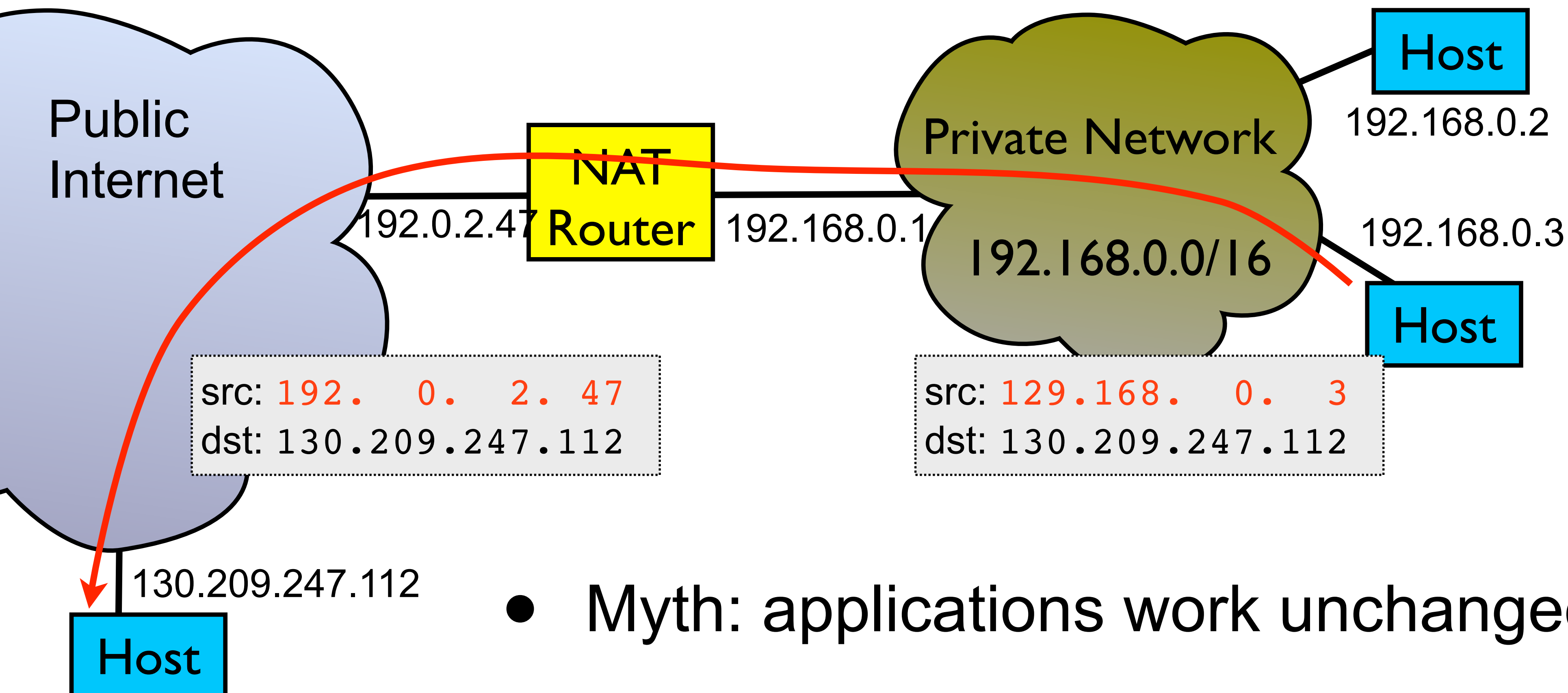
- However, this has serious consequences for the transport layer

Network Address Translation



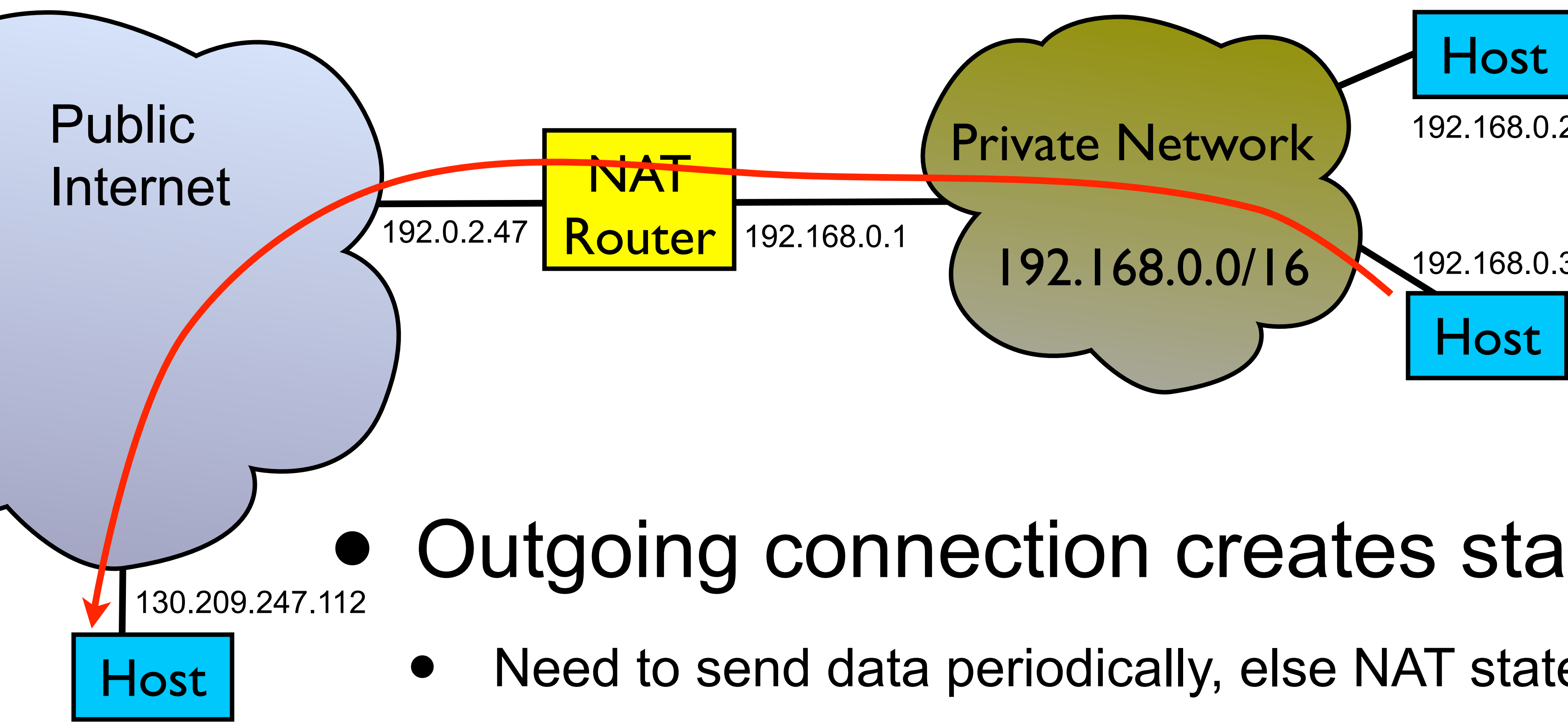
- Hide several hosts on a *private network* behind a single public IP address
  - Private IPv4 addresses are 10.0.0.0/8, 192.168.0.0/16, 176.16.0.0/12
- Rewrite packet headers at network boundary
  - Doesn't require changes to hosts or routers (other than the NAT)
- Tries to give the illusion of more address space

Network Address Translation



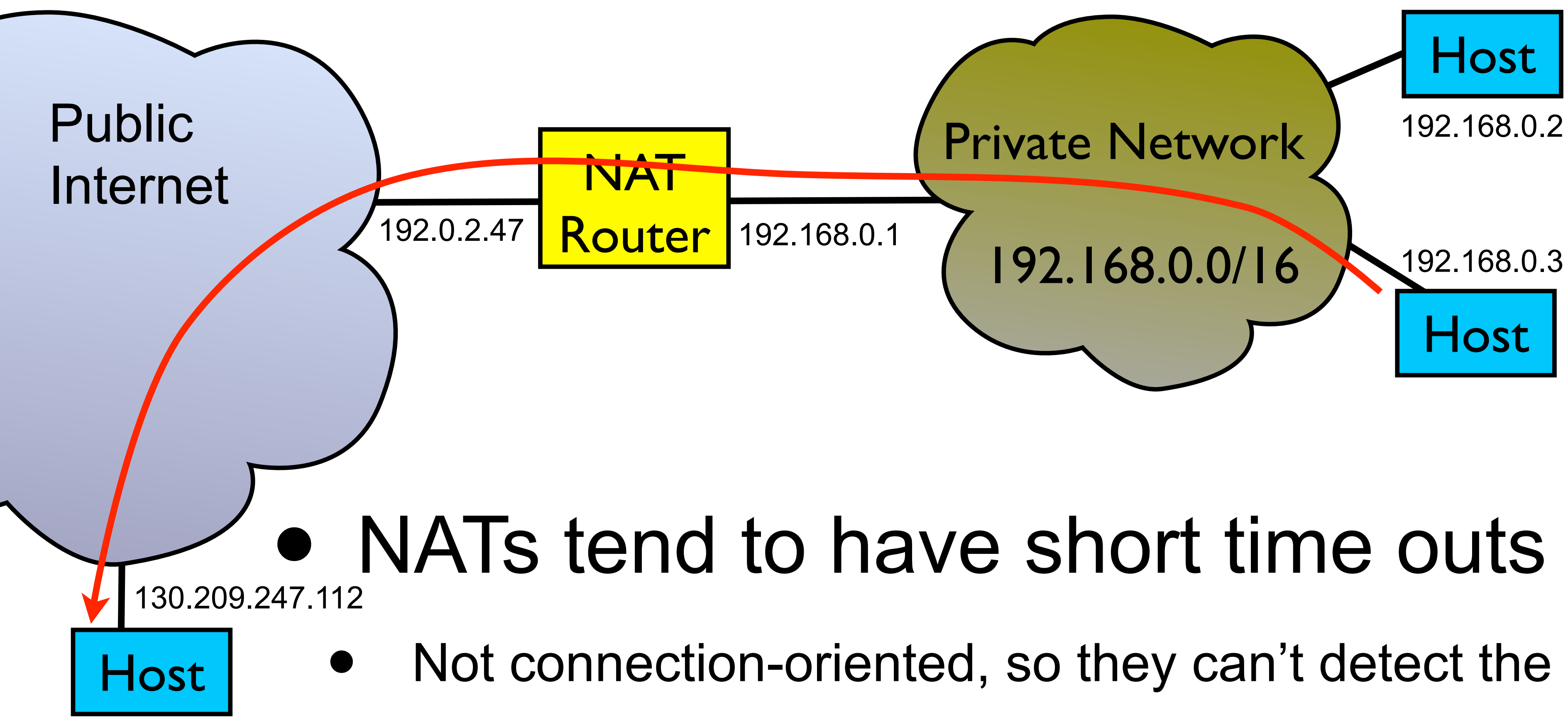
- Myth: applications work unchanged
  - *Some* client-server applications (e.g., web, email) work without changes
  - But peer-to-peer applications (e.g., VoIP, WebRTC) need extensive changes before they work through a NAT (~200 pages spec to describe algorithm!)
- Myth: provides security
  - Most NATs include a firewall to provide security, the NAT function gives no security benefit

Implications of NAT for TCP Connections



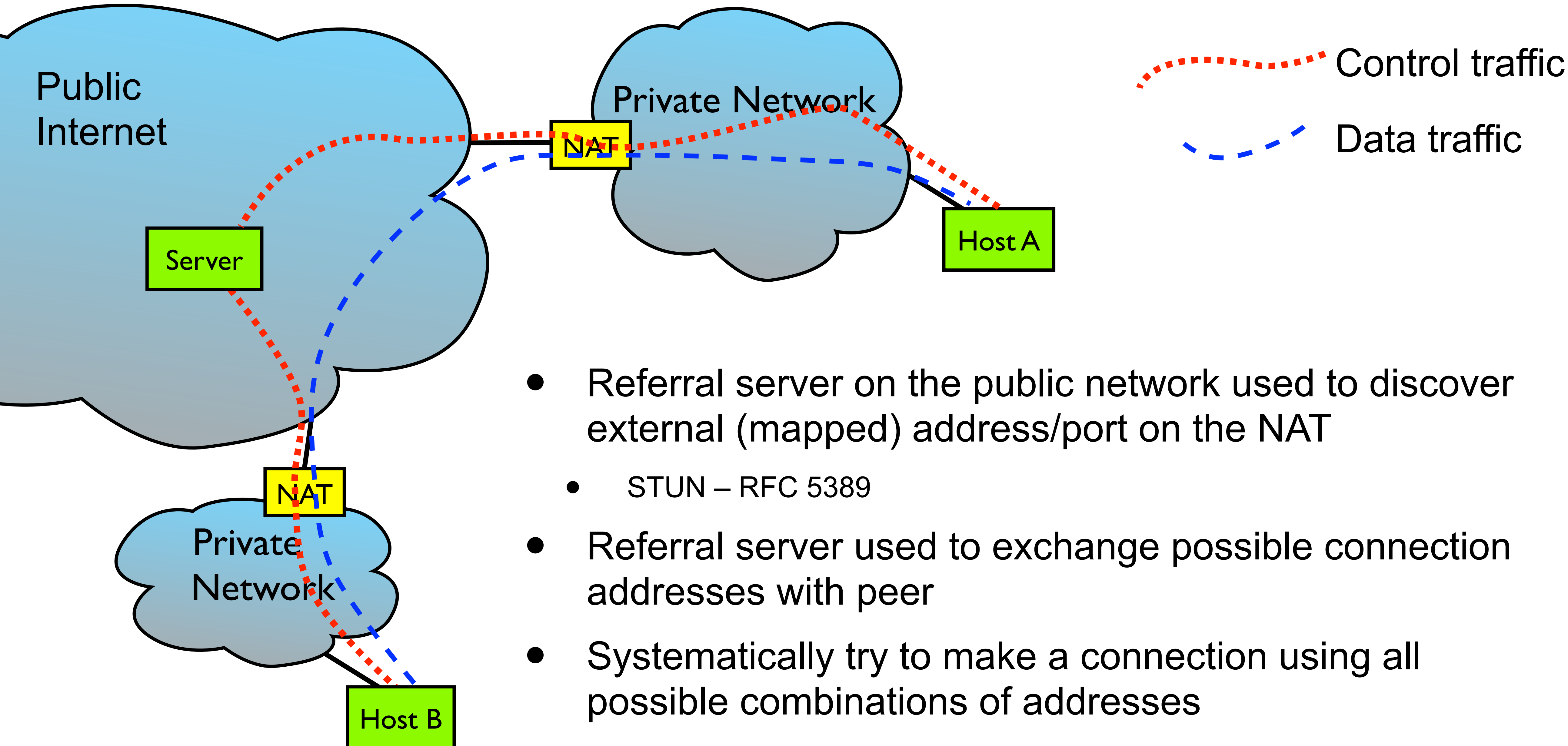
- Outgoing connection creates state in NAT
  - Need to send data periodically, else NAT state times out
  - Recommended time out interval is 2 hours, many NATs use shorter RFC5382
- Server behind NAT requires configured mapping
- Peer-to-peer connections difficult
  - Simultaneous open with external mapping service

Implications of NAT for UDP Flows



- NATs tend to have short time outs for UDP
  - Not connection-oriented, so they can't detect the end of flows
  - Recommended time out interval is not less than two minutes, but many NATs use shorter intervals – the VoIP NAT traversal standards suggest sending a keep alive message every 15 seconds RFC4787
- Peer-to-peer connections easier than TCP
  - UDP NATs are often more permissive about allowing incoming packets than TCP NATs; many allow replies from anywhere to an open port

NAT Traversal Concepts



- Referral server on the public network used to discover external (mapped) address/port on the NAT
  - STUN – RFC 5389
- Referral server used to exchange possible connection addresses with peer
- Systematically try to make a connection using all possible combinations of addresses
  - Every possible network interface and protocol, mapped and local
  - Complex and generates significant traffic overhead
  - The ICE algorithm – RFC 5245

Summary

- UDP and datagram sockets
- Network address translation
  - Impact on TCP connections
  - Impact on UDP traffic
  - NAT traversal concepts