

Using the Transport Layer

Networked Systems 3
Lecture 12

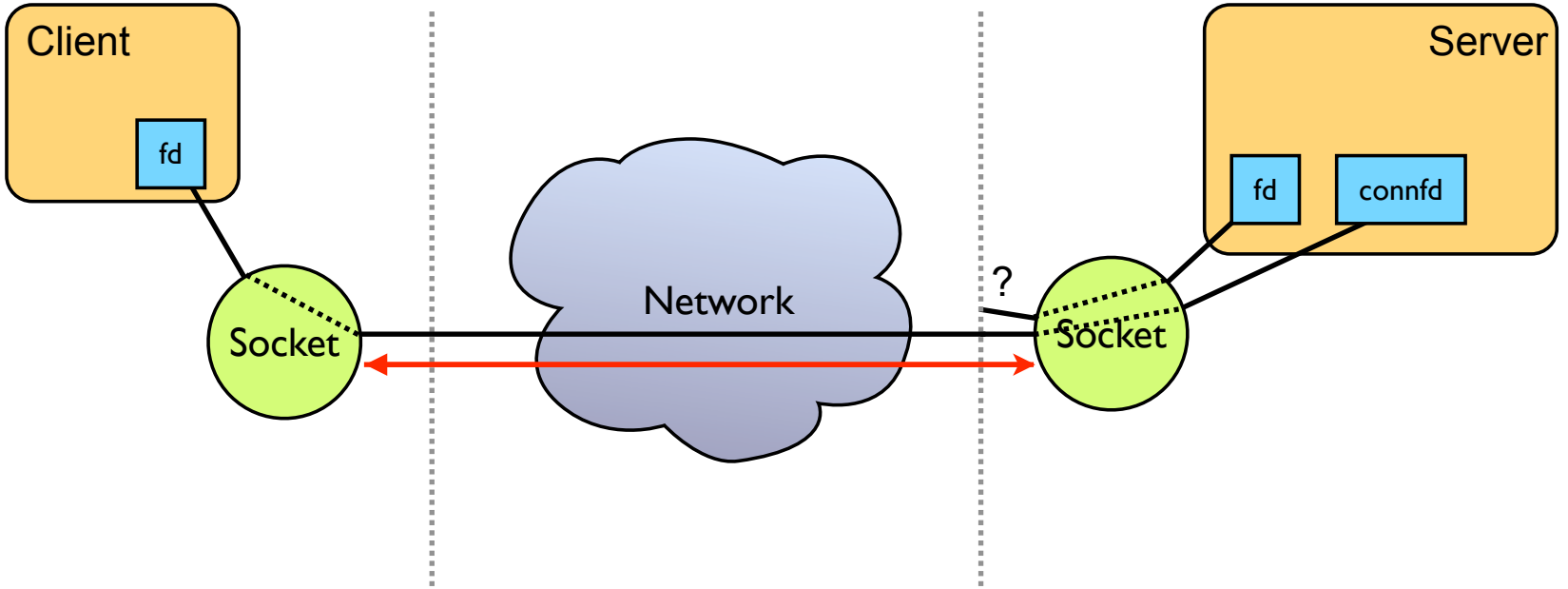
Lecture Outline

- Using TCP connections
- Using UDP datagrams
- NAT traversal concepts

Using TCP Connections

- A TCP connection provides a reliable byte stream abstraction, identifying applications by a 16 bit port number
  - Role of the port number
  - Implications of the reliable byte stream abstraction
    - Record boundaries
    - Head of line blocking
  - Implications of NAT on TCP connections

Using TCP Connections




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

connect(fd, ..., ...)
write(fd, data, datalen)
read(fd, buffer, buflen)
close(fd)
```

source and destination ports specified

```
int fd = socket(...)
bind(fd, ..., ...)
listen(fd, ...)
connfd = accept(fd, ...)
read(connfd, buffer, buflen)
write(connfd, data, datalen)
close(connfd)
```

Role of the TCP Port Number

Port Range		Name	Intended use
0	1023	Well-known (system) ports	Trusted operating system services
1024	49151	Registered (user) ports	User applications and services
49152	65535	Dynamic (ephemeral) ports	Private use, peer-to-peer applications, source ports for TCP client connections

draft-ietf-tsvwg-iana-ports-01.txt

- Servers must listen on a known port; IANA maintains a registry
- Distinction between system and user ports ill-advised – security problems resulted
- Insufficient port space available (>75% of ports are registered)

- TCP clients traditionally connect from a randomly chosen port in the ephemeral range
  - The port must be chosen randomly, to prevent spoofing attacks
  - Many systems use the entire port range for source ports, to increase the amount of randomness available

http://www.iana.org/assignments/port-numbers

TCP Congestion Control

- A TCP connection reliably delivers a byte stream
  - The transmission speed depends on network conditions (→ lecture 14)
  - The `write()` call will block if there is insufficient network or buffer capacity
    - Can use the `select()` function to determine if a call will block, but cannot determine how long a particular `write()` will take

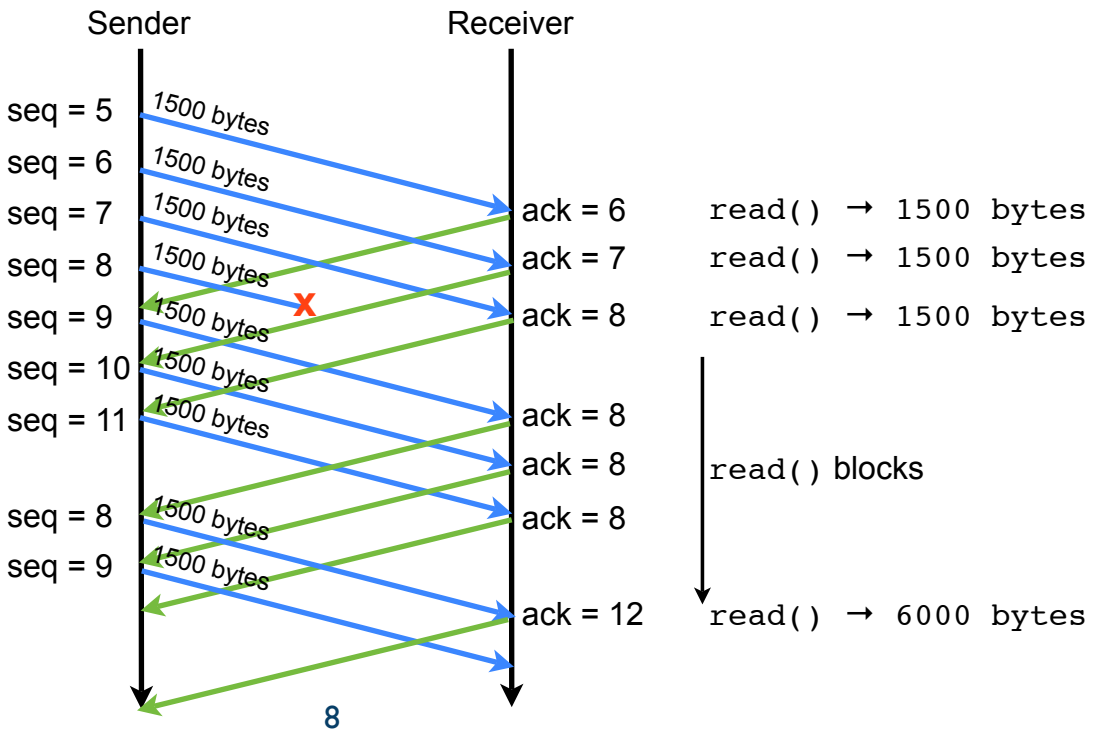
```
int    fd;
fd_set wfds;
...
FD_ZERO(&wfds);
FD_SET(fd, & wfds);
if (select(fd+1, NULL, &wfds, NULL, NULL) > 0) {
    // Space is available to write()
}
```

Record Boundaries in TCP Connections

- If the data in a `write()` is bigger than the data link layer MTU, TCP will send the data as fragments
- Similarly, multiple small `write()` requests may be aggregated into a single TCP packet
- Implication: the data returned by a `read()` doesn't necessarily match that sent in a single `write()`
  - There often appears to be a correspondence, but this *is not* guaranteed (it may work in the lab, but not when you use it over a different link)

Head of Line Blocking in TCP

- What if data is lost due to network congestion?
  - TCP will retransmit the missing data, transparently to the application (→ lecture 13)
  - A `read()` for the missing data will block until it arrives; TCP delivers all data in-order



Application Level Framing

Data may arrive in arbitrary sized chunks; must parse and understand the data, no matter where it is split by the network – it's a byte stream (colours indicate one possible split of the data into chunks)

```
HTTP/1.1 200 OK
Date: Mon, 19 Jan 2009 22:25:40 GMT
Server: Apache/2.0.46 (Scientific Linux)
Last-Modified: Mon, 17 Nov 2003 08:06:50 GMT
ETag: "57c0cd-e3e-17901a80"
Accept-Ranges: bytes
Content-Length: 3646
Connection: close
Content-Type: text/html; charset=UTF-8

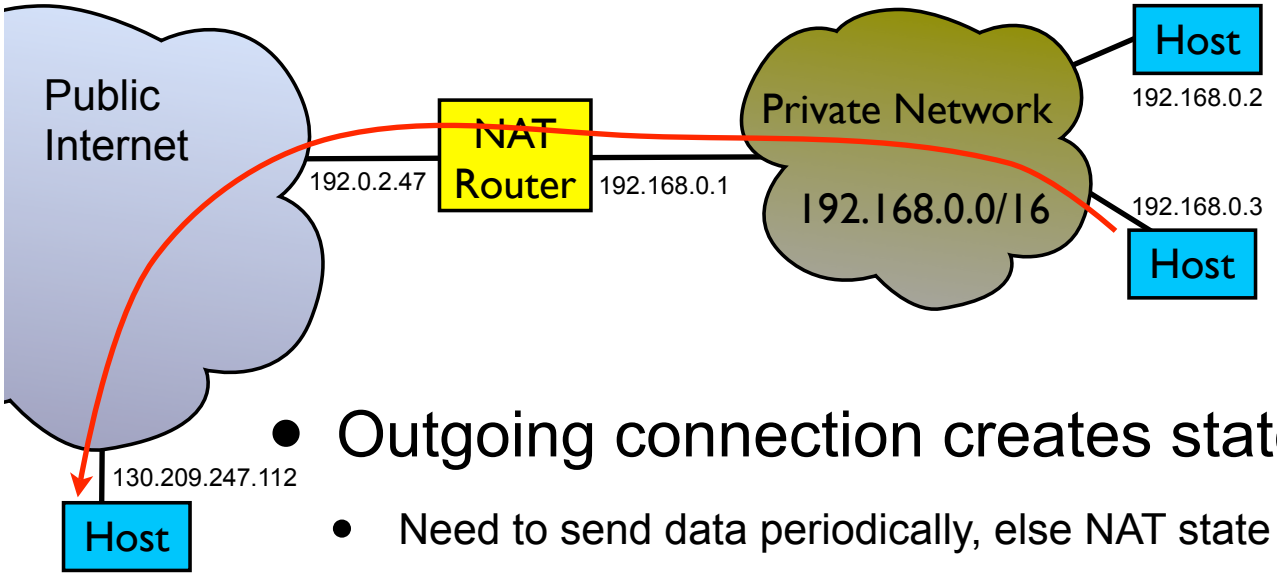
<HTML>
<HEAD>
<TITLE>Computing Science, University of Glasgow </TITLE>
...
</BODY>
</HTML>
```

Example: HTTP response

Known marker (blank line) signals end of headers

Size of payload indicated in the headers

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

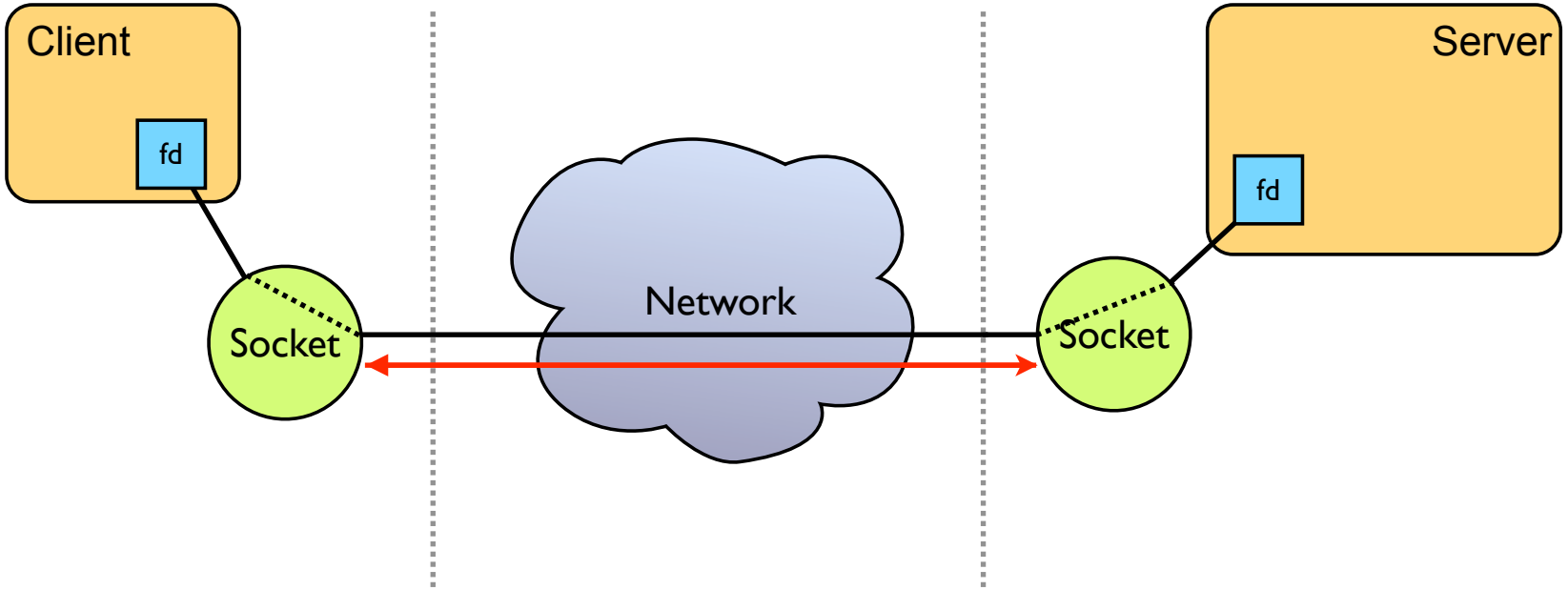
Using UDP Datagrams

- 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:

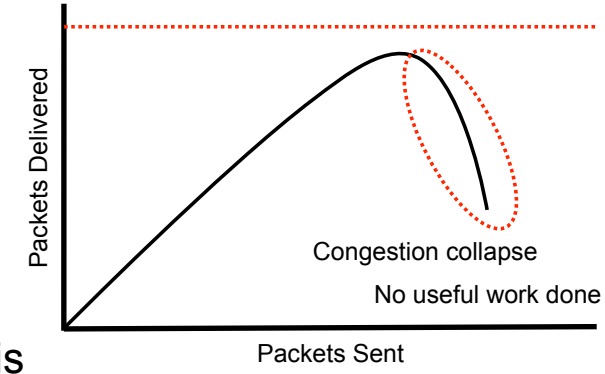
```
int                fd;
char               buffer[...];
int                buflen = sizeof(buffer);
struct sockaddr    addr;
socklen_t          addr_len = sizeof(addr);
int                rlen;
...
rlen = recvfrom(fd, buffer, buflen, 0, &addr, &addrlen);
if (rlen < 0) {
    // Error...
}
```

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

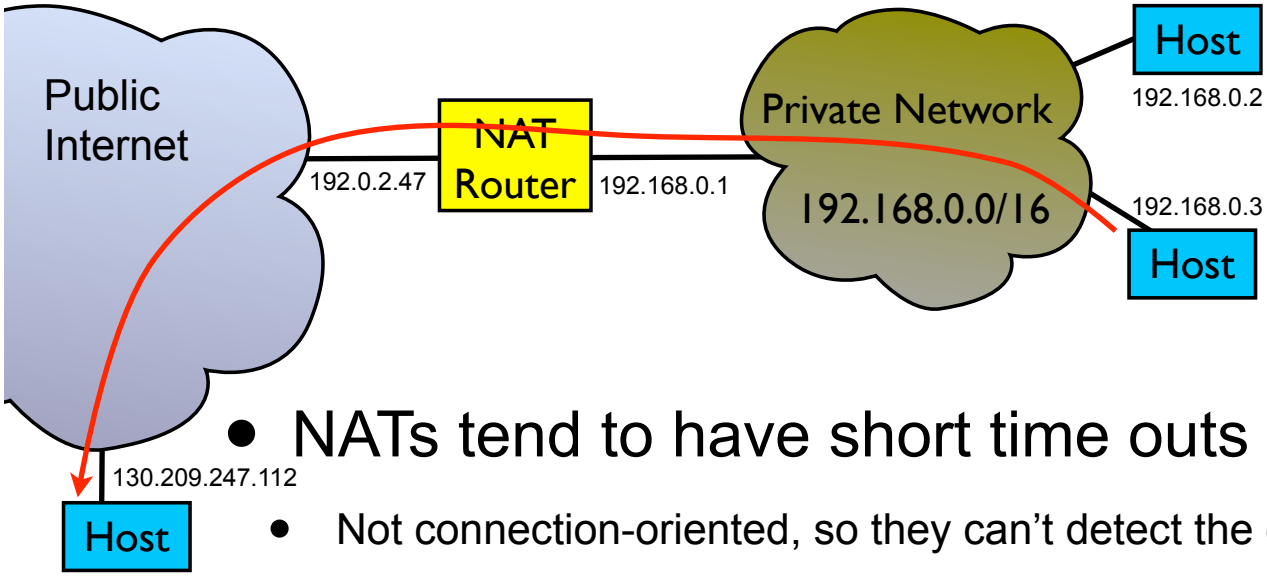
UDP Guidelines

- 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



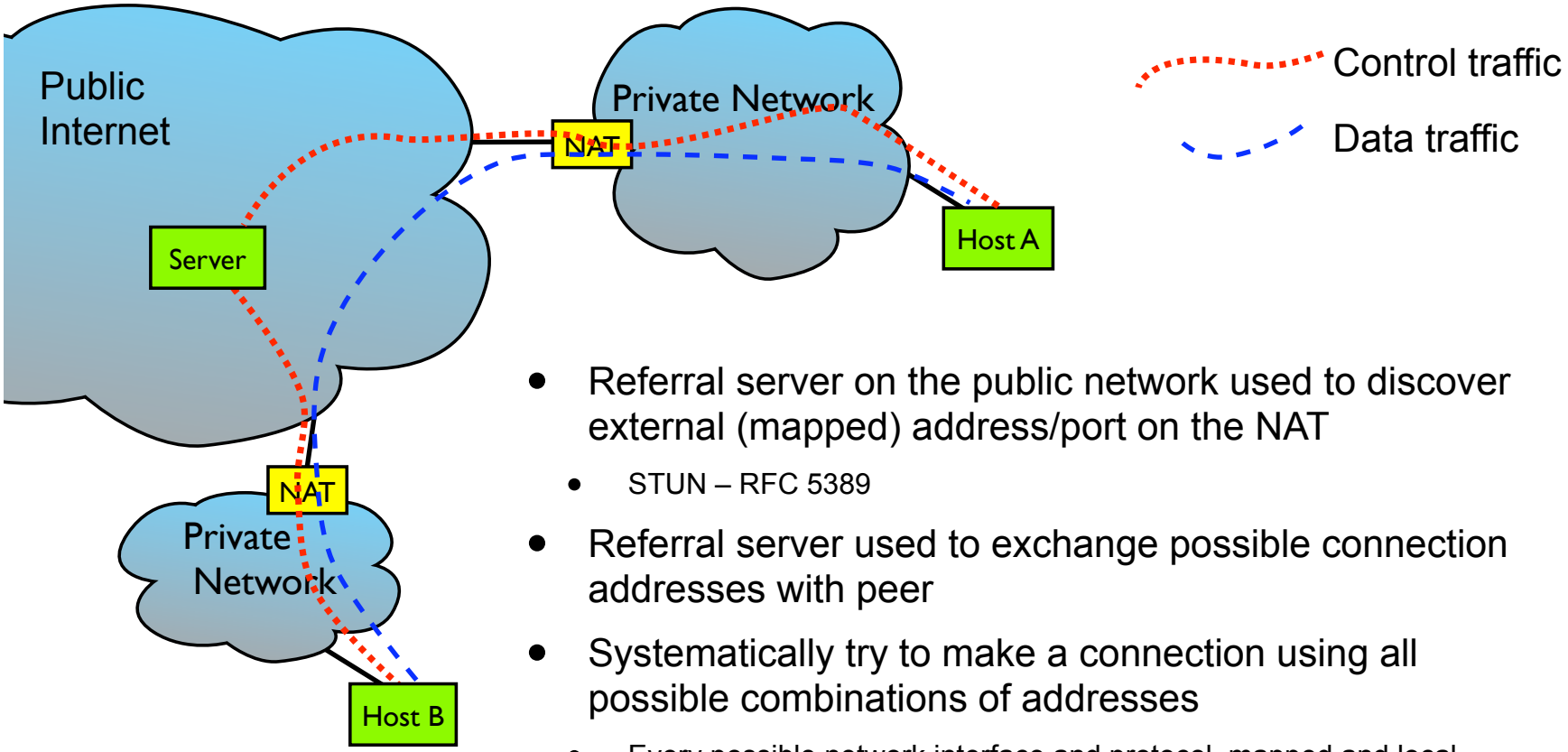
- Need to provide sequencing, reliability, and timing in applications
  - Sequence numbers and acknowledgements
  - Retransmission and/or forward error correction
  - Timing recovery
- UDP programming guidelines: RFC 5405

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
  - ICE – http://tools.ietf.org/html/draft-ietf-mmusic-ice-19

Questions?