Socket Programming

socket: a data structure containing connection information

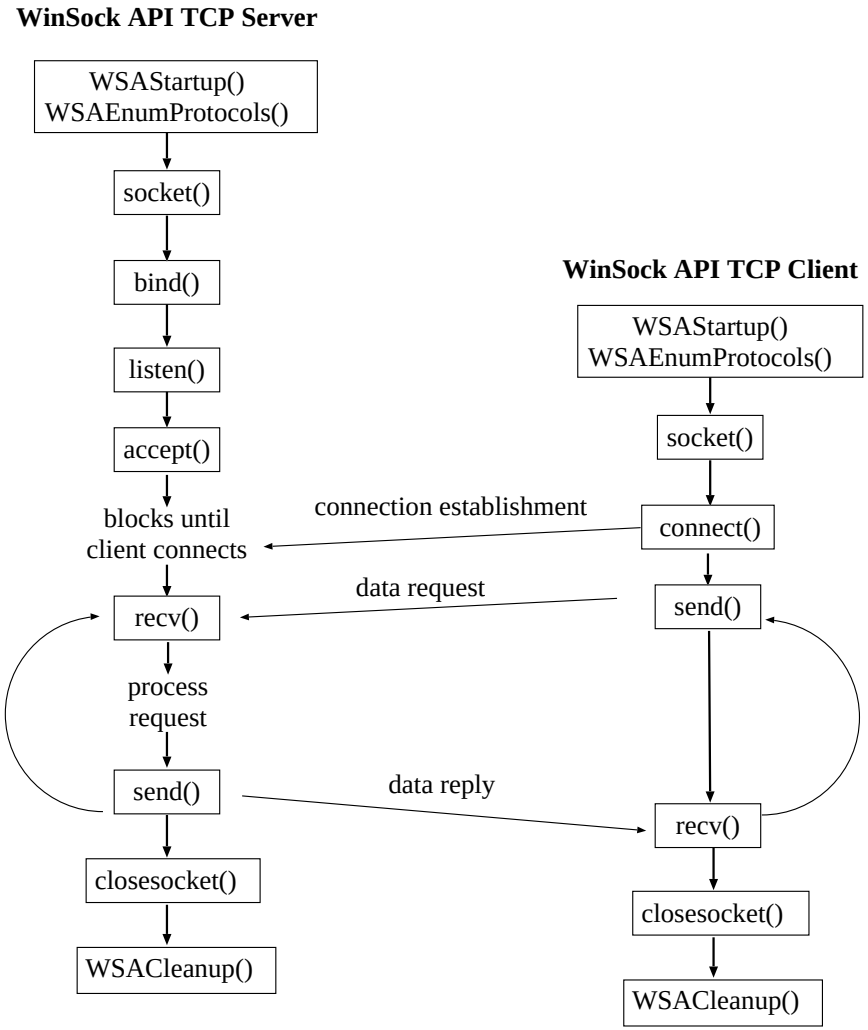
Connection identifying information:

- client IP (Internet Protocol) address
- client port number
- source IP address
- source port number

Client-server connection:

- server creates a socket and listens for connections on a well-known port number
- client creates a socket and connects to the server address at the well-known port number

TCP Connection



Sugih Jamin (jamin@eecs.umich.edu)

server.c

```
int visits;
int
main(int argc, char *argv[])
{
  struct sockaddr_in self, client;
  struct hostent *cp;
  int sd, td, len;
  char buf[BLEN];

  sd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
  memset((char *) &self, 0, sizeof(struct sockaddr_in));
  self.sin_family = AF_INET;
  self.sin_addr.s_addr = INADDR_ANY;
  self.sin_port = htons((u_short) PORT);
  bind(sd, (struct sockaddr *) &self, sizeof(struct sockaddr_in));
  listen(sd, QLEN);

  while (1) {
    len = sizeof(struct sockaddr_in);
    td = accept(sd, (struct sockaddr *) &client, &len);
    cp = gethostbyaddr((char *) &client.sin_addr, sizeof(struct in_addr), AF_INET);
    printf("Connected from %s\n", cp->h_name);
    visits++;
    sprintf(buf, "This server has been contacted %d time(s).\n", visits);
    send(td, buf, strlen(buf), 0);
    close(td);
  }
}
```

client.c

```
int
main(int argc, char *argv[])
{
  struct sockaddr_in server;
  struct hostent *sp;
  int sd;
  int n;
  char buf[BLEN];

  sd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

  memset((char *) &server, 0, sizeof(struct sockaddr_in));
  server.sin_family = AF_INET;
  server.sin_port = htons((u_short) PORT);
  sp = gethostbyname(SERVER);
  memcpy(&server.sin_addr, sp->h_addr, sp->h_length);

  connect(sd, (struct sockaddr *) &server, sizeof(struct sockaddr_in));

  n = recv(sd, buf, sizeof(buf), 0);
  while (n > 0) {
    write(1, buf, n);
    n = recv(sd, buf, sizeof(buf), 0);
  }

  close(sd);
  exit(0);
}
```

includes and defines

To be prepended to both `server.c` and `client.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define SERVER "localhost"
#define PORT 4897
#define BLEN 256
#define QLEN 200
```

Socket APIs Highlights

WinSock APIs:

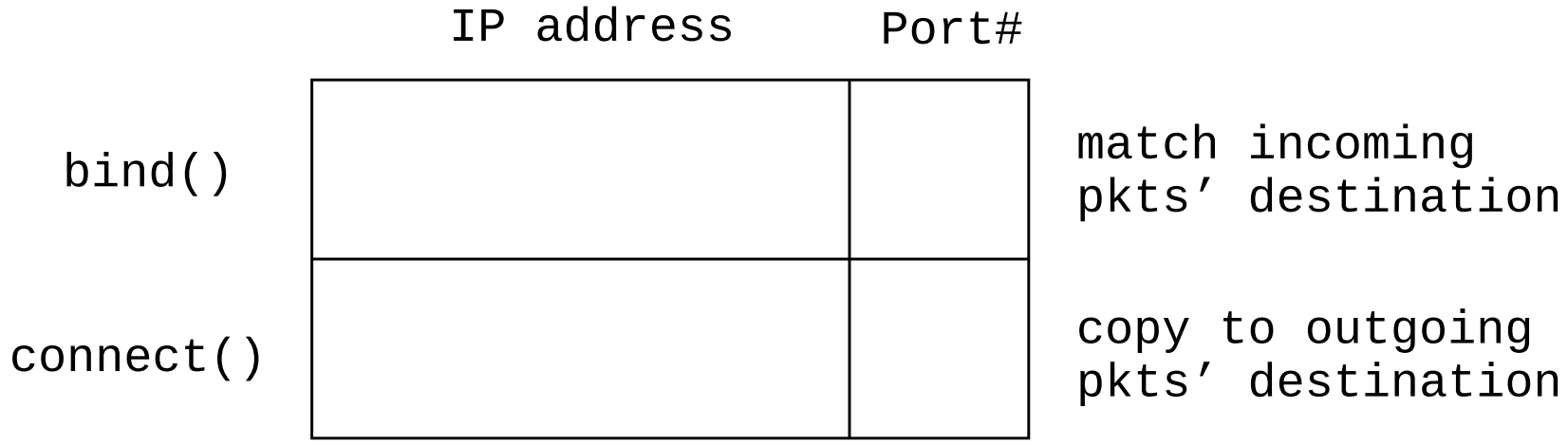
`socket`: creates a socket data structure

Then we need to populate the structure with the connection identifying information:

- client IP (Internet Protocol) address
- client port number
- source IP address
- source port number

TCP Socket Addresses

In the socket structure:



`bind`: used by server **only**, gives the server socket an IP address and/or port#

`connect`:

- TCP: initiates connection
- udp: remembers remote address

TCP Socket Addresses

TCP Server:

IP address	Port#
INADDR_ANY	well-known
client's address	ephem-eral

TCP Client:

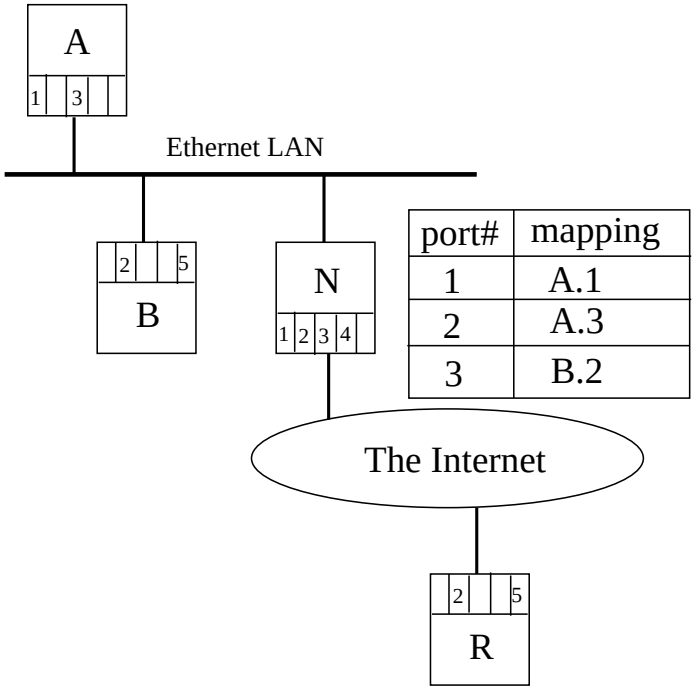
IP address	Port#
client's address	ephem-eral
server's address	well-known

NAT and Firewalls

What are NAT (Network Address Translation) and firewalls?

NAT and Firewalls

NAT boxes remap port numbers (Why?)



Firewalls may filter out all unknown ports and all UDP packets

Socket APIs Hightlights (cont)

`listen`:

- specifies max # of *pending* TCP connections
- only useful for connection oriented services
- TCP SYN denial of service attack

`accept`:

- waits for client connection
- returns a connected socket (different from the `listen`ing socket)

Socket APIs Hightlights (cont)

send:

- returns how many bytes are actually sent
- must loop to make sure that all is sent
  (except for blocking I/O, see UNP Section 6.2)

What is blocking and non-blocking I/O?

Why do you want to use non-blocking I/O?

Different Types of I/O

Synchronous: blocks (puts process to sleep) until I/O is ready
By default operations on sockets are blocking

Waiting for I/O:

1. wait for device availability
2. wait for I/O completion

Non-blocking I/O

Non-blocking I/O: keeps on checking (polling) until device is available

- set socket non-blocking:
  `int on = 1; ioctlsocket(socket, FIONBIO, &on);`
- call `select` on non-blocking socket

Signal-driven I/O: process gets a signal when device is available

- use `WSAAsyncSelect()` for signals tied to a window
- or `WSAEventSelect()` for signals not tied to a window

Asynchronous I/O: process notified when I/O completed

- Not widely supported yet

(See UNP Section 6.2 for more info)

Socket APIs Hightlights (cont)

`recv`:

- returns how many bytes are received
- 0 if connection is closed, -1 on error
- if non-blocking: -1 if no data with `errno` set to `EWOULDBLOCK`
- must loop to make sure that all is received (in TCP case)
- How do you know you have received everything sent?
  fixed size (part of protocol definition), prior handshake

Select

`select(maxfd, readset, writeset, acceptset, timeout)`

- synchronous (blocking) I/O multiplexing
- `maxfd` is the maximum file descriptor number + 1,
  so if you have only one descriptor, number 5, `maxfd` is 6.
- descriptor sets provided as bit mask. Use
  `FD_ZERO`, `FD_SET`, `FD_ISSET`, and `FD_CLR`
  to work with the descriptor sets
- (the fourth parameter is usually called the `exceptset`)

Select (cont)

`select(maxfd, readset, writeset, acceptset, timeout)`

- returns as soon as one of the specified socket is ready for I/O
- returns # of ready sockets, -1 on error, 0 if timed out and no device is ready (what for?)

`recv` with `select` vs. Polling

Which of the following would you use? Why?

```
loop {
  select(. . . , timeout);

  recv();
} till done;
```

or:

```
loop {
  sleep(seconds)

  recv();
} till done;
```

Socket APIs Hightlights (cont)

`closesocket`:

- marks socket unusable
- actual tear down depends on TCP
  if `bind()` fails, check `WSAGetLastError()` for `WSEADDRINUSE`

Socket Options: `getsockopt` and `setsockopt` (UNP Ch. 7)

- `SO_REUSEADDR`: allows server to restart or multiple servers to bind to the same port with different IP addresses
- `SO_LINGER`: whether `close` should return immediately or abort connection or wait for termination
- `SO_RCVBUF` and `SO_SNDBUF`: set buffers sizes
- `SO_KEEPALIVE`: server pings client periodically

UDP Socket Programming

Server must always call `bind()`, but not `listen()` nor `accept()`.

Client doesn't need to call `connect()`.

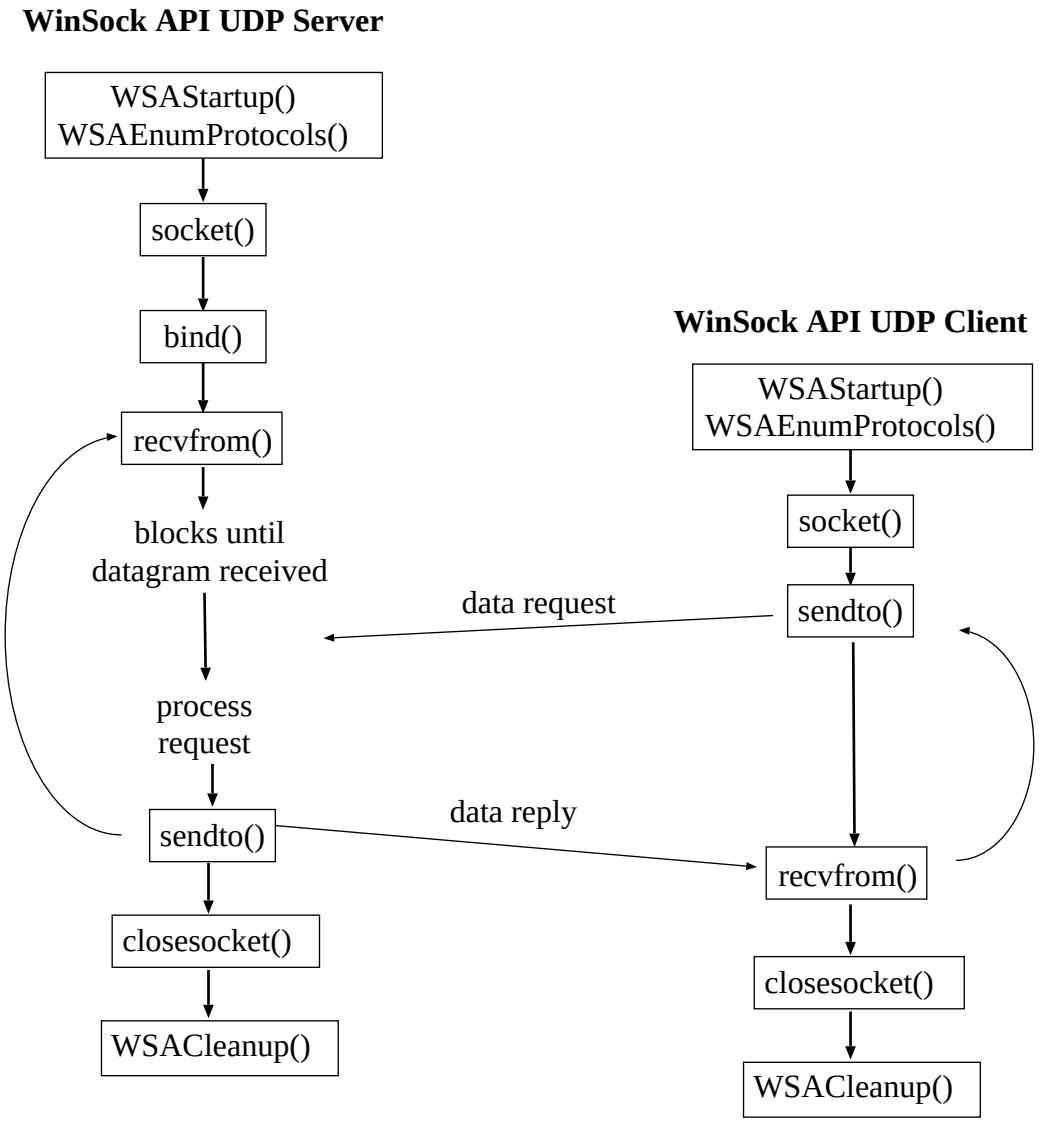
Use `sendto()` instead of `send()`.

However, `connect()` can still be used to tell the system to remember the remote address. Then `send()` instead of `sendto()` can be used.

Call either `recv()` or `recvfrom()` to recv. `recvfrom()` also returns the address of the client.

UDP packets have boundary, not a byte-stream as in TCP, so `recv()` retrieves one message at a time, i.e. no need to call `recv()` in a loop.

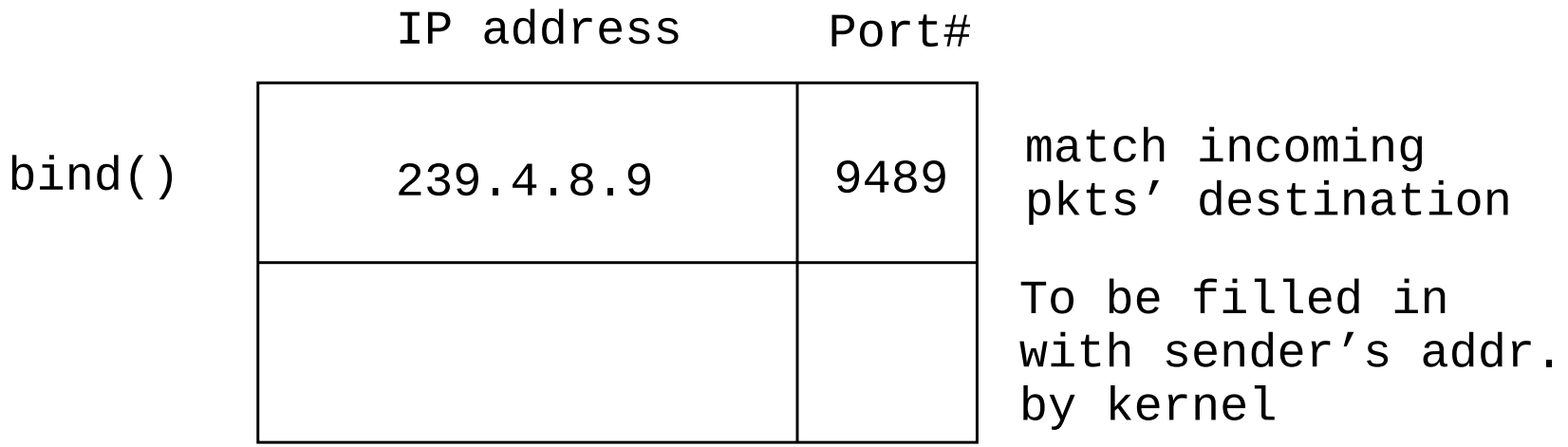
UDP Datagram



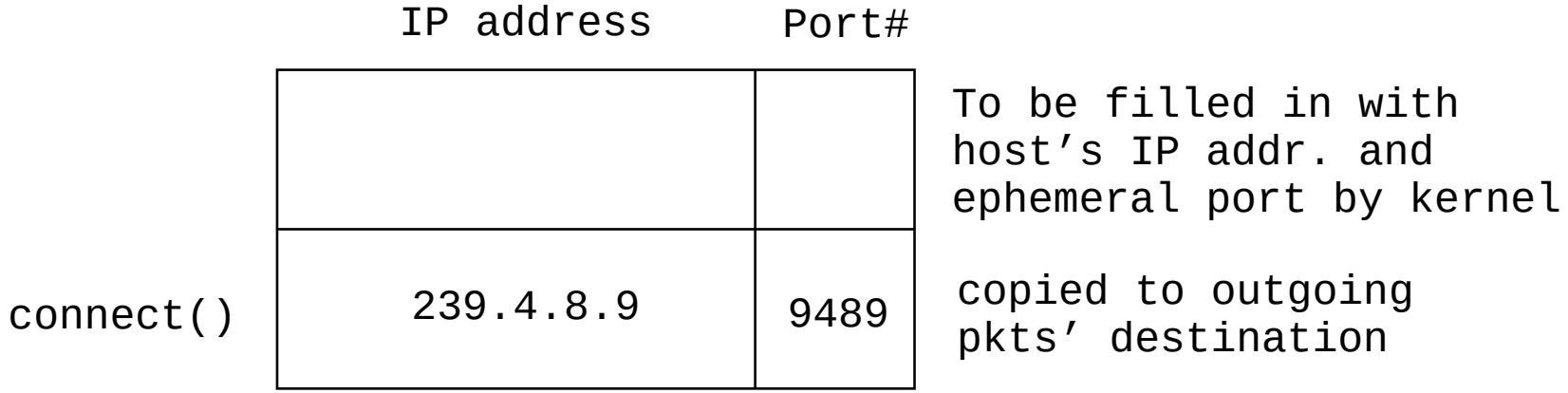
Sugih Jamin (jamin@eecs.umich.edu)

UDP Socket Addresses

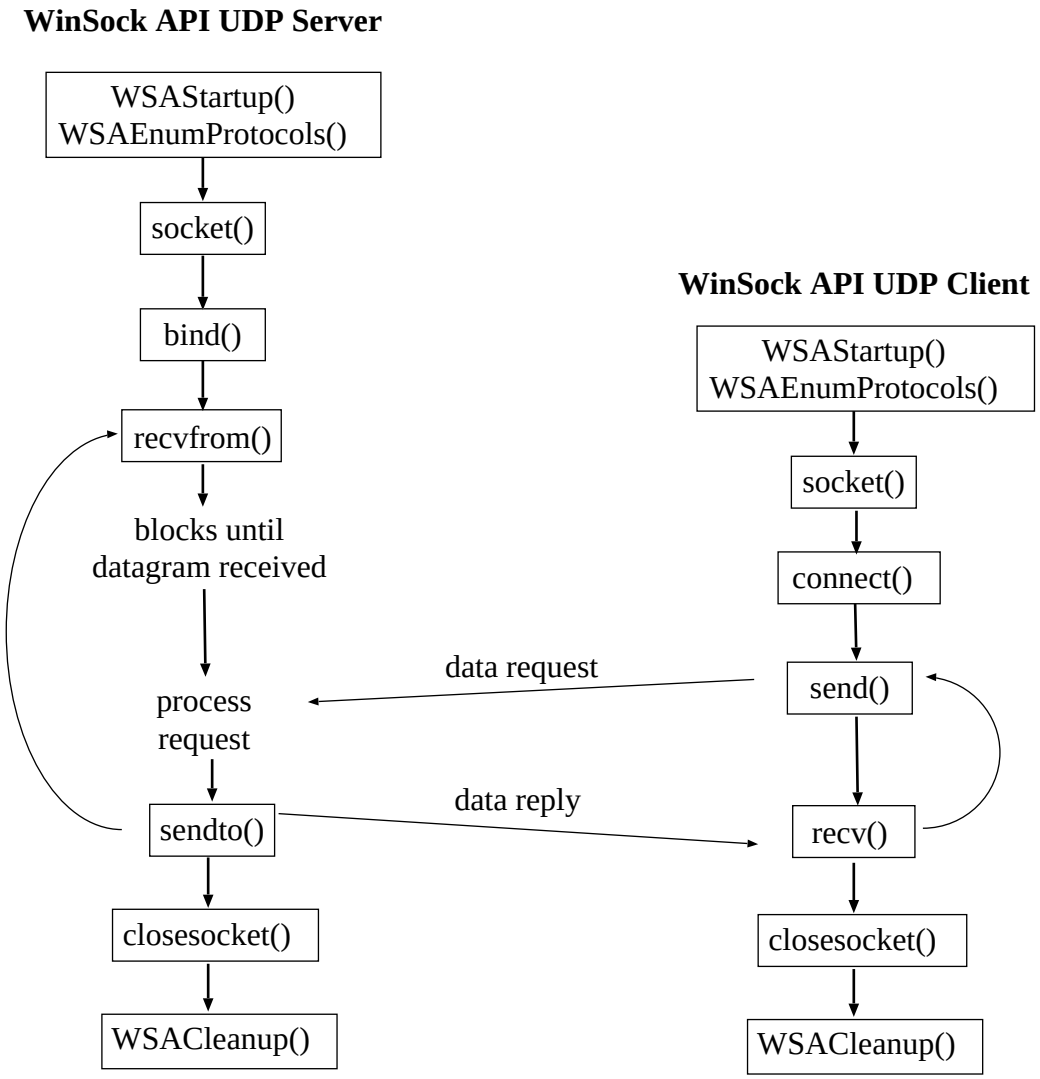
UDP Server:



UDP Client:



UDP with `connect()`



Sugih Jamin (jamin@eecs.umich.edu)

Byte Ordering

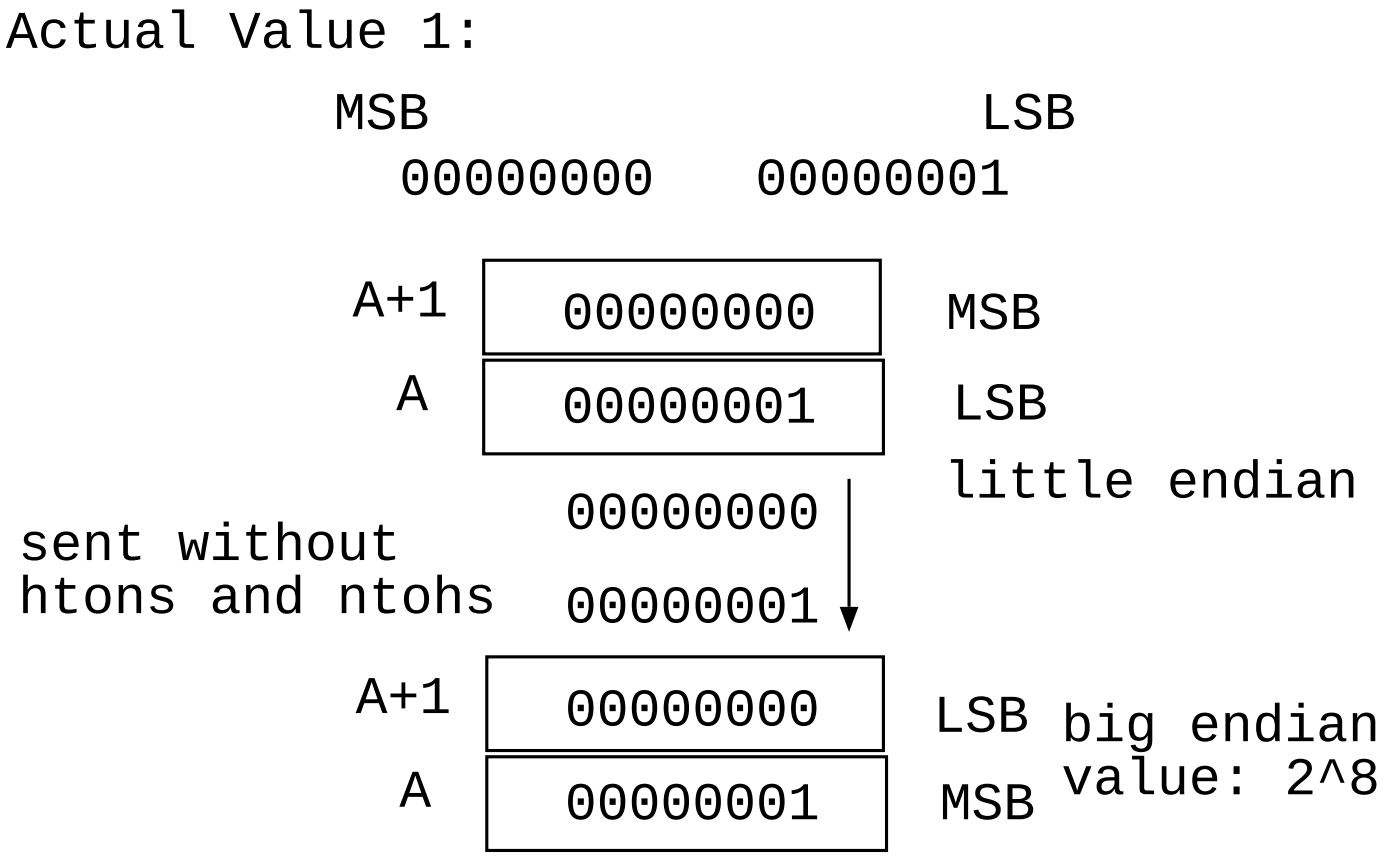
Big-endian: Most Significant Byte (MSB) in low address (sent/arrives first) (Sun Sparc, HP-PA)

Little-endian: MSB in high address (sent/arrives later) (Intel x86, PS2)

PowerPC and Alpha can be set to either mode

MMORG servers and backend servers may live on big-endian machines

Byte Ordering (cont)

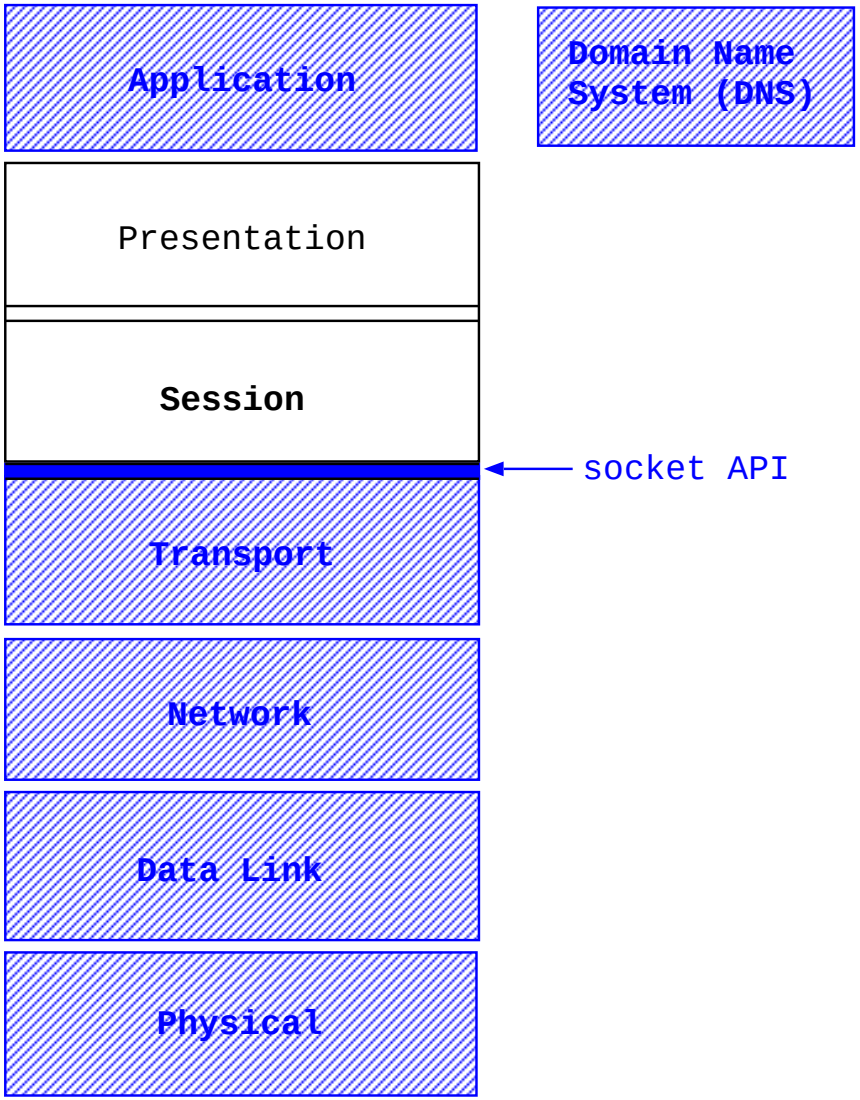


Byte Ordering (cont)

To ensure interoperability, ALWAYS translate
`short, long, int`
to (from) “network byte order” before (after) transmission
by using these macros:

`htons()`: host to network short
`htonl()`: host to network long
`ntohs()`: network to host short
`ntohl()`: network to host long

Protocol Layers



Naming and Addressing

DNS (Domain Name System) name in ASCII string:
`irl.eecs.umich.edu`

IP address written out in dotted-decimal (dd) ASCII string:
`141.213.8.193`

IP address in 32-bit binary representation:
`10001101 11010101 00001000 11000001`

Use names instead of addresses:
symbolic, easier to remember, variable length string

But fixed-length address provides more efficient handling and faster comparison, uses less memory and less bandwidth (bw)

Name and Adress Manipulation

Syscalls to map name to/from address:

- dns to b: `gethostbyname`
- b to dns: `gethostbyaddress`

and to change representation:

- dd to b: `inet_addr` (`inet_aton`)
- b to dd: `inet_ntoa`

dns to dd: `gethostbyname` plus `inet_ntoa`

Other useful functions:

- `gethostname`: returns DNS name of current host
- `getsockname`: returns IP address bound to socket (in binary) Used when addr and/or port is not specified (`INADDR_ANY`), to find out the actual addr and/or port used
- `getpeername`: returns IP address of peer (in binary)

Debugging Tools: use `tcpdump` to look at packets on the network
http://windump.polito.it/install/

tcpdump Output

```
% sudo tcpdump -i fxp0 host tail
tcpdump: listening on fxp0
08:52:07.913485 irl.eecs.umich.edu.3465 > tail.eecs.umich.edu.ssh: S 1334090569:1334090569(
08:52:07.913766 tail.eecs.umich.edu.ssh > irl.eecs.umich.edu.3465: S 1738389661:1738389661(
08:52:07.913820 irl.eecs.umich.edu.3465 > tail.eecs.umich.edu.ssh: . ack 1 win 57920 <nop,n
08:52:07.965499 tail.eecs.umich.edu.ssh > irl.eecs.umich.edu.3465: P 1:41(40) ack 1 win 579
08:52:07.965857 irl.eecs.umich.edu.3465 > tail.eecs.umich.edu.ssh: P 1:40(39) ack 41 win 57
08:52:07.966126 tail.eecs.umich.edu.ssh > irl.eecs.umich.edu.3465: . ack 40 win 57881 <nop,
08:52:07.966392 irl.eecs.umich.edu.3465 > tail.eecs.umich.edu.ssh: P 40:584(544) ack 41 win
08:52:07.966842 tail.eecs.umich.edu.ssh > irl.eecs.umich.edu.3465: . ack 584 win 57376 <nop
08:52:07.995417 tail.eecs.umich.edu.ssh > irl.eecs.umich.edu.3465: P 41:577(536) ack 584 wi
08:52:07.995842 irl.eecs.umich.edu.3465 > tail.eecs.umich.edu.ssh: P 584:608(24) ack 577 wi
08:52:07.996143 tail.eecs.umich.edu.ssh > irl.eecs.umich.edu.3465: . ack 608 win 57896 <nop
08:52:08.053504 tail.eecs.umich.edu.ssh > irl.eecs.umich.edu.3465: P 577:1001(424) ack 608
08:52:08.146672 irl.eecs.umich.edu.3465 > tail.eecs.umich.edu.ssh: . ack 1001 win 57920 <no
08:52:08.182531 irl.eecs.umich.edu.3465 > tail.eecs.umich.edu.ssh: P 608:1024(416) ack 1001
08:52:08.183112 tail.eecs.umich.edu.ssh > irl.eecs.umich.edu.3465: . ack 1024 win 57504 <no
08:52:08.566220 tail.eecs.umich.edu.ssh > irl.eecs.umich.edu.3465: P 1001:1929(928) ack 102
08:52:08.656695 irl.eecs.umich.edu.3465 > tail.eecs.umich.edu.ssh: . ack 1929 win 57920 <no
08:52:08.755094 irl.eecs.umich.edu.3465 > tail.eecs.umich.edu.ssh: P 1024:1040(16) ack 1929
08:52:08.755369 tail.eecs.umich.edu.ssh > irl.eecs.umich.edu.3465: . ack 1040 win 57904 <no
08:52:08.755452 irl.eecs.umich.edu.3465 > tail.eecs.umich.edu.ssh: P 1040:1088(48) ack 1929
08:52:08.755683 tail.eecs.umich.edu.ssh > irl.eecs.umich.edu.3465: . ack 1088 win 57872 <no
08:52:08.756357 tail.eecs.umich.edu.ssh > irl.eecs.umich.edu.3465: P 1929:1977(48) ack 1088
08:52:08.756654 irl.eecs.umich.edu.3465 > tail.eecs.umich.edu.ssh: P 1088:1152(64) ack 1977
```

Sources

Stevens, R., *Unix Network Programming*, 2nd. or 3rd. ed., Prentice-Hall, 2004. All you ever want to know about socket programming, even if you're using WinSock.

Mulholland & Hakala, *Programming Multiplayer Games*, Wordware Publishing, 2004. Useful mainly for the WinSock coverage.

Bettner, P. and Terrano, M., "1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond," GDC 2001

Bernier, Y.W., "Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization," GDC 2001

Rothschild, J., "Mpath: Technical Considerations," Mpath Interactive, 1997

Sources (cont)

Spohn, D., “Internet Game Timeline,” 2003
http://internetgames.about.com/library/weekly/aatimelinea.htm
http://internetgames.about.com/cs/gamingnews/a/goty2003.htm

Bartle, R., “Early MUD History,” 1990
http://www.mud.co.uk/richard/mudhist.htm

Ng, Y-S., “Internet Game Design,” Gamasutra, Aug. 1, 1997

Ng, Y-S., “Designing Fast Action Games for the Internet,” Gamasutra, Sept. 5, 1997

Rosedale and Ondrejka, “Enabling Player-Created Online Worlds with Grid Computing and Streaming,” Gamasutra, Sep. 18, 2003

Sources (cont)

Filstrup, B., Cronin, E., and Jamin, S., “An Evaluation of Cheat-Proofing Methods for Multiplayer Games,” NetGames 2002

Brockington and Greig, “Neverwinter Nights Client/Server Postmortem,” GDC 2003, Gamasutra Mar. 6, 2003

Isensee and Ganem, “Developing Online Console Games,” Gamasutra, Mar. 28, 2003

Palm, “The Birth of the Mobile MMOG,” Gamasutra, Sep. 19, 2003