# Some more on Project 1

PALLABI GHOSH (PALLABIG@ANDREW.CMU.EDU)

15-441 COMPUTER NETWORKS

RECITATION 1

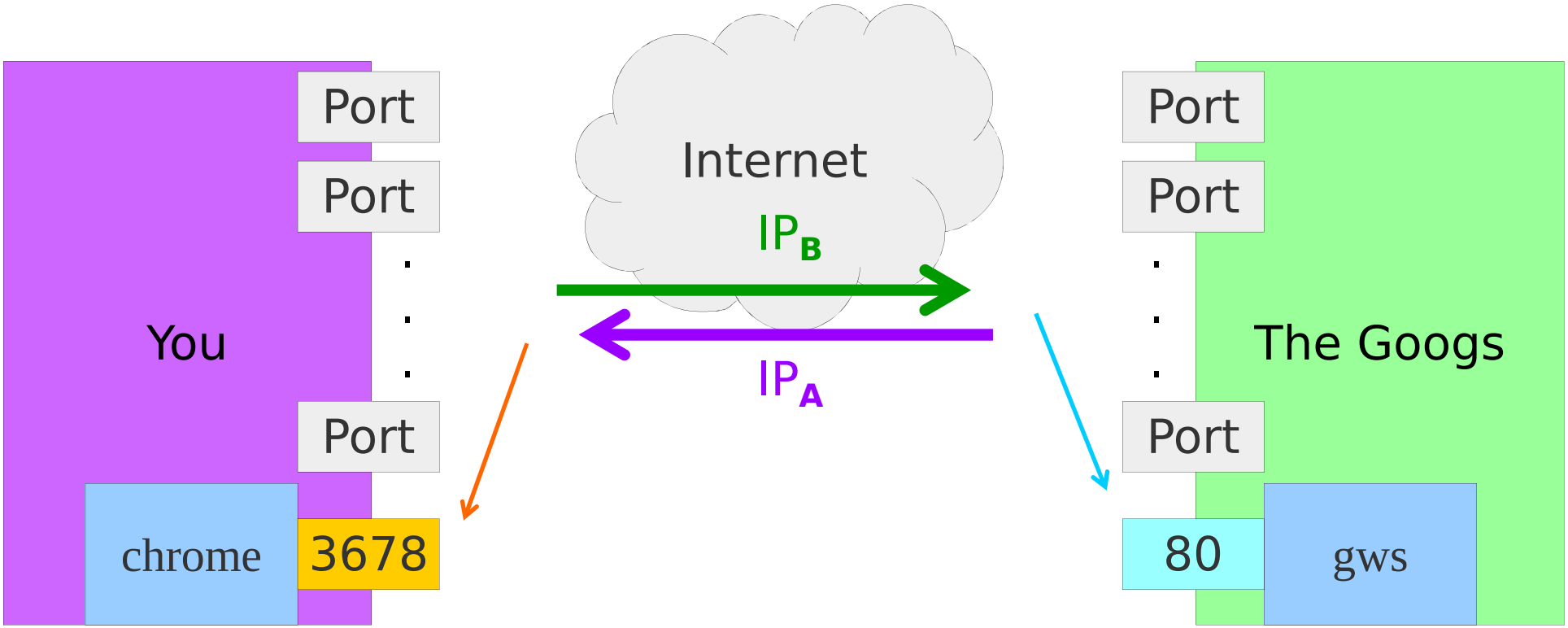
# Socket basics

# What's “in” a socket?

I want a program on computer A to talk to a program on computer B

Source <ip,port> + Destination <ip,port>

# IP? Port!?




Source <ip,port> + Destination <ip,port>

Identifies the Machine

Identifies a Socket
(multiple apps want to network!)

# How to: Server

- Create a socket via `socket()`
- Bind to an endpoint via `bind()`
- Listen for connections via `listen()`
- Accept connections via `accept()`
- Read from socket via `recv()`
- Write to socket via `send()`

# socket()

```
#include <sys/socket.h>

int socket(int socket_family, int socket_type,
 int protocol);
```

Generally set `socket_family` to PF_INET → IPv4

`socket_type` to SOCK_STREAM → TCP

`protocol` to 0 → Yes, TCP

# `socket()` Code Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>

int main(int argc, char* argv[])
{
    int sock = socket(PF_INET, SOCK_STREAM, 0);
    return EXIT_SUCCESS;
}
```

# `socket()` Error States

| Type | Meaning |
|---|---|
| EACCESS | Permission denied |
| EAFNOSUPPORT | Address family unsupported |
| EINVAL | Unknown protocol |
| EMFILE | Process file table overflow |
| ENFILE | Total open files limit reached |
| ENOBUFS or ENOMEM | System out of memory |
| EPROTONOSUPPORT | Specified protocol not supported |

# bind()

```
#include <sys/socket.h>

int bind(int sock, const struct
 sockaddr* addr, socklen_t addrlen);
```

After creating a socket as described, then create a sockaddr struct.

# bind() Code Example

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(int argc, char* argv[])
{    int sock = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(1025);
    addr.sin_addr.s_addr = INADDR_ANY;
    int err = bind(sock, (struct sockaddr *)
                                        &addr,
 sizeof(addr));
    return EXIT_SUCCESS;
}
```

# `bind()` Error States

| Type | Meaning |
| --- | --- |
| EADDRINUSE | Address in use already |
| EBADF | Invalid socket |
| EINVAL | Socket already bound |
| ENOTSOCK | File descriptor, not socket |

# listen()

```
#include <sys/socket.h>
int listen(int sock, int backlog);
```

After `bind()`ing, you can `listen()` for connections.

# `listen()` Code Example

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(int argc, char* argv[])
{
    int sock = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(1025);
    addr.sin_addr.s_addr = INADDR_ANY;
    int err = bind(sock, (struct sockaddr *)
                                    &addr, sizeof(addr));
    int err2 = listen(sock, 5);
    return EXIT_SUCCESS;
}
```

# `listen()` Error States

| Type | Meaning |
|---|---|
| EADDRINUSE | Port already listened to |
| EBADF | Not a valid descriptor |
| ENOTSOCK | Not a socket passed in |
| EOPNOTSUPP | Can't `listen()` on this socket |

# accept()

```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *
 addr, struct socklen_t * len);
```

After `listen()`ing, you can `accept()` connections.

Pass in the socket from before, and pointers to data structures defined by you.

These represent connected client state for future use.

# accept() Code Example

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(int argc, char* argv[])
{
    int sock = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr, caddr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(1025);
    addr.sin_addr.s_addr = INADDR_ANY;
    int err = bind(sock, (struct sockaddr *)
                                    &addr, sizeof(addr));
    int err2 = listen(sock, 5);
    socklen_t len = sizeof(caddr);
    int client = accept(sock, (struct sockaddr *)
                                                &caddr, &len);
    return EXIT_SUCCESS;
}
```

# `accept()` Error States 1

| Type | Meaning |
|---|---|
| EAGAIN or EWOULDBLOCK | No connections, non-blocking |
| EBADF | Not a valid descriptor |
| ECONNABORTED | Connection was aborted |
| EINTER | Interrupted before connection |
| EINVAL | Socket not listening |
| EMFILE | Per-process open file limit reached |
| ENFILE | System open file limit reached |

# `accept()` Error States 2

| Type | Meaning |
|---|---|
| ENOTSOCK | Descriptor of file, not socket |
| EOPNOTSUPP | Socket is not SOCK_STREAM |
| EFAULT | Can not write to addr |
| ENOBUFS, ENOMEM | Out of memory |
| EPROTO | Protocol error |
| EPERM | Firewall rules forbid connection |

# recv()

```
#include <sys/socket.h>

int recv(int sockfd, void * buf, size_t
 len, int flags);
```

After `accept()`ing, you can `recv()` data.

For now set `flags` to `0`

# recv() Code Example

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(int argc, char* argv[])
{
  char buf[256];
    int sock = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr, caddr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(1025);
    addr.sin_addr.s_addr = INADDR_ANY;
    int err = bind(sock, (struct sockaddr *)
                                        &addr, sizeof(addr));
    int err2 = listen(sock, 5);
    socklen_t len = sizeof(caddr);
    int client = accept(sock, (struct sockaddr *)

 &caddr, &len);
  ssize_t read = recv(client, buf, 256, 0);
    return EXIT_SUCCESS;
}
```

# send()

```
#include <sys/socket.h>

int send(int sockfd, void * buf, size_t
 len, int flags);
```

After `accept()`ing, you can `send()` data.

For now set `flags` to `0`

# send() Code Example

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(int argc, char* argv[])
{
  char buf[256];
    int sock = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr, caddr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(1025);
    addr.sin_addr.s_addr = INADDR_ANY;
    int err = bind(sock, (struct sockaddr *)
                                &addr, sizeof(addr));
    int err2 = listen(sock, 5);
    socklen_t len = sizeof(caddr);
    int client = accept(sock, (struct sockaddr *)
                                                    &caddr, &len);
  ssize_t sent = send(client, buf, 256, 0);
    return EXIT_SUCCESS;
}
```

# How to: Client

- Create a socket via `socket()`
- Connect to an endpoint via `connect()`
- Read from socket via `recv()`
- Write to socket via `send()`

# connect()

```
#include <sys/socket.h>

int connect(int socket, const struct sockaddr
 *serv_addr, socklen_t protocol);
```

Use socket as before, get `serv_addr` from `getaddrinfo()`.

Free with `freeaddrinfo()`.

# connect() Code Example

```
#include <netdb.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>

int main(int argc, char* argv[])
{
 struct addrinfo addr, *caddr;
 memset(&addr, 0, sizeof(addr));
 addr.ai_family = AF_UNSPEC;
 addr.ai_socktype = SOCK_STREAM;
 getaddrinfo("www.google.com", "80", &addr, &caddr);
 int sock = socket(caddr->ai_family, caddr->ai_socktype,
                                          caddr->ai_protocol);
 connect(sock, caddr->ai_addr, caddr->ai_addrlen);
   return EXIT_SUCCESS;
}
```

# Socket Programming Gotchas

## Endianness Matters: Network Byte Order

- `htons()` - host to network short
- `ntohs()` -network to host short

## Cleanup state—avoid memory leaks

- `freeaddrinfo()`
- Check correctness with `valgrind`

## Error Handling

- Tedious, but worth it (and required!)

## Timeouts

- Implement for robust networking behavior

# Socket Programming Gotchas

Never expect to `recv()` what you `send()`

- Assume partial receipt of data possible
- Use buffers intelligently to mitigate this
- Send byte counts first, read until finished

Prepare your code for random failures

- We introduce random faults when grading
- Test too—`ctrl+c` server and client randomly

Cleanup Allocated Memory

- `close()` sockets, etc.

# Concurrency

Threads

- Server gives each client its own thread
- Not in this class!

`select()`

- Watch a set of sockets (in main thread)
- Use `select()` to find sockets ready for I/O
- Server-side only—clients are agnostic

# Post Recitation 1 – More on Project1

Okay, so you can handle multiple connections!

But that is not enough...

# Reading data

- Check return value of recv()
  - Error – handle the error and clear up state.
  - If peer shutdown the connection, clear up state.
- Maintain state
  - Maintain a read buffer
  - Keep track of the number of bytes left to be read
  - May need multiple reads to get all data
  - But only one read per socket when select() returns.

# Writing data

- Check return value of send()
  - Error – handle the error and clear up state.
  - If peer shutdown the connection, clear up state.
- Maintain state
  - Maintain a write buffer
  - Keep track of the number of bytes left to be written
  - May need multiple writes to send all data
  - Number of bytes actually sent should be checked from the return value
  - Only one write per socket when select() returns.

# Exceptfds

- For handling out of band data
- Should be read one byte at a time!
- Not really needed for this project.

# Remember

- Code quality
- Code documentation
- Robustness
  - Handle all errors
  - Buffer overflows
  - Connection reset by peer

# Common mistakes

- Make sure your executable is named correctly
- tar your complete git repo and submit. Autolab expects to find a .git folder with your submission
- Don’t forget to update the tag when you make changes to your code. We run a checkout with the tag name and not your last commit
- Make sure your implementation is robust

# Checkpoint 1 Docs

- Makefile - make sure nothing is hard coded specific to your user; should build a file which runs the echo server (name it lisod)
- All of your source code - all .c and .h files
- readme.txt - file containing a brief description of your current implementation of server
- tests.txt - file containing a brief description of your testing methods for server
- vulnerabilities.txt - identify at least one vulnerability in your current implementation

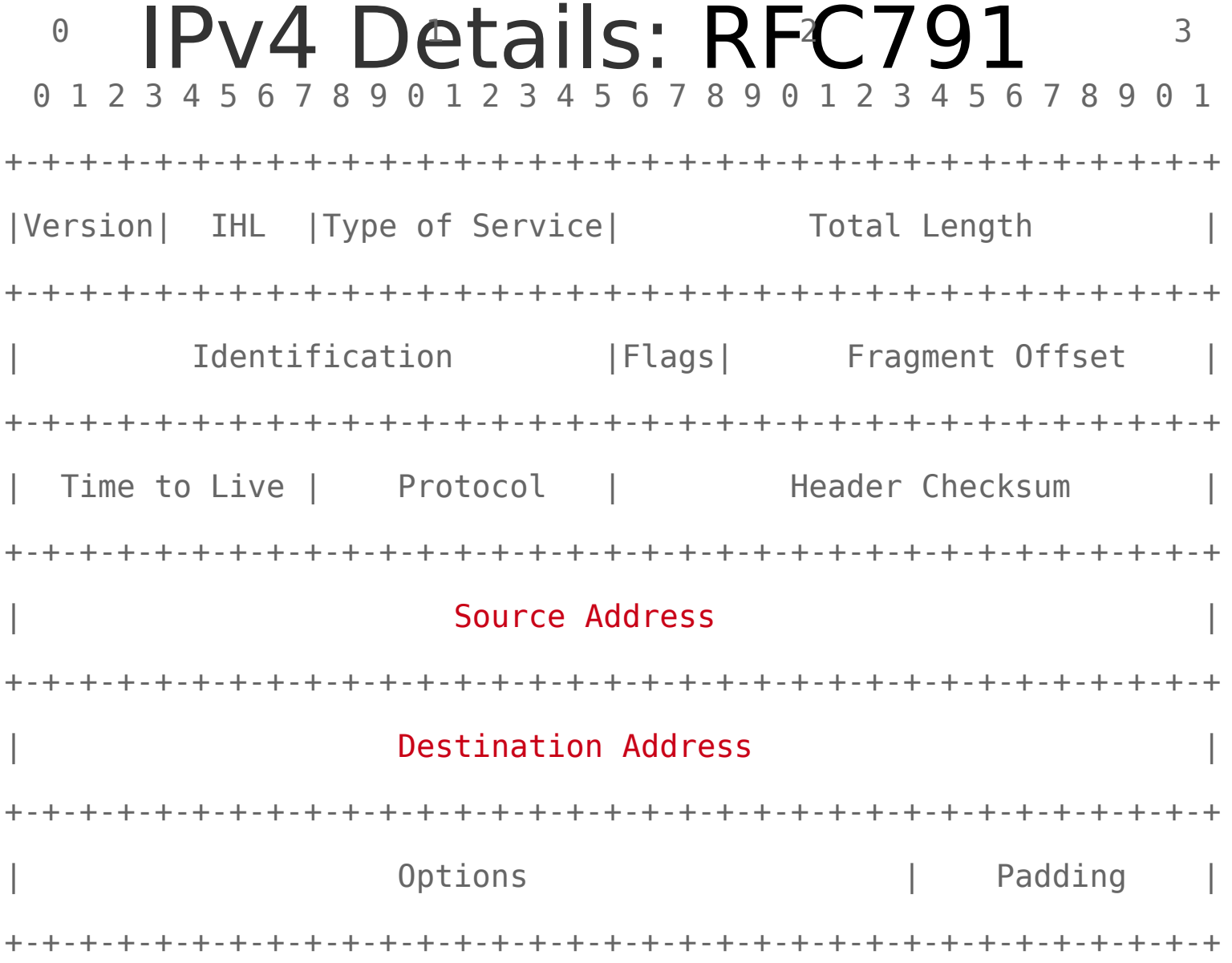
# Peek into the future

- Checkpoint 2 – September 19
  - Implement HTTP 1.1 parser and persistent connections – next recitation
- Checkpoint 3 – October 3
  - Implement HTTPS handshaking and persistent connections via TLS
  - Implement CGI server-side.

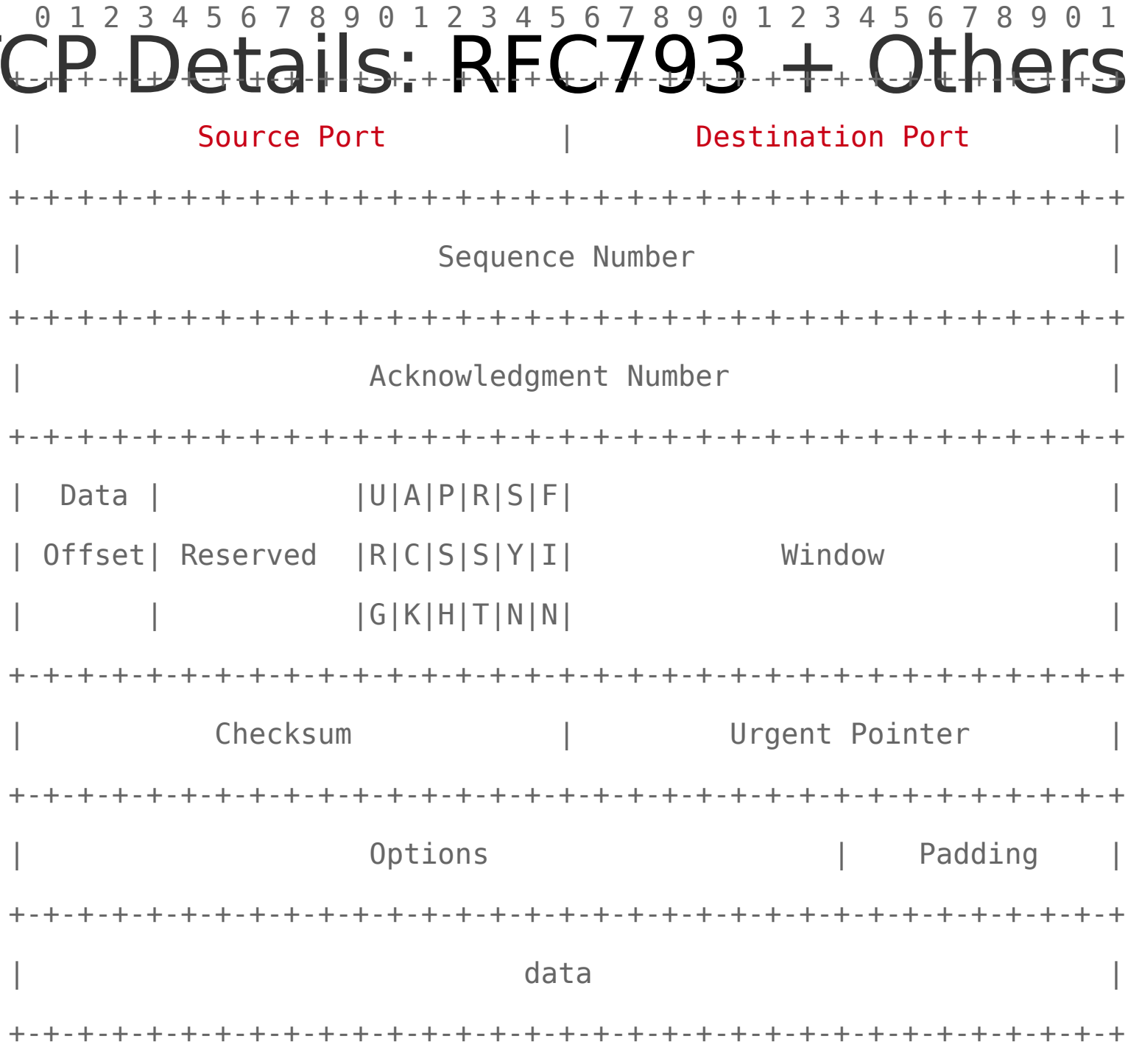
# All questions?

# Appendix

# IPv4 Details: RFC791



# TCP Details: RFC793 + Others



# socket_family

| Family | Meaning |
|---|---|
| PF_UNIX, PF_LOCAL | Local communication |
| PF_INET | IPv4 |
| PF_INET6 | IPv6 |
| PF_IPX | IPX - Novell |
| PF_NETLINK | Kernel user interface device |
| PF_X25 | ITU-T X.25 |
| PF_AX25 | Amateur radio AX.25 |
| PF_ATMPVC | Raw ATM PVCs |
| PF_APPLETALK | Appletalk |
| PF_PACKET | Low level packet interface |

# socket_type

| Type | Meaning |
| --- | --- |
| SOCK_STREAM | Sequenced, reliable TCP |
| SOCK_DGRAM | Connectionless UDP |
| SOCK_SEQPACKET | SOCK_STREAM w/ fixed length |
| SOCK_RAW | Raw network protocol access |
| SOCK_RDM | Reliable datagram no order |
| SOCK_PACKET | Deprecated |

# sin_family

| Address Family | Meaning |
|---|---|
| AF_INET | IPv4 |
| AF_INET6 | IPv6 |
| AF_UNIX | Local communication |
| AF_APPLETALK | Appletalk addressing |
| AF_PACKET | Packet addressing |
| AF_X25 | ITU-T X.25 addressing |
| AF_NETLINK | Between kernel and user |

# sin_port

Set to port you want to connect/listen

| Conversion Function | Explanation |
|---|---|
| `uin16_t htons(uint16_t)` | Host to network short |
| `uin32_t htonl(uint32_t)` | Host to network long |
| `uin16_t ntohs(uint16_t)` | Network to host short |
| `uin32_t htonl(uint32_t)` | Network to host long |

# recv() flags

| Flag | Meaning |
|---|---|
| MSG_DONTWAIT | Non-blocking check for data |
| MSG_ERRQUEUE | Receive errors from socket |
| MSG_OOB | Out-of-band data request |
| MSG_PEEK | Return data without removing |
| MSG_TRUNC | Return real packet length |
| MSG_WAITALL | Block until full request satisfied |

# recv() Error States 1

| Type | Meaning |
|---|---|
| EAGAIN | Would block but not supposed |
| EBADF | Not a valid descriptor |
| ECONNREFUSED | Remote host refused conn |
| EFAULT | Receive buffer pointer bad |
| EINTER | Receive interrupted by interrupt |
| EINVAL | Invalid argument passed |
| ENOMEM | Out of Memory |
| ENOTCONN | Socket not connected yet |
| ENOTSOCK | Socket argument isn't a socket |

# send() flags

| Flag | Meaning |
|---|---|
| MSG_CONFIRM | Link layer forward progress |
| MSG_DONTROUTE | Don't use a gateway |
| MSG_DONTWAIT | Non-blocking send |
| MSG_EOR | Terminate record |
| MSG_MORE | Caller has more data to send |
| MSG_NOSIGNAL | No SIGPIPE on error |
| MSG_OOB | Send out-of-band data |

# `send()` Error States 1

| Type | Meaning |
|---|---|
| EACCESS | Write permission denied |
| EAGAIN or EWOULDBLOCK | Block not supposed to |
| EBADF | Invalid descriptor specified |
| ECONNRESET | Connection reset by peer |
| EDESTADDRREQ | Not connecting, no peer set |
| EFAULT | Invalid user address specified |
| EINTER | Signal interrupt |
| EINVAL | Invalid argument |
| EISCONN | Connected already |

# `send()` Error States 2

| Type | Meaning |
|---|---|
| EMSGSIZE | Atomic send, too big message |
| ENOBUFS | Output queue physically full |
| ENOMEM | Out of memory |
| ENOTCONN | Socket not connected |
| ENOTSOCK | Argument is not a socket |
| EOPNOTSUPP | Some flags are incorrect |
| EPIPE | Local end shutdown |

# ai_family

| Family | Meaning |
| --- | --- |
| AF_UNSPEC | Any protocol family |
| AF_INET | IPv4 |
| AF_INET6 | IPv6 |

# `connect()` Error States 1

| Type | Meaning |
|---|---|
| EACCESS | Permission denied |
| EACCESS, EPERM | Connect broadcast without flag |
| EADDRINUSE | Local address already in use |
| EAFNOSUPPORT | Incorrect address family |
| EAGAIN | No more free local ports |
| EALREADY | Non-blocking, previous incomplete connection |
| EBADF | File descriptor not valid |

# `connect()` Error States 2

| Type | Meaning |
|---|---|
| ECONNREFUSED | No remote listening socket |
| EFAULT | Socket address structure outside user address space |
| EINPROGRESS | Non-blocking, cannot immediately connect |
| EINTER | Interrupt by caught signal |
| EISCONN | Socket already connected |
| ENETUNREACH | Unreachable network |
| ENOTSOCK | File descriptor not a socket |