TCP Gearup I

(Take 2)
Overview

• How this project fits into IP
• What you will build
• How to debug/test in wireshark
• Implementation notes
• Any questions you have
The Big Picture: Last time

- User commands
  - Upper layers
    - Test packets
    - Routing (RIP) (routers only)
  - "Network layer"
    - IP API
      - IP Forwarding
    - "Link layer" (Interfaces)
      - if0: UDP Socket
      - if1: UDP Socket
  - Other commands (up, down, l*, …)
Where we are now

A new “higher layer” in your IP stack (on the same level as test packets)
Where we are now

- A new "higher layer" in your IP stack (on the same level as test packets)
- **For hosts ONLY**
  - You are done modifying your router at this point
Remember this picture?

One or more nodes within the network

End host

Application
Presentation
Session
Transport
Network
Data link
Physical

End host

Application
Presentation
Session
Transport
Network
Data link
Physical

Application Protocol

Transport Protocol

Network Protocol

Link-Layer Protocol

One or more nodes within the network

JUST CONSIDER ENDPOINTS!
Let’s break it down

“Applications”

IP REPL Commands
send, lr, ...

TCP REPL Commands
send (s), recv (r),
send file (sf), receive file (rf)

“Network stack”

Test Pkt. Handler
Test packets
(Protocol 0)

Sockets API
TCP Stack
(Transport layer)

TCP packets
(Protocol 6)

IP stack
What goes in your TCP stack?
TCP STACK: THE COMPONENTS

- 'APPS' API calls
  - 'REPL' to 9595
  - 'REPL' to C 10.0.0.1 9899

TCP STACK

Socket API
  - (vconnect, vlisten, ...) (like go/c/... socket API)

Socket: Two Types

- "Normal" sockets
  - One per active TCP connection
  - Has TCB (buffers, TCP state, etc.)

- Listen sockets
  - One per open listen port
  - Has no TCB (can't send/recv)

TCP Logic
  - State Machine
  - Sliding Window

Packet Events

- Socket table
  - Maps packets => sockets based on header info

NEW HANDLER (PROTO = 6)

TCP STACK

IP

IP Layer

SendIP(destAddr, protocol, bytes)

IP

TCP STACK

"Normal" sockets

Listen sockets

Socket table
THE PARTS:

API for sockets: abstraction for creating and using TCP connections

Example: Go’s socket API
```go
c conn, err := net.Dial("tcp", "10.0.0.1:80")
... 
someBuf := make([]byte, ...) 
conn.Write(someBuf)
```

Example: our socket API (yours can look different)
```go
c conn, err := tcpstack.VConnect(addr, port)
... 
someBuf := make([]byte, ...) 
conn.VWrite(someBuf)
```

Guidelines: “Socket API” specification in docs
(You get to design your own API!!)
VListen(port)          // Listen on a port
VConnect(addr, port)   // Connect to a socket
VAccept(...)          // Accept new connections (more on this later)

VWrite(...)            // Send on a socket
VRead(...).            // Recv on a socket

VClose(...)            // Close a socket

Guidelines: “Socket API” specification in docs
Focus for Milestone 1

VListen(port)          // Listen on a port
VConnect(addr, port)   // Connect to a socket
VAccept(. . .)         // Accept new connections (more on this later)

VWrite(. . .)           // Send on a socket
VRead(. . .).           // Recv on a socket

VClose(. . .)           // Close a socket

Guidelines: “Socket API” specification in docs
TCP stack: logic that happens “under the hood” to make sockets work (i.e., the TCP protocol)

- Should be a separate library you initialize at host startup (like your IP stack)
- Uses your IP stack to send/recv packets
  - IPSend(destIP, protocol, bytes)
  - New handler for TCP (protocol #6)

Guidelines: “TCP notes” in docs
REPL commands: how we’ll test your

=> Think of these like “applications” that use your socket API

// Basic stuff (test your API)
a Listen on a port; accept new connections
c Connect to a TCP socket
ls List sockets

s Send on a socket
r Receive on a socket

c1 Close socket

// Ultimate goal
sf Send a file
rf Receive a file

Focus for Milestone 1
Demo!
How to test TCP

Most of the time, use linear-r1h2 network
• Only one router, no need for RIP
• Can mainly use reference router
  – Will release an updated reference router next week
    (has extra features for later in project)

Note: watching traffic in wireshark works differently in this project!
=> See "TCP getting started" guide for details
Roadmap

Milestone 1
- Initial design for API and TCP stack
- Listen and establish connections => create sockets/TCB
- TCP handshake
- accept, connect, and start of ls REPL commands
How to think about connections

aka. Most important thing for Milestone 1

<table>
<thead>
<tr>
<th>SID</th>
<th>LAddr</th>
<th>LPort</th>
<th>RAddr</th>
<th>RPort</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0.0.0</td>
<td>9999</td>
<td>0.0.0.0</td>
<td>0</td>
<td>LISTEN</td>
</tr>
<tr>
<td>1</td>
<td>10.1.0.2</td>
<td>9999</td>
<td>10.0.0.1</td>
<td>58060</td>
<td>ESTABLISHED</td>
</tr>
</tbody>
</table>

Relevant concept material
• Lec 12 (ports), Lec 13 (TCP handshake)
• HW2 problem 3
How to think about connection setup

Scenario:  
- B listens on port 1234 (i.e., “a 1234”)  
- A connects to B’s port (i.e., “c 10.1.0.2 1234”)

A initiates connection using VConnect. This creates a new normal socket for this connection (see table) and sends a SYN.

When the socket is created, A’s TCP stack picks a random (unused) source port for this connection.

How to know it goes to this specific socket, and not the listen socket? See next page.

B opens a new listen port, using VListen. This creates a new Listen socket (see table below).

When the SYN is received, B maps it to the open listen socket.

When a normal socket receives a packet, we handle it according to the TCP state machine. Here, state == SYN_SENT, so per the protocol, the action when receiving a SYN+ACK is to send an ACK and move to ESTABLISHED.

A’s table

<table>
<thead>
<tr>
<th>LOCAL</th>
<th>REMOTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0.0.1</td>
<td>33578</td>
</tr>
<tr>
<td>10.0.1.2</td>
<td>1234</td>
</tr>
</tbody>
</table>

B’s table

<table>
<thead>
<tr>
<th>LOCAL</th>
<th>REMOTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0.0.1</td>
<td>33578</td>
</tr>
<tr>
<td>10.0.1.2</td>
<td>1234</td>
</tr>
<tr>
<td>10.0.0.1</td>
<td>33578</td>
</tr>
<tr>
<td>10.0.0.1</td>
<td>33578</td>
</tr>
</tbody>
</table>

When the SYN is received, B maps it to the open listen socket. When the listen socket receives a SYN, it creates a new normal socket for this specific connection between A and B, then sends SYN+ACK.
How do we map an incoming packet to a socket? To take a look at this, let’s examine what happens to the last packet in the handshake when it’s received by B (step 5 above):

The packet’s source/dest IP and port numbers act like a unique identifier that identifies this connection => this is called the 4-tuple. We map packets to normal sockets based on the 4-tuple.

To summarize, here’s how the matching process works. When receiving packet P, check the socket table for a matching socket:
1. Check for a normal socket with a matching 4-tuple (dstIP, dstPort, srcIP, srcPort)
2. If there is no matching normal socket, check for a listen socket where localPort == P.dstPort
3. If no match, this packet isn’t for any known socket, so drop the packet.

Another example: What if we received a different packet that looked like this?

This packet has a different source port, so it has a different 4-tuple! Therefore, it must be for another connection (or it’s an attempt to start a new one).

=> Thus, this packet should map to the listen socket
Connection setup API: recap

VConnect

• “Active OPEN” in RFC
• Initiates new connection, returns normal socket
• Blocks until connection is established, or times out
Connection setup API: recap

**VConnect**
- “Active OPEN” in RFC
- Initiates new connection, returns normal socket
- Blocks until connection is established, or times out

**VListen**
- “Passive OPEN” in RFC
- Returns new listen socket

**VAccept**
- Input: a listen socket
- Blocks until a client connection is established
- Returns new normal socket
Connection setup API: recap

**VConnect**
- "Active OPEN" in RFC
- Initiates new connection, returns normal socket
- Blocks until connection is established, or times out

**VListen**
- "Passive OPEN" in RFC
- Returns new listen socket

**VAccept**
- Input: a listen socket
- Blocks until a client connection is established
- Returns new normal socket

How exactly you implement this is up to you, but your API should have calls like this (This isn’t arbitrary—it matches what the kernel API looks like)
Think back to your Snowcast server...

```go
// Create listen socket (bind)
listenConn, err := net.ListenTCP("tcp4", addr)

for {
    // Wait for a client to connect
    clientConn, err := listenConn.Accept()
    if err != nil {
        // . . .
    }
    // . . .
    go handleClient(clientConn)
}

func handleClient (conn net.Conn) {
    conn.Read(. . .)
}
```
// Create listen socket (bind)
listenConn, err := net.ListenTCP("tcp4", addr)

for {
    // Wait for a client to connect
    clientConn, err := listenConn.Accept()
    if err != nil {
        // . . .
    }
    // . . .
    go handleClient(clientConn)
}

func handleClient (conn net.Conn) {
    conn.Read(. . .)
}
func ACommandREPL() { // Runs as separate thread/goroutine

    // Create listen socket (bind)
    listenConn, err := tcpstack.VListen(port)

    for {
        // Wait for a client to connect
        clientConn, err := listenConn.VAccept()
        if err != nil {
            // . . .
        }

        // Store clientConn to use by other REPL commands
    }
}
Summary: two types of sockets

<table>
<thead>
<tr>
<th>Type</th>
<th>When created</th>
<th>What it does</th>
<th>What’s in it?*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listen sockets</td>
<td>&quot;a&quot; command (VListen)</td>
<td>• &quot;I want to receive new connections on this port&quot;</td>
<td>• List of sockets for new/pending connections</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Always in state LISTEN</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Not connected to another endpoint! (can’t send/recv on it, has no TCB</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>*: At minimum, for now</td>
<td></td>
</tr>
<tr>
<td>&quot;Normal&quot; sockets</td>
<td>&quot;c&quot; command (VConnect)</td>
<td>• Used for &quot;normal&quot; TCP connections between endpoints</td>
<td>• TCB (send/recv buffers, all other TCP protocol</td>
</tr>
<tr>
<td></td>
<td>&quot;a&quot; command (VAccept)</td>
<td></td>
<td>state)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* => VTCPLListener in API example

* => VCTPConn in API example
Implementation stuff
Ways to build the API

```go
conn, err := tcpstack.VConnect(addr, port)
...
conn.VWrite(someBuf)
```

Go-style

- VConnect/VAccept/VListen return structs for normal/listen sockets
- Other functions (VAccept, VWrite, …) are methods on these structs

More info: “Socket API example” in docs
Ways to build the API

**Go-style**
- `VConnect/VAccept/VListen` return **structs** for normal/listen sockets
- Other functions (`VAccept`, `VWrite`, …) are **methods** on these structs

```go
conn, err := tcpstack.VConnect(addr, port)
...
conn.VWrite(someBuf)
```

**C-style**
- `VConnect/VAccept/VListen` return **numbers** (like file descriptors)
- Other functions (`VAccept`, `VRead`, …) take **socket number** as argument

```c
int sock_fd = VConnect(addr, port)
...
VWrite(sock_fd, some_buffer)
```
Ways to build the API

Go-style
- VConnect/VAccept/VListen return structs for normal/listen sockets
- Other functions (VAccept, VWrite, ...) are methods on these structs
- In REPL: map socket ID => struct

C-style
- VConnect/VAccept/VListen return numbers (like file descriptors)
- Other functions (VAccept, VRead, ...) take socket number as argument
- In TCP stack: map socket ID => struct
Ways to build the API

Go-style
- VConnect/VAccept/VListen return structs for normal/listen sockets
- Other functions (VAccept, VWrite, …) are methods on these structs
- In REPL: map socket ID => struct

```
int sock_fd = VConnect(addr, port)

VWrite(sock_fd, some_buffer)
```

C-style
- VConnect/VAccept/VListen return numbers (like file descriptors)
- Other functions (VAccept, VRead, …) take socket number as argument
- In TCP stack: map socket ID => struct

```
conn, err := tcpstack.VConnect(addr, port)

... conn.VWrite(someBuf)
```

=> How you implement this is up to you (don’t even need to pick one of these)!

More info: “Socket API example” in docs
Building TCP packets

• MUST use standard TCP header
• Encapsulation: TCP packet => payload of virtual IP packet
• Once again, you don’t need to build/parse this yourself

⇒ See the TCP-in-IP example for a demo on how to build/parse a TCP header (mostly uses same libraries as before)
The TCP checksum

... is pretty weird

Computing the TCP checksum involves making a “pesudo-header” out of some IP and TCP header fields:
The TCP checksum... is pretty weird

Computing the TCP checksum involves making a “pesudo-header” out of some IP and TCP header fields:

⇒ You don’t need this working for milestone 1
⇒ See the TCP-in-IP example for a demo of how to compute/verify it
Reference implementation

• Our implementation of TCP
• Try it and compare with your version!
Reference implementation

• Our implementation of TCP
• Try it and compare with your version!

Note: we’re using a new reference this year (after 8+ years!)
• We’ve tested as best we can, but there may be bugs
• See Ed FAQ, docs FAQ for list of known bugs
• Let us know if you have issues!

⇒ If the spec disagrees with the reference implementation, the spec wins—don’t propagate buggy behavior (please help us find any discrepancies!)
Roadmap

Milestone I

• Start of your API and TCP stack
• Listen and establish connections => create sockets/TCB
• TCP handshake
• accept, connect, and start of ls REPL commands

Be prepared to talk about what goes in your data structures, design plan, etc, similar to your IP milestone
Roadmap

Milestone II

• Basic **sending and receiving** using your sliding window/send receive buffers
• Plan for the remaining features
Roadmap

**Final deadline**
- Retransmissions (+ computing RTO from RTT)
- Zero-window probing
- Connection teardown
- Sending and receiving files \((sf, rf)\)
Where to get more info

TCP REPL Commands
- send (s), recv (r)
- send file (sf), receive file (rf)

Our docs: "REPL commands" spec

Our docs: "Socket API" example

Sockets API

TCP Stack

IP stack
Where to get more info

TCP REPL Commands
- send (s), recv (r),
- send file (sf), receive file (rf)

Our docs: “REPL commands” spec

Sockets API

Our docs: “Socket API” example

Guidelines: “TCP notes” in our docs
- Links to relevant RFCs (eg. RFC9293)
- Our modifications/notes on the RFCs

TCP Stack

- TCP-in-IP example (how to make/parse packets)
- IP docs

IP stack
Closing thoughts

• Use your milestone time wisely!

• **Wireshark is the best way to test**—**use it!**

• As you work with your IP code, consider refactoring!
  – You’re going to be working with this code for \( \geq 3 \) weeks

• Stuck? Don’t know what’s required? Just ask! (And see Ed FAQ)

  We are here to help!