A UDP server and client in Go

etting from Golang's net package down to the Linux kernel methods invoked whe UDP messages are sent.

by Ciro S. Costa - Sep 30, 2018
tags: linux , networking , go

Hey,

While it's prevalent to see implementations of TCP servers in Golang, it's not very common to see the same when it comes to UDP.

Besides the many differences between UDP and TCP, using Go it feels like these are pretty much alike, except for little details that arise from each protocol specifics.

If you feel like some Golang UDP knowledge would be valuable, make sure you stick to the end.

As an extra, this article also covers the underlying differences between TCP and UDP when it comes to the syscalls that Golang uses under the hood, as well as some analysis of what the Kernel does when those syscalls get called.

Stay tuned!

Overview

As a goal for the blog post, the final implementation should look like an "echo channel", where whatever a client writes to the server, the server echoes back.

```
        .---> HELLO!! -->-.
        |                 |
client -*                 *--> server --.
   ^                                    |
   |                                    |
   *---<----- HELLO!! ---------<--------*
```

Being UDP a protocol that doesn't guarantee reliable delivery, it might be the case that the server receives the message, and it might be the case that the client receives the echo back from the server.

The flow **might** complete successfully (or not).

```
        .---> HELLO!! -->-.
        |                 |
client -*                 *--> server --.
                                        |
                                        |
            whoops, lost! -----<--------*
```

Not being connection-oriented, the client won't really "establish a connection", like in TCP; whenever a message arrives at the server, it won't "write a response back to the connection", it will only direct a message to the address that wrote to it.

With that in mind, the flow should look like this:

```
TIME    DESCRIPTION

t0      client and server exist

          client                          server
          10.0.0.1                        10.0.0.2


t1      client sends a message to the server
```

```
          client      ------------>      server
          10.0.0.1         msg           10.0.0.2
                     (from:10.0.0.1)
                     (to:  10.0.0.2)


t2      server receives the message, then it takes
        the address of the sender and then prepares
        another message with the same contents and
        then writes it back to the client

          client      <------------      server
          10.0.0.1         msg2          10.0.0.2
                     (from:10.0.0.1)
                     (to:  10.0.0.2)


t3      client receives the message

          client                         server
          10.0.0.1                       10.0.0.2

           thxx!! :D :D


ps.: ports omitted for brevity
```

That said, let's see how that story rolls out in Go.

> Also,
>
> If you'd like a real deep dive, make sure you consider these books:
>
> - Computer Networking: A top-down approach
> - Unix Network Programming, Volume 1: The Sockets Networking API (3rd Edition)
> - The Linux Programming Interface
>
> The first takes the approach of going from the very high level part of the networking stack (application layer), and then goes down to the very bottom, explaining the details of the protocols in there as it goes through them - if you need an excellent refresher on networking concepts without digging into the implementation details, check this one out!
>
> The other two are more about Linux and Unix in general - very worthwhile if you're more focused on the implementation.

Have a good reading!

Sending UDP packets using Go

Kicking off with the whole implementation at once (full of comments), we can start depicting it, understanding piece by piece, until the point that we can understand each and every interaction that happens behind the scenes.

```
// client wraps the whole functionality of a UDP client that sends
// a message and waits for a response coming back from the server
// that it initially targetted.
func client(ctx context.Context, address string, reader io.Reader) (err erro
	// Resolve the UDP address so that we can make use of DialUDP
	// with an actual IP and port instead of a name (in case a
	// hostname is specified).
	raddr, err := net.ResolveUDPAddr("udp", address)
	if err != nil {
		return
	}

	// Although we're not in a connection-oriented transport,
	// the act of `dialing` is analogous to the act of performing
	// a `connect(2)` syscall for a socket of type SOCK_DGRAM:
	// - it forces the underlying socket to only read and write
	//   to and from a specific remote address.
	conn, err := net.DialUDP("udp", nil, raddr)
	if err != nil {
		return
	}

	// Closes the underlying file descriptor associated with the,
	// socket so that it no longer refers to any file.
	defer conn.Close()

	doneChan := make(chan error, 1)

	go func() {
		// It is possible that this action blocks, although this
		// should only occur in very resource-intensive situations:
		// - when you've filled up the socket buffer and the OS
		//   can't dequeue the queue fast enough.
		n, err := io.Copy(conn, reader)
		if err != nil {
			doneChan <- err
			return
		}

		fmt.Printf("packet-written: bytes=%d\n", n)

		buffer := make([]byte, maxBufferSize)

		// Set a deadline for the ReadOperation so that we don't
		// wait forever for a server that might not respond on
		// a resonable amount of time.
```

```
                deadline := time.Now().Add(*timeout)
                err = conn.SetReadDeadline(deadline)
                if err != nil {
                        doneChan <- err
                        return
                }

                nRead, addr, err := conn.ReadFrom(buffer)
                if err != nil {
                        doneChan <- err
                        return
                }

                fmt.Printf("packet-received: bytes=%d from=%s\n",
                        nRead, addr.String())

                doneChan <- nil
        }()

        select {
        case <-ctx.Done():
                fmt.Println("cancelled")
                err = ctx.Err()
        case err = <-doneChan:
        }

        return
}
```

Having the client code, we can now depict it, exploring each of its nuances.

Address resolution

Before we even start creating a socket and carrying about sending the information to the server, the first thing that happens is a name resolution that translates a given name (like, `google.com`) into a set of IP addresses (like, `8.8.8.8`).

The way we do that in our code is with the call to net.ResolveUDPAddr, which in a Unix environment, goes all the way down to performing the DNS resolution via the following stack:

```
(in a given goroutine ...)
    >>  0  0x00000000004e5dc5 in net.(*Resolver).goLookupIPCNAMEOrder
           at /usr/local/go/src/net/dnsclient_unix.go:553
    >>  1  0x00000000004fbe69 in net.(*Resolver).lookupIP
           at /usr/local/go/src/net/lookup_unix.go:101
        2  0x0000000000514948 in net.(*Resolver).lookupIP-fm
           at /usr/local/go/src/net/lookup.go:207
        3  0x000000000050faca in net.glob..func1
           at /usr/local/go/src/net/hook.go:19
    >>  4  0x000000000051156c in net.(*Resolver).LookupIPAddr.func1
```

```
          at /usr/local/go/src/net/lookup.go:221
        5 0x00000000004d4f7c in internal/singleflight.(*Group).doCall
          at /usr/local/go/src/internal/singleflight/singleflight.go:95
        6 0x000000000045d9c1 in runtime.goexit
          at /usr/local/go/src/runtime/asm_amd64.s:1333

(in another goroutine...)
0  0x0000000000431a74 in runtime.gopark
   at /usr/local/go/src/runtime/proc.go:303
1  0x00000000004416dd in runtime.selectgo
   at /usr/local/go/src/runtime/select.go:313
2  0x00000000004fa3f6 in net.(*Resolver).LookupIPAddr          <<
   at /usr/local/go/src/net/lookup.go:227
3  0x00000000004f6ae9 in net.(*Resolver).internetAddrList      <<
   at /usr/local/go/src/net/ipsock.go:279
4  0x000000000050807d in net.ResolveUDPAddr                     <<
   at /usr/local/go/src/net/udpsock.go:82
5  0x000000000051e63b in main.main
   at ./resolve.go:14
6  0x0000000000431695 in runtime.main
   at /usr/local/go/src/runtime/proc.go:201
7  0x000000000045d9c1 in runtime.goexit
   at /usr/local/go/src/runtime/asm_amd64.s:1333
```

If I'm not mistaken, the overall process looks like this:

1. it checks if we're giving an already IP address or a hostname; if a hostname, then
2. looks up the host using the local resolver according to the lookup order specified by the system; then,
3. eventually performs an actual DNS request asking for records for such domain; then,
4. if all of that succeeds, a list of IP addresses is retrieved and then sorted out according to an RFC; which gives us the highest priority IP from the list.

Follow the stack trace above and you should be able to see by yourself the source code where the "magic" happens (it's an interesting thing to do!).

With an IP address chosen, we can proceed.

note.: this process is **not** different for TCP.

> The book Computer Networking: A top-down approach has a great section about DNS.
>
> I'd **really** recommend you going through it to know more about.

I also wrote a blog post about writing something that is able to resolve `A` records from scratch using Go: Writing DNS messages from scratch using Go.

TCP Dialing vs UDP Dialing

Instead of using a regular `Dial` commonly used with TCP, for our UDP client, a different method was used: `DialUDP`.

The reason for that is that we can enforce the type of address passed, as well as receive a specialized connection: the "concrete type" `UDPConn` instead of the generic `Conn` interface.

Although both `Dial` and `DialUDP` might sound like the same (even when it comes to the syscalls used while talking to the kernel), they end up being pretty different concerning the network stack implementation.

For instance, we can check that both methods use `connect(2)` under the hood:

- TCP

```
// TCP - performs an actual `connect` under the hood,
// trying to establish an actual connection with the
// other side.
net.Dial("tcp", "1.1.1.1:53")

// strace -f -e trace=network ./main
//  [pid  4891] socket(
        AF_INET,
 -----> SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK,
        IPPROTO_IP) = 3
//  [pid  4891] connect(3, {sa_family=AF_INET, sin_port=htons(53), sin_addr=
...
```

- UDP

```
// UDP - calls `connect` just like TCP, but given that
// the arguments are different (it's not SOCK_STREAM),
// the semantics differ - it constrains the socket
```

```
// regarding to whom it might communicate with.
net.Dial("udp", "1.1.1.1:53")

// strace -f -e trace=network ./main
// [pid  5517] socket(
        AF_INET,
 -----> SOCK_DGRAM|SOCK_CLOEXEC|SOCK_NONBLOCK,
        IPPROTO_IP) = 3
// [pid  5517] connect(3, {sa_family=AF_INET, sin_port=htons(53), sin_addr=i
...
```

While they're pretty much the same, from the documentation we can see how they are semantically different depending on the way we configure the `socket` created via the `socket(2)` call that happens before `connect(2)` :

> *If the socket sockfd is of type **SOCK_DGRAM**, then addr is the address to which datagrams are sent by default, and the only address from which datagrams are received.*

> *If the socket is of type **SOCK_STREAM** or SOCK_SEQ-PACKET, this call attempts to make a connection to the socket that is bound to the address specified by addr.*

Should we be able to verify that with the TCP transport the `Dial` method would perform the act of really connecting to the other side? Sure!

```
./tools/funccount -p $(pidof main) 'tcp_*'
Tracing 316 functions for "tcp_*"... Hit Ctrl-C to end.
^C
FUNC                                    COUNT
tcp_small_queue_check.isra.28               1
tcp_current_mss                             1
tcp_schedule_loss_probe                     1
tcp_mss_to_mtu                              1
tcp_write_queue_purge                       1
tcp_write_xmit                              1
tcp_select_initial_window                   1
tcp_fastopen_defer_connect                  1
tcp_mtup_init                               1
tcp_v4_connect                              1
tcp_v4_init_sock                            1
tcp_rearm_rto.part.61                       1
tcp_close                                   1
```

```
tcp_connect                                     1
tcp_send_fin                                    1
tcp_rearm_rto                                   1
tcp_tso_segs                                    1
tcp_event_new_data_sent                         1
tcp_check_oom                                   1
tcp_clear_retrans                               1
tcp_init_xmit_timers                            1
tcp_init_sock                                   1
tcp_initialize_rcv_mss                          1
tcp_assign_congestion_control                   1
tcp_sync_mss                                    1
tcp_init_tso_segs                               1
tcp_stream_memory_free                          1
tcp_setsockopt                                  1
tcp_chrono_stop                                 2
tcp_rbtree_insert                               2
tcp_set_state                                   2
tcp_established_options                         2
tcp_transmit_skb                                2
tcp_v4_send_check                               2
tcp_rate_skb_sent                               2
tcp_options_write                               2
tcp_poll                                        2
tcp_release_cb                                  4
tcp_v4_md5_lookup                               4
tcp_md5_do_lookup                               4
```

In the case of UDP though, in theory, it merely takes care of marking the socket for reads and writes to the specified address.

Going through the same process that we did for TCP (going further from looking at the syscall interface), we can trace the underlying kernel methods used by both `DialUDP` and `Dial` to see how they differ:

```
./tools/funccount -p $(pidof main) 'udp_*'
Tracing 57 functions for "udp_*"... Hit Ctrl-C to end.
^C
FUNC                                        COUNT
udp_v4_rehash                                   1
udp_poll                                        1
udp_v4_get_port                                 1
udp_lib_close                                   1
udp_lib_lport_inuse                             1
udp_init_sock                                   1
udp_lib_unhash                                  1
udp_lib_rehash                                  1
udp_lib_get_port                                1
udp_destroy_sock                                1
```

Much… Much less.

If we go even further, try to explore what happens at each of these calls, we can notice how `connect(2)` in the case of TCP ends up really transmitting data (to establish perform the handshake, for instance):

```
PID     TID     COMM            FUNC
6747    6749    main            tcp_transmit_skb
        tcp_transmit_skb+0x1
        tcp_v4_connect+0x3f5
        __inet_stream_connect+0x238
        inet_stream_connect+0x3b
        SYSC_connect+0x9e
        sys_connect+0xe
        do_syscall_64+0x73
        entry_SYSCALL_64_after_hwframe+0x3d
```

While in the case of UDP, nothing is transmitted, just some set up is performed:

```
PID     TID     COMM            FUNC
6815    6817    main            ip4_datagram_connect
        ip4_datagram_connect+0x1 [kernel]
        SYSC_connect+0x9e [kernel]
        sys_connect+0xe [kernel]
        do_syscall_64+0x73 [kernel]
        entry_SYSCALL_64_after_hwframe+0x3d [kernel]
```

If you're not convinced yet that these two are **really** different (in the sense that the TCP one sends packets to establish the connection, while UDP doesn't), we can set up some triggers in the network stack to tell us whenever packets flow:

```
# By creating a rule that will only match
# packets destined at `1.1.1.1` and that
# match a specific protocol, we're able
# to see what happens at the time that
# `connect(2)` happens with a given protocol
# or another.
iptables \
        --table filter \
        --insert OUTPUT \
        --jump LOG \
        --protocol udp \
        --destination 1.1.1.1 \
        --log-prefix="[UDP] "

iptables \
        --table filter \
        --insert OUTPUT \
        --jump LOG \
```

```
        --protocol tcp \
        --destination 1.1.1.1 \
        --log-prefix="[TCP] "
```

Now, run `Dial` against a TCP target and `DialUDP` target and compare the differences.

You should only see `[TCP]` logs:

```
[46260.105662] [TCP] IN= OUT=enp0s3 DST=1.1.1.1 SYN URGP=0
[46260.120454] [TCP] IN= OUT=enp0s3 DST=1.1.1.1 ACK URGP=0
[46260.120718] [TCP] IN= OUT=enp0s3 DST=1.1.1.1 ACK FIN URGP=0
[46260.150452] [TCP] IN= OUT=enp0s3 DST=1.1.1.1 ACK URGP=0
```

> If you're not familiar with the inner workings of `dmesg`, check out my other blog post - dmesg under the hood.
>
> By the way, The Linux Programming Interface is a great book to know more about sockets and other related topics!

Writing to a UDP "connection"

With our UDP socket properly created and configured for a specific address, we're now on time to go through the "write" path - when we actually take some data and write to the `UDPConn` object received from `net.DialUDP`.

A sample program that just sends a little bit of data to a given UDP server would be as follow:

```
// Perform the address resolution and also
// specialize the socket to only be able
// to read and write to and from such
// resolved address.
conn, err := net.Dial("udp", *addr)
if err != nil {
        panic(err)
}
defer conn.Close()

// Call the `Write()` method of the implementor
```

```
// of the `io.Writer` interface.
n, err = fmt.Fprintf(conn, "something")
```

Given that `conn` returned by `Dial` implements the `io.Writer` interface, we can make use of something like `fmt.Fprintf` (that takes an `io.Writer` as its first argument) as let it call `Write()` with the message we pass to it.

> If interfaces and other Golang concepts are not clear for you yet, make sure you check Kernighan's book: The Go Programming Language.
>
> *yeah, from the guy who wrote The C Programming Language with Dennis Ritchie.*

Under the hood, `UDPConn` implements the `Write()` method from the `io.Writer` interface by being a composition of `conn`, a struct that implements the most basic methods regarding writing to and reading from a given file descriptor:

```
type conn struct {
	fd *netFD
}

// Write implements the Conn Write method.
func (c *conn) Write(b []byte) (int, error) {
	if !c.ok() {
		return 0, syscall.EINVAL
	}
	n, err := c.fd.Write(b)
	if err != nil {
		err = &OpError{Op: "write", Net: c.fd.net, Source: c.fd.ladd
	}
	return n, err
}

// UDPConn is the implementation of the Conn
// and PacketConn interfaces for UDP network
// connections.
type UDPConn struct {
	conn
}
```

Now, knowing that in the end `fmt.Fprintf(conn, "something")` ends up in a `write(2)` to a file descriptor (the UDP socket), we can investigate even further and see how does the kernel path look for such `write(2)` call:

```
PID     TID     COMM            FUNC
14502   14502   write.out       ip_send_skb
        ip_send_skb+0x1
        udp_sendmsg+0x3b5
        inet_sendmsg+0x2e
        sock_sendmsg+0x3e
        sock_write_iter+0x8c
        new_sync_write+0xe7
        __vfs_write+0x29
        vfs_write+0xb1
        sys_write+0x55
        do_syscall_64+0x73
        entry_SYSCALL_64_after_hwframe+0x3d
```

At that point, the packet should be on its way to the other side of the communication channel.

Receiving from a UDP "connection" in a client

The act of receiving from `UDPConn` can be seen as pretty much the same as the "write path", except that at this time, a buffer is supplied (so that it can get filled with the contents that arrive), and we don't really know how long we have to wait for the content to arrive.

For instance, we could have the following code path for reading from a known address:

```go
buf := make([]byte, *bufSize)
_, err = conn.Read(buf)
```

This would turn into a `read(2)` syscall under the hood, which would then go through `vfs` and turn into a `read` from a socket:

```
22313   22313   read            __skb_recv_udp
        __skb_recv_udp+0x1
        inet_recvmsg+0x51
        sock_recvmsg+0x43
        sock_read_iter+0x90
        new_sync_read+0xe4
        __vfs_read+0x29
        vfs_read+0x8e
        sys_read+0x55
        do_syscall_64+0x73
```

```
    entry_SYSCALL_64_after_hwframe+0x3d
```

Something important to remember is that when it comes to reading from the socket, that's going to be a blocking operation.

Given that a message might never return from such socket, we can get stuck waiting forever.

To avoid such situation, we can set a reading deadline that would kill the whole thing in case we wait for too long:

```
buf := make([]byte, *bufSize)

// Sets the read deadline for now + 15seconds.
// If you plan to read from the same connection again,
// make sure you expand the deadline before reading
// it.
conn.SetReadDeadline(time.Now().Add(15 * time.Second))
_, err = conn.Read(buf)
```

Now, In case the other end takes too long to answer:

```
read udp 10.0.2.15:41745->1.1.1.1:53: i/o timeout
```

Receiving from a UDP "connection" in a server

While that's great for the `client` (we know whom we're reading from), it's not for a server.

The reason why is that at the server side, we don't know who we're reading from (the address is unknown).

Differently from the case of a TCP server where we have `accept(2)` which returns to the server implementor the connection that the server can write to, in the case of UDP, there's no such thing as a "connection to write to". There's only a "whom to write to", that can be retrieved by inspecting the packet that arrived.

```
WITH READ

  "Hmmm, let me write something to
   my buddy at 1.1.1.1:53"

   client --.
            |
            | client: n, err := udpConn.Write(buf)
            | server: n, err := udpConn.Read(buf)
            |
            *---> server
                  "Oh, somebody wrote me something!
                   I'd like to write back to him/her,
                   but, what's his/her address?

                   I don't have a connection... I need
                   an address to write to! I can't to
                   a thing now!"


WITH READFROM

   client --.
            |
            | client: n, err := udpConn.Write(buf)
            | server: n, address, err := udpConn.Read(buf)
            |
            *---> server
                  "Oh, looking at the packet, I can
                   see that my friend Jane wrote to me,
                   I can see that from `address`!

                   Let me answer her back!"
```

For that reason, on the server, we need the specialized connection: `UDPConn` .

Such specialized connection is able of giving us `ReadFrom` , a method that instead of just reading from a file descriptor and adding the contents to a buffer, it also inspects the headers of the packet and gives us information about who sent the package.

Its usage looks like this:

```
buffer := make([]byte, 1024)

// Given a buffer that is meant to hold the
// contents from the messages arriving at the
// socket that `udpConn` wraps, it blocks until
// messages arrive.
//
// For each message arriving, `ReadFrom` unwraps
```

```
// the message, getting information about the
// sender from the protocol headers and then
// fills the buffer with the data.
n, addr, err := udpConn.ReadFrom(buffer)
if err != nil {
        panic(err)
}
```

An interesting way of trying to understand how things work under the hood is looking at the plan9 implementation (`net/udpsock_plan9.go`).

Here's how it looks (with comments of my own):

```
func (c *UDPConn) readFrom(b []byte) (n int, addr *UDPAddr, err error) {
        // creates a buffer a little bit bigger than
        // the one we provided (to account for the header of
        // the UDP headers)
        buf := make([]byte, udpHeaderSize+len(b))

        // reads from the underlying file descriptor (this might
        // block).
        m, err := c.fd.Read(buf)
        if err != nil {
                return 0, nil, err
        }
        if m < udpHeaderSize {
                return 0, nil, errors.New("short read reading UDP header")
        }

        // strips out the parts that were not readen
        buf = buf[:m]

        // interprets the UDP header
        h, buf := unmarshalUDPHeader(buf)

        // copies the data back to our supplied buffer
        // so that we only receive the data, not the header.
        n = copy(b, buf)
        return n, &UDPAddr{IP: h.raddr, Port: int(h.rport)}, nil
}
```

Naturally, under Linux, that's not the path that `readFrom` takes. It uses `recvfrom` which does the whole "UDP header interpretation" under the hood, but the idea is the same (except that with `plan9` it's all done in userspace).

To verify the fact that under Linux we're using `recvfrom` , we trace `UDPConn.ReadFrom` down (you can use `delve` for that):

```
0  0x00000000004805b8 in syscall.recvfrom
   at /usr/local/go/src/syscall/zsyscall_linux_amd64.go:1641
1  0x000000000047e84f in syscall.Recvfrom
   at /usr/local/go/src/syscall/syscall_unix.go:262
2  0x0000000000494281 in internal/poll.(*FD).ReadFrom
   at /usr/local/go/src/internal/poll/fd_unix.go:215
3  0x00000000004f5f4e in net.(*netFD).readFrom
   at /usr/local/go/src/net/fd_unix.go:208
4  0x0000000000516ab1 in net.(*UDPConn).readFrom
   at /usr/local/go/src/net/udpsock_posix.go:47
5  0x00000000005150a4 in net.(*UDPConn).ReadFrom
   at /usr/local/go/src/net/udpsock.go:121
6  0x0000000000526bbf in main.server.func1
   at ./main.go:65
7  0x000000000045e1d1 in runtime.goexit
   at /usr/local/go/src/runtime/asm_amd64.s:1333
```

At the kernel level, we can also check what are the methods involved:

```
24167   24167   go-sample-udp   __skb_recv_udp
        __skb_recv_udp+0x1
        inet_recvmsg+0x51
        sock_recvmsg+0x43
        SYSC_recvfrom+0xe4
        sys_recvfrom+0xe
        do_syscall_64+0x73
        entry_SYSCALL_64_after_hwframe+0x3d
```

A UDP Server in Go

Now, going to the server-side implementation, here's how the code would look like (heavily commented):

```
// maxBufferSize specifies the size of the buffers that
// are used to temporarily hold data from the UDP packets
// that we receive.
const maxBufferSize = 1024

// server wraps all the UDP echo server functionality.
// ps.: the server is capable of answering to a single
// client at a time.
func server(ctx context.Context, address string) (err error) {
        // ListenPacket provides us a wrapper around ListenUDP so that
        // we don't need to call `net.ResolveUDPAddr` and then subsequential
        // perform a `ListenUDP` with the UDP address.
        //
        // The returned value (PacketConn) is pretty much the same as the on
```

```
// from ListenUDP (UDPConn) - the only difference is that `Packet*`
// methods and interfaces are more broad, also covering `ip`.
pc, err := net.ListenPacket("udp", address)
if err != nil {
        return
}

// `Close`ing the packet "connection" means cleaning the data struct
// allocated for holding information about the listening socket.
defer pc.Close()

doneChan := make(chan error, 1)
buffer := make([]byte, maxBufferSize)

// Given that waiting for packets to arrive is blocking by nature an
// to be able of canceling such action if desired, we do that in a s
// go routine.
go func() {
        for {
                // By reading from the connection into the buffer, w
                // new content in the socket that we're listening fo
                //
                // Whenever new packets arrive, `buffer` gets filled
                // the execution.
                //
                // note.: `buffer` is not being reset between runs.
                //        It's expected that only `n` reads are read
                //        inspecting its contents.
                n, addr, err := pc.ReadFrom(buffer)
                if err != nil {
                        doneChan <- err
                        return
                }

                fmt.Printf("packet-received: bytes=%d from=%s\n",
                        n, addr.String())

                // Setting a deadline for the `write` operation allo
                // for longer than a specific timeout.
                //
                // In the case of a write operation, that'd mean wai
                // queue to be freed enough so that we are able to p
                deadline := time.Now().Add(*timeout)
                err = pc.SetWriteDeadline(deadline)
                if err != nil {
                        doneChan <- err
                        return
                }

                // Write the packet's contents back to the client.
                n, err = pc.WriteTo(buffer[:n], addr)
                if err != nil {
                        doneChan <- err
                        return
                }

                fmt.Printf("packet-written: bytes=%d to=%s\n", n, ad
        }
}()

select {
case <-ctx.Done():
        fmt.Println("cancelled")
```

```
                err = ctx.Err()
        case err = <-doneChan:
        }

        return
}
```

As you might have noticed, it's not all that different from the client! The reason why is that not having an actual connection involved (like in TCP), both client and servers end up going through the same path: preparing a socket to read and write from and to, then checking the content from the packets and doing the same thing over and over again.

Closing thoughts

It was great to go through this exploration, checking what's going on behind the scenes in the Go source code (very well written, by the way), as well as in the Kernel.

I think I finally got a great workflow when it comes to debugging with Delve and verifying the Kernel functions with bcc, maybe I'll write about that soon - let me know if that'd be interesting!

If you have any questions, please let me know! I'm @cirowrc on Twitter, and I'd love to receive your feedback.

Have a good one!

Recommended articles

If you've gotten some knowledge from this article, these are some others that you might take advatange of as well!

- Implementing a TCP server in C
- Sending files via gRPC
- A Raspberry PI Concourse Worker
- How Linux creates sockets and counts them
- Writing DNS messages from scratch using Go

Stay in touch!

From time to time I'll deliver some content to you.

The emails are not automatic - it's all about things I thought were worth sharing that I'd personally like to receive.

JOIN THE GROUP

If you're into Twitter, reach me at @cirowrc.

About Tags Advertise Twitter GitHub LinkedIn