Building a net.Conn type from the ground up

Matt Layher

GopherCon UK, August 23rd, 2019

Matt Layher

- GitHub, Twitter: **@mdlayher**
- mdlayher.com
- github.com/mdlayher/talks


fastly
R

Package net

Package net

- A fundamental building block for network clients and servers in Go
- TCP, UDP, IP, and UNIX sockets
  - **`net.Conn`**: “a generic stream-oriented network connection”
  - **`net.Listener`**: “a generic network listener for stream-oriented protocols”

What is a “stream-oriented” connection?

- Data flows as a reliable, in-order, stream between network sockets
- TCP sockets are stream-oriented: **`net.TCPConn`**, **`net.TCPListener`**
    - HTTP, HTTPS, and SSH run on top of TCP

net.Conn usage

```
// Produces net.Conn of type *net.TCPConn.
conn, err := net.Dial("tcp", "golang.org:80")
if err != nil {
    // handle error
}

fmt.Fprintf(conn, "GET / HTTP/1.0\r\n\r\n")
status, err := bufio.NewReader(conn).ReadString('\n')
// ...
```

https://golang.org/pkg/net/#pkg-overview

net.Listener usage

```
// Produces net.Listener of type *net.TCPListener.
ln, err := net.Listen("tcp", ":8080")
if err != nil {
    // handle error
}
for {
    conn, err := ln.Accept()
    if err != nil {
        // handle error
    }
    go handleConnection(conn)
}
```

https://golang.org/pkg/net/#pkg-overview

What about other socket types?

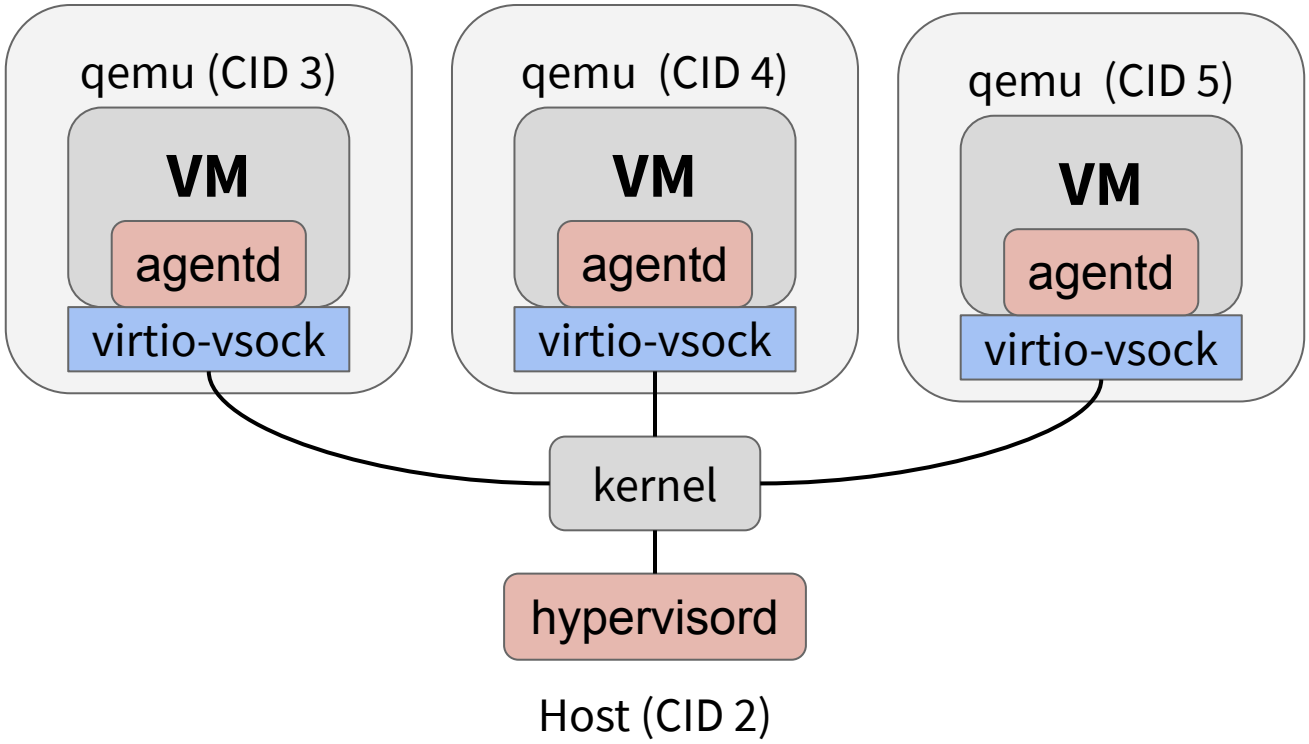
- TCP and UDP aren’t the only options for socket communications
- `net.Conn` and `net.Listener` are interfaces
    - **We can implement our own socket types!**

AF_VSOCK

AF_VSOCK

- Linux virtual machine (VM) sockets
- Bidirectional communication between a hypervisor and its VMs
- Addresses contain a context ID (CID) and port
    - Akin to an IP address and port in TCP/IP

AF_VSOCK diagram



AF_VSOCK in Go

- github.com/mdlayher/vsock
  - Provides package `net` types that use VM sockets in Go
- Used in some interesting open source projects:
  - github.com/firecracker-microvm/firecracker-containerd
  - github.com/kata-containers/agent

net.Conn usage with *vsock.Conn

```
// Produces net.Conn of type *vsock.Conn.
conn, err := vsock.Dial(vsock.Host, 8080)
if err != nil {
    // handle error
}

fmt.Fprintf(conn, "GET / HTTP/1.0\r\n\r\n")
status, err := bufio.NewReader(conn).ReadString('\n')
// ...
```

net.Listener usage with *vsock.Listener

```
// Produces net.Listener of type *vsock.Listener.
ln, err := vsock.Listen(8080)
if err != nil {
    // handle error
}
for {
    conn, err := ln.Accept()
    if err != nil {
        // handle error
    }
    go handleConnection(conn)
}
```

How do we implement a new socket type in Go?

The BSD sockets API

- System calls used to manipulate network sockets on UNIX-like systems
  - `socket(2)`, `send(2)`, `recv(2)`, etc.
  - **Package `net` uses these system calls internally**
- Beej's Guide to Network Programming
  - An awesome reference for learning how to use BSD sockets
  - Examples use C, but they apply to Go as well

Mapping net APIs to BSD sockets

net.Conn

- socket(2)
- connect(2)
- getsockname(2)
- send(2)
- recv(2)
- close(2)

net.Listener

- socket(2)
- bind(2)
- listen(2)
- getsockname(2)
- accept4(2)
- close(2)

Building a `net.Conn` type

net.Addr and net.Conn interfaces

```
package net

// Conn is a generic stream-oriented network connection.
type Conn interface {
        Read(b []byte) (n int, err error)
        Write(b []byte) (n int, err error)
        Close() error
        LocalAddr() Addr  // A network endpoint address.
        RemoteAddr() Addr //
        SetDeadline(t time.Time) error
        SetReadDeadline(t time.Time) error
        SetWriteDeadline(t time.Time) error
}
```

net.Addr and net.Conn implementations

```
package vsock

// An Addr is the address of a VM sockets endpoint.
type Addr struct {
    ContextID uint32
    Port      uint32
}

// A Conn is a VM sockets implementation of a net.Conn.
type Conn struct {
    // Operating system-specific implementation.
    c *conn
}
```

vsock.Dial

- A constructor which dials a VM sockets connection
    - System calls from `golang.org/x/sys/unix`

```
// Dial dials a net.Conn to a VM sockets server.
func Dial(contextID, port uint32) (*Conn, error) {
    // Operating system-specific code: // +build linux
    return dial(contextID, port)
}
```

socket(2)

- Open a connection-oriented VM socket which closes on fork/exec

```
// dial is the entry point for Dial on Linux.
func dial(cid, port uint32) (*Conn, error) {
    fd, err := unix.Socket(
        unix.AF_VSOCK,
        unix.SOCK_STREAM|unix.SOCK_CLOEXEC,
        0,
    )
    if err != nil {
            return nil, err
    }
```

connect(2)

- Connect to the remote address using `unix.SockaddrVM`

```
rsa := &unix.SockaddrVM{
    CID:  cid,
    Port: port,
}

if err := unix.Connect(fd, rsa); err != nil {
    return nil, err
}
```

getsockname(2)

- Get the address of our local socket, convert to `*vsock.Addr`

```
local, err := unix.Getsockname(fd)
if err != nil {
    return nil, err
}

lsa := local.(*unix.SockaddrVM)
local := &Addr{
    ContextID: lsa.CID,
    Port:      lsa.Port,
}
```

os.NewFile

- Use the methods of *os.File to handle Read/Write/Close

```
remote := &Addr{
    ContextID: cid,
    Port:      port,
}

return &conn{
    file:   os.NewFile(fd, "vsock"),
    local:  local,
    remote: remote,
}, nil
```

The initial conn type

- Use the methods of `*os.File` to handle `Read`/`Write`/`Close`
    - Deadlines are not yet implemented

```
// A conn is the net.Conn implementation for VM sockets.
type conn struct {
     file          *os.File
     local, remote *Addr
}
// Implement net.Conn for type conn.
func (c *conn) LocalAddr() net.Addr                { return c.local }
func (c *conn) RemoteAddr() net.Addr               { return c.remote }
func (c *conn) SetDeadline(t time.Time) error      { return errNotImplemented }
func (c *conn) SetReadDeadline(t time.Time) error  { return errNotImplemented }
func (c *conn) SetWriteDeadline(t time.Time) error { return errNotImplemented }
func (c *conn) Read(b []byte) (n int, err error)   { return c.file.Read(b) }
func (c *conn) Write(b []byte) (n int, err error)  { return c.file.Write(b) }
func (c *conn) Close() error                       { return c.file.Close() }
```

Building a `net.Listener` type

net.Listener interface

```
package net

// A Listener is a network listener for stream-oriented protocols.
type Listener interface {
    // Accept waits for and returns the next connection to the
    // listener.
    Accept() (Conn, error)
    // Close closes the listener. Any blocked Accept operations
    // will be unblocked and return errors.
    Close() error
    // Addr returns the listener's network address.
    Addr() Addr
}
```

net.Listener and vsock.Listen

```
package vsock
// A Listener is a VM sockets implementation of a net.Listener.
type Listener struct {
    l *listener
}

// Listen opens a net.Listener for VM sockets connections.
func Listen(port uint32) (*Listener, error) {
    cid, err := ContextID()
    if err != nil {
        return nil, err
    }
    return listen(cid, port)
}
```

vsock.ContextID

- Use ioctl(2) to fetch the VM sockets context ID of the current system

```
// ContextID retrieves the local context ID for this system.
func ContextID() (uint32, error) {
    f, err := os.Open("/dev/vsock")
    if err != nil {
        return 0, err
    }
    defer f.Close()

    return unix.IoctlGetUint32(int(f.Fd()),
        unix.IOCTL_VM_SOCKETS_GET_LOCAL_CID,
    )
}
```

socket(2)

- Open a connection-oriented VM socket which closes on fork/exec

```
// listen is the entry point for Listen on Linux.
func listen(cid, port uint32) (*Conn, error) {
    fd, err := unix.Socket(
        unix.AF_VSOCK,
        unix.SOCK_STREAM|unix.SOCK_CLOEXEC,
        0,
    )
    if err != nil {
            return nil, err
    }
```

bind(2)

- Bind the socket to the host's context ID and specified port

```
sa := &unix.SockaddrVM{
    CID:  cid,
    Port: port,
}
if err := unix.Bind(fd, sa); err != nil {
    return nil, err
}
```

listen(2)

- Start listening for connections on the socket

```
if err := unix.Listen(fd, unix.SOMAXCONN); err != nil {
    return nil, err
}
```

getsockname(2)

- Get the address of our local socket, convert to `*vsock.Addr`

```
lsa, err := unix.Getsockname(fd)
if err != nil {
    return nil, err
}
return &listener{
    fd:   fd,
    addr: &Addr{
        ContextID: lsa.(*unix.SockaddrVM).CID,
        Port:      lsa.(*unix.SockaddrVM).Port,
    },
}, nil
```

accept4(2)

- Accept new connections and return net.Conn implementations

```
func (l *listener) Accept() (net.Conn, error) {
    fd, sa, err := unix.Accept4(l.fd, unix.SOCK_CLOEXEC)
    if err != nil {
        return nil, err
    }

    // ...
```

accept4(2)

- Accept new connections and return net.Conn implementations

```
    // ...

    return &conn{
        file:   os.NewFile(fd, "vsock"),
        local:  l.addr,
        remote: &Addr{
            ContextID: sa.(*unix.SockaddrVM).CID,
            Port:      sa.(*unix.SockaddrVM).Port,
        },
    }, nil
}
```

The initial `listener` type

- Perform raw system calls to accept new connections

```
// A listener is the net.Listener implementation for VM sockets.
type listener struct {
    fd   int
    addr *Addr
}
// Implement net.Listener for type listener.
func (l *listener) Addr() net.Addr { return l.addr }
func (l *listener) Close() error   { return unix.Close(l.fd) }

func (l *listener) Accept() (net.Conn, error) {
    // unix.Accept4 ...
}
```

Problems with our implementations

`net.Conn` subtleties

- “Multiple goroutines may invoke methods on a Conn simultaneously.”
  - `Read` and `Write` can be unblocked by `Close`
  - Deadline methods apply to all future and pending I/O
- Must satisfy `net.Error` for certain error situations
  - `net.OpError` is the de-facto network error type

`net.Listener` subtleties

- “Multiple goroutines may invoke methods on a Listener simultaneously.”
  - `Accept` can be unblocked by `Close`
  - Some standard library implementations support a SetDeadline method
- Must satisfy `net.Error` for certain error situations
  - `net.OpError` is the de-facto network error type

The runtime network poller

Non-blocking I/O

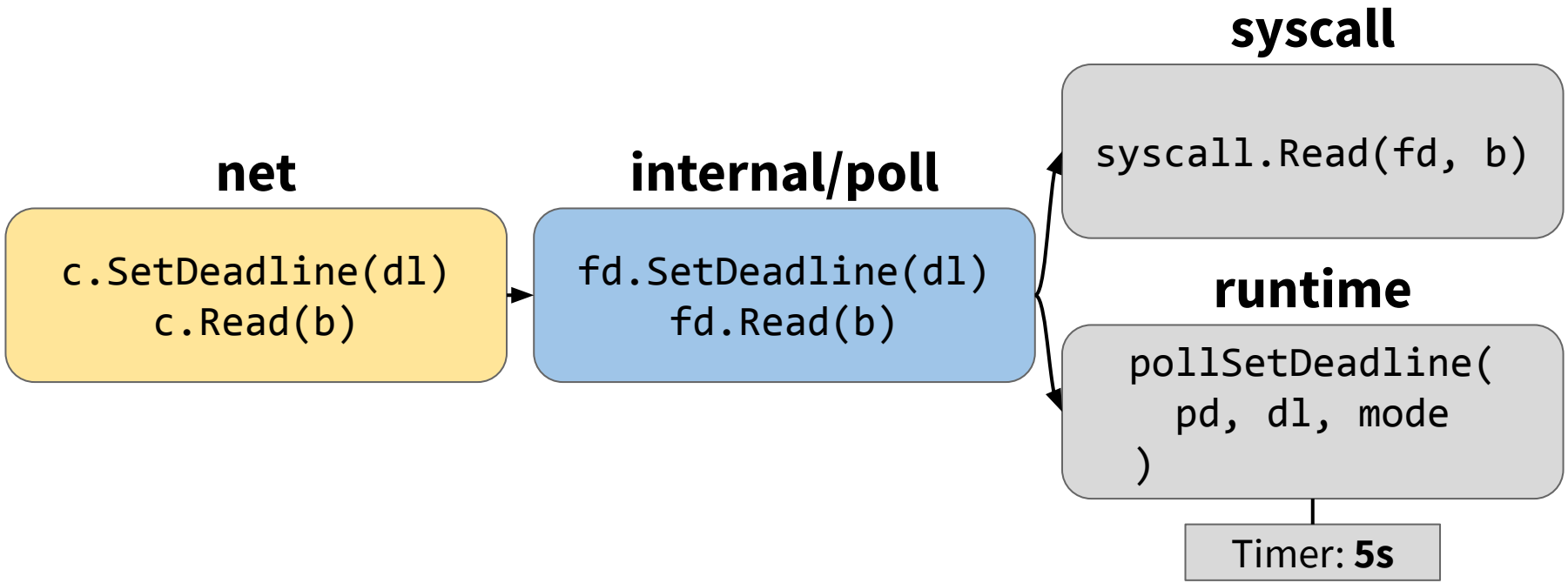
- System calls **which would block** return immediately with **`EAGAIN`**
  - **Problem:** potentially inefficient busy loops to check readiness
  - **Solution:** I/O event notification mechanisms like Linux's `epoll(7)`
- `unix.SetNonblock` sets a file descriptor's blocking mode

The runtime network poller

- Implemented in `runtime` and `internal/poll`
    - Uses OS-specific event notification mechanisms: epoll, kqueue, IOCP
- One of Go’s best features, and something we all take for granted

How does the runtime network poller work?

- A goroutine uses the `net.Conn` `Read` method with an associated deadline



How does the runtime network poller work?

- The `internal/poll.FD Read` method attempts to read using non-blocking I/O

```
for {
    n, err := syscall.Read(fd.Sysfd, p)
    if err != nil {
        if err == syscall.EAGAIN && fd.pd.pollable() {
            if err = fd.pd.waitRead(fd.isFile); err == nil {
                continue
            }
        }
    }
    return n, fd.eofError(n, err)
}
```

How does the runtime network poller work?

- If `syscall.Read` completed without `syscall.EAGAIN`, **return to the caller**

```
for {
    n, err := syscall.Read(fd.Sysfd, p)
    if err != nil {
        if err == syscall.EAGAIN && fd.pd.pollable() {
            if err = fd.pd.waitRead(fd.isFile); err == nil {
                continue
            }
        }
    }
    return n, fd.eofError(n, err)
}
```

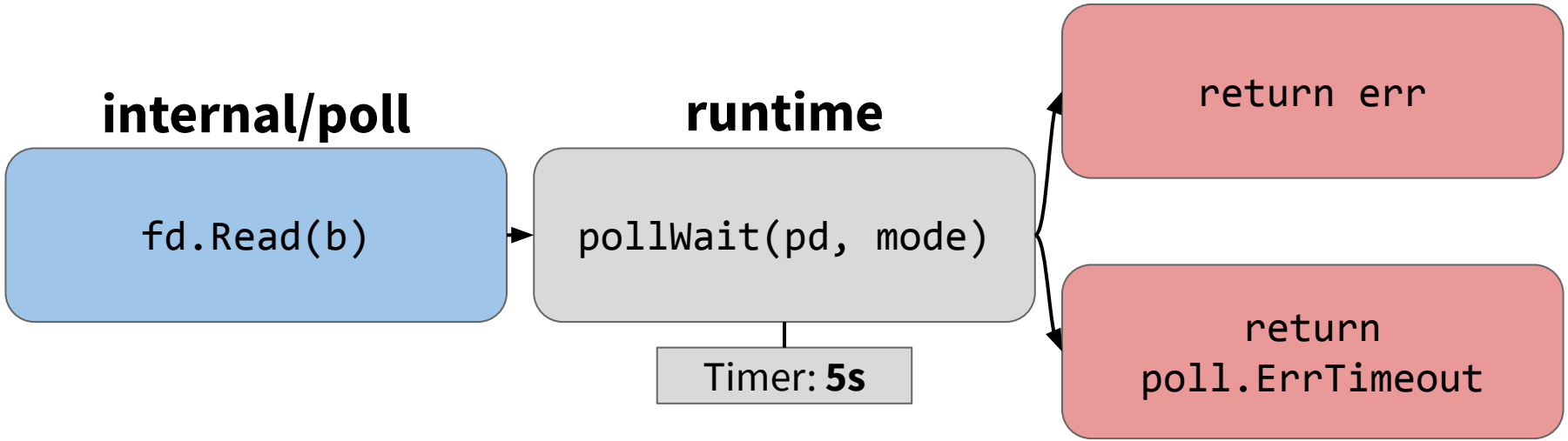
How does the runtime network poller work?

- If `syscall.Read` returned `syscall.EAGAIN`, **wait for readiness**

```
for {
    n, err := syscall.Read(fd.Sysfd, p)
    if err != nil {
        if err == syscall.EAGAIN && fd.pd.pollable() {
            if err = fd.pd.waitRead(fd.isFile); err == nil {
                continue
            }
        }
    }
    return n, fd.eofError(n, err)
}
```

How does the runtime network poller work?

- Control returns to the caller when **I/O completes** or the **timer expires**



The runtime network poller in summary

- Enables **non-blocking I/O** that *appears* to be **blocking I/O**
  - System calls **block one goroutine** instead of an **entire OS thread**
- **Makes concurrent calls to `net.Conn` types possible**
  - Built-in `net.Conn` types automatically use the runtime network poller
  - What about our custom type…?

Accessing the runtime network poller

- It took approximately 3.5 years for all the necessary pieces to fall into place:
  - April 23, 2015: golang.org/issue/10565
  - Go 1.9: new `syscall.Conn` and `syscall.RawConn` interfaces
  - Go 1.10: `os.File` now has `SetDeadline` family of methods
  - Go 1.11: `os.NewFile` registers non-blocking files with the network poller
  - Go 1.12: `os.File` now has `SyscallConn` method

syscall.RawConn

- Provides raw file descriptor control/read/write methods
    - Enables raw system calls on high-level types
    - Can indicate read/write completion to the runtime network poller

```
// A RawConn is a raw network connection.
type RawConn interface {
    Control(f func(fd uintptr)) error
    Read(f func(fd uintptr) (done bool)) error
    Write(f func(fd uintptr) (done bool)) error
}
```

Using vsock.Conn with the runtime network poller

os.NewFile with a non-blocking file

- Set connection file to non-blocking and register with the runtime network poller

```
if err := unix.SetNonblock(fd, true); err != nil {
    return nil, err
}

return &conn{
    // Now registered with the runtime network poller.
    file:   os.NewFile(fd, "vsock"),
    local:  local,
    remote: remote,
}, nil
```

The updated conn type

- Use the methods of `*os.File` to handle almost all methods

```
// A conn is the net.Conn implementation for VM sockets.
type conn struct {
     file          *os.File
     local, remote *Addr
}

// Implement net.Conn for type conn.
func (c *conn) LocalAddr() net.Addr                { return c.localAddr }
func (c *conn) RemoteAddr() net.Addr               { return c.remoteAddr }
func (c *conn) SetDeadline(t time.Time) error      { return c.file.SetDeadline(t) }
func (c *conn) SetReadDeadline(t time.Time) error  { return c.file.SetReadDeadline(t) }
func (c *conn) SetWriteDeadline(t time.Time) error { return c.file.SetWriteDeadline(t) }
func (c *conn) Read(b []byte) (n int, err error)   { return c.file.Read(b) }
func (c *conn) Write(b []byte) (n int, err error)  { return c.file.Write(b) }
func (c *conn) Close() error                       { return c.file.Close() }
```

Testing net.Conn compliance

- x/net/nettest can be used to check for compliance with net.Conn behaviors

```
func TestIntegrationNettestTestConn(t *testing.T) {
    // Create a pair of *vsock.Conns pointed at each other.
    nettest.TestConn(t, makeLocalPipe(
        func() (net.Listener, error) { return vsock.Listen(0) },
        func(addr net.Addr) (net.Conn, error) {
            a := addr.(*vsock.Addr)
            return vsock.Dial(a.ContextID, a.Port)
        },
    ))
}
```

net.Error compliance

- x/net/nettest and callers need to check for temporary/timeout errors
    - This is non-trivial, but achievable through trial and error

```
// Read implements the net.Conn Read method.
func (c *Conn) Read(b []byte) (int, error) {
    n, err := c.fd.Read(b)
    if err != nil {
        // Wrap all errors with *net.OpError.
        return n, c.opError(opRead, err)
    }
    return n, nil
}
```

Using `vsock.Listener` with the runtime network poller

`os.NewFile` with a non-blocking file

- Create a non-blocking file descriptor and register with the runtime network poller

```
if err := unix.SetNonblock(fd, true); err != nil {
    return nil, err
}

return &listener{
    // Now registered with the runtime network poller.
    file:   os.NewFile(fd, "vsock"),
    local:  local,
}, nil
```

Non-blocking accept(4)

- Use *os.File's SyscallConn method to get access to the raw file descriptor

```
func (l *listener) accept(flags int) (int, unix.Sockaddr, error) {
    // Note: only available in Go 1.12+!
    rc, err := l.file.SyscallConn()
    if err != nil {
        return 0, nil, err
    }

    // ...
```

Non-blocking accept(4)

- Set up a raw read and return any necessary data by passing it to the closure.

```
var (
    newFD int
    sa    unix.Sockaddr
)
doErr := rc.Read(func(fd uintptr) bool {
    // ...
})
if doErr != nil {
    return 0, nil, doErr
}
return newFD, sa, err
```

Non-blocking accept(4)

- Indicate completion to the runtime network poller by returning `true` or `false`

```
doErr := rc.Read(func(fd uintptr) bool {
    newFD, sa, err = unix.Accept4(int(fd), flags)
    switch err {
    case unix.EAGAIN, unix.ECONNABORTED:
        // Return false to let the poller wait for readiness.
        return false
    default:
        // No error or unrecognized error, Read completed.
        return true
    }
})
```

Testing `net.Listener` compliance

- `x/net/nettest` API has an open CL, but it isn't merged yet
  - https://go-review.googlesource.com/c/net/+/123056
- The same caveats about implementing `net.Error` also apply

Using `net.Conn` and `net.Listener`

HTTP over vsock

vsock HTTP server

- HTTP normally runs over TCP, but you can use any `net.Listener`

```
// Listen for incoming connections over vsock.
l, err := vsock.Listen(8080)
if err != nil {
    return err
}
defer l.Close()

// And serve files from the current directory over HTTP!
err = http.Serve(l, http.FileServer(http.Dir(".")))
if err != nil {
    return err
}
```

vsock HTTP client

- `http.Transport` can dial and use an arbitrary `net.Conn`

```
t := &http.Transport{
    Dial: func(_, addr string) (net.Conn, error) {
        // Address in CID:port format.
        host, sport, err := net.SplitHostPort(addr)
        if err != nil {
            return nil, err
        }
        cid, err := strconv.Atoi(host)
        if err != nil {
            return nil, err
        }
        // ...
```

vsock HTTP client

- `http.Transport` can dial and use an arbitrary `net.Conn`

```
        // ...
        port, err := strconv.Atoi(sport)
        if err != nil {
            return nil, err
        }
        return vsock.Dial(uint32(cid), uint32(port))
    },
}

// Client now uses vsock as its transport.
c := &http.Client{Transport: t}
```

```
server $ ./vsockhttp -s 8080

client $ ./vsockhttp http://2:8080/hello.txt
HTTP/1.1 200 OK
Content-Length: 21
Accept-Ranges: bytes
Content-Type: text/plain; charset=utf-8
Date: Wed, 31 Jul 2019 16:56:36 GMT
Last-Modified: Tue, 30 Jul 2019 19:03:09 GMT

Hello, GopherCon UK!
```

gRPC over vsock

Using the Go gRPC “helloworld” demo

https://github.com/grpc/grpc-go/tree/master/examples

vsock gRPC server

- Make a server type that implements the generated gRPC interface.

```
// helloServer implements the pb.GreeterServer interface.
type helloServer struct{}

func (*helloServer) SayHello(
    ctx context.Context,
    in *pb.HelloRequest,
) (*pb.HelloReply, error) {
    return &pb.HelloReply{
        Message: fmt.Sprintf("Hello, %s!", in.Name),
    }, nil
}
```

vsock gRPC server

- Serve `GreeterServer` RPCs using `vsock.Listener` (which is a `net.Listener`)

```
l, err := vsock.Listen(8080)
if err != nil {
    return err
}
defer l.Close()

// Register our type to handle GreeterServer RPCs.
s := grpc.NewServer()
pb.RegisterGreeterServer(s, &helloServer{})
if err := s.Serve(l); err != nil {
    return err
}
```

vsock gRPC client

- Dial a gRPC connection that uses vsock as its transport instead of TCP.

```
cc, err := grpc.Dial(addr,
    // You should use TLS!! ...but this is just a demo.
    grpc.WithInsecure(),
    grpc.WithDialer(func(addr string, _ time.Duration) (net.Conn,
error) {
        // ... same process to convert CID:port into integers.
        return vsock.Dial(uint32(cid), uint32(port))
    }),
)
if err != nil {
    return err
}
```

vsock gRPC client

- Use the gRPC connection to invoke the `SayHello` RPC and print the result.

```
c := pb.NewGreeterClient(cc)
r, err := c.SayHello(context.Background(), &pb.HelloRequest{
    Name: "GopherCon UK",
})
if err != nil {
    return err
}

fmt.Println(r.Message)
```

```
server $ ./vsockgrpc -s 8080

client $ ./vsockgrpc 2:8080
Hello, GopherCon UK!
```

Summary

The possibilities are endless

- Implementing standard interfaces provides a huge amount of flexibility
- In the case of vsock, you could imagine:
  - A VM guest agent that use gRPC over vsock to communicate with a hypervisor
  - A VM with no network interfaces that uses an HTTP proxy over vsock

Resources and thanks

- Blog: Linux VM sockets in Go (from 2017, may be slightly out of date)
  - mdlayher.com/blog/linux-vm-sockets-in-go
- Source code:
  - github.com/mdlayher/vsock
- Thanks to @acln and many other folks on #networking on Gophers Slack

Thanks!

Matt Layher

mdlayher.com
github.com/mdlayher
twitter.com/mdlayher