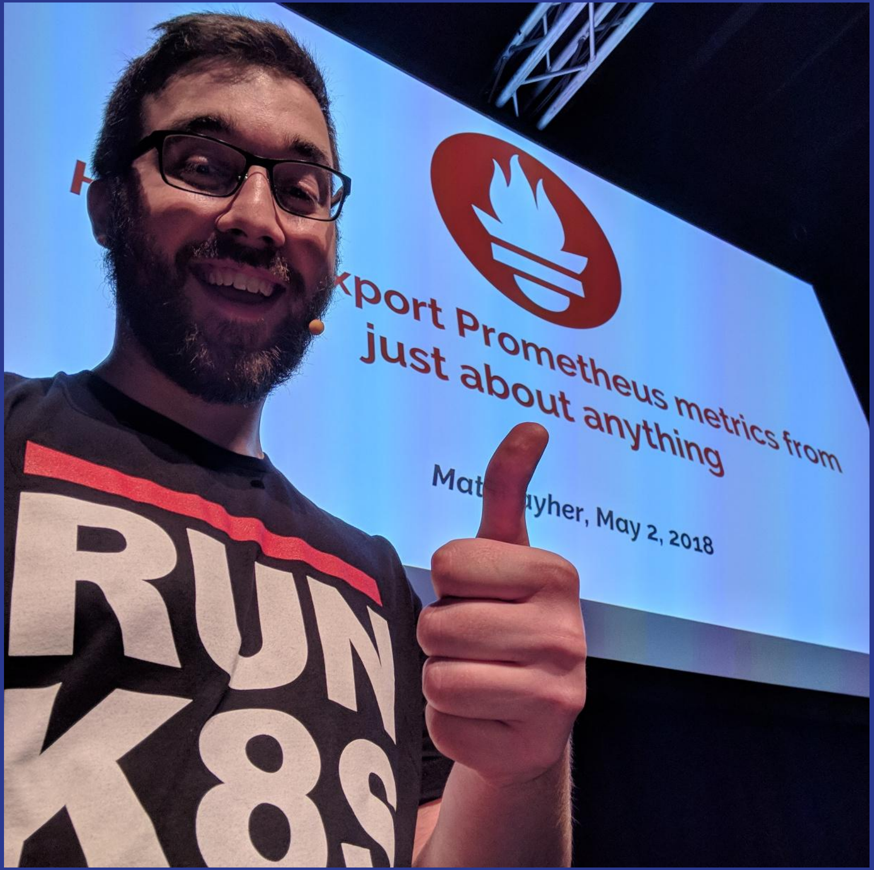
Implementing a Network Protocol in Go

Matt Layher, August 29, 2018

Matt Layher

- Engineer at DigitalOcean
- GitHub + Twitter: @mdlayher
- github.com/mdlayher/talks



Intro

- The IPv6 **Neighbor Discovery Protocol (NDP)** is our focus for today
- IPv6 is an important step for the Internet
- Go can be used for many low-level networking applications

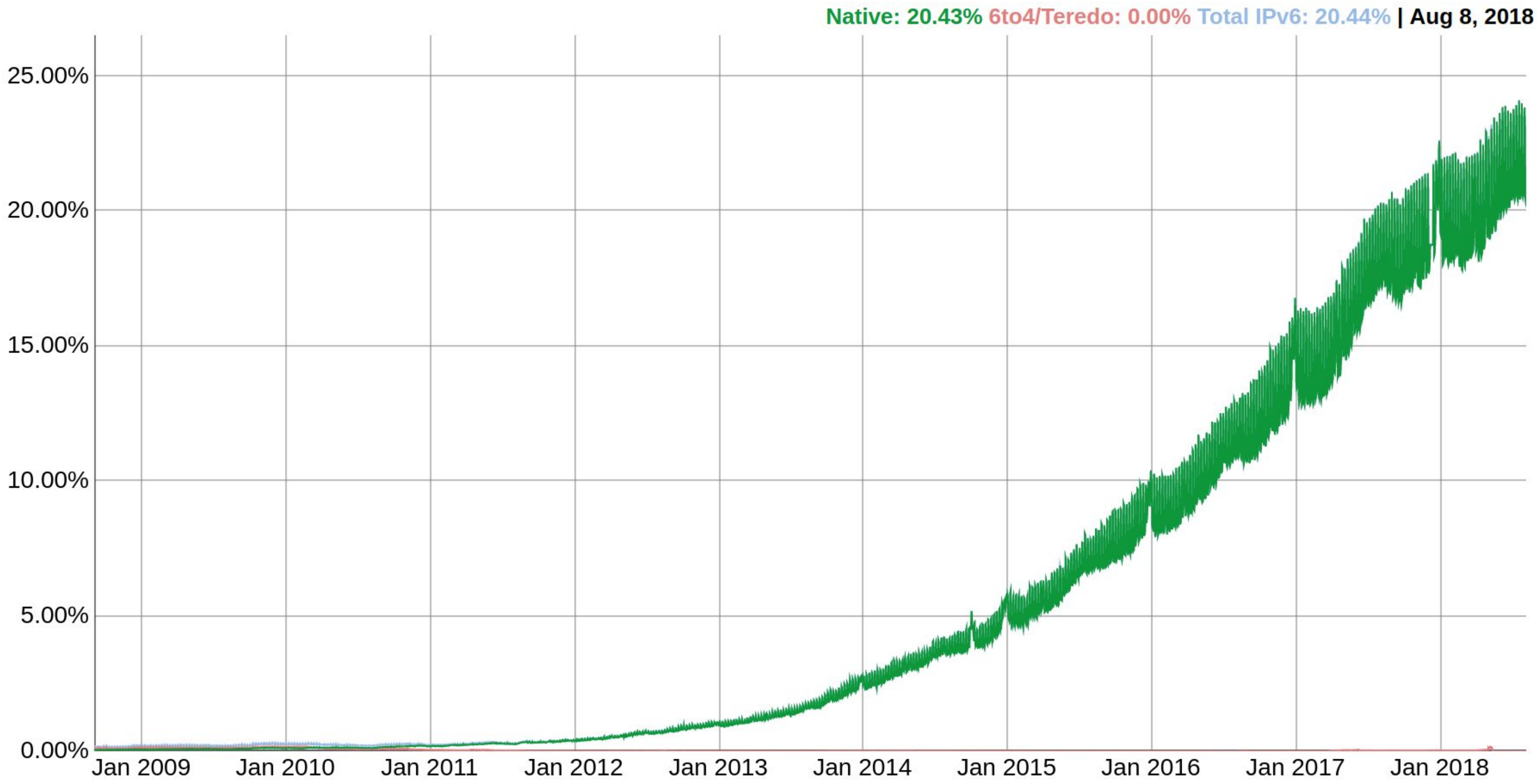
Agenda

- Introduction to IPv6 and NDP
- Using NDP with Go: github.com/mdlayher/ndp
- Building and testing network protocol packages

Introduction to IPv6

How many audience members are familiar with IPv6?

How many audience members use IPv6 at home?



User IPv6 adoption on Google services: https://www.google.com/intl/en/ipv6/statistics.html

What is IPv6?

- The next generation of the Internet Protocol
- Draft standard in RFC 2460 (December 1998!)
- 128 bit IP addresses, huge improvement over 32 bit IPv4 addresses:
    - **IPv6: 2^{128}** addresses, but not all used for hosts
    - IPv4: 2^{32} addresses

How is IPv6 different from IPv4?

- Wire format simplified, more extensible
- Residential ISPs can offer entire IPv6 prefixes instead of 1 IPv4 address:
    - **IPv6:** 2001:db8:abcd:ffff::/64: **2^{64} addresses**
    - IPv4: 192.0.2.10/32: 1 address

IPv6 tips and tricks

- Many shell utilities have a “-6” flag to use IPv6
- A useful website for testing IPv6 configuration: ipv6-test.com
- My favorite ping target:

```
$ ping6 2600::
PING 2600::(2600::) 56 data bytes
64 bytes from 2600::: icmp_seq=1 ttl=48 time=56.9 ms
```

Introduction to NDP

What is NDP?

- Effectively the IPv6 equivalent to IPv4’s ARP
- Runs on top of IPv6 + ICMPv6 with link-local addresses: fe80::/10
- Used to ask a network neighbor for its MAC address using IPv6 address
  - **A:** Who has “**B**”? Tell “**A**”.
  - **B:** “**B**” is at “**04:18:d6:a1:ce:b7**”.

ICMPv6 header

Type (8 bits)	Code (8 bits)	Checksum (16 bits)
Data (N bytes)		

NDP vs ARP

	NDP	ARP
Transport	IPv6 + ICMPv6	Ethernet
Traffic type	Multicast	Broadcast
Options	Yes	No
Router discovery	Yes	No
Address assignment	Yes	No

IPv6 and NDP's big advantage

- **DHCP is not usually necessary** to configure globally-routable IPv6 addresses:
  - **Stateless Address Autoconfiguration (SLAAC) via NDP**
    - No DHCPv6 required whatsoever
  - SLAAC + Stateless DHCPv6
    - Addresses via SLAAC, more configuration via DHCPv6
  - Stateful DHCPv6
    - All addresses and information from DHCPv6

SLAAC via NDP

- SLAAC uses NDP router advertisements to provide IPv6 prefix information
  - “**A**” sends a router solicitation
  - “**R**” sends a router advertisement:
    - Prefix **“P::/64”,** use SLAAC, valid for 24 hours
  - **“P:76d4:35ff:fee7:cbc4”** computed and assigned

NDP and Go

NDP and Go

- Your operating system usually handles NDP; why is it useful for Go programs?
- github.com/mdlayher/ndp: Go package for using NDP
  - MetalLB: implements IPv6 Layer 2 mode for Kubernetes load balancer
  - cmd/ndp: tool for generating and capturing NDP traffic
  - DigitalOcean internal use: responding to neighbor/router solicitations

Package ndp overview

- Primary types:
    - **`ndp.Message`** interface: marshaling/unmarshaling of NDP messages
    - **`ndp.Option`** interface: marshaling/unmarshaling of NDP options
    - **`ndp.Conn`** struct: manage ICMPv6 connection, read/write `ndp.Messages`

How do we go from bytes to a complete NDP package?

From bytes to messages

NDP message basics

- ICMPv6 header determines which NDP message is used
  - Type specifies NDP message, Code always 0
- Initial NDP messages and options defined in RFC 4861
  - Fixed length messages, variable options

Parsing bytes

- An ICMPv6 header will always precede an NDP message
- NDP messages on their own are not useful without the ICMPv6 header
- Exporting marshal/unmarshal methods bloats the API and GoDoc
    - **Solution:** add functions which always add/remove the ICMPv6 header

ndp.Message interface

- Exported Type method for documentation, but other methods unexported

```go
// A Message is a Neighbor Discovery Protocol message.
type Message interface {
    // Type specifies the ICMPv6 type for a Message.
    Type() ipv6.ICMPType

    // Called via MarshalMessage and ParseMessage.
    marshal() ([]byte, error)
    unmarshal(b []byte) error
}
```

ndp.ParseMessage function

- Bounds checking validation, determine concrete type, continue parsing

```go
func ParseMessage(b []byte) (Message, error) {
    // Bounds check!!!

    // Determine ndp.Message from ICMPv6 type.

    // Unmarshal ICMPv6 data into ndp.Message implementation.
}
```

Bounds checking

- When using slice elements, you must perform bounds checks to avoid panics

```
// The ICMPv6 header is fixed length.
const icmpLen = 4
if len(b) < icmpLen {
    return nil, io.ErrUnexpectedEOF
}
```

Determining ndp.Message type

- Use a switch to choose the right interface implementation

```go
// Select the correct ndp.Message type based on ICMPv6 header.
var m Message
switch t := ipv6.ICMPType(b[0]); t {
case ipv6.ICMPTypeNeighborSolicitation:
    m = new(NeighborSolicitation)
default:
    return nil, fmt.Errorf("ndp: unrecognized ICMPv6 type: %d", t)
}
```

Unmarshal the `ndp.Message` implementation

- Call into the type's methods to do the rest of the work; skipping the header

```
// Unmarshal remaining bytes into correct ndp.Message type.
if err := m.unmarshal(b[icmpLen:]); err != nil {
    return nil, err
}
```

Parsing messages

- Using `ndp.ParseMessage`, it’s easy to parse `ndp.Message` types
- Concise API: one parsing function
- **Correctness and simplicity first**, performance optimizations later

Our first `ndp.Message`

- Neighbor Solicitation (NS) messages ask a machine for its MAC address
- For now, `ndp.Option` is unimplemented

```
// An Option is a Neighbor Discovery Protocol option.
type Option interface {
    // TODO!
}
```

ICMPv6 header + NDP NS message

135	**0**	Checksum (16 bits)
Reserved (32 bits)		
Target IPv6 Address (128 bits)		
Options (N bytes)		

ndp.NeighborSolicitation type

- Mimic structure defined by RFC, use doc comments to provide references

```go
// A NeighborSolicitation is a Neighbor Solicitation message as
// described in RFC 4861, Section 4.3.
type NeighborSolicitation struct {
    TargetAddress net.IP
    Options       []Option
}
```

Checking for IPv4 and IPv6 addresses

- `net.IP` can contain IPv4, IPv6, or totally invalid IP addresses
  - A combination of `To4` and `To16` methods determine the actual type
  - Something I'd love to see improved upon in Go 2
    - `net.IP` interface? `net.IPv4` and `net.IPv6` types?

checkIPv6 function

```go
// checkIPv6 verifies that ip is an IPv6 address.
func checkIPv6(ip net.IP) error {
    // To16 returns nil when ip is not a valid IPv4/IPv6 address.
    //
    // To4 returns non-nil when ip is an IPv4 address.
    if ip.To16() == nil || ip.To4() != nil {
        return fmt.Errorf("ndp: invalid IPv6 address: %q",
            ip.String())
    }

    return nil
}
```

ndp.NeighborSolicitation unmarshaling

- Validate incoming bytes, replace the structure all at once

```
func (ns *NeighborSolicitation) unmarshal(b []byte) error {
    // Bounds checking!!!

    // Validation

    // Replacing contents of the NeighborSolicitation
}
```

Validating byte inputs

- Ensure that field values make sense, typically using rules defined by RFC

```
// Skip reserved area.
addr := b[4:nsLen]
if err := checkIPv6(addr); err != nil {
    return err
}
```

Replacing the structure while unmarshaling

- Dereference pointer and replace contents with completed structure
- Always make a copy of data from the input slice; don't assume it's safe to retain

```
*ns = NeighborSolicitation{
    TargetAddress: make(net.IP, net.IPv6len),
    Options:       options,
}

copy(ns.TargetAddress, addr)
```

From messages to bytes

Marshaling messages

- An ICMPv6 header will always precede an NDP message
- NDP messages on their own are not useful without the ICMPv6 header
- Do the parsing operation in reverse

`ndp.MarshalMessage` function

- Marshal an `ndp.Message` into binary, prepend ICMPv6 header

```go
func MarshalMessage(m Message) ([]byte, error) {
    // Call m’s marshal method

    // Pack bytes into an ICMPv6 header
}
```

Marshaling `ndp.Messages`

- **Simplicity wins**; allocating is okay until your performance needs are not met

```
mb, err := m.marshal()
if err != nil {
    return nil, err
}
```

Marshaling ICMPv6 messages

- **Simplicity wins**; allocating is okay until your performance needs are not met

```
im := icmp.Message{
    Type: m.Type(),
    Body: &icmp.DefaultMessageBody{
        Data: mb,
    },
}

return im.Marshal(nil)
```

ndp.NeighborSolicitation marshaling

- **Validate** before you **allocate**

```go
func (ns *NeighborSolicitation) marshal() ([]byte, error) {
    // Validation

    // Allocation
}
```

Validate before you allocate

- **Don't bother allocating memory** until you've checked your inputs

```go
// Only accept IPv6 target.
if err := checkIPv6(ns.TargetAddress); err != nil {
    return nil, err
}
```

Allocate once, if possible

- Allocating once is ideal for speed, but keep it simple

```go
// Allocate enough space for base message.
b := make([]byte, nsLen)
copy(b[4:], ns.TargetAddress)

// Append any option bytes.
ob, err := marshalOptions(ns.Options)
if err != nil {
    return nil, err
}
b = append(b, ob...)
```

When allocating memory...

- **Simplicity wins**; allocating is okay!
- **Write comprehensive unit tests** to lock in your behavior
- **Measure** for bottlenecks using Go benchmarks and `pprof`
- **Optimize** only after finding evidence of performance issues

`ndp.Message` API

ndp.Message types

NDP RFC messages	github.com/mdlayher/ndp
Neighbor Advertisement	`ndp.NeighborAdvertisement`
Neighbor Solicitation	`ndp.NeighborSolicitation`
Router Advertisement	`ndp.RouterAdvertisement`
Router Solicitation	`ndp.RouterSolicitation`
Redirect	n/a, not necessary yet

ndp.Message usage

```
m := &ndp.NeighborSolicitation{
    TargetAddress: target,
    Options: []ndp.Option{&ndp.LinkLayerAddress{
        Direction: ndp.Source,
        Addr:      addr,
    }},
}

b, err := ndp.MarshalMessage(m)
if err != nil {
    return fmt.Errorf("failed to marshal: %v", err)
}
```

ndp.Message usage

```go
m, err := ndp.ParseMessage(b[:n])
if err != nil {
    return fmt.Errorf("failed to parse: %v", err)
}

switch m := m.(type) {
case *ndp.NeighborAdvertisement:
    printNA(m)
case *ndp.NeighborSolicitation:
    printNS(m)
default:
        log.Printf("%#v", m)
}
```

From bytes to options

NDP option basics

- Options are encoded in type, length, value format
    - Fixed length: type
    - Fixed length: length
    - Variable length: value/data

TLV options

Type (8 bits)	Length (8 bits)	Data (N bytes)
...		

ndp.Option interface

```go
// An Option is a Neighbor Discovery Protocol option.
type Option interface {
    // Code specifies the NDP option code for an Option.
    Code() uint8

    // Called when dealing with a Message's Options.
    marshal() ([]byte, error)
    unmarshal(b []byte) error
}
```

Parsing options

- An NDP message will always precede options
- NDP options on their own are not useful without an NDP message
- Exporting marshal/unmarshal methods bloats the API and GoDoc
  - **Solution:** use unexported functions with `ndp.ParseMessage` and `ndp.MarshalMessage`

marshalOptions function

```go
// marshalOptions marshals Options into a single byte slice.
func marshalOptions(options []Option) ([]byte, error) {
    // For each option...

        // Marshal the option

        // Append it to the output
}
```

parseOptions function

```go
// parseOptions parses a slice of Options from a byte slice.
func parseOptions(b []byte) ([]Option, error) {
    // Iterate until no bytes remain...

        // Bounds check!!!

        // Read 2 bytes: type/length

        // Determine if option is known

        // Append to output slice
}
```

`ndp.Option` types

- Common option types:
  - Source/target link-layer address
  - MTU
  - Prefix information
  - Recursive DNS server
  - … and more! API could bloat quickly!

Tips for implementing options

- Consider only implementing the most common options in your package
  - Prevent API bloat, support 90% of use cases
- **Tip:** add a “raw option” type or similar to enable further extension

ndp.RawOption type

```go
// A RawOption is an Option in its raw and unprocessed format.
// Unknown Options can be represented using a RawOption.
type RawOption struct {
    Type   uint8
    Length uint8
    Value  []byte
}

// Code implements Option.
func (r *RawOption) Code() byte { return r.Type }
```

`ndp.Option` types

NDP RFC options	github.com/mdlayher/ndp
Source/Target link-layer address	`ndp.LinkLayerAddress`
MTU	`ndp.MTU`
Prefix information	`ndp.PrefixInformation`
Recursive DNS server (RDNSS)	`ndp.RecursiveDNSServer`
???	`ndp.RawOption`

ndp.Option usage

```
var ra ndp.RouterAdvertisement
ra.Options = []ndp.Option{
    &ndp.LinkLayerAddress{
        Direction: ndp.Source,
        Addr:      addr,
    },
    ndp.NewMTU(1500),
    &ndp.PrefixInformation{
        PrefixLength: 32,
        Prefix:       net.ParseIP("2001:db8::"),
        SLAAC:        true,
    },
}
```

Fuzzing byte parsers

panic: runtime error: slice bounds out of range

```
goroutine 127 [running]:
testing.tRunner.func1(0xc4201453b0)
        /usr/local/go/src/testing/testing.go:742 +0x29d
panic(0x5dd240, 0x7378d0)
        /usr/local/go/src/runtime/panic.go:502 +0x229
github.com/mdlayher/ndp.(*NeighborAdvertisement).unmarshal(0xc42013d280, 0xc4200dd924,
0x10, 0x1c, 0xc4200dd920, 0x10)
        /home/matt/src/github.com/mdlayher/ndp/message.go:149 +0x2b1
github.com/mdlayher/ndp.ParseMessage(0xc4200dd920, 0x14, 0x20, 0x4, 0x14, 0xc4200dd920,
0x4)
        /home/matt/src/github.com/mdlayher/ndp/message.go:85 +0x168
github.com/mdlayher/ndp_test.TestParseMessageError.func1.1(0xc4201453b0)
        /home/matt/src/github.com/mdlayher/ndp/message_test.go:176 +0xd3
testing.tRunner(0xc4201453b0, 0xc420141440)
        /usr/local/go/src/testing/testing.go:777 +0xd0
created by testing.(*T).Run
        /usr/local/go/src/testing/testing.go:824 +0x2e0
```

Enter `go-fuzz`

- If you're parsing raw bytes, there's a high potential for unexpected behavior:
  - Bad input causing application problems
  - Possibility of a panic taking down your program!
- github.com/dvyukov/go-fuzz
  - Throws arbitrary bytes at your parser and finds crashers!
  - Mark inputs as "interesting" or not to guide fuzzer

go-fuzz setup

```go
//+build gofuzz

package ndp

// Fuzz is an entry point for go-fuzz.
func Fuzz(data []byte) int {
    return fuzz(data)
}
```

go-fuzz setup

```go
func fuzz(data []byte) int {
    m, err := ParseMessage(data)
    if err != nil {
        return 0 // Invalid, not interesting!
    }
    b2, err := MarshalMessage(m)
    if err != nil {
        panic(err)
    }
    if _, err := ParseMessage(b2); err != nil {
        panic(err)
    }
    return 1 // Valid, interesting!
}
```

go-fuzz usage

- Prepare the fuzzer by building an instrumented test program

```
$ CGO_ENABLED=0 go-fuzz-build github.com/mdlayher/ndp
```

- Run go-fuzz with multiple CPUs and output results to ./fuzz/

```
$ go-fuzz -bin ./ndp-fuzz.zip -procs 16 -workdir ./fuzz/
… workers: 16, corpus: 78 (0s ago), crashers: 5
^C
```

go-fuzz usage

- Inspect the resulting crasher inputs

```
$ cat fuzz/crashers/4c24217a9963fae05ea48d657c342549a731989.quoted
        "\x860000000000000000\x05"
```

- Write a test, fix the bug, and repeat!

```
fuzz([]byte("\x860000000000000000\x05"))
```

`go-fuzz` conclusions (use it!)

- Use go-fuzz on ALL byte parsers: github.com/dvyukov/go-fuzz
  - Find parsing problems now, not during a 3am outage
  - A multi-worker mode is available for use with clusters of machines
  - Submit “trophies” to the `go-fuzz` README

ndp.Conn struct

Building connection types

- "`Conn`" types represent network connections
  - "`Dial`" and/or "`Listen`" constructors
  - "`Close`" to free resources
  - "`Read`" and "`Write`" to pass messages

`net` vs `x/net`

- NDP is transported over IPv6 + ICMPv6
- Standard library `net` doesn't quite provide all the functionality we need
  - golang.org/x/net is designed for advanced use-cases!

ICMPv6 networking in Go

- golang.org/x/net/icmp
- golang.org/x/net/ipv6
- Huge shout-out to Mikio Hara for his work on these packages and countless other low-level networking packages for Go

ICMPv6 listener

- Privileged operation; usually needs root

```go
// Open raw ICMPv6 listener on eth0's link-local address.
addr := "fe80::7d64:35ff:fee7:cbc4%eth0"
ic, err := icmp.ListenPacket("ip6:ipv6-icmp", addr)
if err != nil {
    return err
}
```

Reading ICMPv6 messages

- Similar to standard APIs, but also returns IPv6 control messages

```go
b := make([]byte, 1024)
n, cm, src, err := c.pc.ReadFrom(b)
if err != nil {
    return nil, nil, nil, err
}

return b[:n], cm, src.IP, nil
```

Writing ICMPv6 messages

- Similar to standard APIs, but you can specify IPv6 control messages

```
// Write bytes to the specified target.
_, err := c.pc.WriteTo(b, cm, &net.IPAddr{
    IP:   ip,
    Zone: c.ifi.Name,
})
return err
```

ndp.Conn usage

Creating an ndp.Conn

- Select an interface, dial ICMPv6, specify an address to listen on

```go
ifi, err := net.InterfaceByName("eth0")
if err != nil {
    log.Fatalf("failed to get interface: %v", err)
}

// Dial IPv6 + ICMPv6 connection.
c, ip, err := ndp.Dial(ifi, ndp.LinkLocal)
if err != nil {
    log.Fatalf("failed to dial NDP: %v", err)
}
```

Reading `ndp.Messages`

- Keep reading and printing messages until an error occurs

```go
for {
    msg, _, from, err := c.ReadFrom()
    if err != nil {
        return nil, err
    }

    printMessage(msg, from)
}
```

Writing ndp.Messages

- Send a router solicitation to trigger router advertisements on the network

```go
m := &ndp.RouterSolicitation{
    Options: []ndp.Option{&ndp.LinkLayerAddress{
            Direction: ndp.Source, Addr: addr,
    }},
}

dst := net.IPv6linklocalallrouters
if err := c.WriteTo(m, nil, dst); err != nil {
    return nil, err
}
```

Build a tool to test your package

Build a tool to test your package

- Add a cmd/ directory with a testing utility during development
- Consider building it out to become a useful tool
    - cmd/ndp: tool for generating and capturing NDP traffic

Introducing the ndp tool

- `$ ndp [listen]`
    - Listen for any NDP messages that pass through the interface
- `$ ndp rs`
    - Send a router solicitation; wait for a router advertisement
- `$ ndp -t fd00::1 ns`
    - Send a neighbor solicitation; wait for a neighbor advertisement

tcpdump?

```
$ sudo tcpdump -i enp4s0 'icmp6 && (ip6[40] == 133 or ip6[40] == 134)'
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on enp4s0, link-type EN10MB (Ethernet), capture size 262144
bytes
16:02:42.774725 IP6 nerr-2 > ip6-allrouters: ICMP6, router
solicitation, length 16
16:02:42.777116 IP6 _gateway > ip6-allnodes: ICMP6, router
advertisement, length 88
```

ndp!

```
$ sudo ./bin/ndp rs
ndp> interface: enp4s0, link-layer address: 74:d4:35:e7:cb:c4, IPv6
address: fe80::e563:9887:3aca:e01e
ndp rs> router solicitation:
    - source link-layer address: 74:d4:35:e7:cb:c4

ndp rs> router advertisement from: fe80::618:d6ff:fea1:ceb7:
    - hop limit:        64
    - router lifetime:  30m0s
    - options:
        - prefix information: 2600:6c4a:787f:d200::/64, flags: [OA],
valid: 24h0m0s, preferred: 4h0m0s
        - prefix information: fd00::/64, flags: [OA], valid: 24h0m0s,
preferred: 4h0m0s
        - source link-layer address: 04:18:d6:a1:ce:b7
```

Troubleshooting your ISP's equipment with Go

- IPv6 works for a few days...
- Ubiquiti EdgeRouter Lite can run Go programs!

```
desktop $ GOARCH=mips64 go build -o ndp_mips64
desktop $ scp ndp_mips64 router:~/ndp
 router $ sudo ./ndp -i eth1 rs
```

```
$ sudo ./ndp -i eth1 rs
ndp> interface: eth1, link-layer address: 04:18:d6:a1:ce:b7, IPv6
address: fe80::618:d6ff:fea1:ceb7
ndp rs> router solicitation:
    - source link-layer address: 04:18:d6:a1:ce:b7
.......................................................................
........................^C
ndp rs> sent 95 router solicitation(s)
```

Troubleshooting your ISP’s equipment with Go

- No luck with tech support: “your WiFi router isn’t working”
- … a modem swap during an upgrade made the problem disappear

¯_(ツ)_/¯

Conclusions

Conclusions

- IPv6 is great, check out ipv6-test.com to see if you're using it
- Network protocols are powerful building blocks
- Go is an excellent language for exploring low-level network protocols
- Build tools to solve real problems on your network!

Resources

- github.com/mdlayher/ndp
- Network Protocol Breakdown: NDP and Go
- RFC 4861

Thanks!

Matt Layher
github.com/mdlayher
twitter.com/mdlayher

Image credit: worldipv6launch.org