Porting STUNMESH-go to FreeBSD and macOS: Building Peer-to-Peer WireGuard Networks

Yu-Chiang (Date) Huang <tjjh89017 [at] hotmail.com>
@AsiaBSDCon 2026

Who am I: Date (Yu-Chiang) Huang

- Cloud and Network Solution Architect with 7+ years of experience
- Creator of STUNMESH-go and EZIO Project
- Expertise in major public cloud networking services and on-prem datacenter networking design
- Specialized in Cloud Network, OpenStack, Kubernetes, SD-WAN, and open-source
- Extensive speaking experience at international conferences

Agenda

- **The Problem** — NAT/CGNAT blocks peer-to-peer WireGuard
- **Background** — WireGuard, NAT types, STUN protocol
- **STUNMESH-go** — Architecture and how it works on Linux
- **Porting to FreeBSD** — What broke and why
- **BSD-Specific Solutions** — BPF devices, packet capture, ICMP, wgctrl
- **Q&A**

Why This Talk Matters for BSD

- **pfSense** and **OPNsense** run on FreeBSD, thousands of firewall deployments
- WireGuard is in the **FreeBSD kernel** (if-wg), but NAT traversal tools have been **Linux-only**
- This work brings **direct P2P WireGuard** to FreeBSD, macOS, no relay servers needed

The Problem

NAT/CGNAT/Stateful Firewall Are Everywhere

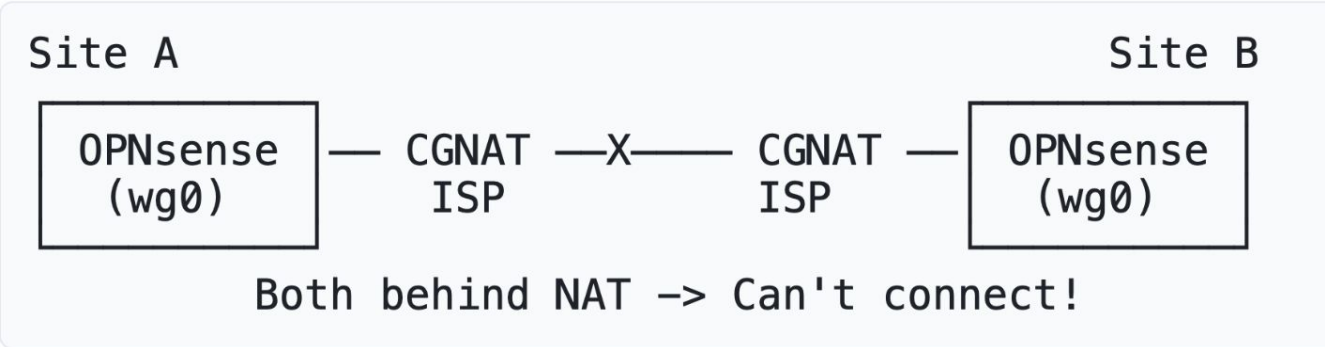
- **NAT** (Network Address Translation) conserves IPv4 addresses
- **CGNAT** (Carrier-Grade NAT) adds another layer at the ISP
- **Stateful Firewall**
  - Even in IPv6
- **Block inbound connections** by default (IT security reason)
- Peer-to-peer communication becomes difficult
  - Neither side can initiate a connection
  - Creates a "deadlock" situation

The WireGuard Challenge

- WireGuard is fast, secure, modern (~4,000 lines of code)
- But it assumes **at least one side has a stable endpoint**
- When **both peers are behind NAT:**
  - No stable public IP on either side
  - No way to start connections
- **Traditional fix:** relay servers
  - Added latency, bandwidth cost
  - Extra infrastructure to maintain for relay

This Matters for FreeBSD Firewalls

- **Scenario:** Two OPNsense firewalls at remote sites
- Both sites want a WireGuard tunnel
- Neither has a static public IP
- Without NAT traversal → need a relay server
- STUNMESH-go solves this natively on FreeBSD



Background

WireGuard on FreeBSD

- Kernel module: **if-wg** driver
  - Native kernel implementation
  - High performance, integrated with FreeBSD network stack
- **Cryptographic primitives:**
  - Curve25519 (key exchange)
  - ChaCha20 (symmetric encryption)
  - Poly1305 (authentication)
- Managed via `wg(8)` and `wgctrl-go` library

NAT Types and P2P Feasibility

- STUNMESH-go works with all cone types

NAT Type	Behavior	P2P Friendly?
Full-Cone	Any host can reach mapped port	✅ Best
Restricted-Cone	Only hosts you contacted can reply	✅ Good
Port-Restricted-Cone	IP + port must match	✅ Works
Symmetric	Different port per destination	❌ Hard

NAT Compatibility Matrix

	Full-Cone	Restricted	Port-Restricted	Symmetric
Full-Cone	✅	✅	✅	✅
Restricted	✅	✅	✅	⚠️
Port-Restricted	✅	✅	✅	❌
Symmetric	✅	⚠️	❌	❌

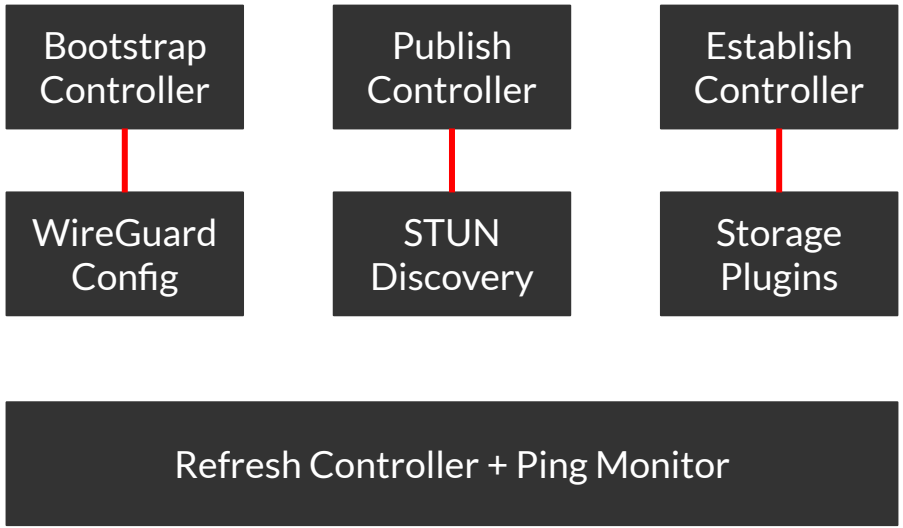
STUN Protocol (RFC 5389)

1. Client sends **Binding Request** to STUN server
2. Server reads **source IP:port** after NAT translation
3. Server responds with observed public endpoint
4. Client now knows its external address
5. Peers **exchange** endpoints → direct connection

```
Client (192.168.1.10:5000)
   → NAT → (203.0.113.5:40000)
      → STUN Server (stun.example.com:3478)
         ← Response: "You are 203.0.113.5:40000"
```

STUNMESH-go Architecture

Architecture Overview



The Four Controllers

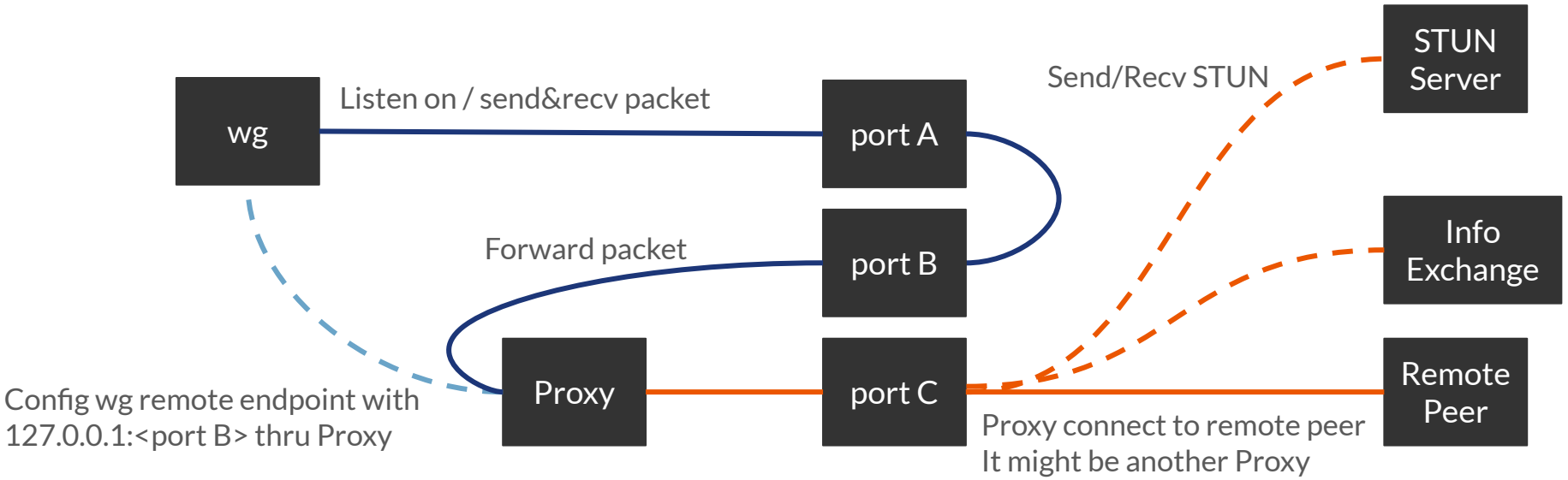
- **Bootstrap:** Init WireGuard devices, read config, map peers
- **Publish:** STUN discovery → encrypt endpoint → store via plugin
- **Establish:** Fetch peer endpoints → decrypt → configure WireGuard
- **Refresh:** Periodic re-discovery + re-establishment

The Key Innovation: UDP Port Sharing

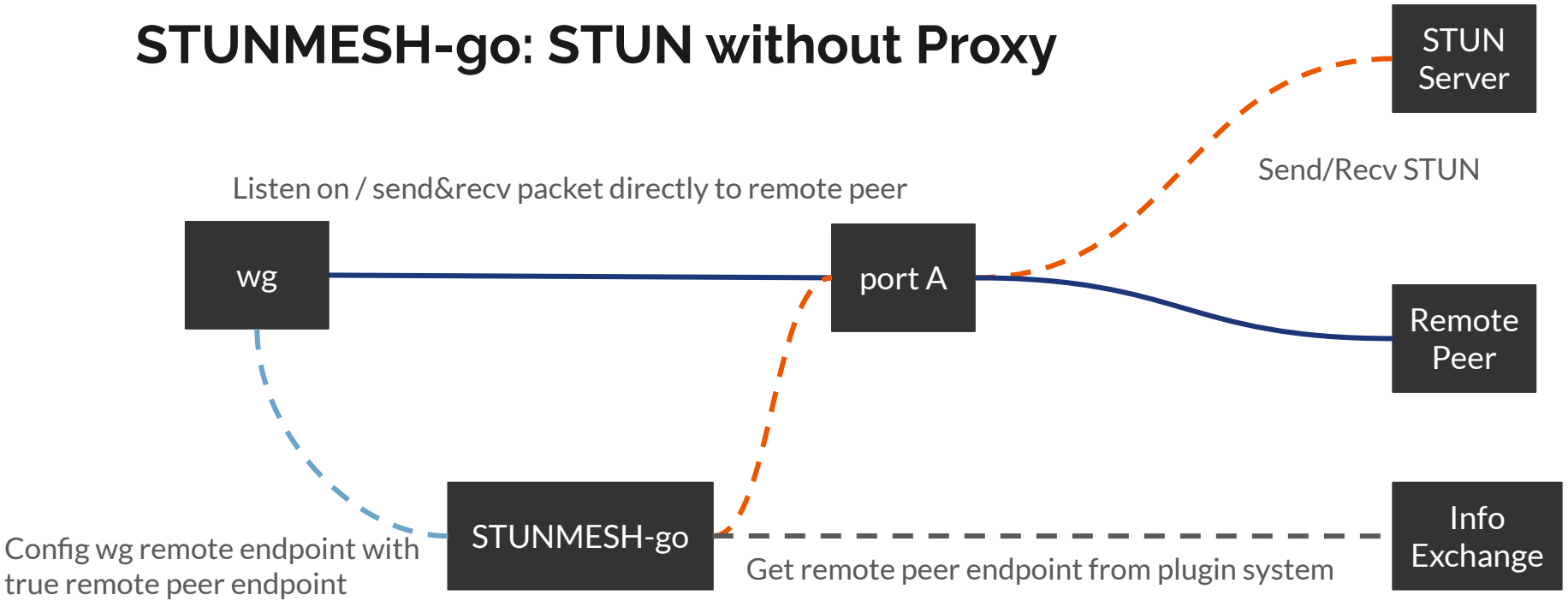
- **Raw IP socket** intercepts STUN responses before kernel UDP stack
- **BPF filter:** check dest port + STUN magic cookie 0x2112A442
- WireGuard traffic flows normally, zero interference
- No extra ports, no proxy layers

Tranditional way: STUN with Proxy

Proxy does the “Port Translation” for WG for easy STUN procedure



STUNMESH-go: STUN without Proxy



Linux STUN Implementation

This works because Linux raw sockets are system-wide:

- One socket → all interfaces
- BPF attaches directly to the socket
- Receives packets before kernel UDP stack
- Simple, single capture loop

```
// System-wide raw IP socket for UDP (protocol 17)
conn, err := net.ListenPacket("ip4:17", "0.0.0.0")

// Attach BPF filter — captures STUN on WireGuard port
conn.SetBPF(stunBpfFilter)
```

Linux BPF Quirk: IPv4 vs IPv6 Asymmetry

- **IPv4:** IP header + UDP header + payload UDP port @ byte 22, cookie @ byte 32
- **IPv6:** UDP header + payload (IP stripped!) UDP port @ byte 2, cookie @ byte 12

- **IPv4:** BPF filter runs **before** the kernel removes the IP header
- **IPv6:** BPF filter runs **after** the kernel removes the IP header
- Application code always receives IP headers removed for both

Plugin Architecture

- Flexible endpoint storage
- **Encryption:** Curve25519 (same keys as WireGuard).

Built-in:	Compiled into binary	Cloudflare DNS TXT
Exec:	JSON over stdin/stdout	Custom API client
Shell:	Variable-based protocol	Simple file/Redis

Porting to FreeBSD

Challenge Overview: Linux vs FreeBSD

Area	Linux	FreeBSD
Packet capture	System-wide raw socket	Per-interface `/dev/bpf`
BPF attachment	Socket-level `SetBPF()`	Device-level `/dev/bpfN`
Link layer in BPF	IP level (no L2 header)	Full frame (Ethernet/Null)
ICMP binding	`SO_BINDTODEVICE` (VRF support)	Not needed (no VRF)
WireGuard API	`UpdateOnly` supported	`UpdateOnly` **not** supported
Build	`CGO_ENABLED=0`	`CGO_ENABLED=1`

Challenge #1: Raw Socket Model

FreeBSD — No system-wide raw sockets for this

- FreeBSD uses the **classic BSD Packet Filter** design
- Must open `/dev/bpf` **per interface**
- Must enumerate interfaces, filter eligible ones
- Must run **concurrent capture** across all interfaces

BSD Packet Filter: The /dev/bpf Model

```
Application
    ├── open(/dev/bpf0) → bind to em0  → set filter → read
    ├── open(/dev/bpf1) → bind to igb0 → set filter → read
    └── open(/dev/bpf2) → bind to vtnet0 → set filter → read
```

	(Linux (Centralized)	FreeBSD (Per-interface)
Scope	One raw socket → all interfaces	One `/dev/bpf` per interface
BPF	Single filter program	Separate filter per device
Packets	IP-level (no L2 header)	**Full frame with link layer header**

Consequences of the BSD Model

1. **Interface Enumeration:** Must discover all interfaces, exclude WireGuard (avoid self-capture) and link-down interfaces
2. **Concurrent Capture:** One goroutine per interface, channel-based synchronization for STUN responses
3. **Link Layer Headers Affect BPF Offsets:** Ethernet (14B) vs BSD loopback Null (4B)
    a. different offsets for the same fields

Challenge #2: BPF Filter Offset Problem

BPF uses absolute byte offsets: the link layer changes everything:

```
Linux raw socket (no L2 header):
  [IP Header (20B)][UDP Header (8B)][STUN Payload]
   byte 0           byte 20          byte 28

FreeBSD Ethernet (/dev/bpf on em0):
  [Eth Header (14B)][IP Header (20B)][UDP Header (8B)][STUN]
   byte 0            byte 14          byte 34           byte 42

FreeBSD Null (/dev/bpf on lo0):
  [Null Header (4B)][IP Header (20B)][UDP Header (8B)][STUN]
   byte 0            byte 4           byte 24           byte 32
```

Challenge #3: Link Layer Type Detection

Ethernet interfaces (em0, igb0, vtnet0)

```
[Dst MAC (6B)][Src MAC (6B)][EtherType (2B)][IP...][UDP...][Payload]
                            0x0800 = IPv4
                            0x86DD = IPv6
```

Challenge #3: Link Layer Type Detection

BSD Loopback / Null (lo0, pppoe0)

- IPv6 on Null interfaces needs three-way comparison in BPF: significantly more complex filter.

```
[Protocol Family (4B)][IP Header...][UDP...][Payload]
 0x02000000 = IPv4
 0x18000000 = IPv6 (AF_INET6=24)  ┐
 0x1C000000 = IPv6 (AF_INET6=28)  ├── Three possible values!
 0x1E000000 = IPv6 (AF_INET6=30)  ┘     (varies by BSD variant)
```

Challenge #4: ICMP Without SO_BINDTODEVICE

FreeBSD doesn’t have VRF support, just skip the VRF specific code

```
// Linux
syscall.SetsockoptString(fd, syscall.SOL_SOCKET,
    syscall.SO_BINDTODEVICE, "wg0")
//Pings ALWAYS go through wg0, and lookup wg0’s VRF route table


// FreeBSD
conn, _ := icmp.ListenPacket("ip4:icmp", "0.0.0.0")
// deviceName accepted but IGNORED — routing table determines interface
```

Challenge #5: WireGuard API — UpdateOnly

FreeBSD's if-wg driver does not support UpdateOnly:

- `wgctrl-go` returns error when UpdateOnly is set on FreeBSD
- Workaround: set `UpdateOnly = false` → library handles remove + re-add
- Brief connection interruption during re-add (acceptable for our use case)

```
peerConfig.UpdateOnly = false  // Must remove + re-add peer
```

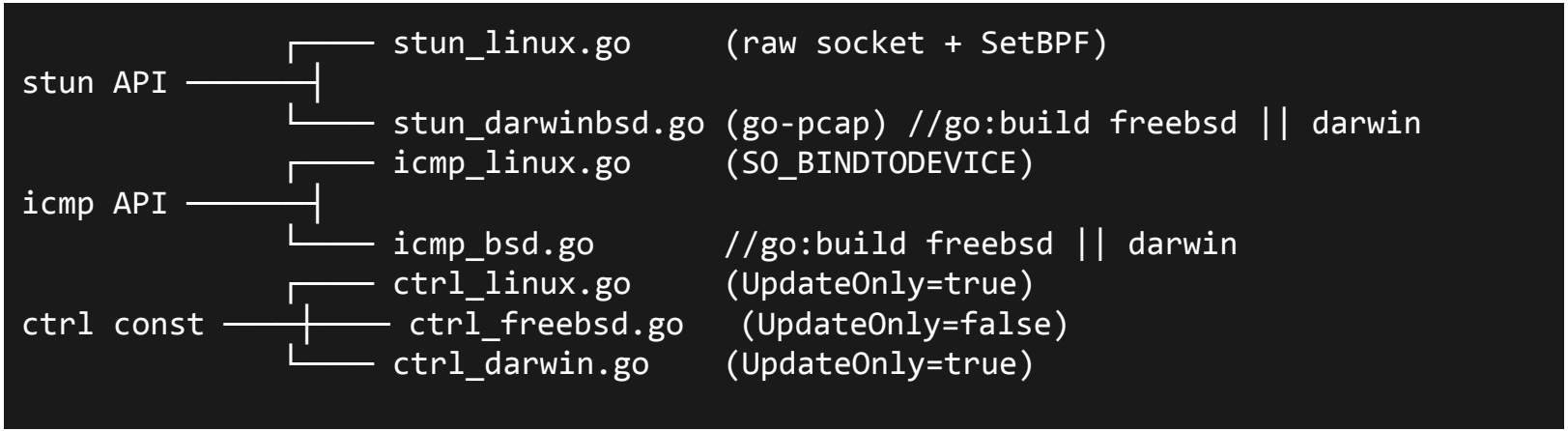
Challenge #6: CGO Requirement on FreeBSD

- FreeBSD's `wgctrl-go` needs **CGO** to include FreeBSD kernel structures:
- `wgctrl-go/internal/wgfreebsd/internal/wgh/defs.go`

BSD-Specific Solutions

Solution Strategy

- **Goal:** Identical functionality, minimal code duplication
- **Go build tags** select the right file at compile time.



Go Build Tags: Platform-Specific Compilation

Same API signature, different implementations — selected at compile time.

```
//go:build linux
package stun
func New(ctx context.Context, excludeInterface string, port uint16, protocol string) (*Stun,
error) {
    conn, _ := net.ListenPacket("ip4:17", "0.0.0.0")  // System-wide
    // SetBPF(...)
}
//go:build freebsd || darwin
package stun
func New(ctx context.Context, excludeInterface string, port uint16, protocol string) (*Stun,
error) {
    interfaces := getAllEligibleInterfaces(excludeInterface)
    // Open pcap handle per interface...
}
```

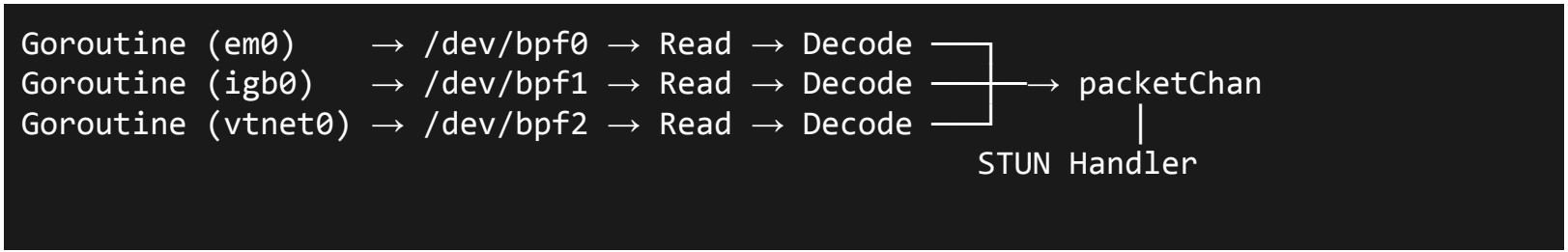
Solution: Per-Interface BPF Setup

For each eligible interface on FreeBSD:

```
for _, ifaceName := range eligibleInterfaces {
    // 1. Open /dev/bpf for this interface
    handle, _ := pcap.OpenLive(ctx, ifaceName, PacketSize, false, timeout, pcap.DefaultSyscalls)
    // 2. Detect link layer type
    linkType := handle.LinkType()
    // → pcap.LinkTypeEthernet or pcap.LinkTypeNull
    // 3. Build BPF filter with CORRECT offsets
    filter := buildBPFFilter(linkType, port, protocol)
    handle.SetRawBPFFilter(filter)
    // 4. Calculate payload offset for STUN parsing
    payloadOff := calculatePayloadOffset(linkType, protocol)
    // 5. Store handle + metadata for capture loop
    s.handles = append(s.handles, interfaceHandle{...})
}
```

Solution: Concurrent Packet Capture

- One goroutine per interface, each reads its own `/dev/bpf` device
- First to capture a valid STUN response sends to `packetChan`
- Others cancelled via Go context; `sync.Once` ensures single Start()



Solution: ICMP on FreeBSD

- Common interface, different implementation:
  - Define a new Interface for ICMP implementation

```
type ICMPConn struct { /* platform-specific */ }

func NewICMPConn(deviceName string) (*ICMPConn, error)
func (c *ICMPConn) Send(data []byte, addr net.Addr) error
func (c *ICMPConn) Recv(buf []byte, timeout time.Duration) (int, net.Addr, error)
func (c *ICMPConn) Close() error
```

Solution: WireGuard UpdateOnly Constants

- One constant per platform — clean, no runtime checks.

```
//go:build freebsd
package ctrl
const UpdateOnly = false
// FreeBSD if-wg doesn't support UpdateOnly
// wgctrl falls back to remove + re-add peer

//go:build linux
package ctrl
const UpdateOnly = true

//go:build darwin
package ctrl
const UpdateOnly = true
```

Conclusion

Key Takeaways

- STUNMESH-go now runs on FreeBSD
- BSD's BPF model is fundamentally different from Linux raw sockets
- Link layer awareness is critical
- Go build tags + interface abstractions

Get the Code

- github.com/tjjh89017/stunmesh-go
- License: GPLv2 or later
- Language: Go
- FreeBSD: amd64, arm64 (CGO enabled)
- Also: Linux (amd64/arm/arm64/mipsle), macOS (amd64/arm64)
- Contributions welcome

Q&A