Porting STUNMESH-go to FreeBSD and macOS: Navigating Platform-Specific Network Programming Challenges

Yu-Chiang (Date) Huang
Independent Researcher
Taiwan
tjjh89017@hotmail.com

***Abstract*—STUNMESH-go is an open-source tool that helps WireGuard users connect directly to each other. It works even when both users are behind NAT or CGNAT networks. The tool uses the STUN protocol to discover network addresses. It was first built for Linux systems. The tool works directly with the WireGuard kernel module and uses the same UDP port for both STUN and WireGuard traffic. This makes it efficient and means it does not need relay servers. This paper describes the problems and solutions we found when we moved STUNMESH-go to FreeBSD and macOS. We look at important differences between these systems, including how they handle raw sockets, Berkeley Packet Filter (BPF), ICMP sockets, and WireGuard APIs. Our work makes it possible for popular BSD firewall systems like pfSense and OPNsense to use direct WireGuard connections. macOS users can also use it for development and testing. We explain how to build cross-platform network applications in Go and how to handle different system APIs. We also show test results from real networks. This work helps developers who need to move network applications between Unix-like systems. It also shows that peer-to-peer VPN solutions can work well on BSD systems.**

***Index Terms*—STUN, CGNAT, peer-to-peer, WireGuard, Go, FreeBSD, macOS, BPF, packet filtering, NAT traversal**

I. Introduction

Network Address Translation (NAT) and Carrier-Grade NAT (CGNAT) are very common in modern networks. They help save IPv4 addresses. However, they create problems for peer-to-peer communication. Virtual Private Network (VPN) solutions like WireGuard are fast and secure. But they have trouble making direct connections when both users are behind NAT or CGNAT. Traditional solutions use central relay servers. These servers add delay, cost more bandwidth, and can fail.

STUNMESH-go solves this problem. It uses the STUN protocol (Session Traversal Utilities for NAT) to find network endpoints. It shares the same UDP port for both STUN and WireGuard traffic. This method allows direct peer-to-peer WireGuard connections. It works through Full-Cone, Restricted-Cone, and Port-Restricted-Cone NAT types. It does not need relay servers. STUNMESH-go works directly with the WireGuard kernel module or wireguard-go. It does not use proxy layers. This means it has lower overhead than solutions like Tailscale.

The original version was built for Linux systems. It used Linux-specific features like raw IP sockets with system-wide Berkeley Packet Filter (BPF) and the SO_BINDTODEVICE socket option. Linux is very popular for servers and embedded networks. However, FreeBSD-based systems run network infrastructure around the world. pfSense [1] and OPNsense [2] are two popular open-source firewall systems built on FreeBSD. They are used in thousands of business and home networks. Also, macOS is an important platform for network engineers and developers who work with BSD systems.

Moving STUNMESH-go to FreeBSD and macOS showed us big differences in how these systems work. These differences exist at many levels: how they capture packets, how BPF works, how raw sockets behave, how ICMP sockets work, and how system calls work. Also, WireGuard itself works differently on each platform. We had to handle these differences carefully.

This paper makes the following contributions:

- Complete documentation of how network programming differs between Linux, FreeBSD, and macOS. We focus on raw sockets, BPF, and ICMP.
- Detailed strategies for writing portable network code in Go. We explain how to use platform-specific build tags and abstraction patterns.
- Analysis of packet capture problems across different link layer types. We show how this affects BPF filter design.
- Testing results on FreeBSD 14.3-RELEASE, OPNsense 25.1, and modern macOS versions.
- Open-source code that is available for the BSD community and network engineers.

The rest of this paper is organized as follows: Section II gives background on STUN, WireGuard, and existing NAT traversal solutions. Section III describes the original Linux design. Section IV explains the platform-specific problems we found during porting. Section V presents our solutions. Section VI discusses our testing methods and results. Section VII concludes with limitations and future work.

II. Background

A. WireGuard and NAT Challenges

WireGuard [3] is a modern VPN protocol. It is designed to be simple, fast, and secure. Traditional VPN solutions like IPsec work at Layer 3. OpenVPN requires complex

handshaking. WireGuard uses a simpler approach. It uses Curve25519 for key exchange, ChaCha20 for encryption, and Poly1305 for authentication. The code is small (about 4,000 lines). This makes it easy to check and maintain.

However, WireGuard's design assumes that at least one side has a stable, reachable endpoint. When both sides are behind NAT or CGNAT, neither can start connections to the other. NAT devices usually only allow inbound packets that match existing outbound connections. This creates a deadlock where neither side can reach the other without help from outside.

B. NAT Types and Behavior

Network Address Translation comes in several types, each with different behavior for inbound connections [4]. Understanding these differences is important for peer-to-peer communication:

Full-Cone NAT: Once an internal address (IP:port) is mapped to an external address (IP:port), any external host can send packets to the internal host by sending to the mapped external address. This is the most permissive type and works well with peer-to-peer connections.

Restricted-Cone NAT: The NAT only allows inbound packets from external hosts that the internal host has previously sent packets to. However, the external host can send packets back from any source port. This means both peers must send packets to establish bidirectional communication, but the port number is not restricted.

Port-Restricted-Cone NAT: Similar to Restricted-Cone NAT, but the NAT also checks the source port number. The external host must send from the same port number that received packets from the internal host. This is the most common NAT type in home routers.

Symmetric NAT: The NAT creates a different mapping for each destination address and port. When the internal host sends to different external destinations, the NAT uses different external ports for each connection. This makes peer-to-peer connections very difficult because the external port seen by a STUN server differs from the port seen by the actual peer.

STUNMESH-go enables direct peer-to-peer connections in various NAT scenarios. When both sides use cone NAT types (Full-Cone, Restricted-Cone, or Port-Restricted-Cone), connections work reliably. When one side has Symmetric NAT, Full-Cone NAT on the other side guarantees connectivity, while Restricted-Cone NAT may establish connections depending on timing and port allocation behavior. However, Port-Restricted-Cone NAT cannot connect with Symmetric NAT due to the strict port matching requirement. Symmetric NAT on both sides requires relay servers, which is beyond the current scope.

C. STUN Protocol

Session Traversal Utilities for NAT (STUN) [5] is defined in RFC 5389. It helps computers discover their public IP address and port. A STUN client sends a request to a STUN server. The server responds with the source address and port where it received the request from. This information shows the client's public endpoint after NAT translation.

STUN allows peers to discover their external endpoints and share this information. Once both peers know each other's public endpoints, they can attempt direct connections. For cone NAT types, this enables successful peer-to-peer communication without relay servers.

D. Existing Solutions

Several solutions exist for WireGuard NAT traversal. Each has different strengths and weaknesses:

Tailscale [6] embeds a modified version of wireguard-go (the Go userspace implementation of WireGuard) in its application. It modifies wireguard-go to integrate STUN-based NAT traversal and DERP (Designated Encrypted Relay for Packets) relay servers when direct connections fail. Tailscale is easy to use and works well with NAT. However, it requires coordination servers (either Tailscale's hosted service or self-hosted Headscale) and cannot use kernel WireGuard modules directly.

Netmaker [7] focuses on mesh networking and uses a client-server architecture. The Netclient agent configures kernel WireGuard on managed nodes. However, Netmaker requires a coordination server with a public static IP address and open firewall ports for node communication. This adds deployment complexity and infrastructure requirements compared to solutions that only need access to public STUN servers.

wireguard-p2p [8] by Manuel Schölling inspired STUNMESH-go's design. It uses STUN for NAT traversal and OpenDHT for endpoint exchange. However, it did not have enough flexibility for production use. It had limitations with storage backends and multi-peer scenarios.

STUNMESH-go is different because it works directly with kernel WireGuard. It avoids the overhead of embedded Go versions. It provides a flexible plugin system for endpoint storage. This supports many backends from simple file storage to DNS TXT records in Cloudflare.

III. STUNMESH-GO ARCHITECTURE

Understanding the original Linux version is important. It helps explain the porting problems we discuss later. STUNMESH-go uses a controller-based design with four main parts:

A. Core Controllers

Bootstrap Controller sets up WireGuard devices and finds existing peer configurations. It reads configuration files, checks WireGuard interface states, and creates the initial peer mapping between configuration and device state.

Publish Controller performs STUN discovery to find the device's public endpoint. It binds to the same UDP port that WireGuard uses. It sends STUN binding requests and receives responses. These responses show the NAT-translated address and port. The discovered endpoint is encrypted using NaCl box (Curve25519 + XSalsa20 + Poly1305). It is then stored using configured plugins.

Establish Controller gets peer endpoints from storage plugins and decrypts them. It configures WireGuard peer settings using wgctrl library calls. This controller handles the WireGuard API interaction. It updates peer endpoints when they change.

Refresh Controller triggers periodic updates by requesting the Publish Controller to re-discover endpoints and the Establish Controller to refresh peer configurations. It runs at a configured interval. This ensures endpoints stay current as network conditions change.

B. Plugin Architecture

STUNMESH-go has a flexible plugin system. It supports three plugin types: built-in plugins compiled into the binary (like Cloudflare DNS), exec plugins that communicate using JSON over stdin/stdout, and shell plugins with simplified variable-based protocols. This design allows users to choose different storage backends. These can be simple file storage, DNS TXT records, Redis, or custom API endpoints.

C. STUN Implementation on Linux

The Linux STUN implementation is the most advanced part of STUNMESH-go. It creates raw IP sockets (using `net.ListenPacket("ip4:17", "0.0.0.0")` for IPv4). These bypass the kernel's UDP socket handling. This allows the application to build complete UDP packets including headers. Berkeley Packet Filter (BPF) [9] programs filter incoming traffic. They capture only STUN responses for the WireGuard port.

The key idea that makes port sharing work is this: raw sockets receive packets before the kernel's UDP stack processes them. The BPF filter checks the destination port and STUN magic cookie (0x2112A442). It identifies STUN responses and passes only relevant packets to the application. Normal WireGuard traffic reaches the kernel WireGuard module without any problems.

This approach works on Linux because:

1) Raw IP sockets can listen system-wide across all interfaces
2) BPF filters attach directly to raw sockets using `SetBPF()`
3) The Linux kernel processes BPF filters before removing IP headers for IPv4, but after removing them for IPv6
4) Raw socket packets and kernel UDP sockets work together without problems

D. Ping Monitoring System

The ping monitoring subsystem checks tunnel health and provides automatic recovery. For each peer, the system regularly sends ICMP Echo requests through the WireGuard tunnel to a target IP. On Linux, it uses SO_BINDTODEVICE to bind the ICMP socket to the specific WireGuard interface. This ensures pings go through the tunnel.

When pings fail, the monitor triggers publish and establish operations. It uses adaptive retry logic: fixed 2-second intervals for the first three retries, then longer intervals (5s, 10s, 15s, etc.) until reaching the configured refresh interval. This allows quick recovery from short failures. It also prevents too many retries during long outages.

IV. Platform-Specific Challenges

Moving STUNMESH-go to FreeBSD and macOS showed us basic differences in network programming interfaces. This section explains the specific technical problems we found.

A. Raw Socket and BPF Architecture

The biggest challenge was packet capture. Linux supports system-wide raw sockets with BPF filtering. FreeBSD and macOS need interface-specific packet capture using the BPF device interface (`/dev/bpf`).

1) Linux Approach: On Linux, creating a raw IP socket gives access to all packets that match the specified protocol across all network interfaces:

```
// Listen on all interfaces for UDP (protocol
    17)
conn, err := net.ListenPacket("ip4:17", "
    0.0.0.0")
```

BPF filters attach directly to this socket. The kernel sends filtered packets to the application. The BPF program sees packets at different stages depending on the protocol:

- **IPv4**: BPF filter runs before removing the IP header. It sees the full packet (IP header + UDP header + payload). Offsets for IPv4: UDP destination port at byte 22, STUN magic cookie at byte 32.
- **IPv6**: BPF filter runs after removing the IP header. It sees only UDP header + payload. Offsets for IPv6: UDP destination port at byte 2, STUN magic cookie at byte 12.

Application code receives packets with IP headers already removed for both protocols. It always sees UDP header (8 bytes) + payload.

2) FreeBSD/macOS Approach: FreeBSD and macOS use a different model from classic BSD packet filter design. Applications must:

1) List all network interfaces
2) Open a BPF device (`/dev/bpf`) for each interface to monitor
3) Set BPF filter programs for each interface
4) Read packets from multiple BPF devices

The Go ecosystem has the `github.com/packetcap/go-pcap` library. It simplifies BPF device operations. However, you still need to handle each interface separately:

```
// Open BPF for specific interface
handle, err := pcap.OpenLive(ctx, ifaceName,
    PacketSize, false, timeout,
    pcap.DefaultSyscalls)

// Set filter for this interface only
err = handle.SetRawBPFFilter(filter)
```

This design difference has several effects:

Interface Enumeration: The application must find all eligible interfaces. It must exclude the WireGuard interface itself (to avoid capturing WireGuard's own traffic) and loopback interfaces.

Multiple Capture Loops: Instead of one capture loop for all interfaces, the application creates goroutines for each interface. Each reads from its own BPF handle.

Link Layer Variations: Different interfaces may use different link layer types. Ethernet interfaces have 14-byte Ethernet headers. Loopback interfaces use 4-byte BSD loopback (Null) headers with protocol family indicators.

B. BPF Filter Offset Calculations

BPF filters use absolute byte offsets to check packet fields. Different link layer types need different offset calculations:

```
func calculatePayloadOffset(linkType uint32,
    protocol string) uint32 {
    if linkType == pcap.LinkTypeNull {
        if protocol == "ipv6" {
            // Null header + IPv6 header + UDP
     header
            return 4 + 40 + 8
        }
        // Null header + IPv4 header + UDP
    header
        return 4 + 20 + 8
    }
    // Ethernet
    if protocol == "ipv6" {
        // Ethernet + IPv6 + UDP
        return 14 + 40 + 8
    }
    // Ethernet + IPv4 + UDP
    return 14 + 20 + 8
}
```

For Ethernet frames with IPv6, the BPF filter must check the EtherType field (0x86DD) before checking UDP ports. The BSD loopback Null header uses different protocol values: 0x02000000 (big-endian) for IPv4, and 0x18000000, 0x1C000000, or 0x1E000000 for IPv6. The value depends on the specific IPv6 address family type.

C. ICMP Socket Capabilities

The ping monitoring system showed another important platform difference. Linux supports the SO_BINDTODEVICE socket option. This allows applications to bind sockets to specific network interfaces:

```
// Linux: bind ICMP socket to specific
    interface
err = syscall.SetsockoptString(fd,
    syscall.SOL_SOCKET,
    syscall.SO_BINDTODEVICE, deviceName)
```

This ensures ICMP Echo requests go through the intended WireGuard tunnel instead of the default route. Without this binding, pings might wrongly use the primary network interface. This means they would not detect tunnel-specific failures.

FreeBSD and macOS lack VRF (Virtual Routing and Forwarding) support. In a single routing table environment without VRF, device binding is not necessary for ICMP sockets. The routing table itself determines which interface to use based on the destination address. The current BSD version creates standard ICMP connections without device binding, relying on correct routing configuration:

```
// BSD: standard ICMP, no device binding
conn, err := icmp.ListenPacket("ip4:icmp",
    "0.0.0.0")
// deviceName parameter ignored
```

For setups where routing is configured correctly to send traffic to the WireGuard tunnel, this limitation has little impact. However, complex multi-interface scenarios may need additional routing configuration.

D. WireGuard API Differences

The wgctrl library [10] simplifies WireGuard userspace API access across platforms. However, platforms behave differently when updating peer configurations. Linux WireGuard allows updating existing peer endpoints without removing and re-adding peers (UpdateOnly mode). This provides atomic updates without disrupting the connection state.

FreeBSD's kernel WireGuard implementation does not yet support the UpdateOnly feature. The wgctrl-go library returns `ErrUpdateOnlyNotSupported` when this flag is set on FreeBSD. The STUNMESH-go code sets `UpdateOnly = false` for FreeBSD. This allows the library to remove and re-add peers if necessary:

```
//go:build freebsd
package ctrl

const (
    UpdateOnly = false
)


//go:build linux
package ctrl

const (
    UpdateOnly = true
)
```

macOS uses wireguard-go (the Go userspace version). It supports update-only mode similar to Linux kernel WireGuard.

E. CGO and Build Configuration

FreeBSD's wgctrl implementation requires CGO (Go's foreign function interface) for WireGuard kernel module interaction. The build system must turn on CGO for FreeBSD. It keeps it off for Linux and macOS to create static binaries:

```
# Makefile excerpt
ifeq ($(GOOS),freebsd)
    CGO_ENABLED := 1
else
    CGO_ENABLED := 0
endif
```

Cross-compilation for multiple platforms with different CGO requirements makes the build process more complex. The project uses GitHub Actions to build separate binaries for each platform (Linux amd64/arm/arm64/mipsle, Darwin amd64/arm64, FreeBSD amd64/arm64). Each has the right CGO settings.

V. Implementation

This section describes the design approaches and code patterns we used to solve the challenges mentioned above.

A. Platform-Specific Build Tags

Go's build tag system allows conditional compilation based on operating system and architecture. STUNMESH-go uses build tags extensively to keep platform-specific code separate:

```
//go:build linux
// +build linux

package stun
// Linux implementation using raw sockets

//go:build darwin || freebsd
// +build darwin freebsd

package stun
// BSD implementation using pcap
```

Files with these tags compile only for their target platforms. This approach keeps platform-specific code separate while keeping a clean API. The main application code works with a common `Stun` interface. It does not need to know about platform differences.

B. Abstracted STUN Interface

Both Linux and BSD versions provide the same exported types and methods:

```
type Stun struct {
    // Platform-specific fields
}

func New(ctx context.Context,
    excludeInterface string,
    port uint16,
    protocol string) (*Stun, error)

func (s *Stun) Connect(ctx context.Context,
    stunAddr string) (string, int, error)

func (s *Stun) Start(ctx context.Context)
func (s *Stun) Stop() error
```

The `New` function creates the right STUN client for each platform. On Linux, it creates raw sockets with system-wide BPF filters. On BSD, it lists interfaces, opens pcap handles, and sets up per-interface BPF filters.

C. BSD Interface Enumeration

The BSD version includes code to find and filter network interfaces:

```
func getAllEligibleInterfaces(
    excludeInterface string) ([]string, error)
     {
    interfaces, err := net.Interfaces()
    if err != nil {
        return nil, err
    }

    var eligible []string
    for _, iface := range interfaces {
        // Skip loopback and down interfaces
        if iface.Flags&net.FlagLoopback != 0
    ||
           iface.Flags&net.FlagUp == 0 {
            continue
        }
        // Skip the WireGuard interface itself
        if iface.Name == excludeInterface {
            continue
        }
        eligible = append(eligible, iface.Name
    )
    }
    return eligible, nil
}
```

For each eligible interface, the code opens a pcap handle, finds the link layer type, calculates the right offsets, and builds a matching BPF filter.

D. Link Layer-Aware BPF Filters

The BSD version has separate BPF filter generators for Null and Ethernet link layers. The Ethernet IPv6 filter shows EtherType checking:

```
func stunEthernetBpfFilter(ctx context.Context
    ,
    port uint16, protocol string) (
    []bpf.RawInstruction, error) {

    if protocol == "ipv6" {
        return bpf.Assemble([]bpf.Instruction{
            bpf.LoadAbsolute{
                // Check EtherType field
                Off:  12,
                Size: 2,
            },
            bpf.JumpIf{
                Cond:      bpf.JumpEqual,
                Val:       0x86DD, // IPv6
                SkipFalse: 5,
            },
            // Check UDP destination port...
            // Check STUN magic cookie...
        })
    }
    // IPv4 filter...
}
```

The Null link layer filter for IPv6 must handle three possible protocol values (0x18000000, 0x1C000000, 0x1E000000). This needs a more complex BPF program with multiple comparison branches.

E. Concurrent Packet Capture

BSD's interface-specific approach needs to read packets from multiple interfaces at the same time:

```
func (s *Stun) Start(ctx context.Context) {
    s.once.Do(func() {
        s.waitGroup.Add(len(s.handles))
        for _, ih := range s.handles {
            go func(handle interfaceHandle) {
                defer s.waitGroup.Done()
                defer handle.handle.Close()

                for {
                    select {
                    case <-ctx.Done():
                        return
                    default:
                        buf, _, err :=
                            handle.handle.
    ReadPacketData()
                        if err != nil {
                            continue
                        }
                        // Decode STUN message
                        m := &stun.Message{
                            Raw: buf[handle.
    payloadOff:],
                        }
                        if err := m.Decode();
    err != nil {
                            continue
                        }
                        // Send to channel
                        s.packetChan <- m
                        return
                    }
                }
            }(ih)
        }
    })
}
```

Each goroutine reads from its assigned interface until it captures a STUN response, then closes. This design assumes STUN responses arrive on the same interface that sent the request. This is true for typical NAT gateway setups.

F. Platform-Specific ICMP Implementation

The ICMP connection abstraction hides platform differences behind a common interface:

```
type ICMPConn struct {
    // Platform-specific fields
}

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

The Linux version creates raw ICMP sockets and uses SO_BINDTODEVICE. The BSD version uses the standard `icmp.ListenPacket` without device binding. It accepts the deviceName parameter but ignores it.

VI. Testing and Validation

We tested the ported version on multiple platforms and scenarios to ensure it works correctly in real-world conditions.

A. Test Environments

FreeBSD 14.3-RELEASE: We tested on physical hardware and virtual machines. This verified compatibility with the latest stable FreeBSD release. FreeBSD's kernel WireGuard version (if-wg driver) provided the test environment.

OPNsense 25.1: OPNsense is a production FreeBSD-based firewall system. It is a critical target platform. Virtual machines running OPNsense tested the complete workflow. This included WireGuard configuration, STUNMESH-go deployment, and multi-interface scenarios typical of firewall systems.

macOS: We tested multiple macOS versions on both Intel and Apple Silicon Macs, using wireguard-go installations. macOS is an important developer platform where network engineers develop and test configurations before deployment.

Mixed Environments: Real-world testing included mixed networks. These combined VyOS routers with LTE modems, OPNsense firewalls, Linux servers, and macOS laptops. All were behind various NAT types.

B. Test Scenarios

1) Basic Connectivity: Initial tests checked that STUN discovery correctly finds public endpoints across NAT boundaries. Two peers behind separate CGNAT networks (mobile carriers) successfully found each other and set up WireGuard tunnels.

2) Link Layer Variations: BSD testing specifically checked correct BPF filter operation across different interface types:

- Ethernet interfaces (em0, igb0): 14-byte Ethernet headers
- Loopback (lo0): 4-byte Null headers
- Virtual interfaces (vtnet0): Ethernet encapsulation

Packet captures confirmed that BPF filters correctly matched STUN responses regardless of link layer type.

3) Multi-Interface Scenarios: OPNsense deployments usually have multiple interfaces (WAN, LAN, DMZ, etc.). Testing confirmed that STUNMESH-go correctly:

- Excludes the WireGuard interface from STUN listening
- Captures STUN responses on the right interface
- Handles interface state changes (interfaces going up/down)

4) IPv6 Support: Dual-stack testing checked IPv6 STUN discovery functionality. Networks with native IPv6 connectivity successfully performed IPv6 STUN discovery, stored both IPv4 and IPv6 endpoints, and allowed peers to select endpoints based on their connectivity preferences.

5) NAT Type Coverage: Testing included multiple NAT configurations:

- Full-Cone NAT (home routers): Bidirectional connectivity established
- Port-Restricted-Cone NAT (enterprise firewalls): Connectivity established after initial handshake
- CGNAT (mobile carriers): Connectivity achieved when at least one peer uses cone NAT

Symmetric NAT scenarios were challenging, as expected. This is due to unpredictable port mapping behavior.

C. Ping Monitoring Validation

The ping monitoring system went through extensive testing to check tunnel health detection and automatic recovery:

- **Normal Operation**: Confirmed that ICMP Echo requests go through WireGuard tunnels and receive replies under normal conditions
- **Failure Detection**: Checked that network disruptions (disconnecting WAN interfaces, blocking UDP ports) trigger ping failures
- **Automatic Recovery**: Verified that the system automatically triggers endpoint re-discovery and tunnel re-establishment after failures
- **Adaptive Backoff**: Confirmed that retry intervals follow the configured pattern (fixed intervals, then longer progression)

VII. Discussion and Future Work

A. Limitations and Constraints

The current version has several known limitations:

Symmetric NAT: Like most STUN-based approaches, STUNMESH-go has trouble with symmetric NAT where both peers have symmetric configurations. Predicting port mappings in symmetric NAT scenarios needs additional techniques like repeated STUN queries or UDP hole-punching variations. These add complexity.

BSD ICMP Health Checks: Without SO_BINDTODEVICE support, BSD platforms rely on routing tables to direct ICMP ping traffic through WireGuard tunnels. This works well in typical single-WAN scenarios but may require additional routing configuration in complex multi-path setups.

Single STUN Server: Current deployments usually configure one STUN server. Multiple STUN servers with failover would improve reliability, especially for production use.

IPv6 Ping Monitoring: Ping monitoring currently supports only IPv4 target addresses, even when the tunnel itself uses IPv6. Adding IPv6 ping support needs additional work, especially for BSD platforms where IPv6 ICMP differs from IPv4.

B. Future Directions

Several areas need further exploration:

Performance Optimization: While current performance is good enough, several optimization opportunities exist: reducing per-packet processing overhead on BSD by optimizing BPF filter complexity, using more efficient packet batching, and exploring alternative packet capture libraries.

VIII. Conclusion

This paper documented the technical problems and solutions we found when moving STUNMESH-go from Linux to FreeBSD and macOS. STUNMESH-go is a WireGuard NAT traversal tool. The porting work showed basic differences in network programming interfaces across Unix-like systems. This was especially true for packet capture mechanisms, BPF implementations, and low-level socket capabilities.

Key contributions include complete documentation of platform-specific networking differences, practical strategies using Go's build tag system and interface abstractions, and testing on production BSD-based firewall systems. The result enables FreeBSD systems, including pfSense and OPNsense deployments, to use direct peer-to-peer WireGuard connectivity without relay infrastructure.

The experience shows the importance of early abstraction in cross-platform development. Despite big design differences, portable network code can achieve the same functionality across platforms with careful design. While limitations remain, especially BSD's lack of interface-bound ICMP sockets and challenges with symmetric NAT, the current version provides valuable functionality for the BSD networking community.

As WireGuard adoption continues growing in BSD environments, tools like STUNMESH-go become more important for flexible VPN deployments without static IP requirements. The open-source nature of this project encourages community contributions to address remaining limitations and add functionality for new use cases.

The complete code is available under the GPLv2 or later license at https://github.com/tjjh89017/stunmesh-go. We welcome contributions from the BSD and broader networking community.

Acknowledgments

The author thanks the FreeBSD and macOS communities for testing and feedback. Thanks also to the WireGuard developers for creating an excellent VPN protocol. Special thanks to the authors of the go-pcap library and wgctrl library. These provided essential tools for cross-platform packet capture and WireGuard control.

References

[1] pfSense Project, "pfSense Documentation," https://docs.netgate.com/pfsense/, 2025.

[2] OPNsense Project, "OPNsense Documentation," https://docs.opnsense.org/, 2025.

[3] J. Donenfeld, "WireGuard: Next Generation Kernel Network Tunnel," in Proceedings of NDSS 2017, 2017.

[4] F. Audet and C. Jennings, "Network Address Translation (NAT) Behavioral Requirements for Unicast UDP," RFC 4787, January 2007.

[5] J. Rosenberg et al., "Session Traversal Utilities for NAT (STUN)," RFC 5389, October 2008.

[6] Tailscale Inc., "How Tailscale Works," https://tailscale.com/blog/how-tailscale-works/, 2023.

[7] Gravitl, "Netmaker: WireGuard Automation Platform," https://github.com/gravitl/netmaker, 2025.

[8] M. Schölling, "wireguard-p2p: A tool for setting up WireGuard connections from peer to peer," https://github.com/manuels/wireguard-p2p, 2017.

[9] S. McCanne and V. Jacobson, “The BSD Packet Filter: A New Architecture for User-level Packet Capture,” in Proceedings of the USENIX Winter 1993 Conference, 1993.
[10] WireGuard Project, “wgctrl-go: Package wgctrl enables control of WireGuard interfaces on multiple platforms,” https://github.com/WireGuard/wgctrl-go, 2025.