Building a Home Server with HAProxy and VPN (WireGuard) on a Local Network

H. A. Mooduto[a,*]

[a] *Department of Information Technology, Politeknik Negeri Padang, Limau Manis, Padang, 25164, Indonesia*
Corresponding author: mooduto@pnp.ac.id

***Abstract*— Self-hosting web services on a home or campus server is attractive for teaching, prototyping, and low-cost publication, but many broadband subscribers sit behind network address translation (NAT) or carrier-grade NAT and cannot receive unsolicited inbound connections, while router port forwarding requires a public IPv4 address and widens the attack surface. This study aims to design, implement, and evaluate an architecture in which a low-cost virtual private server (VPS) is the only public entry point: HAProxy runs on the VPS as a reverse proxy, and a WireGuard tunnel, initiated outbound by the local server, carries requests to a web server inside the local network. The materials are open-source components, namely HAProxy, WireGuard, and an Apache/PHP/MariaDB content management stack. The method follows a six-stage network development cycle and an emulated testbed of Linux network namespaces with a NAT router without port forwarding and a 50 Mbit/s uplink, comparing direct port forwarding, HAProxy without a tunnel, and the proposed design. The local server, unreachable from the internet before deployment, became reachable through the VPS while the home router exposed no services. The tunnel added 0.85 ms to the time to first byte, kept 95.8% of underlay TCP throughput, matching the expected encapsulation overhead, and reduced request rate by 4.8% at low concurrency. Health checks detected a failed backend in 6.45 s and restored service 3.83 s after recovery, and a persistent keepalive was needed for reliable reachability. Future work should evaluate TLS termination, IPv6, and multi-site deployments on physical hardware.**

***Keywords*— Home server; HAProxy; WireGuard; reverse proxy; virtual private network; carrier-grade NAT; self-hosting.**

I. INTRODUCTION

Self-hosted services have become a practical option for educational institutions, small organizations, and individual developers who want full control over their data, software stack, and operating cost. A single server placed in a home or laboratory network can host a content management system (CMS), a learning portal, a file repository, or a student project, and it can be administered with the same tools used in production data centers. For vocational higher education in particular, such a server is also an effective learning medium because students configure every layer of the service themselves, from the operating system to the application.

The obstacle is reachability. The global IPv4 pool is exhausted, and access providers increasingly place subscribers behind carrier-grade NAT (CGN), often using the shared address space 100.64.0.0/10 [1], [2]. Measurements across many networks have shown that CGN deployment is widespread, especially in cellular and emerging-market access networks, and that it breaks applications expecting inbound connectivity [3]. Indonesia has a very large and rapidly growing internet user base [4], and most residential fixed and mobile connections are provisioned without a dedicated public IPv4 address. Under these conditions, the conventional approach of forwarding a port on the home router does not work, because the router itself does not own a publicly reachable address. Even when a public address is available, port forwarding exposes the home server and router directly to internet-wide scanning, which conflicts with zero-trust principles that recommend minimizing implicitly trusted entry points [5].

Several workarounds exist. Reverse SSH tunnels are simple but fragile and rely on a single TCP session. Commercial tunnel services and overlay networks remove the need for a public address but introduce dependence on a third-party control plane, and traversal techniques such as STUN still depend on the behavior of each NAT device [6]. A more controllable alternative is to rent a small virtual private server (VPS) with a public IPv4 address, connect the home server to it with a virtual private network (VPN), and let a reverse proxy on the VPS publish selected services. Two open-source components are well suited to this design. WireGuard is a layer-3 VPN that runs in the Linux kernel and uses a fixed modern cryptographic suite based on the Noise protocol framework, Curve25519, and ChaCha20-Poly1305 [7]–[10]; its handshake has been analyzed and mechanically verified [11], [12], and post-quantum variants have been proposed [13]. HAProxy is an event-driven layer-4/layer-7 load balancer and reverse proxy with active health checking, widely used for high-availability web services [14].

Prior studies consistently report that WireGuard has a lower overhead than OpenVPN and competitive or better performance than IPsec. Early benchmarks found WireGuard faster than OpenVPN in throughput and connection set-up [15], [16], and an analysis of VPN gateway architectures showed how the multi-core design of WireGuard affects scalability [17]. Recent work evaluated WireGuard against OpenVPN in cloud and virtualized environments [18], against IPsec across network configurations [19] and in an industrial private network [20], against OpenVPN in terms of latency and CPU consumption [21], in IoT settings together with overlay VPNs [22], [23], and in line-rate multi-tunnel deployments [24]. A comprehensive survey summarized security assessments of

VPN technologies [25]. In Indonesia, WireGuard has been applied to secure access to websites on MikroTik routers [26], to protect a community network against sniffing attacks [27], and to connect on-premise servers with a public cloud, where it outperformed a managed cloud VPN [28]. On the reverse-proxy side, HAProxy has been studied for load-balancing algorithm tuning [29], weighted scheduling of web servers [30], academic information systems [31], and TLS-protected website delivery behind a firewall [32].

These studies, however, examine either the VPN or the reverse proxy in isolation, and most assume that every node has a routable address. Little empirical work combines a VPS-hosted reverse proxy with an outbound WireGuard tunnel specifically to publish a server that has no public address, and quantifies what the combination costs in latency, throughput, and availability. This paper addresses that gap. Its contributions are: (1) a reproducible architecture and configuration for publishing a home server through HAProxy and WireGuard without port forwarding; (2) an emulated testbed that compares the design with direct port forwarding and with a reverse proxy without encryption; and (3) an analysis of reachability, response time, throughput, resource use, failover, and NAT keepalive behavior, with practical recommendations for deployment in campus and home networks.

II. Materials and Method

A. Research Framework

The study adopts the Network Development Life Cycle as used in applied networking research, adapted into six stages (Fig. 1). The problem analysis stage identified the constraints of the target environment: a local server without a public IPv4 address, a consumer router whose configuration may not be controllable, and the requirement that no inbound port be opened on the local network. The design stage defined the topology, the IP addressing plan, and the roles of HAProxy and WireGuard. Simulation prototyping built a faithful emulation of the topology so that every scenario could be repeated under identical conditions. The implementation stage generated keys and wrote the configuration files, the testing stage executed the measurement plan, and the monitoring stage observed the system through the HAProxy statistics socket and the `wg` utility.

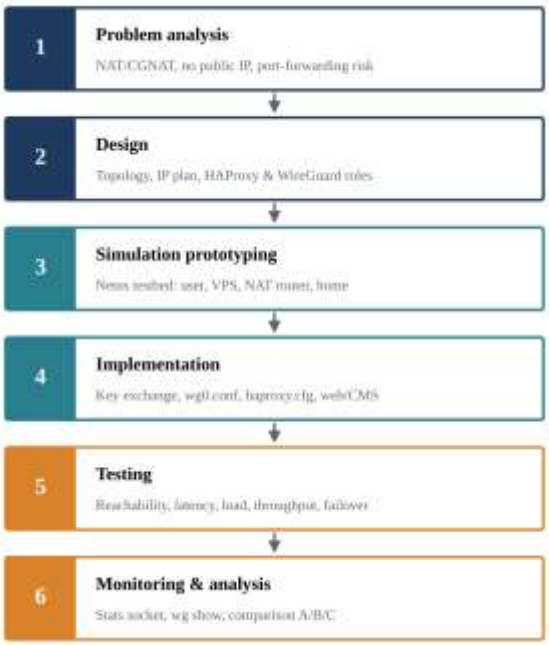


Fig. 1 Research framework adapted from the Network Development Life Cycle

B. Materials

All software components are open source. The VPS and the local server run a Linux distribution of the Red Hat family (CentOS Stream 9) in the reference deployment, with packages installed through `dnf`: `haproxy` and `wireguard-tools` on the VPS, and `wireguard-tools`, `httpd`, `php`, `php-mysqlnd`, and `mariadb-server` on the local server. Services are managed by `systemd`, using the units `haproxy` and `wg-quick@wg0`. The minimum VPS requirement is modest, because the VPS only forwards traffic: one virtual CPU, 1 GB of memory, and one public IPv4 address are sufficient for a small institutional site.

C. System Architecture

Fig. 2 shows the proposed architecture. Internet users never contact the local network directly. Their HTTP requests reach the public address of the VPS, where HAProxy accepts them on port 80 through the frontend `http_front` and forwards them to the backend `http_back`. The only backend server, `local_vpn`, is addressed by its tunnel address 10.8.0.2, so the request travels inside the encrypted WireGuard tunnel to the local server, where Apache serves the CMS. The tunnel is initiated by the local server towards UDP port 51820 of the VPS; the home router therefore only sees outbound UDP traffic and keeps a NAT mapping through which replies and subsequent requests return. Table I summarizes the addressing plan.

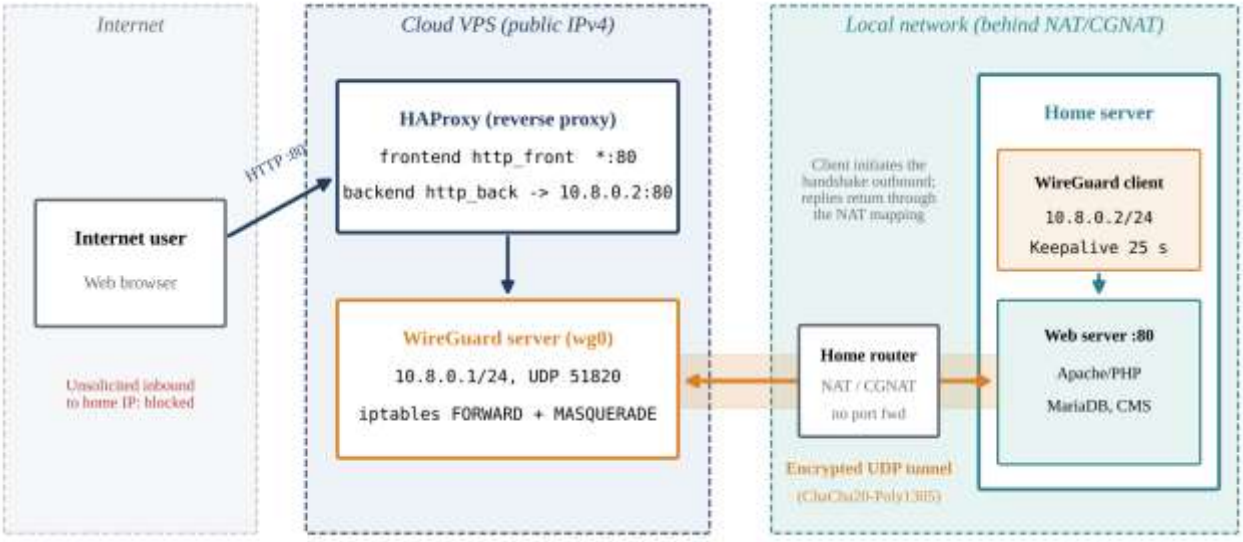


Fig. 2 Proposed architecture: HAProxy on a public VPS publishes a home server through an outbound WireGuard tunnel, so no port is opened on the home router

TABLE I
ADDRESSING PLAN AND COMPONENT ROLES

Component	Host	Address / port	Role
HAProxy frontend	VPS	public IP, TCP 80	Entry point
WireGuard server	VPS	10.8.0.1/24, UDP 51820	Tunnel hub
WireGuard client	Local server	10.8.0.2/24	Tunnel peer
Web server (CMS)	Local server	10.8.0.2, TCP 80	Backend
Home router	Local LAN	NAT, no forwarding	Gateway

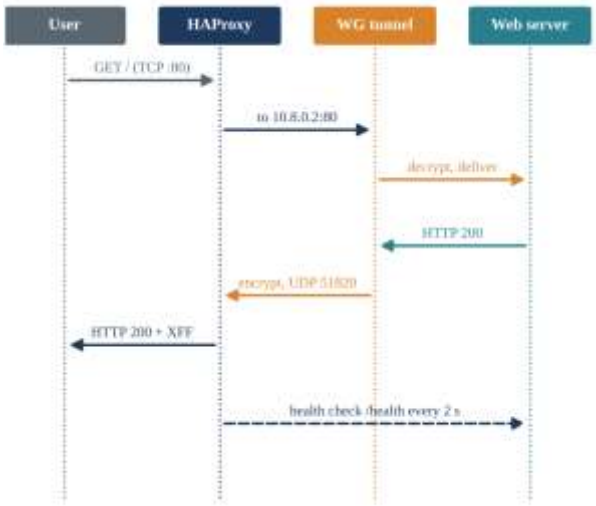


Fig. 3 Request flow from the internet user to the home web server

Fig. 3 details the request flow. HAProxy terminates the client TCP connection, opens a second connection to 10.8.0.2:80, and adds the `X-Forwarded-For` header so that the CMS can log the real client address. Independently of user traffic, HAProxy probes the backend every two seconds; after three consecutive failures the server is marked down and traffic is redirected to a backup that returns a maintenance page.

D. WireGuard Configuration

Each peer generates a key pair with `umask 077; wg genkey | tee privatekey | wg pubkey > publickey`, and the public keys are exchanged out of band. Fig. 4 shows the server configuration on the VPS. The `PostUp` and `PostDown` hooks add and remove the forwarding and masquerading rules, and `AllowedIPs = 10.8.0.2/32` binds the client key to exactly one tunnel address, which is the cryptokey routing principle of WireGuard [7]. IP forwarding must also be enabled with `sysctl -w net.ipv4.ip_forward=1`, and the egress interface name (`eth0`) must match the VPS.

```
[Interface]
PrivateKey = <vps_private_key>
Address    = 10.8.0.1/24
ListenPort = 51820
PostUp   = iptables -A FORWARD -i wg0 -j ACCEPT; \
  iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
PostDown = iptables -D FORWARD -i wg0 -j ACCEPT; \
  iptables -t nat -D POSTROUTING -o eth0 -j MASQUERADE

[Peer]
PublicKey  = <client_public_key>
AllowedIPs = 10.8.0.2/32
```

Fig. 4 WireGuard server configuration on the VPS (/etc/wireguard/wg0.conf)

Fig. 5 shows the client configuration on the local server. `PersistentKeepalive = 25` makes the client send an authenticated keepalive every 25 seconds so that the NAT mapping on the home router does not expire. The reference configuration routes all traffic through the tunnel (`AllowedIPs = 0.0.0.0/0`); Section III-G evaluates this choice against a split-tunnel alternative.

```
[Interface]
PrivateKey = <client_private_key>
Address    = 10.8.0.2/24

[Peer]
PublicKey  = <vps_public_key>
Endpoint   = <vps_ip>:51820
AllowedIPs = 0.0.0.0/0      # or 10.8.0.0/24
PersistentKeepalive = 25
```

Fig. 5 WireGuard client configuration on the local server (/etc/wireguard/wg0.conf)

E. HAProxy Configuration

Fig. 6 lists the HAProxy configuration. The `defaults` section sets HTTP mode, a 5 s connect timeout, and 50 s client and server timeouts. The backend uses an HTTP health check on a lightweight `/health` endpoint with `inter 2s fall 3 rise 2`, and a backup server that returns HTTP 503 with a maintenance page. The statistics socket supports runtime monitoring without restarting the service.

```
global
    log /dev/log local0
    stats socket /run/haproxy/admin.sock mode 660 level admin
    user haproxy
    group haproxy
    daemon
defaults
    mode http
    option httplog
    option forwardfor
    timeout connect 5s
    timeout client  50s
    timeout server  50s
frontend http_front
    bind *:80
    default_backend http_back
backend http_back
    option httpchk GET /health
    default-server inter 2s fall 3 rise 2
    server local_vpn   10.8.0.2:80   check
    server maintenance 127.0.0.1:8080 backup
```

Fig. 6 HAProxy configuration on the VPS (/etc/haproxy/haproxy.cfg, abridged)

F. Testbed and Test Scenarios

To make the measurements repeatable, the topology was emulated on a single Ubuntu 24.04 host (Linux kernel 6.18, two virtual CPUs, 8 GB memory) with four Linux network namespaces: *user* (internet client), *vps* (HAProxy 2.8.30 and WireGuard server), *router* (home gateway), and *home* (WireGuard client and web server). The router masquerades the home subnet and drops every unsolicited inbound packet, emulating a subscriber line without a usable public address; its UDP connection-tracking timeouts were set to 30 s (unreplied) and 60 s (assured). Token-bucket shaping limited the home uplink to 50 Mbit/s and the VPS link to 100 Mbit/s. WireGuard ran as the userspace implementation `wireguard-go` because the host kernel did not provide the WireGuard module; the web server served a 25 kB CMS page and a 20 MB file. The three scenarios in Table II isolate the cost of the reverse proxy and of the tunnel.

TABLE II
TEST SCENARIOS

ID	Request path	Purpose
A	User → router (port forwarding) → web server	Baseline; needs public IP
C	User → HAProxy → router (port forwarding) → web	Cost of reverse proxy
B	User → HAProxy → WireGuard → web server	Proposed design

Six measurements were carried out: (1) reachability and exposed ports, using `curl` and `nmap`; (2) round-trip time with 200 ICMP probes; (3) TCP connect time, time to first byte (TTFB), and total time for 200 sequential HTTP requests per scenario; (4) request throughput with ApacheBench at 1–200 concurrent clients; (5) bulk TCP throughput with `iperf3` (three 10 s runs) and 20 MB HTTP downloads (five runs); and (6) backend failover and NAT keepalive behavior. The latency overhead of the tunnel and the throughput efficiency are defined as

$$\Delta T = T_B - T_A \tag{1}$$

$$\eta = (Th_{WG} / Th_{UL}) \times 100\% \tag{2}$$

where T_A and T_B are the mean TTFB of scenarios A and B, and Th_{WG} and Th_{UL} are the TCP throughput through the tunnel and over the underlay path. Because no propagation delay was emulated, absolute latencies represent processing and encapsulation overhead only; in a real deployment the round-trip time between the user, the VPS, and the home network is added to every value.

III. RESULTS AND DISCUSSION

A. Reachability and Attack Surface

Before the tunnel was used, an HTTP request from the internet client to the WAN address of the home router failed (no response), confirming that the local server could not be published directly. Through the VPS, the same CMS page was returned with HTTP 200. While the tunnel was active, `wg show` on the VPS reported the client endpoint as the translated address and port allocated by the home router, showing that the tunnel had traversed NAT without any forwarding rule. Table III summarizes the exposure observed by a TCP scan of ports 1–1024 and a UDP probe of port 51820.

TABLE III
EXPOSURE OF EACH HOST TO THE INTERNET

Target	Open TCP ports (1–1024)	HTTP from internet
Home router WAN	none (all closed)	no response
VPS public address	80 (HAProxy)	HTTP 200 (CMS)
VPS UDP 51820	open|filtered	silent to probes

The home network presents no listening service to the internet, and the only TCP service on the VPS is the HAProxy frontend. WireGuard appeared as *open|filtered* because it does not answer unauthenticated packets, a deliberate property of its design that makes the VPN port invisible to scanners [7]. Consequently, the attack surface is concentrated on a single, easily hardened host that can be rebuilt quickly, rather than on the local server that stores the institution's data.

B. Latency and Response Time

The mean ICMP round-trip time from the VPS to the home server was 0.078 ms on the underlay path and 0.686 ms through the tunnel, with no packet loss. Fig. 7 and Table IV compare the HTTP timing of the three scenarios. TCP connect time is almost identical because the client always connects to the nearest hop. The reverse proxy alone (C) added 0.17 ms to the TTFB, and the tunnel (B) added a further 0.69 ms, giving a total overhead of ΔT = 0.85 ms according to (1).

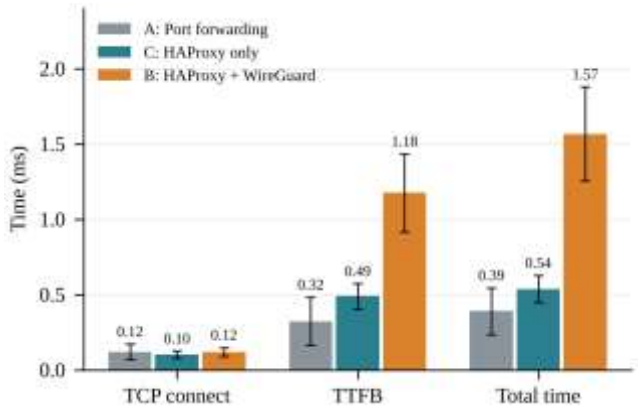


Fig. 7 Mean HTTP timing per scenario (200 requests; error bars show one standard deviation)

TABLE IV
HTTP RESPONSE TIME (N = 200 PER SCENARIO)

Scenario	Connect (ms)	TTFB mean (ms)	TTFB p95 (ms)	Total (ms)
A: Port fwd.	0.12	0.32	0.43	0.39
C: HAProxy	0.10	0.49	0.62	0.54
B: Proposed	0.12	1.18	1.70	1.57

In relative terms the overhead looks large, but in absolute terms it is below one millisecond. A typical round trip between an Indonesian subscriber and a domestic data center is in the order of tens of milliseconds, so the encryption, decryption, and extra forwarding hop are negligible compared with the path that the request travels anyway. The overhead is also bounded by the userspace implementation used in the testbed; the in-kernel implementation avoids copies between kernel and user space and is expected to be lower [16], [17]. The real latency cost of the design is therefore the geographic detour through the VPS, which is minimized by choosing a VPS in the same region as the users.

C. Request Throughput under Load

Fig. 8 presents the ApacheBench results. With one to fifty concurrent clients, scenarios A and C served about 234 requests per second, while the proposed design served 223, a reduction of 4.8%. At this level the 25 kB page saturates the 50 Mbit/s uplink, so the request rate is bounded by bandwidth rather than by HAProxy or WireGuard, and the small reduction corresponds to the bytes consumed by tunnel headers. No request failed in any run. At 100 and 200 concurrent clients all scenarios degraded similarly, and the 95th-percentile latency grew from about 5 ms to more than two seconds because requests queued behind the bottleneck link. This indicates that, for a home or campus server, the uplink capacity rather than the proxy or the tunnel determines how many visitors can be served.

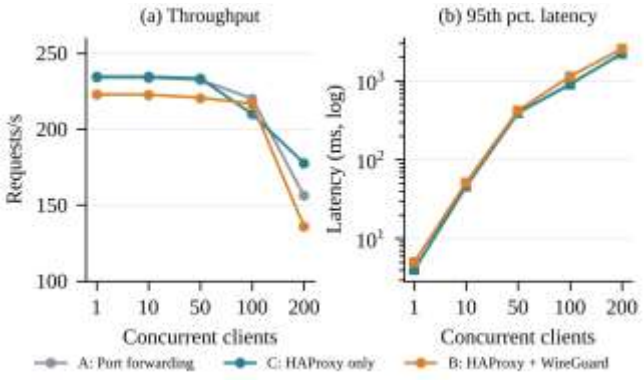


Fig. 8 ApacheBench results for 1–200 concurrent clients: (a) requests per second and (b) 95th-percentile latency

D. Bulk Throughput and Encapsulation Efficiency

With the 50 Mbit/s uplink, `iperf3` achieved 47.79 Mbit/s on the underlay and 45.79 Mbit/s through the tunnel (Fig. 9a), an efficiency of $\eta = 95.8\%$ according to (2). HTTP downloads of the 20 MB file showed the same pattern: 47.94 Mbit/s in A, 47.96 Mbit/s in C, and 45.69 Mbit/s in B (95.3% of A). The loss is explained entirely by encapsulation. Each tunnel packet carries 60 bytes of outer IPv4, UDP, and WireGuard header and tag, and the tunnel MTU is 1420 bytes, so the expected efficiency is

$$\eta_{\text{exp}} = (1380 / 1480) / (1460 / 1500) \approx 95.8\% \qquad (3)$$

where 1380 bytes is the TCP payload of a full packet inside the 1420-byte tunnel MTU, 1480 bytes is the size of that packet after encapsulation, and 1460 of 1500 bytes is the corresponding ratio on the underlay. The expected value agrees closely with the measurement. When shaping was removed (Fig. 9b), the underlay reached 15.26 Gbit/s and the tunnel 2.53 Gbit/s, limited by the single-threaded cryptographic processing of the userspace implementation on two virtual CPUs. This ceiling is still fifty times higher than a typical residential uplink, so the tunnel is not the bottleneck of the system.

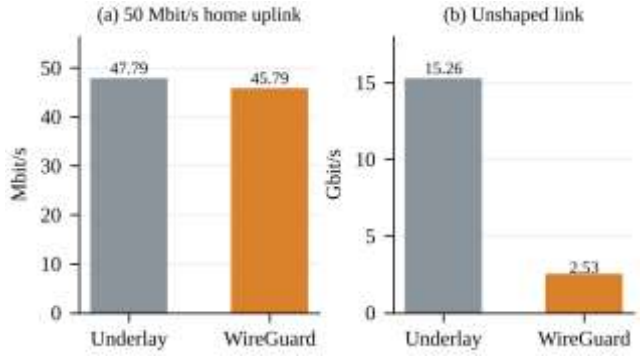


Fig. 9 TCP throughput over the underlay and through WireGuard: (a) with a 50 Mbit/s uplink and (b) without shaping

E. Resource Utilization

During an 18 s load test at 50 concurrent clients (about 221 requests per second and 45 Mbit/s), the mean CPU utilization was 29.1% for the WireGuard process on the VPS, 27.9% on the local server, and 12.4% for HAProxy, measured as a percentage of one core with `pidstat`. In the idle state all three processes used about 0.1% or less, because WireGuard only sends a keepalive every 25 s and HAProxy only sends health checks. These values confirm

that a single-core entry-level VPS is sufficient for the workload of a small institutional site, and that the local server retains most of its CPU capacity for the CMS and database.

F. Health Checking and Failover

The availability features of HAProxy were tested by stopping the web server on the local server for 12 s while a client polled the VPS ten times per second; the procedure was repeated five times (Fig. 10). HAProxy marked the backend as down after 6.45 ± 0.02 s and then served the maintenance page from the backup server, and it returned traffic to the local server 3.83 ± 0.06 s after the web server restarted. Both values agree with the configured parameters: three failed checks at a 2 s interval (6 s plus check timing) for detection and two successful checks for recovery. During the detection window, three requests per run stalled until the client timeout, because HAProxy still attempted to connect to the failed backend. This window can be shortened by reducing `inter`, adding `option redispatch` with `retries`, or enabling `observe layer7` so that live traffic errors also mark the server down.

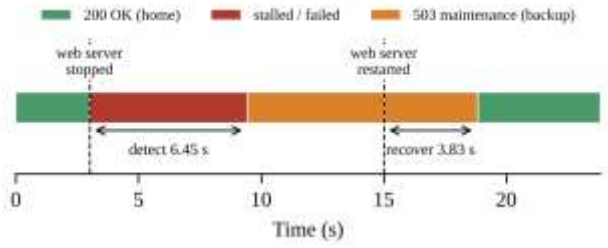


Fig. 10 Mean failover timeline over five runs when the local web server is stopped and restarted

G. NAT Keepalive and Tunnel Routing Mode

Because the home router only keeps a NAT mapping while traffic flows, the keepalive setting was examined by issuing requests after idle periods of 20, 45, and 90 s, longer than the 30 s and 60 s UDP timeouts of the router. Table V shows the results. Without `PersistentKeepalive`, the client never initiated a handshake after start-up, so the VPS did not know its endpoint; health checks failed, and visitors received only the maintenance page. With a 25 s keepalive, every request succeeded. When the tunnel was primed by a single packet from the local server but no keepalive was configured, the mapping was kept alive by the HAProxy health checks as long as they ran; after health checks were disabled, requests still succeeded after 20 and 45 s but timed out after 90 s, once the 60 s mapping had expired. The keepalive is therefore essential: it establishes the tunnel without manual intervention after a reboot and preserves the NAT mapping independently of application traffic.

TABLE V
REACHABILITY AFTER IDLE PERIODS

Configuration	Idle 20 s	Idle 45 s	Idle 90 s
No keepalive, not primed	HTTP 503	HTTP 503	HTTP 503
No keepalive, primed, health checks on	HTTP 200	HTTP 200	HTTP 200
No keepalive, primed, health checks off	HTTP 200	HTTP 200	Timeout
PersistentKeepalive = 25	HTTP 200	HTTP 200	HTTP 200

The routing mode of the client was also compared. With `AllowedIPs = 0.0.0.0/0` (full tunnel), all outbound traffic of the local server leaves through the VPS. In this mode the server remained reachable through HAProxy and from the local LAN, but a request arriving through any other path, such as a port-forwarding rule added for comparison, failed because the reply was routed into the tunnel instead of back through the home router. With `AllowedIPs = 10.8.0.0/24` (split tunnel), every path worked and only tunnel traffic was encrypted. For a server whose only purpose is to be published, the split tunnel is preferable: it avoids asymmetric routing, keeps software updates on the local uplink, and reduces the bandwidth billed on the VPS. The full tunnel is appropriate only when the local server must also appear on the internet with the VPS address for its outbound connections.

H. Comparison and Practical Implications

Table VI positions the design against common alternatives. Compared with port forwarding, it does not need a public address on the local line and exposes nothing on the home router. Compared with commercial tunnels and overlay VPN services, it keeps the control plane, certificates, and logs under the institution's control at the cost of operating one small VPS. Compared with a reverse SSH tunnel, it uses a connectionless, roaming-capable transport and adds layer-7 features such as health checks, header injection, and, if needed, load balancing across several local servers [29], [30].

TABLE VI
QUALITATIVE COMPARISON OF PUBLISHING APPROACHES

Approach	Public IP at home	Third-party control	Ports open at home	L7 health checks
Port forwarding	Yes	No	Yes	No
Reverse SSH tunnel	No	No	No	No
Commercial tunnel	No	Yes	No	Vendor
Overlay VPN service	No	Yes	No	No
HAProxy + WireGuard	No	No	No	Yes

Several limitations should be noted. The evaluation used an emulated topology without propagation delay and the userspace WireGuard implementation; absolute values will differ on physical hardware, although the relative overheads are governed by the same mechanisms. The reference configuration serves plain HTTP. Production deployments should terminate TLS on HAProxy with certificates obtained automatically through ACME [33], redirect HTTP to HTTPS, and follow current HTTP semantics for forwarding headers [34]. The VPS should restrict its firewall to TCP 80/443 and UDP 51820, disable password-based SSH, and enable the HAProxy statistics page only on the loopback interface. Finally, because the VPS is a single point of failure, a second VPS with its own tunnel can be added and published through DNS failover.

IV. CONCLUSION

This paper presented and evaluated an architecture that publishes a home or campus server through HAProxy on a

public VPS and an outbound WireGuard tunnel, so that the local network requires neither a public IPv4 address nor any open port. In an emulated testbed, the design made an unreachable server accessible while exposing no service on the home router, added only 0.85 ms of processing latency, retained 95.8% of TCP throughput in agreement with the theoretical encapsulation efficiency, and reduced the request rate by less than 5% on a bandwidth-limited uplink. HAProxy health checks provided automatic failover to a maintenance page in about 6.4 s and recovery in about 3.8 s, and a persistent keepalive proved necessary for reliable operation behind NAT. A split-tunnel configuration is recommended to avoid asymmetric routing. Future research should validate the results on physical hardware with the in-kernel WireGuard implementation and realistic wide-area latency, evaluate TLS termination and HTTP/2 or HTTP/3 on the proxy, add IPv6 and multiple VPS entry points for redundancy, and study the design as a hands-on learning platform for network administration courses.

ACKNOWLEDGMENT

The authors thank the Department of Information Technology, Politeknik Negeri Padang, for supporting this research.

REFERENCES

[1] J. Weil, V. Kuarsingh, C. Donley, C. Liljenstolpe, and M. Azinger, "IANA-reserved IPv4 prefix for shared address space," Internet Engineering Task Force, RFC 6598, Apr. 2012.

[2] S. Perreault, I. Yamagata, S. Miyakawa, A. Nakagawa, and H. Ashida, "Common requirements for carrier-grade NATs (CGNs)," Internet Engineering Task Force, RFC 6888, Apr. 2013.

[3] P. Richter et al., "A multi-perspective analysis of carrier-grade NAT deployment," in *Proc. Internet Measurement Conf. (IMC)*, Santa Monica, CA, USA, 2016, pp. 215–229, doi: 10.1145/2987443.2987474.

[4] Asosiasi Penyelenggara Jasa Internet Indonesia (APJII), "Indonesia internet penetration survey 2024," APJII, Jakarta, Indonesia, 2024. [Online]. Available: https://survei.apjii.or.id

[5] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, "Zero trust architecture," National Institute of Standards and Technology, Gaithersburg, MD, USA, NIST SP 800-207, Aug. 2020, doi: 10.6028/NIST.SP.800-207.

[6] M. Petit-Huguenin, G. Salgueiro, J. Rosenberg, D. Wing, R. Mahy, and P. Matthews, "Session traversal utilities for NAT (STUN)," Internet Engineering Task Force, RFC 8489, Feb. 2020.

[7] J. A. Donenfeld, "WireGuard: Next generation kernel network tunnel," in *Proc. Network and Distributed System Security Symp. (NDSS)*, San Diego, CA, USA, 2017, doi: 10.14722/ndss.2017.23160.

[8] T. Perrin, "The Noise protocol framework," rev. 34, Jul. 2018. [Online]. Available: https://noiseprotocol.org/noise.html

[9] A. Langley, M. Hamburg, and S. Turner, "Elliptic curves for security," Internet Engineering Task Force, RFC 7748, Jan. 2016.

[10] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF protocols," Internet Engineering Task Force, RFC 8439, Jun. 2018.

[11] B. Dowling and K. G. Paterson, "A cryptographic analysis of the WireGuard protocol," in *Applied Cryptography and Network Security (ACNS 2018)*, Lecture Notes in Computer Science, vol. 10892. Cham, Switzerland: Springer, 2018, pp. 3–21, doi: 10.1007/978-3-319-93387-0_1.

[12] B. Lipp, B. Blanchet, and K. Bhargavan, "A mechanised cryptographic proof of the WireGuard virtual private network protocol," in *Proc. IEEE European Symp. Security and Privacy (EuroS&P)*, Stockholm, Sweden, 2019, pp. 231–246, doi: 10.1109/EuroSP.2019.00026.

[13] A. Hülsing, K.-C. Ning, P. Schwabe, F. Weber, and P. R. Zimmermann, "Post-quantum WireGuard," in *Proc. IEEE Symp. Security and Privacy (SP)*, San Francisco, CA, USA, 2021, pp. 304–321, doi: 10.1109/SP40001.2021.00030.

[14] HAProxy Technologies, "HAProxy 2.8 configuration manual," 2023. [Online]. Available: https://docs.haproxy.org/2.8/configuration.html

[15] S. Mackey, I. Mihov, A. Nosenko, F. Vega, and Y. Cheng, "A performance comparison of WireGuard and OpenVPN," in *Proc. 10th ACM Conf. Data and Application Security and Privacy (CODASPY)*, New Orleans, LA, USA, 2020, pp. 162–164, doi: 10.1145/3374664.3379532.

[16] E. Dekker and P. Spaans, "Performance comparison of VPN implementations WireGuard, strongSwan, and OpenVPN in a 1 Gbit/s environment," Univ. of Amsterdam, Amsterdam, The Netherlands, Research Project Rep., 2020. [Online]. Available: https://rp.os3.nl/2019-2020/p71/report.pdf

[17] M. Pudelko, P. Emmerich, S. Gallenmüller, and G. Carle, "Performance analysis of VPN gateways," in *Proc. IFIP Networking Conf.*, Paris, France, 2020, pp. 325–333.

[18] J. Anyam, R. R. Singh, H. Larijani, and A. Philip, "Empirical performance analysis of WireGuard vs. OpenVPN in cloud and virtualised environments under simulated network conditions," *Computers*, vol. 14, no. 8, Aug. 2025, Art. no. 326, doi: 10.3390/computers14080326.

[19] T. Demirdelen and S. Kırmızı, "Performance evaluation of WireGuard and IPSec protocols in various network configurations," *Osmaniye Korkut Ata Univ. J. Inst. Sci. Technol.*, vol. 8, no. 3, pp. 1353–1369, 2025, doi: 10.47495/okufbed.1660352.

[20] H. Djuitcheu, A. Sergeev, K. Alam, D. Santhosh, A. Autenrieth, and J. Seitz, "Lightweight security for private networks: Real-world evaluation of WireGuard," arXiv:2512.10135, Dec. 2025.

[21] S. G. Hussein and S. M. F. U. Rehman, "Protocol efficiency and resource utilization in VPN technologies: A comparative analysis of OpenVPN and WireGuard," *J. Techniques*, vol. 7, no. 4, pp. 37–42, Dec. 2025, doi: 10.51173/jt.v7i4.2768.

[22] H. Jumakhan and A. Mirzaeinia, "WireGuard: An efficient solution for securing IoT device connectivity," arXiv:2402.02093, Feb. 2024.

[23] D.-F. Hritcan, A. Graur, D. G. Balan, and E. M. Timofte, "A performance analysis of VPN technologies used in an IoT environment," *Adv. Electr. Comput. Eng.*, vol. 25, no. 2, pp. 81–88, 2025, doi: 10.4316/AECE.2025.02009.

[24] M. Barone, D. Miola, F. Parola, and F. Risso, "Revisiting WireGuard for line-rate, scalable tunneling," in *Proc. 26th IEEE Int. Conf. High Performance Switching and Routing (HPSR)*, Suita, Osaka, Japan, 2025, pp. 1–6, doi: 10.1109/HPSR64165.2025.11038910.

[25] H. Abbas et al., "Security assessment and evaluation of VPNs: A comprehensive survey," *ACM Comput. Surv.*, vol. 55, no. 13, pp. 1–47, 2023.

[26] D. Novianto, Y. S. Japriadi, and L. Tommy, "Implementation of website access security using WireGuard VPN on a MikroTik routerboard," *J. Ilm. Inform. Global*, vol. 13, no. 2, Aug. 2022, doi: 10.36982/jiig.v13i2.2308 (in Indonesian).

[27] L. Otaviana S. et al., "Implementation of WireGuard VPN for securing the Sahabat Net network server against sniffing attacks," *ZONAsi: J. Sist. Inf.*, vol. 7, no. 3, p. 1103, Sep. 2025 (in Indonesian).

[28] R. Hermawan and Y. M. Saputra, "Comparative analysis of Cloud VPN and WireGuard VPN tunneling in a hybrid cloud infrastructure," *J. Internet Softw. Eng. (JISE)*, vol. 6, no. 1, pp. 1–12, 2025, doi: 10.22146/jise.v6i1.14450 (in Indonesian).

[29] C. Rawls and M. A. Salehi, "Load balancer tuning: Comparative analysis of HAProxy load balancing methods," arXiv:2212.14198, Dec. 2022, doi: 10.48550/arXiv.2212.14198.

[30] T. M. F. Rifat and U. Y. Oktiawati, "Performance analysis of weighted round robin and weighted least connection algorithms in a web server load balancing system with HAProxy integrated with Cacti monitoring," *J. Internet Softw. Eng. (JISE)*, vol. 6, no. 1, pp. 60–68, 2025, doi: 10.22146/jise.v6i1.15494 (in Indonesian).

[31] A. Wirawan, R. Gatra, H. Hidayat, and D. Prasetyawan, "Implementation of load balancing with HAProxy in the academic information system of UIN Sunan Kalijaga," *JISKA (J. Inform. Sunan Kalijaga)*, vol. 9, no. 1, pp. 39–49, 2024 (in Indonesian).

[32] A. Frihadi, S. Windasari, M. Dama, B. Bagaskoro, and A. Abdurohman, "Improved website security using SSL and HAProxy based pfSense routers," *Teknik: J. Ilmu Tek. Inform.*, vol. 5, no. 1, pp. 107–115, May 2025, doi: 10.51903/teknik.v5i1.888.

[33] J. Aas et al., “Let’s Encrypt: An automated certificate authority to encrypt the entire web,” in *Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS)*, London, U.K., 2019, pp. 2473–2487, doi: 10.1145/3319535.3363192.

[34] R. Fielding, M. Nottingham, and J. Reschke, “HTTP semantics,” Internet Engineering Task Force, RFC 9110, Jun. 2022.