University of Rome Tor Vergata
ICT and Internet Engineering

Network and System Defense

Alessandro Pellegrini, Angelo Tulumello

A.A. 2023/2024

Lecture 7: Overlay VPNs - OpenVPN and IPSEC

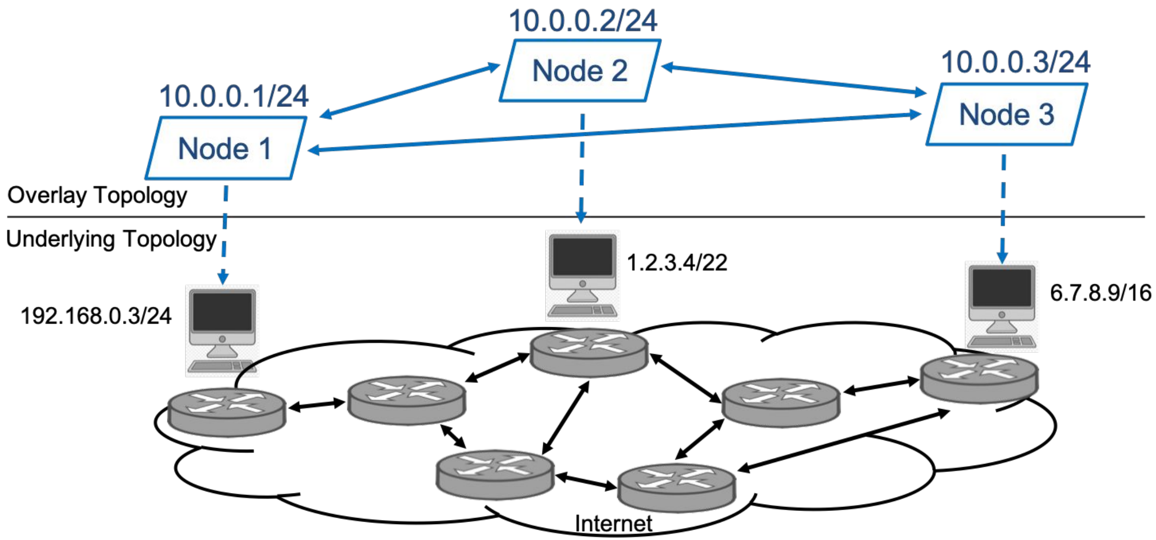
Angelo Tulumello

Virtual Private Network

- A ***virtual private network (VPN)*** extends a private network across a public network
- It enables users to send and receive data across shared or public networks ***as if their computing devices were directly connected to the private network***
- The benefits of a VPN increases the functionality, security, and management of the private network
- It provides access to resources inaccessible on the public network and is typically used for telecommuting workers
- Encryption/integrity is common, although not an inherent part of a VPN connection
  - for example, in intra-AS VPNs encryption/integrity is not enforced...

Overlay VPN

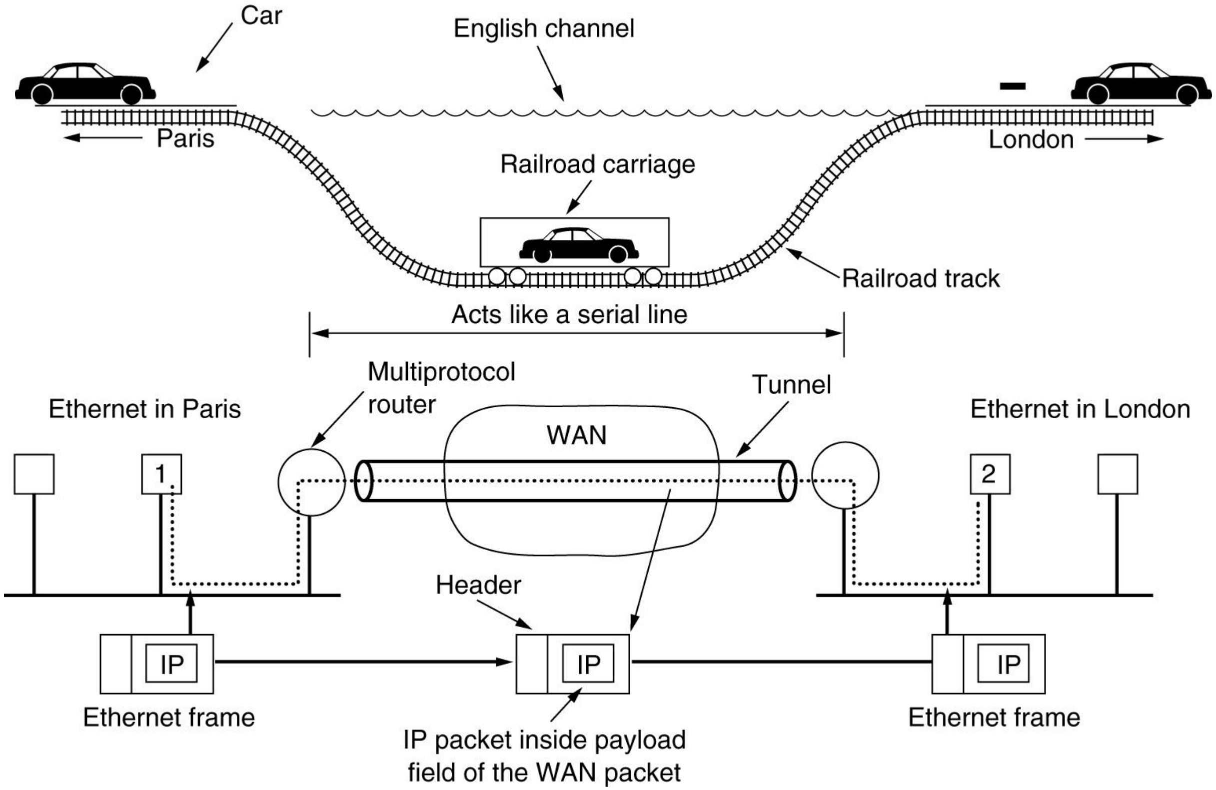
- In general it is a VPN whose topology is built ***on top of another lower (insecure) topology*** which is usually not managed by the same entity
- An overlay network is a computer network that is layered on top of another network
- In the most common scenario, overlay ***VPN are built on top of the public Internet***



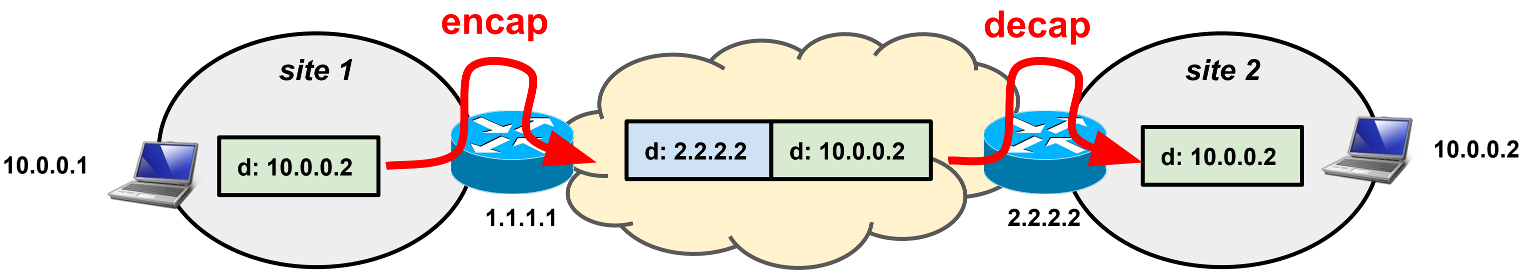
Problem 1

- *How can I realize a (private) network consisting of nodes physically deployed in other independent networks?*
  - (in the previous slide) how can I pretend that node1,2,3 are all in the same (private) 10.0.0.0/24 network?
  - how can I deliver a packet to a destination which is unreachable for the underlying routers?

Virtual Networks → Tunnels



Virtual Networks → Tunnels

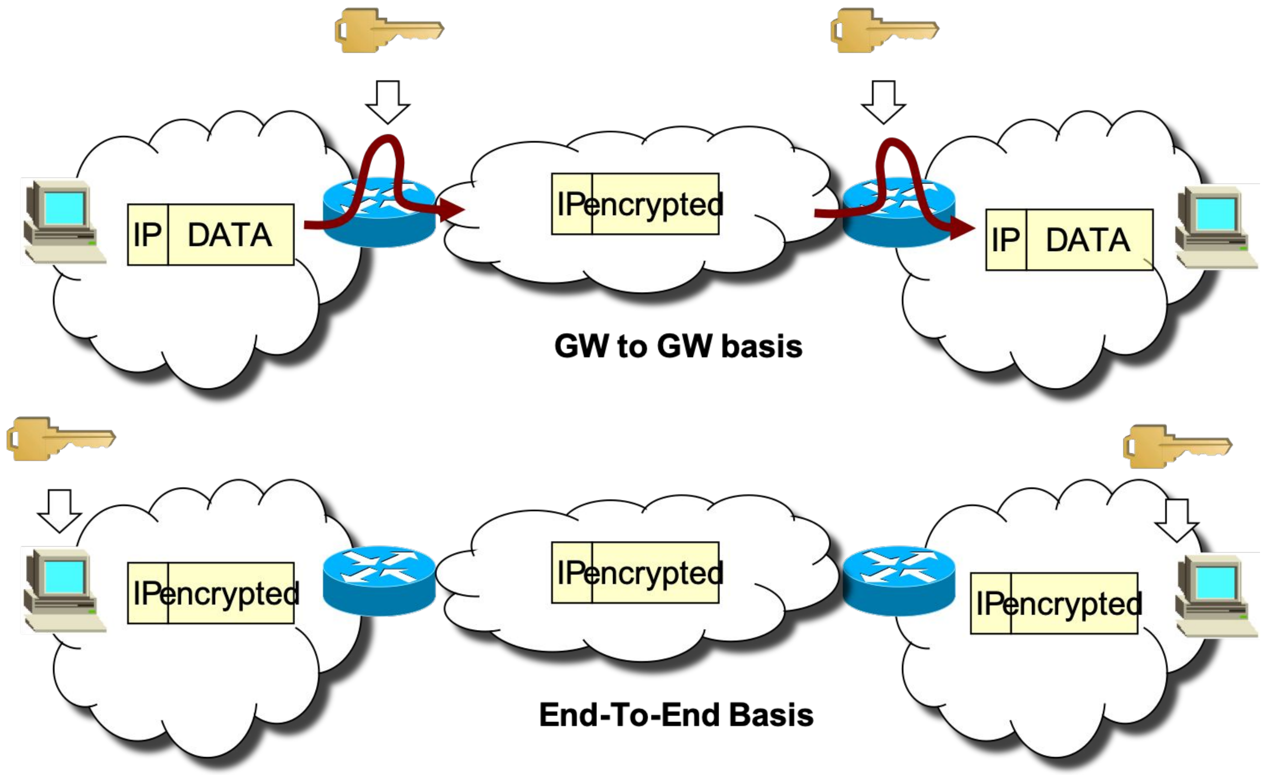


- IP in IP tunnels are not really used for this purpose…
- In this example the tunnel is between the 2 access routers
  - Tunnels can be terminated on the end nodes (if they are reachable)
- In anycase, the outer packet hides the inner packet
- The underlying routers never see the final destination
- Overlay routing requirements
  - R1 needs to know that 10.0.0.2 can be reached through a tunnel to 2.2.2.2
  - Node 1 needs to know that to talk with 10.0.0.2 needs to send te packet to R1
- ***Note that a combination of IP forwarding magic and static/dynamic NATs would solve problem 1 but not the next one…***

Problem 2

- *How do I provide the required security mechanisms (confidentiality, authentication, integrity and authorization) on top of an insecure network?*
- Since it is build on top of an insecure infrastructure, its security is realized with end-2-end mechanisms
  - They can terminate on end devices (e.g. a server, a laptop)
  - or on security gateways that can ”connect” other devices to the VPN
- The main tools for realizing overlay VPNs (over the internet) are secure tunneling protocols
  - TLS, SSH, IPsec, etc..

Private Networks: Encryption + Authentication

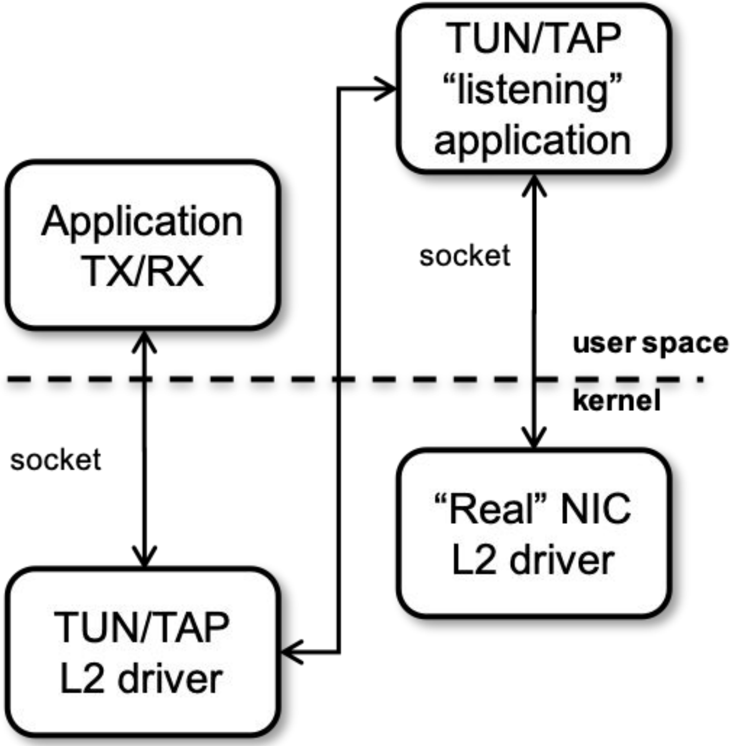


TLS user space VPNs with OpenVPN

OpenVPN (openvpn.net)

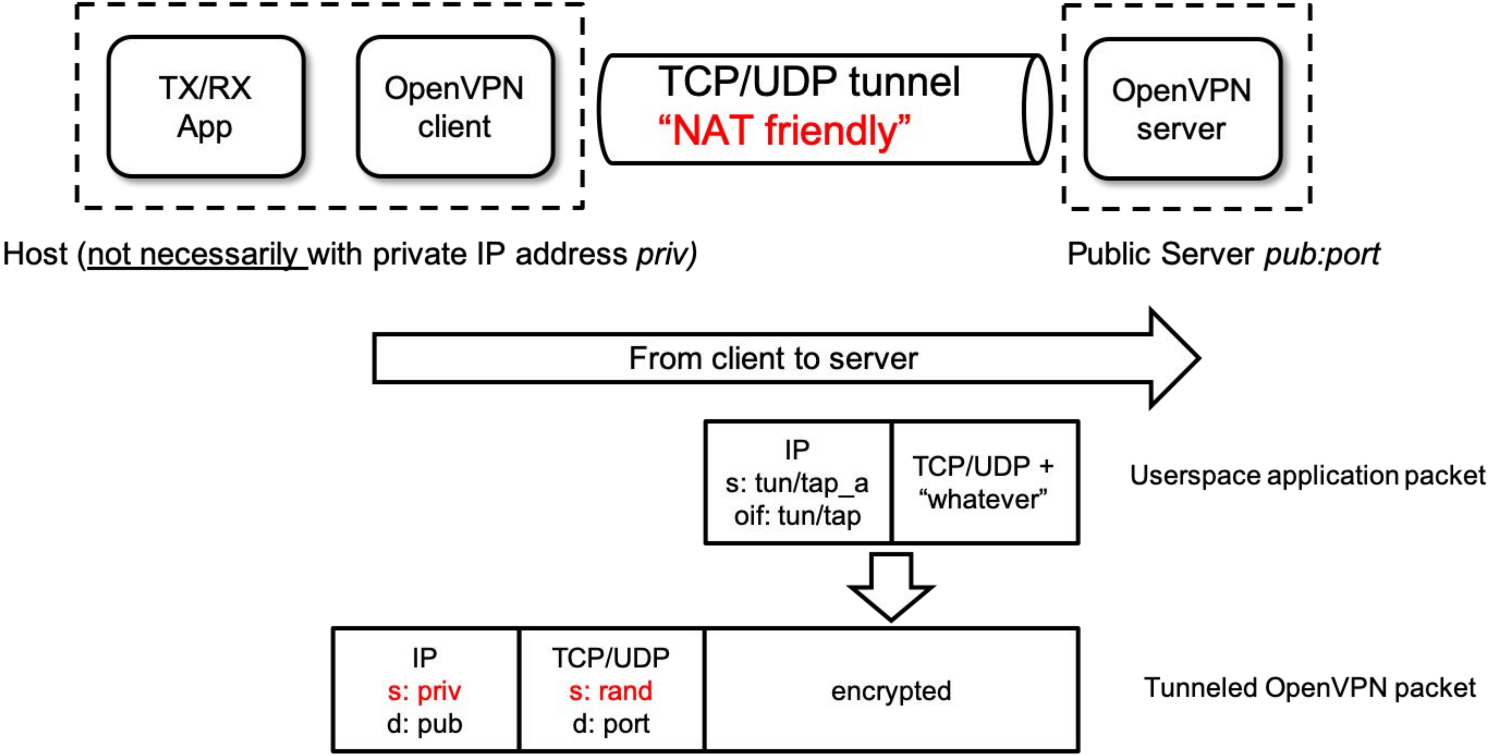
- Tunnel any IP subnetwork or virtual ethernet adapter over a single UDP or TCP port
- configure a scalable, load-balanced VPN server farm using one or more machines which can handle thousands of dynamic connections from incoming VPN clients
- use all of the encryption, authentication, and certification features of the OpenSSL library to protect your private network traffic as it transits the internet
- use any cipher, key size, or HMAC digest (for datagram integrity checking) supported by the OpenSSL library
- choose between static-key based conventional encryption or certificate-based public key encryption
- use static, pre-shared keys or TLS-based dynamic key exchange
- use real-time adaptive link compression and traffic-shaping to manage link bandwidth utilization
- tunnel networks whose public endpoints are dynamic such as DHCP or dial-in clients
- tunnel networks through connection-oriented stateful firewalls without having to use explicit firewall rules
- tunnel networks over NAT
- create secure ethernet bridges using virtual tap devices

TUN/TAP drivers



- ❏ TUN is a virtual Point-to-Point network device for **IP tunneling**
- ❏ TAP is a virtual Ethernet network device for **Ethernet tunneling**
- ❏ Userland application can write {IP|Ethernet} frame to /dev/{tun|tap}X and kernel will receive this frame from {tun|tap}X interface
- ❏ In the same time every frame that kernel writes to {tun|tap}X interface can be read by userland application from {tun|tap}X device

TUN/TAP drivers



ip.s and L4.s may be NATed

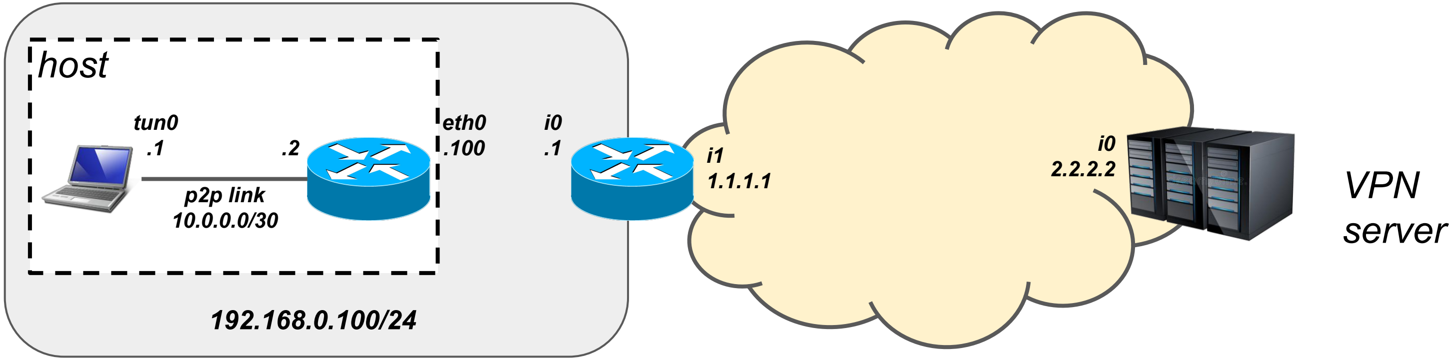
OpenVPN host to server with TUN virtual interfaces

Host Configuration

```
tun0: 10.0.0.1/30
VPN address range: 10.0.0.0/24
eth0: 192.168.0.100/24
```

Host Routing Table

10.0.0.2/32	*	tun0
10.0.0.0/24	10.0.0.2	tun0
192.168.0.1/24	*	eth0
0.0.0.0/0	192.168.0.1	eth0



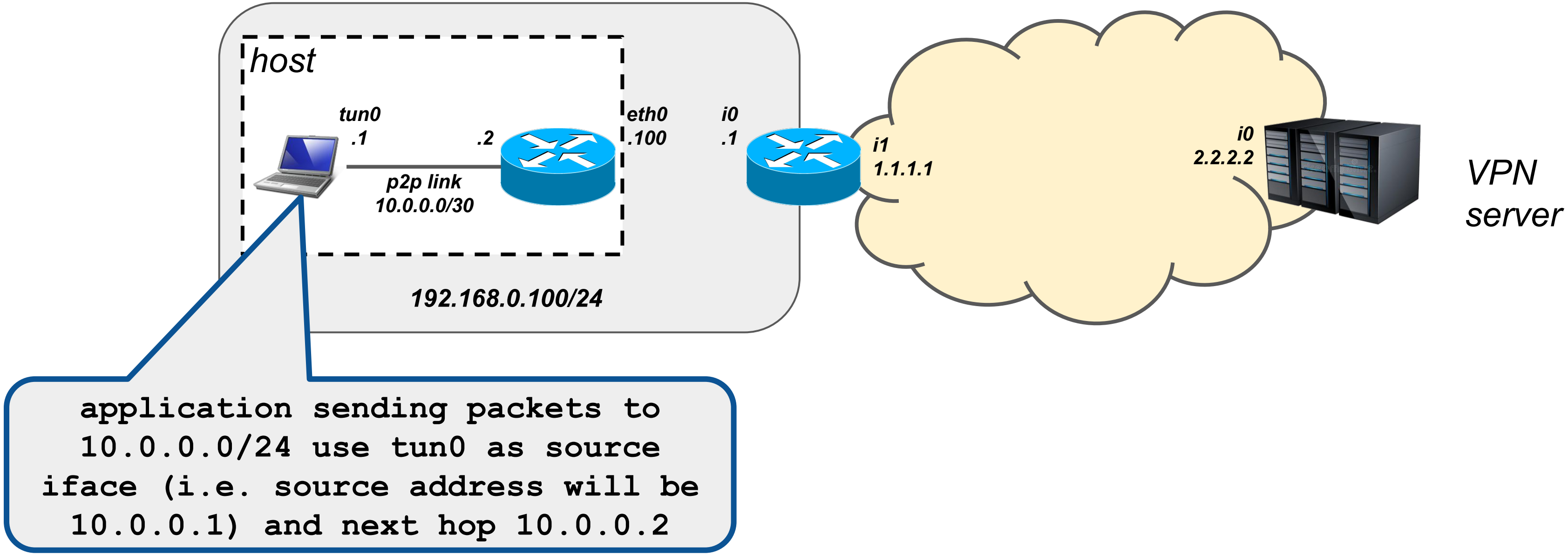
OpenVPN host to server with TUN virtual interfaces

Host Configuration

```
tun0: 10.0.0.1/30
VPN address range: 10.0.0.0/24
eth0: 192.168.0.100/24
```

Host Routing Table

10.0.0.2/32	*	tun0
10.0.0.0/24	10.0.0.2	tun0
192.168.0.1/24	*	eth0
0.0.0.0/0	192.168.0.1	eth0

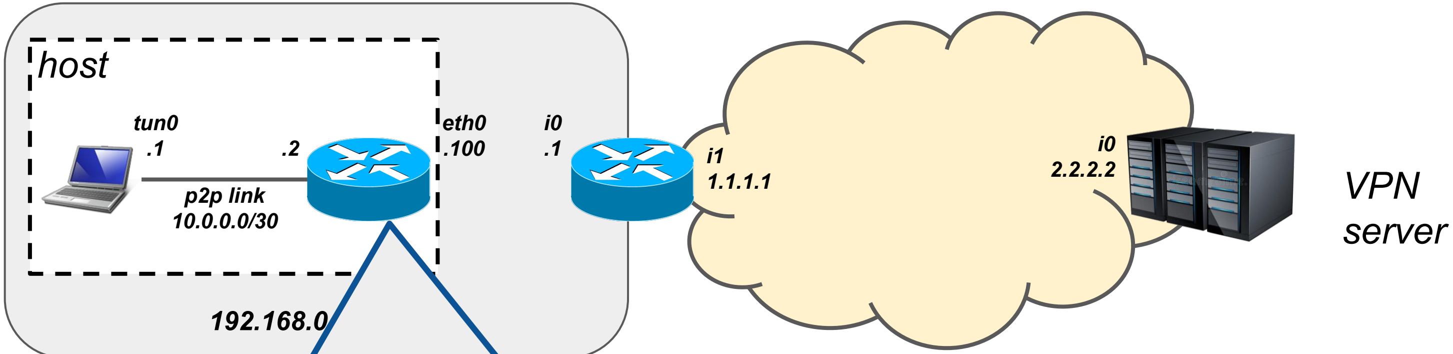


OpenVPN host to server with TUN virtual interfaces

Host Configuration
tun0: 10.0.0.1/30
VPN address range: 10.0.0.0/24
eth0: 192.168.0.100/24

Host Routing Table

10.0.0.2/32	*	tun0
10.0.0.0/24	10.0.0.2	tun0
192.168.0.1/24	*	eth0
0.0.0.0/0	192.168.0.1	eth0



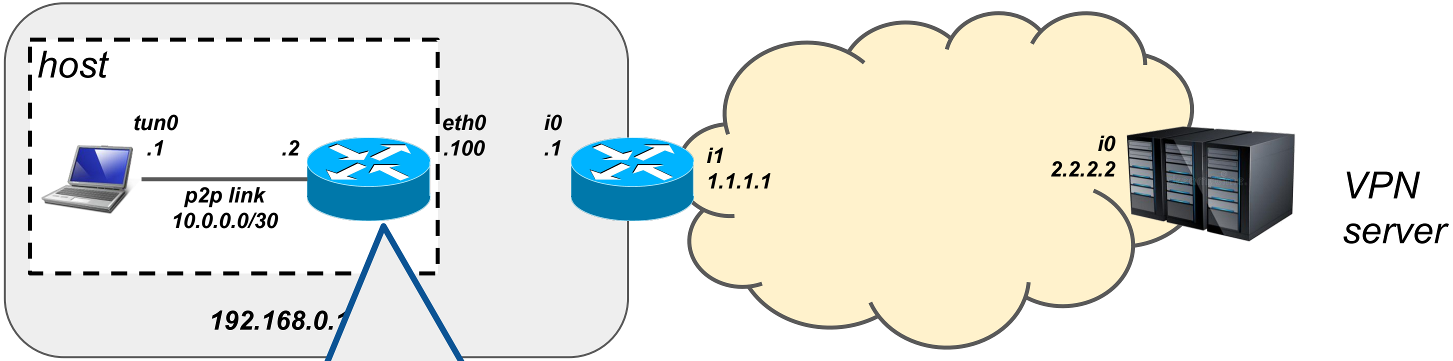
packets received by the VPN next hop (which is a "virtual node" running locally on the host) are encapsulated within a D-TLS tunnel previously established with the VPN server

OpenVPN host to server with TUN virtual interfaces

Host Configuration
tun0: 10.0.0.1/30
VPN address range: 10.0.0.0/24
eth0: 192.168.0.100/24

Host Routing Table

10.0.0.2/32	*	tun0
10.0.0.0/24	10.0.0.2	tun0
192.168.0.1/24	*	eth0
0.0.0.0/0	192.168.0.1	eth0



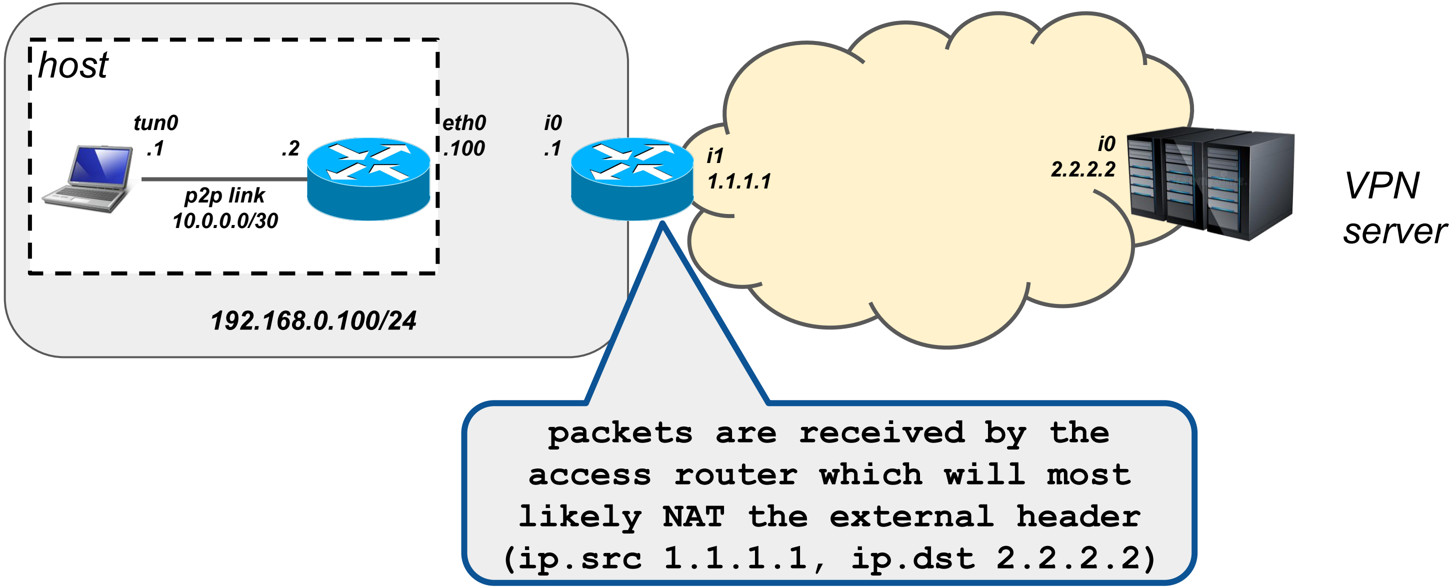
packets are then sent to the real next hop (e.g. the default GW) through the physical interface of the host

OpenVPN host to server with TUN virtual interfaces

Host Configuration
```
tun0: 10.0.0.1/30
VPN address range: 10.0.0.0/24
eth0: 192.168.0.100/24
```

Host Routing Table

10.0.0.2/32	*	tun0
10.0.0.0/24	10.0.0.2	tun0
192.168.0.1/24	*	eth0
0.0.0.0/0	192.168.0.1	eth0

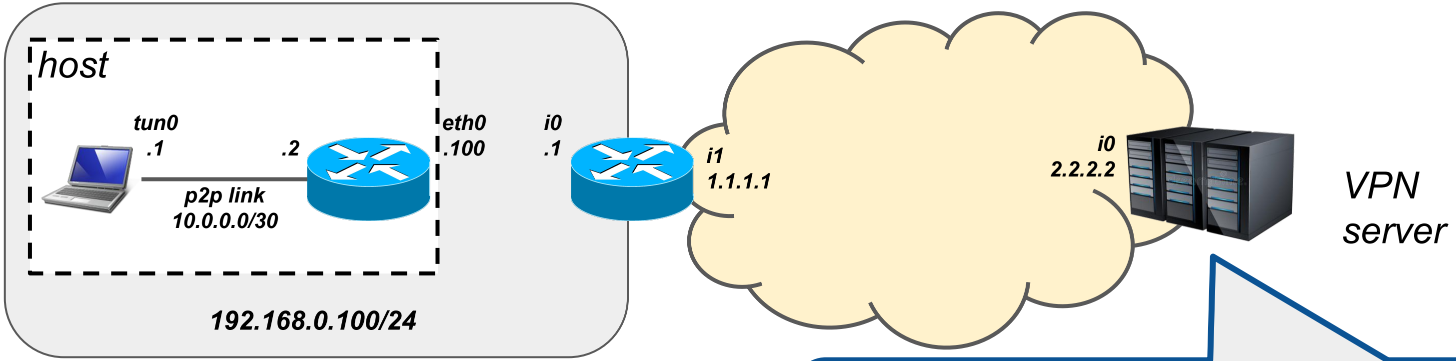


OpenVPN host to server with TUN virtual interfaces

Host Configuration
tun0: 10.0.0.1/30
VPN address range: 10.0.0.0/24
eth0: 192.168.0.100/24

Host Routing Table

10.0.0.2/32	*	tun0
10.0.0.0/24	10.0.0.2	tun0
192.168.0.1/24	*	eth0
0.0.0.0/0	192.168.0.1	eth0



packets are eventually received by the OpenVPN Server. The inner packets is decapsulated and forwarded like in the host

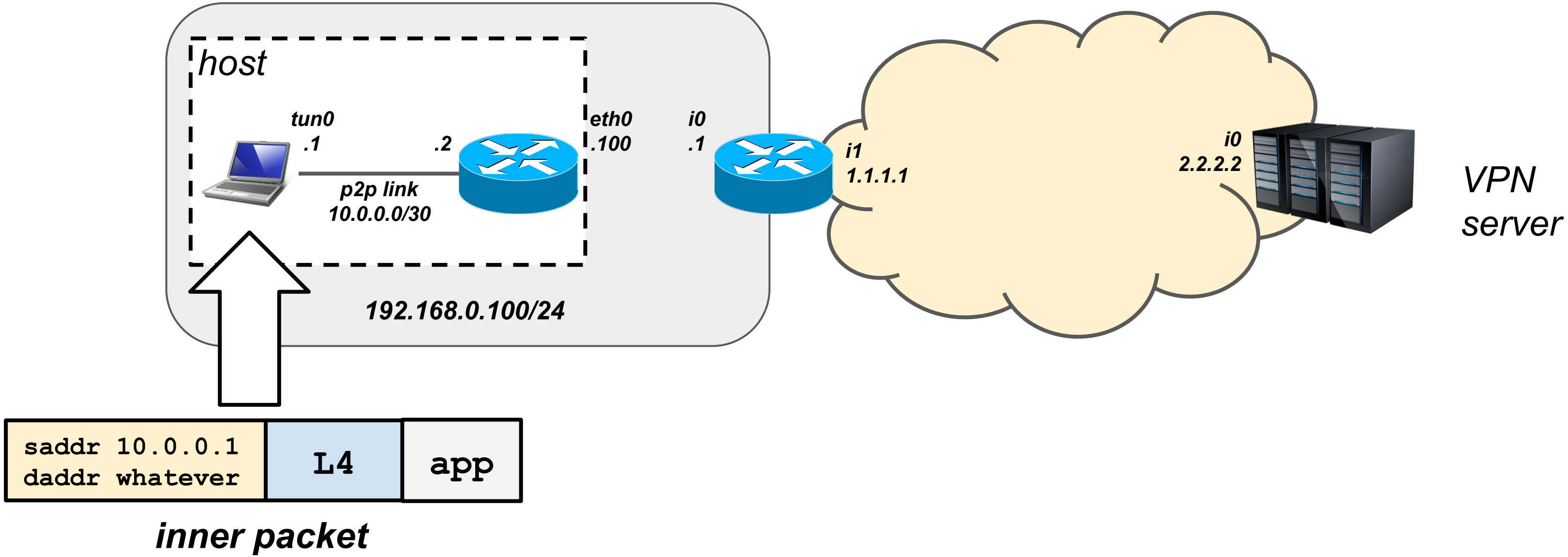
OpenVPN host to server with TUN virtual interfaces

Host Configuration

```
tun0: 10.0.0.1/30
VPN address range: 10.0.0.0/24
eth0: 192.168.0.100/24
```

Host Routing Table

10.0.0.2/32	*	tun0
10.0.0.0/24	10.0.0.2	tun0
192.168.0.1/24	*	eth0
0.0.0.0/0	192.168.0.1	eth0



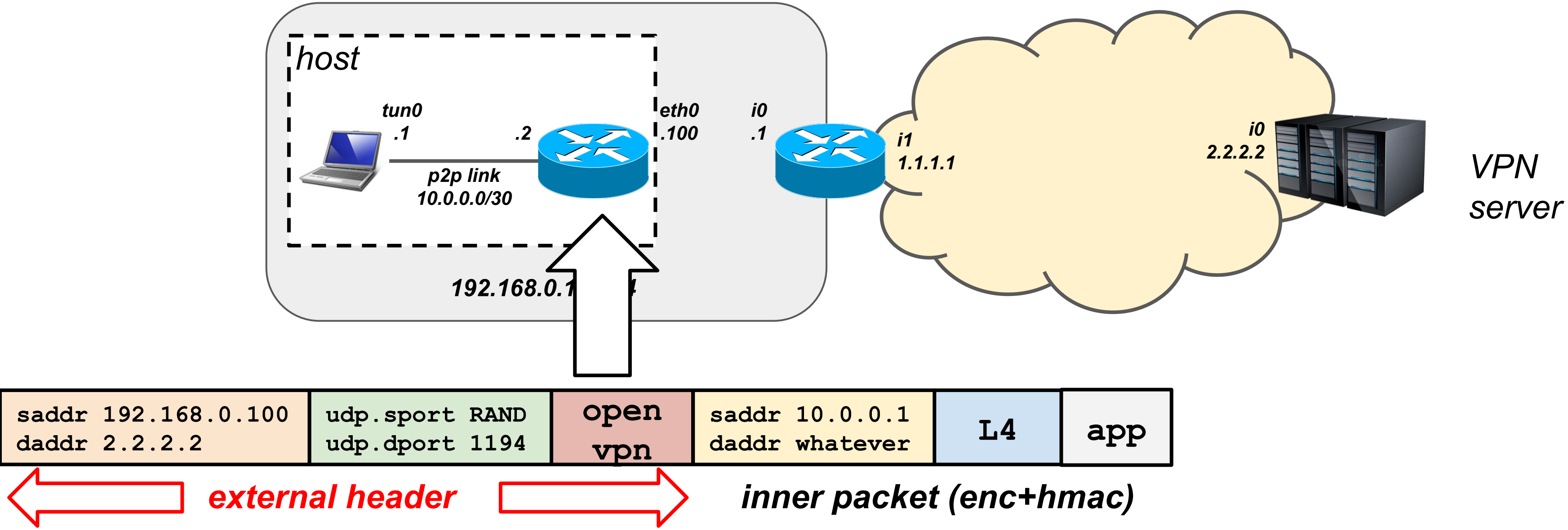
OpenVPN host to server with TUN virtual interfaces

Host Configuration

tun0: 10.0.0.1/30
VPN address range: 10.0.0.0/24
eth0: 192.168.0.100/24

Host Routing Table

10.0.0.2/32	*	tun0
10.0.0.0/24	10.0.0.2	tun0
192.168.0.1/24	*	eth0
0.0.0.0/0	192.168.0.1	eth0



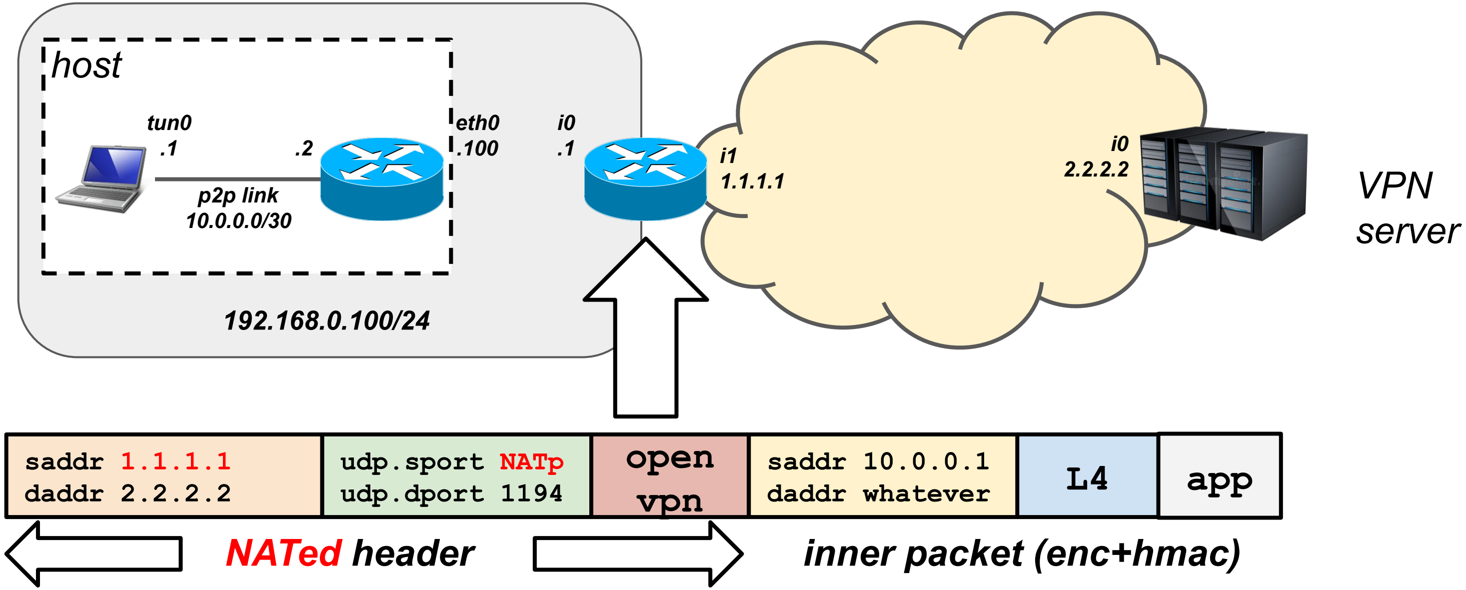
OpenVPN host to server with TUN virtual interfaces

Host Configuration

```
tun0: 10.0.0.1/30
VPN address range: 10.0.0.0/24
eth0: 192.168.0.100/24
```

Host Routing Table

10.0.0.2/32	*	tun0
10.0.0.0/24	10.0.0.2	tun0
192.168.0.1/24	*	eth0
0.0.0.0/0	192.168.0.1	eth0



OpenVPN PKI

- OpenVPN (in tls-mode) uses TLS for protecting the control channel
- The first step in building an OpenVPN 2.0 configuration is to establish a PKI which consists of:
  - a **separate certificate** (also known as a public key) and private key for the server and each client
  - a **master Certificate Authority (CA)** certificate and key which is used to sign each of the server and client certificates
- OpenVPN supports bidirectional authentication based on certificates, meaning that the client must authenticate the server certificate and the server must authenticate the client certificate before mutual trust is established
- Both server and client will authenticate the other by first verifying that the presented certificate was signed by the master certificate authority (CA), and then by testing information in the now-authenticated certificate header, such as the certificate common name or certificate type (client or server)

OpenVPN security model

- This security model has a number of desirable features from the VPN perspective:
  - The server **only needs its own certificate/key** -- it doesn't need to know the individual certificates of every client which might possibly connect to it.
  - The server will only **accept clients whose certificates were signed by the master CA** certificate (which we will generate below).
    - And because the server can perform this signature verification **without needing access** to the CA private key itself, it is possible for the CA key (the most sensitive key in the entire PKI) to reside on a completely different machine, even one without a network connection.
  - If a private key is compromised, it can be disabled by adding its certificate to a **CRL** (certificate revocation list). The CRL allows compromised certificates to be selectively rejected without requiring the entire PKI being rebuilt.
  - The server can enforce **client-specific access rights** based on embedded certificate fields, such as the Common Name.

The OpenVPN protocol

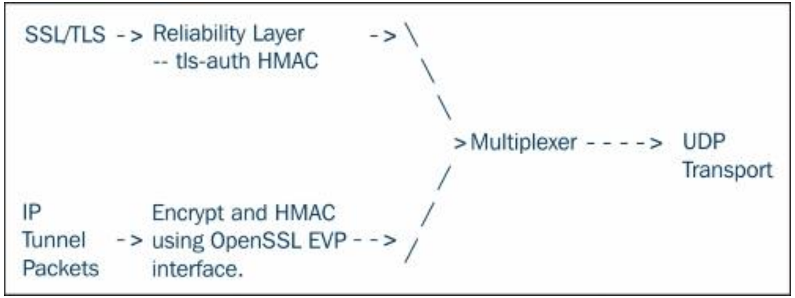
- OpenVPN currently supports two ways to communicate between endpoints: using ***UDP*** packets or using ***TCP*** packets.
- Both modes of communication have their advantages and disadvantages
  - Using a TCP-based application over a TCP-based VPN can result in double performance loss, especially if the underlying network connection is bad
    - Nice article: ***"Why TCP over TCP is a Bad Idea"*** http://sites.inka.de/~W1011/devel/tcp-tcp.html
  - It can be similarly argued that sending UDP over UDP may also be not a good idea
    - Even Though I personally think that if the application requires a connectionless transport layer, why should we have TCP in the underlying layer?
  - The general rule of thumb is as follows: if UDP (mode udp) works for you, then use it; if not, then try TCP

The OpenVPN protocol

- When UDP mode is used, OpenVPN implements TLS over UDP
- OpenVPN implements a ***client/server protocol***
- OpenVPN uses ***two virtual channels*** to communicate between the client and server:
  - A ***TLS control channel*** to exchange configuration information and cipher material between the client and server.
    - This channel is used mostly when the VPN connection is started, as well as for exchanging new encryption keying material and sending keep alives
  - ***A data channel*** over which the (rawly) encrypted payload is exchanged
- The exception to this is the older pre-shared key point-to-point mode, in which only the data channel is used.

The OpenVPN protocol

- The ***control channel*** is initiated using a TLS-style protocol, similar to how a secure website connection is initiated. During control channel initialization, the encryption cipher and hashing algorithm are negotiated between the client and server.
- Encryption and authentication algorithms for the ***data channel*** are not negotiable, but they are set in both the client and server configuration files for OpenVPN



in UDP mode OpenVPN needs to implement a reliability layer (explicit packet acknowledgement)

The OpenVPN protocol

OpenVPN header format

- ***Packet length*** (16 bits, unsigned) -- TCP only, always sent as plaintext. Since TCP is a stream protocol, the packet length words define the packetization of the stream.
- ***Packet opcode/key_id*** (8 bits) -- TLS only, not used in pre-shared secret mode.
  - packet message type, a P_* constant (high 5 bits)
    - P_CONTROL_HARD_RESET_CLIENT, P_CONTROL_HARD_RESET_SERVER P_CONTROL_SOFT_RESET, P_CONTROL, P_ACK, P_DATA
    - _RESET_* messages *_V1 and *_V2 (depending on the key method)
  - key_id (low 3 bits, see key_id in struct tls_session below for comment). The key_id refers to an already negotiated TLS session. OpenVPN seamlessly renegotiates the TLS session by using a new key_id for the new session. Overlap (controlled by user definable parameters) between old and new TLS sessions is allowed, providing a seamless transition during tunnel operation.
- ***Payload*** (n bytes), which may be a P_CONTROL, P_ACK, or P_DATA message.

The OpenVPN protocol

P_CONTROL message format

- local session_id (random 64 bit value to identify TLS session).
- HMAC signature of entire encapsulation header for integrity
- P_ACK packet_id array length (1 byte).
- P_ACK packet-id array (if length > 0).
- P_ACK remote session_id (if length > 0)
- message packet-id (4 bytes).
- TLS payload ciphertext (n bytes) (only for P_CONTROL).

Once the TLS session has been initialized and authenticated, the TLS channel is used to exchange random key material for bidirectional cipher and HMAC keys which will be used to secure actual tunnel packets.

OpenVPN currently implements ***two key methods***

- ***Key method 1*** directly derives keys using random bits obtained from the RAND_bytes OpenSSL function.
- ***Key method 2*** mixes random key material from both sides of the connection using the TLS PRF mixing function. Key method 2 is the preferred method and is the defaultc for OpenVPN 2.0.

The OpenVPN protocol

The ***P_DATA*** payload represents encrypted, encapsulated tunnel packets which tend to be either IP packets or Ethernet frames. This is essentially the "payload" of the VPN.

P_DATA message content:

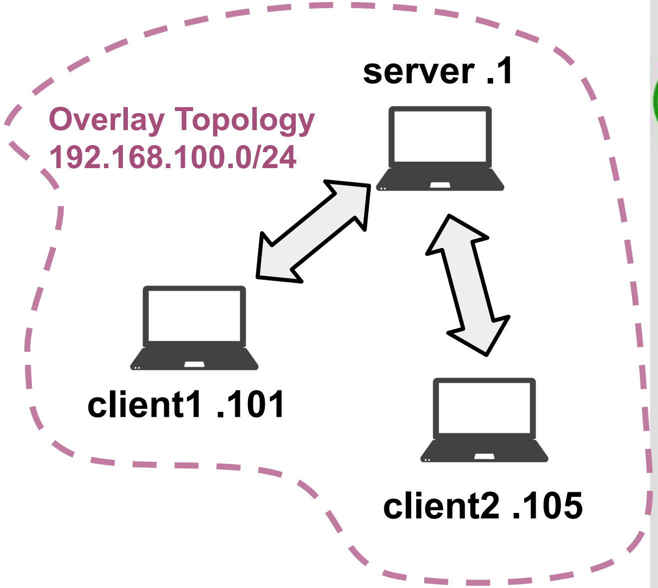
- HMAC of ciphertext IV + ciphertext (if not disabled by --auth none).
- Ciphertext IV (size is cipher-dependent, if not disabled by --no-iv).
- Tunnel packet ciphertext.

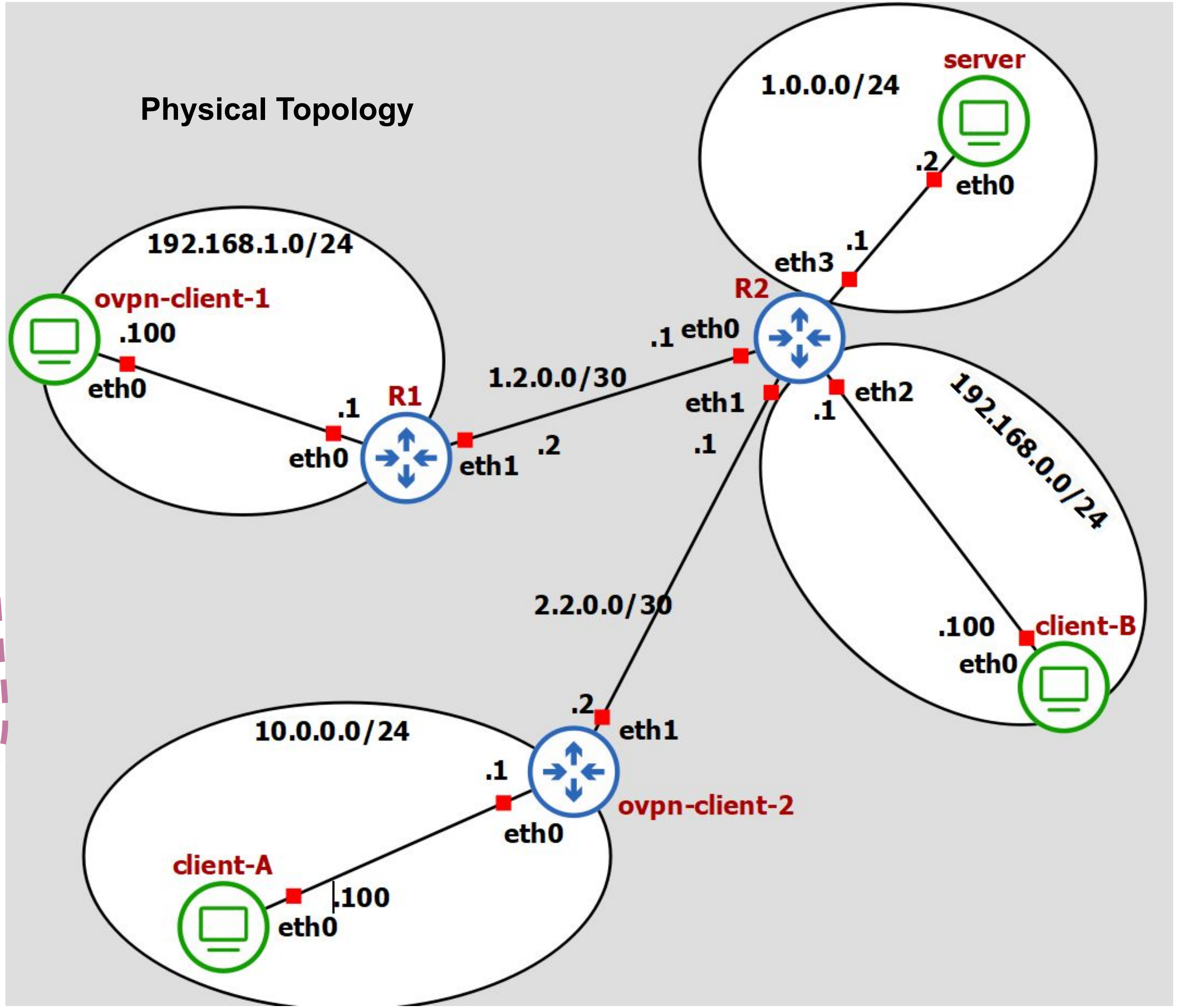
P_DATA plaintext

- packet_id (4 or 8 bytes, if not disabled by --no-replay). In SSL/TLS mode, 4 bytes are used because the implementation can force a TLS renegotation before 2^32 packets are sent. In pre-shared key mode, 8 bytes are used (sequence number and time_t value) to allow long-term key usage without packet_id collisions.
- User plaintext (n bytes).

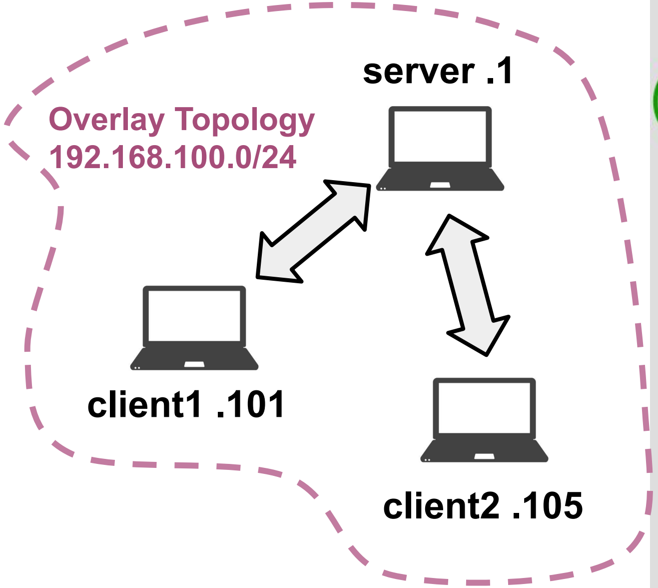
Lab 8: Overlay VPN with OpenVPN

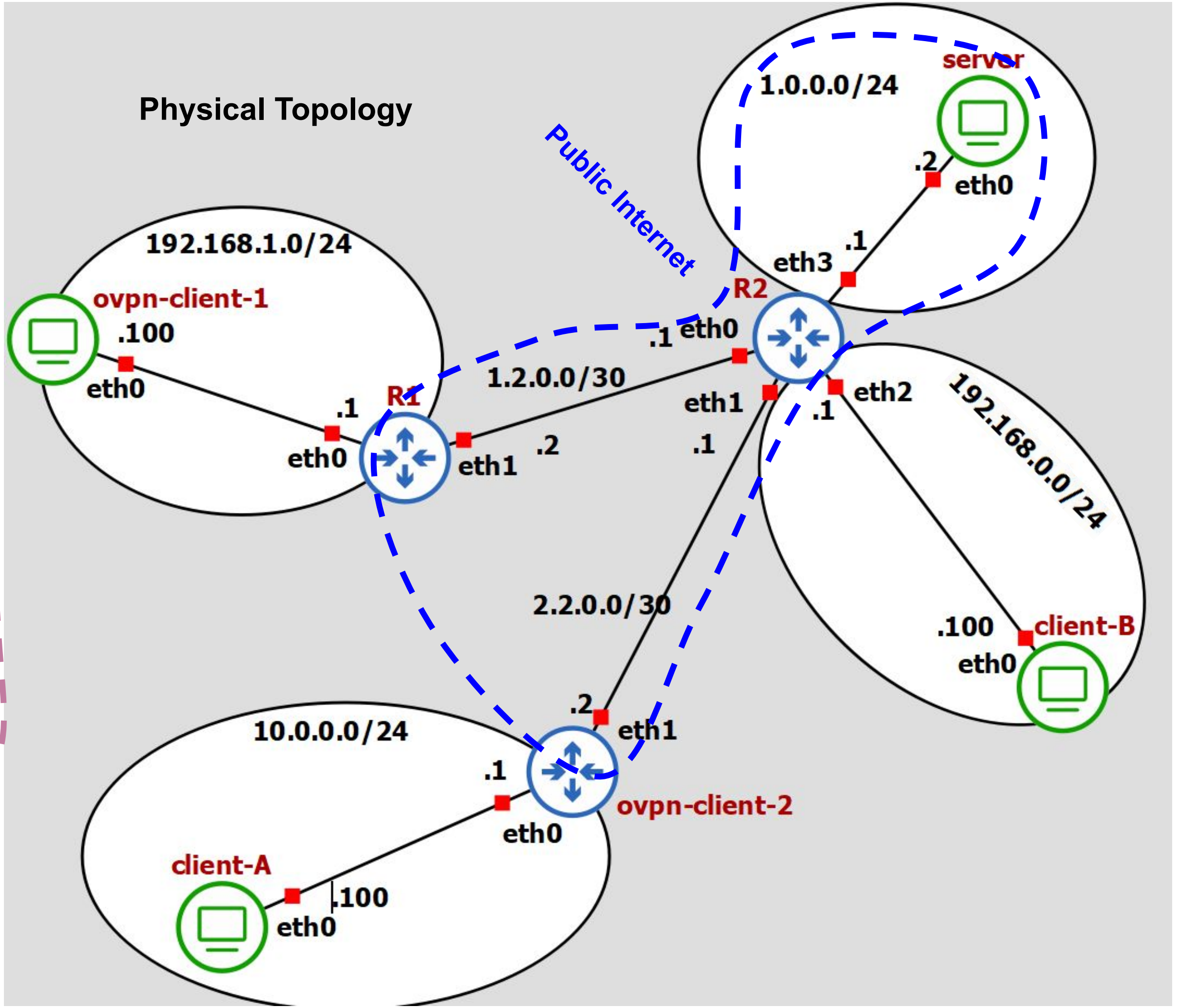
Lab Topology



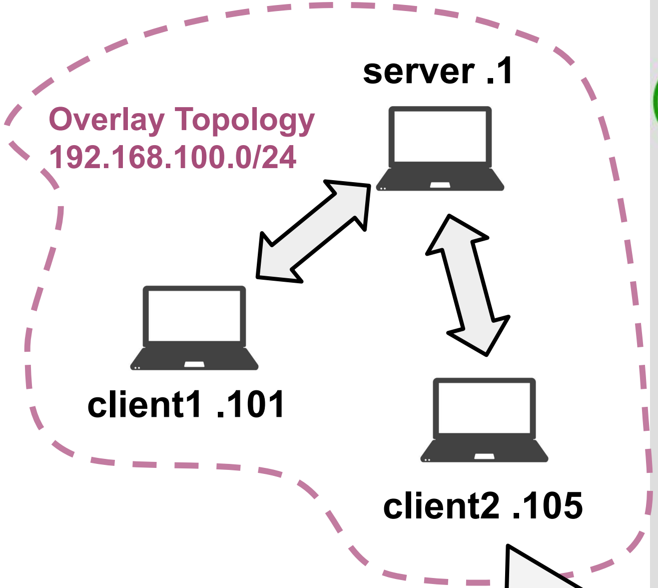


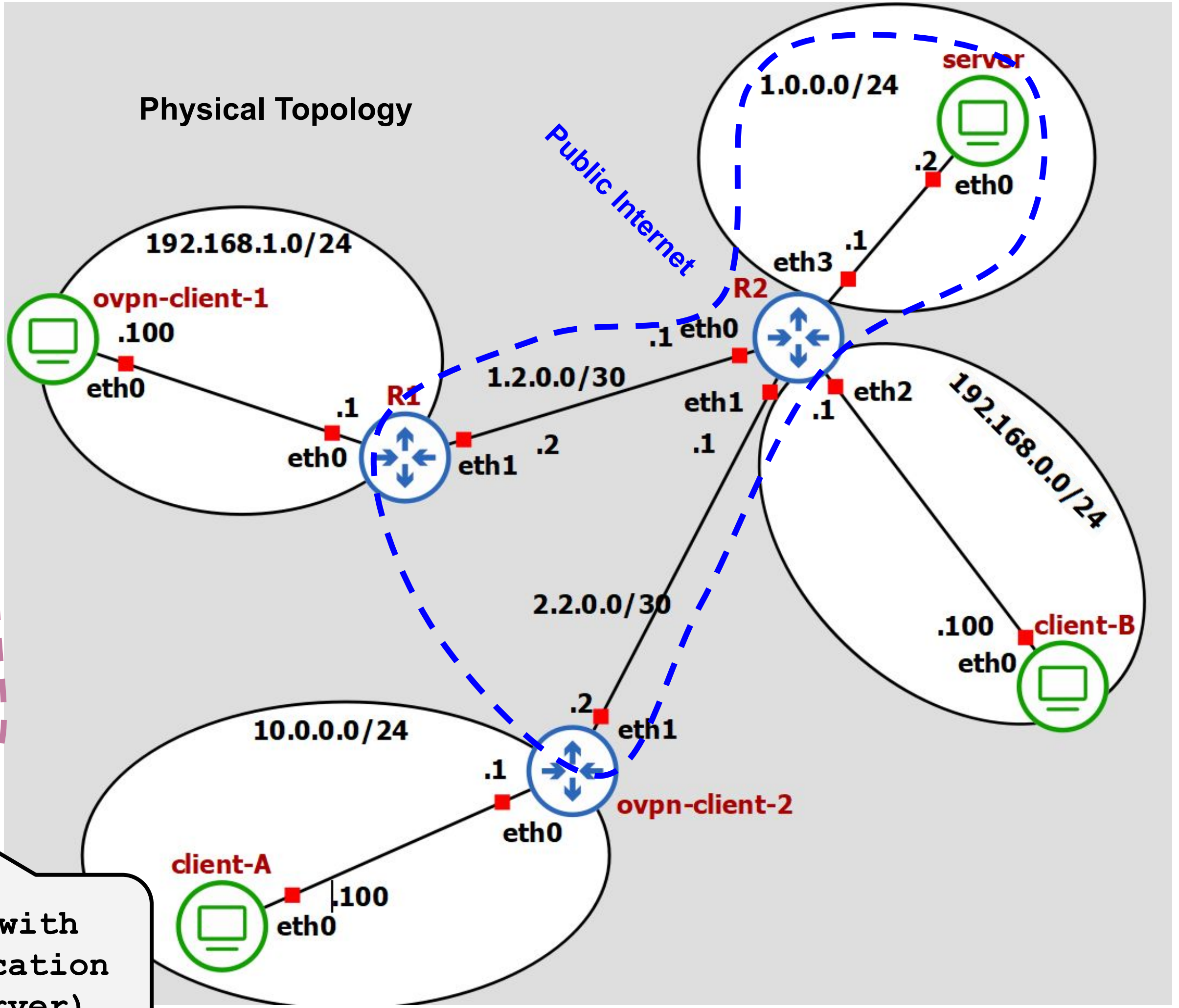
Lab Topology





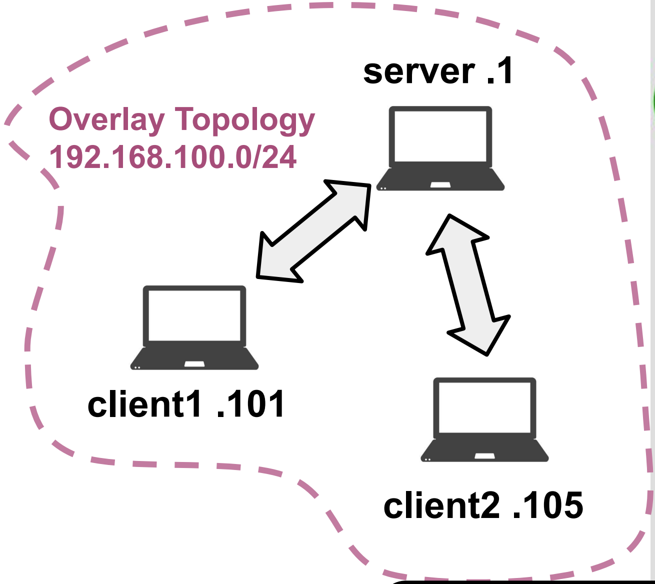
Lab Topology

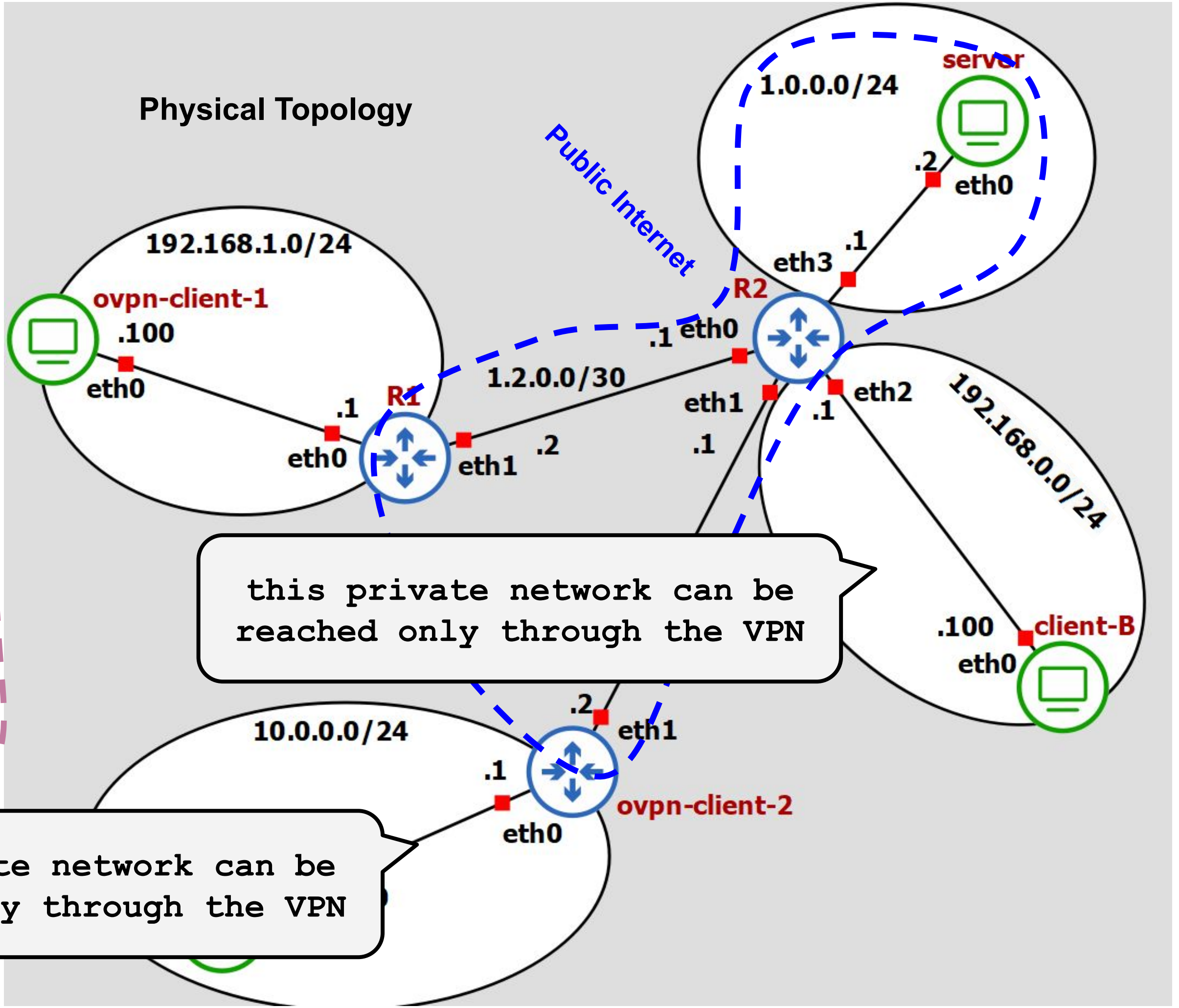




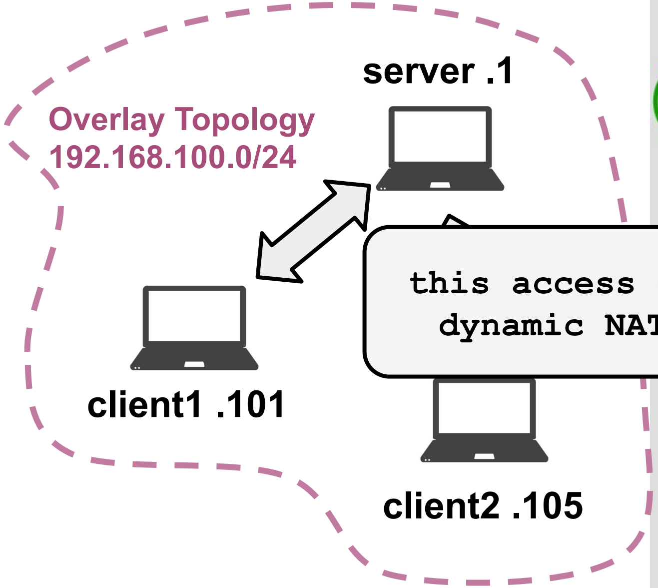

```
hub and spoke topology with
client-to-client communication
allowed (through the server)
```

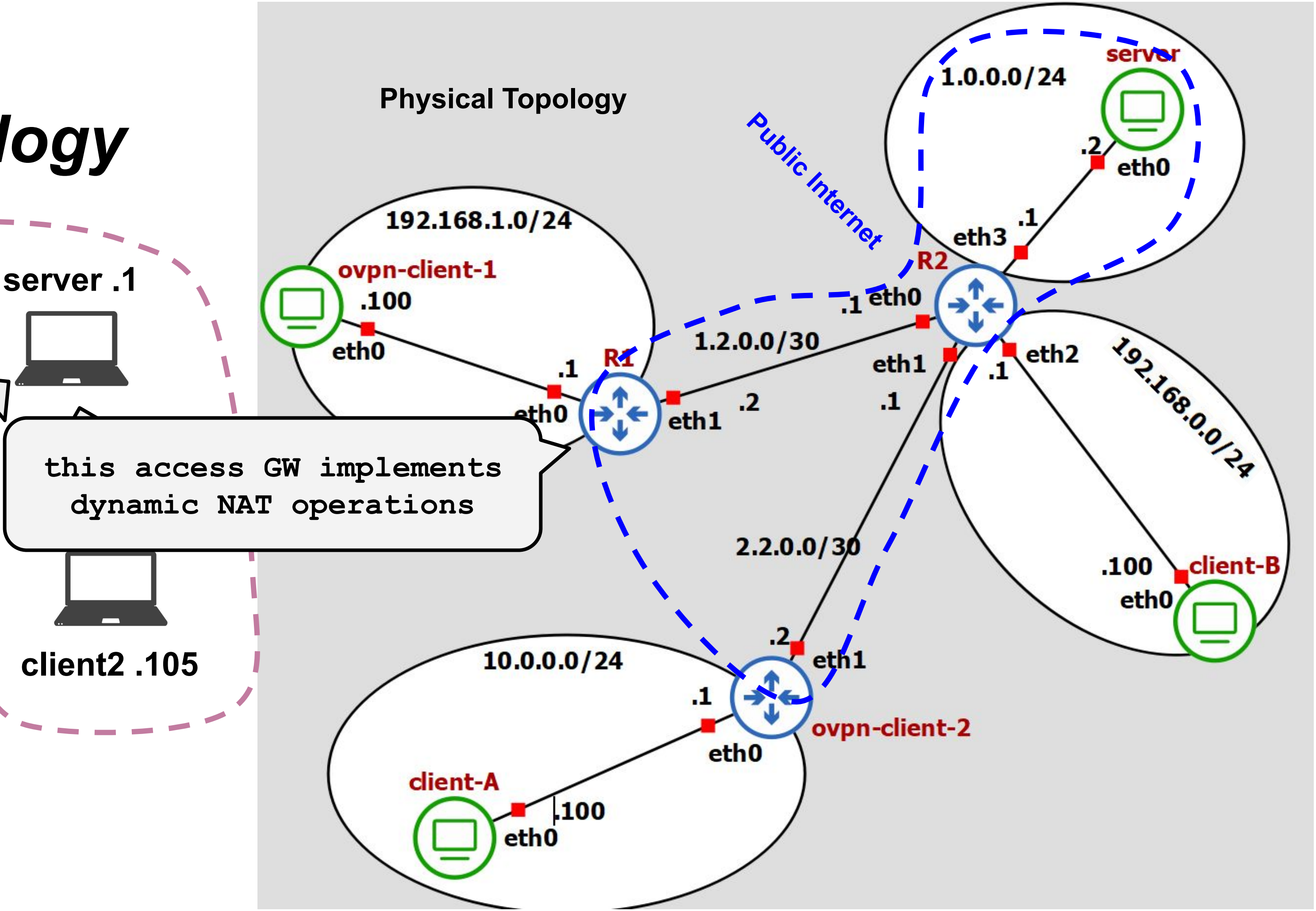
Lab Topology



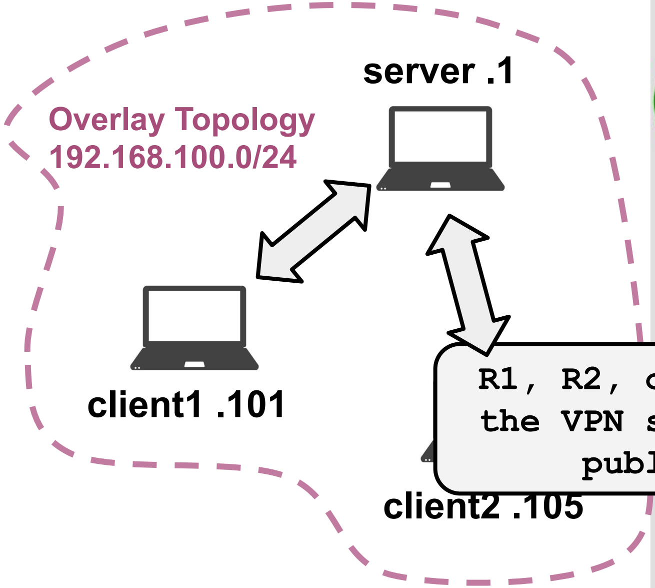


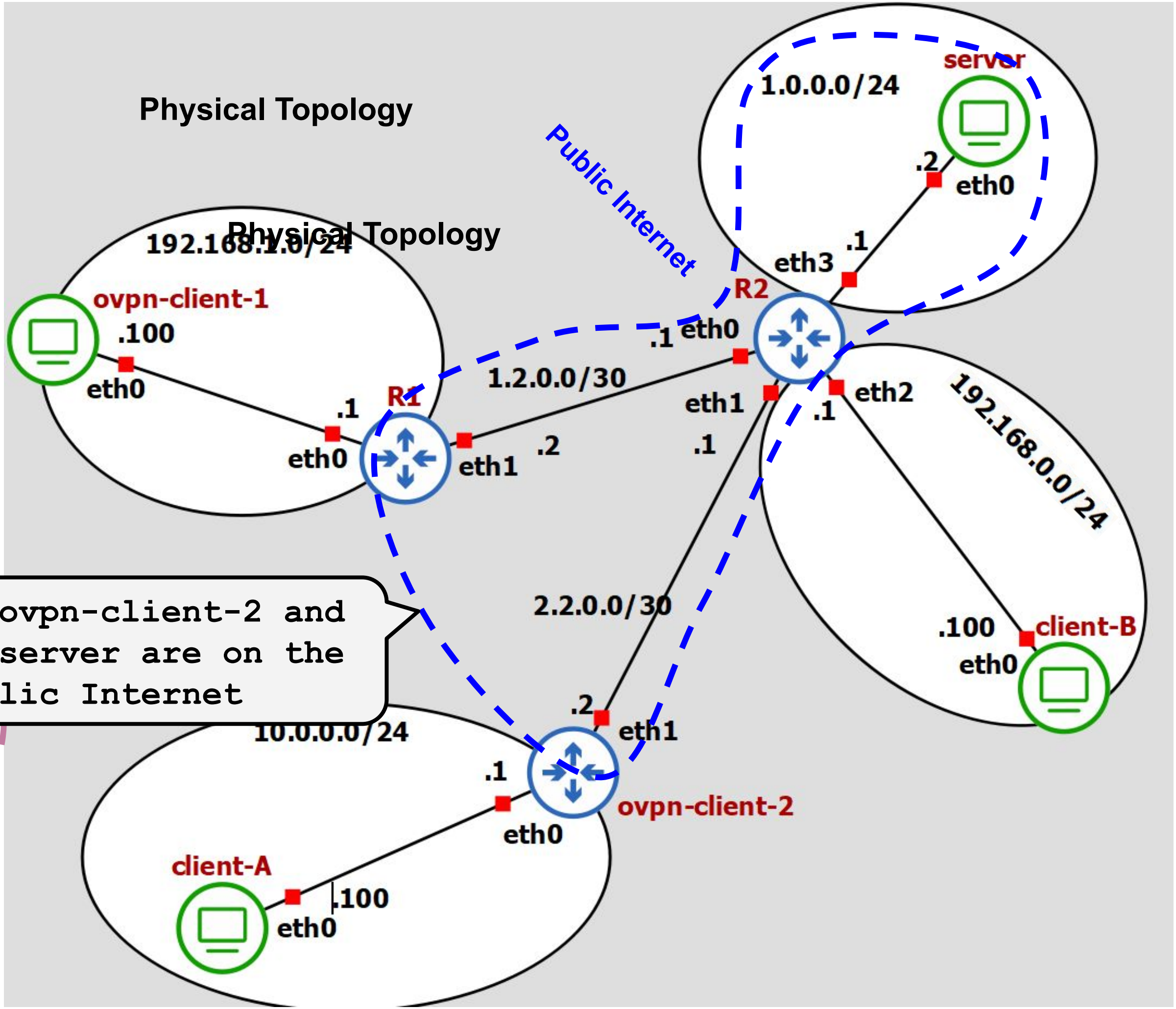
Lab Topology



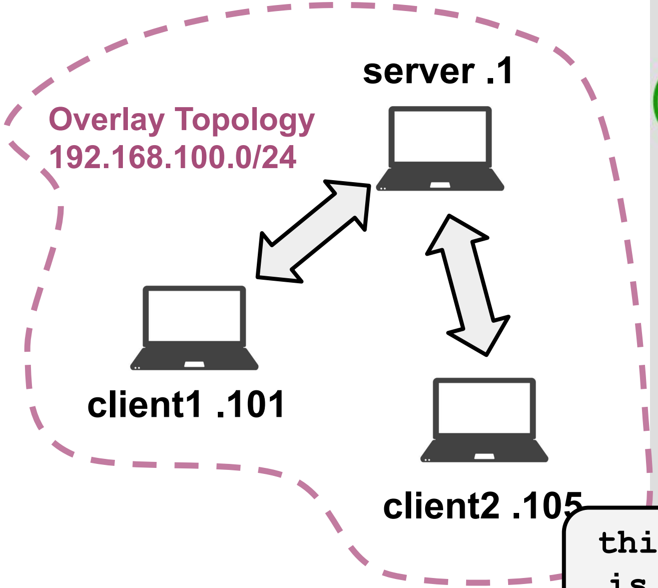


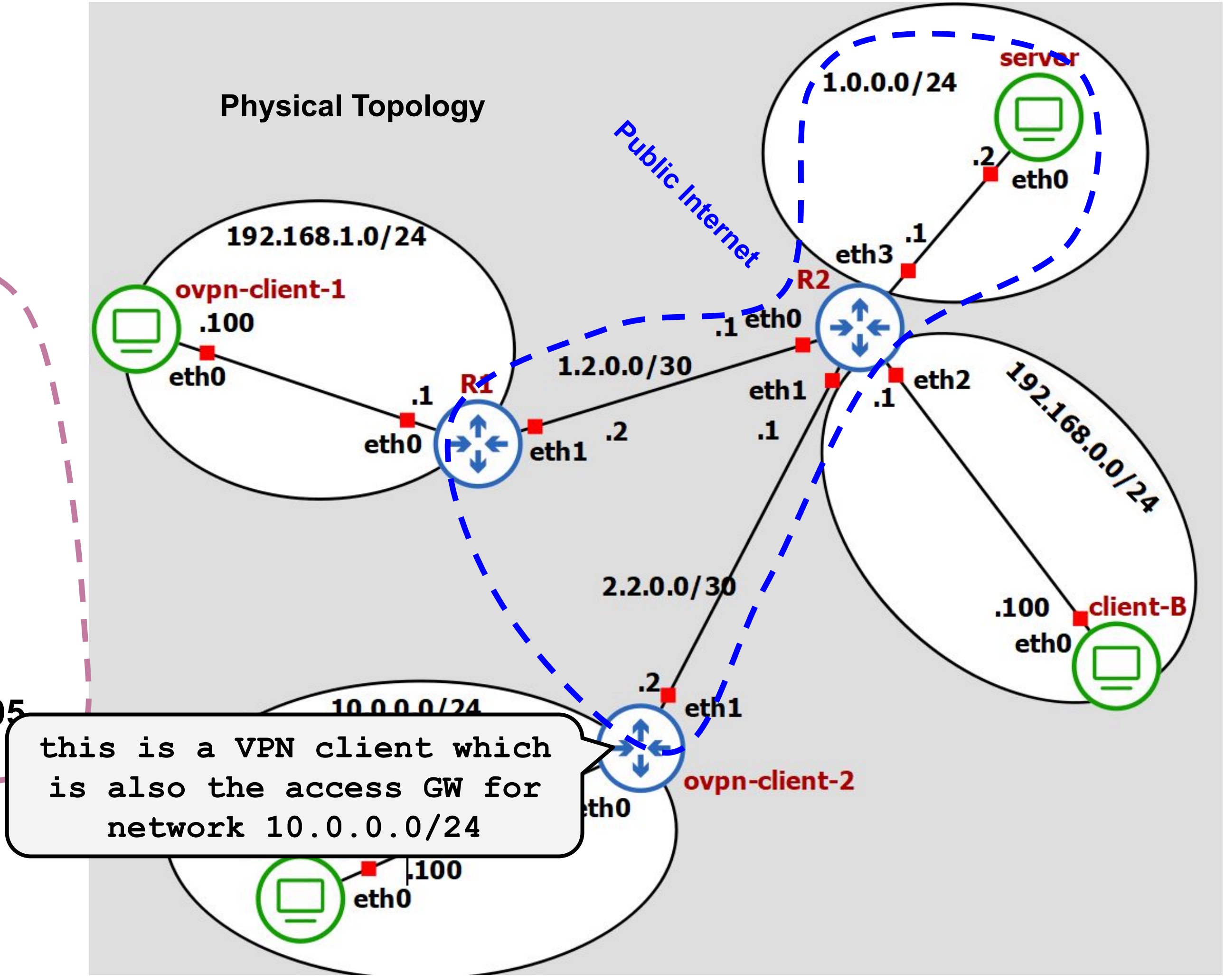
Lab Topology



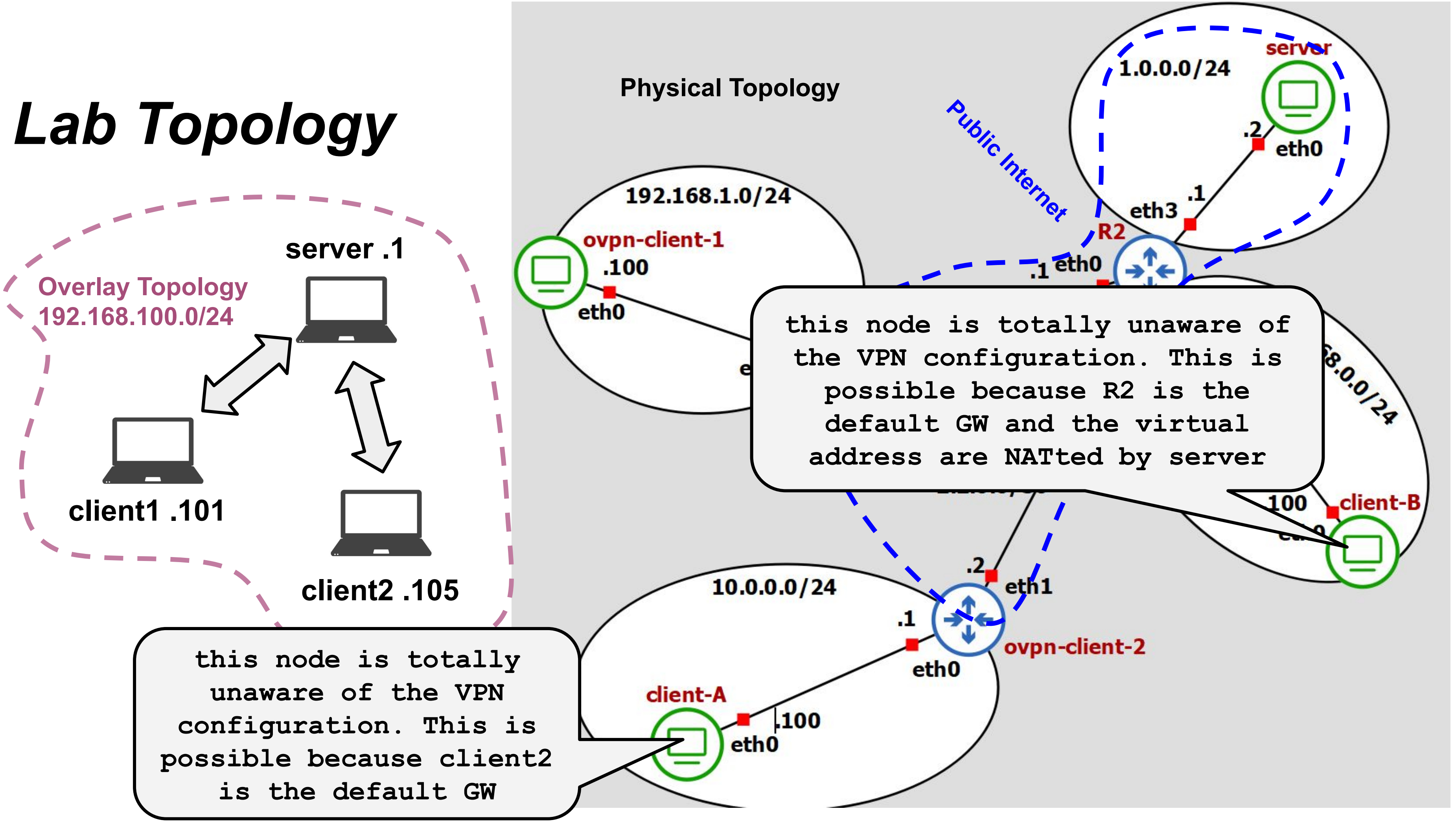


Lab Topology





Lab Topology



Certificate management

Generate the master Certificate Authority (CA) certificate & key

To create and manage our VPN CA we use a tool named easy-rsa (which is basically a wrapper of opensll) which originally was bundeled with openVPN.

Now it is a separate package (on -ubuntu: apt install easy-rsa)

```
# in /usr/share/easy-rsa
server# ./easyrsa init-pki
server# ./easyrsa build-ca nopass
```

Initialization

The final command (build-ca) will build the certificate authority (CA) certificate and key by invoking the interactive openssl command. Most queried parameters were defaulted to the values set in the vars file. The only parameter which must be explicitly entered is the Common Name.

Generate certificate & key for server and clients

Generate a certificate and private key for the server

```
server# ./easyrsa build-server-full server nopass
```

Generate client keys and certificates

```
server# ./easyrsa build-client-full client1 nopass
server# ./build-key client2
```

Diffie Hellman parameters must be generated for the OpenVPN server

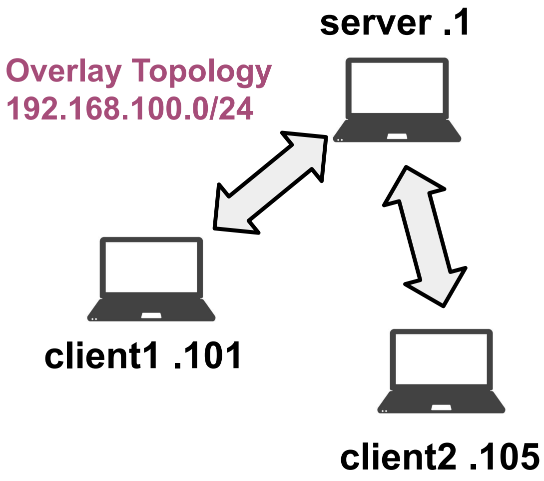
```
server# ./easyrsa gen-dh
```

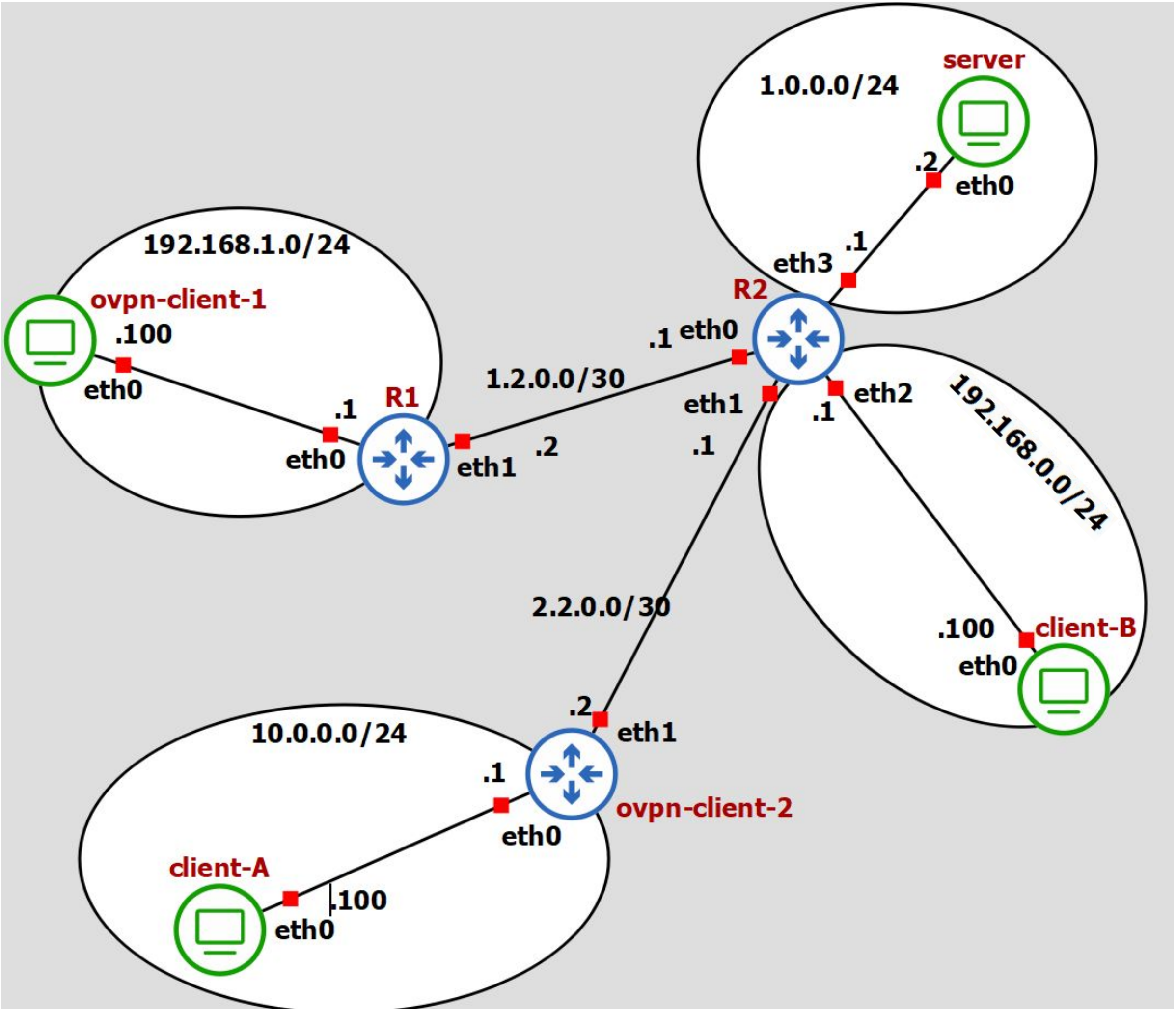
Where do I need the certificates and keys?

Filename	Needed By	Purpose	Secret
ca.crt	server + all clients	Root CA certificate	NO
ca.key	key signing machine only	Root CA key	YES
dh{n}.pem	server only	Diffie Hellman parameters	NO
server.crt	server only	Server Certificate	NO
server.key	server only	Server Key	YES
client1.crt	client1 only	Client1 Certificate	NO
client1.key	client1 only	Client1 Key	YES
client2.crt	client2 only	Client2 Certificate	NO
client2.key	client2 only	Client2 Key	YES

Network Configuration

Configuration



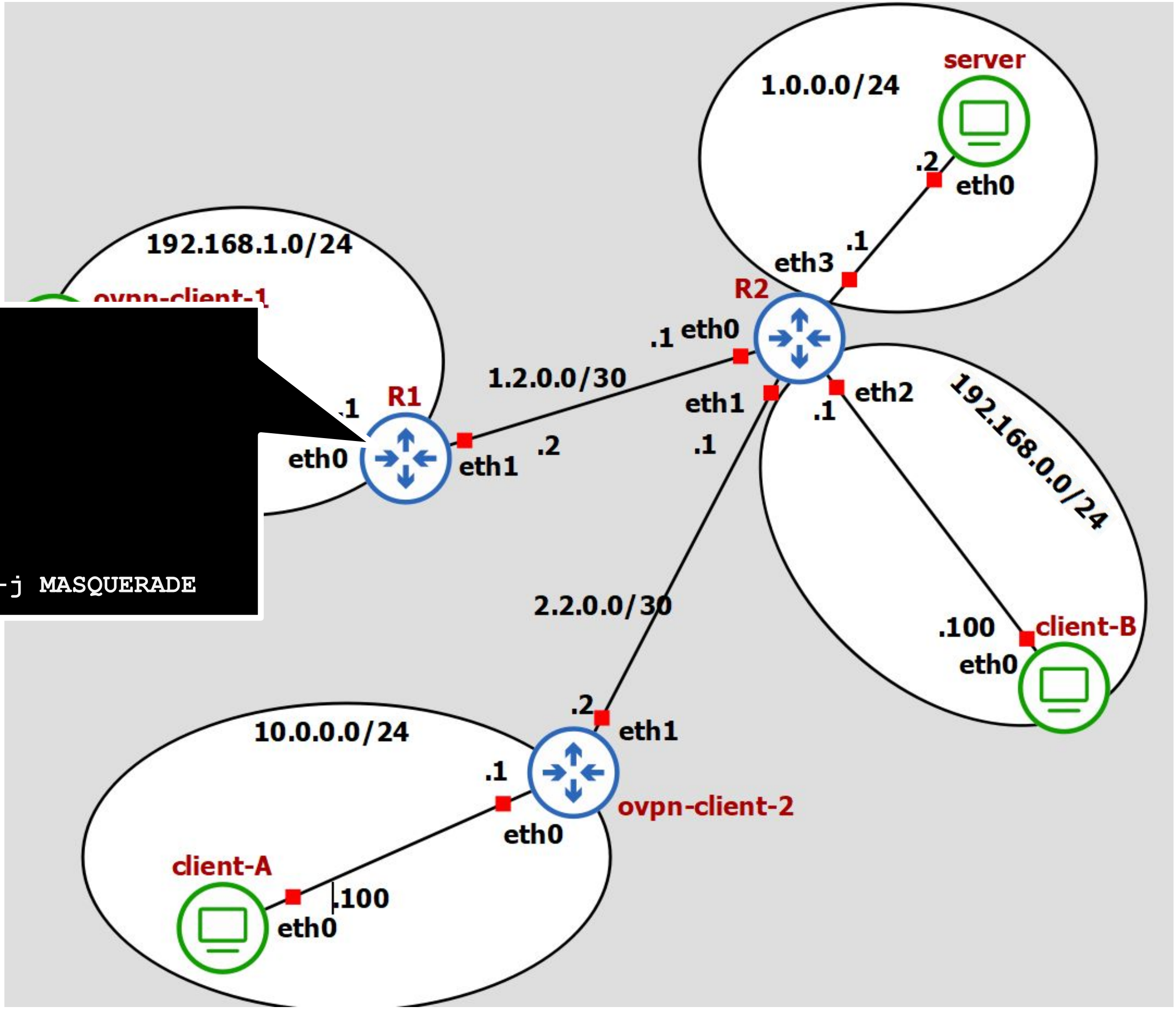


Configuration

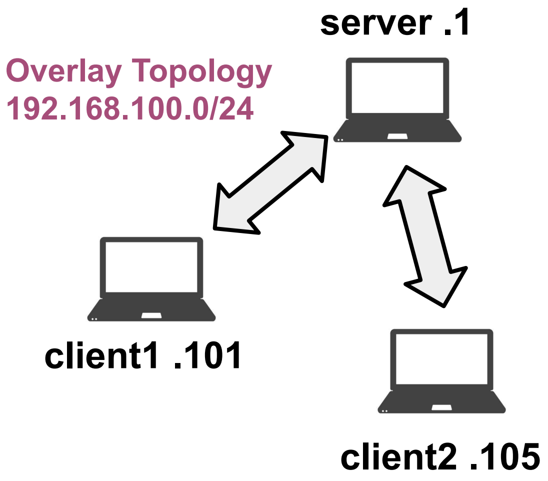
```
sysctl -w net.ipv4.ip_forward=1
ip addr add 192.168.1.1/24 dev eth0
ip addr add 1.2.0.2/30 dev eth1

ip route add 1.0.0.0/24 via 1.2.0.1
ip route add 2.2.0.0/30 via 1.2.0.1

iptables -t nat -A POSTROUTING -o eth1 -j MASQUERADE
```

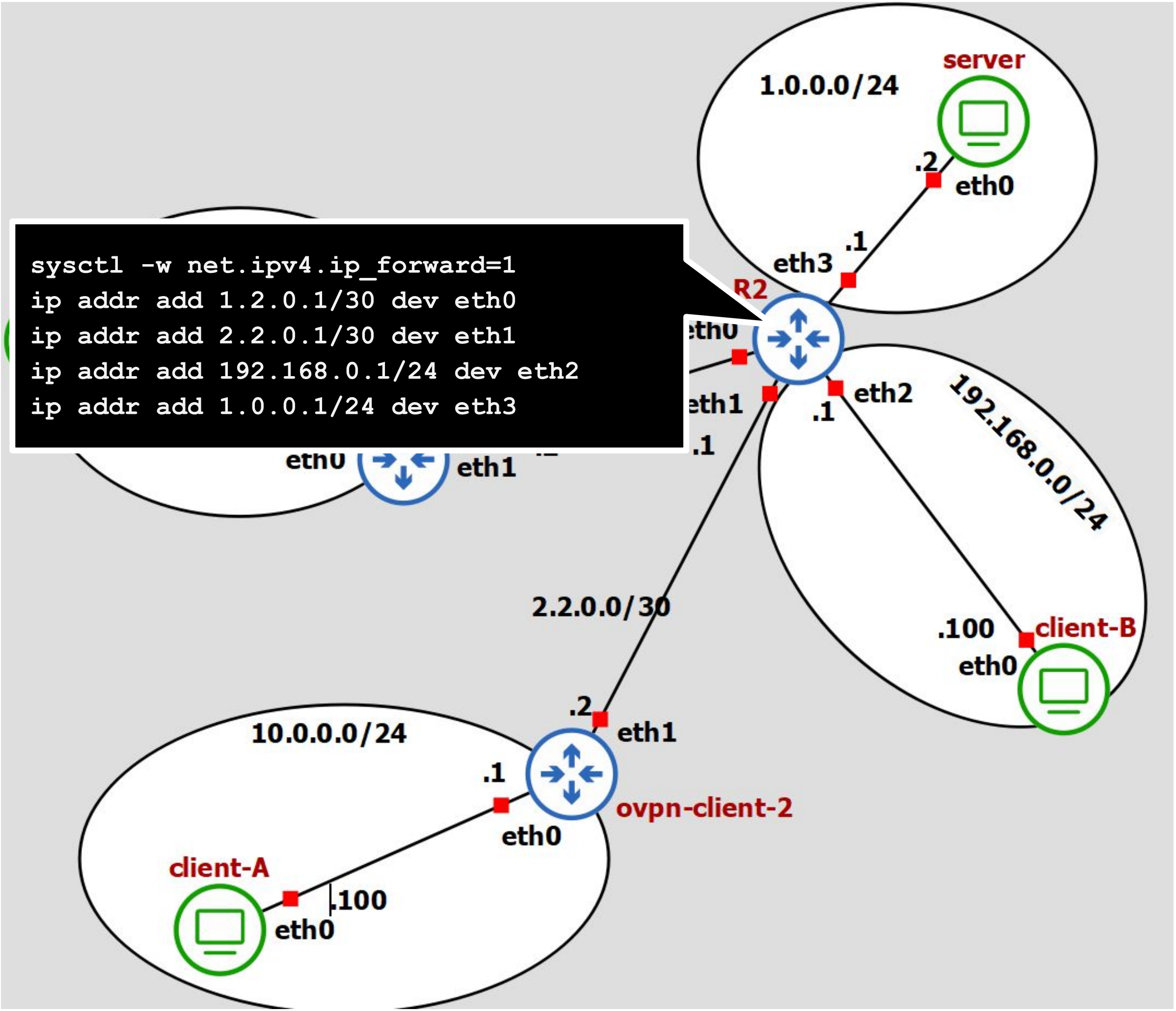
client1 .101

client2 .105



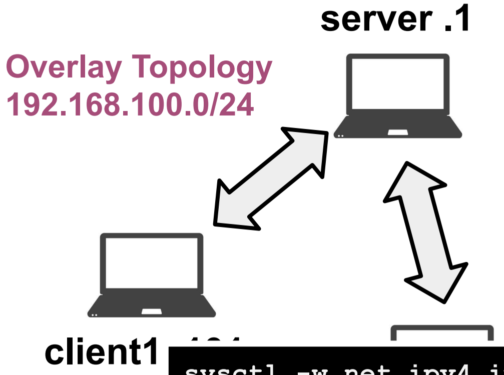
Configuration

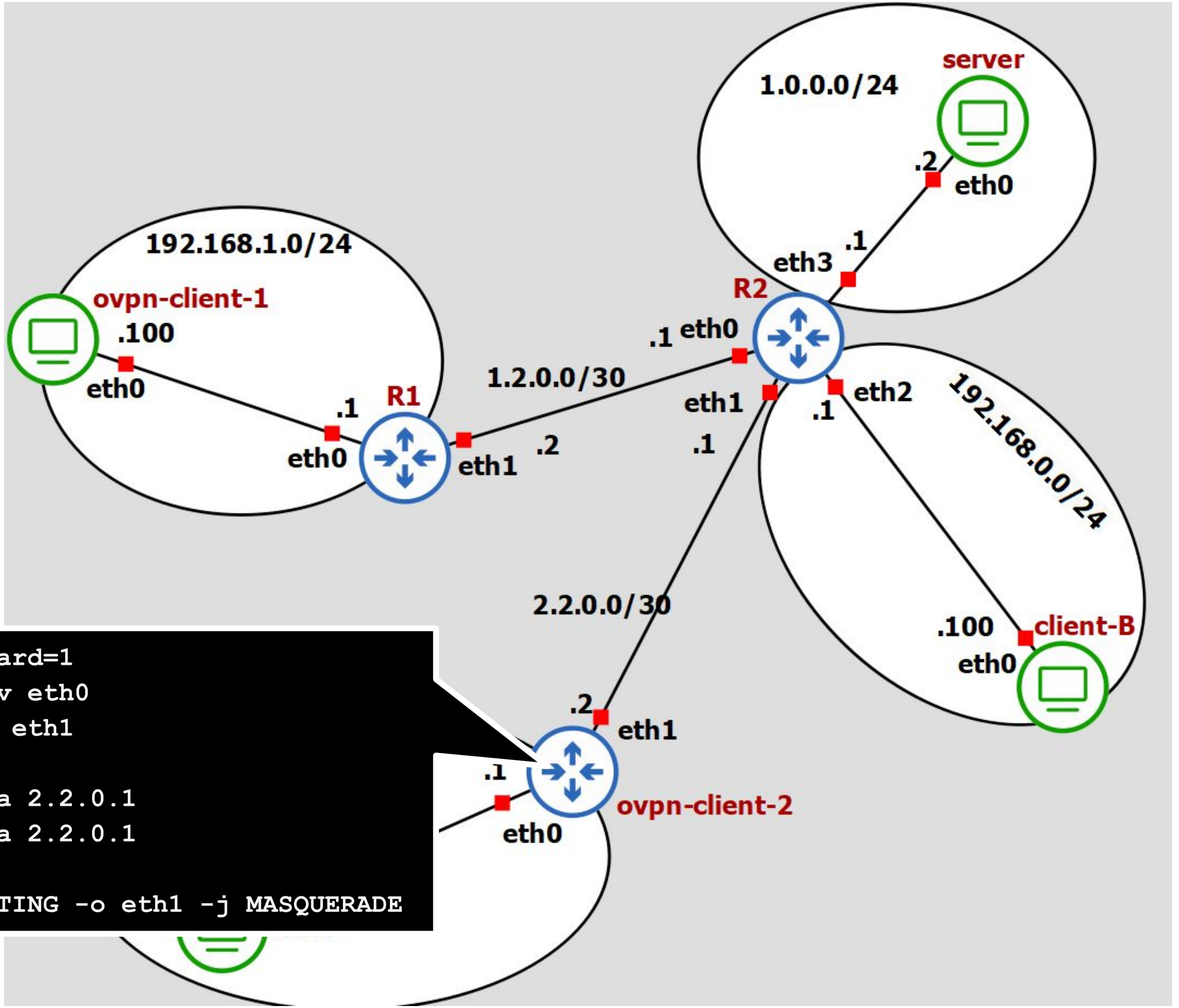





```
sysctl -w net.ipv4.ip_forward=1
ip addr add 1.2.0.1/30 dev eth0
ip addr add 2.2.0.1/30 dev eth1
ip addr add 192.168.0.1/24 dev eth2
ip addr add 1.0.0.1/24 dev eth3
```

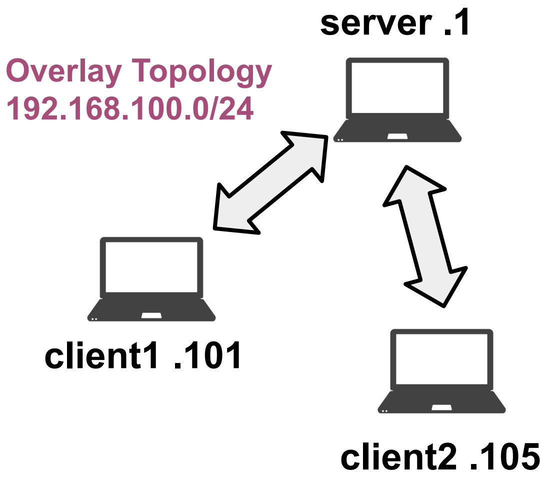
Configuration

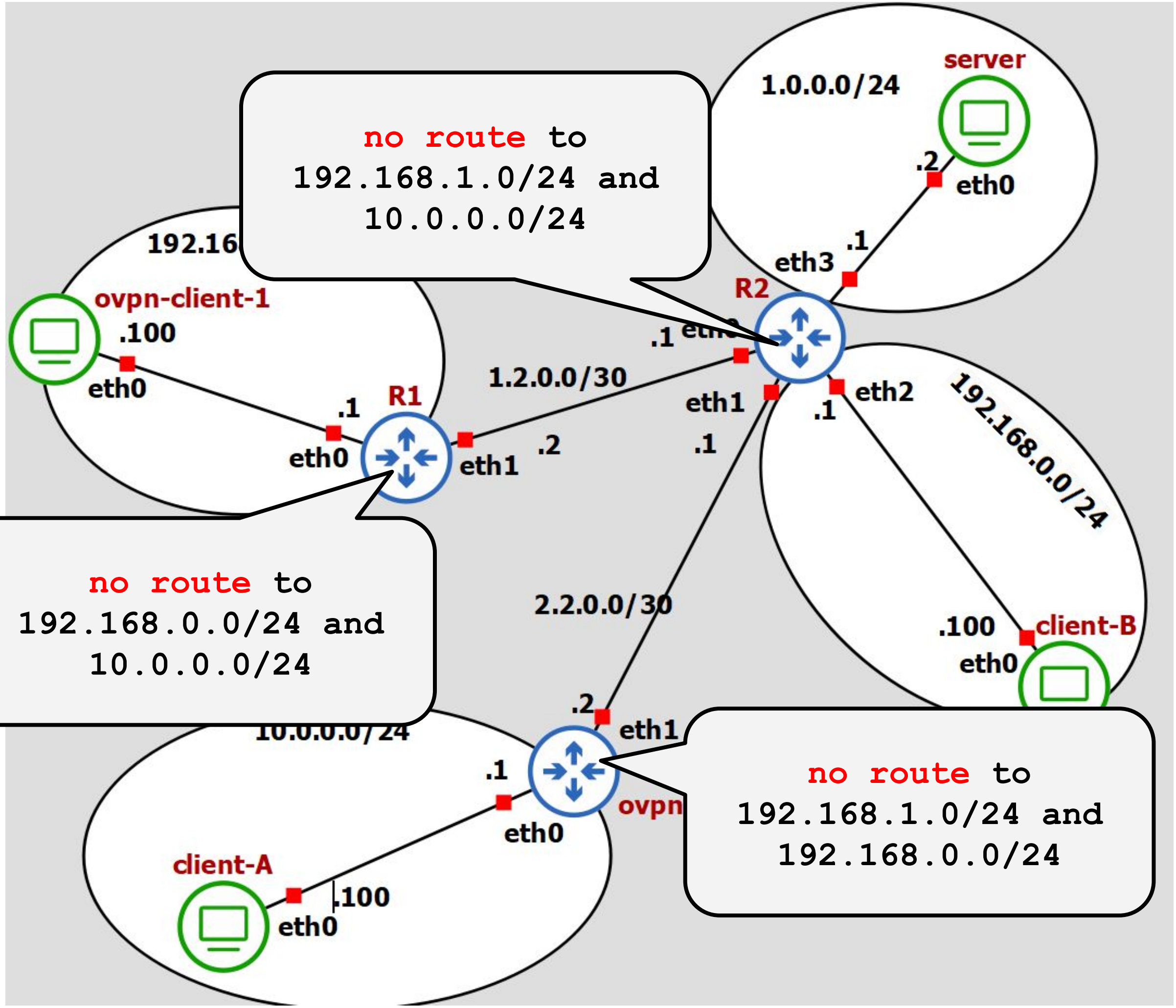





```
sysctl -w net.ipv4.ip_forward=1
ip addr add 10.0.0.1/24 dev eth0
ip addr add 2.2.0.2/30 dev eth1

ip route add 1.0.0.0/24 via 2.2.0.1
ip route add 1.2.0.0/30 via 2.2.0.1

iptables -t nat -A POSTROUTING -o eth1 -j MASQUERADE
```

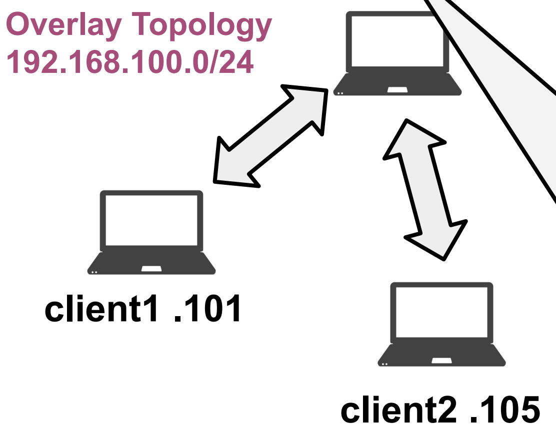
Configuration



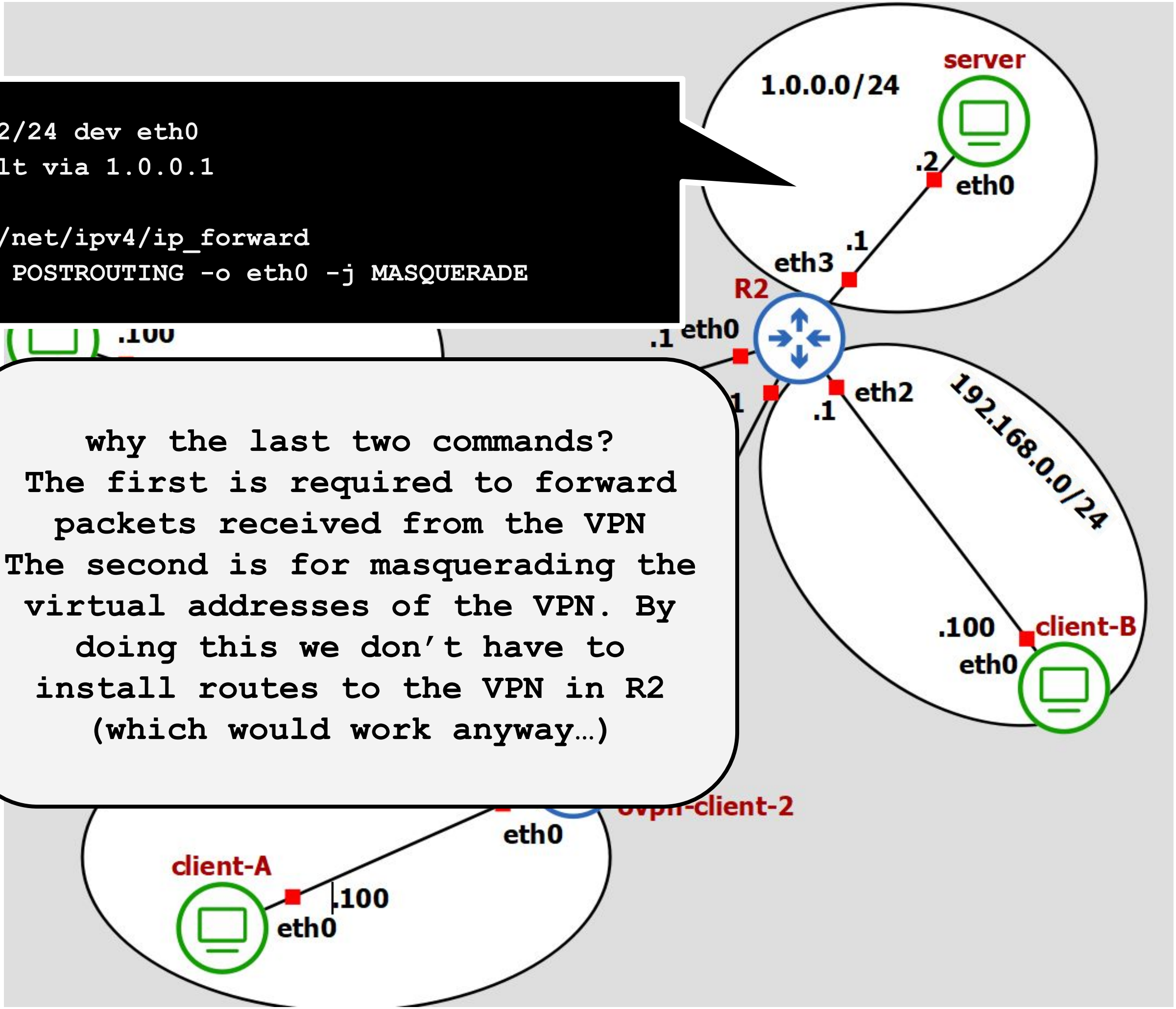


Configur

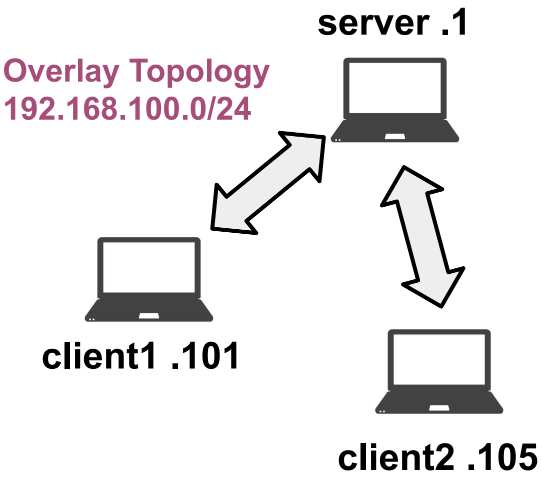
```
ip addr add 1.0.0.2/24 dev eth0
ip route add default via 1.0.0.1

echo 1 > /proc/sys/net/ipv4/ip_forward
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

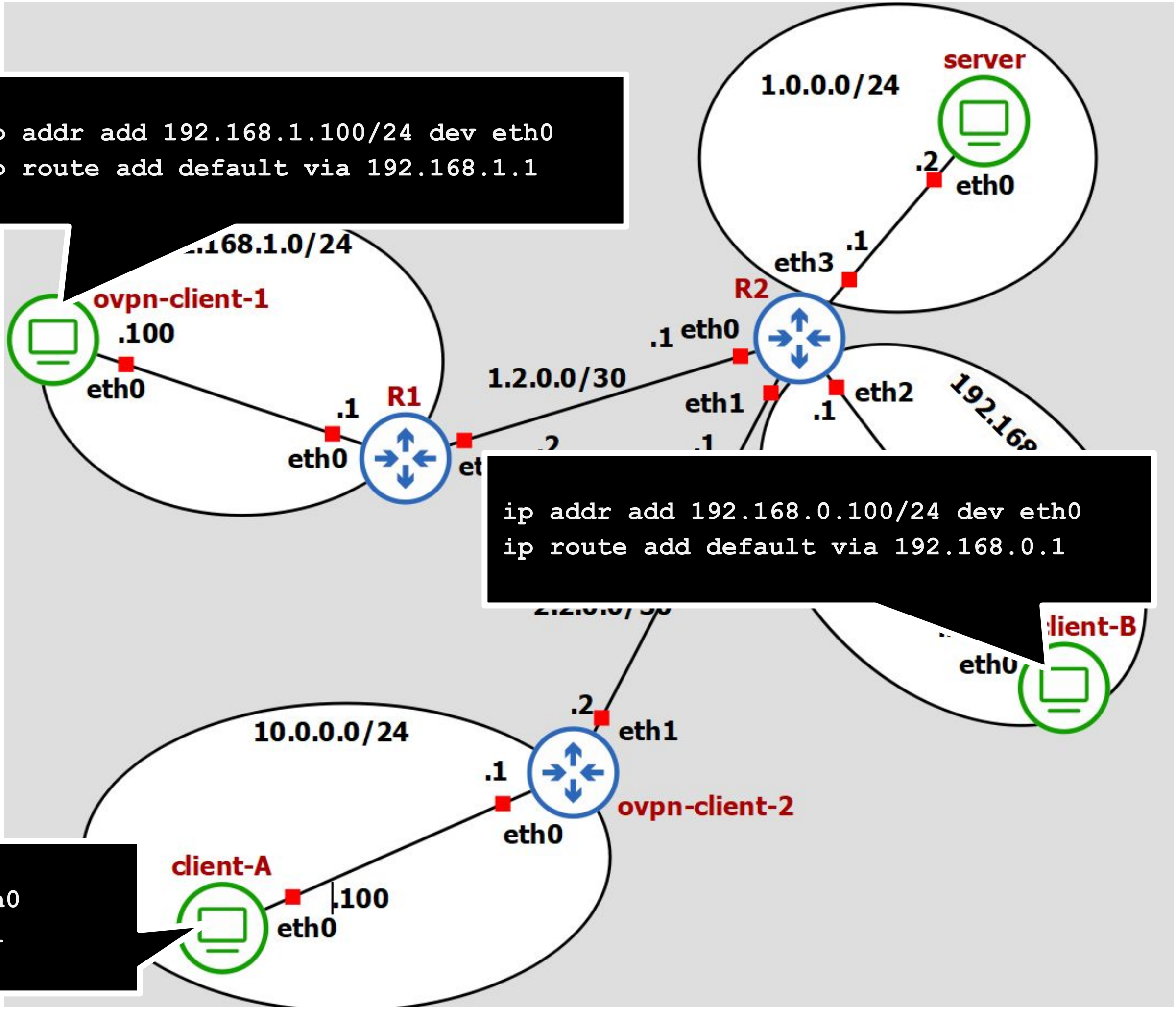


why the last two commands?
The first is required to forward packets received from the VPN
The second is for masquerading the virtual addresses of the VPN. By doing this we don't have to install routes to the VPN in R2 (which would work anyway…)



Configuration






```
ip addr add 192.168.1.100/24 dev eth0
ip route add default via 192.168.1.1
```

```
ip addr add 192.168.0.100/24 dev eth0
ip route add default via 192.168.0.1
```

```
ip addr add 10.0.0.100/24 dev eth0
ip route add default via 10.0.0.1
```

Notes

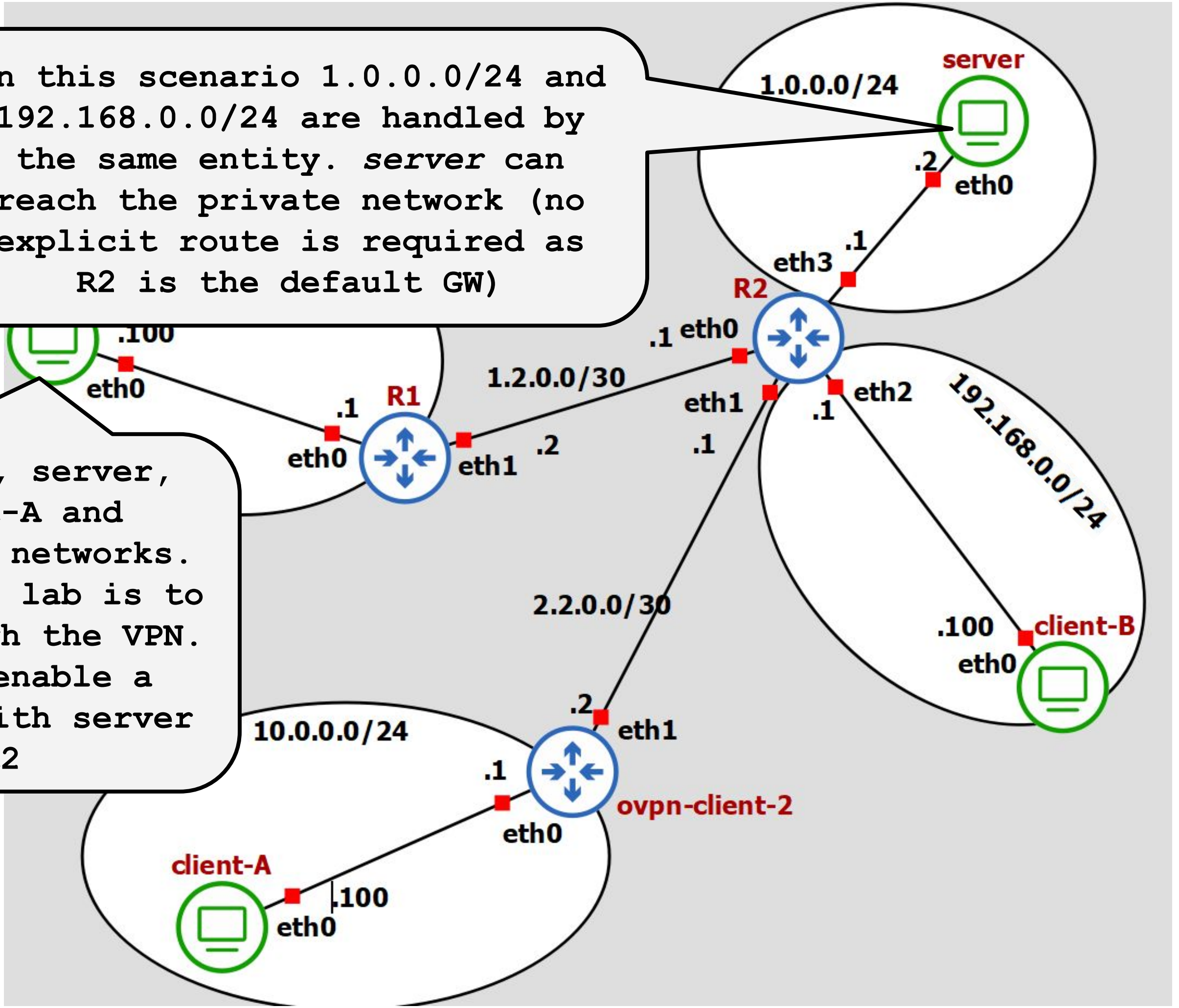
server .1

Overlay Topology
192.168.100.0/24

c

in this scenario 1.0.0.0/24 and 192.168.0.0/24 are handled by the same entity. *server* can reach the private network (no explicit route is required as R2 is the default GW)

this host can reach R2, server, ovpn-client2. client-A and client-B are in private networks. One of the goals of this lab is to reach these nodes through the VPN. The other goal is to enable a private communication with server and ovpn-client2



OpenVPN configuration

Creating configuration files for server and clients

The best way to configure the clients and server is to start from the example configuration files in:

/usr/share/doc/openvpn/example/sample-config-files/

client.conf

server.conf.gz

Important options

- port [port_number]
- proto {udp|tcp}
- dev {tun|tap}
- ca [path]
- cert [path]
- key [path]
- dh [path]
- client
- remote [server_addr] [port]
- server [net_addr] [met_mask]
- client_to_client
- push "route net_addr net_mask"
- route net_addr net_mask
- client-config-dir [path]

this specifies the listening port for the server. This port must match the one specified with the client configuration directive "remote". This port will also be the remote port for the external headers of packets sent from the clients to the server

Important options

- port [port_number]
- proto {udp|tcp}
- dev {tun|tap}
- ca [path]
- cert [path]
- key [path]
- dh [path]
- client
- remote [server_addr] [port]
- server [net_addr] [met_mask]
- client_to_client
- push "route net_addr net_mask"
- route net_addr net_mask
- client-config-dir [path]

this specifies the encapsulation format and must be the same in both clients and server configuration files

Important options

- port [port_number]
- proto {udp|tcp}
- dev {tun|tap}
- ca [path]
- cert [path]
- key [path]
- dh [path]
- client
- remote [server_addr] [port]
- server [net_addr] [met_mask]
- client_to_client
- push "route net_addr net_mask"
- route net_addr net_mask
- client-config-dir [path]

this specifies the virtual interface type and must be the same in both clients and server configuration files

Important options

- port [port_number]
- proto {udp|tcp}
- dev {tun|tap}
- ca [path]
- cert [path]
- key [path]
- dh [path]
- client
- remote [server_addr] [port]
- server [net_addr] [met_mask]
- client_to_client
- push "route net_addr net_mask"
- route net_addr net_mask
- client-config-dir [path]

path to:
1. CA certificate (server/client)
2. local certificate (server/client)
3. key pair file (server/client)
4. DH parameters (server)

Important options

- port [port_number]
- proto {udp|tcp}
- dev {tun|tap}
- ca [path]
- cert [path]
- key [path]
- dh [path]
- client
- remote [server_addr] [port]
- server [net_addr] [met_mask]
- client_to_client
- push "route net_addr net_mask"
- route net_addr net_mask
- client-config-dir [path]

this tells OpenVPN to run as a client instance. In this case the next directive specifies the IP and port of the server

Important options

- port [port_number]
- proto {udp|tcp}
- dev {tun|tap}
- ca [path]
- cert [path]
- key [path]
- dh [path]
- client
- remote [server_addr] [port]
- server [net_addr] [met_mask]
- client_to_client
- push "route net_addr net_mask"
- route net_addr net_mask
- client-config-dir [path]

this tells OpenVPN to run as a server instance and to allocate a given VPN address range. The server's virtual adapter will be configured with the first valid IP address of this address range

Important options

- port [port_number]
- proto {udp|tcp}
- dev {tun|tap}
- ca [path]
- cert [path]
- key [path]
- dh [path]
- client
- remote [server_addr]
- server [net_addr] [n
- client_to_client
- push "route net_addr net_mask"
- route net_addr net_mask
- client-config-dir [path]

this enables overlay client to client communication through the VPN server. The overlay topology is a hub and spoke

Important options

- port [port_number]
- proto {udp|tcp}
- dev {tun|tap}
- ca [path]
- cert [path]
- key [path]
- dh [path]
- client
- remote [server_addr] [po
- server [net_addr] [met_mas
- client_to_client
- push "route net_addr net_mask"
- route net_addr net_mask
- client-config-dir [path]

These two option influence the real routing table of all clients (first directive) and servers (second directive). Multiple routes can be specified. Each directive will result in the automatic insertion of a routing entry:

net_addr/net_mast via virtual_next_hop dev viface

Important options

- port [port_number]
- proto {udp|tcp}
- dev {tun|tap}
- ca [path]
- cert [path]
- key [path]
- dh [path]
- client
- remote [server_addr] [po
- server [net_addr] [met_r
- client_to_client
- push "route net_addr net_mas
- route net_addr net_mask
- client-config-dir [path]

This set the path to the per-client specific configuration directory. In this directory the server can have multiple files names as the CN of the client. Configuration directives in a file will affect only the relative client

Client-specific configuration

- A file for each OpenVPN client “CN”
  - In this lab: client1, client2
- In each file (+ other commands we’re not considering):
  - if-config-push [local_ptp] [remote_ptp]
  - iroute [net_addr] [net_mask]
- client1
  - **`ifconfig-push 192.168.100.101 192.168.100.102`**
- client2
  - **`ifconfig-push 192.168.100.105 192.168.100.106`**
  - **`iroute 10.0.0.0 255.255.255.0`**

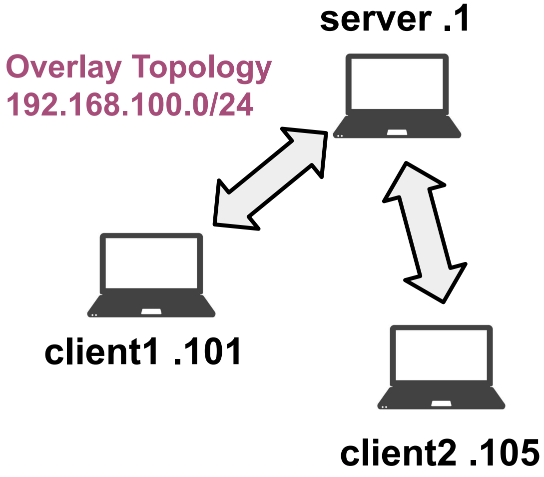
`Allowed /30 pairs`

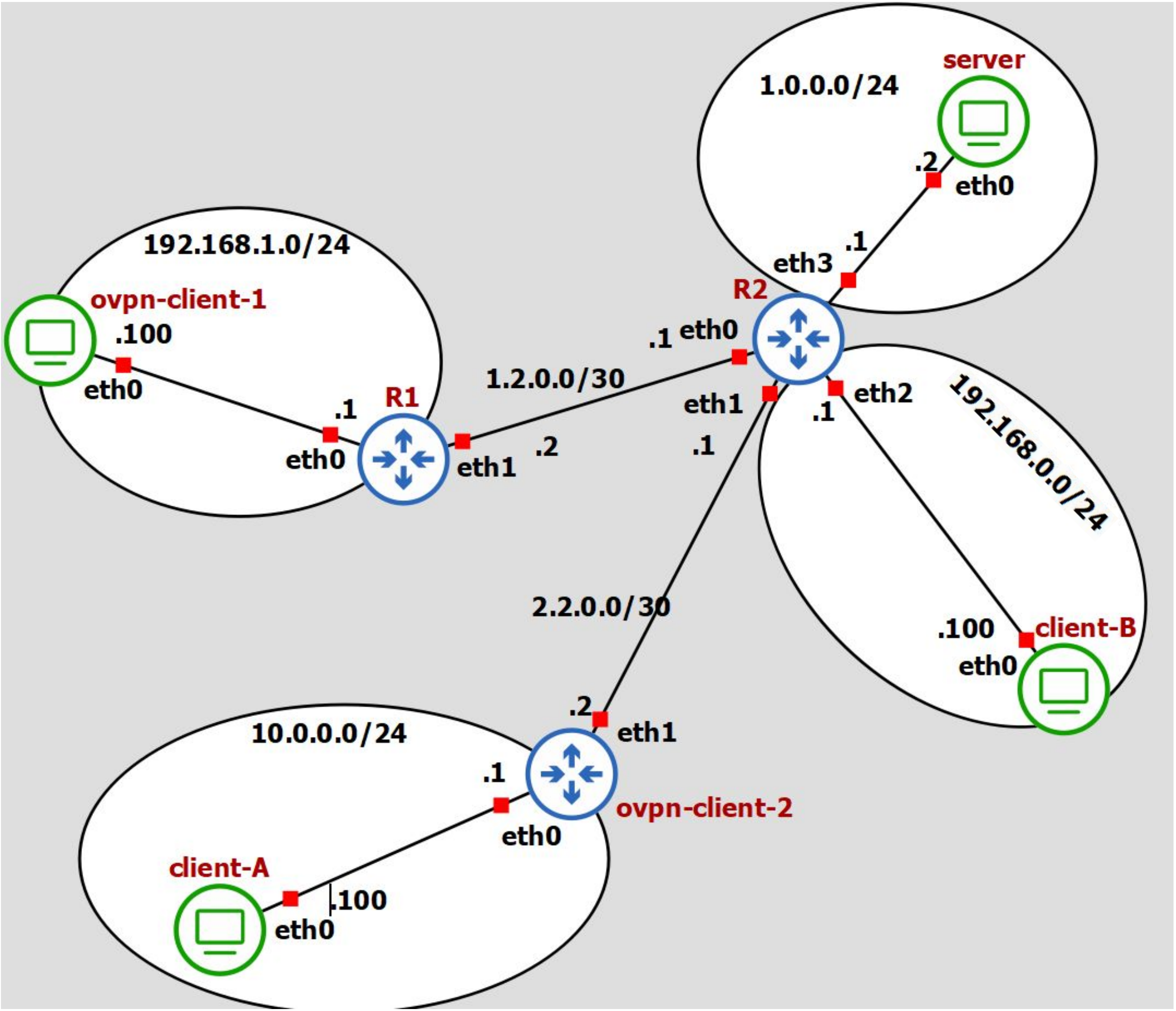
```
[  1,  2] [  5,  6] [  9, 10] [ 13, 14] [ 17, 18]
[ 21, 22] [ 25, 26] [ 29, 30] [ 33, 34] [ 37, 38]
[ 41, 42] [ 45, 46] [ 49, 50] [ 53, 54] [ 57, 58]
[ 61, 62] [ 65, 66] [ 69, 70] [ 73, 74] [ 77, 78]
[ 81, 82] [ 85, 86] [ 89, 90] [ 93, 94] [ 97, 98]
[101,102] [105,106] [109,110] [113,114] [117,118]
[121,122] [125,126] [129,130] [133,134] [137,138]
[141,142] [145,146] [149,150] [153,154] [157,158]
[161,162] [165,166] [169,170] [173,174] [177,178]
[181,182] [185,186] [189,190] [193,194] [197,198]
[201,202] [205,206] [209,210] [213,214] [217,218]
[221,222] [225,226] [229,230] [233,234] [237,238]
[241,242] [245,246] [249,250] [253,254]
```

Why “push route”, “route” and “iroute”?

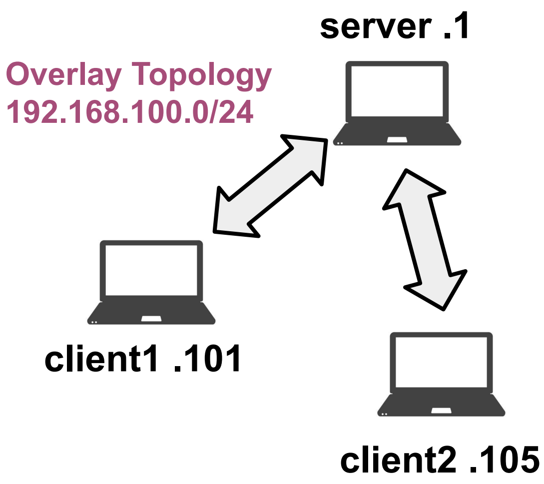
1. ***“push route”*** is pushing a given route to clients
    a. this influences the underlay routing in the clients
    b. after connection, the client will add a route via tun0 p2p peer
2. ***“route”*** controls the routing from the kernel to the OpenVPN server (via the TUN interface)
    a. ***this influences the underlay routing***
    b. routes specified with this command are installed in the real IP routing table
3. ***“iroute”*** controls the routing from the OpenVPN server to the remote clients
    a. ***this influences the overlay routing***
    b. routes specified with this command are installed in the overlay routing table (which is managed by the OpenVPN server process)

Configuration

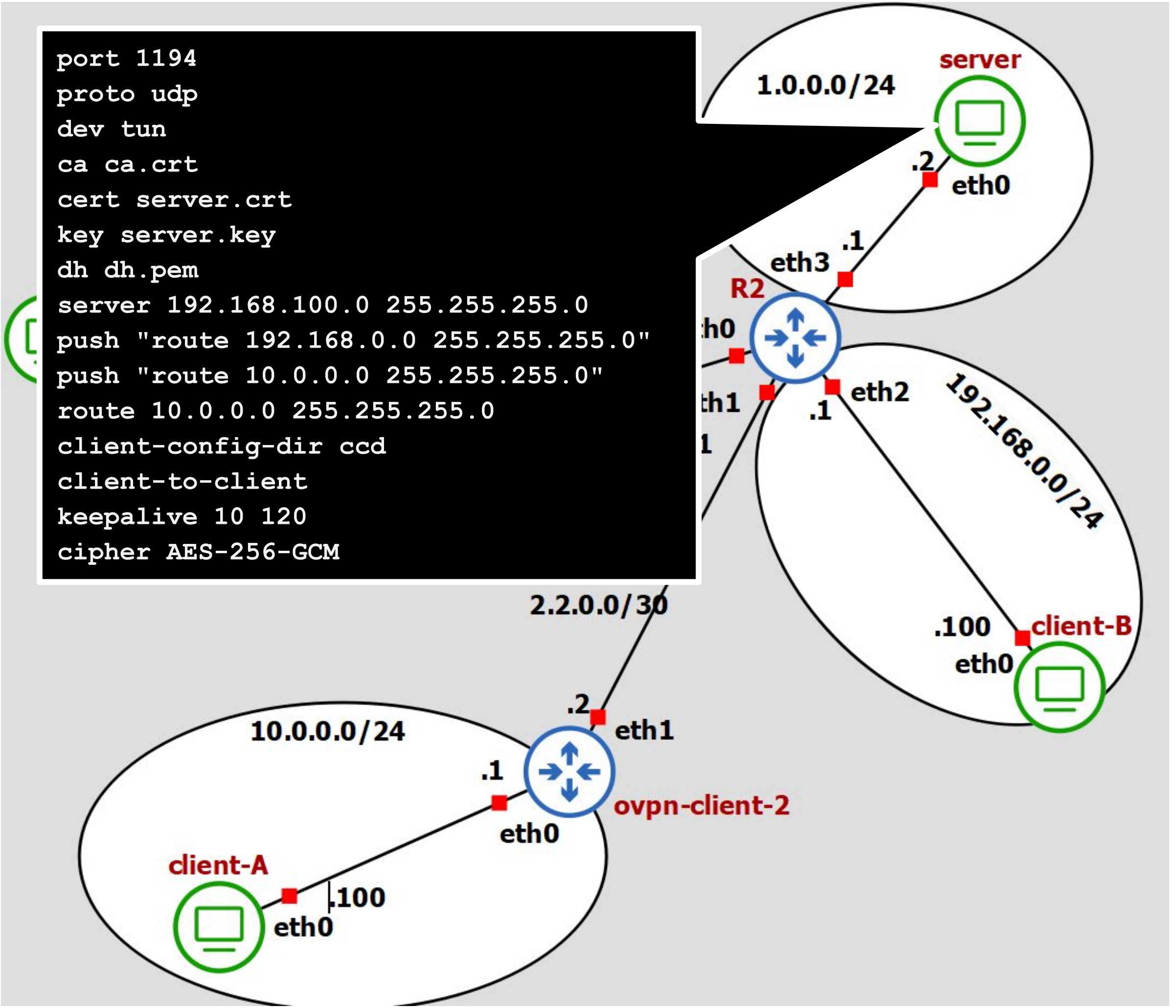




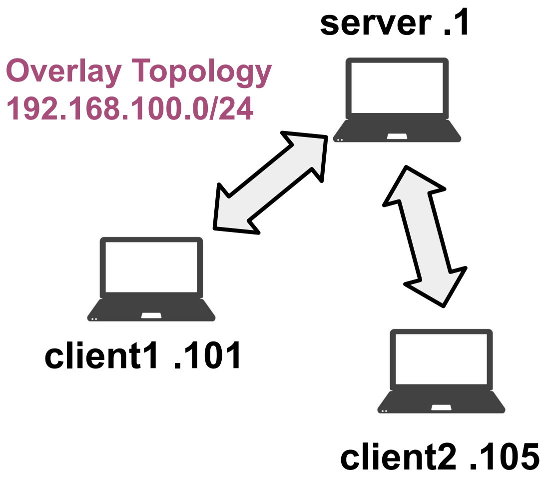
VPN Server



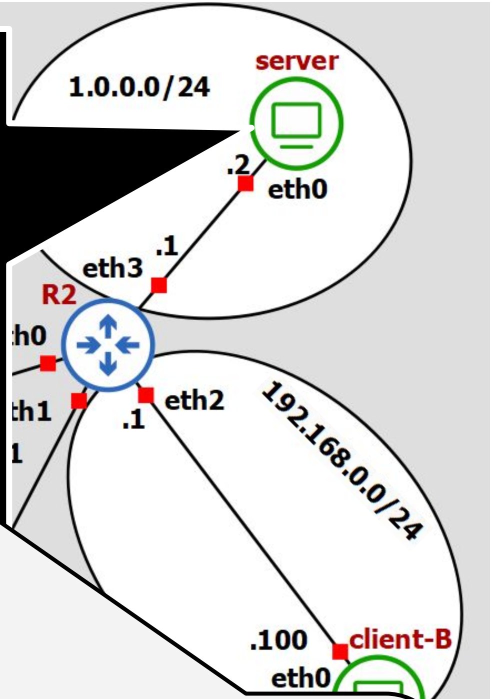

```
port 1194
proto udp
dev tun
ca ca.crt
cert server.crt
key server.key
dh dh.pem
server 192.168.100.0 255.255.255.0
push "route 192.168.0.0 255.255.255.0"
push "route 10.0.0.0 255.255.255.0"
route 10.0.0.0 255.255.255.0
client-config-dir ccd
client-to-client
keepalive 10 120
cipher AES-256-GCM
```



VPN Server



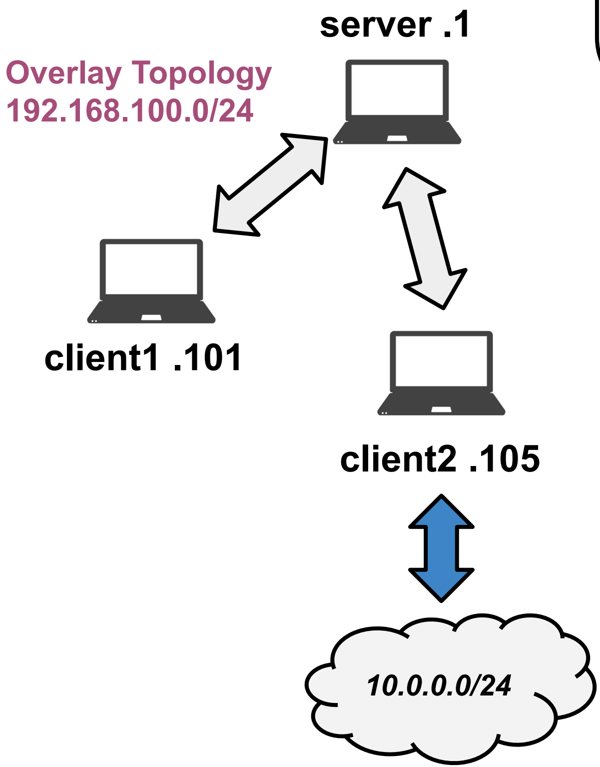

```
port 1194
proto udp
dev tun
ca ca.crt
cert server.crt
key server.key
dh dh.pem
server 192.168.100.0 255.255.255.0
push "route 192.168.0.0 255.255.255.0"
push "route 10.0.0.0 255.255.255.0"
route 10.0.0.0 255.255.255.0
client-config-dir ccd
client-to-client
keepalive 10 120
cipher AES-256-GCM
```



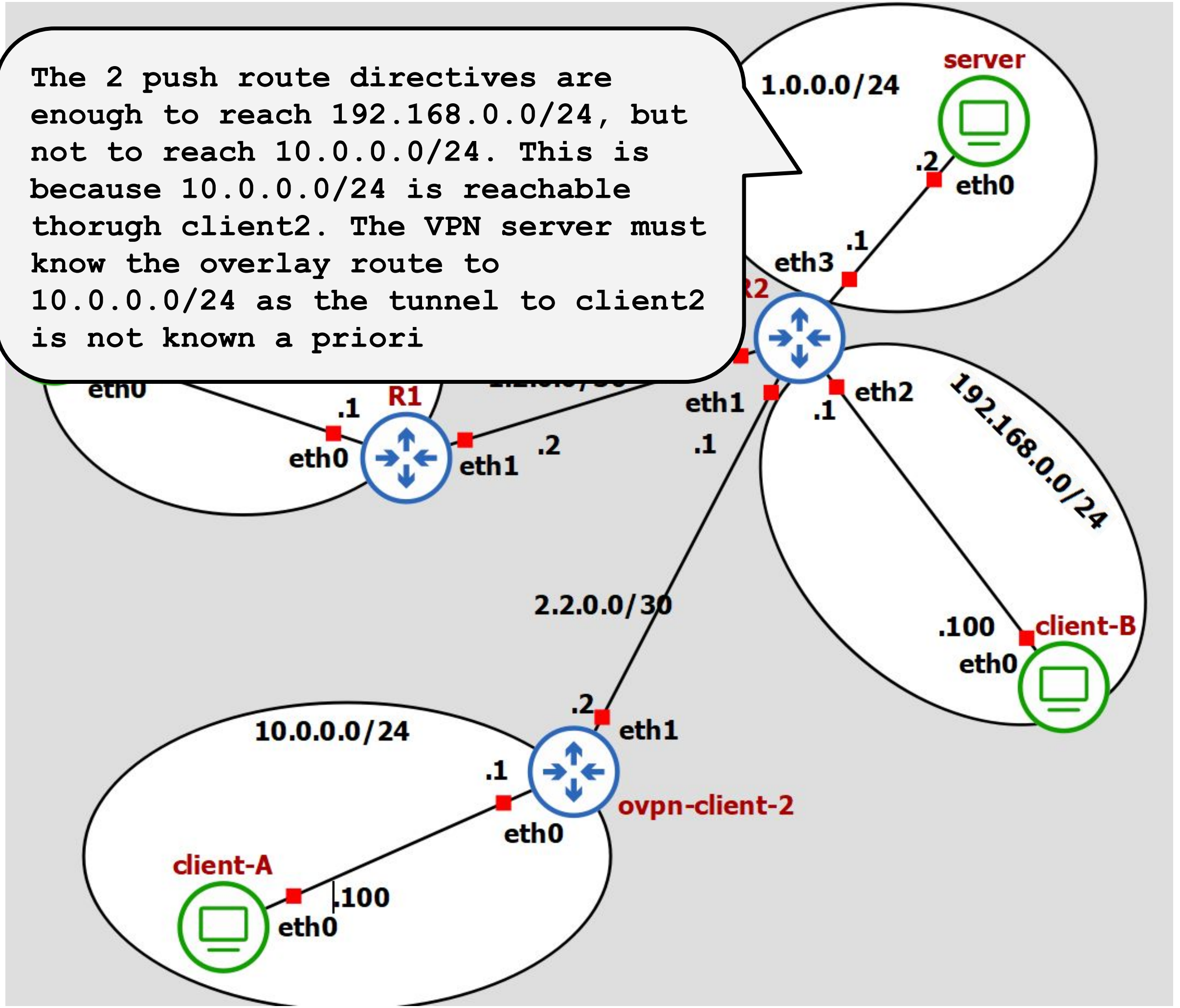
with this 2 directives the VPN server will push to every client a route to reach 192.168.0.0/24 and 10.0.0.0/24 “through the VPN”.
This will result in the following 2 routing entries (for example in client 1):

1. destination 192.168.0.0/24, next hop 192.168.100.102, oiface tun0
2. destination 10.0.0.0/24, next hop 192.168.100.102, oiface tun0

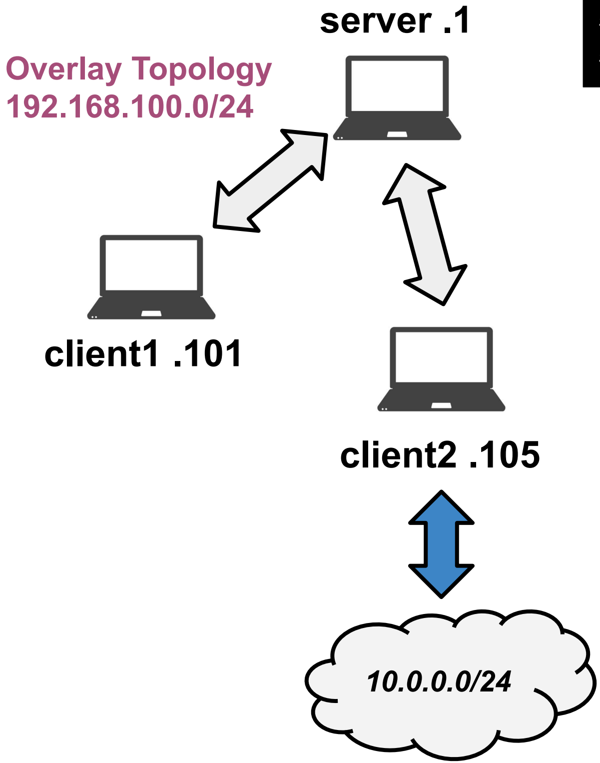
VPN Server



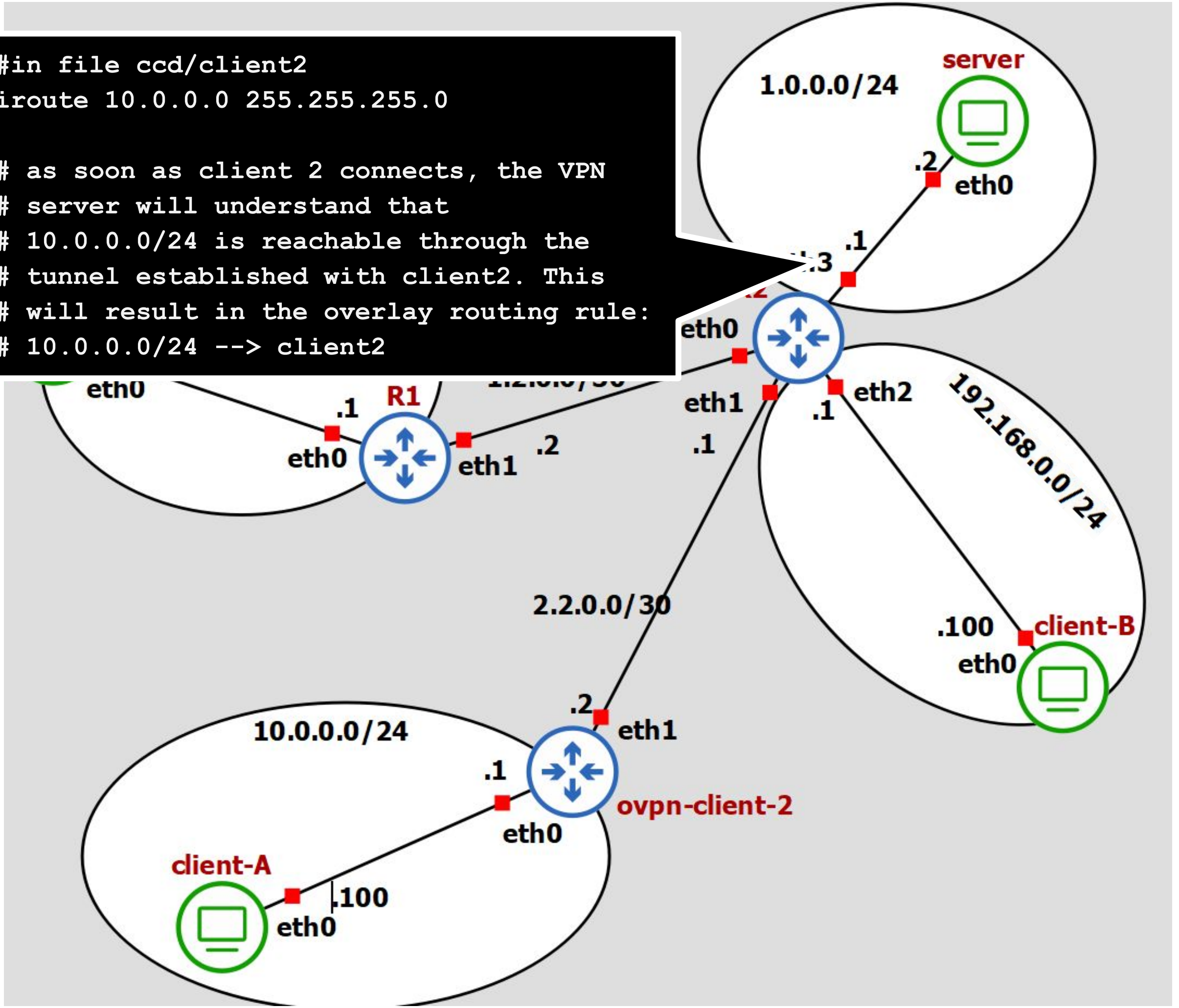
The 2 push route directives are enough to reach 192.168.0.0/24, but not to reach 10.0.0.0/24. This is because 10.0.0.0/24 is reachable thorugh client2. The VPN server must know the overlay route to 10.0.0.0/24 as the tunnel to client2 is not known a priori



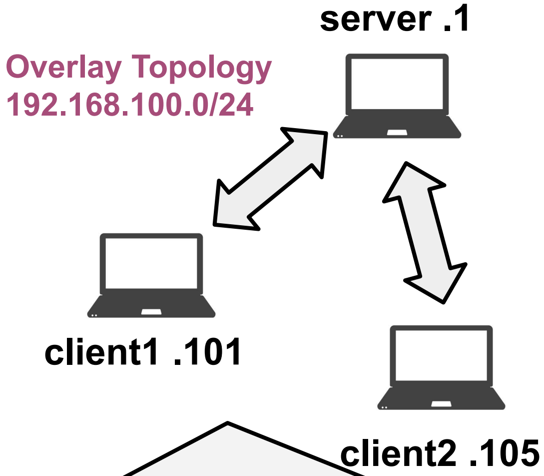
VPN Server

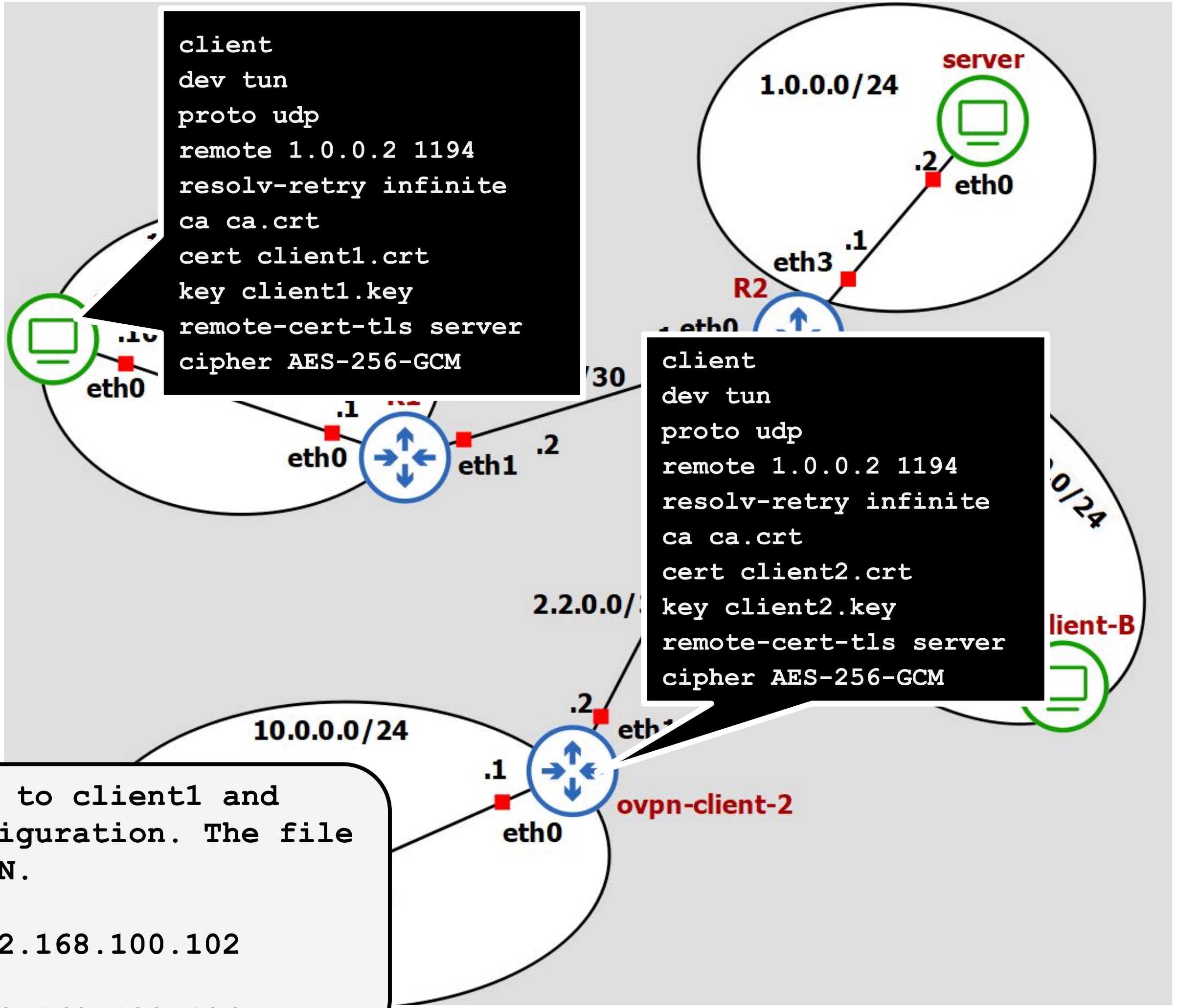



```
#in file ccd/client2
iroute 10.0.0.0 255.255.255.0

# as soon as client 2 connects, the VPN
# server will understand that
# 10.0.0.0/24 is reachable through the
# tunnel established with client2. This
# will result in the overlay routing rule:
# 10.0.0.0/24 --> client2
```



VPN Clients





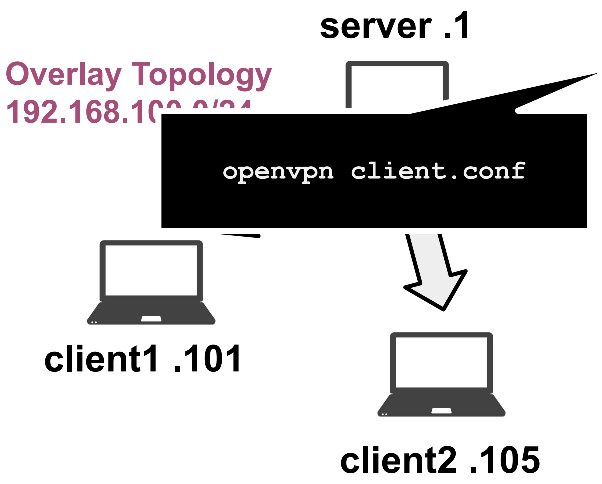

```
client
dev tun
proto udp
remote 1.0.0.2 1194
resolv-retry infinite
ca ca.crt
cert client1.crt
key client1.key
remote-cert-tls server
cipher AES-256-GCM
```

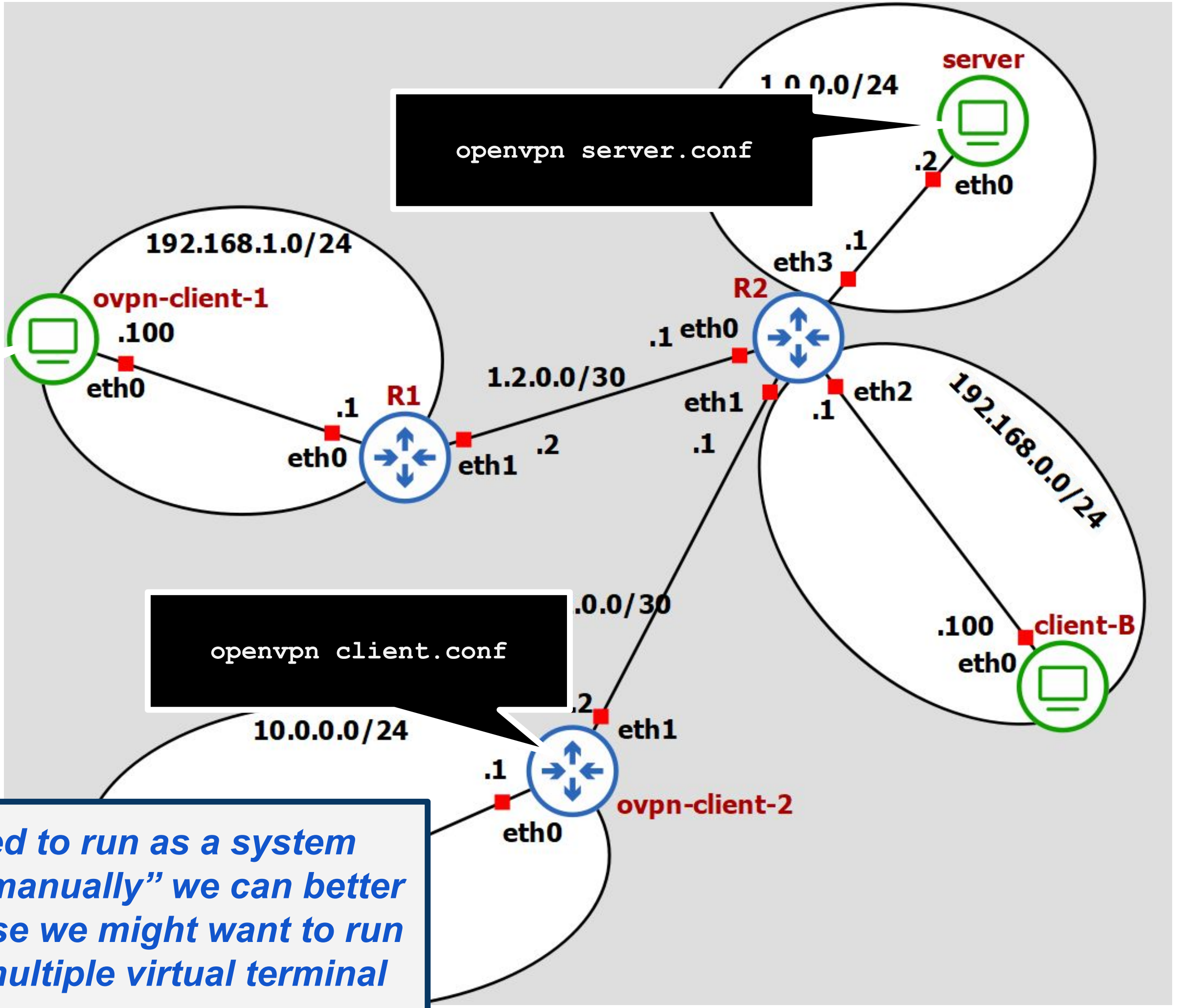
```
client
dev tun
proto udp
remote 1.0.0.2 1194
resolv-retry infinite
ca ca.crt
cert client2.crt
key client2.key
remote-cert-tls server
cipher AES-256-GCM
```

virtual IP addresses are assigned to client1 and client2 according to the CCD configuration. The file name must match the certificate CN.

```
#file ccd/client1
if-config-push 192.168.100.101 192.168.100.102
#file ccd/client2
if-config-push 192.168.100.105 192.168.100.106
```

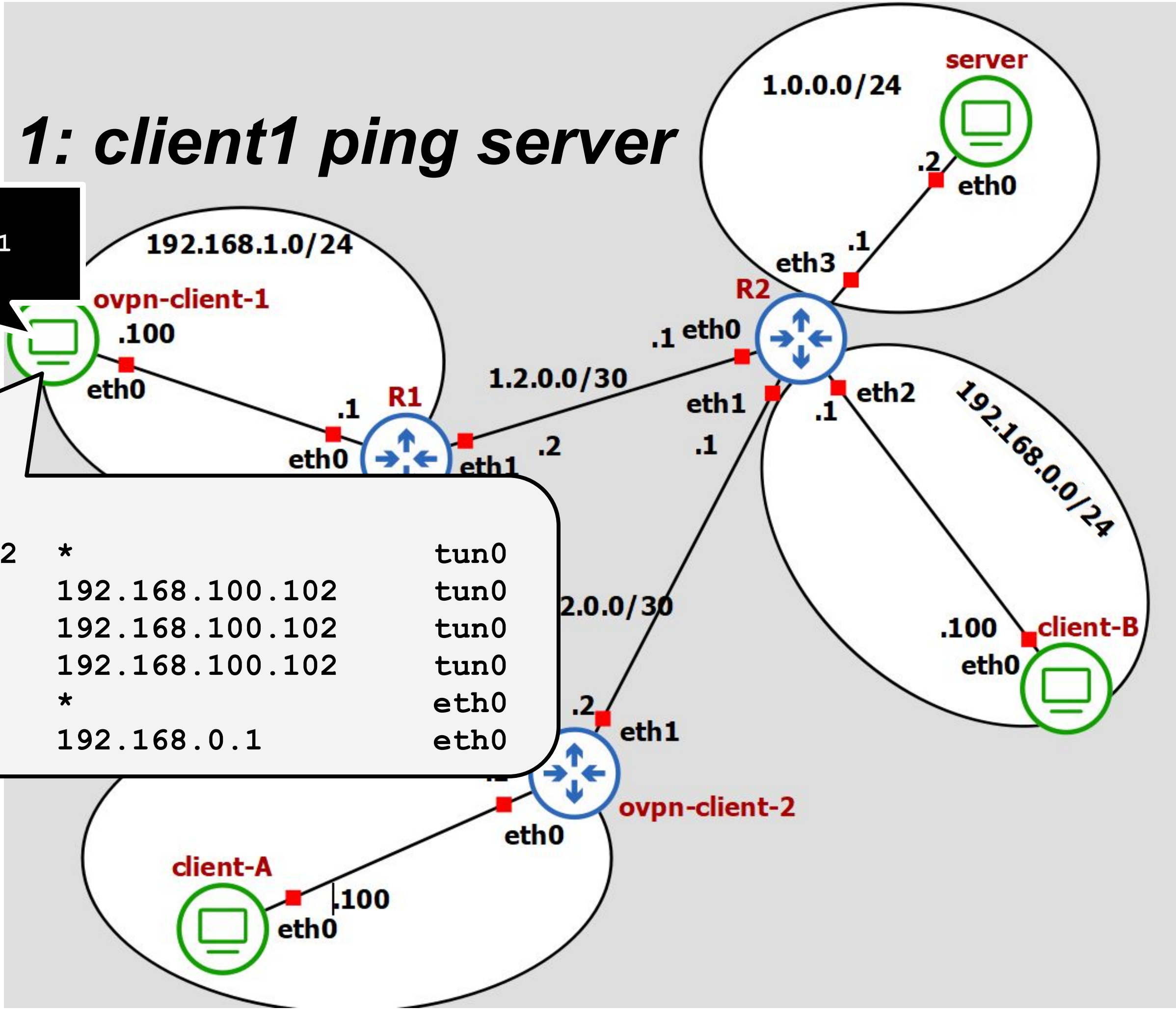
Start OpenVPN





OpenVPN can be also configured to run as a system service. By running the process "manually" we can better debug what is going on. In this case we might want to run screen or tmux in order to have multiple virtual terminal

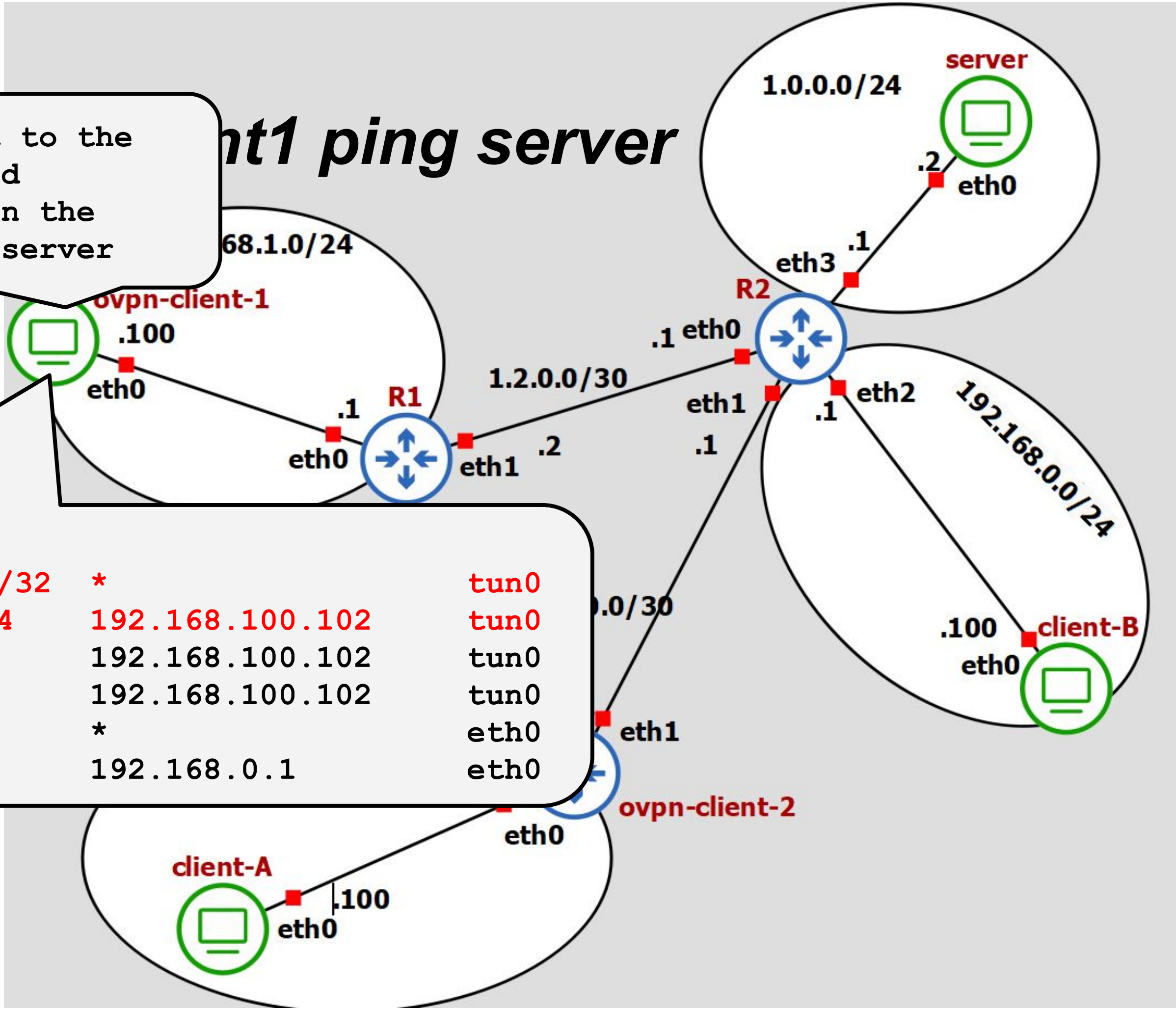
Test case 1: client1 ping server




```
ping 192.168.100.1
```

Routing table

192.168.100.102/32	*	tun0
192.168.100.0/24	192.168.100.102	tun0
192.168.0.0/24	192.168.100.102	tun0
10.0.0.0/24	192.168.100.102	tun0
192.168.1.0/24	*	eth0
0.0.0.0/0	192.168.0.1	eth0

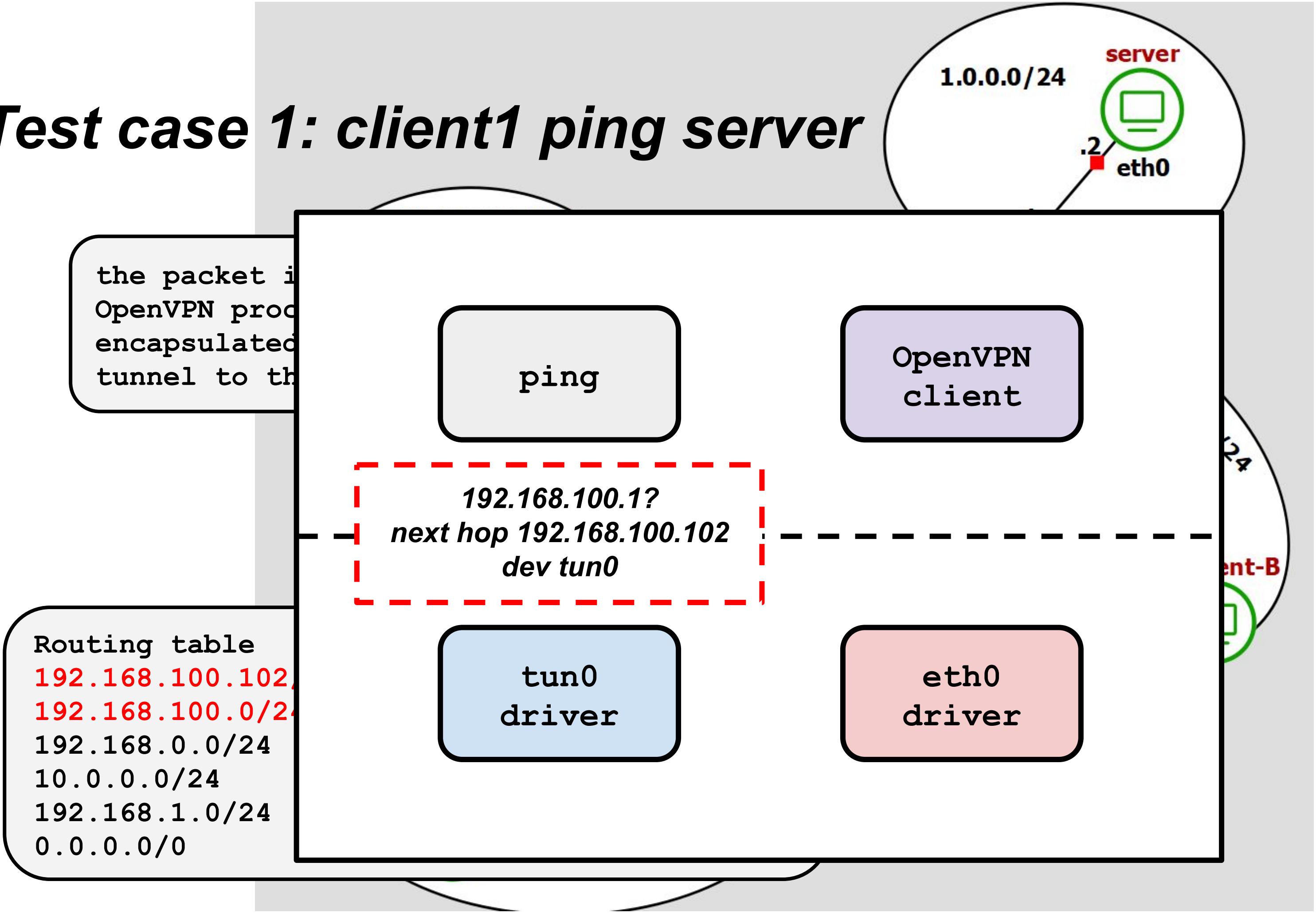


the packet is sent to the OpenVPN process and encapsulated within the tunnel to the VPN server

Routing table

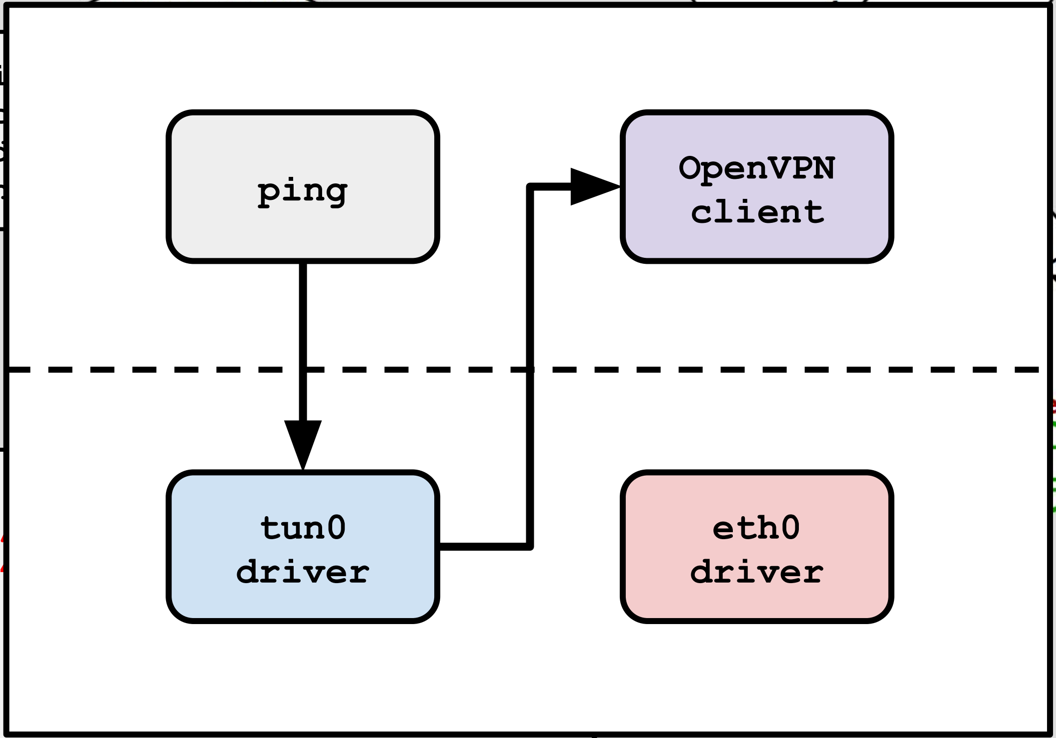
192.168.100.102/32	*	tun0
192.168.100.0/24	192.168.100.102	tun0
192.168.0.0/24	192.168.100.102	tun0
10.0.0.0/24	192.168.100.102	tun0
192.168.1.0/24	*	eth0
0.0.0.0/0	192.168.0.1	eth0

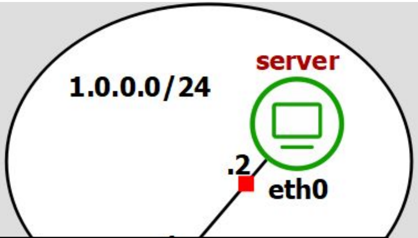
Test case 1: client1 ping server



Test case 1: client1 ping server

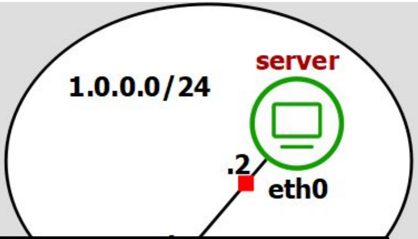
the packet i
OpenVPN proc
encapsulated
tunnel to th





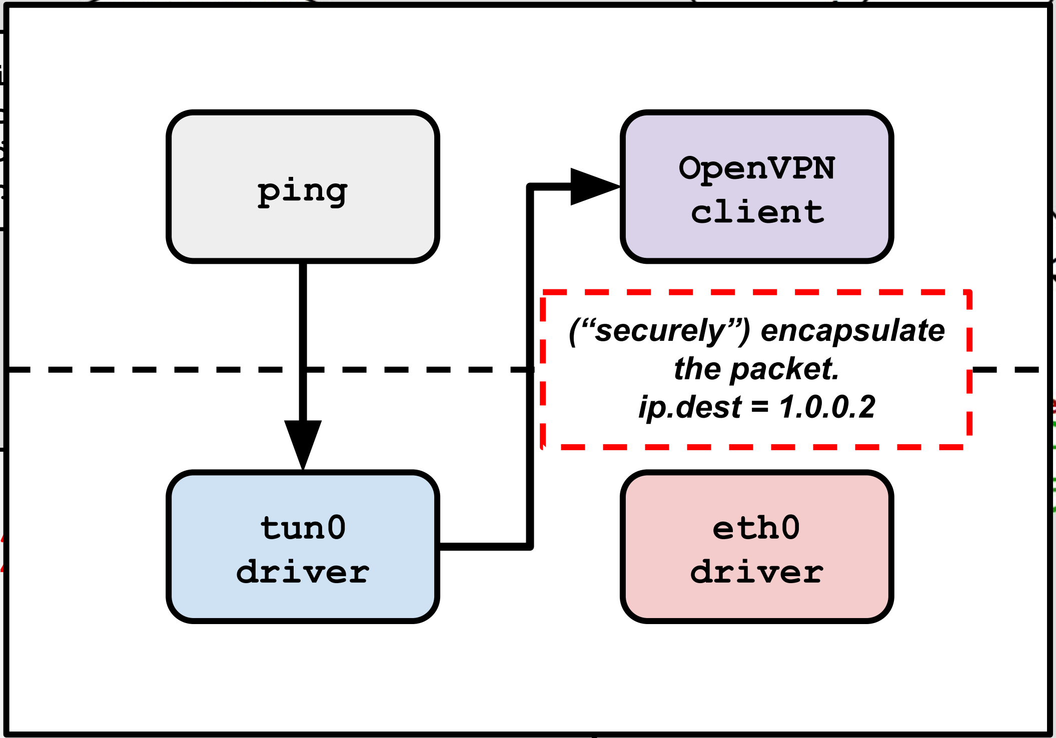
Routing table
192.168.100.102/
192.168.100.0/24
192.168.0.0/24
10.0.0.0/24
192.168.1.0/24
0.0.0.0/0

Test case 1: client1 ping server

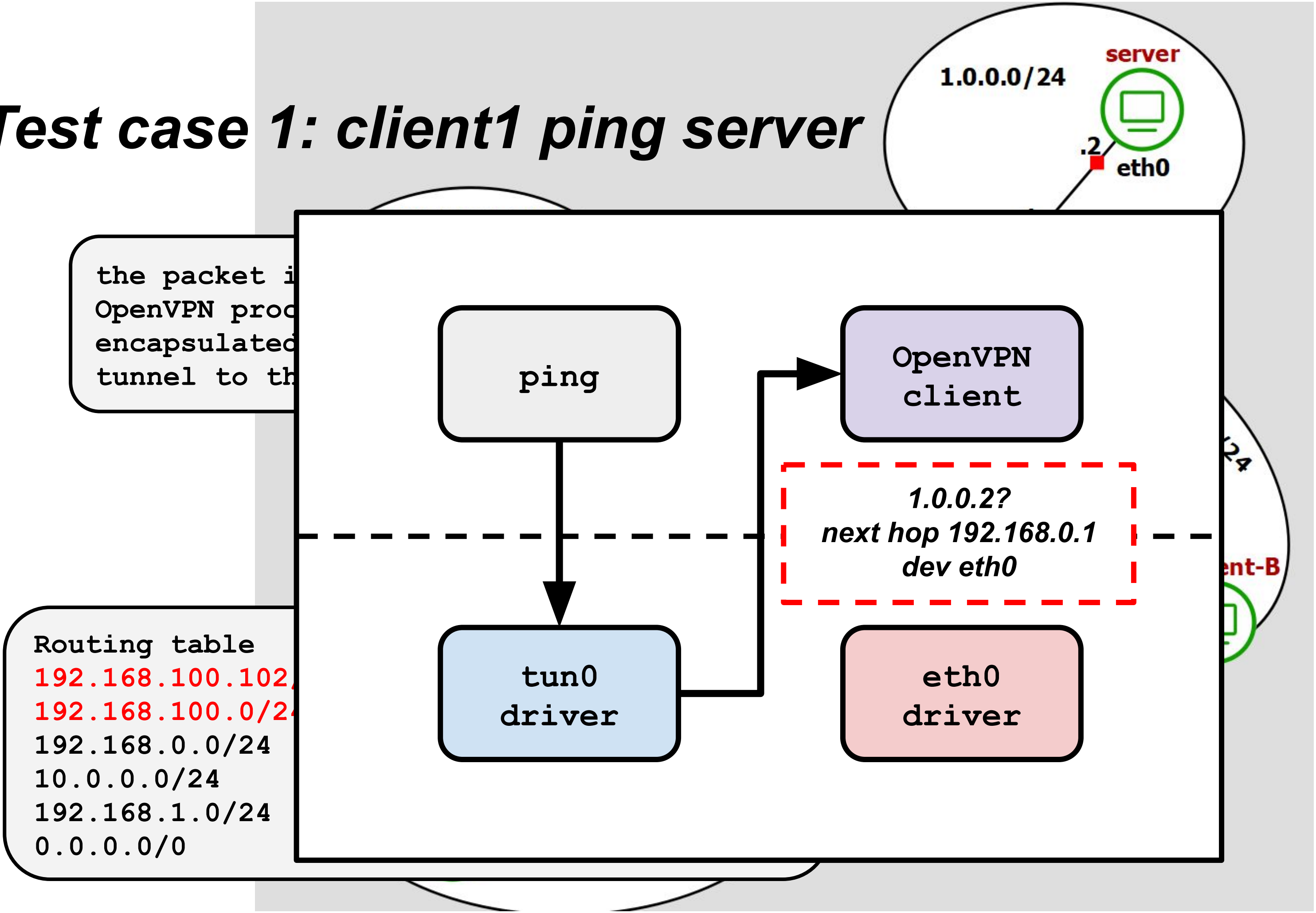


the packet i
OpenVPN proc
encapsulated
tunnel to th

Routing table
192.168.100.102/
192.168.100.0/24
192.168.0.0/24
10.0.0.0/24
192.168.1.0/24
0.0.0.0/0

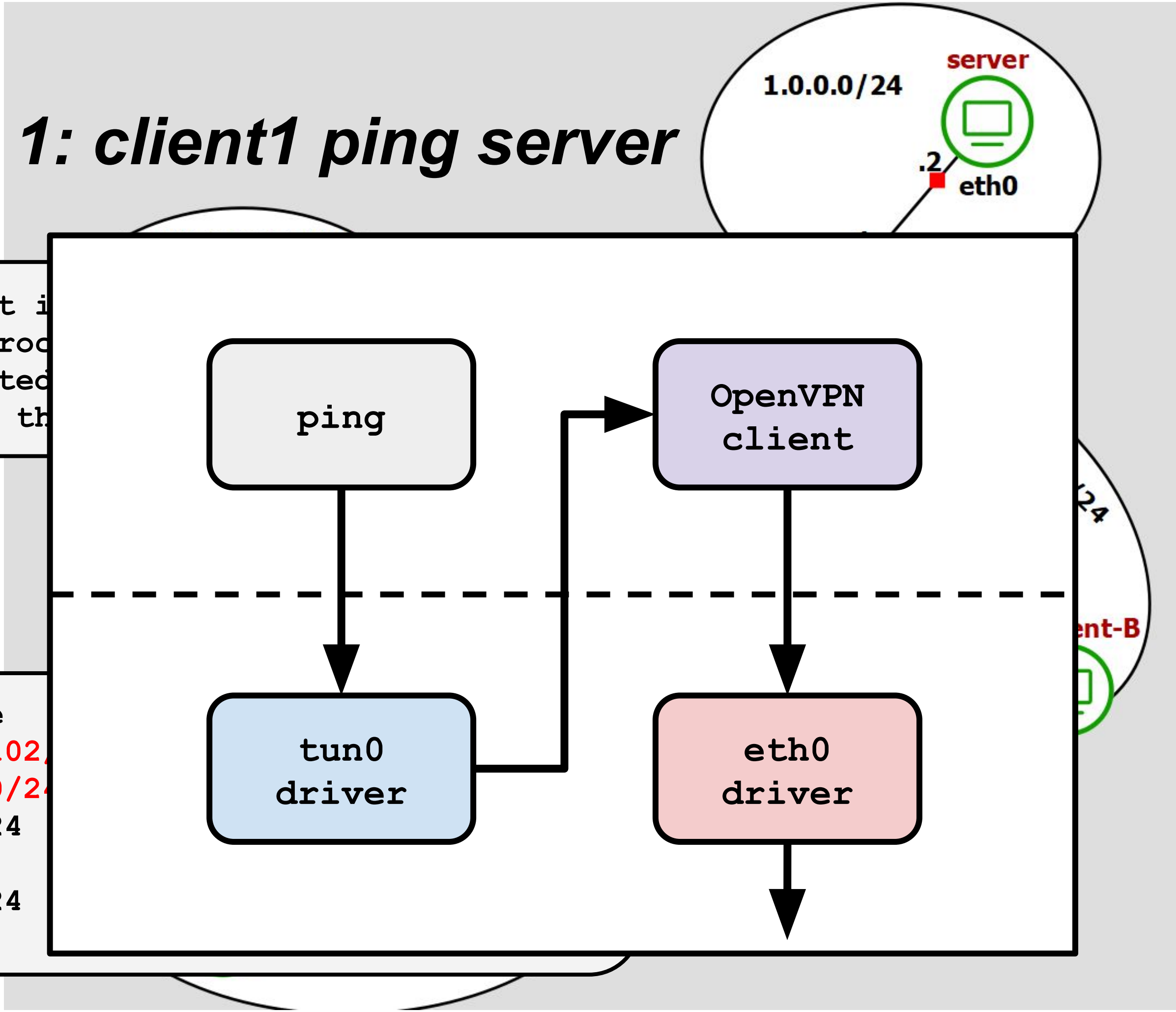


Test case 1: client1 ping server



Test case 1: client1 ping server

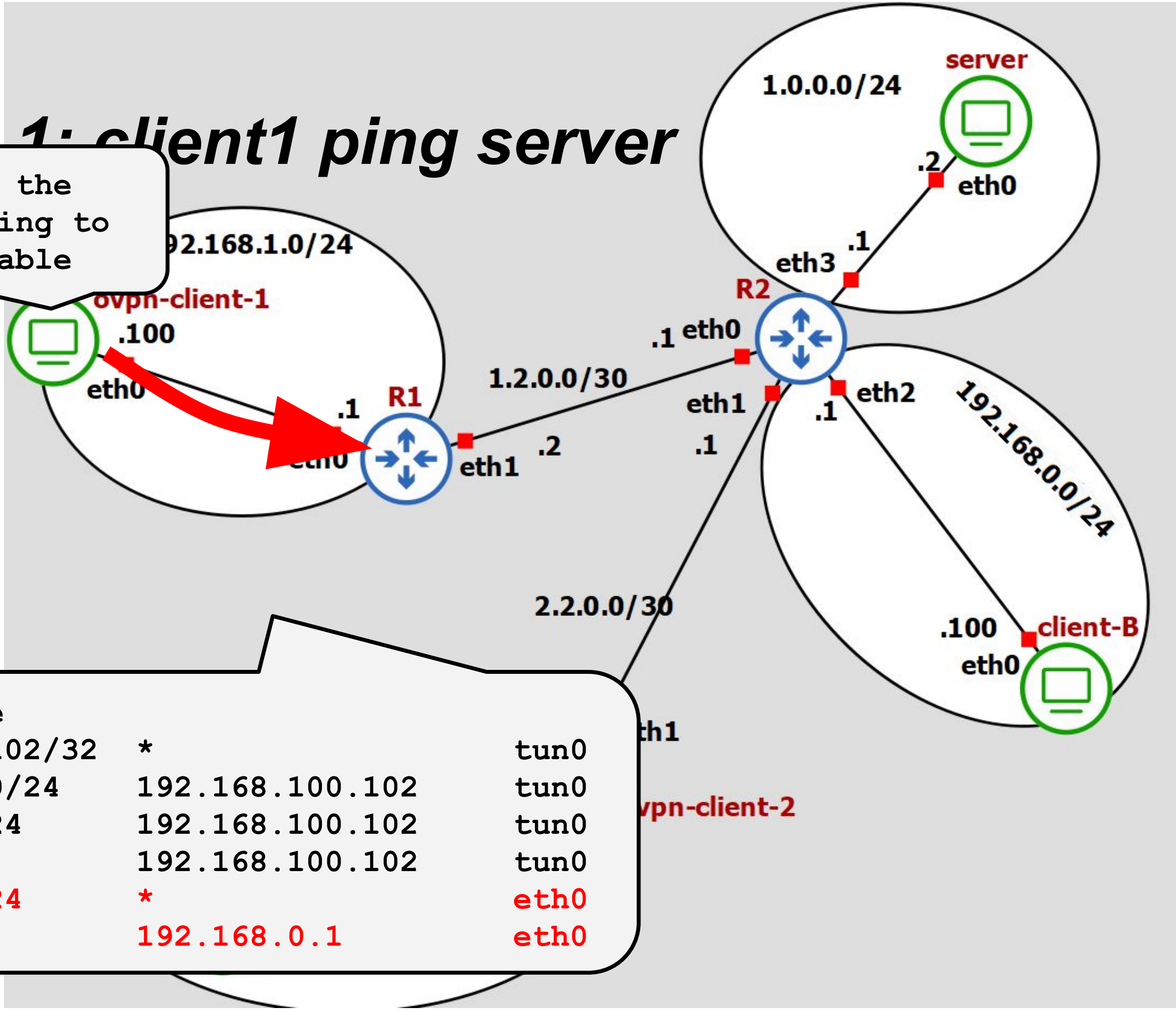
the packet i
OpenVPN proc
encapsulated
tunnel to th



Routing table
192.168.100.102
192.168.100.0/24
192.168.0.0/24
10.0.0.0/24
192.168.1.0/24
0.0.0.0/0

Test case 1: client1 ping server

OpenVPN sends the packet according to the routing table

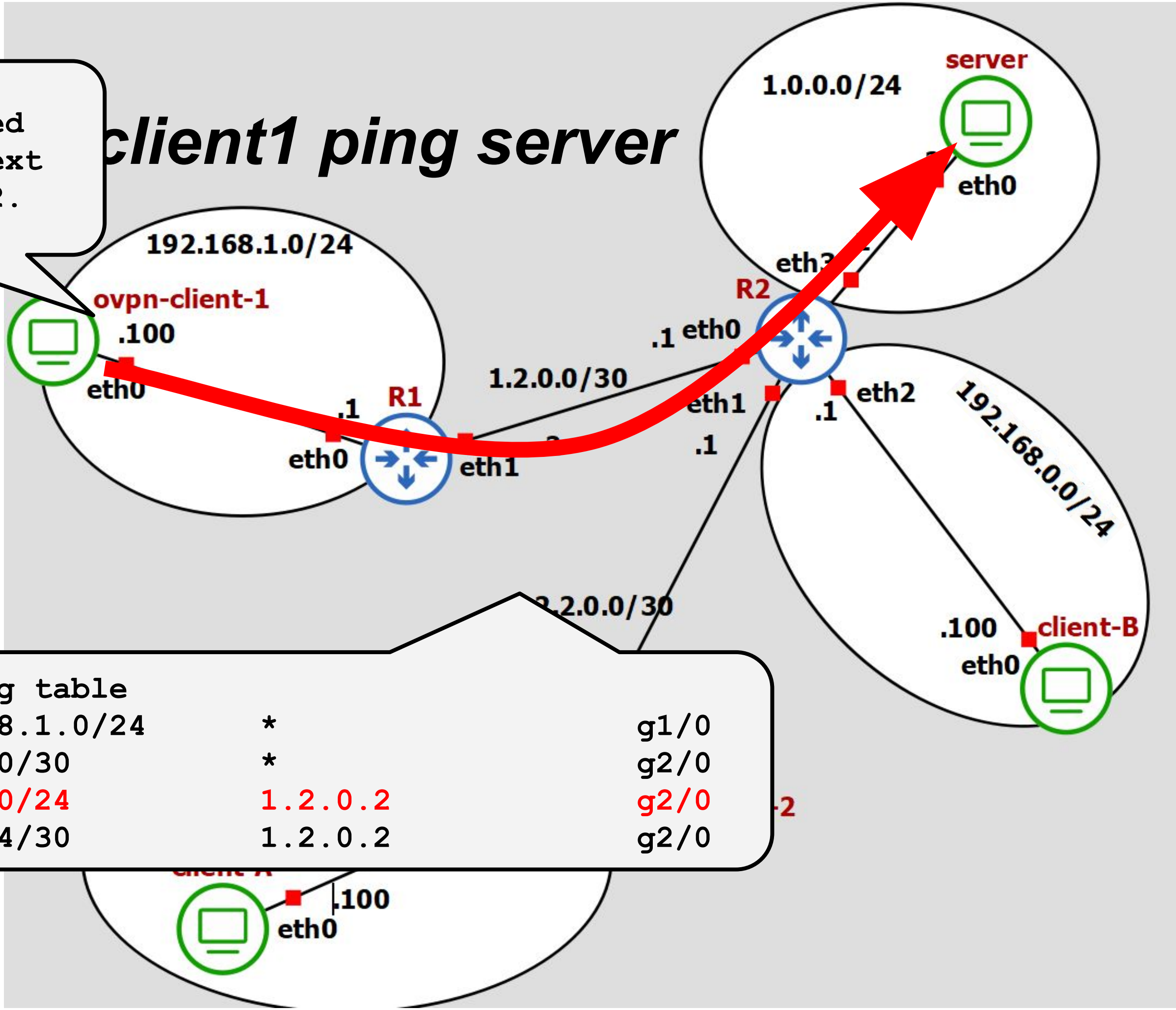


Routing table

192.168.100.102/32	*	tun0
192.168.100.0/24	192.168.100.102	tun0
192.168.0.0/24	192.168.100.102	tun0
10.0.0.0/24	192.168.100.102	tun0
192.168.1.0/24	*	eth0
0.0.0.0/0	192.168.0.1	eth0

client1 ping server

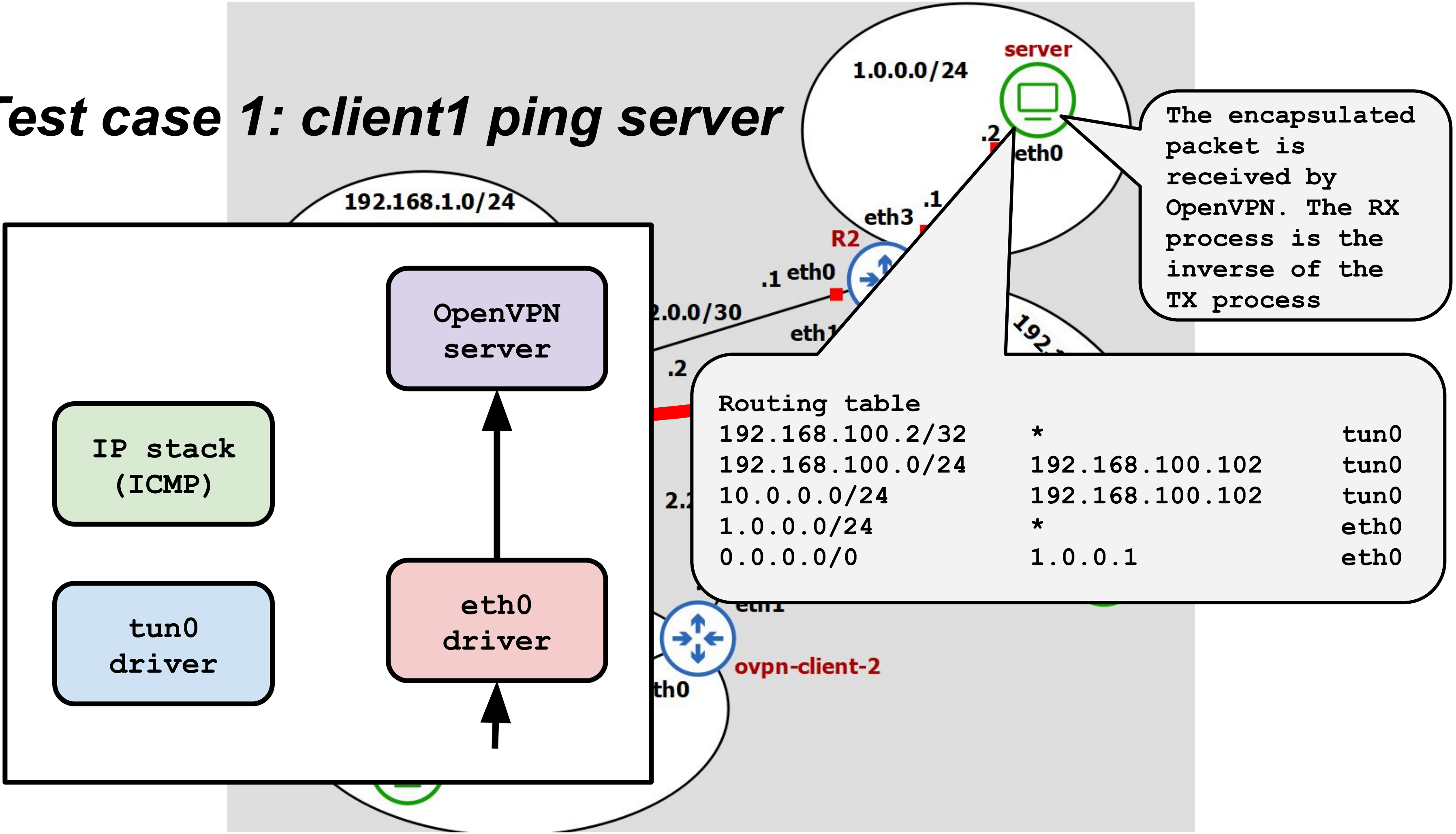
the packet is NATed and sent to the next hop toward 2.0.0.2.



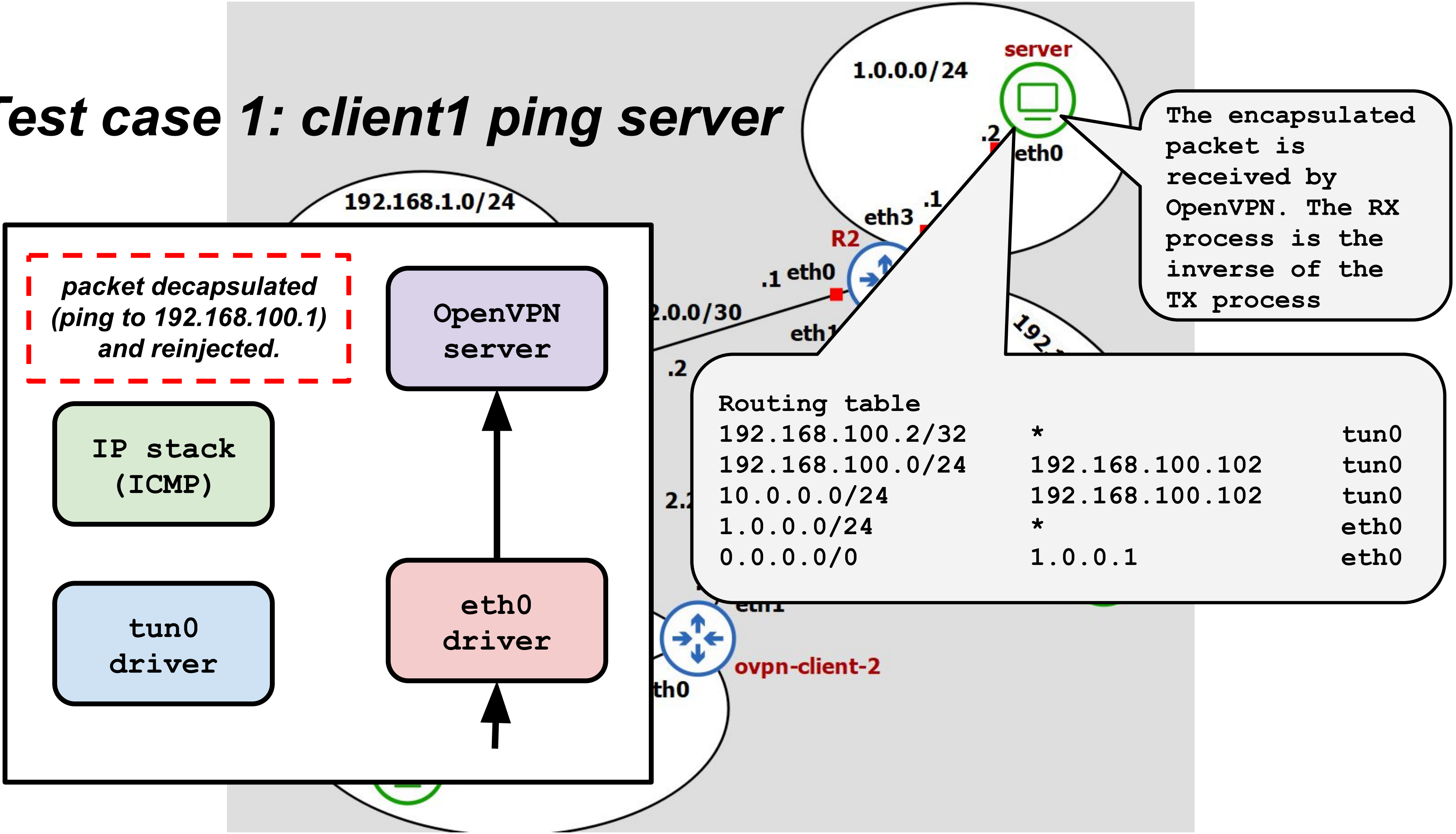
Routing table

192.168.1.0/24	*	g1/0
1.2.0.0/30	*	g2/0
2.0.0.0/24	1.2.0.2	g2/0
1.2.0.4/30	1.2.0.2	g2/0

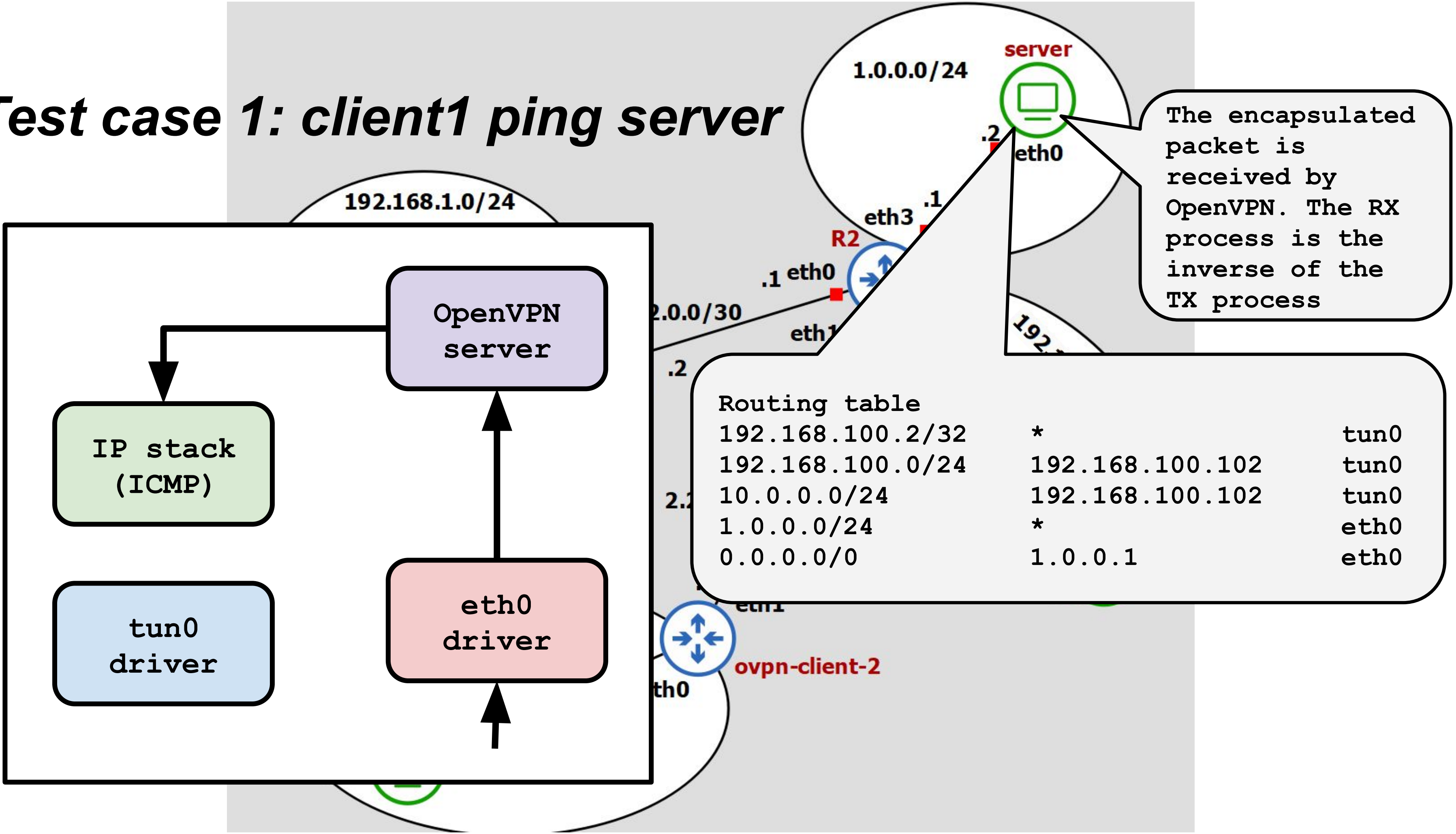
Test case 1: client1 ping server



Test case 1: client1 ping server

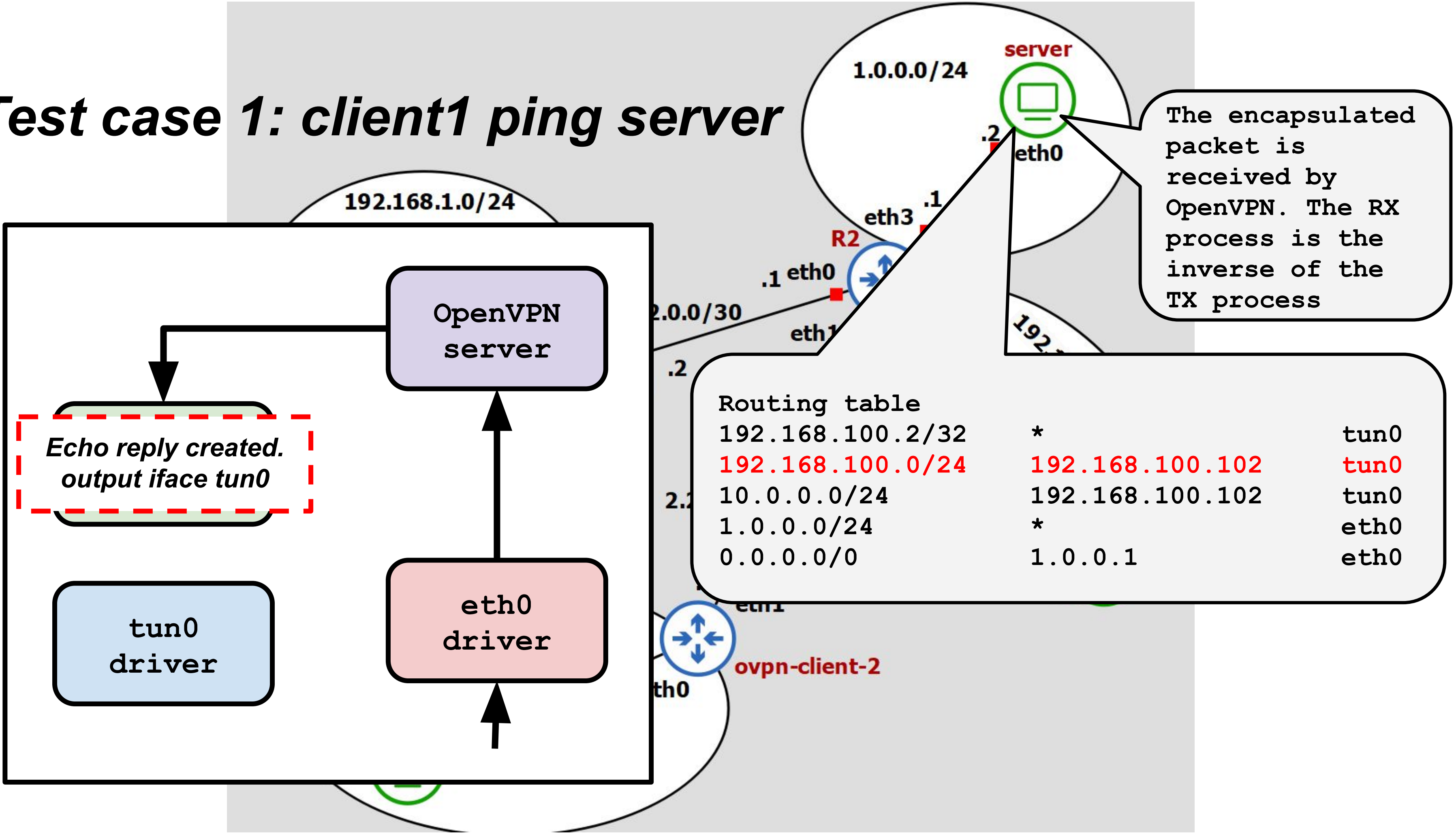


Test case 1: client1 ping server



192.168.100.2/32	*	tun0
192.168.100.0/24	192.168.100.102	tun0
10.0.0.0/24	192.168.100.102	tun0
1.0.0.0/24	*	eth0
0.0.0.0/0	1.0.0.1	eth0

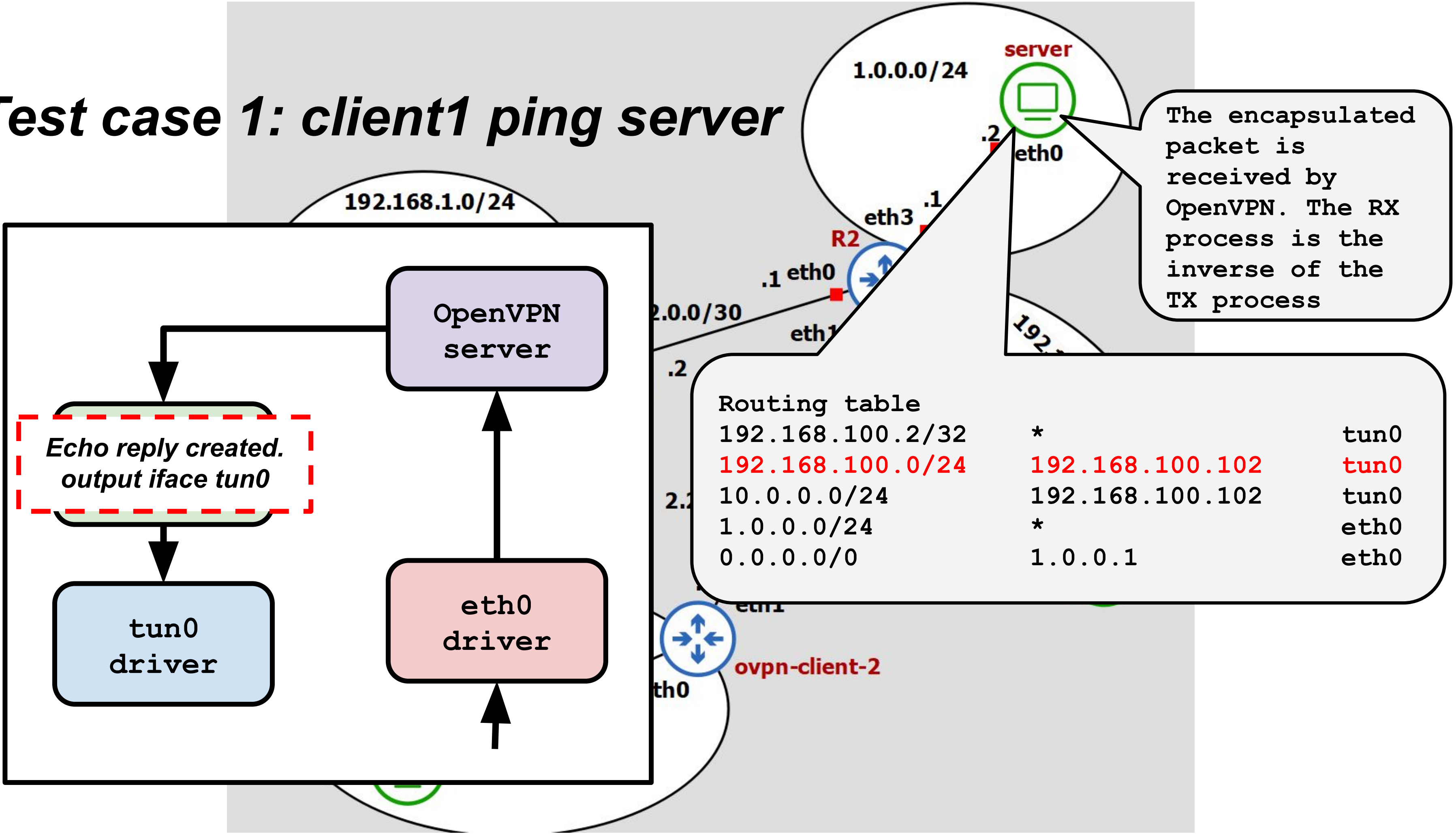
Test case 1: client1 ping server



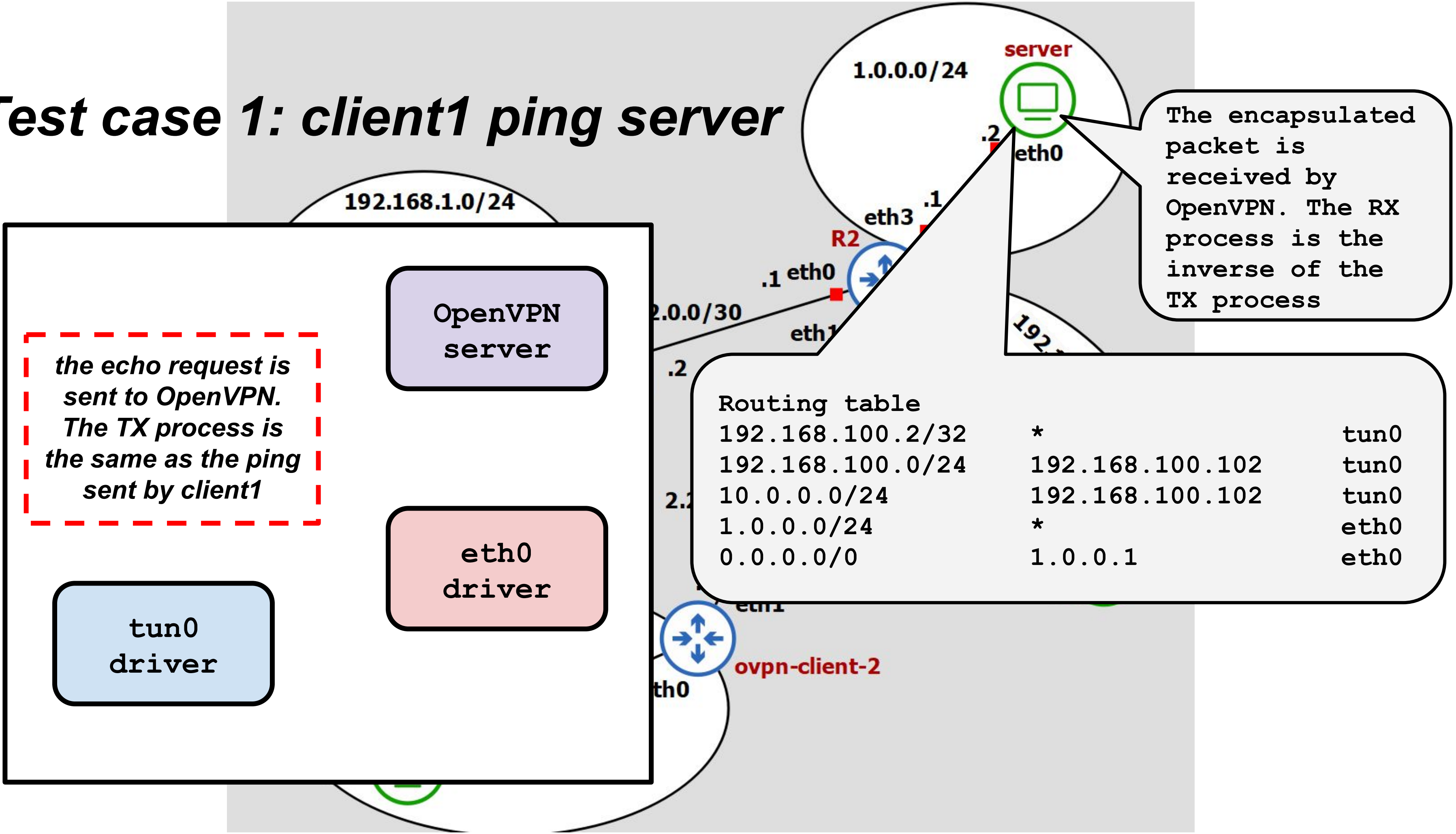
Routing table

192.168.100.2/32	*	tun0
192.168.100.0/24	192.168.100.102	tun0
10.0.0.0/24	192.168.100.102	tun0
1.0.0.0/24	*	eth0
0.0.0.0/0	1.0.0.1	eth0

Test case 1: client1 ping server



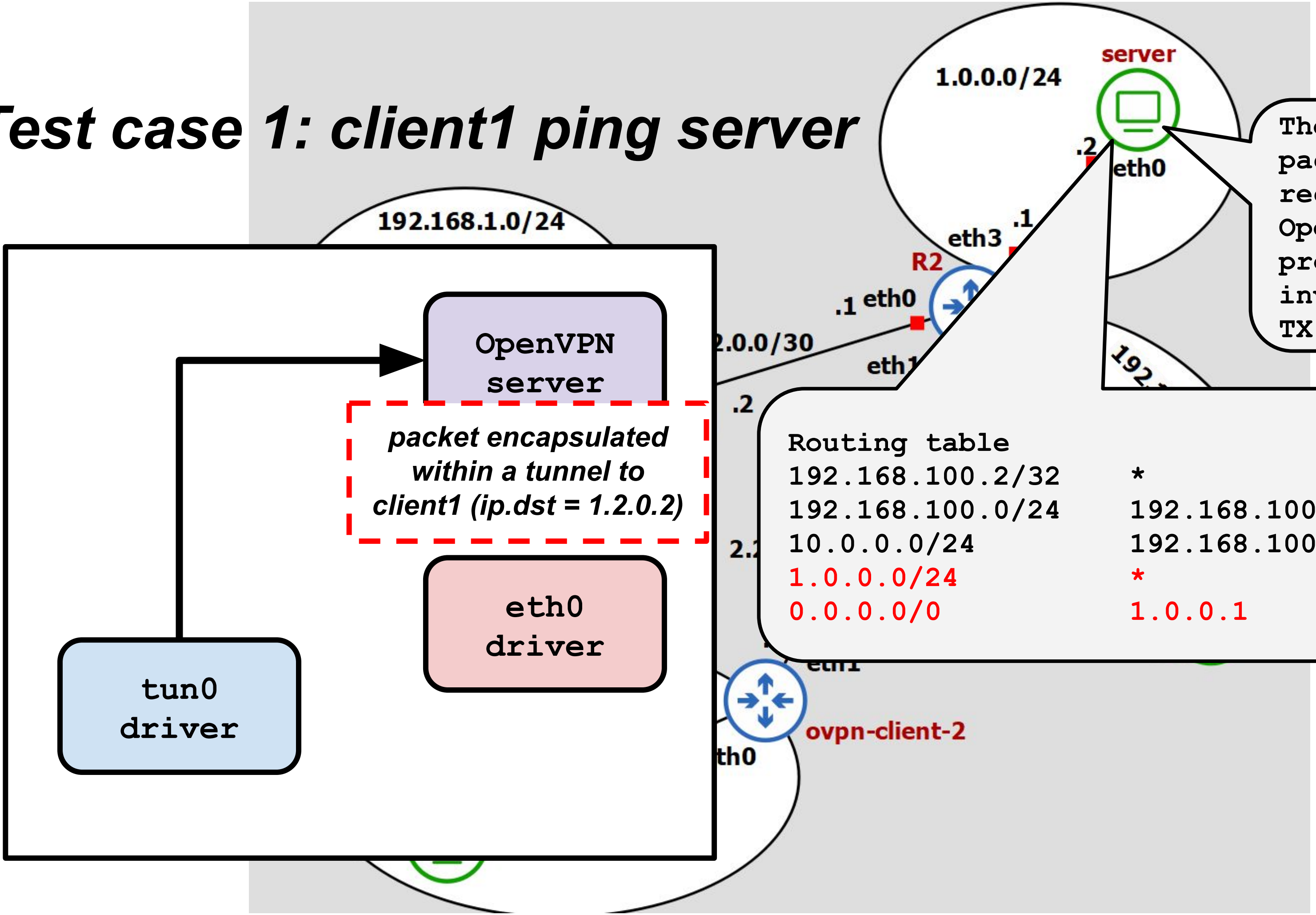
Test case 1: client1 ping server



Routing table

192.168.100.2/32	*	tun0
192.168.100.0/24	192.168.100.102	tun0
10.0.0.0/24	192.168.100.102	tun0
1.0.0.0/24	*	eth0
0.0.0.0/0	1.0.0.1	eth0

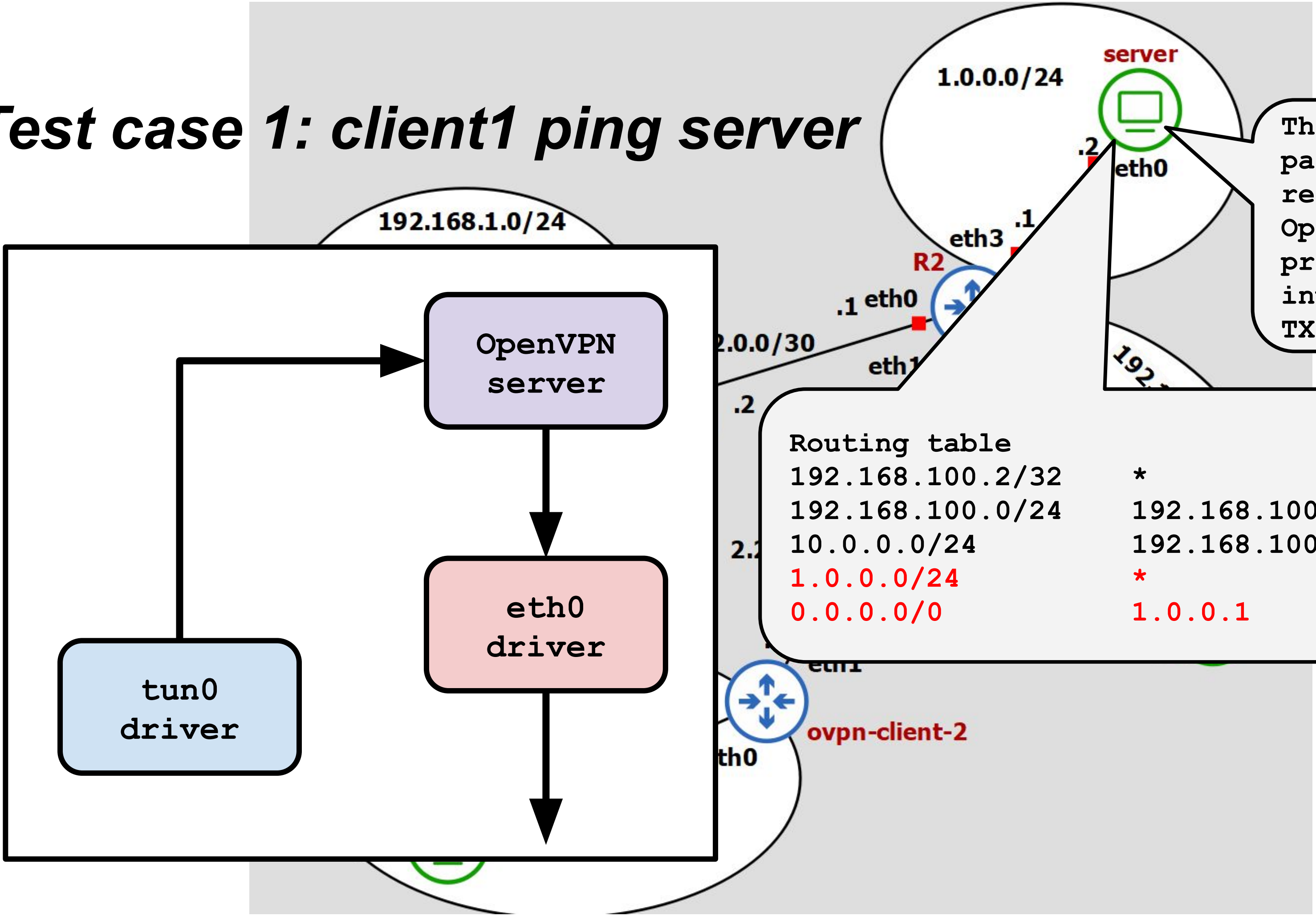
Test case 1: client1 ping server



The encapsulated packet is received by OpenVPN. The RX process is the inverse of the TX process

Routing table		
192.168.100.2/32	*	tun0
192.168.100.0/24	192.168.100.102	tun0
10.0.0.0/24	192.168.100.102	tun0
1.0.0.0/24	*	eth0
0.0.0.0/0	1.0.0.1	eth0

Test case 1: client1 ping server

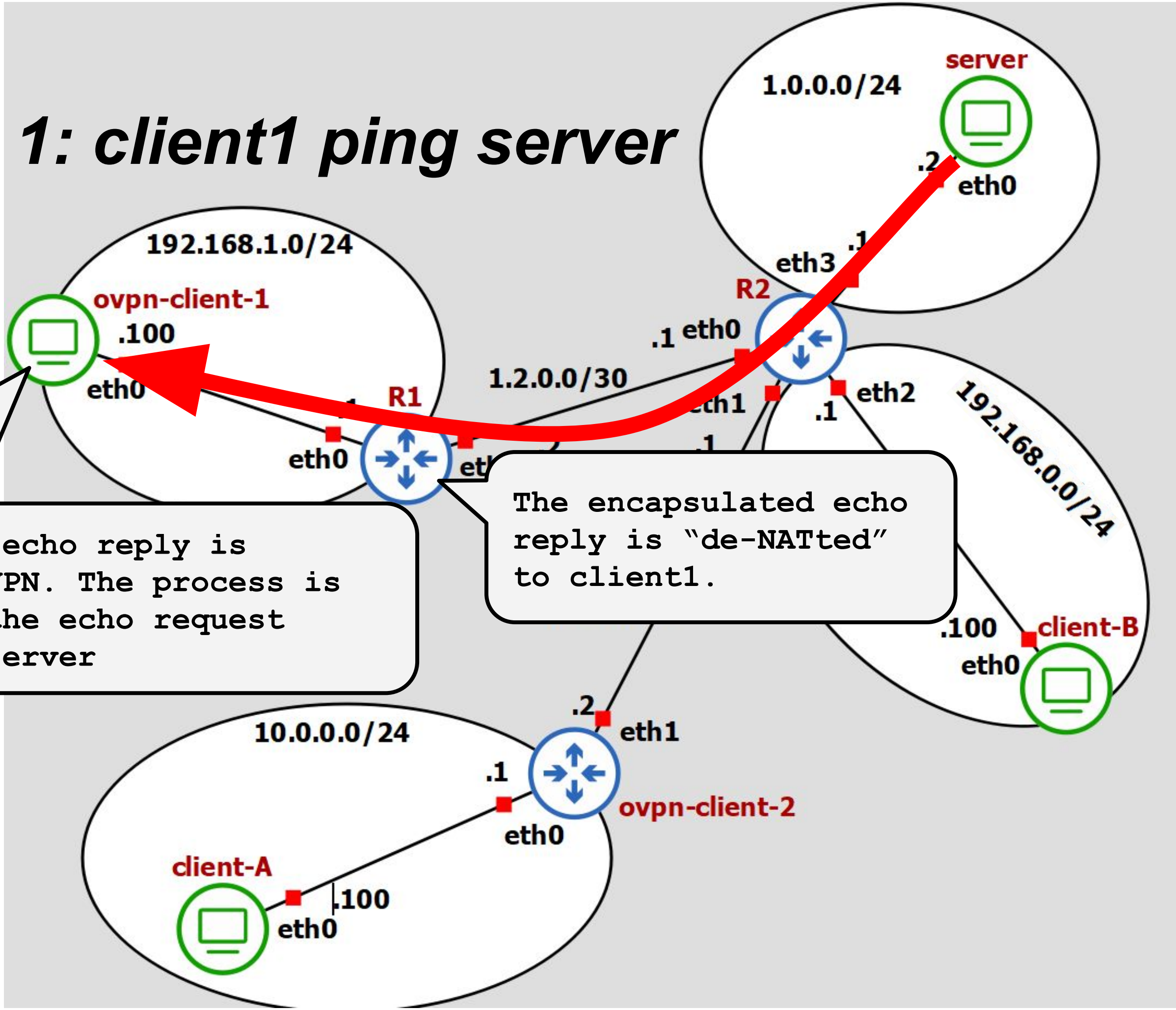


The encapsulated packet is received by OpenVPN. The RX process is the inverse of the TX process

Routing table

192.168.100.2/32	*	tun0
192.168.100.0/24	192.168.100.102	tun0
10.0.0.0/24	192.168.100.102	tun0
1.0.0.0/24	*	eth0
0.0.0.0/0	1.0.0.1	eth0

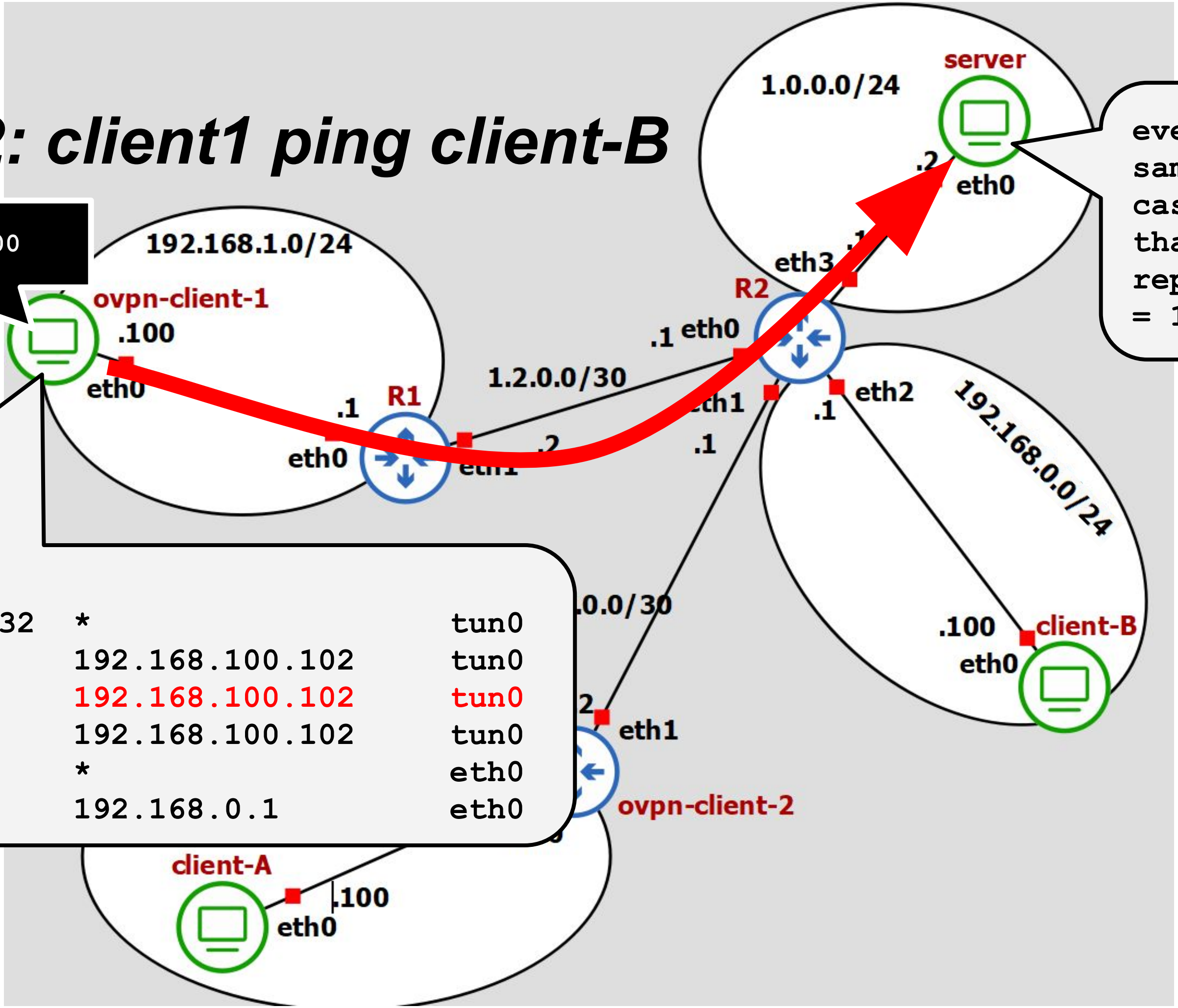
Test case 1: client1 ping server



The encapsulated echo reply is received by OpenVPN. The process is the same as for the echo request received by the server

The encapsulated echo reply is "de-NATted" to client1.

Test case 2: client1 ping client-B



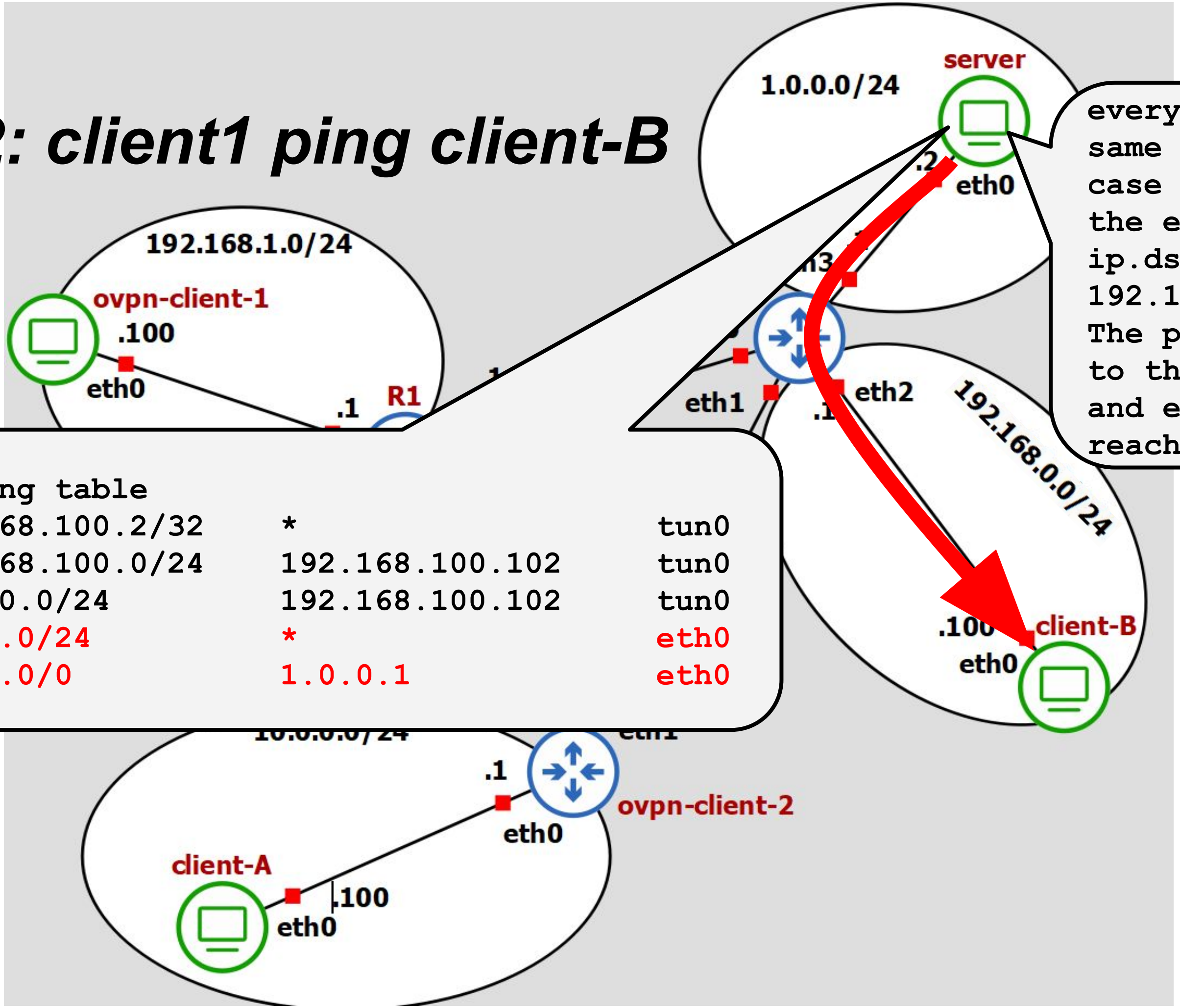

```
ping 192.168.0.100
```

Routing table

192.168.100.102/32	*	tun0
192.168.100.0/24	192.168.100.102	tun0
192.168.0.0/24	192.168.100.102	tun0
10.0.0.0/24	192.168.100.102	tun0
192.168.1.0/24	*	eth0
0.0.0.0/0	192.168.0.1	eth0

everyting is the same as test case 1 except that the ehoc reply has ip.dst = 192.168.0.100

Test case 2: client1 ping client-B

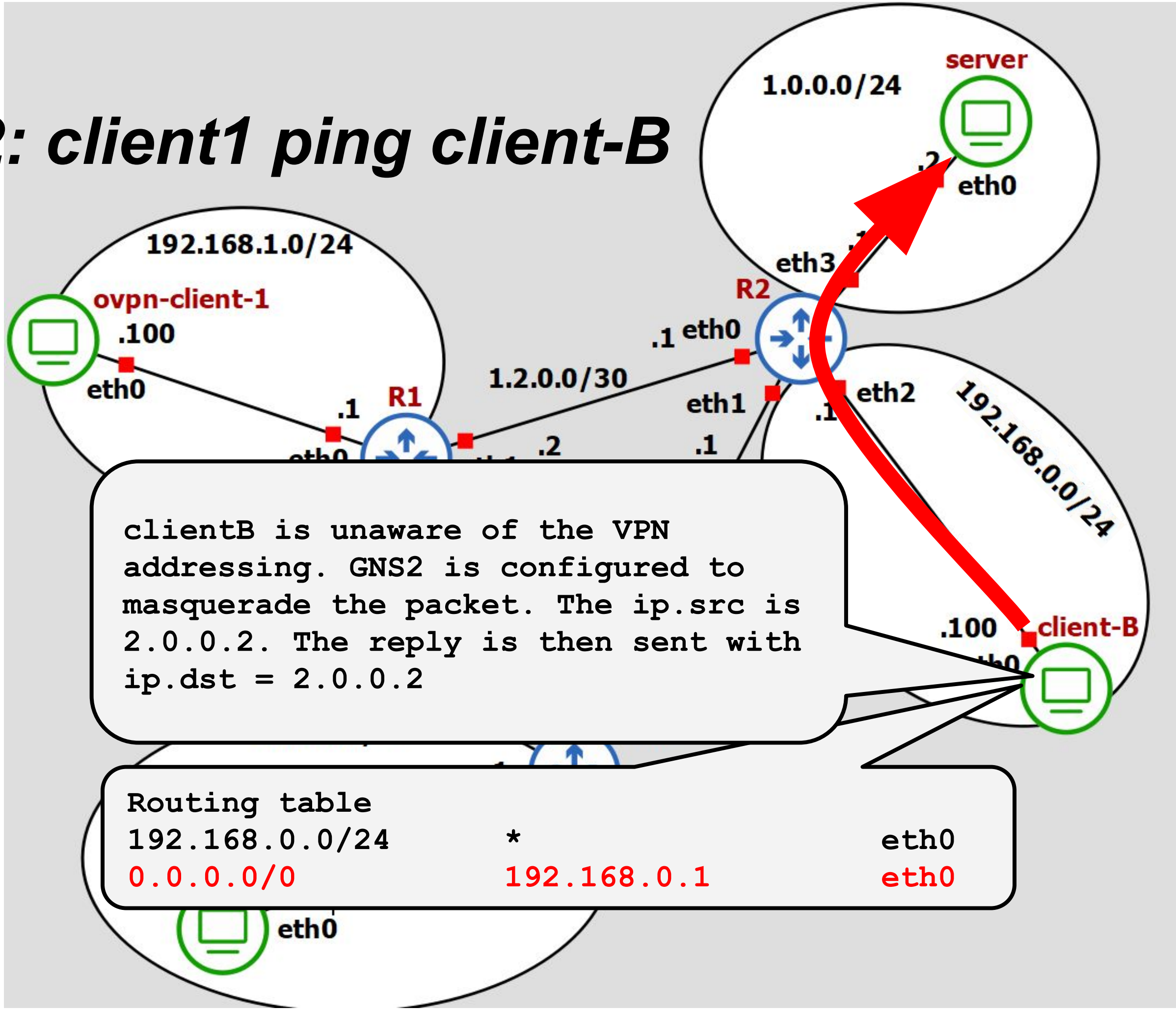


Routing table

192.168.100.2/32	*	tun0
192.168.100.0/24	192.168.100.102	tun0
10.0.0.0/24	192.168.100.102	tun0
1.0.0.0/24	*	eth0
0.0.0.0/0	1.0.0.1	eth0

everyting is the same as in test case 1 except that the echo reply has ip.dst = 192.168.0.100 The packet is sent to the default GW and eventually reaches tiny1

Test case 2: client1 ping client-B

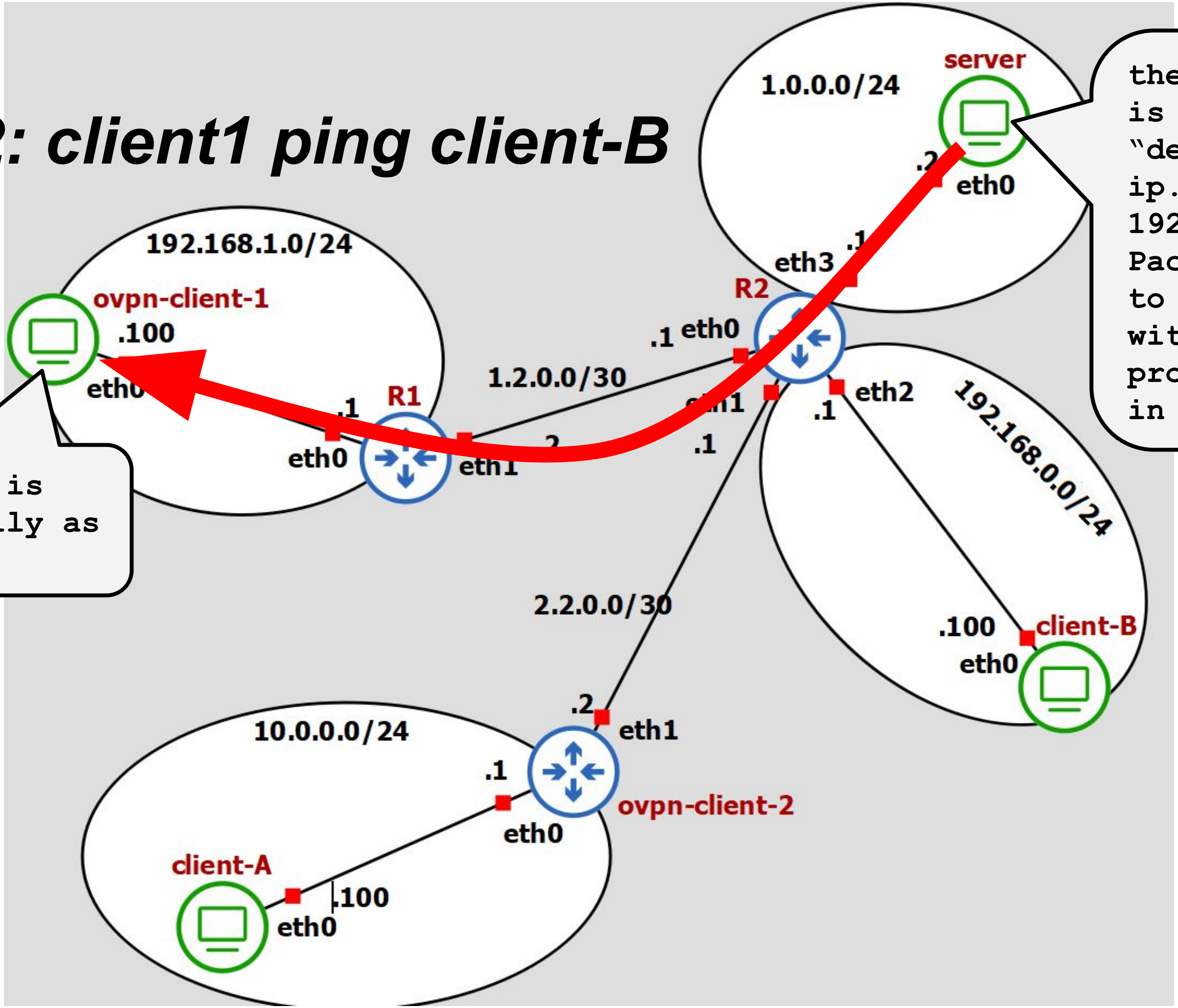


clientB is unaware of the VPN addressing. GNS2 is configured to masquerade the packet. The ip.src is 2.0.0.2. The reply is then sent with ip.dst = 2.0.0.2

Routing table

192.168.0.0/24	*	eth0
0.0.0.0/0	192.168.0.1	eth0

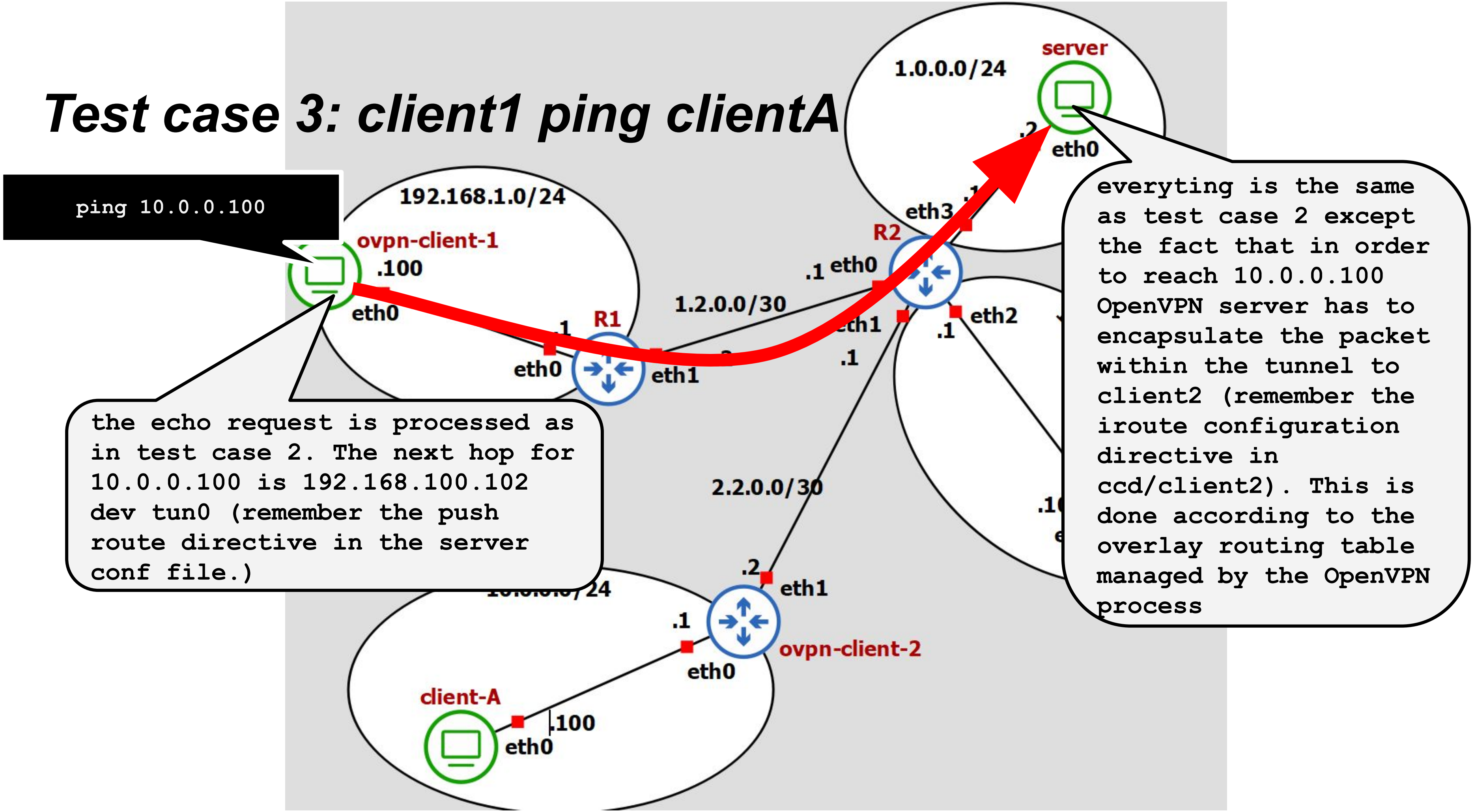
Test case 2: client1 ping client-B



the echo reply is processed locally as in test case 1

the echo reply is locally “de-NATted”. ip.dst = 192.168.100.101. Packet is sent to client1 within the proper tunnel as in test case 1

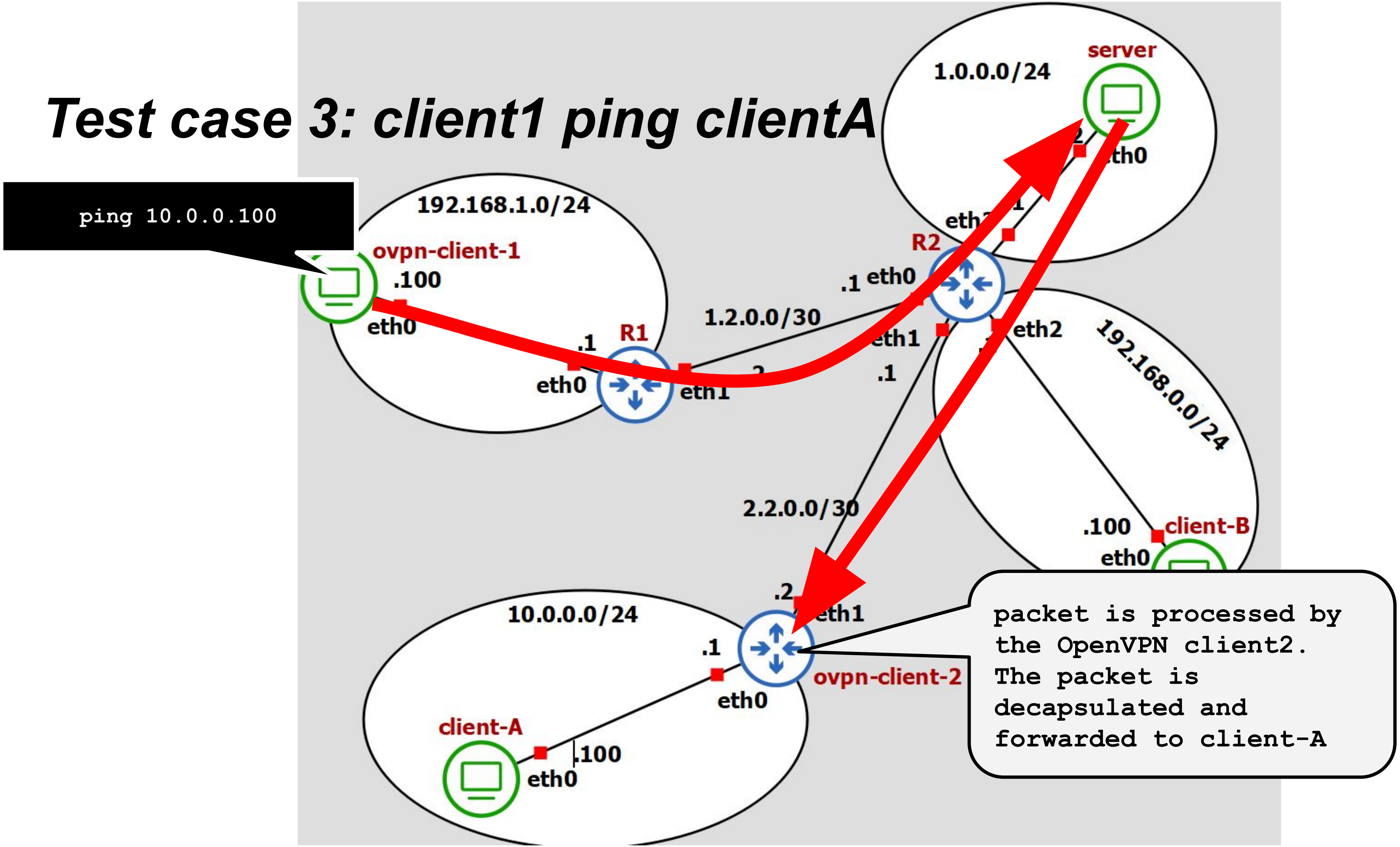
Test case 3: client1 ping clientA



the echo request is processed as in test case 2. The next hop for 10.0.0.100 is 192.168.100.102 dev tun0 (remember the push route directive in the server conf file.)

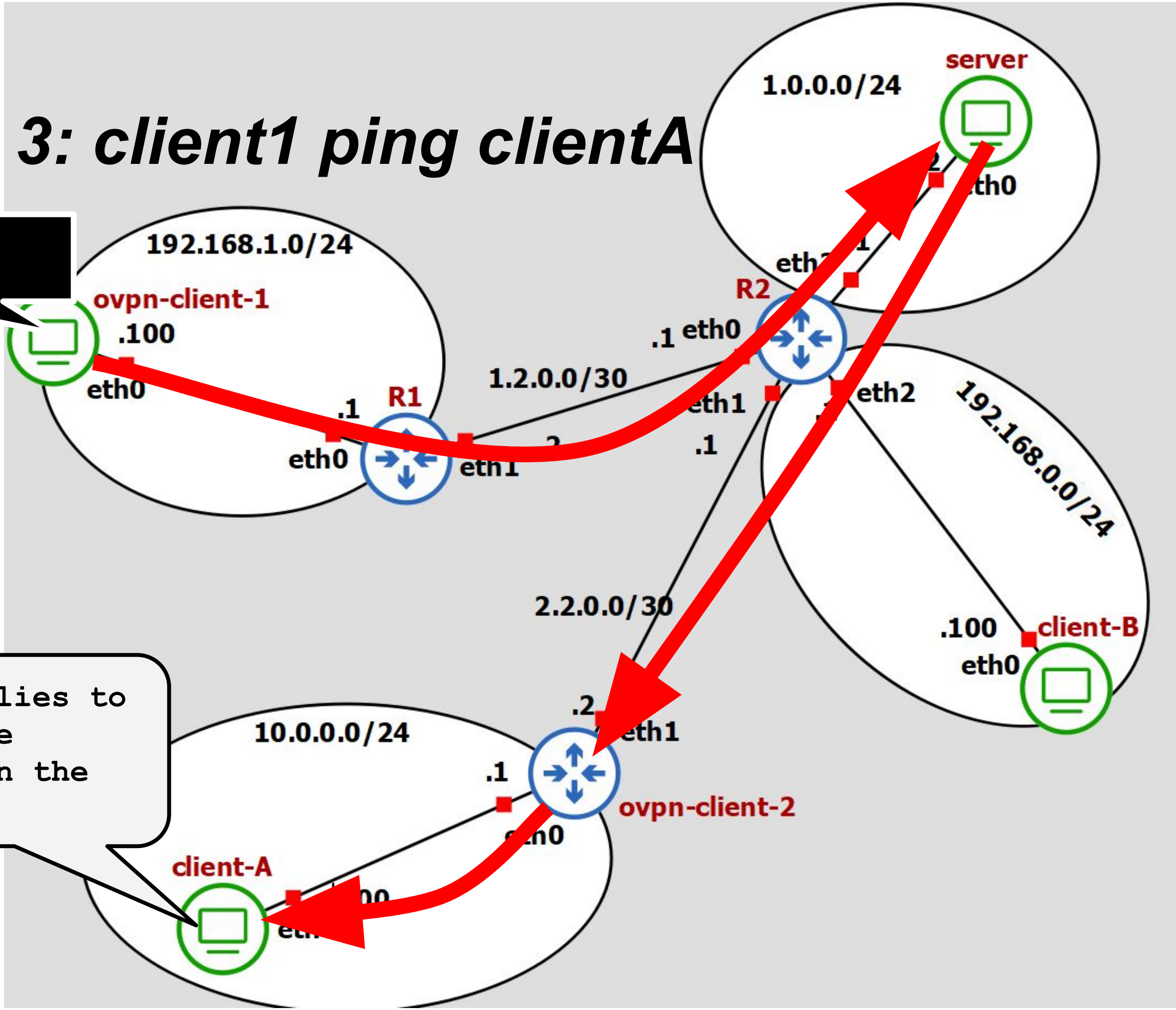
everyting is the same as test case 2 except the fact that in order to reach 10.0.0.100 OpenVPN server has to encapsulate the packet within the tunnel to client2 (remember the iroute configuration directive in ccd/client2). This is done according to the overlay routing table managed by the OpenVPN process

Test case 3: client1 ping clientA



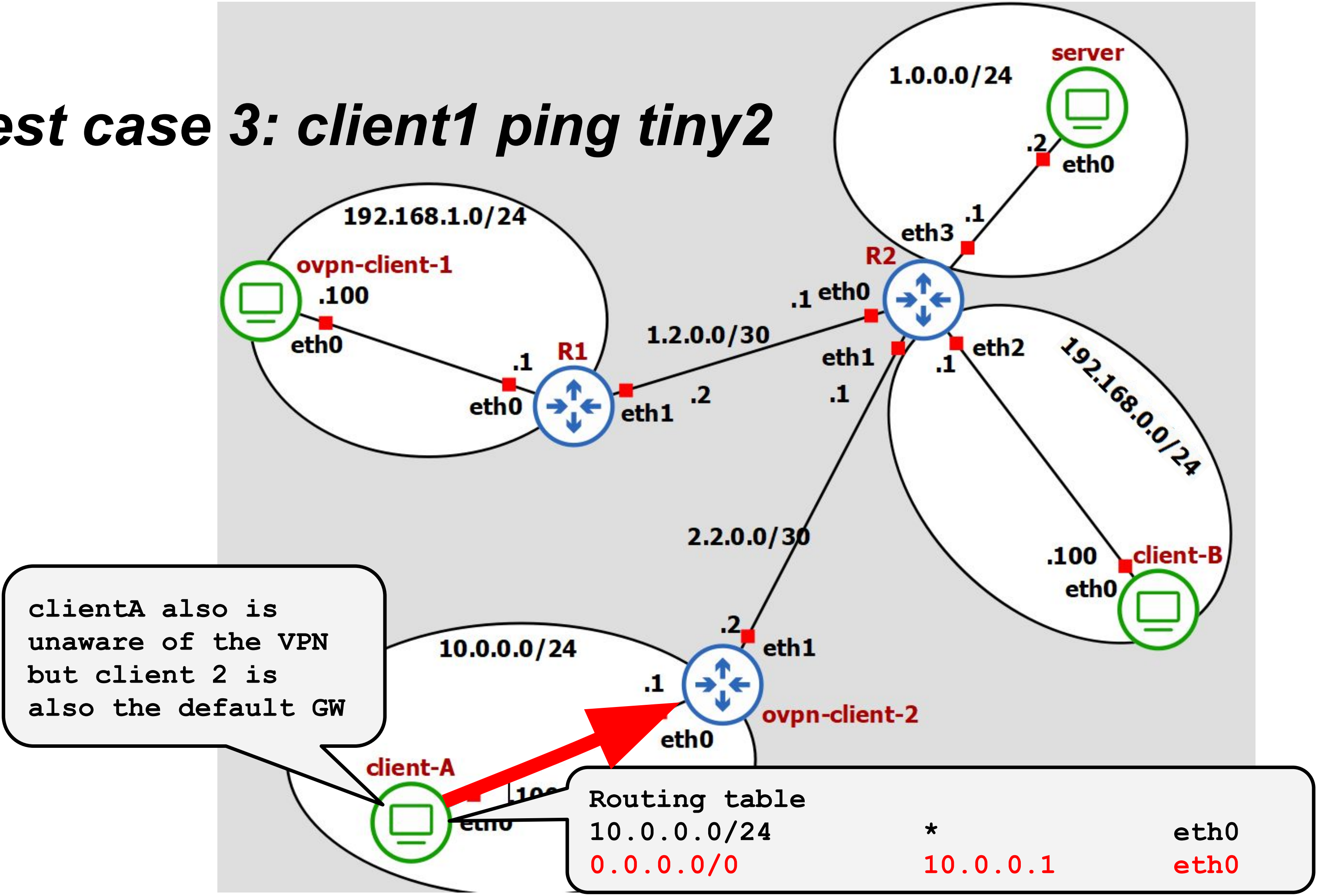
Test case 3: client1 ping clientA

ping 10.0.0.100

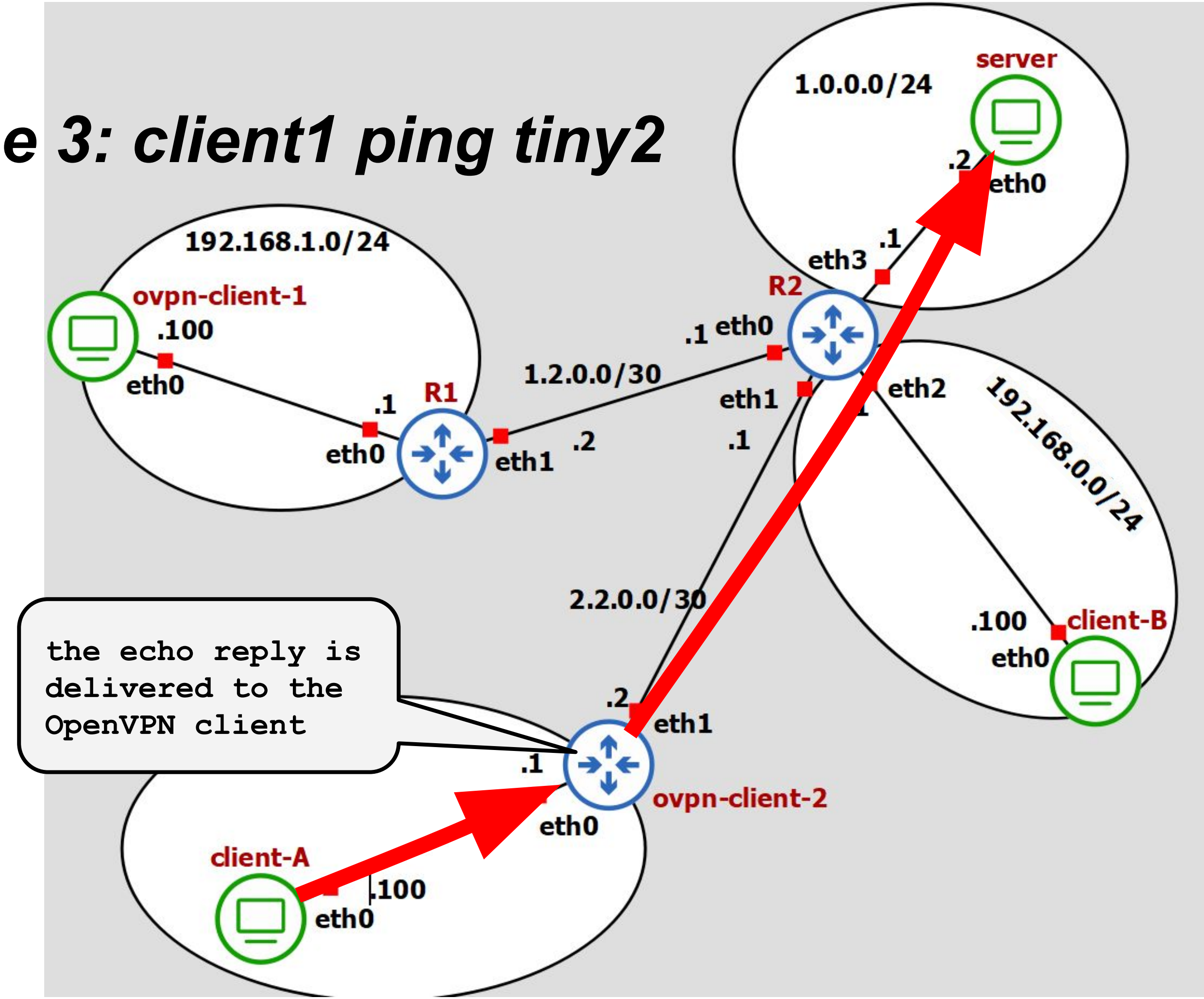


tiny2 replies to the source address in the packet

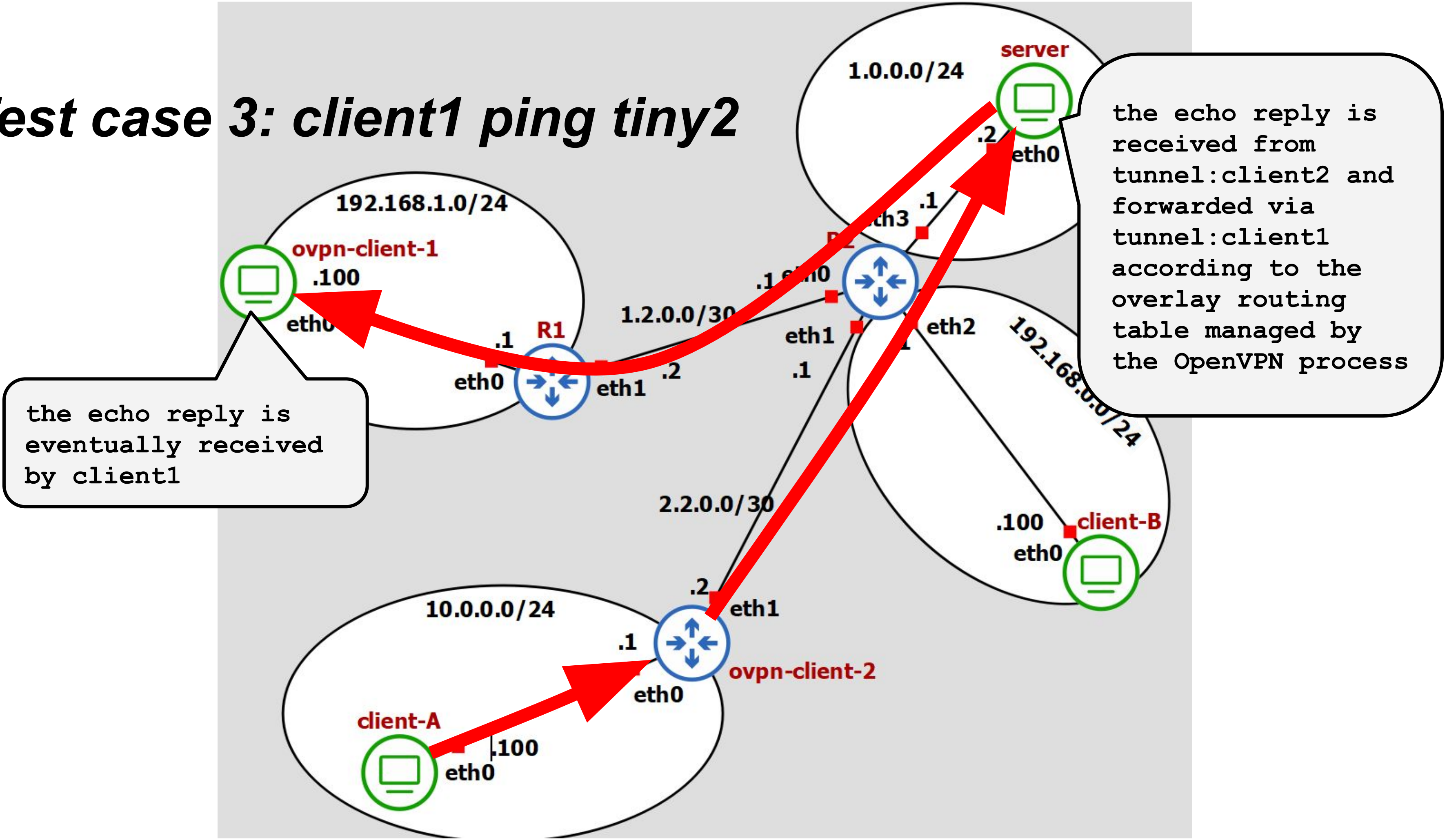
Test case 3: client1 ping tiny2



Test case 3: client1 ping tiny2



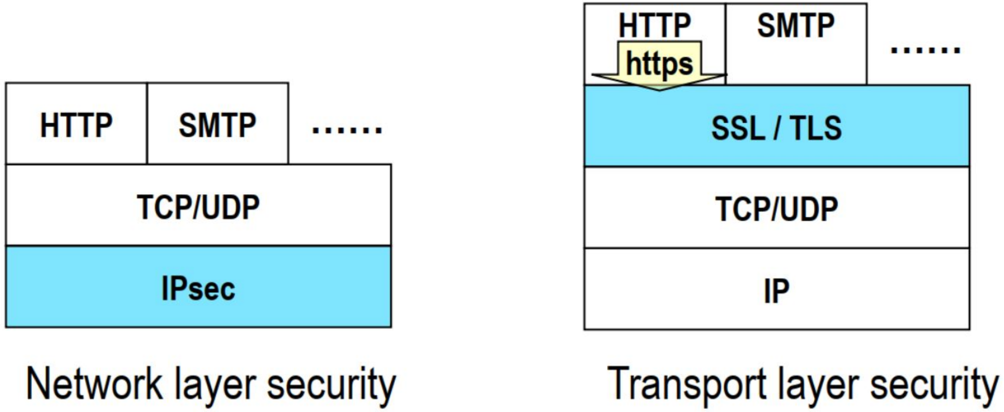
Test case 3: client1 ping tiny2



IPSec

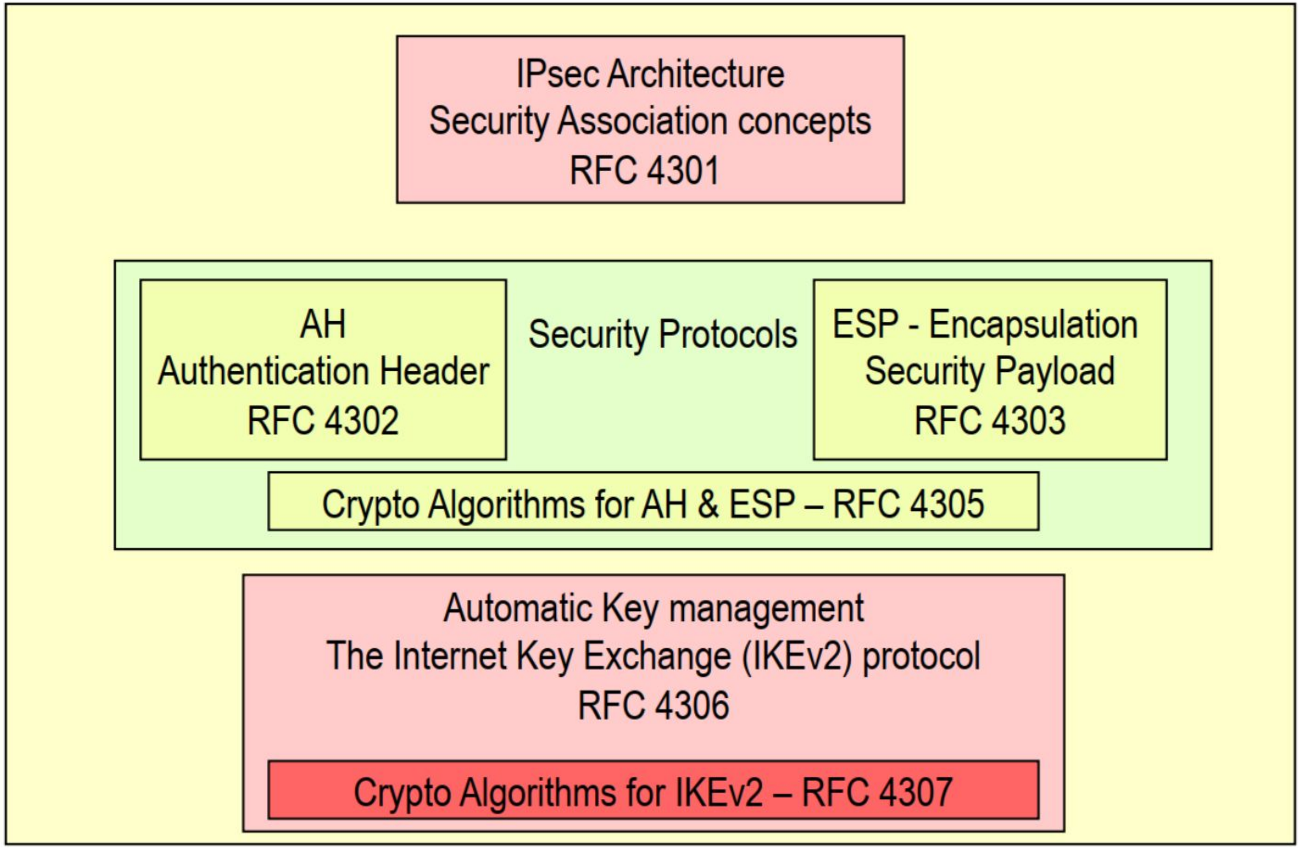
materiale tratto dalle slide del corso ***"Computer and Network Security"*** *di Prof. Giuseppe Bianchi*

IPsec: protocol stack



- IPSec operates at the IP level, at network layer (L3)
  - IPSec and plain-text IP packets can coexist
- Applications and devices are IPSec agnostic
  - IPsec protects every upper layer protocol
- Protection is per-host (IP address)

IPsec: standardization



Security Associations

- Fundamental concept for IPSec
- May include
  - Host to Host
  - Host to Intermediate Router (security gateway)
  - Security Gateway to Security Gateway
- Defines the crypto material for authenticating IP packets
- SAs are monodirectional: one for each transmission direction
  - The **Security Parameter Index (SPI)** is the unique identifier for an SA
    - 32 bits, along with the IP address
  - The **Security Association Databases (SAD)** host the crypto material (cipher keys, certificates, crypto algorithms etc.) for each SA
    - They are referenced by the SPI

Security Associations and Key Management

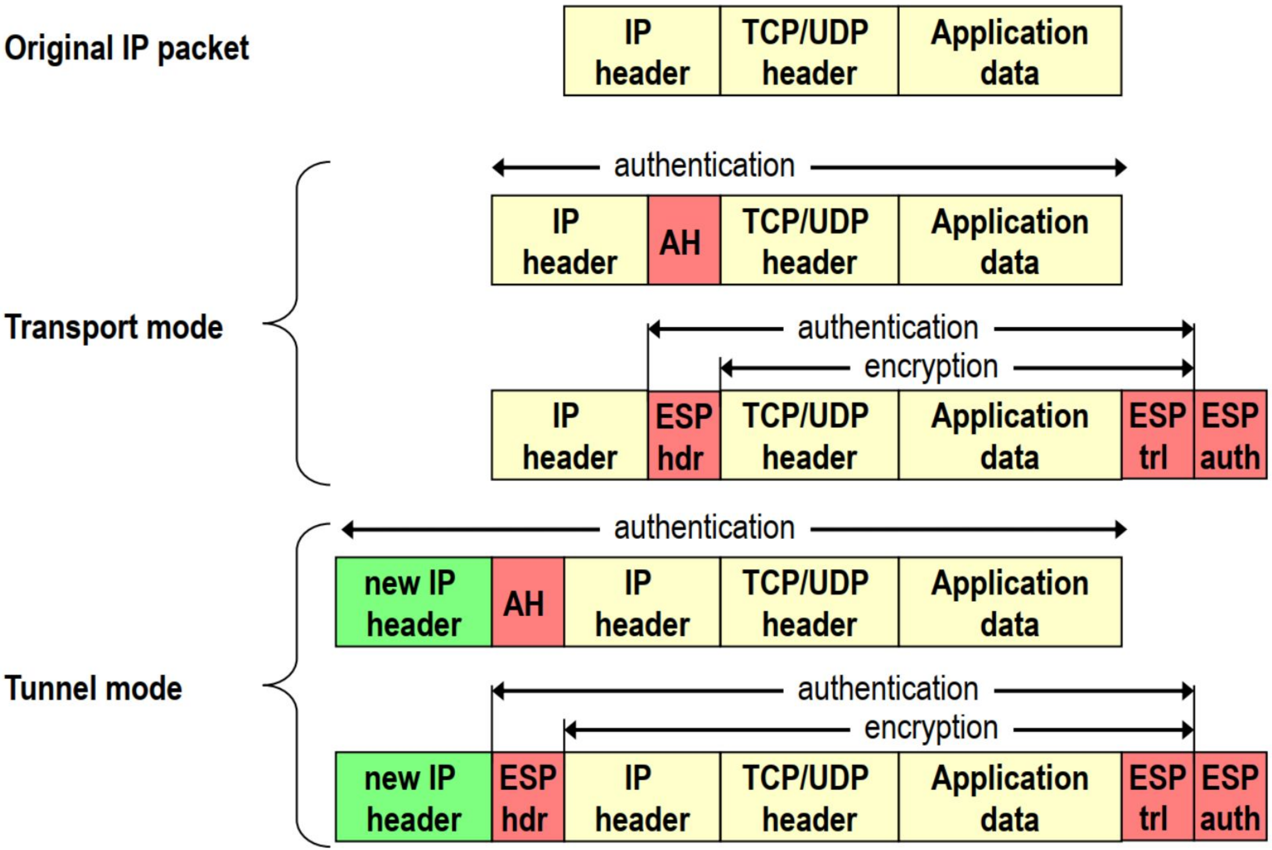
Key management could be

- **MANUAL:**
  - Manually configure SAs and their crypto material
    - static, for symmetric ciphers
  - Generally used in small scale VPN scenarios
    - few Security Gateways (e.g. one for each site)
    - Full Mesh between gateways
- **AUTOMATIC:**
  - SA management is handled by specific protocols (**IKEv2**)
    - In the past, some cooperating protocols (IKE, ISAKAMP, etc)
      - Authentication services
    - Creation of SA is on-demand
      - More later..

IPSec Protocol Suite

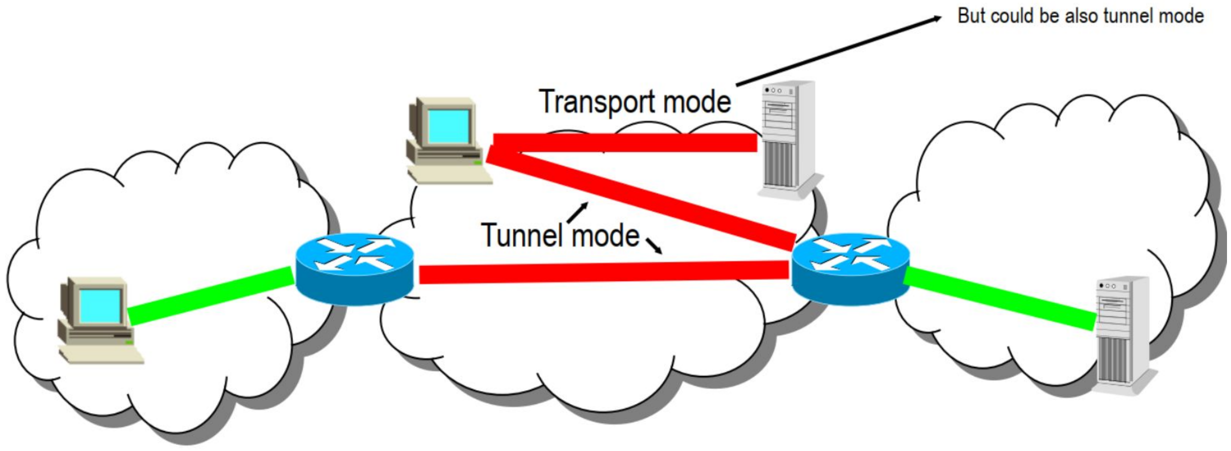
- It is often said that IPSec is a suite of protocols, rather than a single security protocol
- **Authentication Header (AH)**
  - Provides authentication and integrity of the entire IP packet (including the IP header), and nothing else (no encryption)
- **Encapsulated Security Payload (ESP)**
  - Provides authentication and/or encryption of the IP packet payload
    - no header integrity
    - not a problem when used in tunnel mode
- ESP is in the majority of cases the only protocol that is needed
  - in fact AH has been downgraded from MUST to MAY be supported in IPSec implementations
  - In any case, AH and ESP can be combined if necessary (rarely)

Transport vs Tunnel mode



Transport vs Tunnel mode

- Transport mode is used only for end-to-end connections
- Gateways use transport mode only for connections that originate and terminate at the gateway
  - In practice, they cannot route packets from other devices into the secure connection



IPSec Security Policies

- Security policies can be defined with IPSec
  - For example, if the protocol is udp, do not use IPSec
- They are contained in the **Security Policy Database (SPD)**.
- A security policy is a *"match-action"* rule that specifies what to do with unprotected packets
- **match** could be:
  - IP source/destination address [/netmask]
  - IP protocol (tcp, udp, etc…)
  - L4 source/destination ports
- **actions** could be:
  - BYPASS (do nothing)
  - DISCARD (drop)
  - PROTECT (apply IPSec AH/ESP)

IPSec Operations

- **Packet in output:**
  - Checks whether there is a Policy match in the SPD
  - If an "IPSec" policy is found, the packet is passed to the SAD
  - If a static SA is found, the packet is processed according to the security parameters of the SA
    - If not, an SA negotiation procedure with a key exchange protocol (e.g., IKE) can be activated

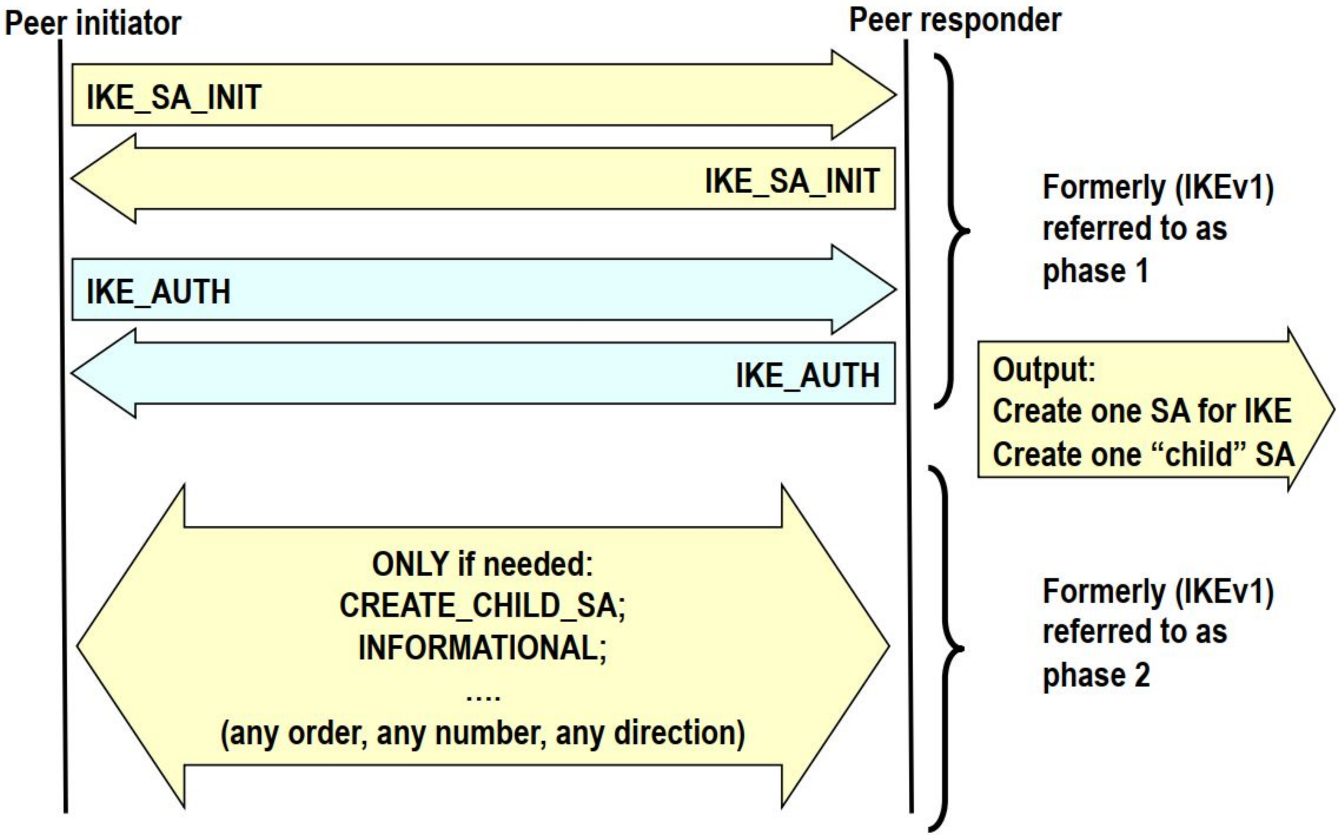
- **Packet in input:**
  - SPI, source and destination IPs are used to find the SA
  - The packet is processed (decrypted and authenticated)
  - SPD is queried for the presence of security policies

IKE(v2)

- Large-scale manual configuration of IPSec can be complex
  - May be fine for small VPNs
  - In any case, less secure as an SA could have infinite life (no rekeying)
- You have to maintain state between the two ends of an IPSec connection:
  - Security services (AH/ESP)
  - Crypto algorithms to be used
  - Crypto keys

- The *Internet Exchange Protocol (IKE)* establishes and dynamically maintain the SA
  - IKEv2 replaces previously used protocols (IKE, ISAKMP, DOI).

IKE protocol phases



IKE SA and Child SA

- **IKE SA:**
  - Establishes the SA for IKE protocol control messages
    - NO AH/ESP

- **IKE Child SA:**
  - Dynamically establishes SAs for data streams
    - Using AH/ESP
  - For each IKE SA, many Child SAs can be defined
    - For example, to define different tunnels for different subnets

Lab 9: IPSec site-to-site VPNs with strongswan

IPSec and Linux

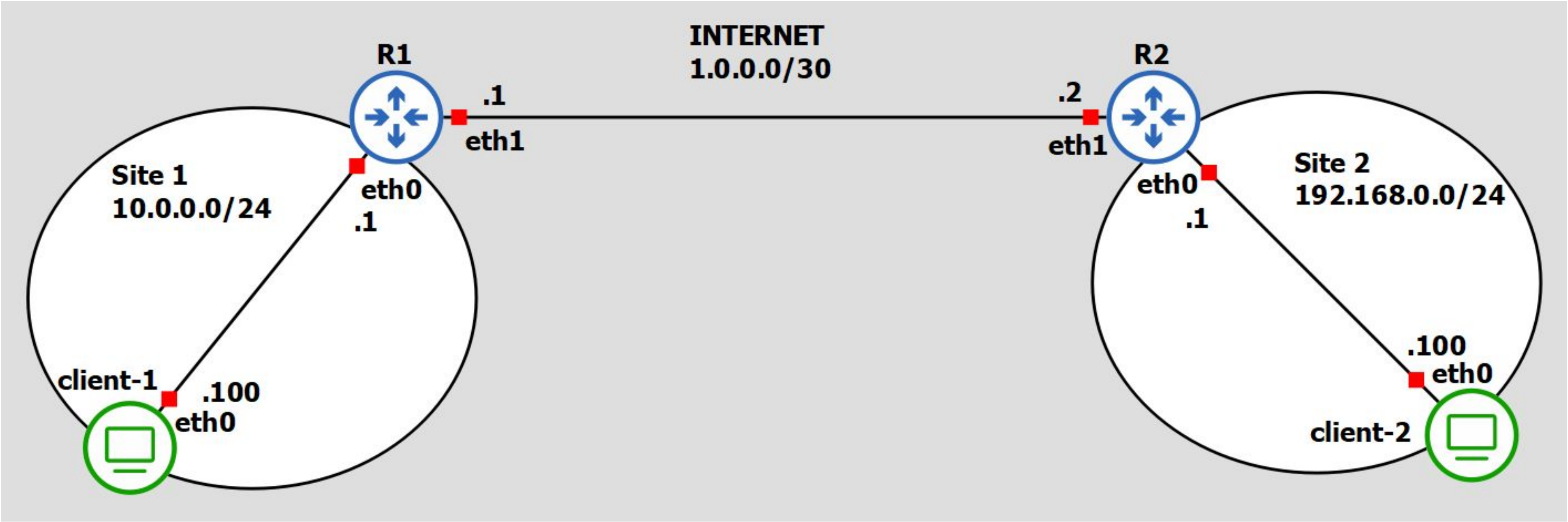
- IPSec is natively supported in the Linux kernel as early as version 2.6.1 (we are now at 6+)
- IPSec data protection (AH and ESP) is implemented in the kernel
- SA, SPD and SAD are configurable through tools in userspace…
  - SAD and SPD (setkey)
  - SA negotiation
    - there are many tools (racoon (IKEv1 only), racoon2, strongSwan, Libreswan)
      - we’ll use *strongswan* (IKEv2)
- … or with iproute2 (not that manageable…)
  - `ip xfrm <state, policy, monitor>`

IPSec Conf

- We will use strongswan's configuration files to automatically generate the SA, SAD, and SPD
- Strongswan allows us to use one file to configure all the security parameters of an IPSec connection
- We will use PSK as the authentication mode
  - so we will create a shared secret in the Secure Gateways of the two sites.
    - Obviously, other methods are possible, e.g. RSA
- **Configuration steps:**
  - Configuring IP addresses
  - Creation of strongswan configuration file
  - Start strongswan service (`service ipsec start`)
  - Uploading credentials (shared secret) to the SAD
    - `swanctl --load-creds`
  - Load policies in the SPD
    - `swanctl --load-conns`
  - Manual activation of SAs negotiation
    - `swanctl --initiate --child <child-name>`

(Install: `apt install strongswan strongswan-pki libcharon-extra-plugins libcharon-extauth-plugins libstrongswan-extra-plugins`)

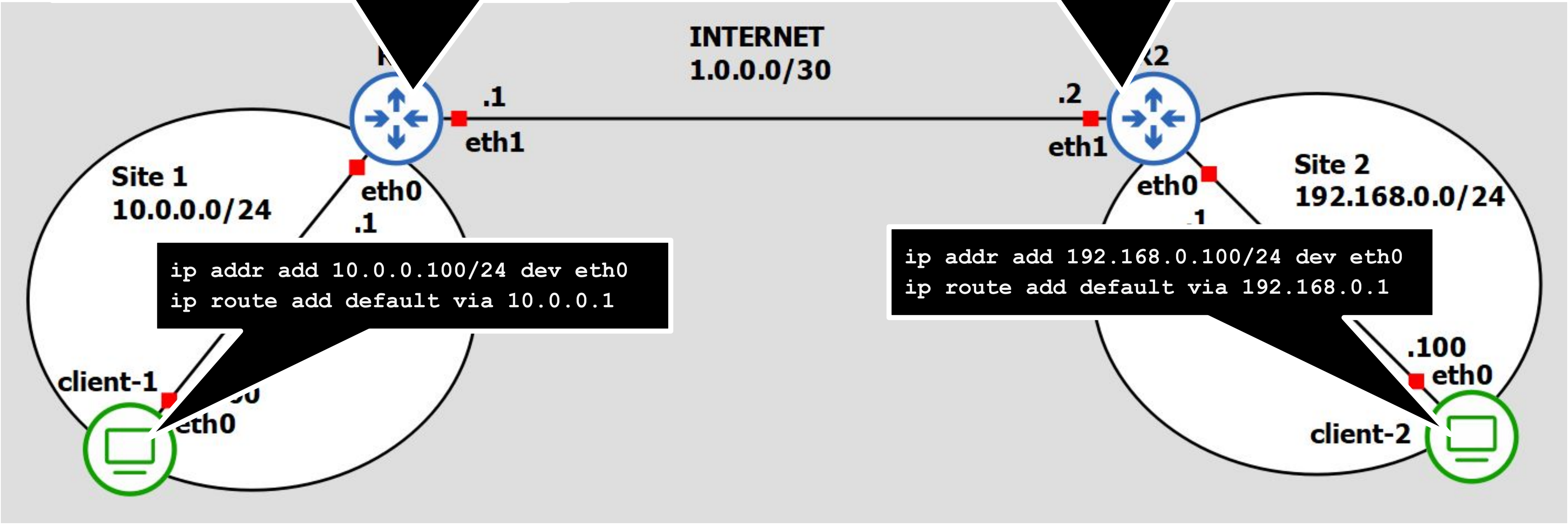
Network topology



Network Configuration

```
ip addr add 10.0.0.1/24 dev eth0
ip addr add 1.0.0.1/30 dev eth1
sysctl net.ipv4.ip_forward=1
```

```
ip addr add 192.168.0.1/24 dev eth0
ip addr add 1.0.0.2/30 dev eth1
sysctl net.ipv4.ip_forward=1
```




```
ip addr add 10.0.0.100/24 dev eth0
ip route add default via 10.0.0.1
```

```
ip addr add 192.168.0.100/24 dev eth0
ip route add default via 192.168.0.1
```

Strongswan conf (site 1)

```
# /etc/swanctl/conf.d/ipsec.conf
connections {
   gw-gw {
      local_addrs  = 1.0.0.1
      remote_addrs = 1.0.0.2

      local {
         auth = psk
         id = site1
      }
      remote {
         auth = psk
         id = site2
      }
      children {...
```

```
      children {
          net-net {
             local_ts  = 10.0.0.0/24
             remote_ts = 192.168.0.0/24

             rekey_time = 5400
             rekey_bytes = 500000000
             rekey_packets = 1000000
             esp_proposals = aes128gcm128-x25519
          }
       }
       version = 2
       mobike = no
       reauth_time = 10800
       proposals = aes128-sha256-x25519
   }
}
```

```
secrets {
   ike-1 {
      secret = 0x45a30759df97dc26a15b88ff
   }
}
```

Strongswan conf (site 2)

```
# /etc/swanctl/conf.d/ipsec.conf
connections {
   gw-gw {
      local_addrs  = 1.0.0.2
      remote_addrs = 1.0.0.1

      local {
         auth = psk
         id = site2
      }
      remote {
         auth = psk
         id = site1
      }
      children {...
```

```
      children {
          net-net {
             remote_ts  = 10.0.0.0/24
             local_ts = 192.168.0.0/24

             rekey_time = 5400
             rekey_bytes = 500000000
             rekey_packets = 1000000
             esp_proposals = aes128gcm128-x25519
          }
       }
       version = 2
       mobike = no
       reauth_time = 10800
       proposals = aes128-sha256-x25519
    }
}
```

```
secrets {
   ike-1 {
      secret = 0x45a30759df97dc26a15b88ff
   }
}
```