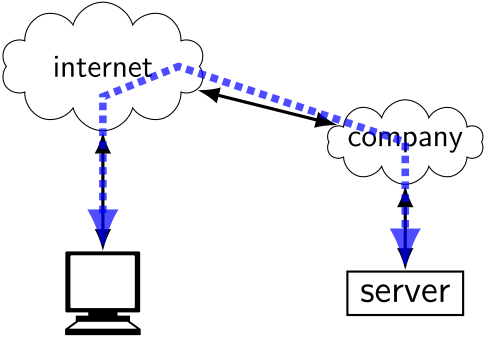
changelog

26 Nov 2024: fixup extra and misplaced lines on SOCKS operation slides

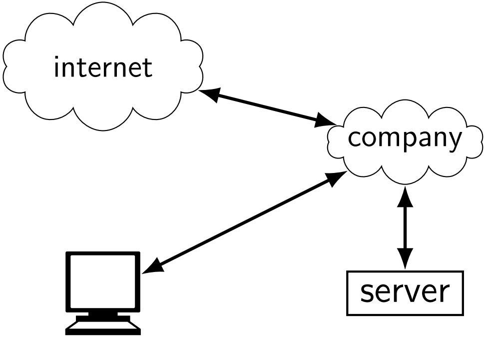
26 Nov 2024: fixup formatting on SSH slide

on a remote network

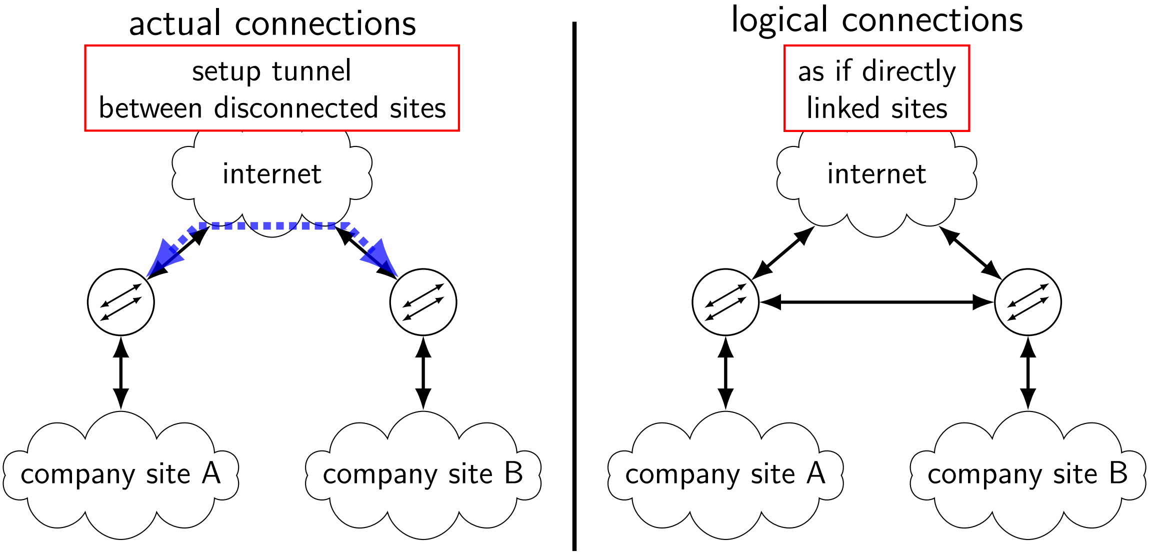
actual connections



logical connections

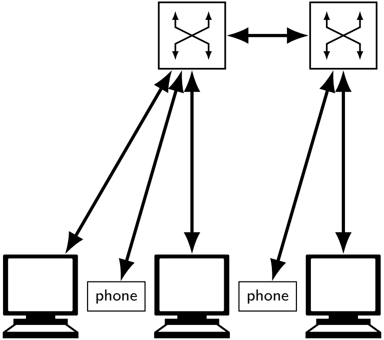


connecting two networks

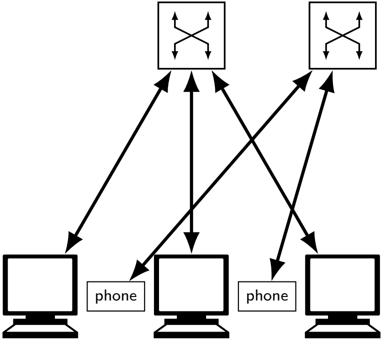


two networks in one

actual connections



logical connections



encapsulation: why? (1)

some possible scnearios (1/2):

add encryption/authentication to data in flight

more explicit way to decide what goes through firewall

be “on University/company network” from home

hide original location of Internet connection

make virtual machines running on different servers appear to be plugged into one switch

encapsulation: why? (2)

some possible scnearios (2/2):

evade overally restrictive firewall rules

make two datacenters connected via Internet appear to be one big network

'separate' networks for phones v. desktops without buying two sets of switches

aside: non-end-to-end encryption?

often encapsulation used to have encrypted link

nice, but really want to encrypt between end-hosts
- example: SSH, HTTPS

exercise: extra vulnerable points if relying on encrypted link idea?

encapsulation options [incomplete]

left in above	TCP/UDP/higher layers	IP	link-layer
above TCP/UDP	HTTP proxy, DNS over HTTP(S)	—	—
TCP/UDP	SOCKS, HTTP CONNECT, SSH conn forwarding, TLS	—	—
IP	OpenVPN, WireGuard	GRE, IPsec	MPLS
link-layer	OpenVPN, ...	?	VLAN, MPLS

encapsulation steps

1. getting the stuff to encapsulate

2. sending it encapsulated

encapsulation steps

1. getting the stuff to encapsulate
2. sending it encapsulated

encapsulating w/ app changes

application might have special code to handle connecting differently

+ might take advantage of extra information in encapsulation

example: many application's TLS support

example: web browser HTTP/SOCKS proxy support

encapsulating w/o app changes

generally: easier to do for lower layers

link-layer in something
- "virtual" (probably Ethernet) device
- sends/receives from 'tunnel'

IP in something
- "virtual" IP link
- destination routing table can go to

UDP/TCP in something
- replace socket API
- convince application to connect to different IP address?
- terminate UDP/TCP connection at 'wrong' machine

encapsulating w/o app changes

generally: easier to do for lower layers

link-layer in something
- “virtual” (probably Ethernet) device
- sends/receives from ‘tunnel’

IP in something
- “virtual” IP link
- destination routing table can go to

UDP/TCP in something
- replace socket API
- convince application to connect to different IP address?
- terminate UDP/TCP connection at ‘wrong’ machine

Linux tap devices

```
# create virtual ethernet device mydev
$ ip tuntap add dev mydev mode tap
# mark ethernet device as up
$ ip link set mydev up
$ dhclient mydev # or other commands to use/config device
```

(dhclient is a DHCP client)

—

```
int opentap(const char * name) {
    ... /* setup code, not shown*/
}
...
int fd = opentap("mydev");
...
write(fd, ethernetPacket, packetSize)
/* and (probably in separate threads) */
read(fd, buffer, SIZE); processEthernetPacket(buffer);
```

encapsulating w/o app changes

generally: easier to do for lower layers

link-layer in something
- "virtual" (probably Ethernet) device
- sends/receives from 'tunnel'

IP in something
- "virtual" IP link
- destination routing table can go to

UDP/TCP in something
- replace socket API
- convince application to connect to different IP address?
- terminate UDP/TCP connection at 'wrong' machine

Linux tun devices

same as ‘tap’ devices, but...

get IP packets, not ethernet packets

```
# create virtual ethernet device mydev
$ ip tuntap add dev mydev mode tun
# setup device to be routed to, example:
$ ip address add 10.0.0.2 dev mydev
$ ip route add 10.0.0.0/24 dev mydev
$ ip -6 address add 3fff:1234::1 dev mydev
$ ip -6 route add 3fff:1234::/32 dev mydev
```

tunneling program can then open device and read/write IP packets

full tunnel routing table

say 198.51.100.5 is running tunnel sever,
and 10.0.2.5 is gateway beyond tunnel,
and 203.0.113.54 is local gateway:

address	next hop	dev	priority
10.0.2.0/24	—	tunnel	normal
198.51.100.5/32	208.0.113.54	real	high
(default)	203.0.113.54	real	normal
(default)	10.0.2.5	tunnel	high

alternate idea

shown: creating special route for tunnel destination

alternate idea: tell OS to use correct interface/IP address

most OSes: which IP address is bound = which network interface to use

- but would need to discover correct IP address
- might be tricky if wireless + wifi connections, or wifi changes

split tunnel routing table

say 198.51.100.5 is running tunnel sever,
and 10.0.2.5 is gateway beyond tunnel,
and 10.0.0.0/16, 198.51.100.0/24 are tunneled networks
and 203.0.113.54 is local gateway:

address	next hop	dev	priority
10.0.2.0/24	—	tunnel	normal
10.0.0.0/16	10.0.2.5	tunnel	normal
198.51.100.0/24	10.0.2.5	tunnel	normal
198.51.100.5/32	208.0.113.54	real	high
(default)	203.0.113.54	real	normal

encapsulating w/o app changes

generally: easier to do for lower layers

link-layer in something
- "virtual" (probably Ethernet) device
- sends/receives from 'tunnel'

IP in something
- "virtual" IP link
- destination routing table can go to

UDP/TCP in something
- replace socket API
- convince application to connect to different IP address?
- terminate UDP/TCP connection at 'wrong' machine

‘transparent’ proxy via library

```
ProxyChains README
current version: 3.1
======================

This is open source software for GNU/Linux systems.

proxychains - a tool that forces any TCP connection made by any given application
to follow through proxy like TOR or any other SOCKS4, SOCKS5 or HTTP(S) proxy.
Supported auth-types: "user/pass" for SOCKS4/5, "basic" for HTTP.
```

configures dynamic library loader to load its library

    LD_PRELOAD

with special versions of `connect`

encapsulating w/o app changes

generally: easier to do for lower layers

link-layer in something

- “virtual” (probably Ethernet) device
- sends/receives from ‘tunnel’

IP in something

- “virtual” IP link
- destination routing table can go to

UDP/TCP in something

- replace socket API
- convince application to connect to different IP address?
- terminate UDP/TCP connection at ‘wrong’ machine

socket-in-socket

example: SSH connection forwarding

```
ssh -L X:remotehost:remoteport hostname
```

example: `stunnel` for 'tunneling' TCP in TLS

run on port X, configure to connect to some remote host

configure program to connect to localhost port X

…instead of remote host

encapsulating w/o app changes

generally: easier to do for lower layers

link-layer in something

- “virtual” (probably Ethernet) device
- sends/receives from ‘tunnel’

IP in something

- “virtual” IP link
- destination routing table can go to

UDP/TCP in something

- replace socket API
- convince application to connect to different IP address?
- terminate UDP/TCP connection at ‘wrong’ machine

‘transparent’ proxy via IP interception

example use case: shared HTTP cache used automatically
- if company/ISP wants everyone to use cache for performance
- if company wants to audit all HTTP proxy

configure router/firewall to direct all HTTP TCP connections to proxy

either:
- configure proxy machine to accept connections on all IP addresses *or*
- have router/firewall do network address translation (other direction)

HTTP proxy `squid` on Linux, some firewall boxes support this

I don’t think this is a good idea…
- problematic with encryption, new HTTP features

encapsulation steps

1. getting the stuff to encapsulate
2. sending it encapsulated

encapsulation options [incomplete]

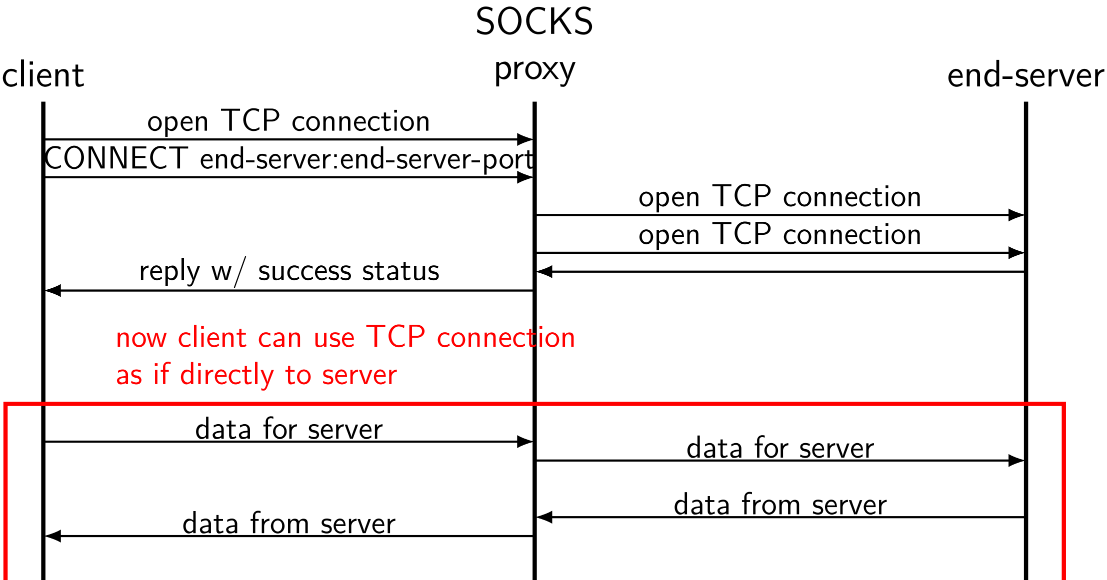
left in above	TCP/UDP/higher layers	IP	link-layer
above TCP/UDP	HTTP proxy, DNS over HTTP(S)	—	—
TCP/UDP	SOCKS, HTTP CONNECT, SSH conn forwarding, TLS	—	—
IP	OpenVPN, WireGuard	GRE, IPsec	MPLS
link-layer	OpenVPN, ...	?	VLAN, MPLS

SOCKS (RFC 1928)

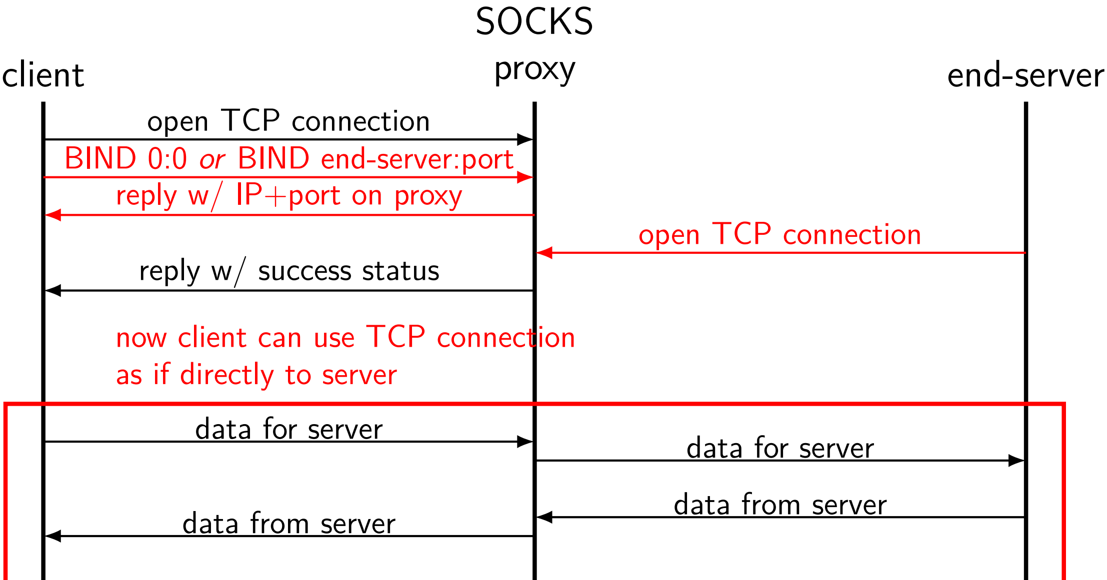
SOCKS motivation in RFC: firewall traversal

supports both TCP and UDP

SOCKS TCP operation (client)



SOCKS TCP operation (server)



SOCKS UDP operation

use TCP to get UDP port for proxy to use

send/receives UDP packets with “request header”:

```
+----+------+------+----------+----------+----------
|RSV | FRAG | ATYP | DST.ADDR | DST.PORT |   DATA
+----+------+------+----------+----------+----------
| 2  |  1   |  1   | Variable |    2     | Variable
+----+------+------+----------+----------+----------
```

FRAG = which fragment number

- added header means we might need to split UDP packets up
- 0 = not fragmented, most sig bit = last fragment

ATYP = address type (IPv4/IPv6/DNS name)

SOCKS as interface

relatively easy to add SOCKS support to (esp. TCP) program

but SOCKS doesn't support encryption, etc.

common trick: run SOCKS proxy program on localhost (127.0.0.1/::1)

that program sets up 'tunnel' with encryption, etc. to other machine

idea supported by OpenSSH, Tor

what if...

consider SOCKS TCP connection forwarding...

what happens to bandwidth, latency, resource usage, error-reporting if...

sending lots of data and proxy to remote link is slow?

remote host goes down in middle of connection?

SSH protocol overview

SSH transport layer protocol
- handles encryption + integrity + host authentication
- uses key exchange, with key share signed by host key
- rest of connection uses symmetric encryption + MACs

transport layer supports sending packets
- initial 1-type "message number" identifies type
- packets encrypted+MAC'd once that is negotiated

on top of transport layer:

SSH (user) authentication protocol
- handles passwords, etc.

SSH connection protocol
- handles terminal sessions, forwarded connections

SSH connection protocol

open channels identified by 32-bit integers

each channel has:

- type (`session` or `pty-req` or `tcpip-forward` or ...)
- "window size"
- maximum packet size

seperate packets for

- opening/closing channels
- adjusting "window size"
- sending data
- sending metadata (example: terminal window size)

SSH connection protocol

open channels identified by 32-bit integers

each channel has:

- type (`session` or `pty-req` or `tcpip-forward` or …)
- "window size"
- maximum packet size

seperate packets for

- opening/closing channels
- adjusting "window size"
- sending data
- sending metadata (example: terminal window size)

SSH connection protocol

open channels identified by 32-bit integers

each channel has:

- type (`session` or `pty-req` or `tcpip-forward` or ...)
- "window size"
- maximum packet size

seperate packets for

- opening/closing channels
- adjusting "window size"
- sending data
- sending metadata (example: terminal window size)

SSH window sizes

SSH connection protocol considers *window size* to be amount of data that can be sent

not same as TCP idea since no acknowledgments

decrements by X when X bytes sent

increases by Y on window adjust message with Y

SSH window sizes?

SSH has way of managing channel window sizes

how should SSH server do that?

(OpenSSH: 2MB max/init window size, not adjusted immediately)

Tor

Tor — “onion routing”

suppose connecting from A to B and A to C

goal: connection is anonymous

method: proxy through several ‘onion routers’

attacker should only know:

A is sending to someone via Tor
B is receiving from someone via Tor
C is receiving from someone via Tor

not be able to tie A and B or A and C or B and C together otherwise

Tor threat model

(from Digledine, Mathewson, Syverson, “Tor: A Second-Generation Onion Router”)

an advserary “who can observe some fraction of network traffic; who can generate, modify, delete, or delay traffic; who can operate onion routers of their own; and who can compromise some fraction of onion routers”

Tor circuit idea (1)

$E_X(Y)$ = Y encrypted to X

to create 'circuit': A $\leftrightarrow$ OR1 $\leftrightarrow$ OR2 $\leftrightarrow$ B

A = probably browser, B = probably webserver

OR = onion router

can choose different number of ORs if desired

A sends OR1 via TLS:
"please setup circuit to OR2: " + E_{OR2}("please connect to B")

OR1 sends A's encrypted data to OR2 with OR1's circuit ID

OR2 sends back responses via OR1 + OR1's circuit iD

OR1 uses circuit ID to send back to A

Tor circuit idea (2)

A $\leftrightarrow$ OR1 $\leftrightarrow$ OR2 $\leftrightarrow$ B

- A = probably browser, B = probably webserver

OR1 doesn't know who A is sending to

OR2 doesn't know who is sending to B

fine if OR1, OR2 independently operated

- in practice: probably add additional OR to circuit
- otherwise, require large portion of ORs to be independent

Tor circuit

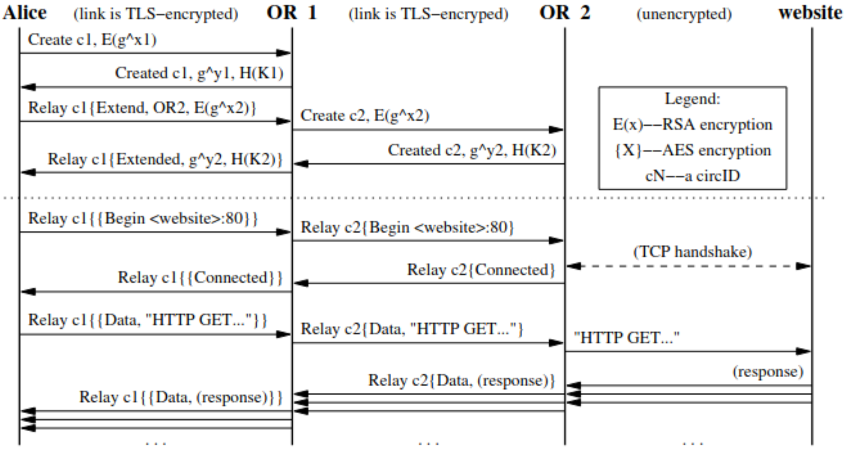


Figure 1: Alice builds a two-hop circuit and begins fetching a web page.

traffic analysis problem

problem 1: if I see A send 1000 bytes, then receive 1749 bytes, and…

at about the same time I see B receive 1000 bytes, then send 1749 bytes

…would be a big tell

worse: B or OR 1 or OR 3 can deliberately generate patterns of traffic to help ID A

mitigations for traffic analysis?

general idea: add data or delay to make everything 'the same'

add padding to traffic sent on 'circuit'

- 512-byte cells only (can't see exact sizes in bytes)
- additional padding cells added, too

"cover traffic" sent periodically between A and OR1

- 1.5 s to 9.5 s in each direction if no traffic
- idea: hard for attacker to tell when user active

but real-time nature limits possible mitigations

- similar idea for email avoids with random delays
- ...but can't really browse the web that way

application-layer tells

browser reveals a lot of information:

- browser, OS version
- screen size
- fonts available
- timezone
- ...

problematic for anonymity

- helps B limit possible other ends very seriously

Tor browser (modified Firefox, essentially) mitigation:
limited set of screen sizes, OS versions, fonts, etc. allowed

other Tor browser paranoia

scripts disabled by default

- seriously limits ‘ordinary’ browser security vulnerabilities

cookies, caches cleared when browser closed

HTTPS-only by default

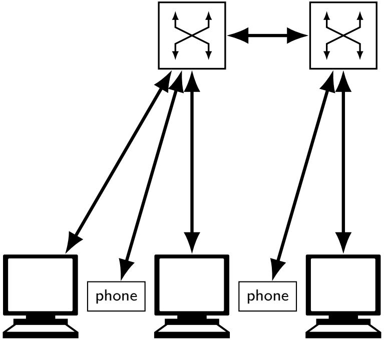
- really dangerous otherwise since we don’t trust last onion router

encapsulation options [incomplete]

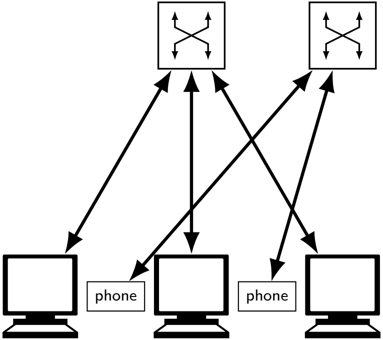
left in above	TCP/UDP/higher layers	IP	link-layer
above TCP/UDP	HTTP proxy, DNS over HTTP(S)	—	—
TCP/UDP	SOCKS, HTTP CONNECT, SSH conn forwarding, TLS	—	—
IP	OpenVPN, WireGuard	GRE, IPsec	MPLS
link-layer	OpenVPN, ...	?	VLAN, MPLS

two networks in one

actual connections



logical connections



GRE packet format

(IP or UDP header (for tunnel))

<table>
<tr><td>C</td><td>0</td><td>K</td><td>S</td><td>0</td><td>vers
0</td><td>protocol type
(EtherType)</td></tr>
<tr><td colspan="6">checksum (if C set)</td><td>0 (if C set)</td></tr>
<tr><td colspan="7">key (if K set)</td></tr>
<tr><td colspan="7">sequence number (if S set)</td></tr>
</table>

encapsulated header+data
(probably IPv4 or IPv6)

GRE packet format

(IP or UDP header (for tunnel))

<table>
<tr><td>C</td><td>0</td><td>K</td><td>S</td><td>0</td><td>vers
0</td><td>protocol type
(EtherType)</td></tr>
<tr><td colspan="6">checksum (if C set)</td><td>0 (if C set)</td></tr>
<tr><td colspan="7">key (if K set)</td></tr>
<tr><td colspan="7">sequence number (if S set)</td></tr>
</table>

encapsulated header+data

checksum, 'key' ($\sim$ port), sequence number optional

GRE packet format

(IP or UDP header (for tunnel))

C	0	K	S	0	vers 0	protocol type (EtherType)
checksum (if C set)						0 (if C set)
key (if K set)						
sequence number (if S set)						

key to allow multiple connections
if over UDP, could use separate ports instead
(but usualy not over UDP)

encapsulatoin with encryption

GRE = sends packets as is, setup in advance

often want to add autoconfiguration + encryption + authentication

typically:

- TLS-handshake like *key exchange* protocol to setup conncetion
- add space for message authentication code, nonce
- encrypt data with symmetric keys

example protocols:

- IKE (setup/key exchange) + IPsec ESP (actual tunnel)
- OpenVPN (both)
- WireGuard (both)

TCP-in-TCP problems

sometimes run IP tunnel over TCP

exercise: what's wrong with GRE for this?

(IP or UDP header (for tunnel))

<table>
<tr><td>C</td><td>0</td><td>K</td><td>S</td><td>0</td><td>vers
0</td><td>protocol type
(EtherType)</td></tr>
<tr><td colspan="6">checksum (if C set)</td><td>0 (if C set)</td></tr>
<tr><td colspan="7">key (if K set)</td></tr>
<tr><td colspan="7">sequence number (if S set)</td></tr>
</table>

encapsulated header+data
(probably IPv4 or IPv6)

TCP-in-TCP problems

sometimes run IP tunnel over TCP

example: picky firewall rules, or non-UDP-supporting NAT

outer TCP connection adds extra buffering

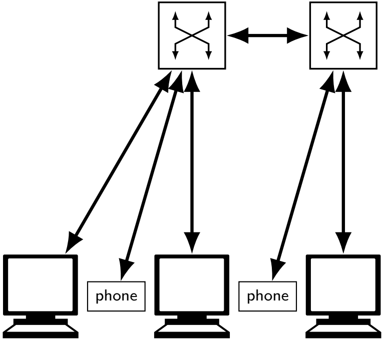
more than UDP, because will usually buffer instead of dropping

$\rightarrow$ very high round-trip time if not careful

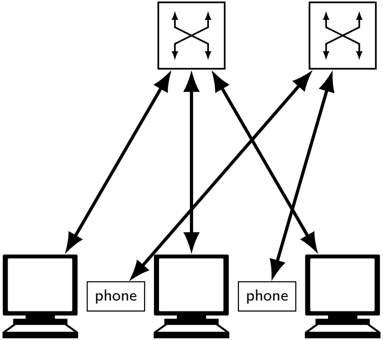
can result in very poor TCP performance

two networks in one

actual connections



logical connections



VLAN idea

multiple ('virtual') local networks over one network

links/ports either shared or assigned to just one network

most common implementation: Ethernt 802.1q:

on shared links, frames tagged with their 'VLAN ID'

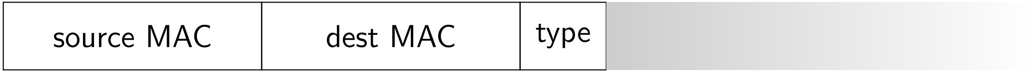
special case: untagged frames part of VLAN ID 0x0

tags added/removed when going to unshared links

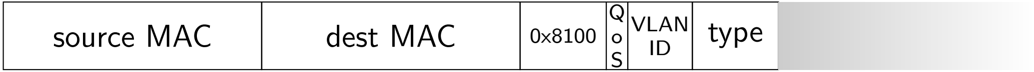
and broadcast frames filtered out if VLAN ID doesn't match

Ethernet encapsulation

unencapsulated:



encapsulated:



encapsulation typically added/removed by switches

- sysadmin configures specific ports to be on a VLAN
- another common case: virtual machine software

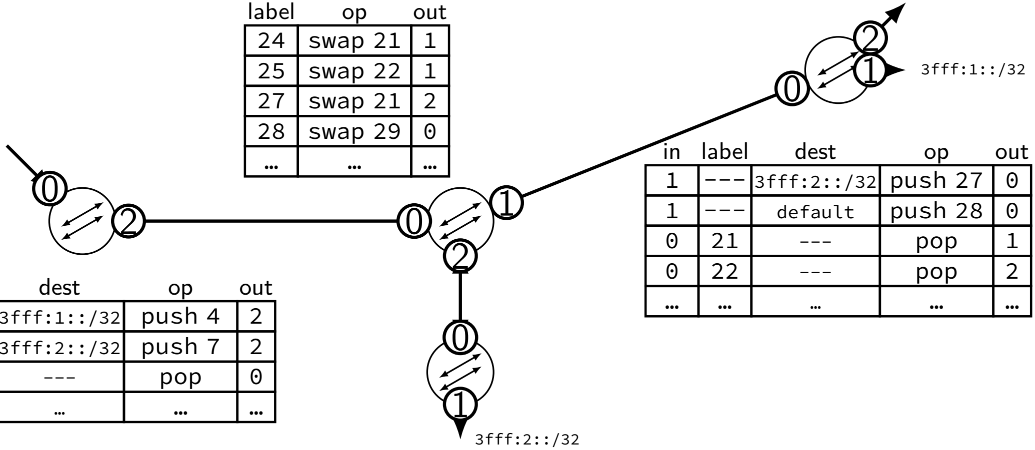
network IDs ('VLAN identifiers') configured by sysadmins

- special case: 0x0 = default (untagged), 0xFFF = reserved

usually increase in supported frame size to accomodate tag

multiprotocol label switching (MPLS)

MPLS: combines encapsulation and routing tables



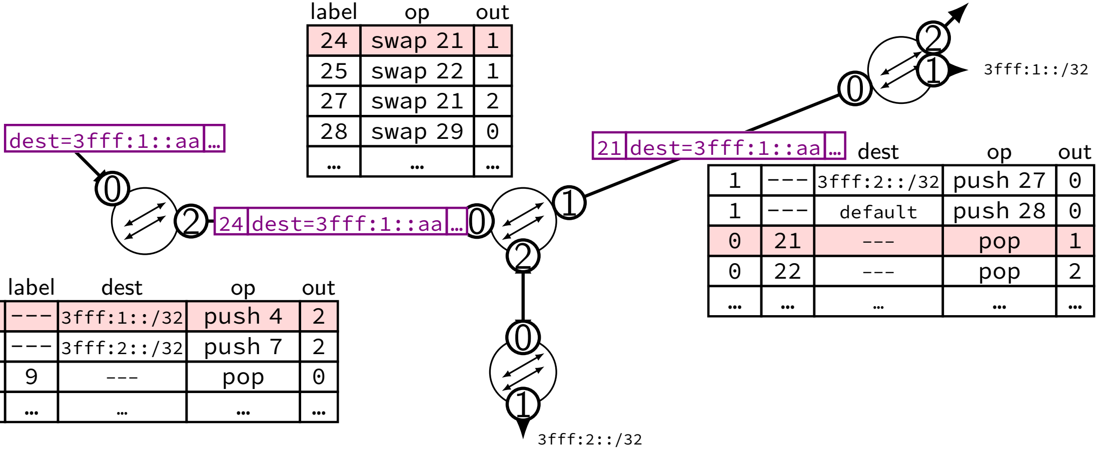
label	op	out
24	swap 21	1
25	swap 22	1
27	swap 21	2
28	swap 29	0
...	...	...

in	label	dest	op	out
1	---	3fff:2::/32	push 27	0
1	---	default	push 28	0
0	21	---	pop	1
0	22	---	pop	2
...	...	...	...	...

in	label	dest	op	out
0	---	3fff:1::/32	push 4	2
0	---	3fff:2::/32	push 7	2
2	9	---	pop	0
...	...	...	...	...

multiprotocol label switching (MPLS)

MPLS: combines encapsulation and routing tables



label	op	out
24	swap 21	1
25	swap 22	1
27	swap 21	2
28	swap 29	0
…	…	…

in	label	dest	op	out
0	---	3fff:1::/32	push 4	2
0	---	3fff:2::/32	push 7	2
2	9	---	pop	0
…	…	…	…	…

		dest	op	out
1	---	3fff:2::/32	push 27	0
1	---	default	push 28	0
0	21	---	pop	1
0	22	---	pop	2
…	…	…	…	…

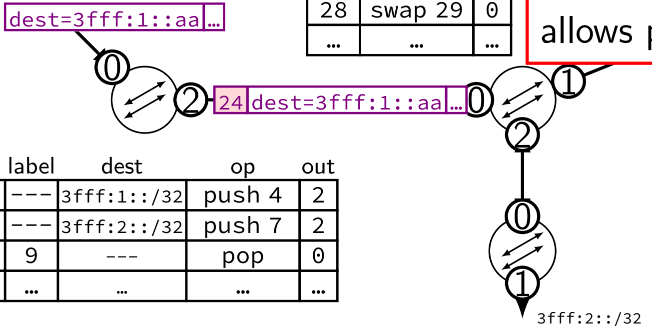
multiprotocol label switching (MPLS)

MPLS: combines encapsulation and routing tables

label	op	out
24	swap 21	1
25	swap 22	1
27	swap 21	2
28	swap 29	0
…	…	…

‘swap’ operation to translate between different label meanings

allows piecemail configuration

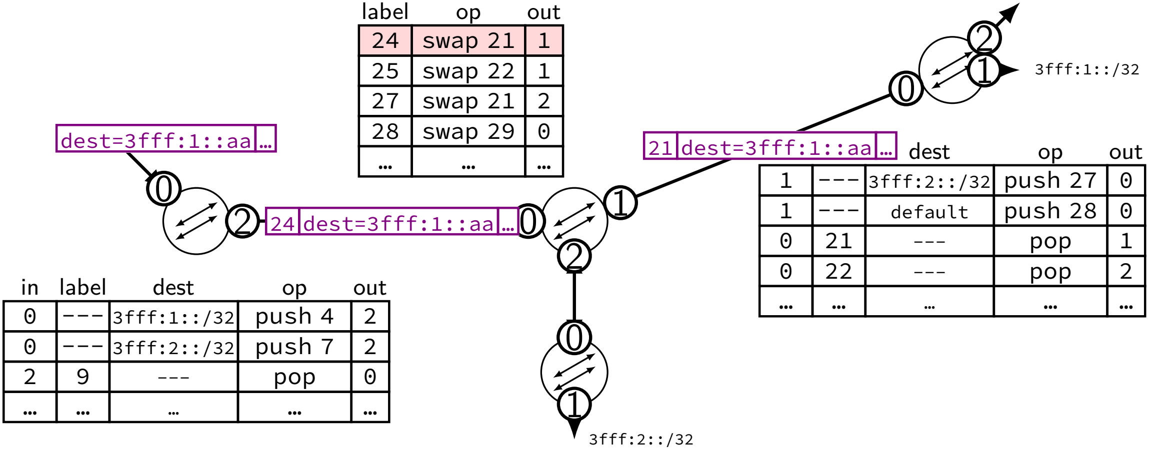


in	label	dest	op	out
0	---	3fff:1::/32	push 4	2
0	---	3fff:2::/32	push 7	2
2	9	---	pop	0
…	…	…	…	…

in	label	dest	op	out
[illegible]	[illegible]	[illegible]	[illegible]	0
1	---	default	push 28	0
0	21	---	pop	1
0	22	---	pop	2
…	…	…	…	…

multiprotocol label switching (MPLS)

MPLS: combines encapsulation and routing tables



label	op	out
24	swap 21	1
25	swap 22	1
27	swap 21	2
28	swap 29	0
…	…	…

in	label	dest	op	out
0	---	3fff:1::/32	push 4	2
0	---	3fff:2::/32	push 7	2
2	9	---	pop	0
…	…	…	…	…

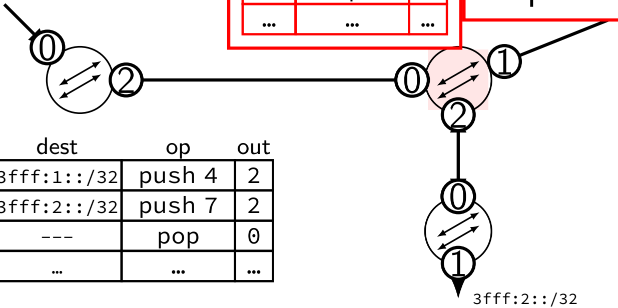
		dest	op	out
1	---	3fff:2::/32	push 27	0
1	---	default	push 28	0
0	21	---	pop	1
0	22	---	pop	2
…	…	…	…	…

multiprotocol label switching (MPLS)

MPLS: combines encapsulation and routing tables

label	op	out
24	swap 21	1
25	swap 22	1
27	swap 21	2
28	swap 29	0
…	…	…

routers in 'middle' of network
have *very simple* routing decisions
no prefix matching



in	label	dest	op	out
0	---	3fff:1::/32	push 4	2
0	---	3fff:2::/32	push 7	2
2	9	---	pop	0
…	…	…	…	…

1	---	3fff:2::/32	push 27	0
1	---	default	push 28	0
0	21	---	pop	1
0	22	---	pop	2
…	…	…	…	…

Label Distribution Protocol

share with neighbors list of:

(ultimate destination, desired label)

use entries to

allocate new local labels

setup appropriate swap entries

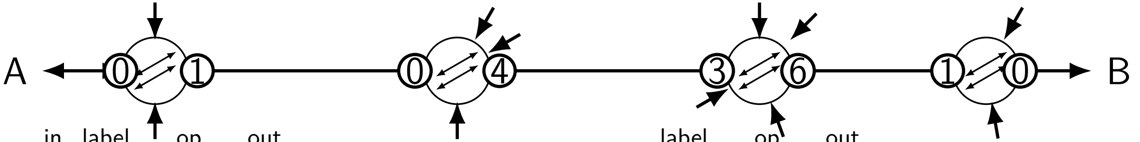
send other neighbors update about new labels

if using normal routing protocol to decide which neighbors routes to accept, just a funny way to implement routing tables

MPLS tunnel

logical view — ‘virtual wire’

actual network



in	label	op	out
0	---	push 32	1
---	1	pop	0
…	…	…	…

label	op	out
32	swap 37	4
26	swap 21	0
…	…	…

label	op	out
37	swap 20	6
39	swap 32	3
…	…	…

in	label	op	out
0	---	push 37	1
---	0	swap 29	6
…	…	…	…

MPLS tunnels for traffic engineering

if multiple paths from A to B often want to:

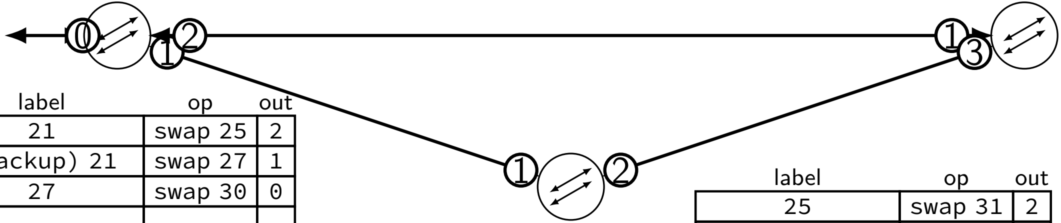
- balance between them to use available bandwidth
- prioritize important traffic on ‘better’ path
- ...

plain OSPF can’t really do any of this unless equal cost

MPLS gives mechanism to do this kind of balancing:

- setup labels along desired paths
- choosing new path (or failover) = changing ‘swap X’ to ‘swap Y’
- can configure backup paths in advance and turn them on later

rapid failover



label	op	out
21	swap 25	2
(backup) 21	swap 27	1
27	swap 30	0
...	...	...

label	op	out
25	swap 31	2
30	swap 27	1
30 (backup)	swap 28	3
...	...	...

label	op	out
27	swap 24	2
28	swap 27	1
...	...	...

RSVP-TE

RSVP-TE (RFC 3209): protocol for setting up MPLS tunnels

idea: routers figure out labels, etc.

end-user can (optionally) specify...

that tunnel go through specific routers

also setting up 'fast reroute' backup paths

bandwidth reservation

- routers give you error if bandwidth not gaurenteed

nested labels

label *stack* allows ...

nested tunnels

marking different types of packet

...

protocol-independence

only first/last routers care about actual protocol

easily allows for...

mix of Ethernet and IP tunnels

using routers that don't support IP (e.g. ATM)

...

actual label format

```
 0                   1                   2
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
|                Label                  | Exp |S|       TTL
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

                Label:  Label Value, 20 bits
                Exp:    Experimental Use, 3 bits
                S:      Bottom of Stack, 1 bit
                TTL:    Time to Live, 8 bits
```

TTL here is different than we've seen:

only processed when label popped

typically (re)set to mirror IP TTL if MPLS run over IP

encapsulation overheads

left in above	TCP/UDP/higher layers	IP	link-layer
above TCP/UDP	HTTP proxy, DNS over HTTP(S)	—	—
TCP/UDP	SOCKS, HTTP CONNECT, SSH conn forwarding, TLS	—	—
IP	OpenVPN, Wire-Guard	GRE, IPsec	MPLS
link-layer	OpenVPN, ...	?	VLAN, MPLS

which of these types of options

which encapuslation (1)

suppose I have two racks of servers in two different buildings

want them to be in the same subnetwork

- so they'll find each other with broadcast, multicast DNS

how should I do this if...

- the two buildings are connected via Ethernet also used for internet access?
- the two buildings are connected via an IP link leased from an ISP?

which encapuslation (2)

suppose I have a rack of servers in my building, but I'm migrating to the cloud

I've moved one server to a cloud provider...

I don't want to reconfigure the other servers that talk to it...

what would be some options to do this? what else do we need to know about the server?

what would be useful features for cloud provider to give us?

which encapsulation (3)

I need to have my machines which handle payment processing go be behind a firewall

but they're in the same rack as machines which should have a direct internet connection

how should I do this?

backup slides