Communication Networks

Prof. Laurent Vanbever

Communication Networks

Spring 2022

Laurent Vanbever

nsg.ee.ethz.ch

ETH Zürich (D-ITET)

April 25 2022

Last week on

Communication Networks

BGP suffers from many rampant problems

Problems

- Reachability
- Security
- Convergence
- Performance
- Anomalies
- Relevance

Problems

- Reachability
- Security
- Convergence
- Performance
- Anomalies
- Relevance

Many security considerations are absent from the BGP specification

ASes can advertise any prefixes
even if they don't own them!

ASes can arbitrarily modify route content
e.g., change the content of the AS-PATH

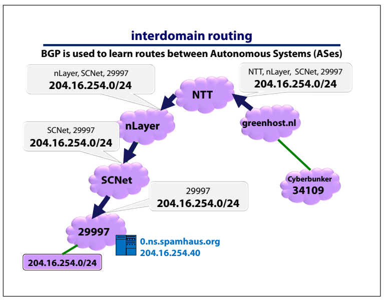
ASes can forward traffic along different paths
than the advertised one

BGP (lack of) security

#1 BGP does not validate the origin of advertisements

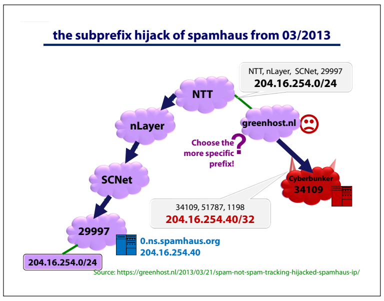
#2 BGP does not validate the content of advertisements

Let's look back at an example of BGP hijack



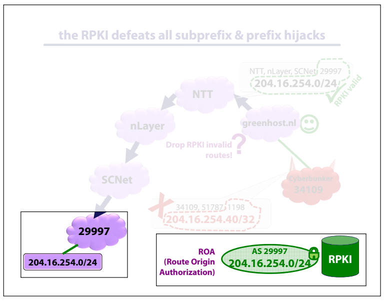
Source: Sharon Goldberg, "The Transition to BGP Security. Is the Juice Worth the Squeeze?"

Without RPKI, a more-specific attack by AS34109 successfully manages to attract the traffic



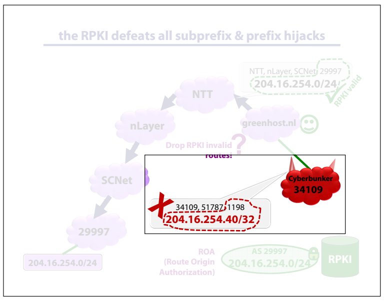
Source: Sharon Goldberg, "The Transition to BGP Security. Is the Juice Worth the Squeeze?"

Let's assume now that AS 29997 registers (204.16.254.0/24-32, 29997) as a new ROA



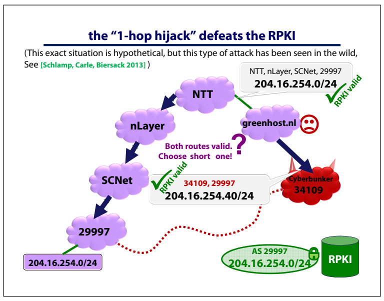
Source: Sharon Goldberg, "The Transition to BGP Security. Is the Juice Worth the Squeeze?"

This announcement is said to be *INVALID*



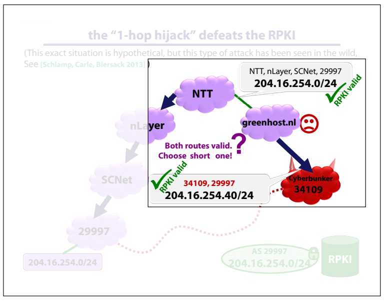
Source: Sharon Goldberg, "The Transition to BGP Security. Is the Juice Worth the Squeeze?"

Now what if AS34109 announce AS29997 as the origin?



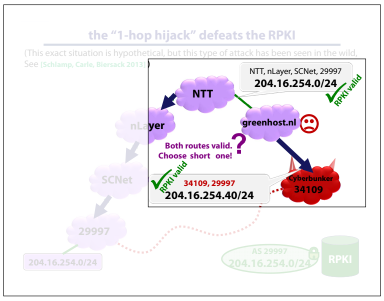
Source: Sharon Goldberg, "The Transition to BGP Security. Is the Juice Worth the Squeeze?"

Here greenhost.nl receives 2 valid RPKI routes: one via NTT and another one via 34109



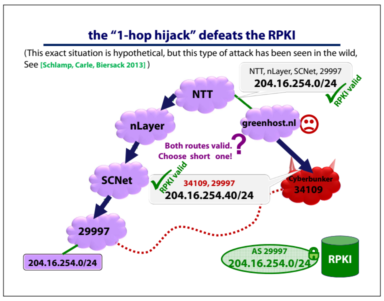
Source: Sharon Goldberg, "The Transition to BGP Security. Is the Juice Worth the Squeeze?"

As the route via 34109 has a shorter path, it is preferred... the attack works again!



Source: Sharon Goldberg, "The Transition to BGP Security. Is the Juice Worth the Squeeze?"

We see that RPKI does not protect against *all* attacks



Source: Sharon Goldberg, "The Transition to BGP Security. Is the Juice Worth the Squeeze?"

This week on
Communication Networks

Communication Networks

Part 2: Concepts

routing

reliable delivery

How do you ensure reliable transport on top of best-effort delivery?

In the Internet, **reliability** is ensured by the **end hosts**, not **by the network**

The Internet puts reliability in **L4**, just above the Network layer

goals	Keep the network simple, dumb make it relatively “easy” to build and operate a network
	Keep applications as network “unaware” as possible a developer should focus on its app, not on the network
design	Implement reliability in-between, in the networking stack relieve the burden from both the app and the network

---

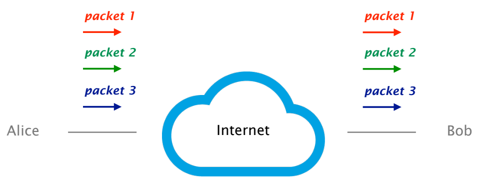
The Internet puts reliability in L4, just above the network layer

	layer	
	Application	
L4	Transport	reliable end-to-end delivery
L3	Network	global best-effort delivery
	Link	
	Physical	

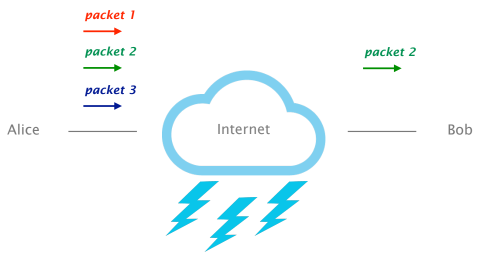
Recall that the network layer provides a best-effort service only

	layer	
	Application	
L4	Transport	reliable end-to-end delivery
L3	Network	global best-effort delivery
	Link	
	Physical	

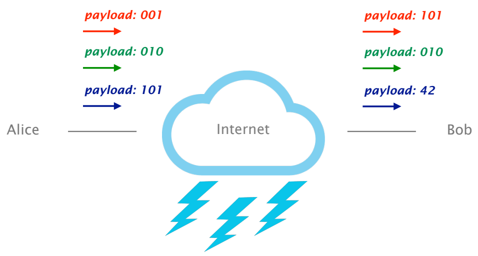
Let’s consider a simple communication between two end-points, **Alice and Bob**



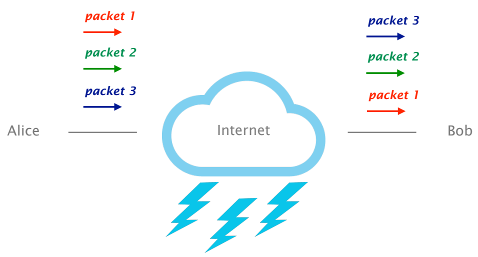
IP packets **can get lost or delayed**



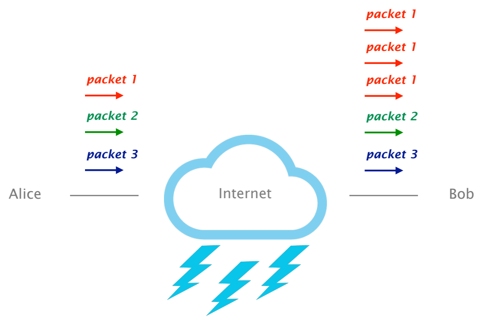
IP packets **can get corrupted**



IP packets can get reordered



IP packets can get duplicated



Reliable Transport

1. Correctness condition
   if-and-only if again
2. Design space
   timeliness *vs* efficiency *vs* …
3. Examples
   Go-Back-N & Selective Repeat

Reliable Transport

1. Correctness condition
   if-and-only if again

Design space
timeliness *vs* efficiency *vs* …

Examples
Go-Back-N & Selective Repeat

The four goals of reliable transfer

goals	
correctness	ensure data is delivered, in order, and untouched
timeliness	minimize time until data is transferred
efficiency	optimal use of bandwidth
fairness	play well with concurrent communications

goals	
correctness	ensure data is delivered, in order, and untouched

Routing had a clean sufficient and necessary correctness condition

sufficient and necessary condition

Theorem a global forwarding state is valid if and only if

- there are no dead ends
  no outgoing port defined in the table
- there are no loops
  packets going around the same set of nodes

We need the same kind of "if and only if" condition for a "correct" reliable transport design

A reliable transport design is correct if…

attempt #1 — packets are delivered to the receiver

Wrong — Consider that the network is partitioned

We cannot say a transport design is *incorrect*
if it doesn't work in a partitioned network…

A reliable transport design is correct if…

attempt #2 — packets are delivered to receiver if and only if
it was possible to deliver them

Wrong — If the network is only available one instant in time,
only an oracle would know when to send

We cannot say a transport design is *incorrect*
if it doesn't know the unknowable

A reliable transport design is correct if…

attempt #3 — It resends a packet if and only if
the previous packet was lost or corrupted

Wrong — Consider two cases

- packet made it to the receiver and
  all packets from receiver were dropped
- packet is dropped on the way and
  all packets from receiver were dropped

A reliable transport design is correct if…

attempt #3 — It resends a packet if and only if
the previous packet was lost or corrupted

Wrong — In both case, the sender has no feedback at all
Does it resend or not?

A reliable transport design is correct if…

attempt #3 — It resends a packet if and only if
the previous packet was lost or corrupted

Wrong

but better as it refers to what the design does (which it can control),
not whether it always succeeds (which it can't)

A reliable transport design is correct if…

attempt #4 — A packet is always resent if
the previous packet was lost or corrupted

A packet may be resent at other times

Correct!

A transport mechanism is correct
if and only if it resends all dropped or corrupted packets

Sufficient "if"	algorithm will always keep trying to deliver undelivered packets
Necessary "only if"	if it ever let a packet go undelivered without resending it, it isn't reliable
Note	it is ok to give up after a while but must announce it to the application

---

Reliable Transport

Correctness condition
if-and-only if again

2 Design space
timeliness *vs* efficiency *vs* …

Examples
Go-Back-N & Selective Repeat

Now, that we have a correctness condition
how do we achieve it and with what tradeoffs?

Design a *correct, timely, efficient* and *fair* transport mechanism
knowing that

packets can get lost → let's focus on these aspects first
corrupted
reordered
delayed
duplicated

Alice

```
for word in list:
    send_packet(word);
    set_timer();

    upon timer going off:
        if no ACK received:
            send_packet(word);
            reset_timer();
    upon ACK:
        pass;
```

Bob

```
receive_packet(p);
if check(p.payload) == p.checksum:
    send_ack();

    if word not delivered:
        deliver_word(word);
else:
    pass;
```

There is a clear tradeoff between timeliness and efficiency
in the selection of the timeout value

```
for word in list:
    send_packet(word);
    set_timer();

    upon timer going off:
        if no ACK received:
            send_packet(word);
            reset_timer();
    upon ACK:
        pass
```

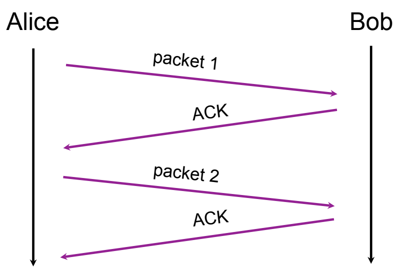
```
receive_packet(p);
if check(p.payload) == p.checksum:
    send_ack();

    if word not delivered:
        deliver_word(word);
else:
    pass;
```

Timeliness argues for small timers,
efficiency for large ones

timeliness	efficiency
small timers	large timers
risk	risk
unnecessary retransmissions	slow transmission

Even with short timers, the timeliness of our protocol is
extremely poor: one packet per Round-Trip Time (RTT)



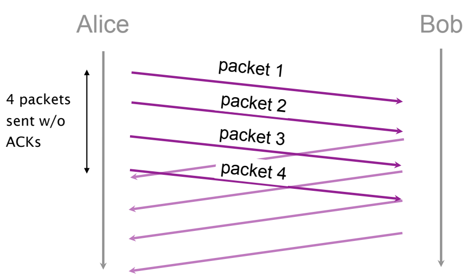
An obvious solution to improve timeliness is
to send multiple packets at the same time

approach

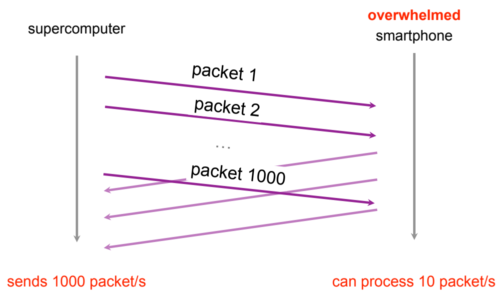
add sequence number inside each packet

add buffers to the sender and receiver

sender store packets sent & not acknowledged
receiver store out-of-sequence packets received



Sending multiple packets improves timeliness,
but it can also overwhelm the receiver



To solve this issue,
we need a mechanism for flow control

Using a sliding window is one way to do that

Sender keeps a list of the sequence # it can send
known as the *sending window*

Receiver also keeps a list of the acceptable sequence #
known as the *receiving window*

Sender and receiver negotiate the window size
sending window <= receiving window

Example with a window composed of 4 packets

unACK'ed packets | forbidden packets

0 1 2 3 [4 5 6 7] 8 9 10 11 …

ACKed packets | available packets

Window after sender receives ACK 4

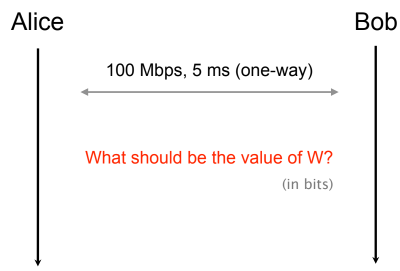
unACK'ed packets | forbidden packets

0 1 2 3 4 [5 6 7 8] 9 10 11 …

ACKed packets | available packets

Timeliness of the window protocol depends on
the size of the sending window

Assuming infinite buffers,
how big should the window be to maximize timeliness?



Timeliness matters,
but what about efficiency?

The efficiency of our protocol
essentially depends on two factors

receiver feedback

How much information does the sender get?

behavior upon losses

How does the sender detect and react to losses?

The efficiency of our protocol
essentially depends on two factors

receiver feedback

behavior upon losses

How much information does the sender get?

ACKing individual packets provides detailed feedback,
but triggers unnecessary retransmission upon losses

advantages	disadvantages
know fate of each packet	loss of an ACK packet requires a retransmission
simple window algorithm W single-packet algorithms	causes unnecessary retransmission
not sensitive to reordering	

Cumulative ACKs enables to recover from lost ACKs,
but provides coarse-grained information to the sender

approach	ACK the highest sequence number for which all the previous packets have been received
advantages	recover from lost ACKs
disadvantages	confused by reordering incomplete information about which packets have arrived causes unnecessary retransmission

Full Information Feedback prevents unnecessary
retransmission, but can induce a sizable overhead

approach	List all packets that have been received highest cumulative ACK, plus any additional packets
advantages	complete information resilient form of individual ACKs
disadvantages	overhead (hence lowering efficiency) *e.g.*, when large gaps between received packets

We see that Internet design is
all about balancing tradeoffs (again)

The efficiency of our protocol
essentially depends on two factors

receiver feedback

behavior upon losses

How does the sender detect and react to losses?

As of now, we detect loss by using timers.
That's only one way though

Losses can also be detected by relying on ACKs

With **individual ACKs**,
missing packets (gaps) are **implicit**

Assume packet 5 is lost
but no other

ACK stream

1
2
3
4
6 — sender can infer that 5 is missing
and resend 5 after *k* subsequent packets
7
...

With **full information**,
missing packets (gaps) are **explicit**

Assume packet 5 is lost
but no other

ACK stream

up to 1
up to 2
up to 3
up to 4
up to 4, plus 6 — sender learns that 5 is missing
retransmits after *k* packets
up to 4, plus 6—7
...

With **cumulative ACKs**,
missing packets are harder to know

Assume packet 5 is lost
but no other

ACK stream

1
2
3
4
4 sent when 6 arrives
4 sent when 7 arrives
...

Duplicated ACKs are a sign of isolated losses.
Dealing with them is trickier though.

situation

Lack of ACK progress means that 5 hasn't made it

Stream of ACKs means that (some) packets are delivered

Sender could trigger resend
upon receiving *k* duplicates ACKs

but *what* do you resend?
only 5 or 5 and everything after?

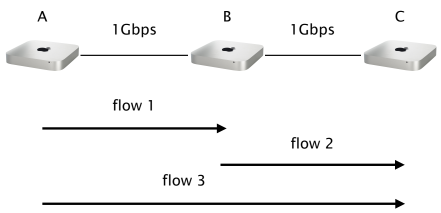
What about fairness?

Design a *correct*, *timely*, *efficient* and *fair* transport mechanism
knowing that

packets can get
- lost
- corrupted
- reordered
- delayed
- duplicated

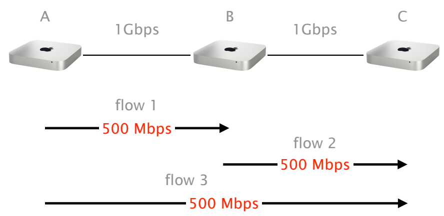
When *n* **entities** are using our transport mechanism,
we want a **fair allocation of the available bandwidth**

Consider this simple network
in which **three hosts are sharing two links**



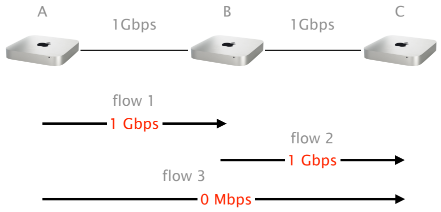
What is a fair allocation for the 3 flows?

An equal allocation is certainly "**fair**",
but what about the **efficiency of the network**?



Total traffic is 1.5 Gbps

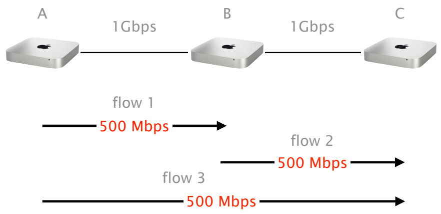
Fairness and efficiency don't always play along, here an unfair allocation ends up *more efficient*



Total traffic is 2 Gbps!

What is fair anyway?

Equal-per-flow isn't really fair as (A,C) crosses two links: it uses *more* resources



Total traffic is 1.5 Gbps

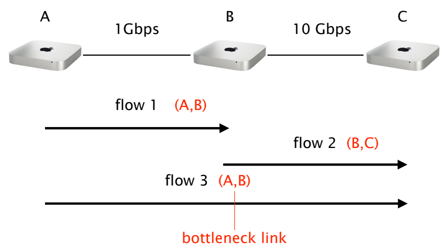
With equal-per-flow, A ends up with 1 Gbps because it sends 2 flows, while B ends up with 500 Mbps

Is it fair?

Seeking an exact notion of fairness is not productive. What matters is to avoid starvation.

equal-per-flow is good enough for this

Simply dividing the available bandwidth doesn't work in practice since flows can see different bottleneck



Intuitively, we want to give users with "small" demands what they want, and evenly distribute the rest

Max-min fair allocation is such that

the lowest demand is maximized

after the lowest demand has been satisfied,
the second lowest demand is maximized

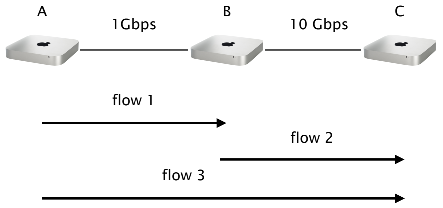
after the second lowest demand has been satisfied,
the third lowest demand is maximized

and so on...

Max-min fair allocation can easily be computed

step 1	Start with all flows at rate 0
step 2	Increase the flows until there is a new bottleneck in the network
step 3	Hold the fixed rate of the flows that are bottlenecked
step 4	Go to step 2 for the remaining flows
	Done!

Let’s try on this network



What’s the max-min fair allocation?

Max-min fair allocation can be approximated
by slowly increasing *W* until a loss is detected

Intuition	Progressively increase the sending window size	max=receiving window
	Whenever a loss is detected, decrease the window size	signal of congestion
	Repeat	

Design a *correct*, *timely*, *efficient* and *fair* transport mechanism
knowing that

packets can get lost
corrupted
reordered
delayed
duplicated

Dealing with corruption is easy:
Rely on a checksum, treat corrupted packets as lost

The effect of reordering depends on
the type of ACKing mechanism used

individual ACKs	no problem
full feedback	no problem
cumm. ACKs	create duplicate ACKs why is it a problem?

Long delays can create useless timeouts,
for all designs

Packets duplicates can lead to duplicate ACKs whose
effects will depend on the ACKing mechanism used

individual ACKs	no problem
full feedback	no problem
cumm. ACKs	problematic

Design a *correct*, *timely*, *efficient* and *fair* transport mechanism
knowing that

packets can get lost
corrupted
reordered
delayed
duplicated

Here is one correct, timely, efficient and fair transport mechanism

ACKing	full information ACK
retransmission	after timeout
	after *k* subsequent ACKs
window management	additive increase upon successful delivery
	multiple decrease when timeouts

We'll come back to this when we see TCP

Reliable Transport

Correctness condition
if-and-only if again

Design space
timeliness *vs* efficiency *vs* …

3 Examples
Go-Back-N & Selective Repeat

Go-Back-N (GBN) is a simple sliding window protocol using cumulative ACKs

principle	receiver should be as simple as possible
receiver	delivers packets in-order to the upper layer
	for each received segment, ACK the last in-order packet delivered (cumulative)
sender	use a single timer to detect loss, reset at each new ACK
	upon timeout, resend all W packets starting with the lost one

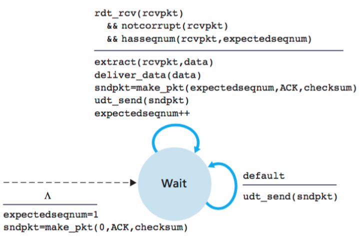
Selective Repeat (SR) avoid unnecessary retransmissions by using per-packet ACKs

see Book 3.4.3

principle	avoids unnecessary retransmissions
receiver	acknowledge each packet, in-order or not
	buffer out-of-order packets
sender	use per-packet timer to detect loss
	upon loss, only resend the lost packet

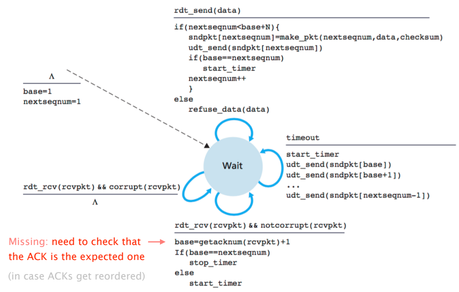
Finite State Machine for the receiver

see Book 3.4.3



Finite State Machine for the sender

see Book 3.4.3



Let's see how it works in practice

visually

http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/

Reliable Transport

Correctness condition
if-and-only if again

Design space
timeliness *vs* efficiency *vs* …

Examples
Go-Back-N & Selective Repeat