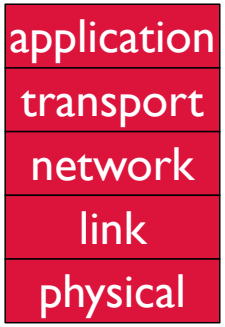
Layers



Firefox, `ping` , ...

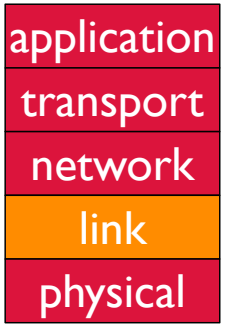
TCP, UDP, ...

IP

ethernet, WiFi, ...

electrons, photons, ...

Layers



Firefox, `ping` , ...

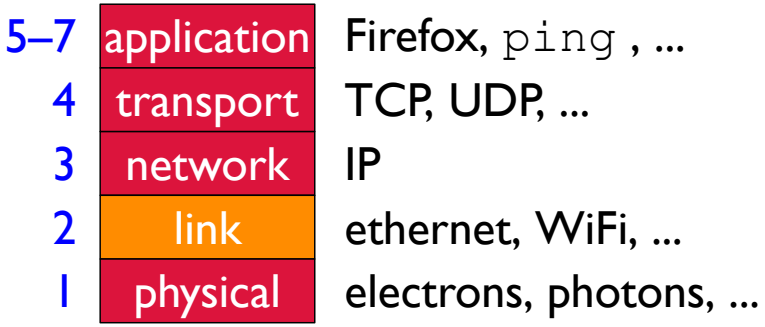
TCP, UDP, ...

IP

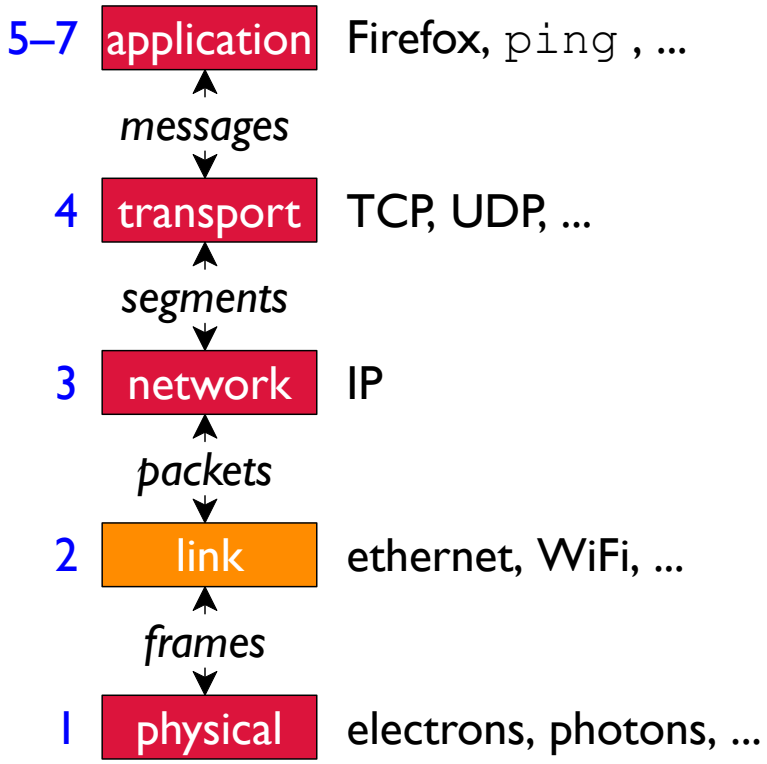
ethernet, WiFi, ...

electrons, photons, ...

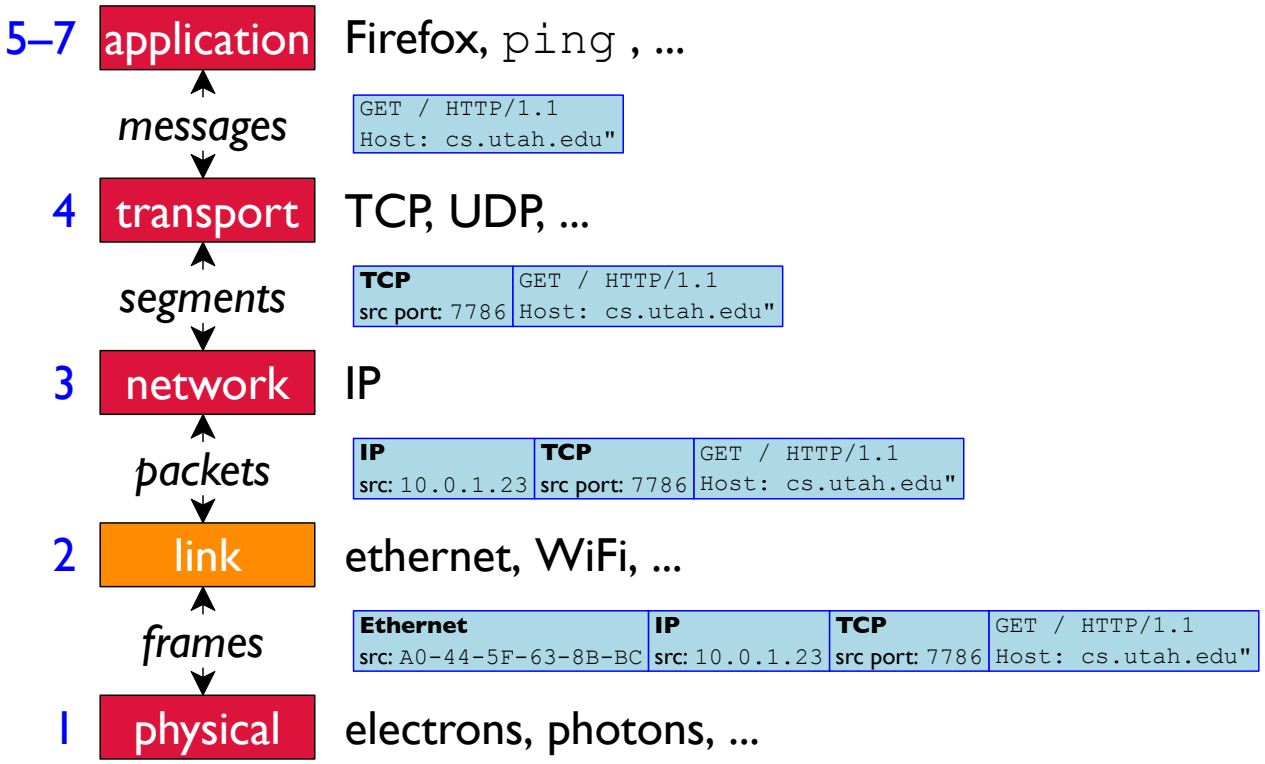
Layers


5–7
application
Firefox, ping , ...
4
transport
TCP, UDP, ...
3
network
IP
2
link
ethernet, WiFi, ...
1
physical
electrons, photons, ...

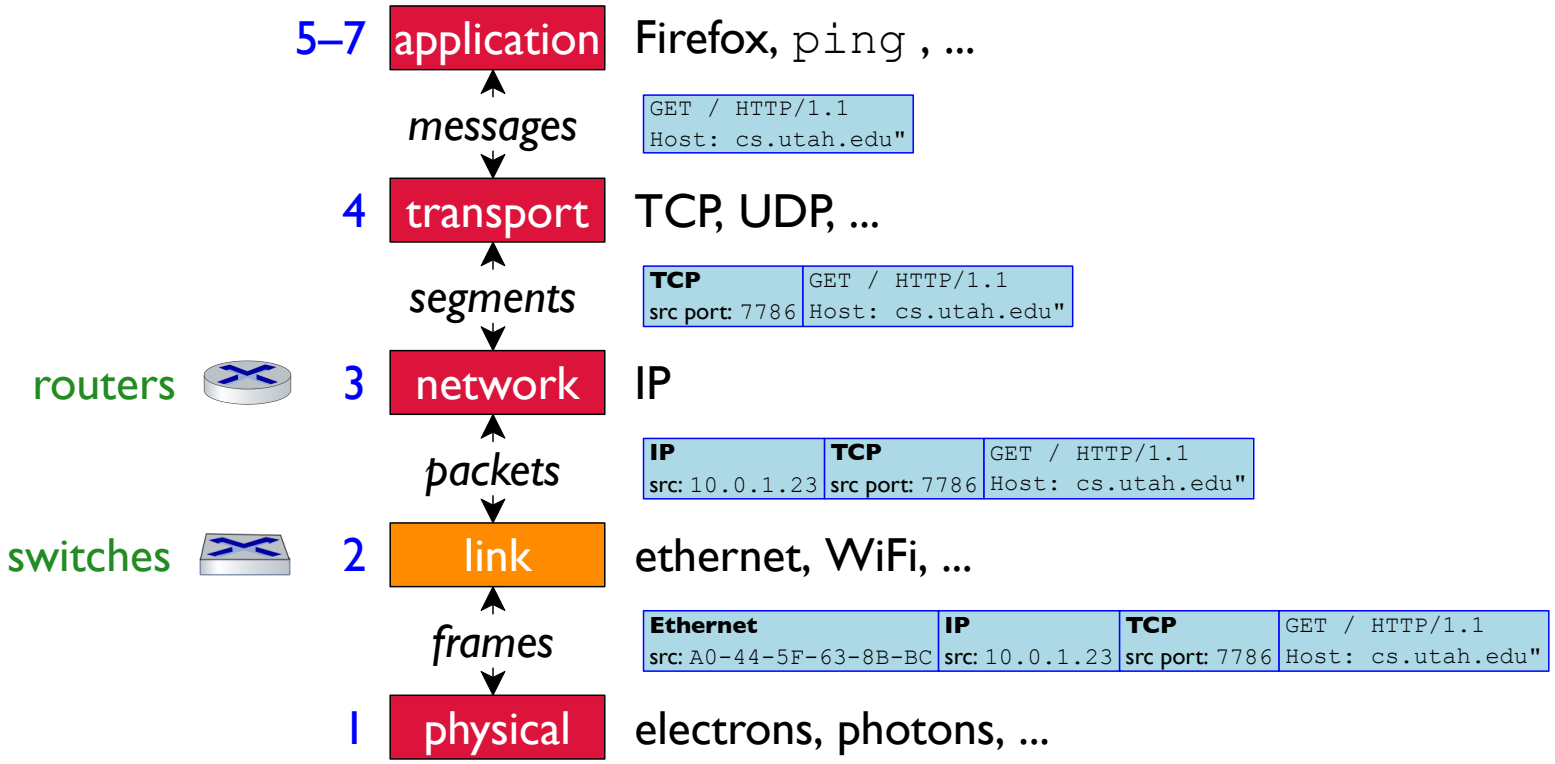
Layers


5–7
application
Firefox, ping , ...
messages
4
transport
TCP, UDP, ...
segments
3
network
IP
packets
2
link
ethernet, WiFi, ...
frames
1
physical
electrons, photons, ...

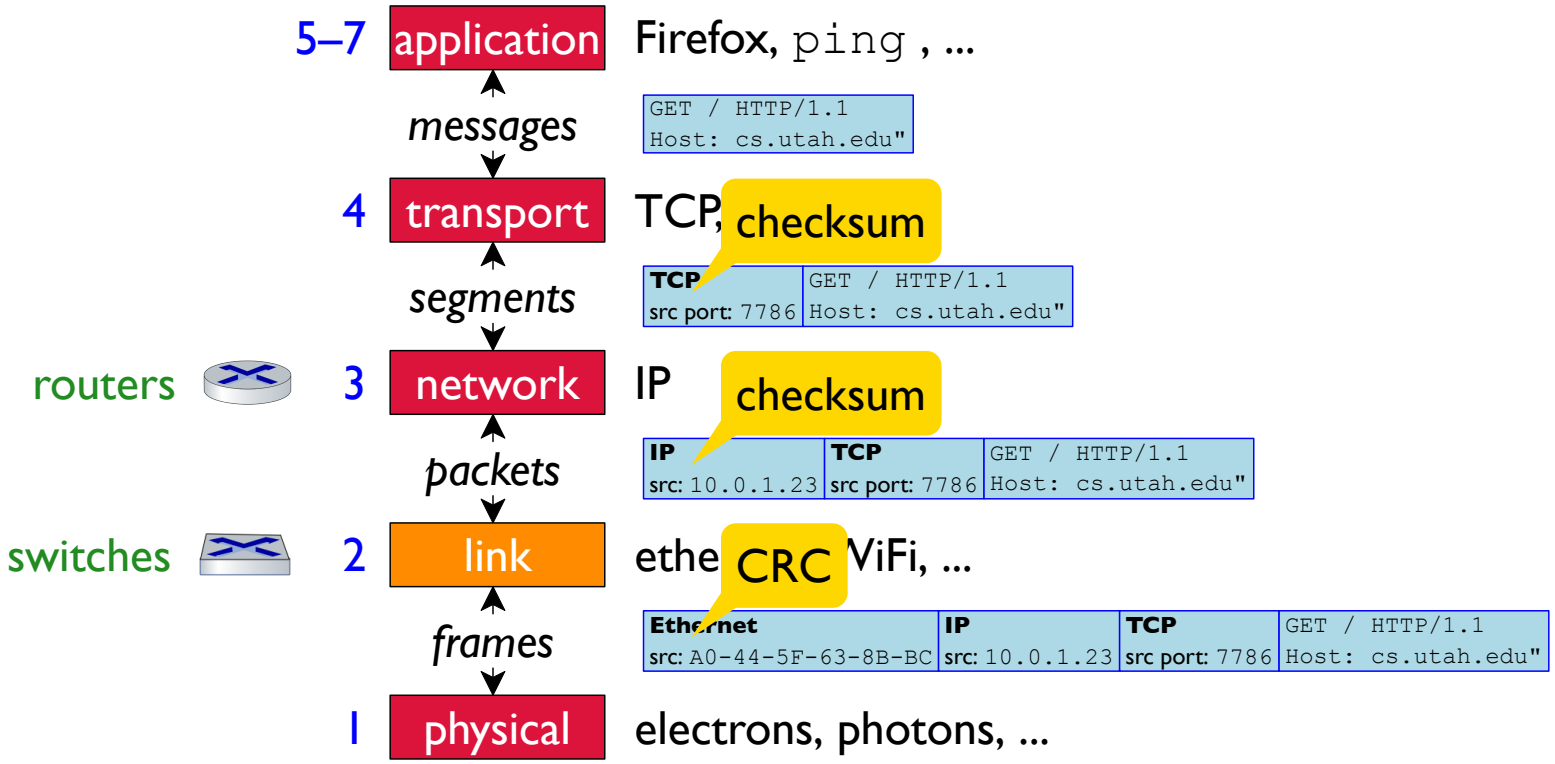
Layers


5–7
application
Firefox, ping , ...
messages
GET / HTTP/1.1
Host: cs.utah.edu"
4
transport
TCP, UDP, ...
segments
TCP
src port: 7786
GET / HTTP/1.1
Host: cs.utah.edu"
3
network
IP
packets
IP
src: 10.0.1.23
TCP
src port: 7786
GET / HTTP/1.1
Host: cs.utah.edu"
2
link
ethernet, WiFi, ...
frames
Ethernet
src: A0-44-5F-63-8B-BC
IP
src: 10.0.1.23
TCP
src port: 7786
GET / HTTP/1.1
Host: cs.utah.edu"
1
physical
electrons, photons, ...

Layers


5–7
application
Firefox, ping , ...
messages
GET / HTTP/1.1
Host: cs.utah.edu"
4
transport
TCP, UDP, ...
segments
TCP
src port: 7786
GET / HTTP/1.1
Host: cs.utah.edu"
routers
3
network
IP
packets
IP
src: 10.0.1.23
TCP
src port: 7786
GET / HTTP/1.1
Host: cs.utah.edu"
switches
2
link
ethernet, WiFi, ...
frames
Ethernet
src: A0-44-5F-63-8B-BC
IP
src: 10.0.1.23
TCP
src port: 7786
GET / HTTP/1.1
Host: cs.utah.edu"
1
physical
electrons, photons, ...

Layers


5–7
application
Firefox, ping , ...
messages
GET / HTTP/1.1
Host: cs.utah.edu"
4
transport
TCP,
checksum
TCP
src port: 7786
GET / HTTP/1.1
Host: cs.utah.edu"
segments
routers
3
network
IP
checksum
IP
src: 10.0.1.23
TCP
src port: 7786
GET / HTTP/1.1
Host: cs.utah.edu"
packets
switches
2
link
ethe
CRC
ViFi, ...
Ethernet
src: A0-44-5F-63-8B-BC
IP
src: 10.0.1.23
TCP
src port: 7786
GET / HTTP/1.1
Host: cs.utah.edu"
frames
1
physical
electrons, photons, ...

Weak Checksum: Parity

A 1-bit checksum is a **parity** check

... since that 1 bit is either on/odd or off/even

```
10001010  →  1
10011010  →  0
10111010  →  1
```

Weak Checksum: Parity

A 1-bit checksum is a **parity** check

... since that 1 bit is either on/odd or off/even

```
10001010  →  1
10011010  →  0
10111010  →  1
```

Fast, but two corrupt bits cancel, which is especially bad when corruption is bursty

Strong: Cyclic Redundancy Check

A checksum based on adding numbers and keeping only low bits
is a kind of hash function... but not an especially good hash function

Strong: Cyclic Redundancy Check

A checksum based on adding numbers and keeping only low bits
is a kind of hash function... but not an especially good hash function

A **cyclic redundancy check (CRC)** is a much better hash function:

d = number of bits to check
D = d bits of data
r = bits for hash result (typically 8, 12, 16, or 32)
R = the hash of D
G = a carefully chosen, agreed-on $r+1$-bit number

$$R = \text{remainder of } \frac{D \times 2^r}{G}$$

Strong: Cyclic Redundancy Check

A checksum based on adding numbers and keeping only low bits
is a kind of hash function... but not an especially good hash function

A **cyclic redundancy check (CRC)** is a much better hash function:

d = number of bits to check
D = d bits of data
r = bits for hash result (typically 8, 12, 16, or 32)
R = the hash of D
G = a carefully chosen, agreed-on r+1-bit number

$$R = \text{remainder of } \frac{D \times 2^r}{G}$$

For $r = 32$, IEEE standard is $G =$ `0x104C11DB7`

Strong: Cyclic Redundancy Check

A checksum based on adding numbers and keeping only low bits
is a kind of hash function... but not an especially good hash function

A **cyclic redundancy check (CRC)** is a much better hash function:

d = number of bits to check
D = d bits of data
r = bits for hash result (typically 8, 12, 16, or 32)
R = the hash of D
G = a carefully chosen, agreed-on r+1-bit number

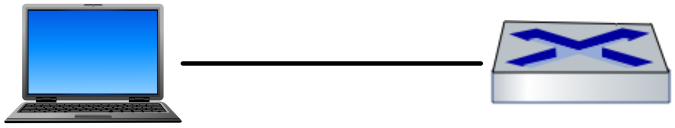
$$R = \text{remainder of } \frac{D \times 2^r}{G}$$

For $r = 32$, IEEE standard is $G =$ `0x104C11DB7`

Detects any r -bit error burst

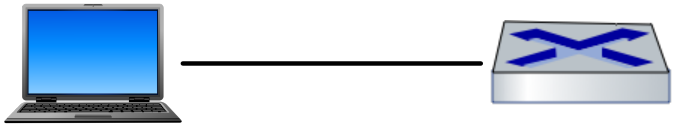
Coordinating Communication

Easy mode : point-to-point communication

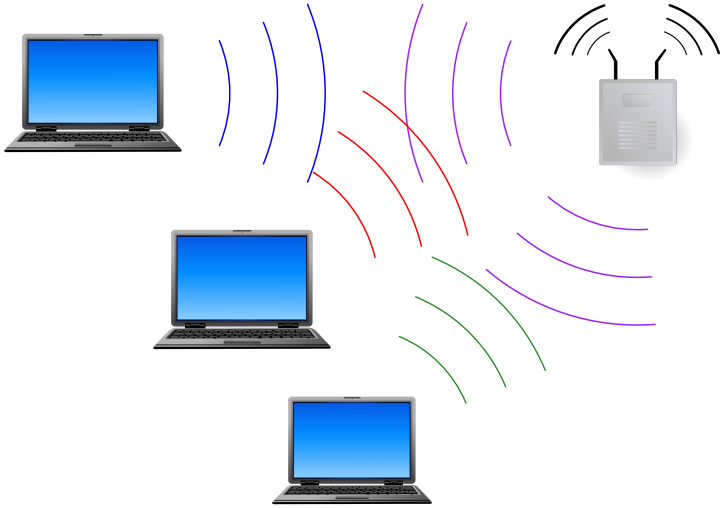


Coordinating Communication

Easy mode : point-to-point communication

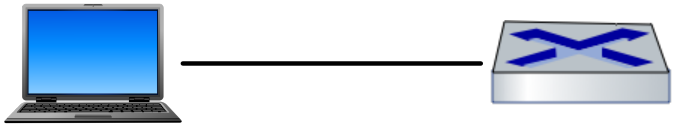


Hard mode : shared communication medium

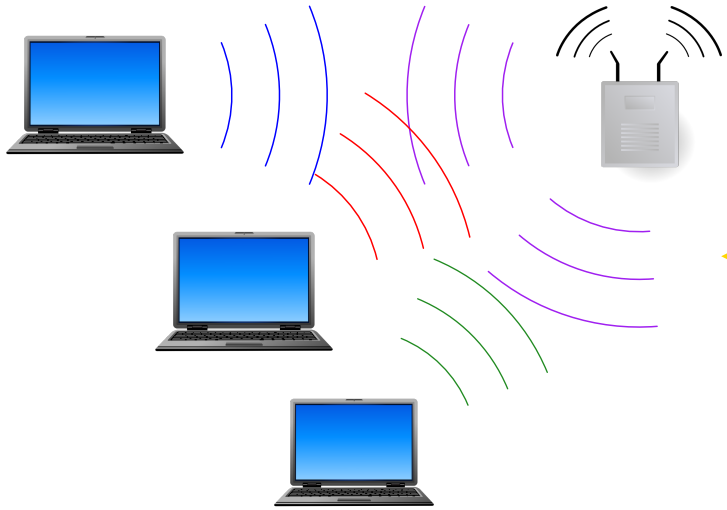


Coordinating Communication

Easy mode : point-to-point communication



Hard mode : shared communication medium

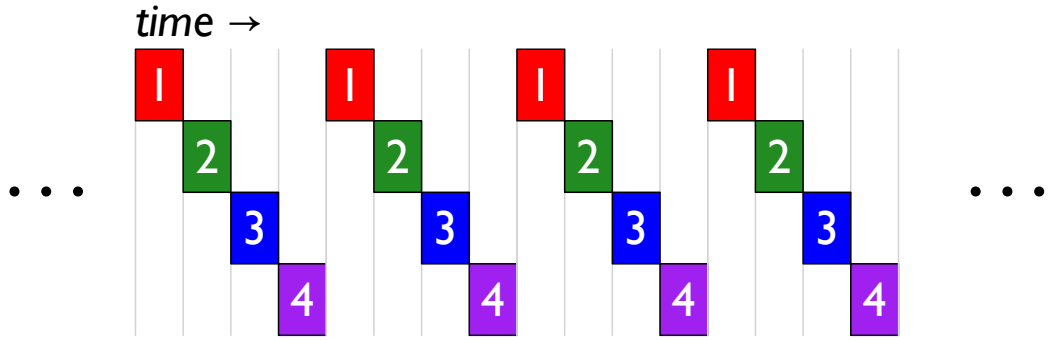


Goals:
- divide bandwidth fairly
- only one active

  ⇒ gets full bandwidth

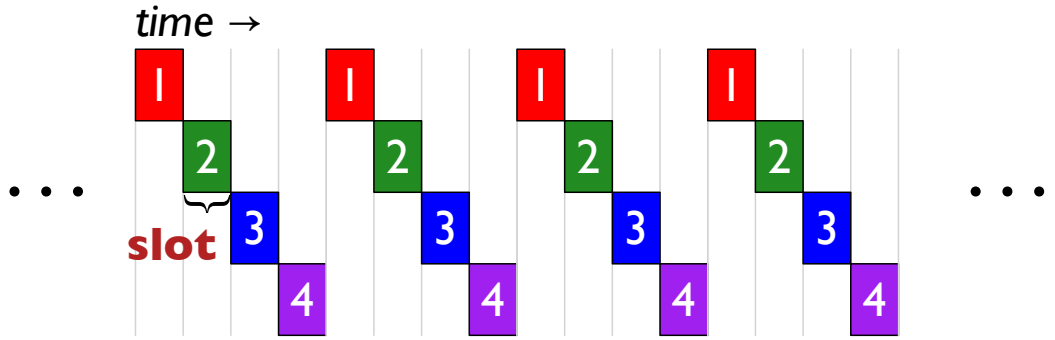
Shared-Medium Strategy 1: Channel Partitioning

Time-devision multiplexing (TDM) :



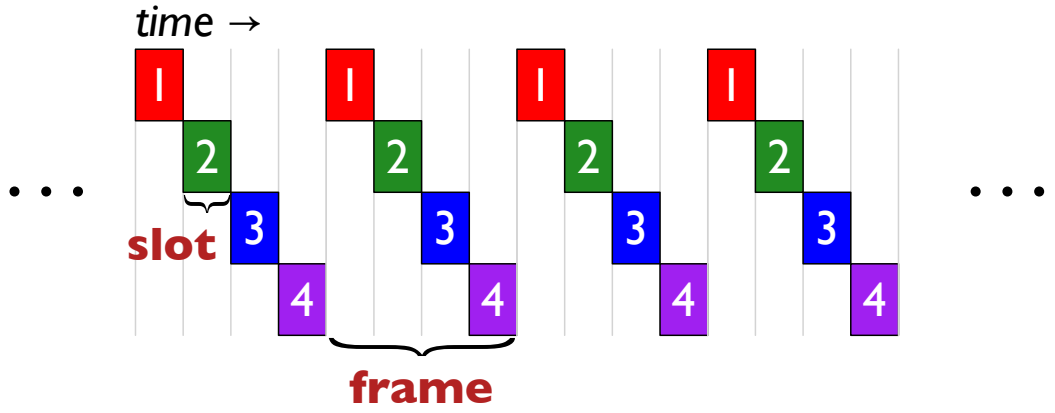
Shared-Medium Strategy 1: Channel Partitioning

Time-devision multiplexing (TDM) :



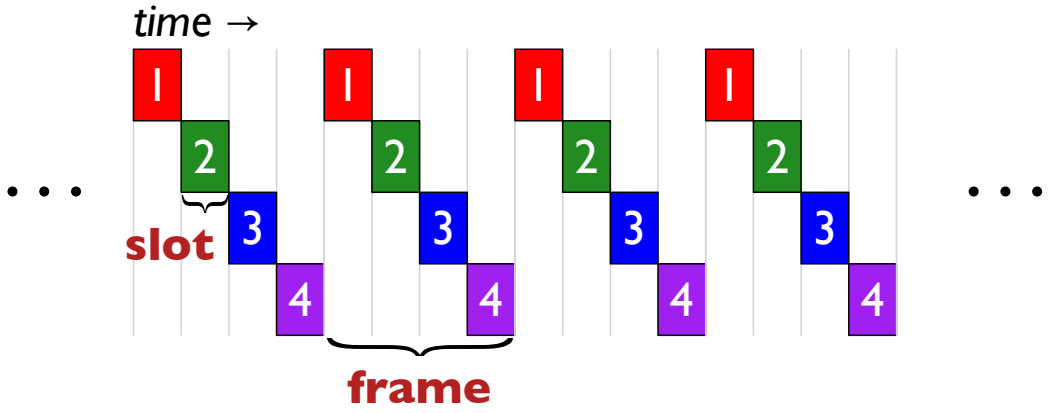
Shared-Medium Strategy 1: Channel Partitioning

Time-devision multiplexing (TDM) :



Shared-Medium Strategy 1: Channel Partitioning

Time-devision multiplexing (TDM) :

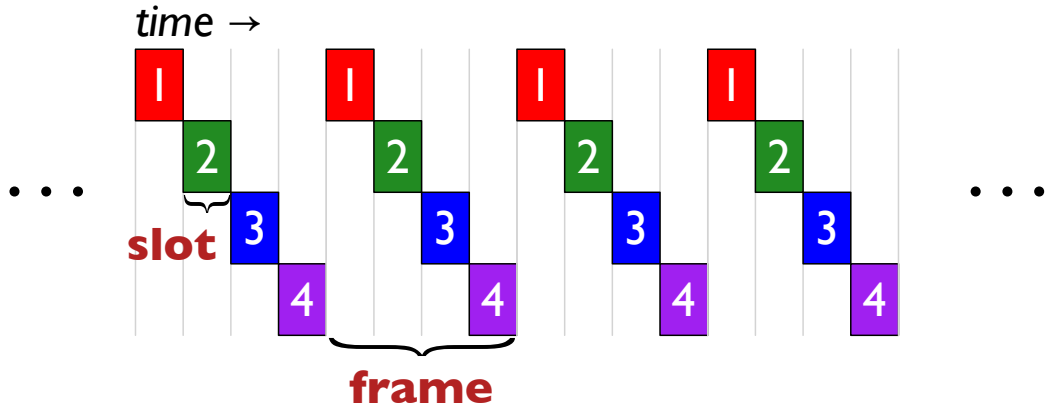


Frequency-devision multiplexing (FDM) :

same idea, but for simultaneous frequencies

Shared-Medium Strategy 1: Channel Partitioning

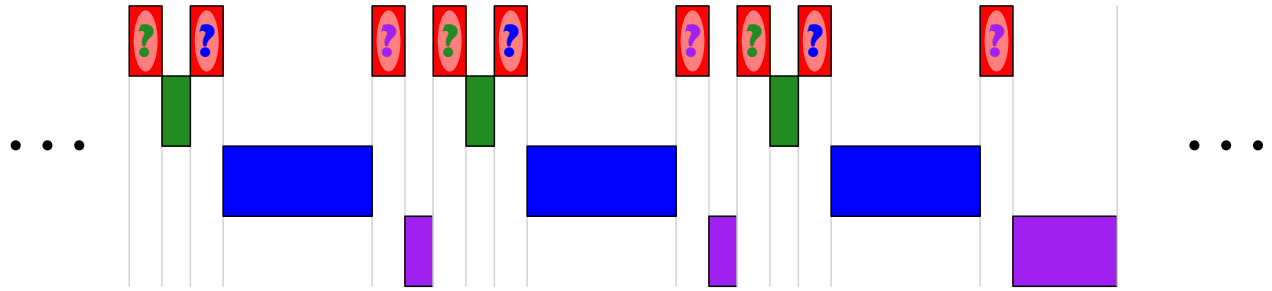
Time-devision multiplexing (TDM) :



+ No collisions
+ Perfectly fair
– Poor utilization when some are idle

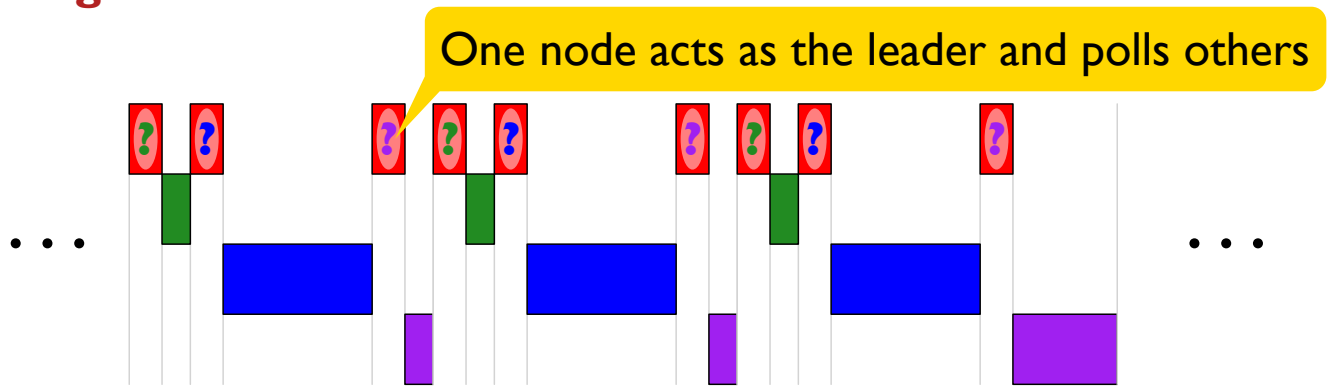
Shared-Medium Strategy 2: Turn Taking

Polling :



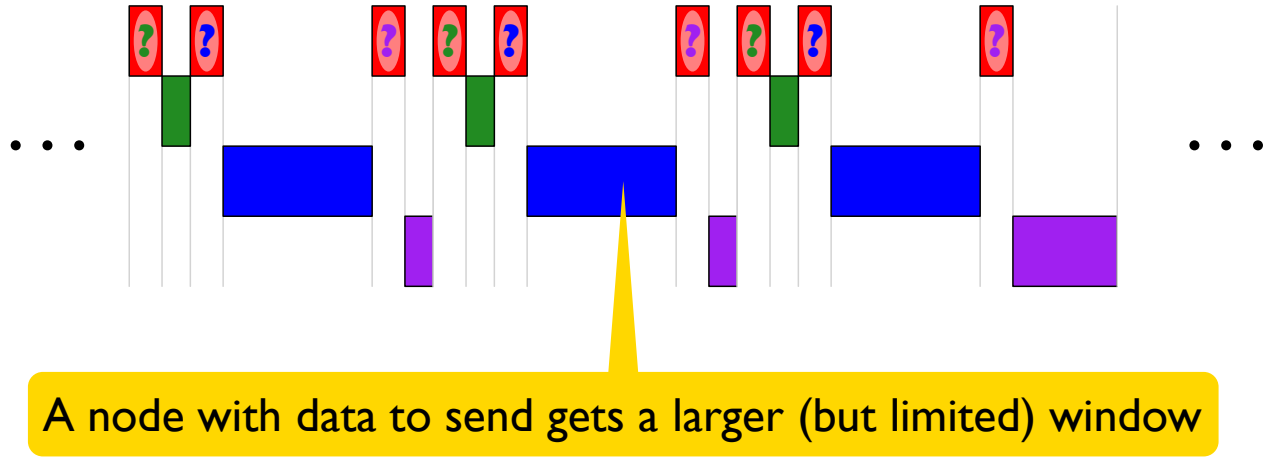
Shared-Medium Strategy 2: Turn Taking

Polling :



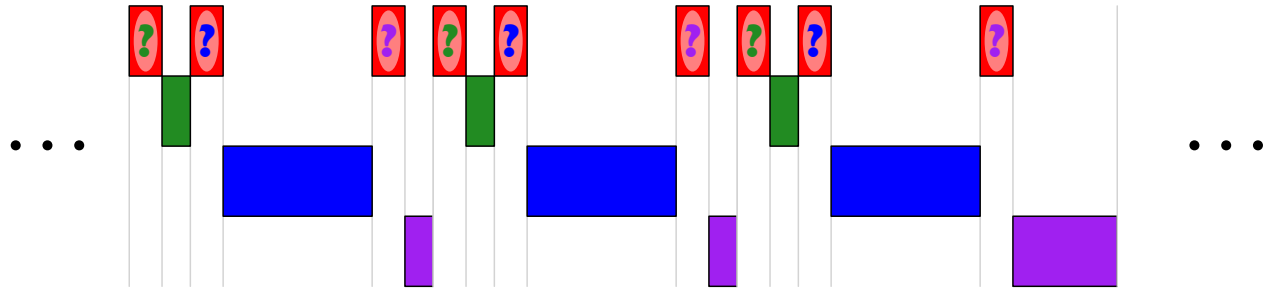
Shared-Medium Strategy 2: Turn Taking

Polling :



Shared-Medium Strategy 2: Turn Taking

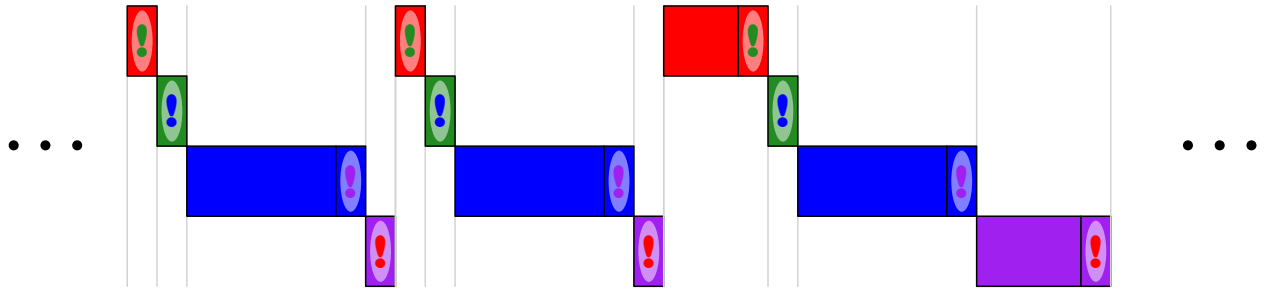
Polling :



+ Better utilization
– Polling causes delays
– Recovery needed if the leader fails

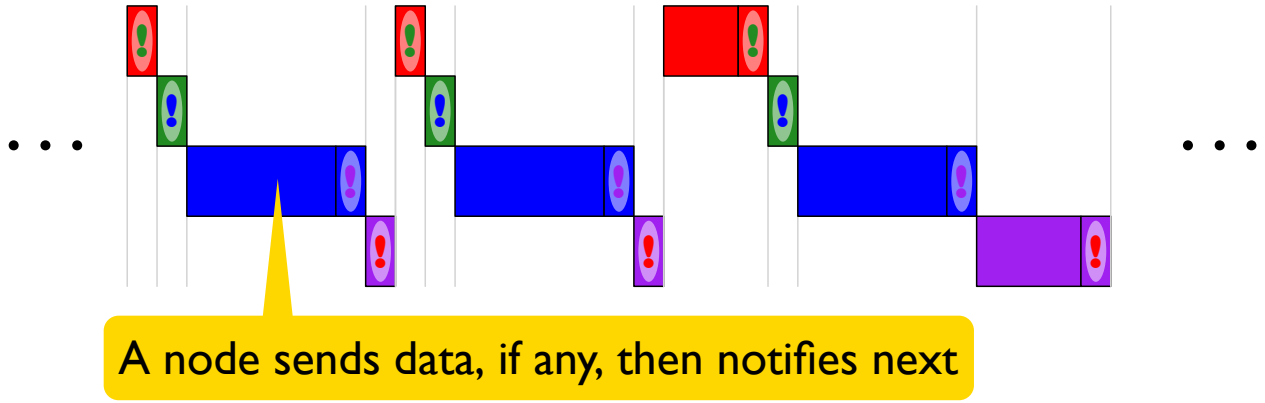
Shared-Medium Strategy 2: Turn Taking

Token passing :



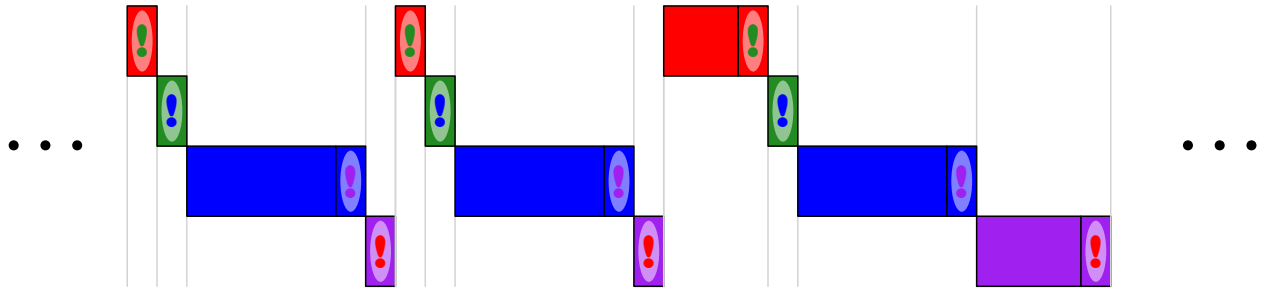
Shared-Medium Strategy 2: Turn Taking

Token passing :



Shared-Medium Strategy 2: Turn Taking

Token passing :



+ Better utilization
- Token-passing causes delays
- Recovery needed if any fails

Shared-Medium Strategy 3: Random Access

A **random access** strategy requires either

- detection of collisions by senders
- **carrier sense** to detect when someone is already sending

or both

Shared-Medium Strategy 3: Random Access

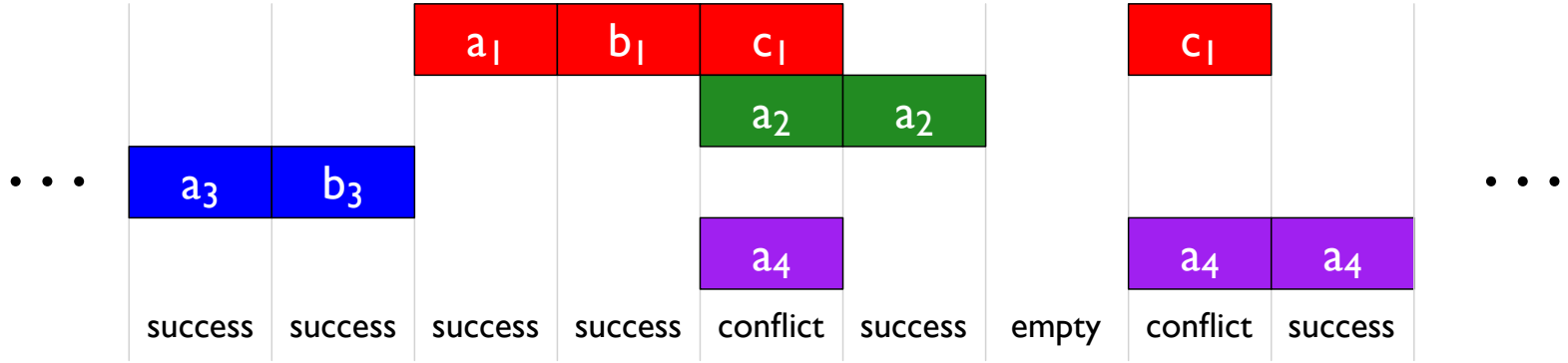
A **random access** strategy requires either

- detection of collisions by senders
- **carrier sense** to detect when someone is already sending

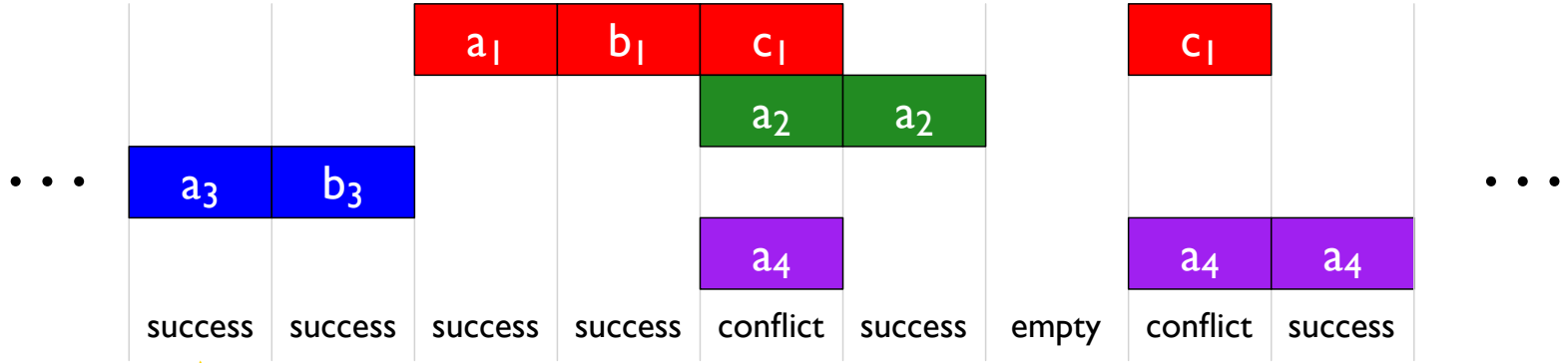
or both

A random delay is used when a collision is detected

Random Access: ALOHA

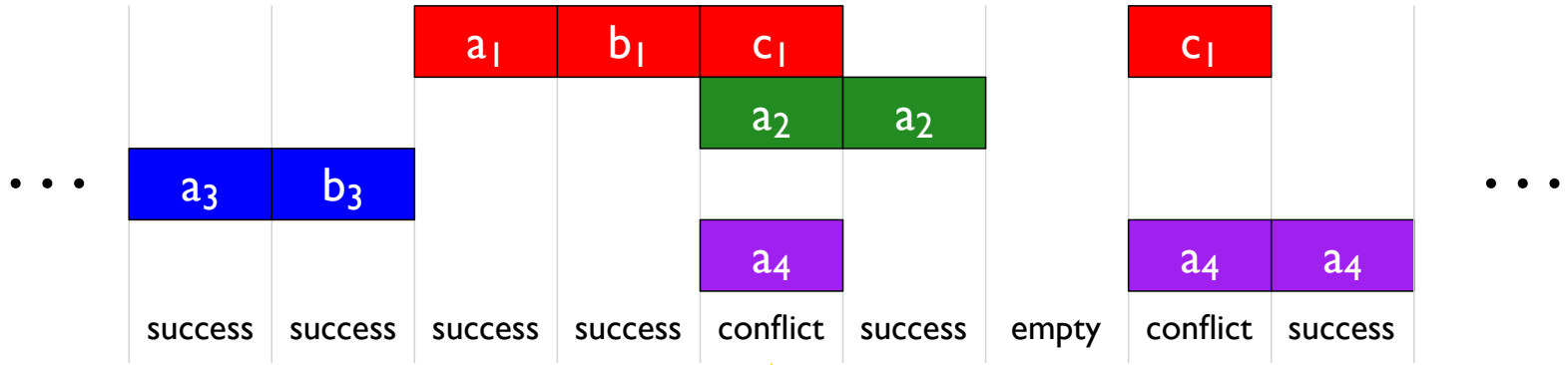


Random Access: ALOHA



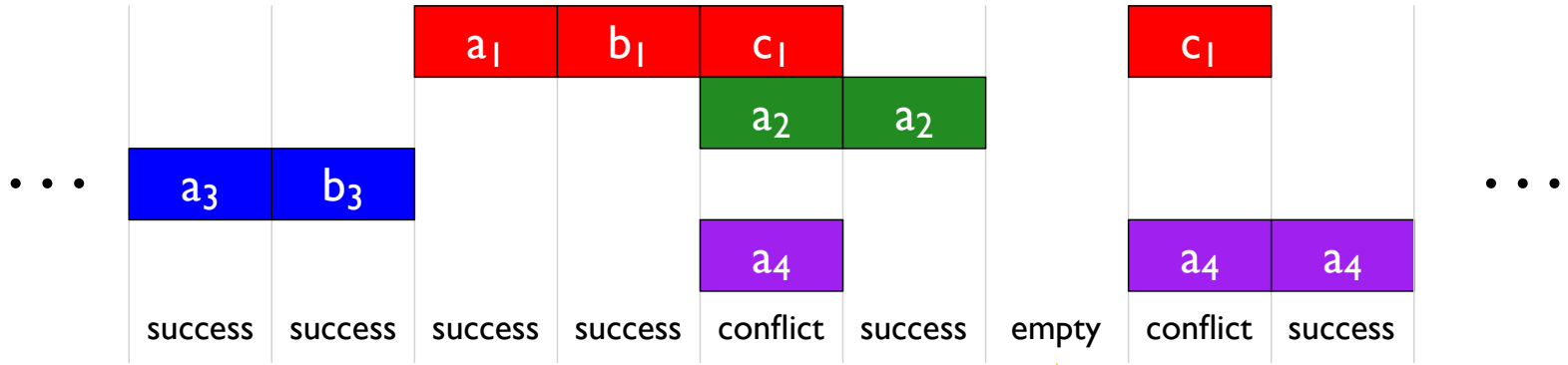
On success, a node can keep sending as long as it has data

Random Access: ALOHA

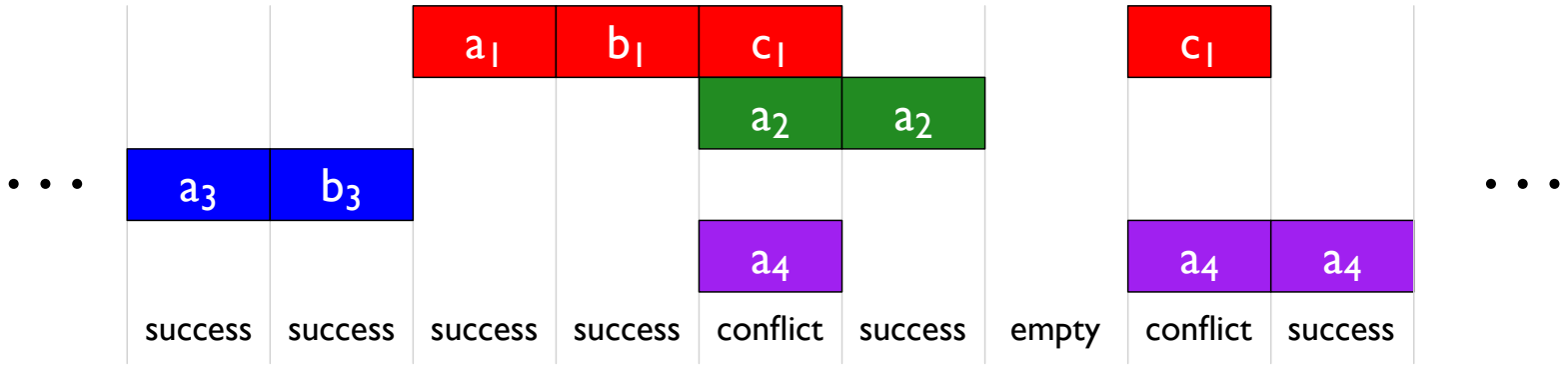


On conflict, each node retries on next slot probability P

Random Access: ALOHA



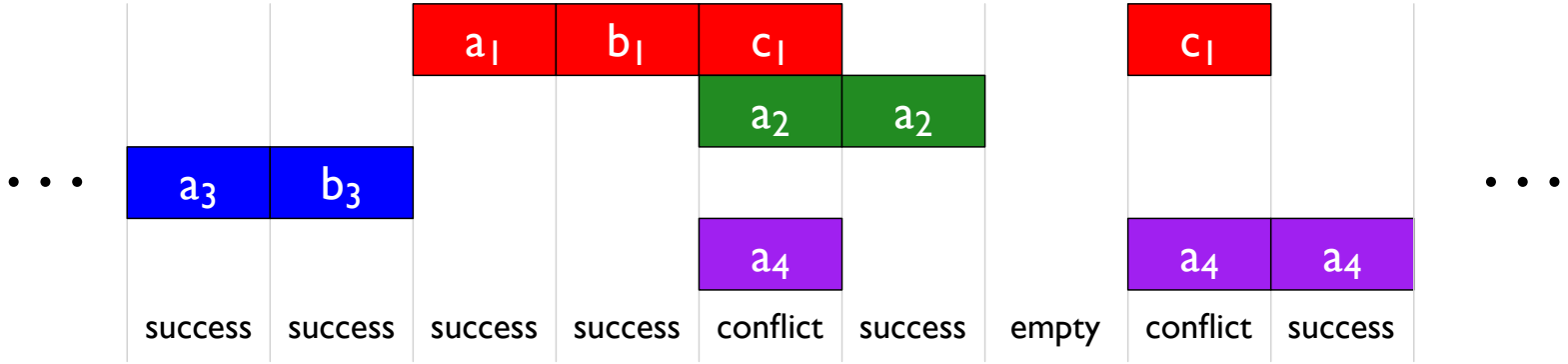
Random Access: ALOHA



Slotted ALOHA , which needs synchronization:

+ Sole active nodes can use full bandwidth
+ Multiple active nodes get fair share
– Even after optimizing P, likely to get only 37% success

Random Access: ALOHA



Original **unslotted ALOHA** avoided synchronization:

- Success drop drops by half

  because each local slot likely overlaps two other peer slots

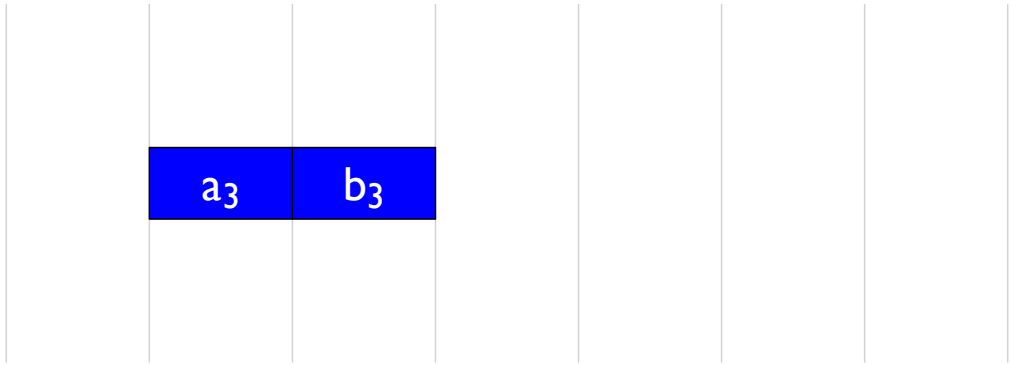
Random Access: Carrier Sense

Carrier Sense Multiple Access (CSMA) means
"don't talk when someone else is talking"

Random Access: Carrier Sense

Carrier Sense Multiple Access (CSMA) means
"don't talk when someone else is talking"

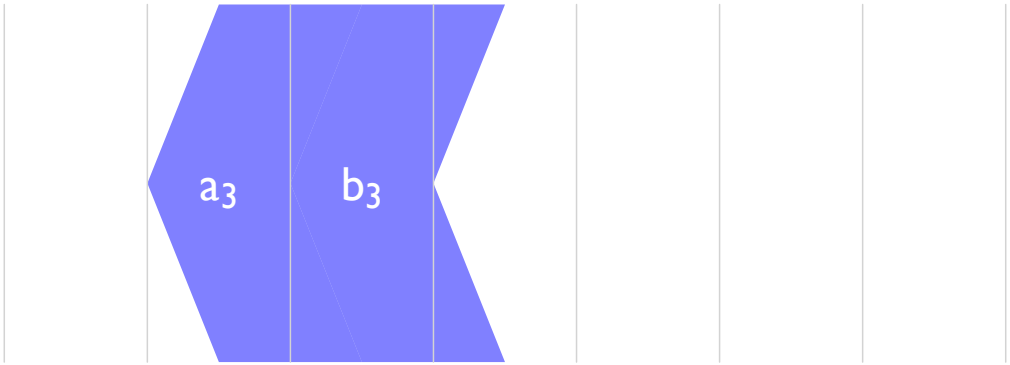
The catch: there's a delay between the time that one node sends and another node starts to sense it



Random Access: Carrier Sense

Carrier Sense Multiple Access (CSMA) means
"don't talk when someone else is talking"

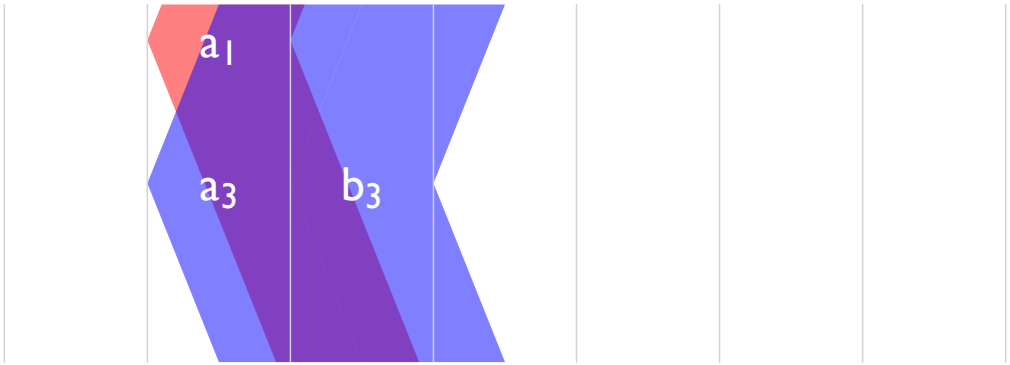
The catch: there's a delay between the time that one node sends and another node starts to sense it



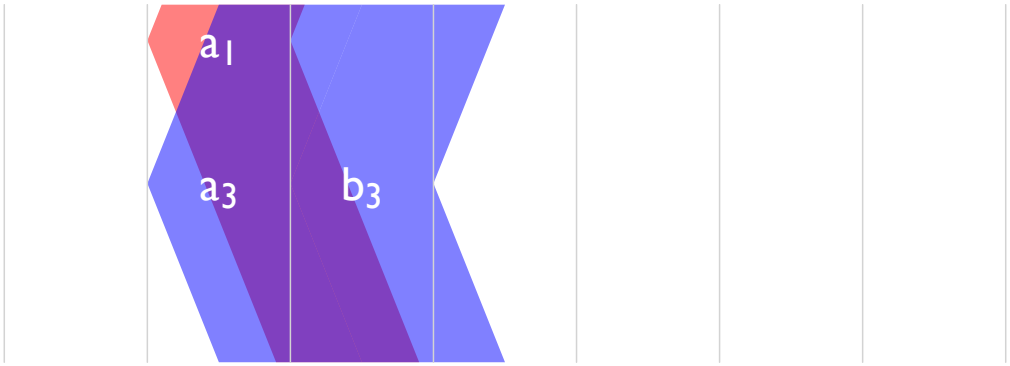
Random Access: Carrier Sense

Carrier Sense Multiple Access (CSMA) means
"don't talk when someone else is talking"

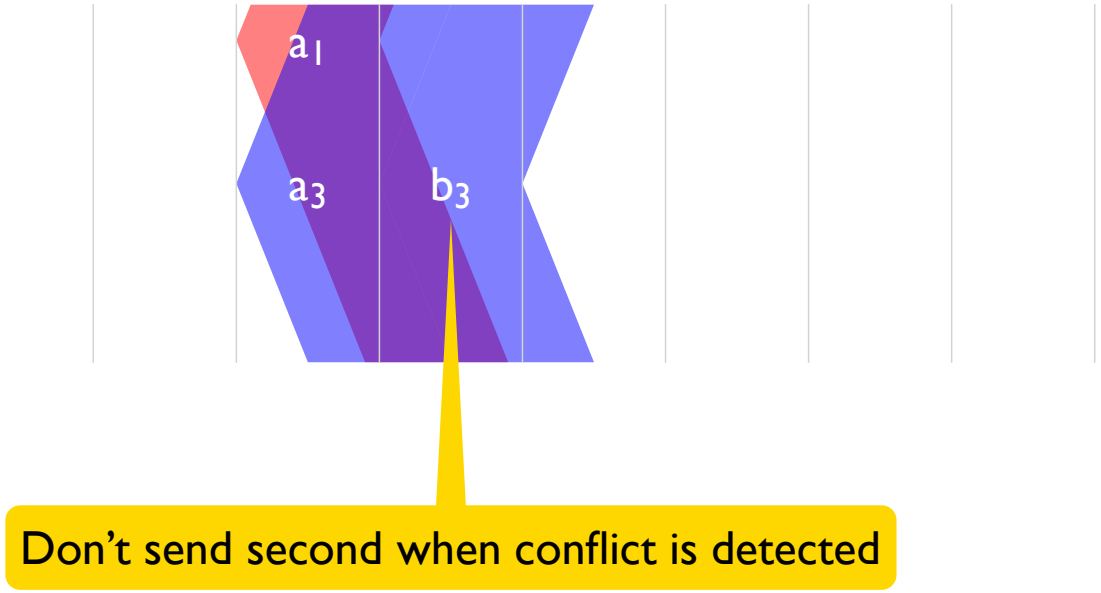
The catch: there's a delay between the time that one node sends and another node starts to sense it



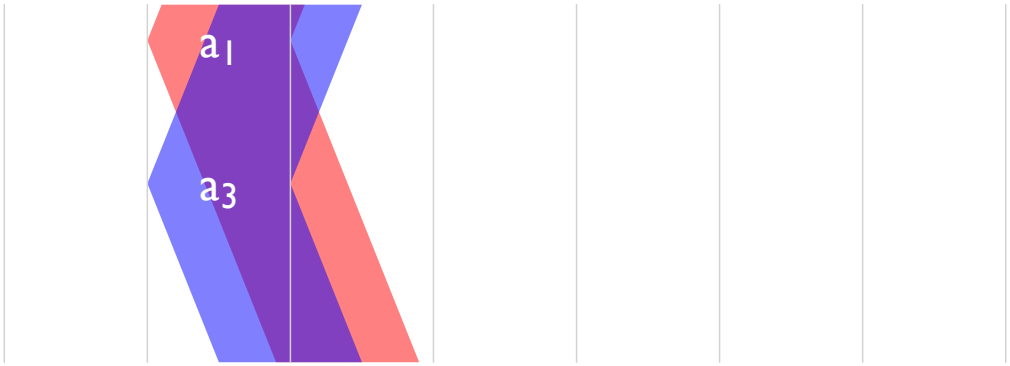
Handling Conflicts in CSMA



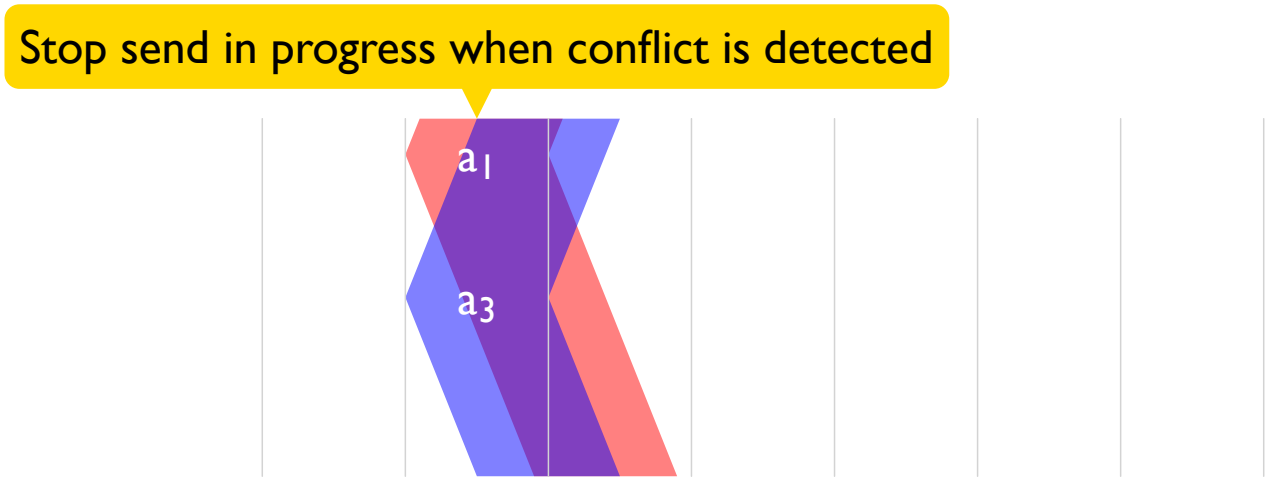
Handling Conflicts in CSMA



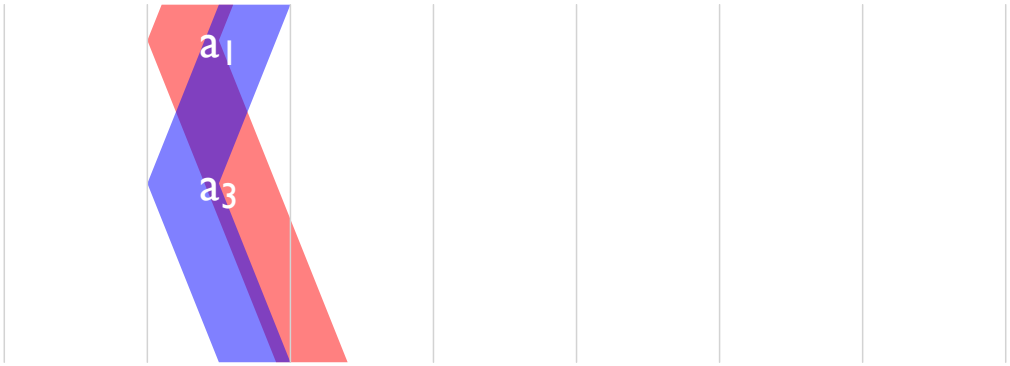
Handling Conflicts in CSMA



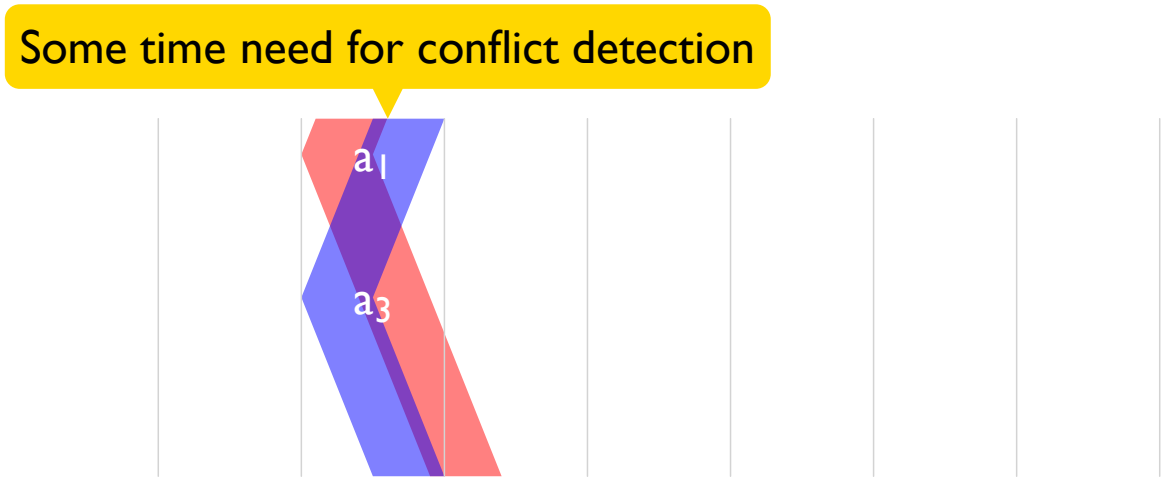
Handling Conflicts in CSMA



Handling Conflicts in CSMA

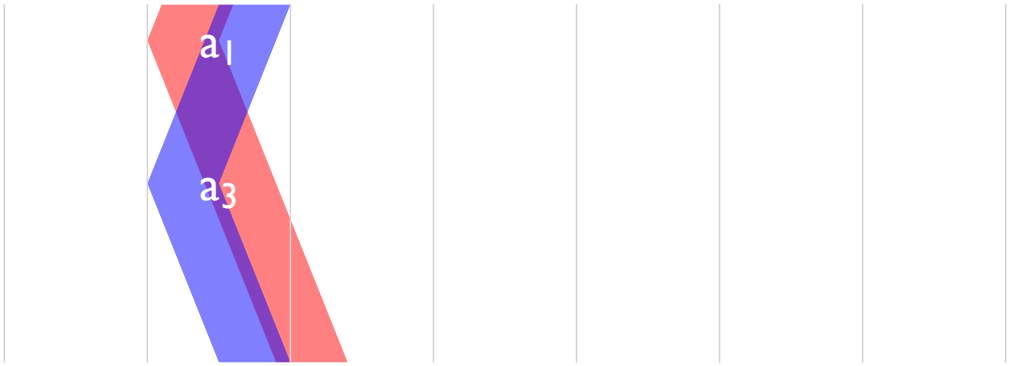


Handling Conflicts in CSMA

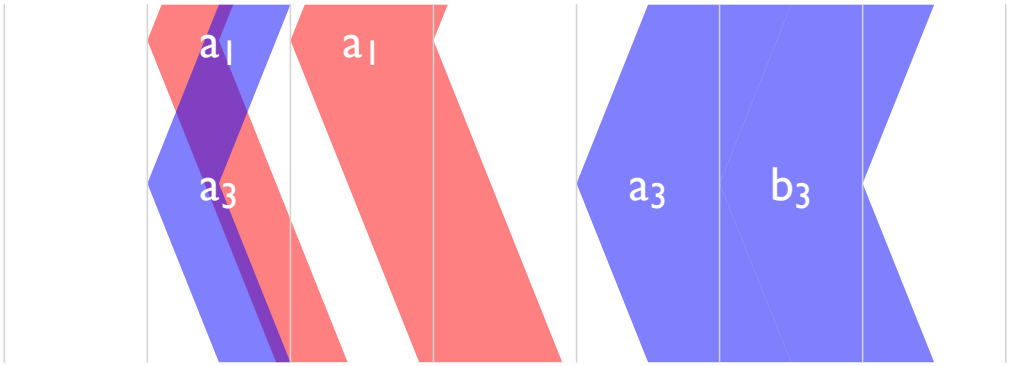


Handling Conflicts in CSMA

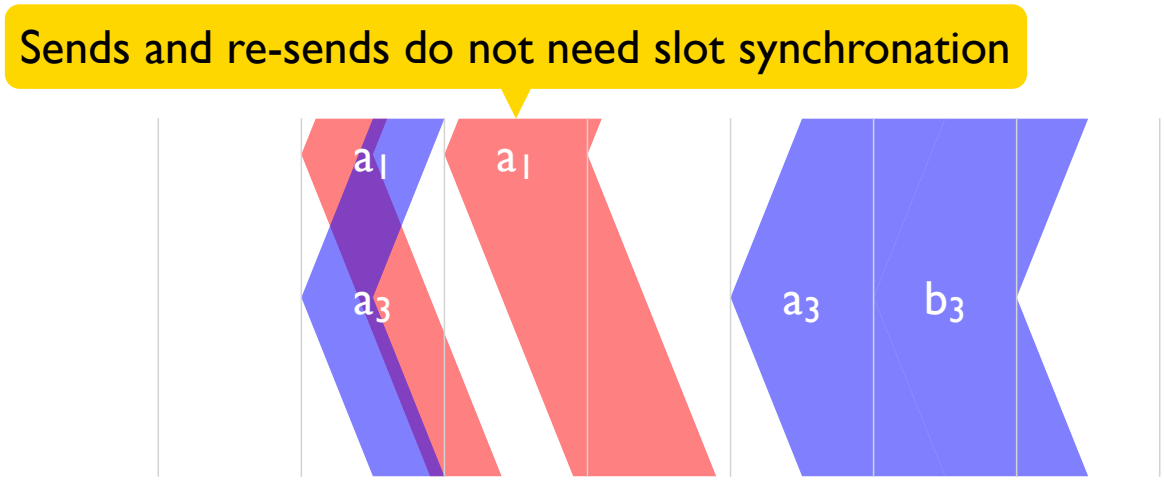
Random delay before retry



Handling Conflicts in CSMA

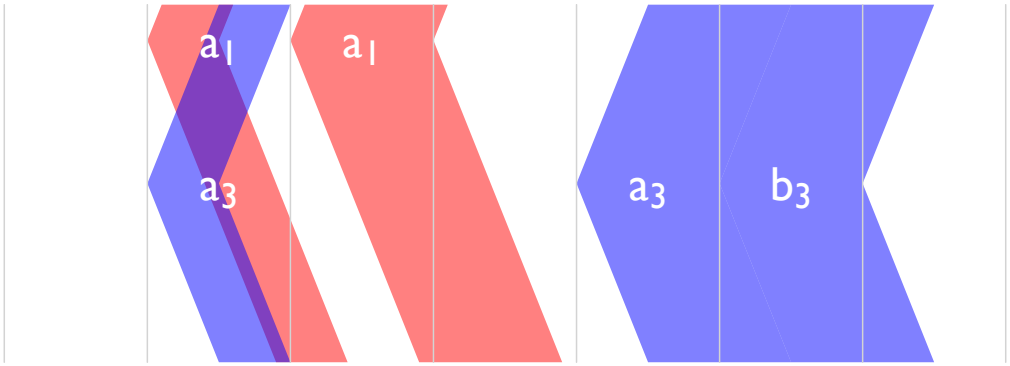


Handling Conflicts in CSMA

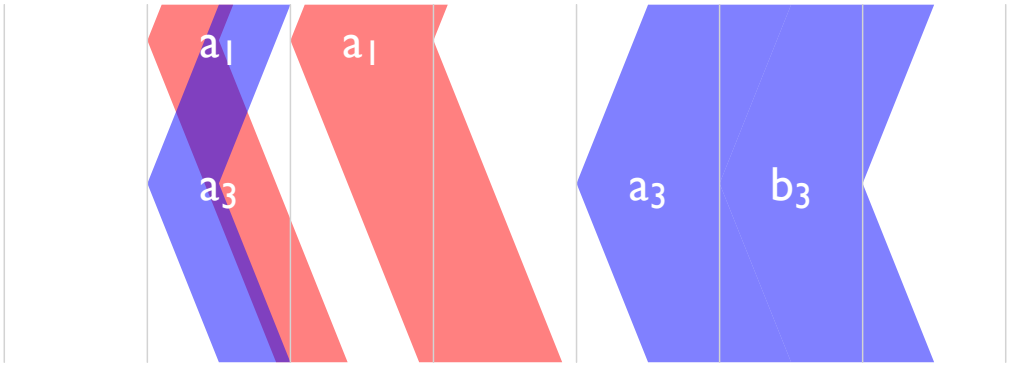


Handling Conflicts in CSMA

Exponential back-off:
If another conflict, double average retry delay



Handling Conflicts in CSMA

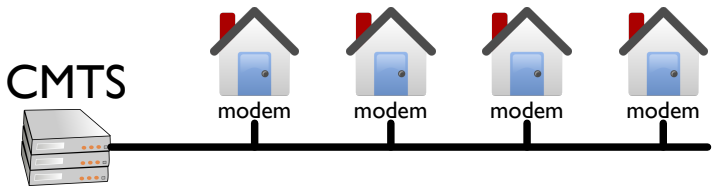


d_{prop} = max delay for signal

d_{trans} = max duration for frame

$$\text{efficiency} = \frac{1}{1 + 5\frac{d_{prop}}{d_{trans}}}$$

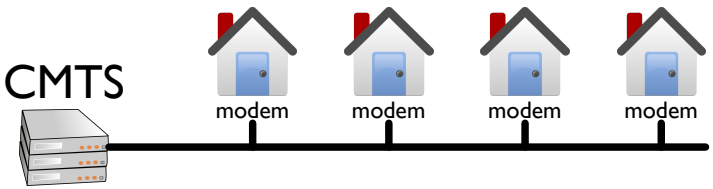
DOCSIS Cable Internet Protocol



Shared line is split into channels by frequency (FDM):

- Some channels are download: all modems receive
- Remaining channels are upload: CMTS receives

DOCSIS Cable Internet Protocol

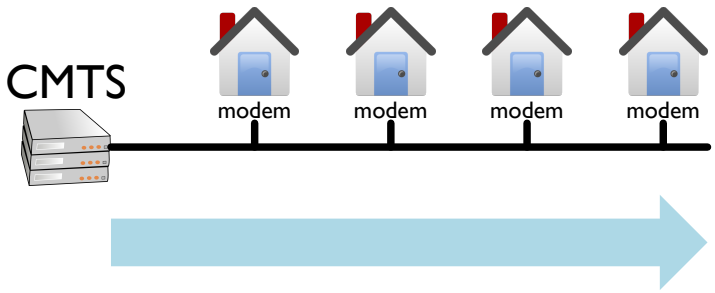


Shared line is split into channels by frequency (FDM):

- Some channels are download: all modems receive
- Remaining channels are upload: CMTS receives

Fewer, which explains asymmetric speed

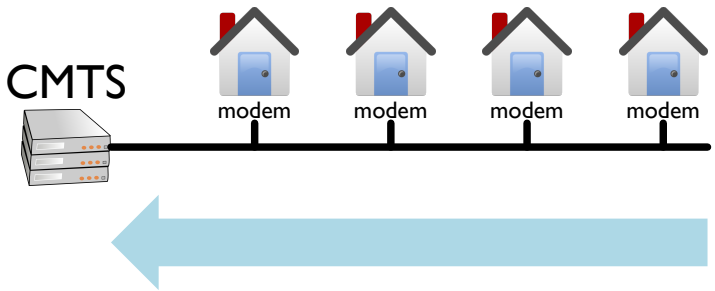
DOCSIS Cable Internet Protocol



Download:

- Only one sender, so no coordination needed
- Modems listen on all channels, but they ignore frames intended for others

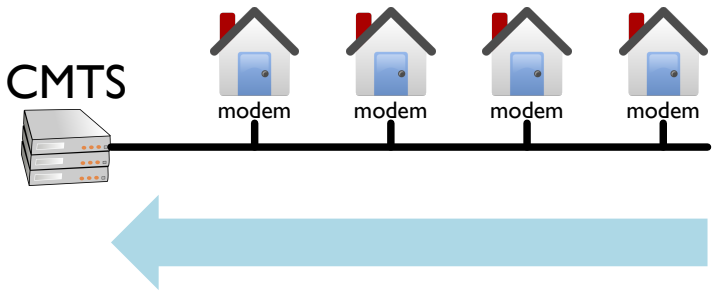
DOCSIS Cable Internet Protocol



Upload uses a hybrid of TDM and polling-ish turn taking:

- Designated short slots used for requests
- CMTS broadcasts slot assignments to grant modem requests
- Modem uses assigned slots for data upload

DOCSIS Cable Internet Protocol



Ok to rely on leader, because CMTS has to work, anyway

Upload uses a hybrid of TDM and polling-ish turn taking:

- Designated short slots used for requests
- CMTS broadcasts slot assignments to grant modem requests
- Modem uses assigned slots for data upload