INFORMATICA®

Running with the Devil: Mechanical Sympathetic Networking

Todd L. Montgomery
@toddlmontgomery
Informatica Ultra Messaging Architecture

Tail of a Networking Stack

Beastie

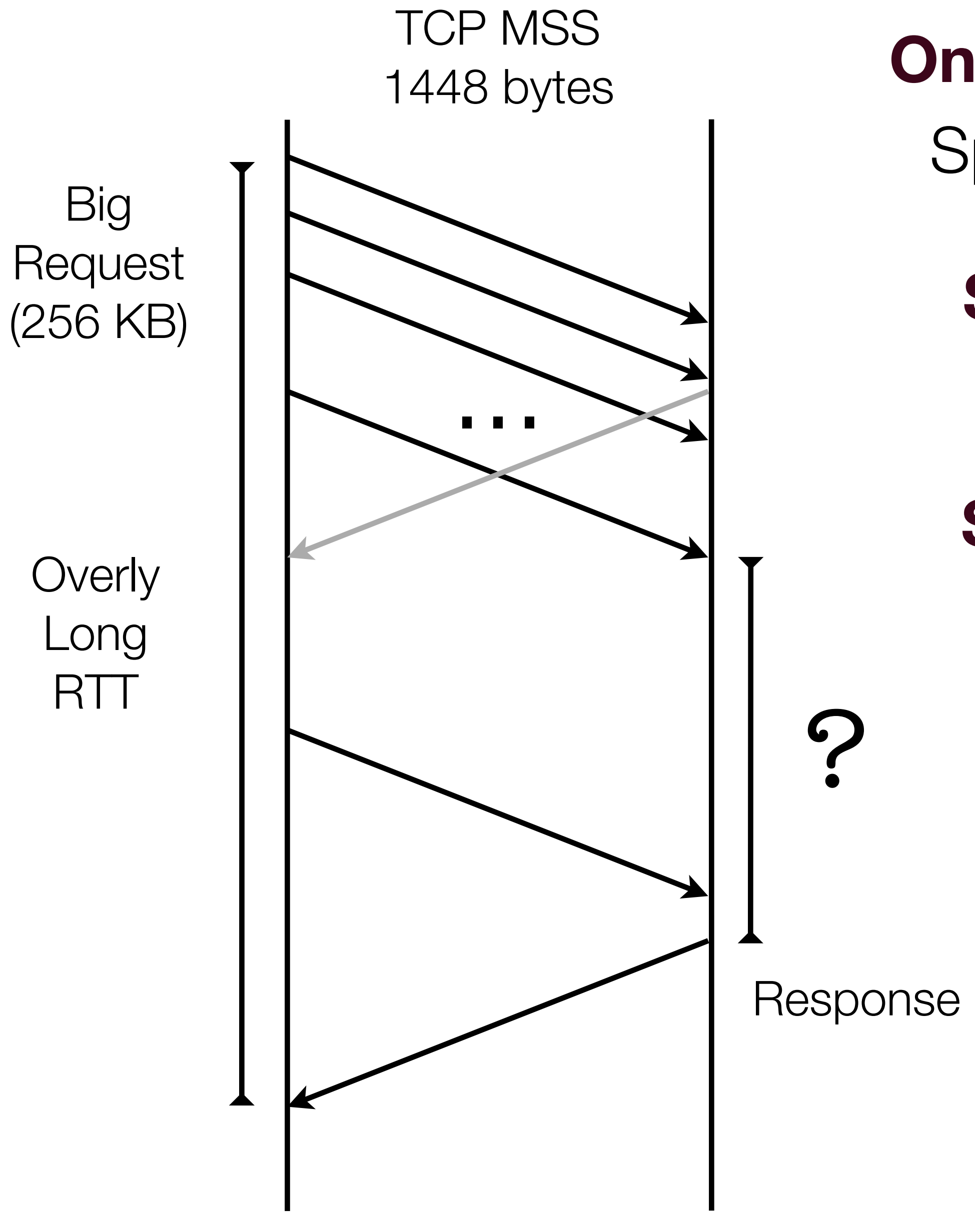
Direct Descendants

FreeBSD
NetBSD
OpenBSD
...
Darwin (Mac OS X)

also
Windows, Solaris, even Linux,
Android, ...

Domain: TCP & UDP

It’s a Trap!



Only

Specific Request Sizes? OSs?

Solution 1

Set SO_RCVBUF Pops up again!

Solution 2

Set TCP_NODELAY

YES! but CPU Skyrockets!

Well understood bad interaction!

Symptom: Overly Long Round-Trip Time for a Request + Response

Challenges with TCP

Nagle

"Don't send 'small' segments if un-acknowledged segments exist"

+

Delayed ACKs

Don't acknowledge data immediately. Wait a small period of time (200 ms) for responses to be generated and piggyback the response with the acknowledgement

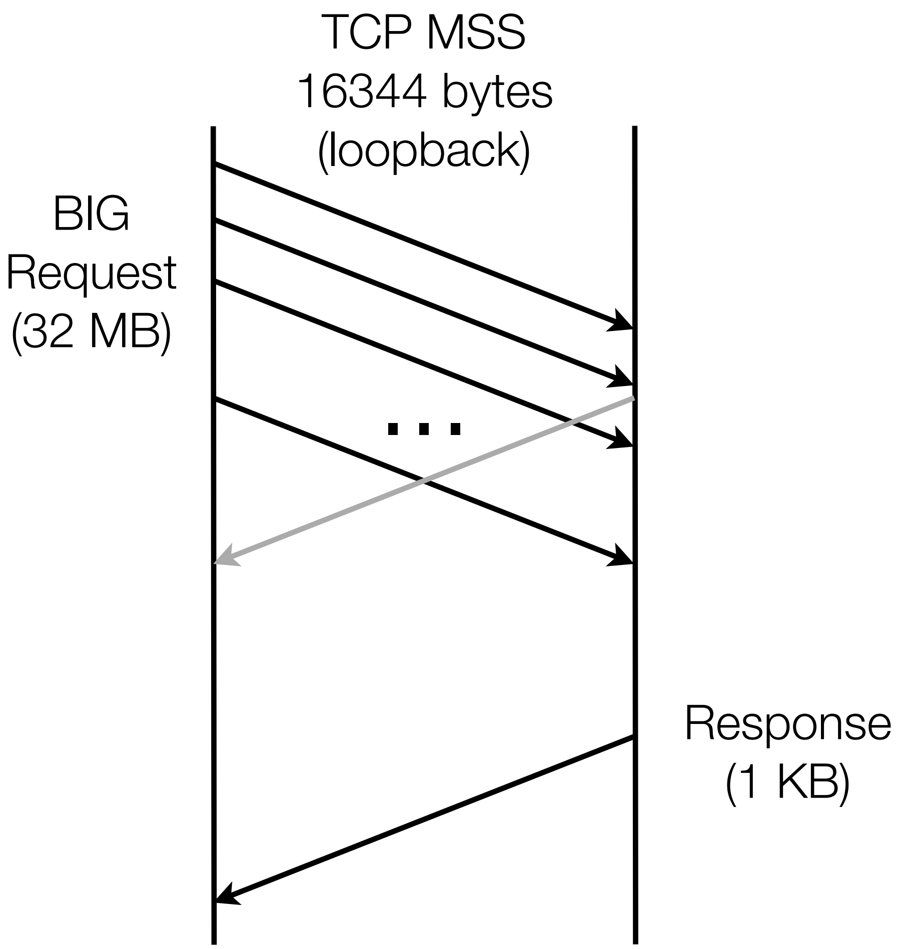
=

Temporary Deadlock

Waiting on an acknowledgement to send any waiting small segment. But acknowledgement is delayed waiting on more data or a *timeout*

Solutions?

Little Experiment



What about `sendfile(2)` and `FileChannel.transferTo()`?

Chunk Size	*RTT (msec)*
1500	32
4096	16
8192	12

Dramatically Higher CPU

Take Away(s)

"Small" messages are evil?

Chunks smaller than MSS are evil?

... *no, or not quite* ...

OS pagesize (4096 bytes) matters!

Why?

Kernel boundary crossings matter!

Question: Does the size of a send matter that much?

Challenges with UDP

Not Reliable

Loss recovery is apps responsibility

Not a Stream

Message boundaries matter!
(kernel boundary crossings)

No Flow Control

Potential to overrun a receiver

No Nagle

Small messages not batched

Causes of Loss

- Receiver buffer overrun
- Network congestion

(neither are strictly the apps fault)

No Congestion Control

Potential impact to *all* competing traffic!!
(unconstrained flow)

Network Utilization & Datagrams

$$\frac{\text{Data}}{\text{Data + Control}}$$

"The percentage of traffic that is data"

Batching?

No. of 200 Byte App Messages	*Utilization (%)*
1	87.7
5	97.3
20	99.3

Plus Fewer interrupts!

** IP Header = 20 bytes, UDP Header = 8 bytes, no response*

Application-Level Batching?

Application Specific Knowledge + Performance Limitations & Tradeoffs = Batching by the Application

Applications *sometimes* know when to send small and when to batch

Nagle, Delayed ACKs, Chunk Sizes, UDP Network Util, etc.

Applications can optimize and make the tradeoffs necessary at the time they are needed

* HTTP (headers + body), etc.

Addressing

- Request/Response idiosyncrasies
- Send-side optimizations

Batching `setsockopt()`s

TCP_CORK

- Linux only
- Only send when MSS full, when unCORKed, or ...
- ... after 200 msec
- unCORKing requires kernel boundary crossing
- Intended to work with TCP_NODELAY

When to Batch?

When to Flush?

TCP_NOPUSH

- BSD (some) only
- Only send when SO_SNDBUF full
- Mostly broken on Darwin

Flush? Batch?

Batch when...

1. Application logic
2. More data is *likely* to follow
3. *Unlikely* to get data out before next one

Flush when...

1. Application logic
2. More data is *unlikely* to follow
3. Timeout (200 msec?)
4. *Likely* to get data out before next one

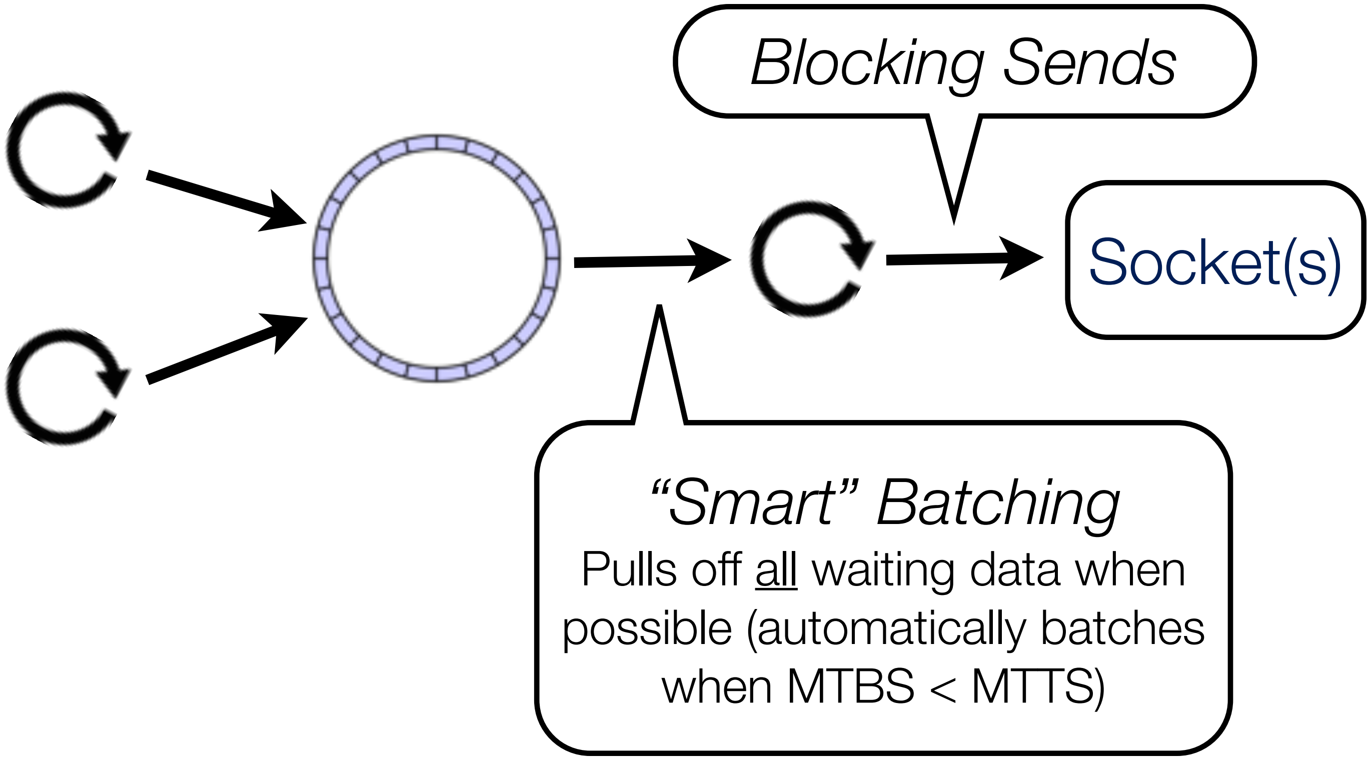
An Automatic Transmission for Batching

1. Always default to flushing
2. Batch when Mean time *between* sends < Mean time *to* send (EWMA?)
3. Flush on timeout as safety measure

Question: *Can you batch too much?* ***YES!***

Large UDP (fragmentation) +
non-trivial loss probability

A Batching Architecture



Blocking Sends

Socket(s)

"Smart" Batching

Pulls off all waiting data when possible (automatically batches when MTBS < MTTS)

MTBS: Mean Time Between Sends

MTTS: Mean Time To Send (on socket)

Advantages

- Non-contended send threads
- Decoupled API and socket sends
- Single writer principle for sockets
- Built-in back pressure (bounded ring buffer)
- Easy to add (async) send notification
- Easy to add rate limiting

Can be re-used for other batching tasks (like file I/O, DB writes, and pipeline requests)!

Multi-Message Send/Receive

`sendmmsg(2)`

- Linux 3.x only
- Send *multiple* datagrams with single call
- Fits nicely with batching architecture

Compliments gather send (`sendmsg, writev`) - which you can do in the same call!

`recvmmsg(2)`

- Linux 3.x only
- Receive *multiple* datagrams with single call
- So, so, sooo SMOKIN' HOT!

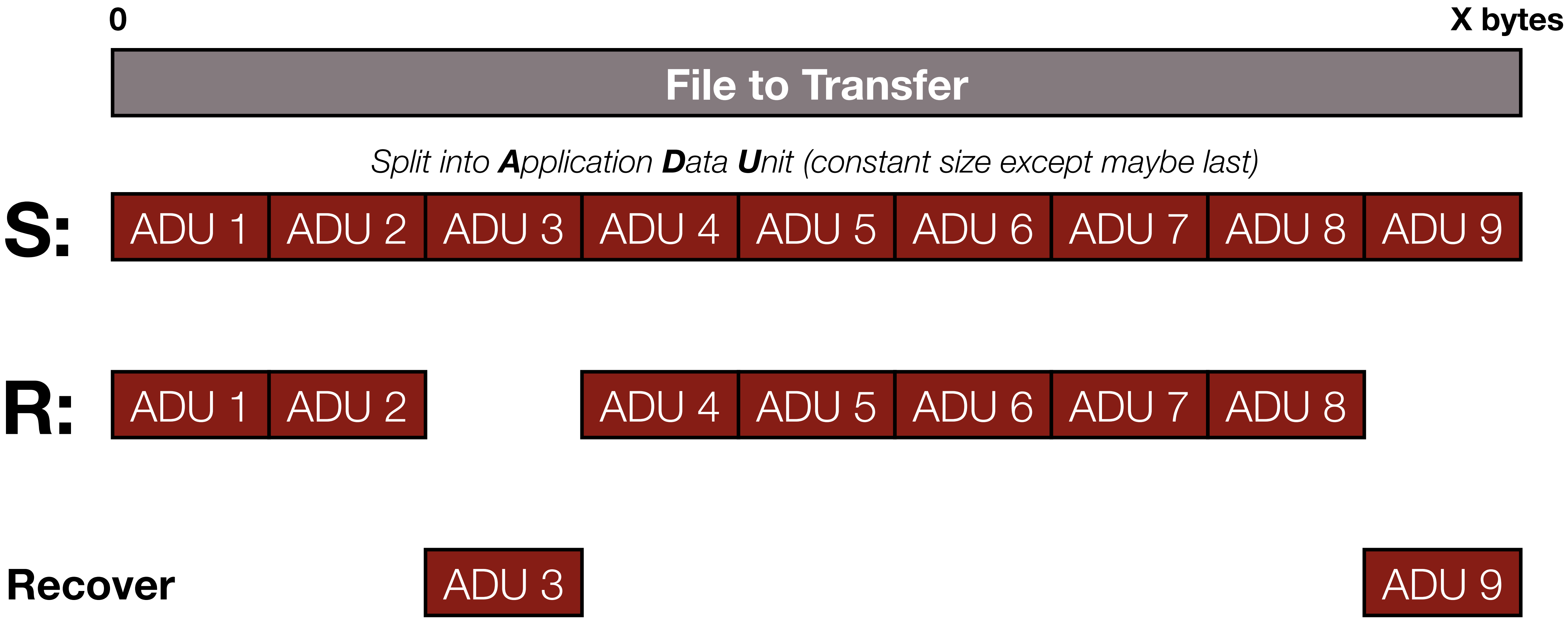
Scatter recv (`recvmsg, readv`) is usually not worth the trouble

Advantages

- Reduced kernel boundary crossings

Domain: Protocol Design

Application-Level Framing



0

X bytes

File to Transfer

Split into ***A****pplication* ***D****ata* ***U****nit (constant size except maybe last)*

S: ADU 1 ADU 2 ADU 3 ADU 4 ADU 5 ADU 6 ADU 7 ADU 8 ADU 9

R: ADU 1 ADU 2 ADU 4 ADU 5 ADU 6 ADU 7 ADU 8

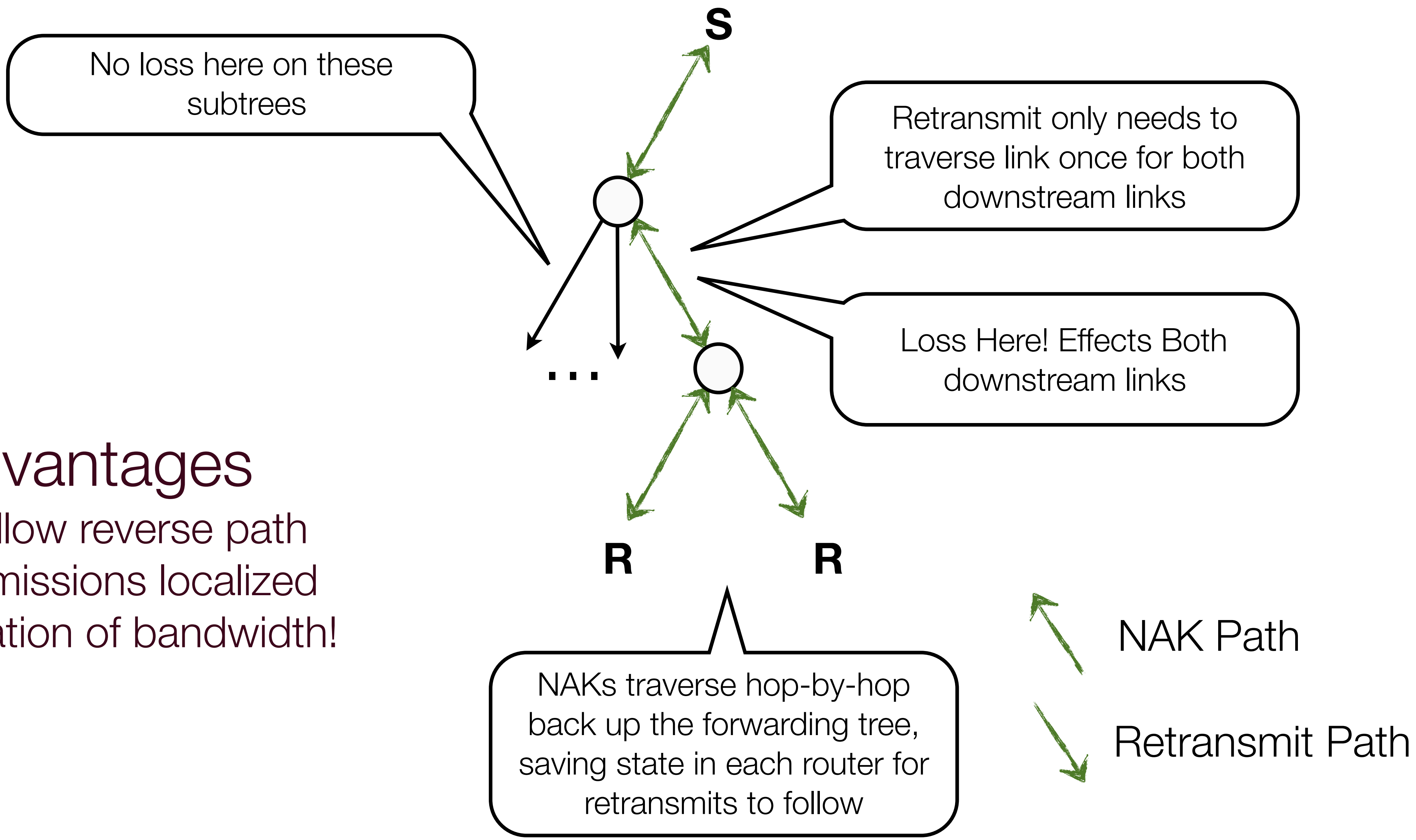
Recover ADU 3 ADU 9

Advantages

- Optimize recovery until end (or checkpoints)
- Works well with multicast and unicast
- Works best over UDP (message boundaries)

Clark and Tennenhouse, ACM SIGCOMM CCR, VOlume 20, Issue 4, Sept. 1990

PGM Router Assist



No loss here on these subtrees

Retransmit only needs to traverse link once for both downstream links

Loss Here! Effects Both downstream links

NAKs traverse hop-by-hop back up the forwarding tree, saving state in each router for retransmits to follow

Advantages

- NAKs follow reverse path
- Retransmissions localized
- Optimization of bandwidth!

Pragmatic General Multicast (PGM), IETF RFC 3208

Questions?