Partially-Reliable HTTP

Partially-Reliable HTTP

a.k.a. scatter-gather HTTP

Partially-Reliable HTTP

a.k.a. scatter-gather HTTP
+ cancellation

Why?

Head-of-line (HoL) Blocking

Head-of-line (HoL)

Blocking hurts.

- HoL blocking affects HTTP/1
- HoL blocking affects HTTP/2

- HoL blocking affects HTTP/1
- HoL blocking affects HTTP/2
- HoL blocking affects QUIC/h3
- HoL blocking affects QUIC/QPACK

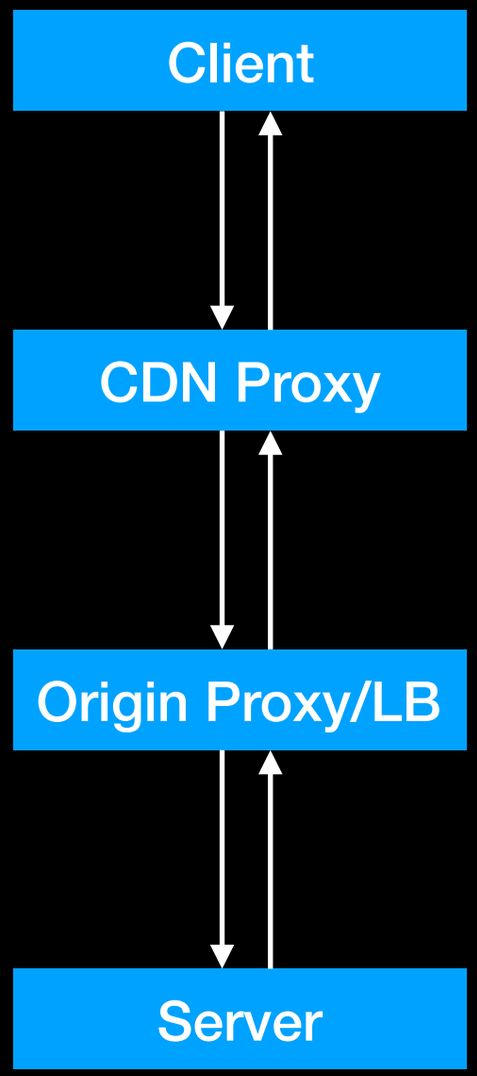
Wait, wut?!

Wait, wut?!

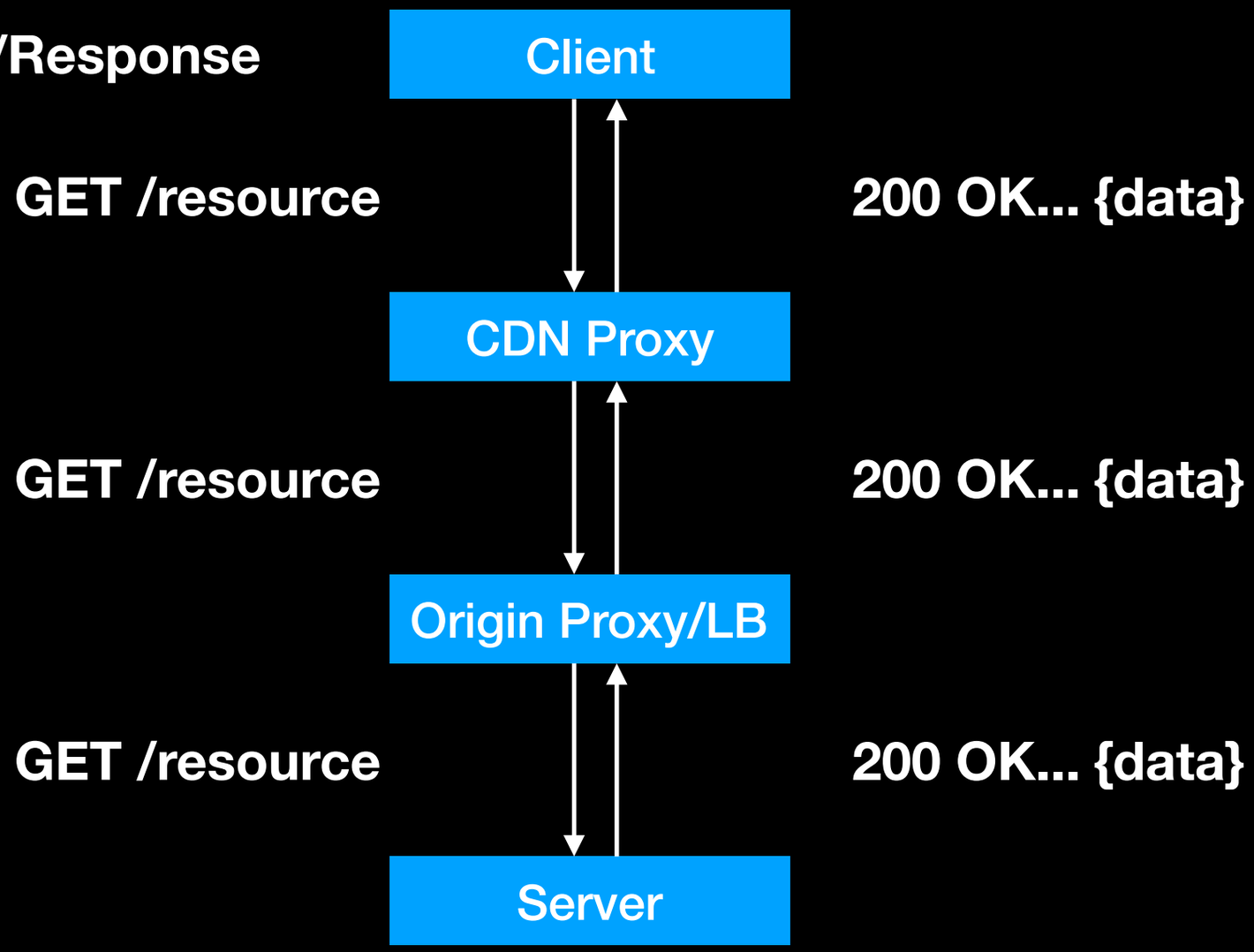
QUIC doesn't have HoL blocking!

Wrong, Padawan!

A typical deployment



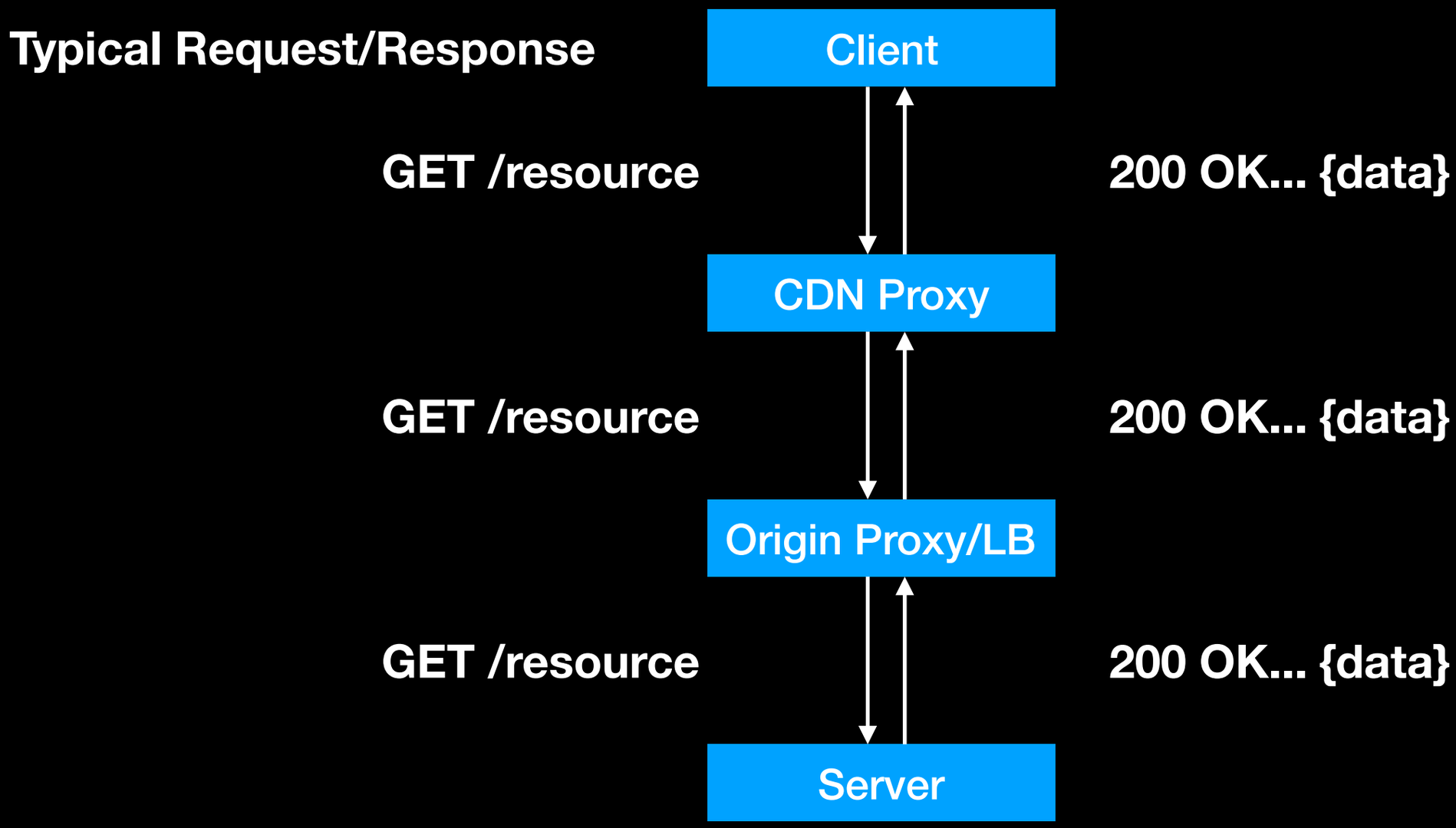
Typical Request/Response



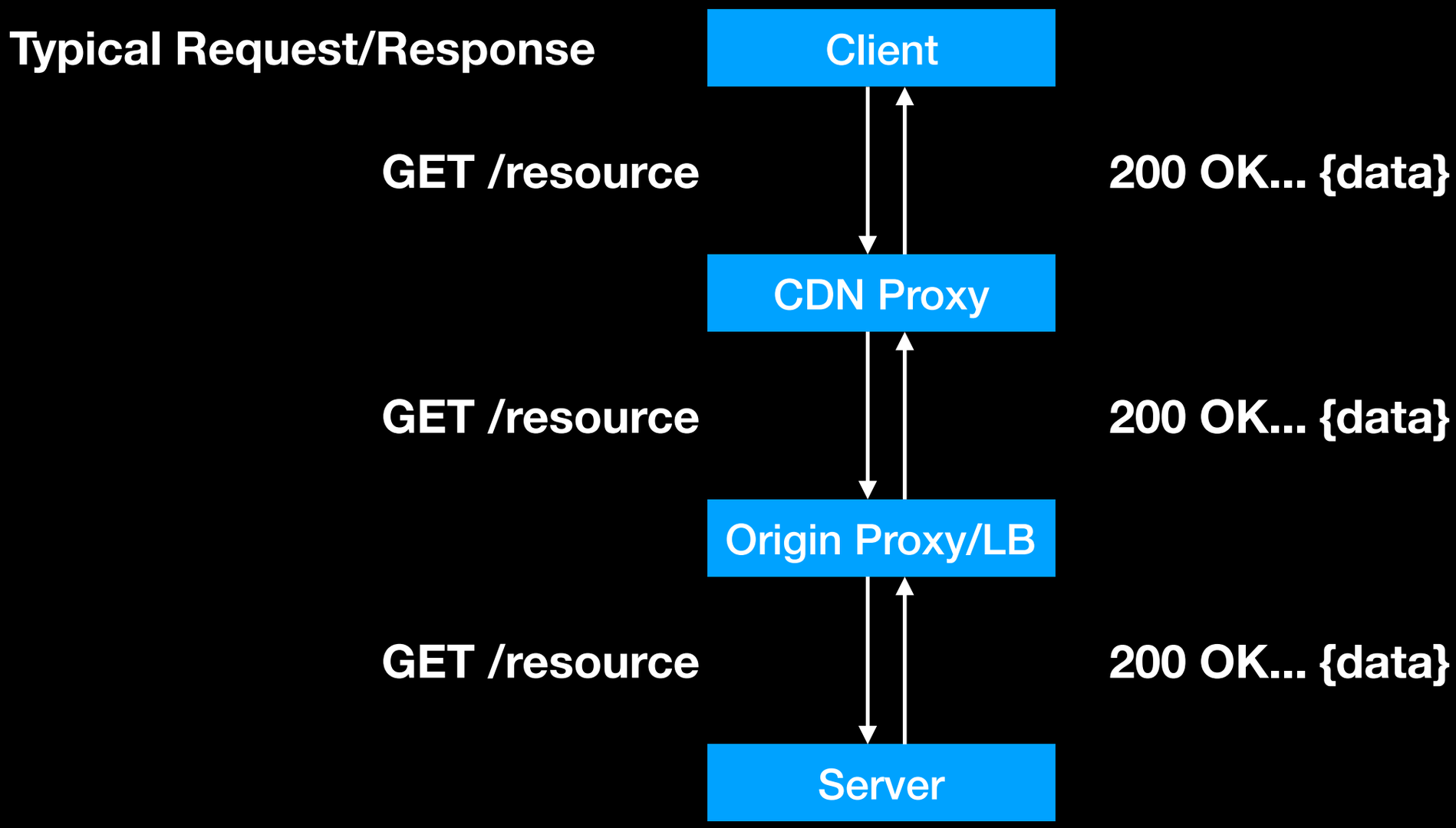
Wow! Amazing!

So smooth!

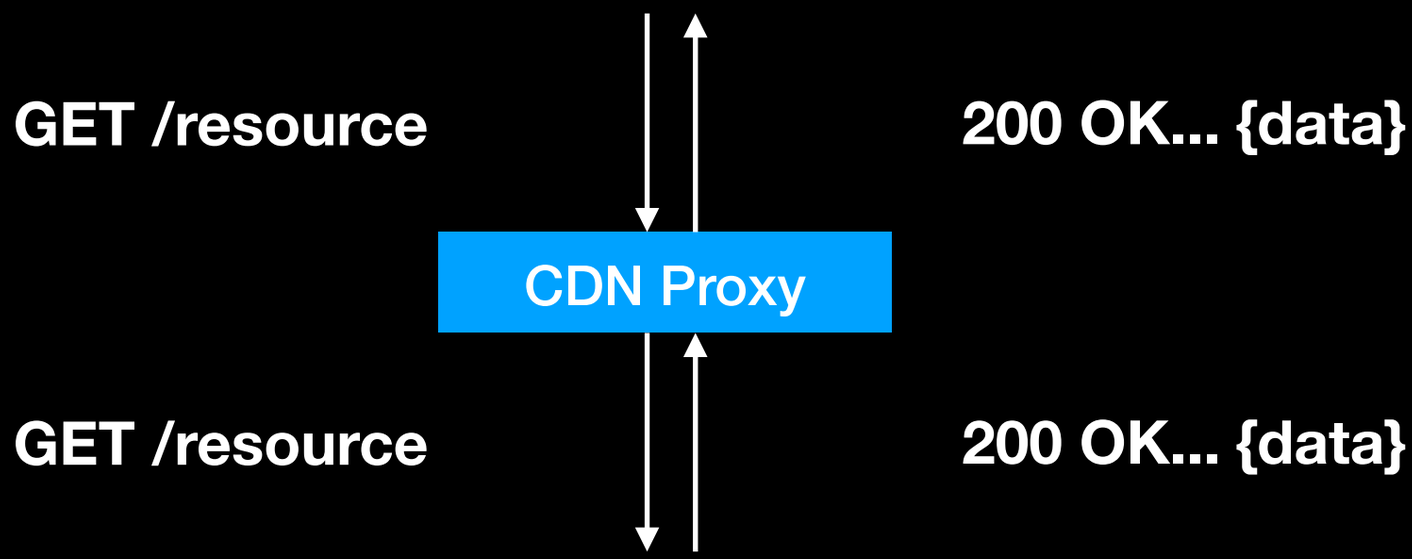
That was a lie.



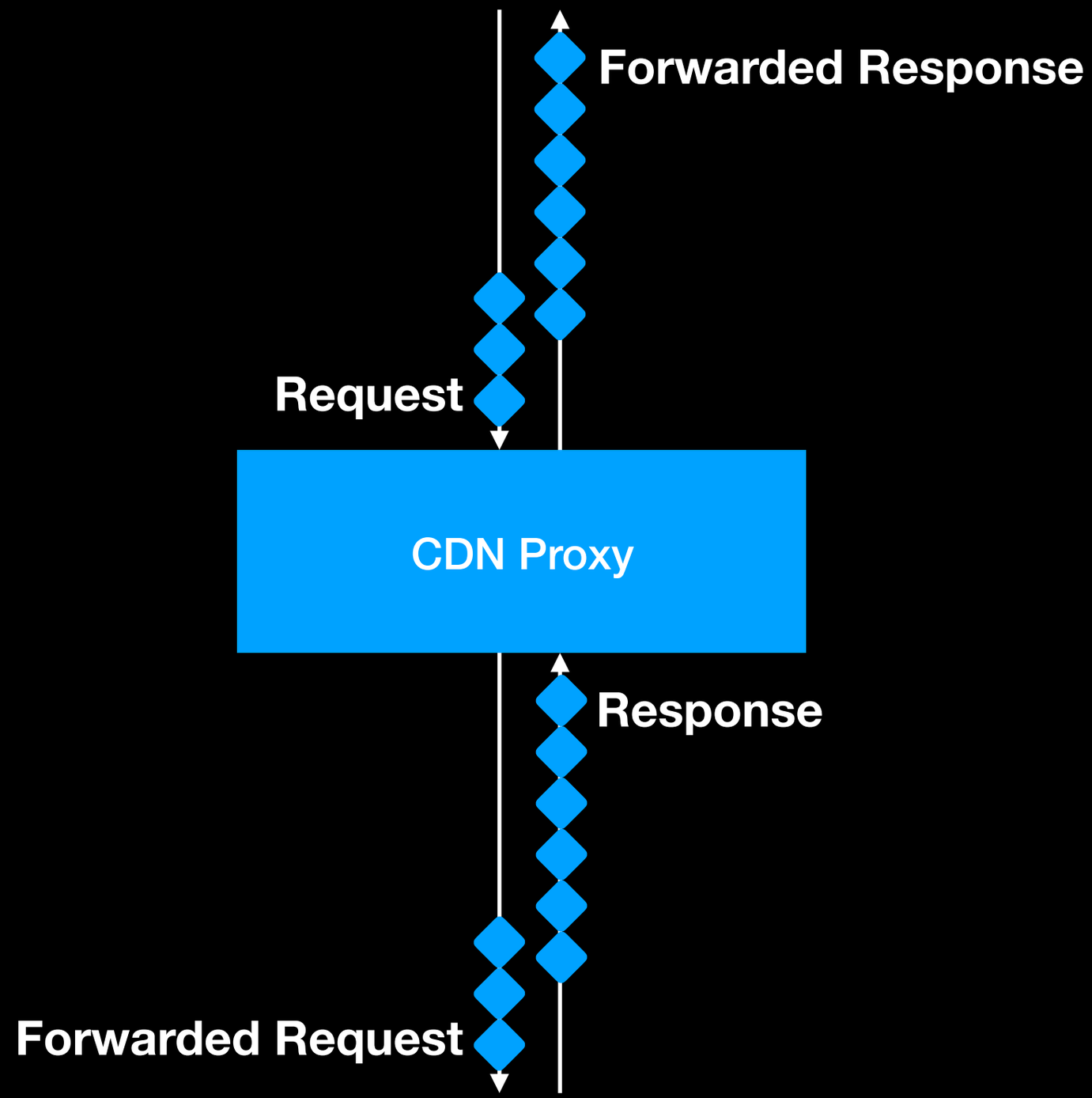
Reality is a bit *dirtier*

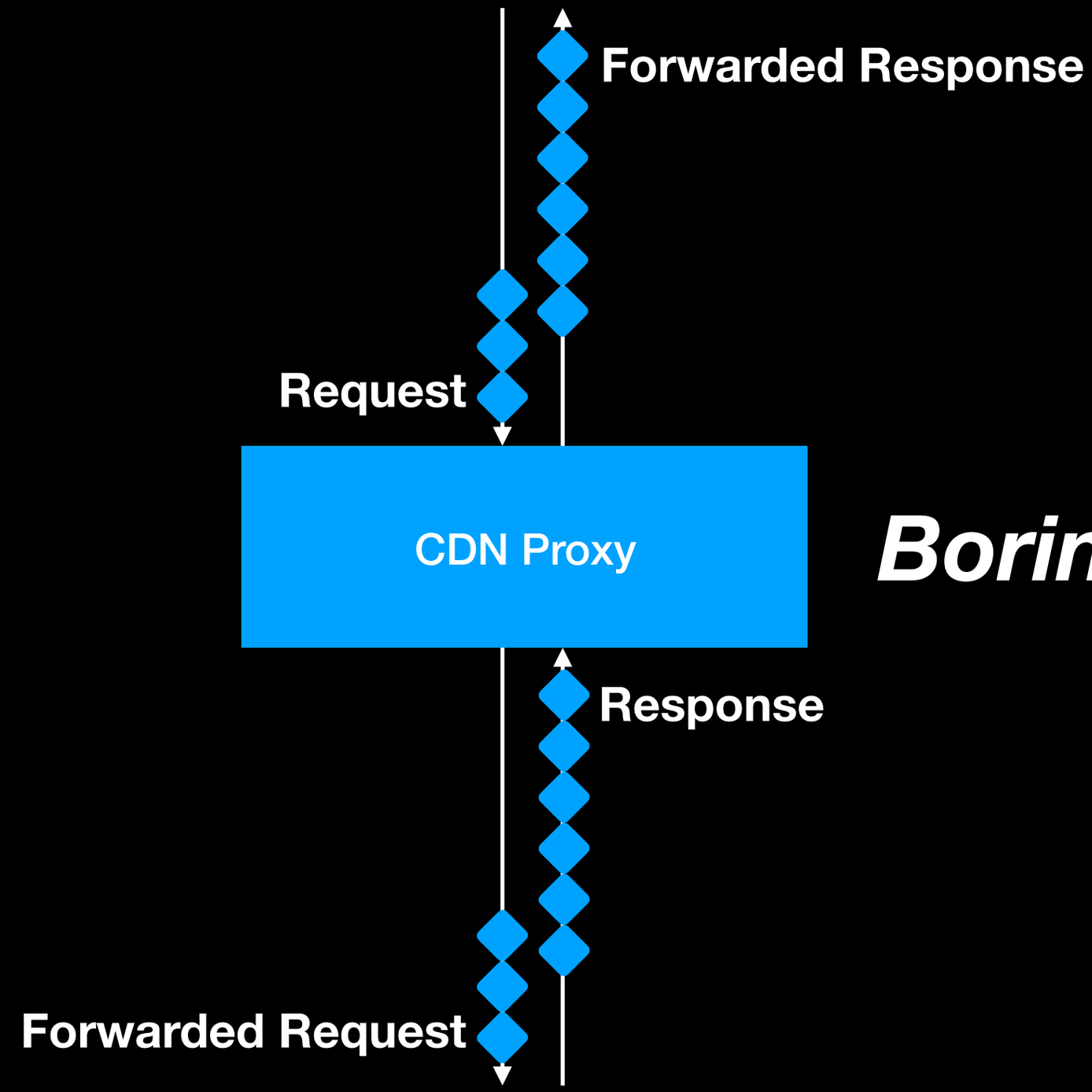


Lets look at the CDN proxy



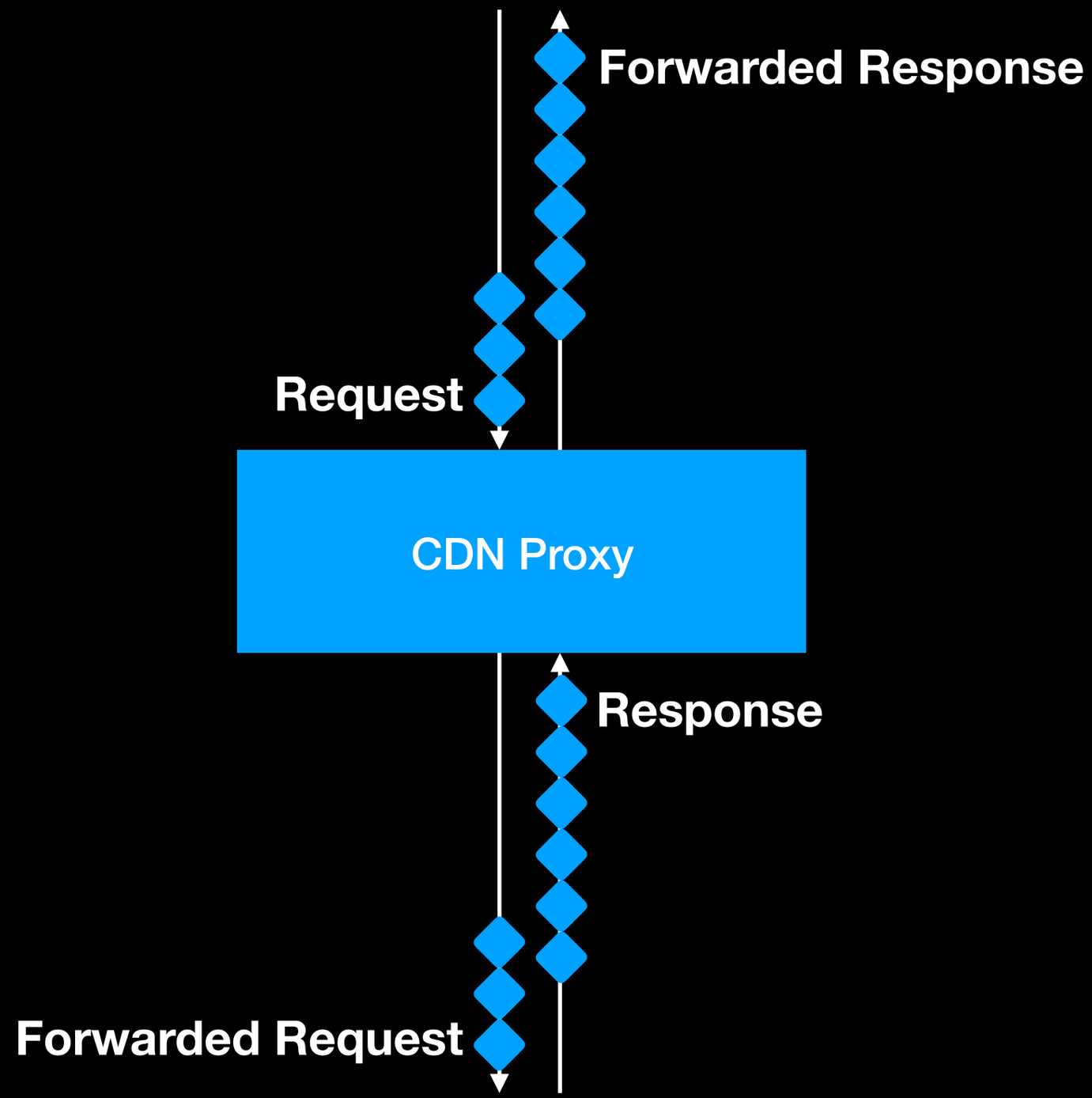
Closer!


Forwarded Response
Request
CDN Proxy
Response
Forwarded Request

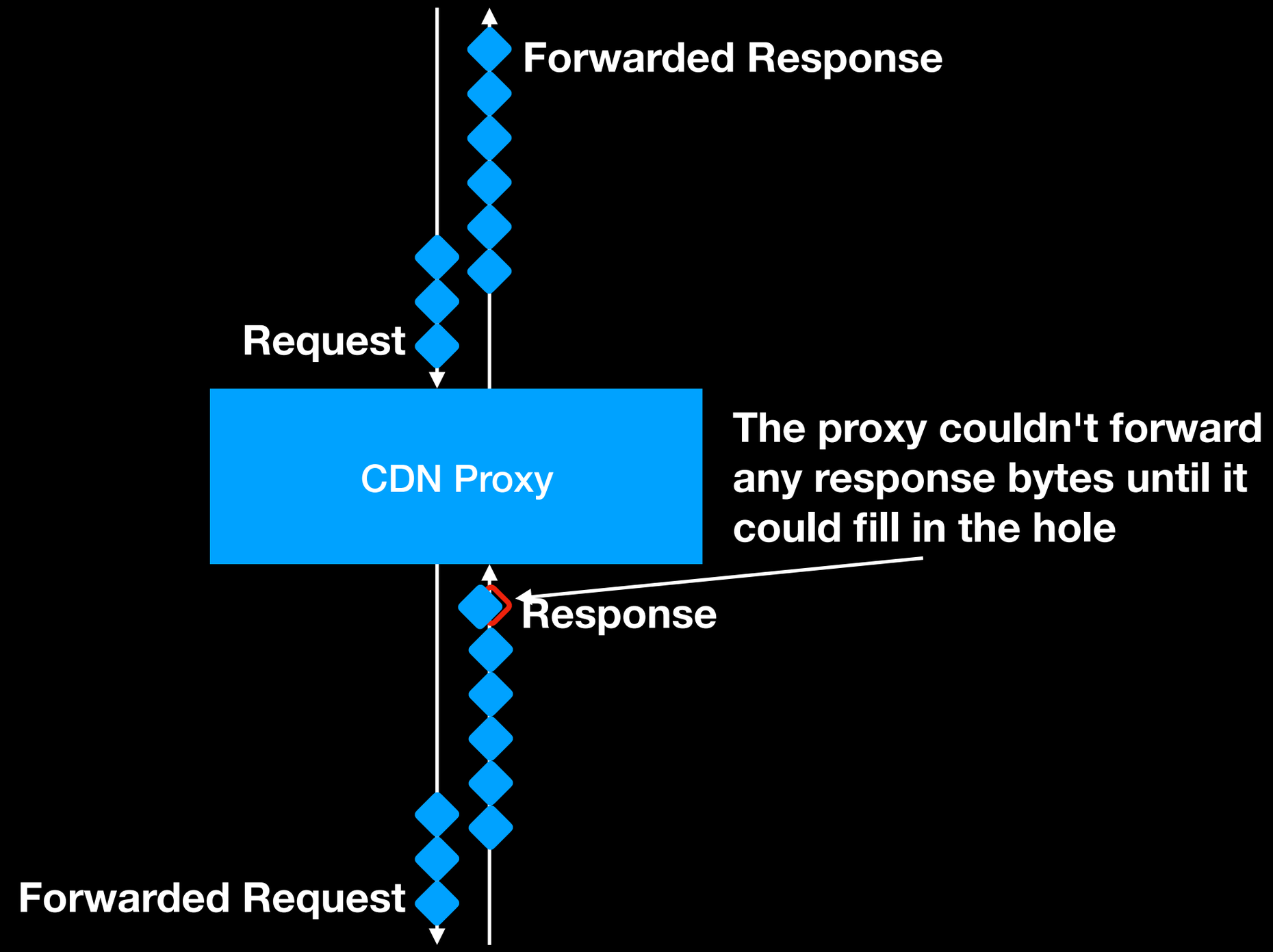


Boring, right?

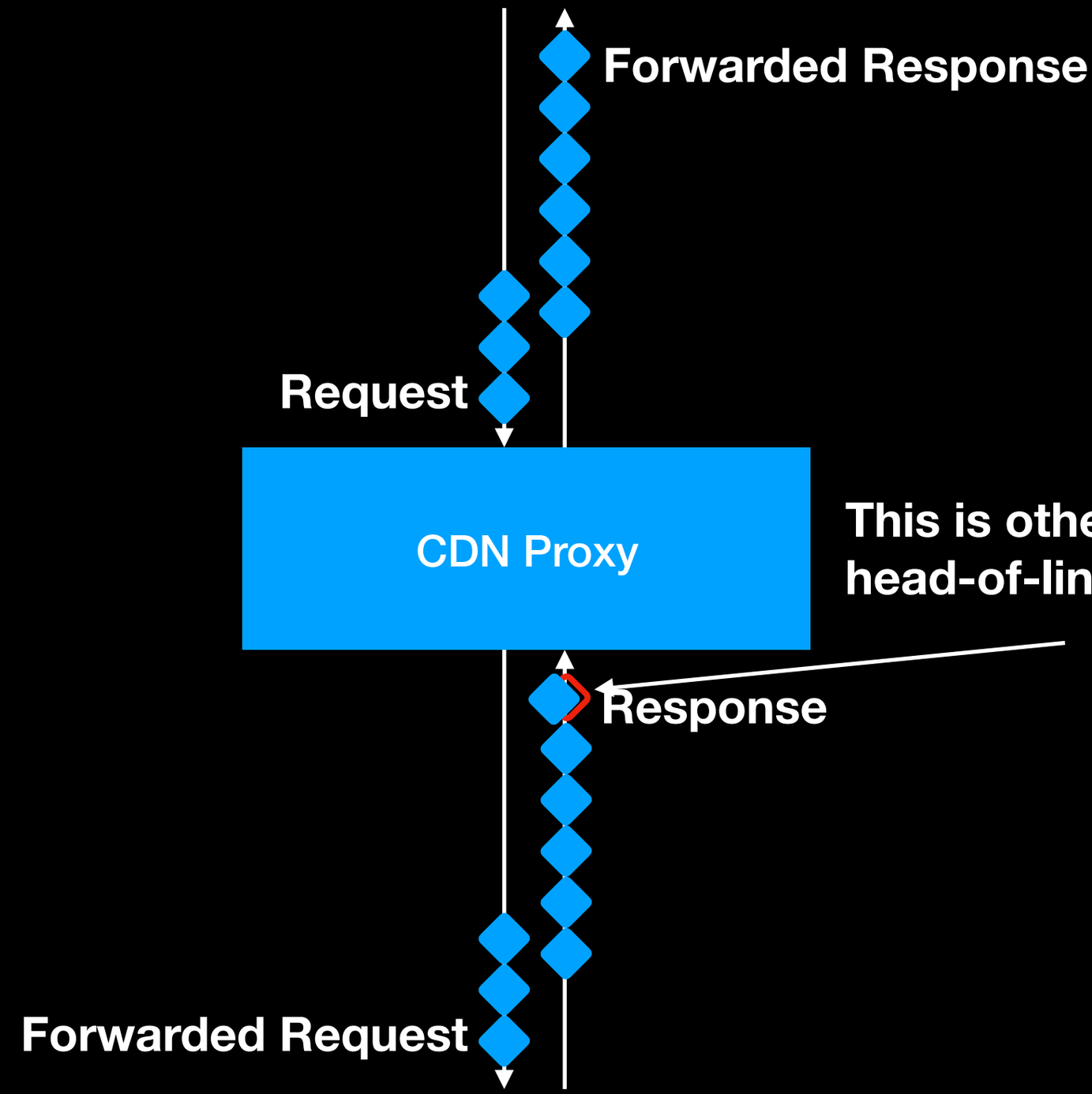
Lets add some packet loss



Lets add some packet loss

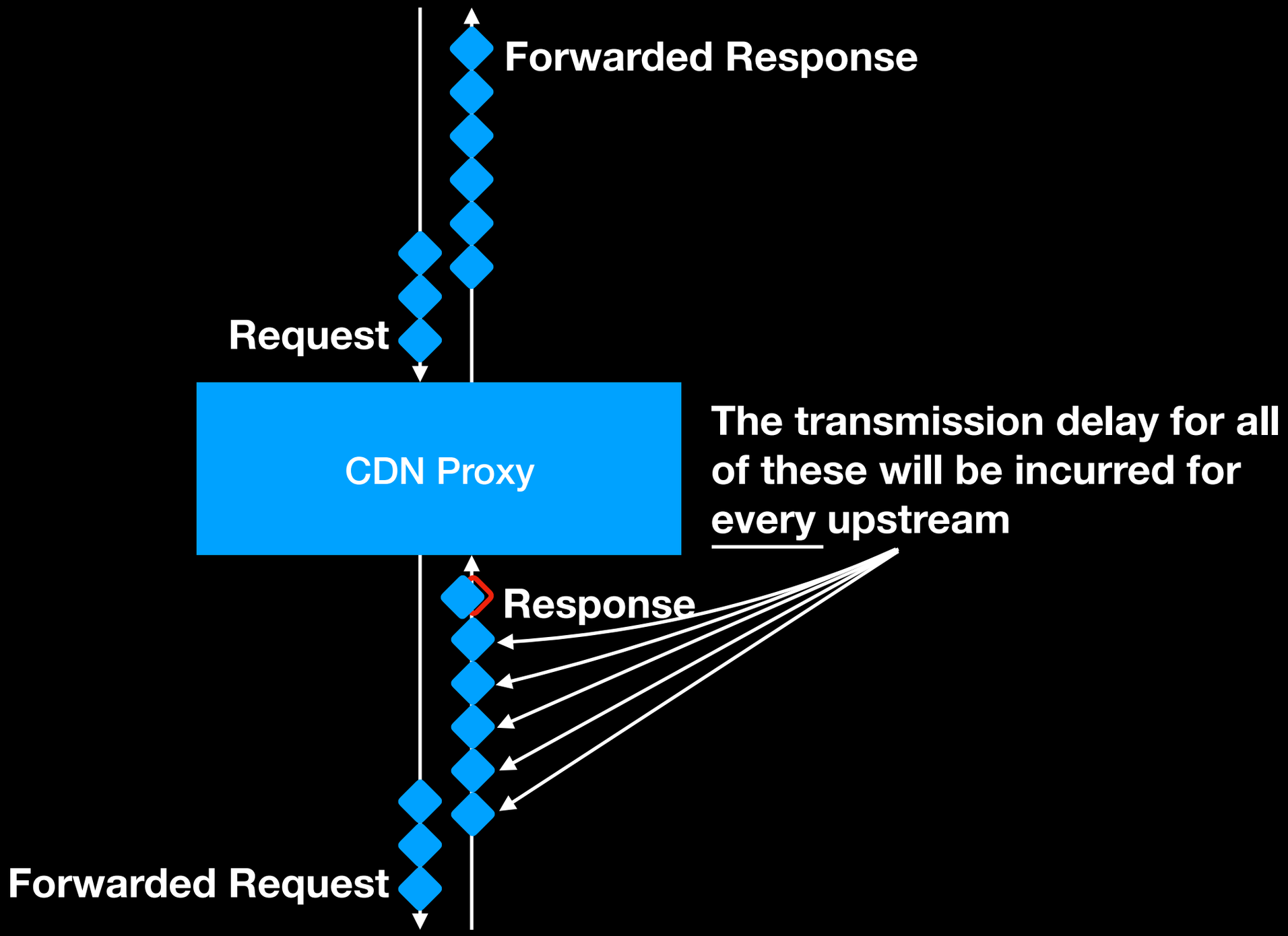


Lets add some packet loss



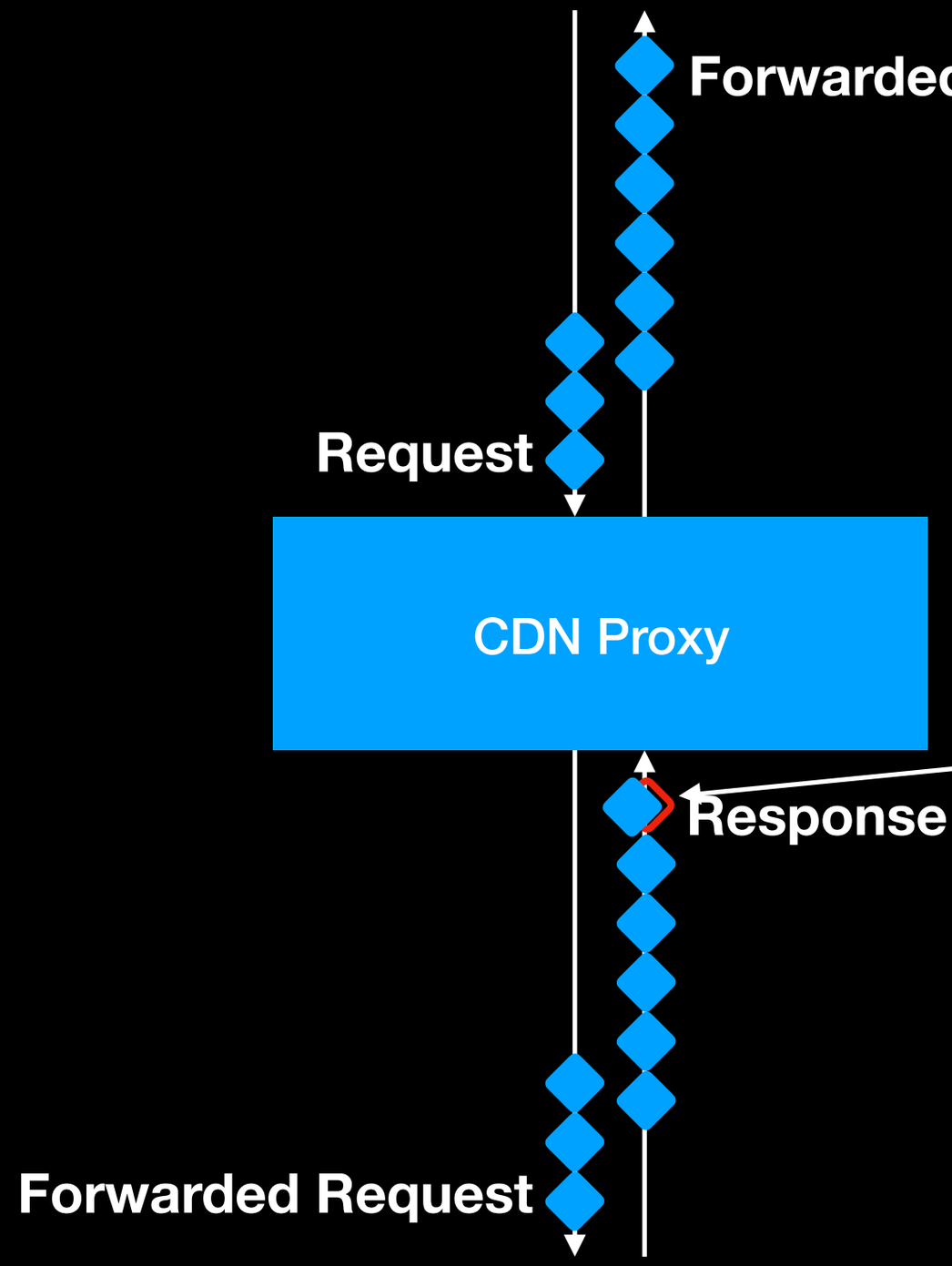
This is otherwise known as head-of-line (HoL) blocking

Lets add some packet loss



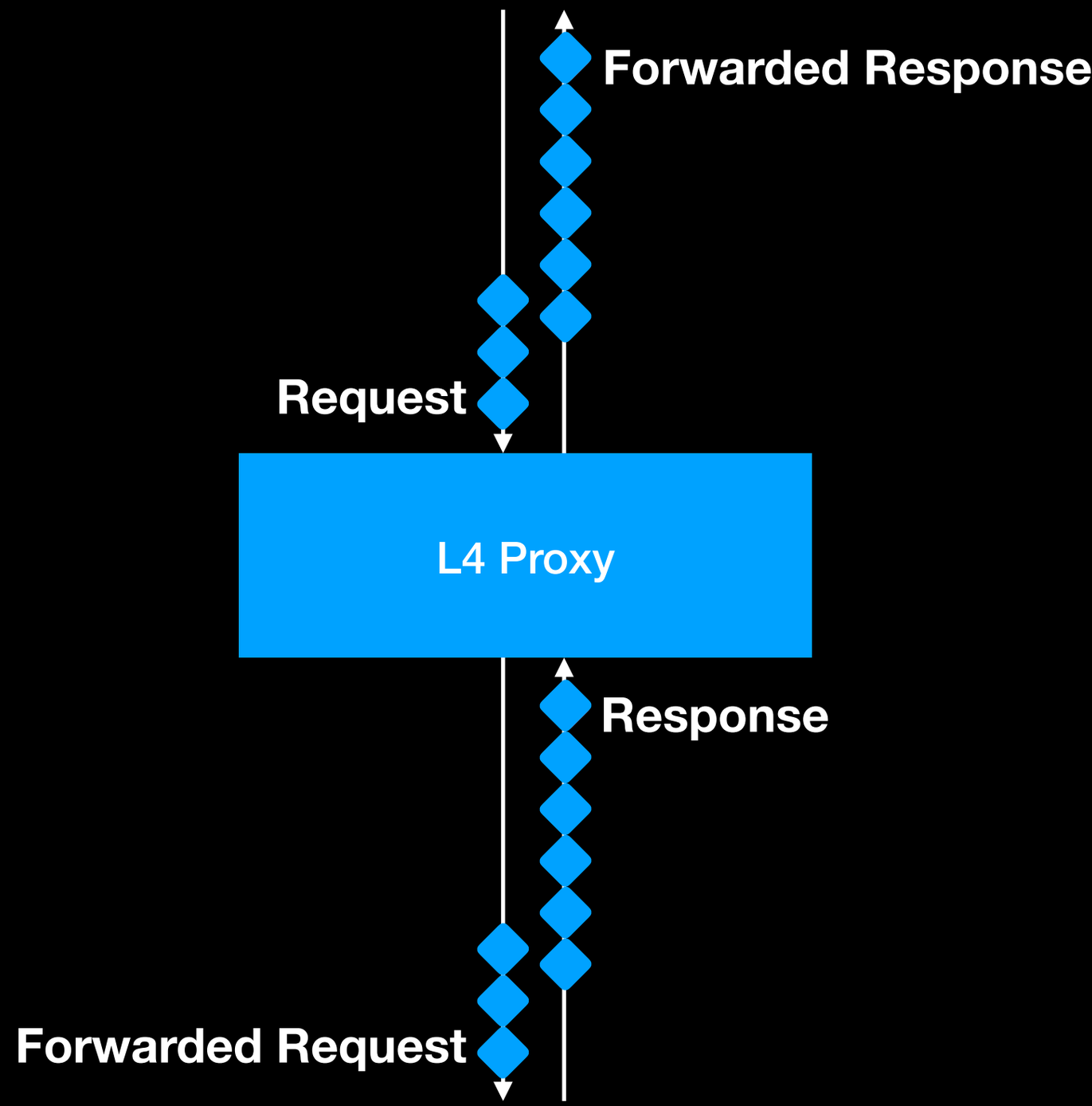
The transmission delay for all of these will be incurred for every upstream

Lets add some packet loss

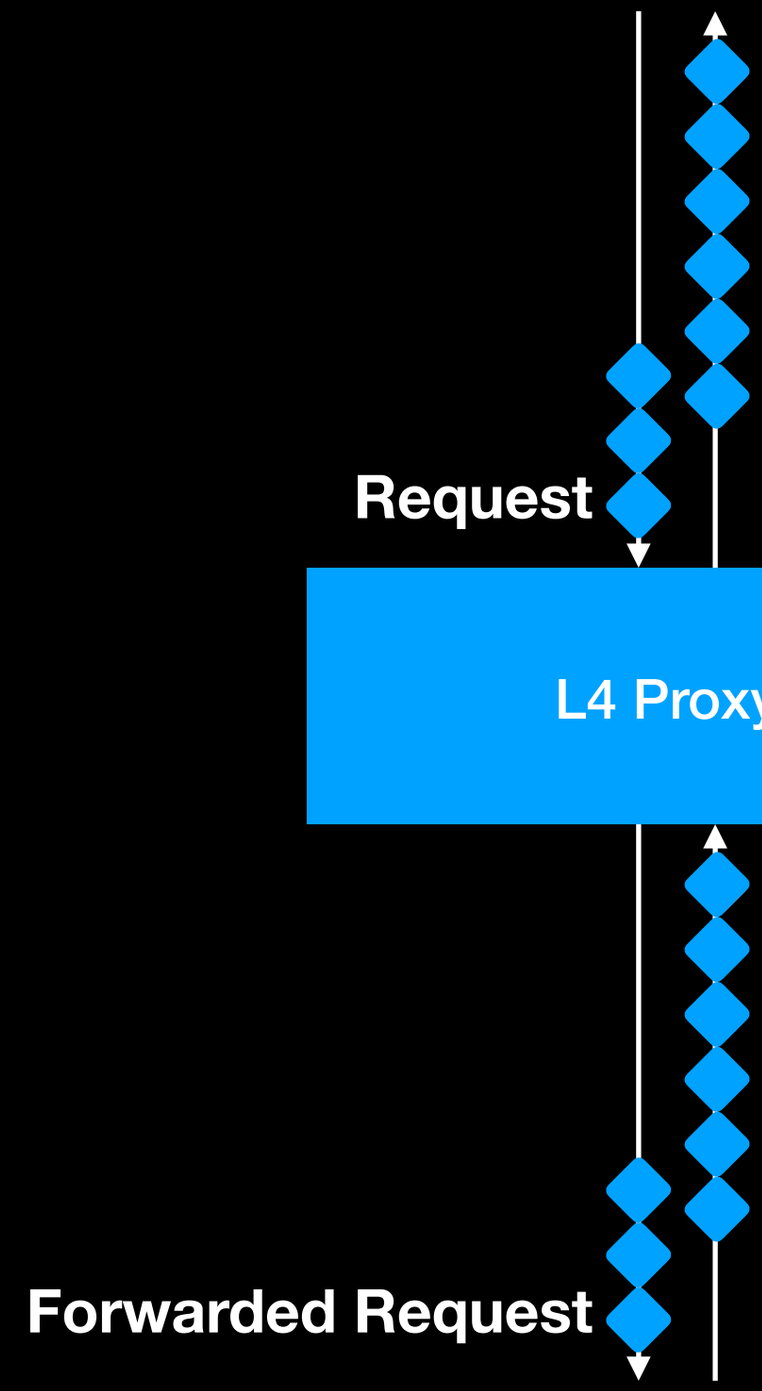


This will end up impacting the transport as well-- It'll likely be forwarding a burst instead of a paced stream.

If this was a 'dumb' L4 proxy instead..

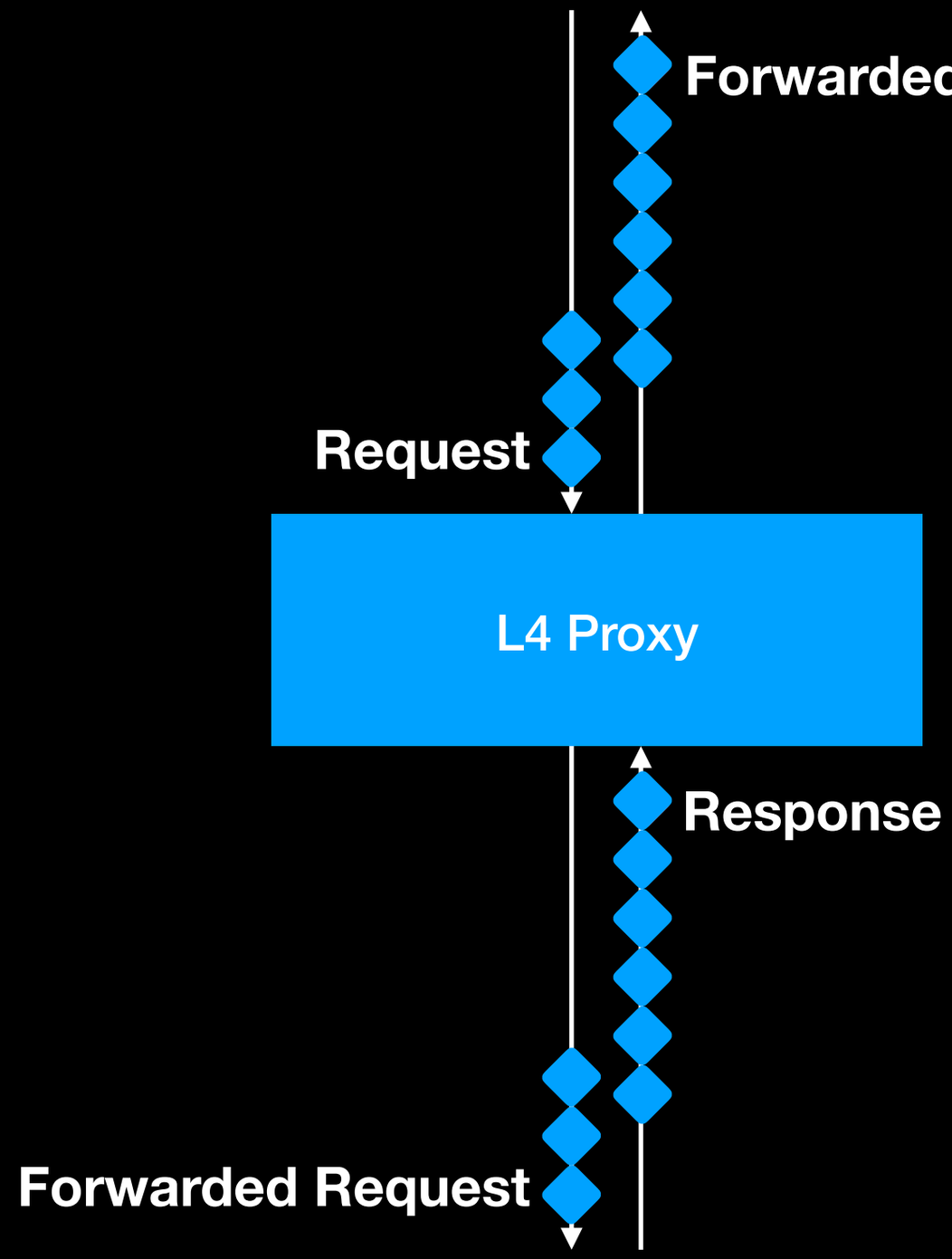


If this was a 'dumb' L4 proxy instead..



The proxy forwards packets as they arrive, thus
no HoL blocking for xmission.

If this was a 'dumb' L4 proxy instead..



The proxy forwards packets as they arrive, thus
fewer bursts at the transport!

The "problem" isn't the "transport".

The "problem" is
the semantics of the HTTP layer

The "problem" is
the semantics of the HTTP layer
as understood by the proxies.

The problem

HTTP servers, clients, and proxies only understand in-order delivery of the requested bytes.

The problem

in-order delivery

The problem

in-order delivery
means
head-of-line blocking.

The solution

not in-order delivery
means
not head-of-line blocking.

The solution

Instead of:

```
auto resp = request(url).readResponse();
doSomething(resp);
```

do:

```
auto handler=[](Offset o, Size s) {
 /* do something with bytes */
};
request(url).readResponseBytes(handler);
```

The solution

Instead of:

```
auto body = genBody();
request(url).writeBody(body);
```

do:

```
auto bytes = genBodyBytes();
request(url).writeBytes(bytes, offset);
```

The solution

In plain language:

Instead of:

Reading/writing a whole body

Do:

Read ranges (offset+size) of the body, and write ranges of the body.

The solution

In plain language:

Scatter-gather.

The solution

But wait!

Now with *free* data-loss*!*

Why is data-loss a 'solution'?

It isn't *unless* it is **intentional** data-loss!

For the purpose of this conversation

intentional data-loss

is

Partial Reliability

Data-loss as 'solution'

Not all data is equal.

Take Videoconferencing (or a phone call) as an example

Videoconferencing prefers low-jitter

In a videoconference:

- the latest information is the most valuable
- without the bias to immediacy, one ends up with stalls and latency increases.

Videoconferencing prefers low-jitter

People higher-level "applic evel"
r sions, but can't ran ging
late

Videoconferencing prefers low-jitter

I'm sorry, what was that?

I missed it.

Can you repeat it?

Videoconferencing prefers low-jitter

People higher-level "applic evel"
r sions, but can't ran ging
late

Videoconferencing prefers low-jitter

People can do higher-level "application-level" retransmissions, but can't abide by randomly changing latency.

Videoconferencing prefers low-jitter

Randomly changing latency == jitter.

Videoconferencing prefers low-jitter

Try having a conversation where you can't anticipate when your other end is supposed to have received and acknowledged.

Videoconferencing prefers low-jitter

That data from 3 seconds ago?

Kill it with *fire*. It might prevent getting the data you need *now*.

With "old-school" HTTP, you can't (efficiently) drop data--
You can only cancel a whole request/response/connection.

To drop only part of a response/request, you need to be able to receive data "out of order" and "partially".

If you're building a scatter-gather API, the changes needed to make it partially-reliable are relatively minimal.

.. and since both have advantages at the application and transport layers ..

Partially-Reliable HTTP

Is thus simply the combination of:

- a scatter-gather API which allows the receipt of request/response bytes out of order

and

- the ability for an endpoint to signal that it either doesn't want some data, or won't be sending it.

Partially-Reliable HTTP

A reminder of the benefits:

Partially-Reliable HTTP

In the face of packet-loss at proxies it:

- *reduces HoL blocking* delays when forwarding HTTP entity-bodies.

In the face of delays/packet-loss on links/routes it:

- *allows endpoints to tradeoff data completeness for jitter*

Partially-Reliable HTTP

... and since it *is* HTTP, all of the caching, authority, and other semantics worked out over the last few decades should continue to work.

Partially-Reliable HTTP

Enables lower latencies in the "normal" HTTP usecase in the face of packet-loss.

Wrapping up!

Partially-Reliable HTTP

Enables low-jitter use cases for which HTTP was previously unsuitable

Partially-Reliable HTTP

Could scale delivery to billions of users

Partially-Reliable HTTP

Could scale delivery to billions of users with cache semantics that are well understood and the result of years of experience.

The solution

Partial Reliability +

Scatter-Gather

The solution

Partial Reliability +

Scatter-Gather

==

Partially Reliable HTTP

Partially Reliable HTTP

Thanks!