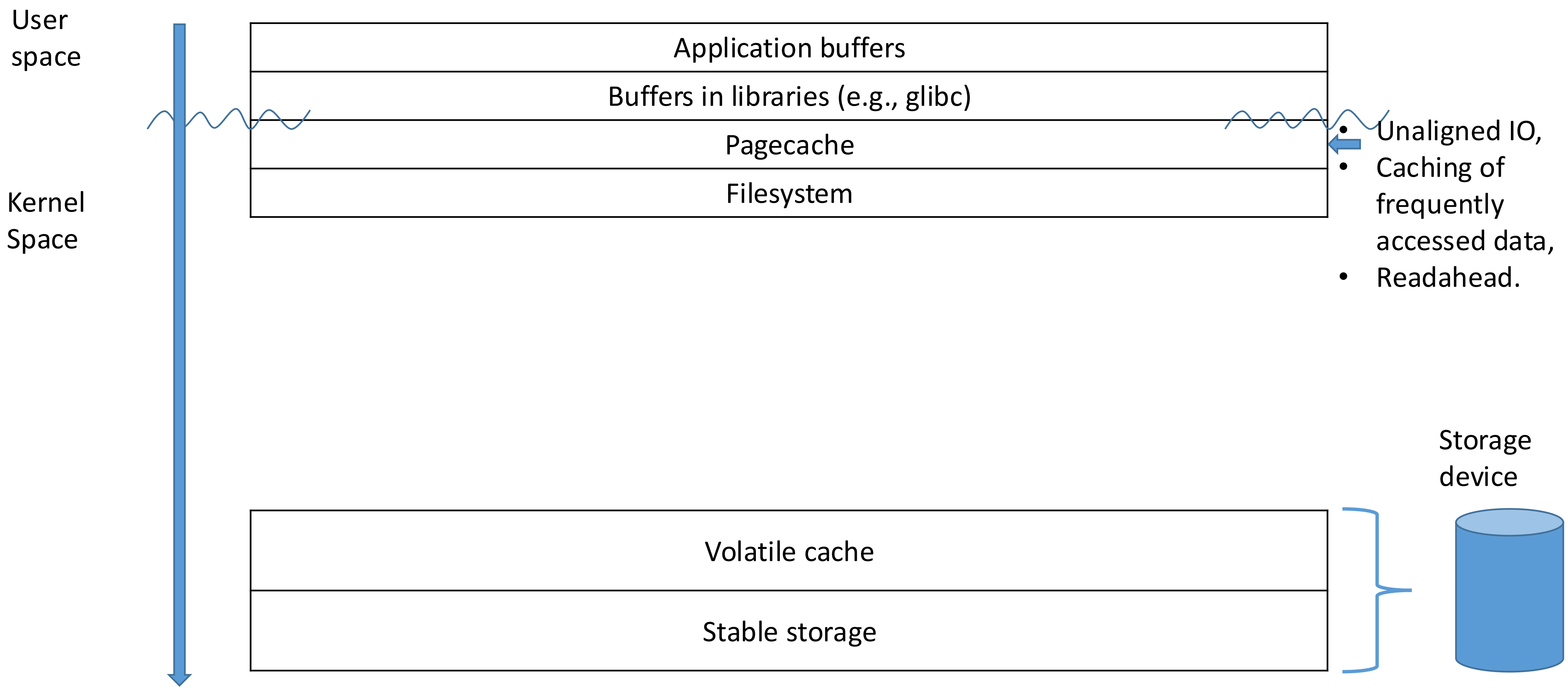
The basics of file systems

Neapolis University Pafos

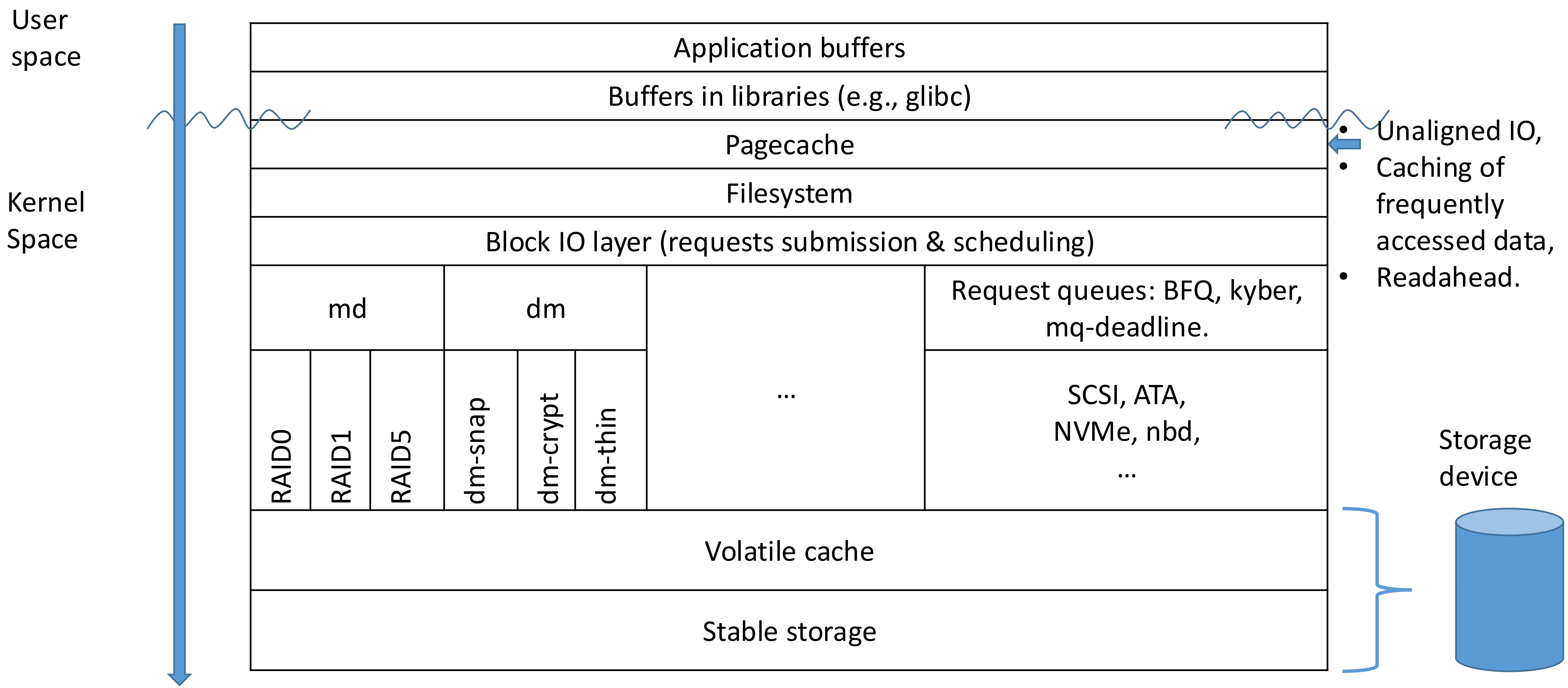
JET BRAINS

The path from application buffers to persistent storage (very approximate)



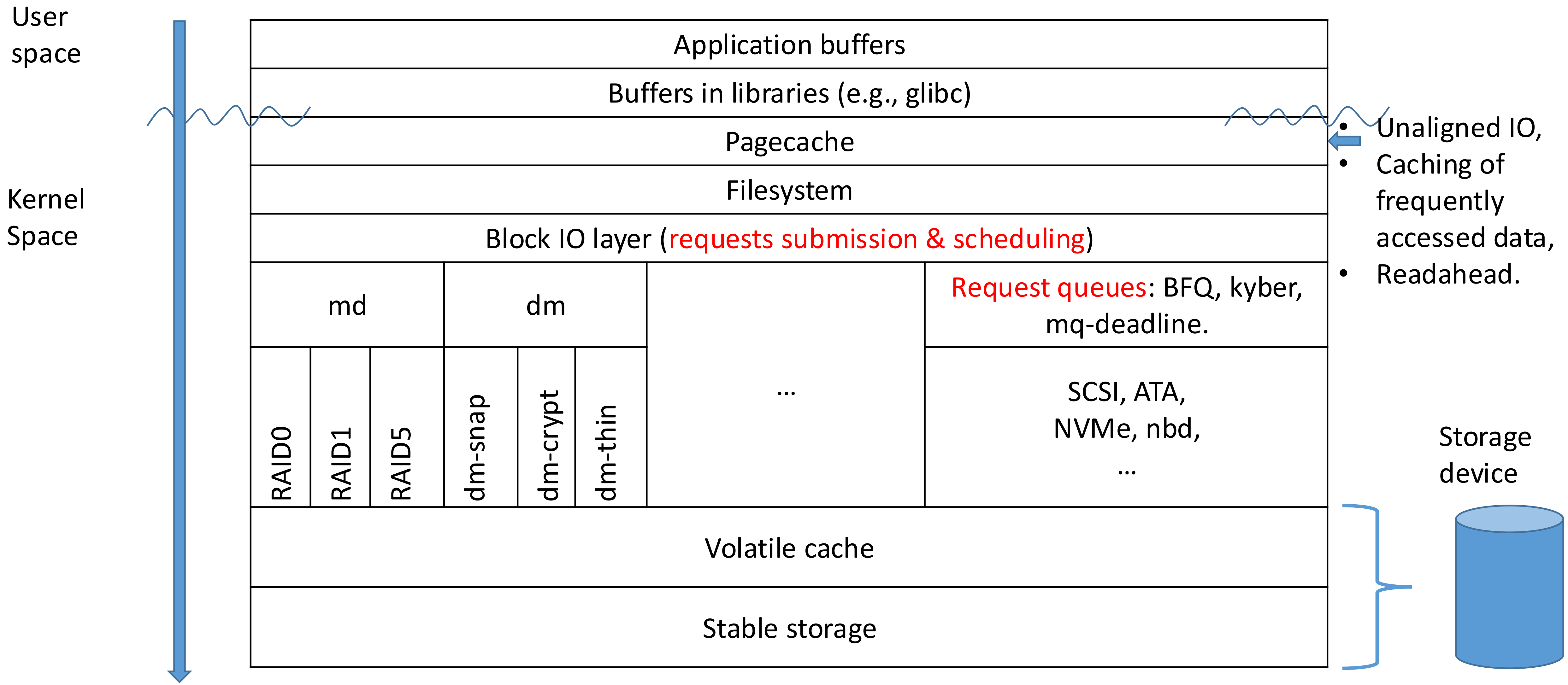
- Unaligned IO,
- Caching of frequently accessed data,
- Readahead.

The path from application buffers to persistent storage (very approximate)



https://lwn.net/Articles/736534

The path from application buffers to persistent storage (very approximate)



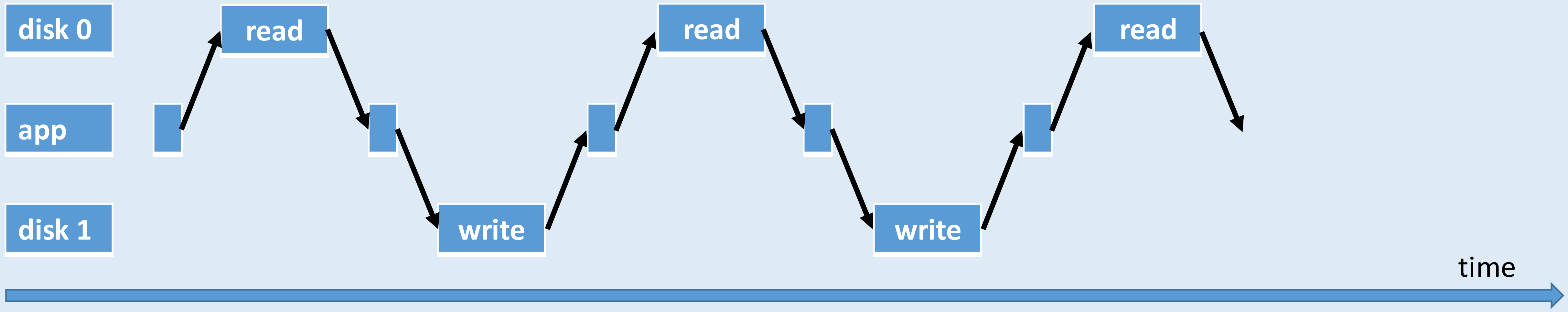
https://lwn.net/Articles/736534

Synchronous and asynchronous IO, pipelining and multiplexing

Consider a naïve implementation of a routine that copies a file from one disk to another:

```
while (!done) {
  r = read(fd_in, buf, sizeof(buf));
  r0 = write(fd_out, buf, r);
  ...
}
```

Let us draw time intervals when each disk is accessed:

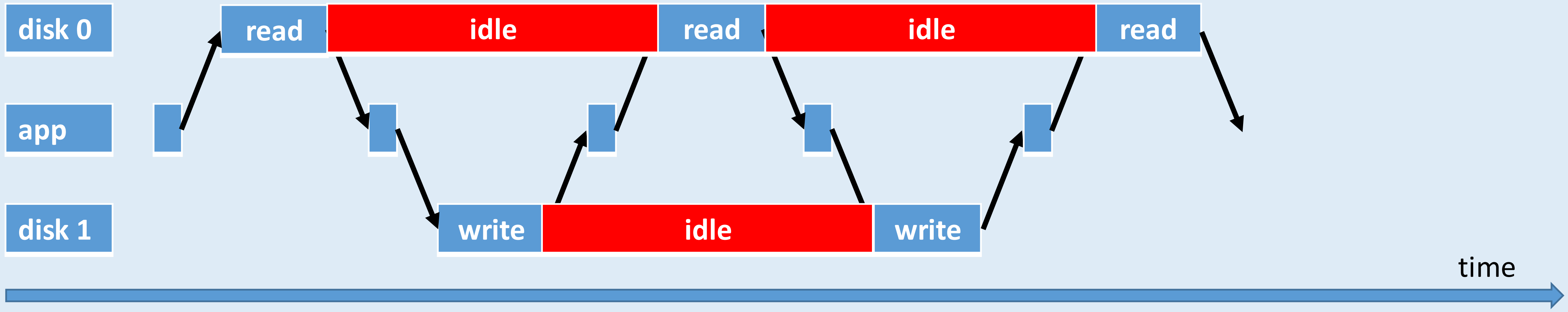


Synchronous and asynchronous IO, pipelining and multiplexing

Consider a naïve implementation of a routine that copies a file from one disk to another:

```
while (!done) {
  r = read(fd_in, buf, sizeof(buf));
  r0 = write(fd_out, buf, r);
  ...
}
```

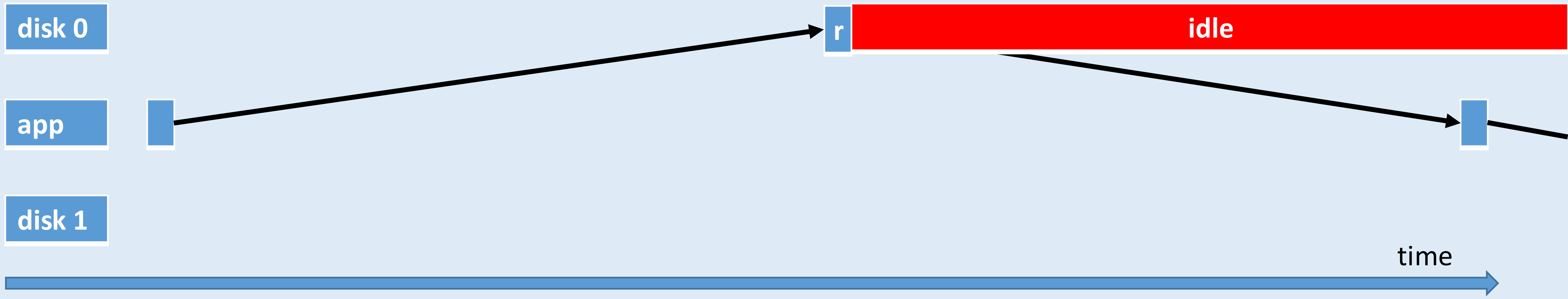
Let us draw time intervals when each disk is accessed:



Synchronous and asynchronous IO, pipelining and multiplexing

Typically, the problem is even worse. If the FS is networked, or the FS is located on a fast NVMe device, then the timeline is going to look this way:

```
while (!done) {
  r = read(fd_in, buf, sizeof(buf));
  r0 = write(fd_out, buf, r);
  ...
}
```



Synchronous and asynchronous IO, pipelining and multiplexing

Typically, the problem is even worse. If the FS is networked, or the FS is located on a fast NVMe device, then the timeline is going to look this way:

```
while (!done) {
  r = read(fd_in, buf, sizeof(buf));
  r0 = write(fd_out, buf, r);
  ...
}
```

```
21-02-18 23:40:38.936 s#1412709.r#6998305: readfile = {offset = 0x4c44d78350, length = 16}
21-02-18 23:40:39.191 s#1412709.r#6998305: send 16 at offset 0x4c44d78350
21-02-18 23:40:39.191 s#1412709.r#6998305: completed

21-02-18 23:40:39.757 s#1412709.r#6998344: readfile = {offset = 0x4c44d78360, length = 944}
21-02-18 23:40:39.757 s#1412709.r#6998344: send 944 at offset 0x4c44d78360
21-02-18 23:40:39.757 s#1412709.r#6998344: completed

21-02-18 23:40:40.242 s#1412709.r#6998358: readfile = {offset = 0x4c44d7e360, length = 16}
21-02-18 23:40:40.361 s#1412709.r#6998358: send 16 at offset 0x4c44d7e360
21-02-18 23:40:40.361 s#1412709.r#6998358: completed
```

Synchronous and asynchronous IO, pipelining and multiplexing

Typically, the problem is even worse. If the FS is networked, or the FS is located on a fast NVMe device, then the timeline is going to look this way:

```
while (!done) {
  r = read(fd_in, buf, sizeof(buf));
  r0 = write(fd_out, buf, r);
  ...
}
```

```
21-02-18 23:40:38.936 s#1412709.r#6998305: readfile = {offset = 0x4c44d78350, length = 16}
21-02-18 23:40:39.191 s#1412709.r#6998305: send 16 at offset 0x4c44d78350
21-02-18 23:40:39.191 s#1412709.r#6998305: completed

21-02-18 23:40:39.757 s#1412709.r#6998344: readfile = {offset = 0x4c44d78360, length = 944}
21-02-18 23:40:39.757 s#1412709.r#6998344: send 944 at offset 0x4c44d78360
21-02-18 23:40:39.757 s#1412709.r#6998344: completed

21-02-18 23:40:40.242 s#1412709.r#6998358: readfile = {offset = 0x4c44d7e360, length = 16}
21-02-18 23:40:40.361 s#1412709.r#6998358: send 16 at offset 0x4c44d7e360
21-02-18 23:40:40.361 s#1412709.r#6998358: completed
```

Synchronous and asynchronous IO, pipelining and multiplexing

Typically, the problem is even worse. If the FS is networked, or the FS is located on a fast NVMe device, then the timeline is going to look this way:

```
while (!done) {
  r = read(fd_in, buf, sizeof(buf));
  r0 = write(fd_out, buf, r);
  ...
}
```

```
21-02-18 23:40:38.936 s#1412709.r#6998305: readfile = {offset = 0x4c44d78350, length = 16}
21-02-18 23:40:39.191 s#1412709.r#6998305: send 16 at offset 0x4c44d78350
21-02-18 23:40:39.191 s#1412709.r#6998305: completed

21-02-18 23:40:39.757 s#1412709.r#6998344: readfile = {offset = 0x4c44d78360, length = 944}
21-02-18 23:40:39.757 s#1412709.r#6998344: send 944 at offset 0x4c44d78360
21-02-18 23:40:39.757 s#1412709.r#6998344: completed

21-02-18 23:40:40.242 s#1412709.r#6998358: readfile = {offset = 0x4c44d7e360, length = 16}
21-02-18 23:40:40.361 s#1412709.r#6998358: send 16 at offset 0x4c44d7e360
21-02-18 23:40:40.361 s#1412709.r#6998358: completed
```

570ms have been wasted

Synchronous and asynchronous IO, pipelining and multiplexing

Typically, the problem is even worse. If the FS is networked, or the FS is located on a fast NVMe device, then the timeline is going to look this way:

```
while (!done) {
  r = read(fd_in, buf, sizeof(buf));
  r0 = write(fd_out, buf, r);
  ...
}
```

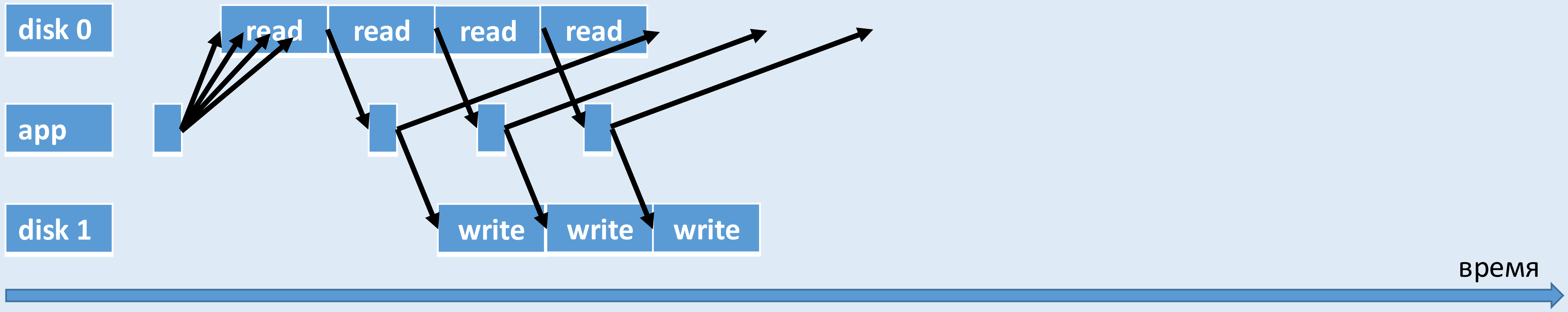
```
21-02-18 23:40:38.936 s#141270                          0x4c44d78350, length = 16}
21-02-18 23:40:39.191 s#14127                          4c44d78350
21-02-18 23:40:39.191 s#14127

21-02-18 23:40:39.757 s#14127                           0x4c44d78360, length = 944}
21-02-18 23:40:39.757 s#14127                          x4c44d78360
21-02-18 23:40:39.757 s#14127

21-02-18 23:40:40.242 s#14127                           0x4c44d7e360, length = 16}
21-02-18 23:40:40.361 s#14127                          4c44d7e360
21-02-18 23:40:40.361 s#141270
```

It took us approx. 1.4 seconds to download 976 bytes.

What is the download speed of this backup software?

Synchronous and asynchronous IO, pipelining and multiplexing

An improvement: issue multiple read requests so that the source disk always have some work to do. The first command still suffers the latency penalty, but subsequent requests have their issue latency masked by preceding requests.



Pipelining and head-of-line blocking

Pipelining has an important inefficiency. Suppose that we've issued requests R_1, R_2, …. The request R_2 can send the reply only after R_1 even if R_2 completes much sooner than R_1. Thus, a slow request blocks all subsequent requests. This scenario is called head-of-line blocking.

Pipelining and head-of-line blocking

Pipelining has an important inefficiency. Suppose that we've issued requests R_1, R_2, …. The request R_2 can send the reply only after R_1 even if R_2 completes much sooner than R_1. Thus, a slow request blocks all subsequent requests. This scenario is called head-of-line blocking.

```
06-09-18 14:12:23.567 s#164034.r#66643120: readfile = {offset = 0x39f0d000, length = 524288}
06-09-18 14:12:23.577 s#164034.r#66643125: readfile = {offset = 0x39f8d000, length = 524288}
06-09-18 14:12:23.593 s#164034.r#66643145: readfile = {offset = 0x3a00d000, length = 524288}
06-09-18 14:12:23.604 s#164034.r#66643147: readfile = {offset = 0x3a08d000, length = 524288}
06-09-18 14:12:23.612 s#164034.r#66643147: send 0x3a08d000:524288
06-09-18 14:12:23.612 s#164034.r#66643147: completed
06-09-18 14:12:23.618 s#164034.r#66643154: readfile = {offset = 0x3a10d000, length = 524288}
06-09-18 14:12:23.627 s#164034.r#66643158: readfile = {offset = 0x3a18d000, length = 524288}
06-09-18 14:12:23.632 s#164034.r#66643154: send 0x3a10d000:524288
06-09-18 14:12:23.632 s#164034.r#66643154: completed
06-09-18 14:12:23.634 s#164034.r#66643166: readfile = {offset = 0x3a20d000, length = 524288}
06-09-18 14:12:23.636 s#164034.r#66643158: send 0x3a18d000:524288
06-09-18 14:12:23.636 s#164034.r#66643158: completed
06-09-18 14:12:23.641 s#164034.r#66643168: readfile = {offset = 0x3a28d000, length = 524288}
06-09-18 14:12:23.643 s#164034.r#66643166: send 0x3a20d000:524288
06-09-18 14:12:23.643 s#164034.r#66643166: completed
06-09-18 14:12:23.649 s#164034.r#66643168: send 0x3a28d000:524288
06-09-18 14:12:23.649 s#164034.r#66643168: completed
06-09-18 14:12:23.783 s#164034.r#66643120: send 0x39f0d000:524288
06-09-18 14:12:23.783 s#164034.r#66643120: completed
```

Pipelining and head-of-line blocking

Pipelining has an important inefficiency. Suppose that we've issued requests R_1, R_2, …. The request R_2 can send the reply only after R_1 even if R_2 completes much sooner than R_1. Thus, a slow request blocks all subsequent requests. This scenario is called head-of-line blocking.

```
06-09-18 14:12:23.567 s#164034.r#66643120: readfile = {offset = 0x39f0d000, length = 524288}
06-09-18 14:12:23.577 s#164034.r#66643125: readfile = {offset = 0x39f8d000, length = 524288}
06-09-18 14:12:23.593 s#164034.r#66643145: readfile = {offset = 0x3a00d000, length = 524288}
06-09-18 14:12:23.604 s#164034.r#66643147: readfile = {offset = 0x3a08d000, length = 524288}
06-09-18 14:12:23.612 s#164034.r#66643147: send 0x3a08d000:524288
06-09-18 14:12:23.612 s#164034.r#66643147: completed
06-09-18 14:12:23.618 s#164034.r#66643154: readfile = {offset = 0x3a10d000, length = 524288}
06-09-18 14:12:23.627 s#164034.r#66643158: readfile = {offset = 0x3a18d000, length = 524288}
06-09-18 14:12:23.632 s#164034.r#66643154: send 0x3a10d000:524288
06-09-18 14:12:23.632 s#164034.r#66643154: completed
06-09-18 14:12:23.634 s#164034.r#66643166: readfile = {offset = 0x3a20d000, length = 524288}
06-09-18 14:12:23.636 s#164034.r#66643158: send 0x3a18d000:524288
06-09-18 14:12:23.636 s#164034.r#66643158: completed
06-09-18 14:12:23.641 s#164034.r#66643168: readfile = {offset = 0x3a28d000, length = 524288}
06-09-18 14:12:23.643 s#164034.r#66643166: send 0x3a20d000:524288
06-09-18 14:12:23.643 s#164034.r#66643166: completed
06-09-18 14:12:23.649 s#164034.r#66643168: send 0x3a28d000:524288
06-09-18 14:12:23.649 s#164034.r#66643168: completed
06-09-18 14:12:23.783 s#164034.r#66643120: send 0x39f0d000:524288
06-09-18 14:12:23.783 s#164034.r#66643120: completed
```

Pipelining and head-of-line blocking

Pipelining has an important inefficiency. Suppose that we've issued requests R_1, R_2, …. The request R_2 can send the reply only after R_1 even if R_2 completes much sooner than R_1. Thus, a slow request blocks all subsequent requests. This scenario is called head-of-line blocking.

```
06-09-18 14:12:23.567 s#164034.r#66643120: readfile = {offset = 0x39f0d000, length = 524288}
06-09-18 14:12:23.577 s#164034.r#66643125: readfile = {offset = 0x39f8d000, length = 524288}
06-09-18 14:12:23.593 s#164034.r#66643145: readfile = {offset = 0x3a00d000, length = 524288}
06-09-18 14:12:23.604 s#164034.r#66643147: readfile = {offset = 0x3a08d000, length = 524288}
06-09-18 14:12:23.612 s#164034.r#66643147: send 0x3a08d000:52428
06-09-18 14:12:23.612 s#164034.r#66643147: completed
06-09-18 14:12:23.618 s#164034.r#66643154: readfile = {offset
06-09-18 14:12:23.627 s#164034.r#66643158: readfile = {offset
06-09-18 14:12:23.632 s#164034.r#66643154: send 0x3a10d000:524
06-09-18 14:12:23.632 s#164034.r#66643154: completed
06-09-18 14:12:23.634 s#164034.r#66643166: readfile = {offset
06-09-18 14:12:23.636 s#164034.r#66643158: send 0x3a18d000:524
06-09-18 14:12:23.636 s#164034.r#66643158: completed
06-09-18 14:12:23.641 s#164034.r#66643168: readfile = {offset
06-09-18 14:12:23.643 s#164034.r#66643166: send 0x3a20d000:5242
06-09-18 14:12:23.643 s#164034.r#66643166: completed
06-09-18 14:12:23.649 s#164034.r#66643168: send 0x3a28d000:524288
06-09-18 14:12:23.649 s#164034.r#66643168: completed
06-09-18 14:12:23.783 s#164034.r#66643120: send 0x39f0d000:524288
06-09-18 14:12:23.783 s#164034.r#66643120: completed
```

Request r#66643120 accessed a disk that was busy with another request.

Request r#66643147 was executed by a disk that had no other IOs. The response was ready very quickly, but the server may not send that response before the response to r#66643120.

Synchronous and asynchronous IO, pipelining and multiplexing

Pipelining has an important inefficiency. Suppose that we've issued requests R_1, R_2, …. The request R_2 can send the reply only after R_1 even if R_2 completes much sooner than R_1. Thus, a slow request blocks all subsequent requests. This scenario is called head-of-line blocking.

One can avoid head-of-line blocking this way:

- add a unique sequence number to each IO request,
- have the server send replies that include the request sequence number.

Synchronous and asynchronous IO, pipelining and multiplexing

Pipelining has an important inefficiency. Suppose that we've issued requests R_1, R_2, …. The request R_2 can send the reply only after R_1 even if R_2 completes much sooner than R_1. Thus, a slow request blocks all subsequent requests. This scenario is called head-of-line blocking.

One can avoid head-of-line blocking this way:

- add a unique sequence number to each IO request,
- have the server send replies that include the request sequence number.

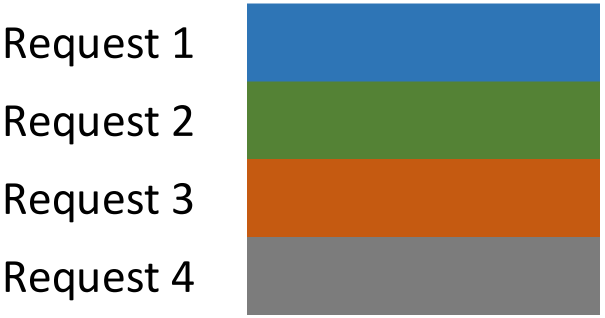
Many protocols use this idea:

- SCTP,
- HTTP/2,
- QUIC*.

* The QUIC Transport Protocol: Design and Internet-scale Deployment: https://research.google.com/pubs/archive/46403.pdf

How the network interacts with concurrent requests

A client that send multiple requests over multiple network connections	The network	The server’s request queue



How the network interacts with concurrent requests

A client that send multiple requests over multiple network connections

The network

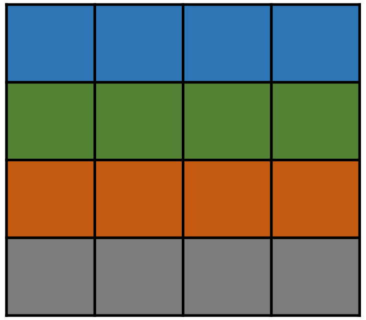
The server’s request queue

Request 1

Request 2

Request 3

Request 4



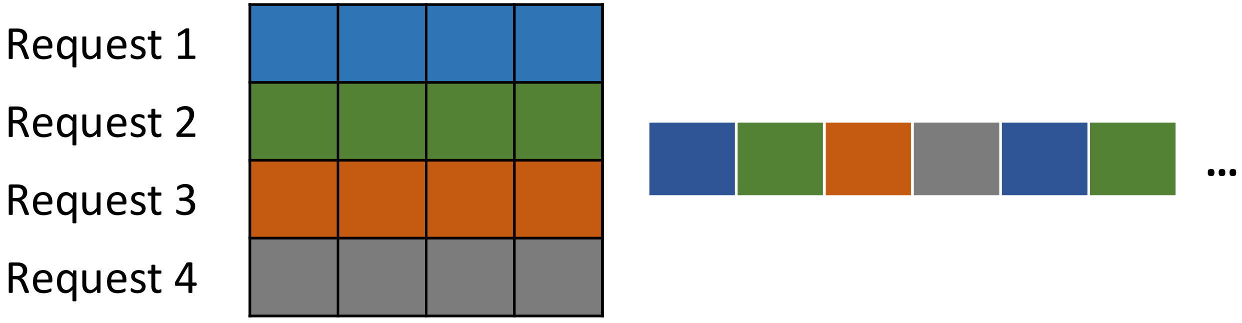
From the point of view of the network different connections are independent streams of bytes and each must get an equal share of the bandwidth.

How the network interacts with concurrent requests

A client that send multiple requests over multiple network connections

The network

The server’s request queue



From the point of view of the network different connections are independent streams of bytes and each must get an equal share of the bandwidth.

How the network interacts with concurrent requests

A client that send multiple requests over multiple network connections

The network

The server's request queue

From the point of view of the network different connections are independent streams of bytes and each must get an equal share of the bandwidth.

How the network interacts with concurrent requests

A client that send multiple requests over multiple network connections

The network

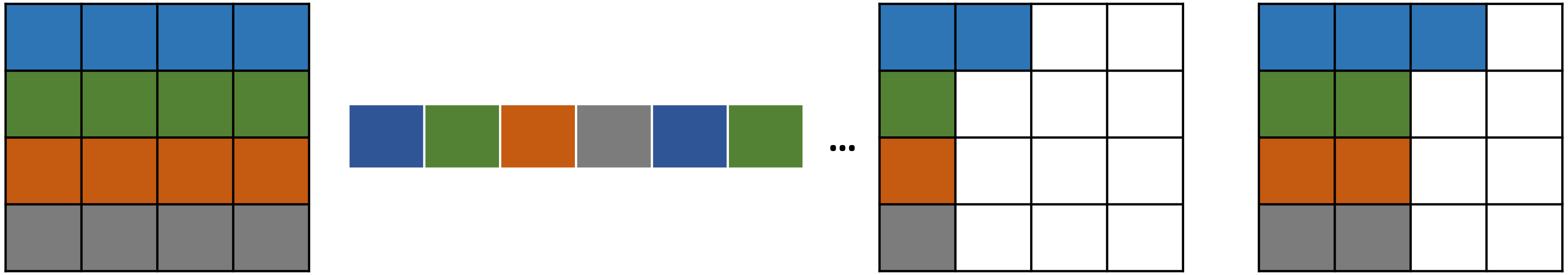
The server's request queue

Request 1

Request 2

Request 3

Request 4



From the point of view of the network different connections are independent streams of bytes and each must get an equal share of the bandwidth.

Some protocols like gRPC cannot handle incomplete requests. They need to fetch all arguments first.

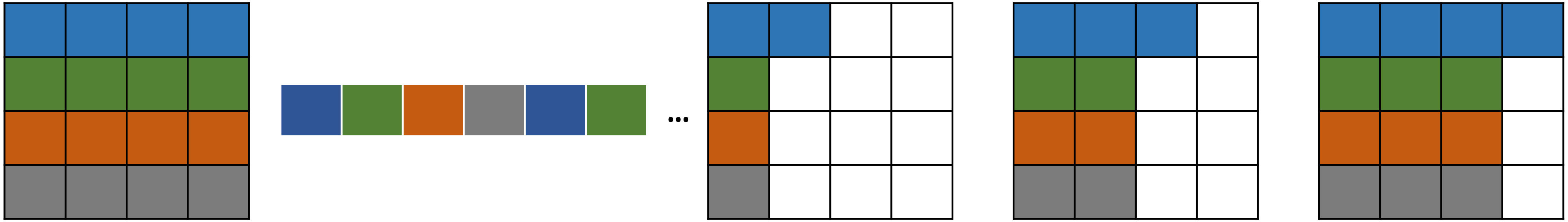
How the network interacts with concurrent requests

A client that send multiple requests over multiple network connections

The network

The server's request queue

Request 1
Request 2
Request 3
Request 4



From the point of view of the network different connections are independent streams of bytes and each must get an equal share of the bandwidth.

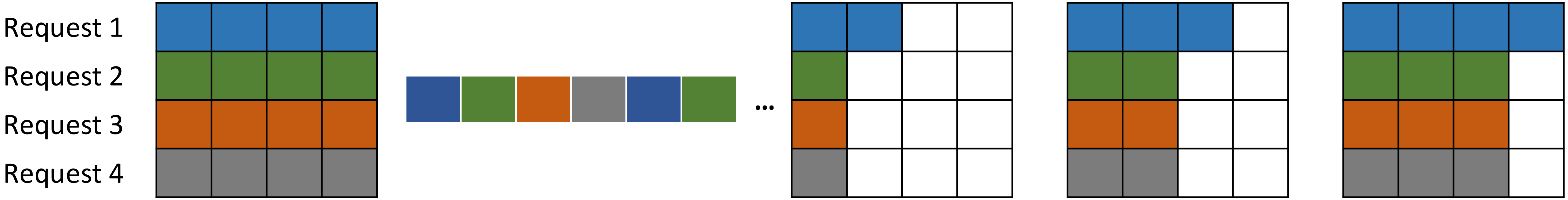
It is only now that the server received a complete request and can handle it. Essentially, it waited for all requests to arrive before starting the first one.

The delay is no better than when we have a single huge request.

** See also It's Time to Replace TCP in the Datacenter*
https://arxiv.org/pdf/2210.00714.pdf

How the network interacts with concurrent requests

A client that send multiple requests over multiple network connections | The network | The server's request queue

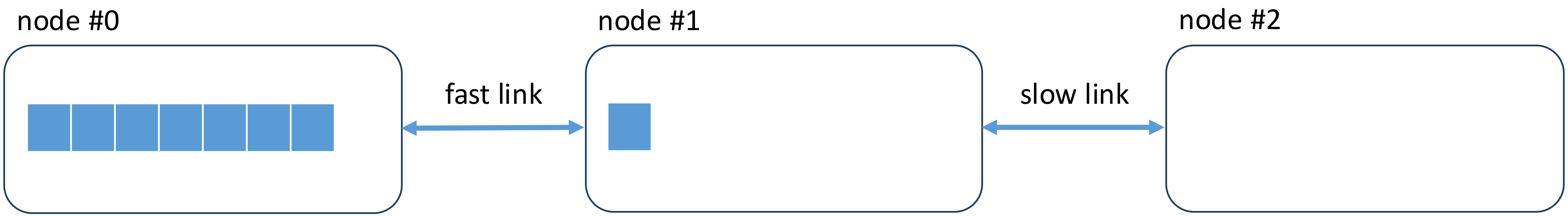


There are various ways this scenario may happen:

1. HTTP/2 splits requests into fixed-size frames and interleaves frames from different requests,
2. A TCP stream is split into IP packets and packets from different connections are interleaved.

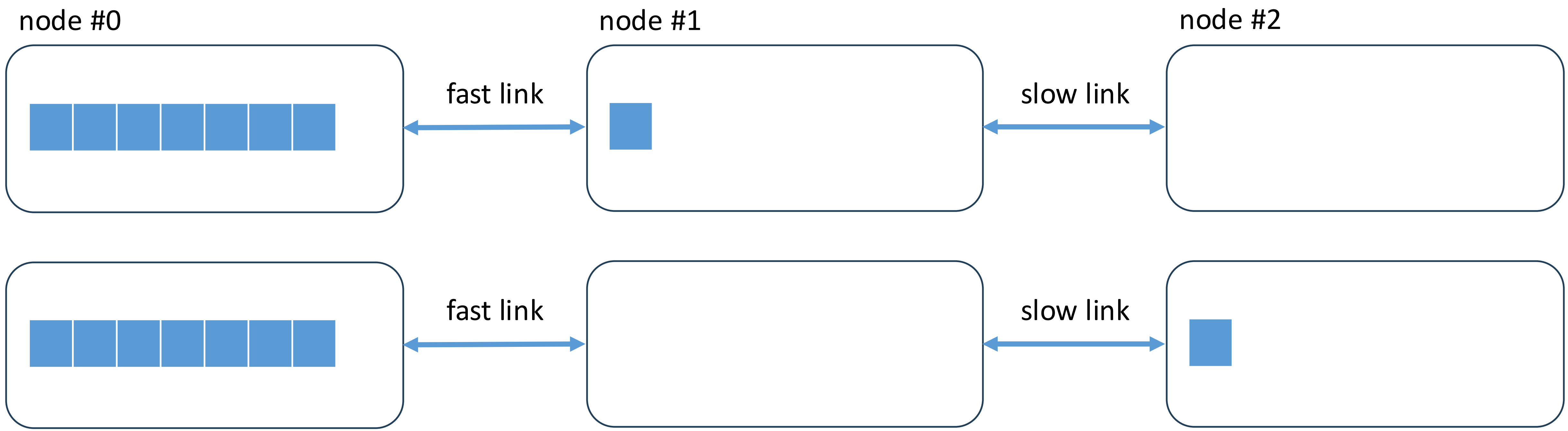
Quiz: it appears that uploading a file via multiple connections is a wrong idea? GRPC had to disable framing in the HTTP/2 client, but a TCP connection cannot monopolise a link.

Buffering and bufferbloat

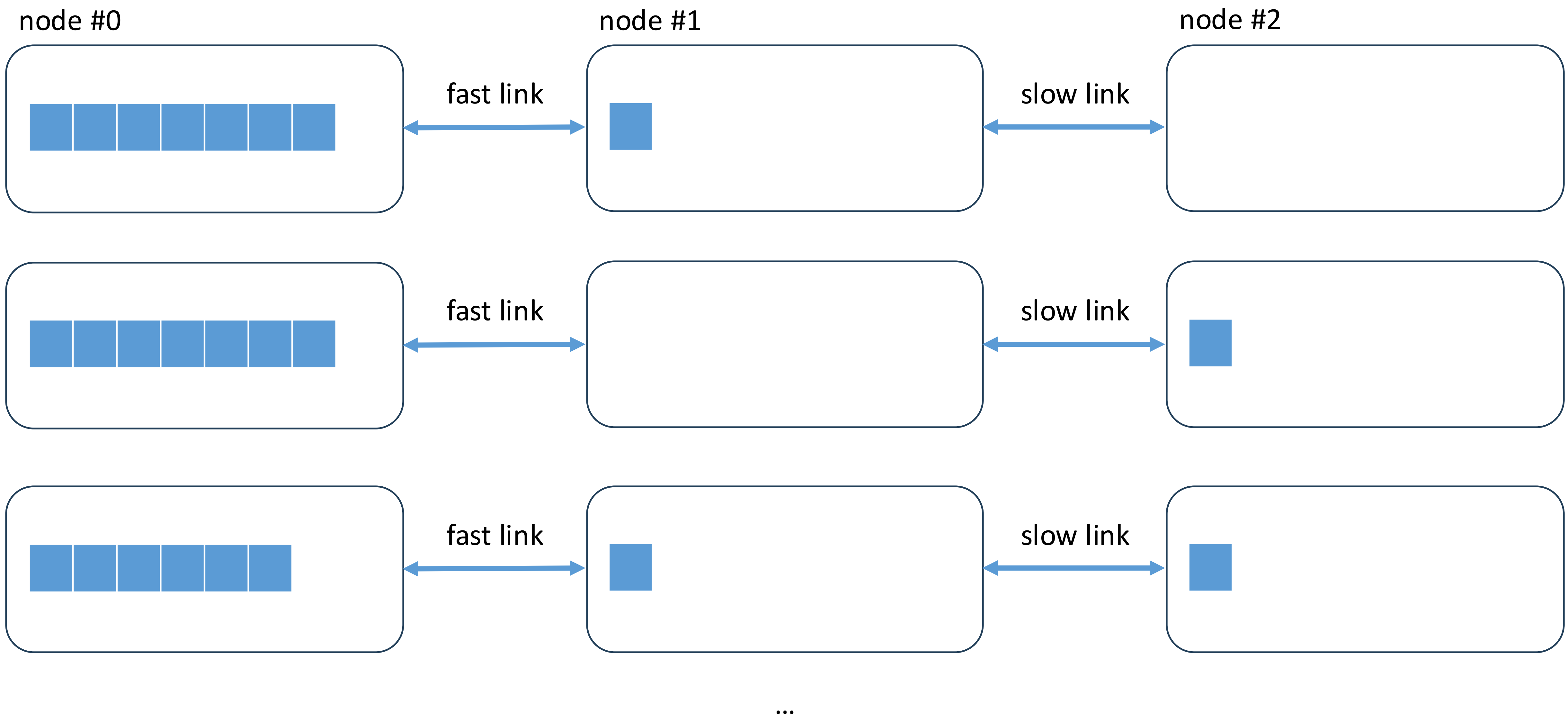


Suppose that node #0 is sending data to node #2 via an intermediate node and there is no buffering in node #1.

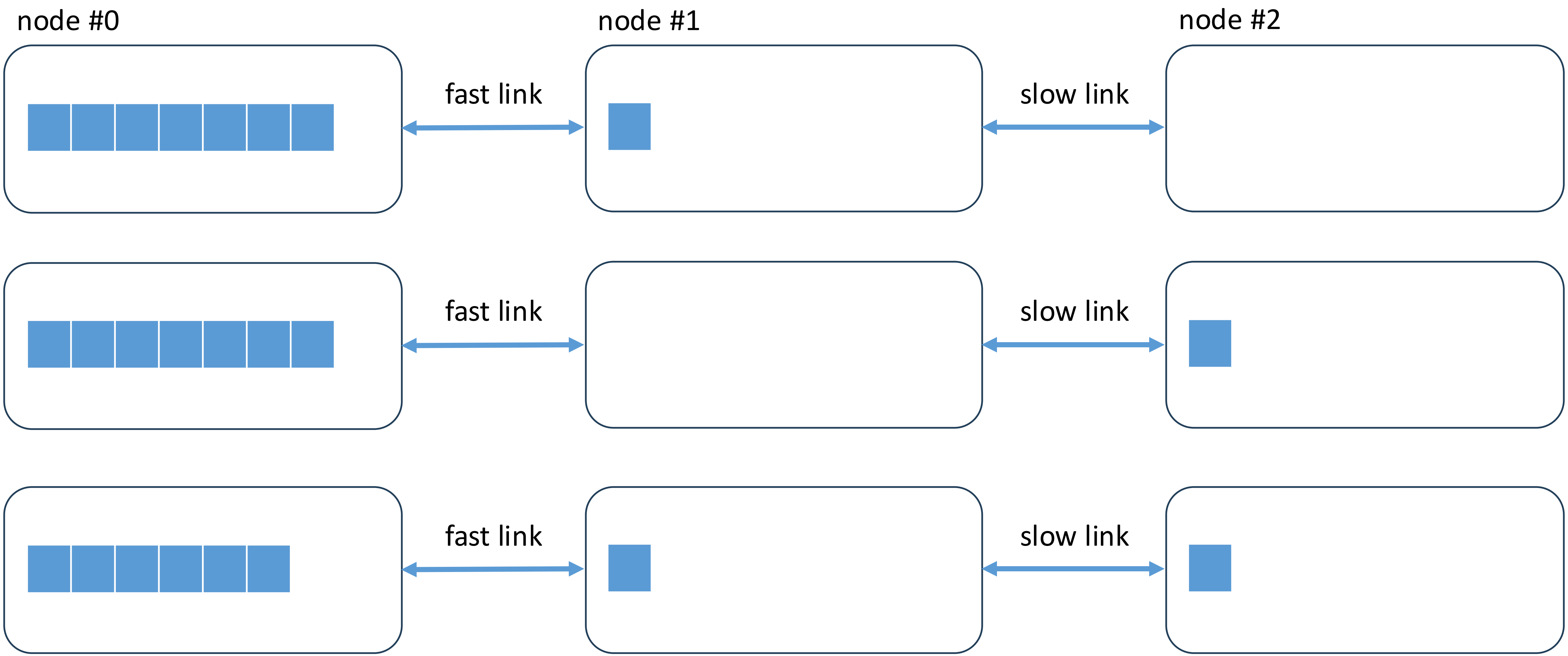
Buffering and bufferbloat



Buffering and bufferbloat

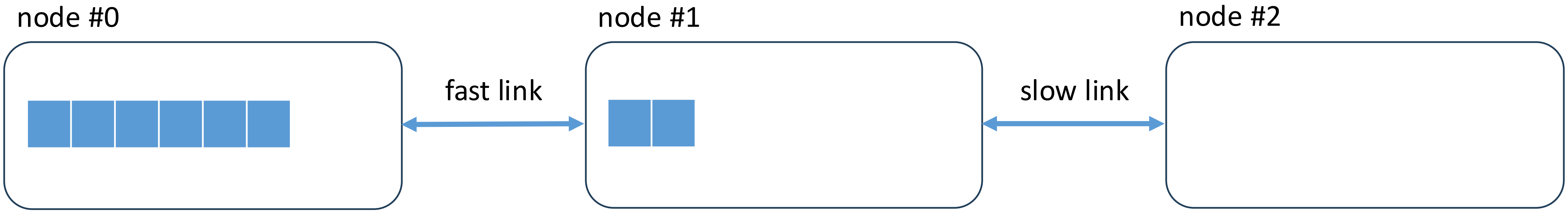


Buffering and bufferbloat



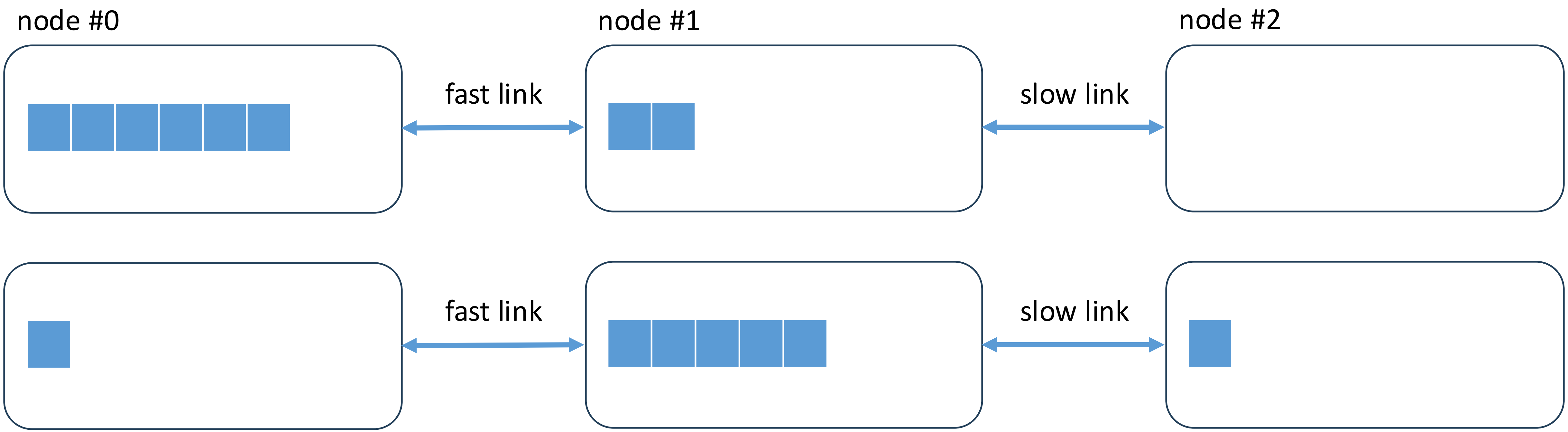
This scheme is far from optimal. Node #0 must keep much data for every slow client. Also, the data transfer rate between #0 and #1 needs time to adjust to changes in the bandwidth of the slow link.

Buffering and bufferbloat

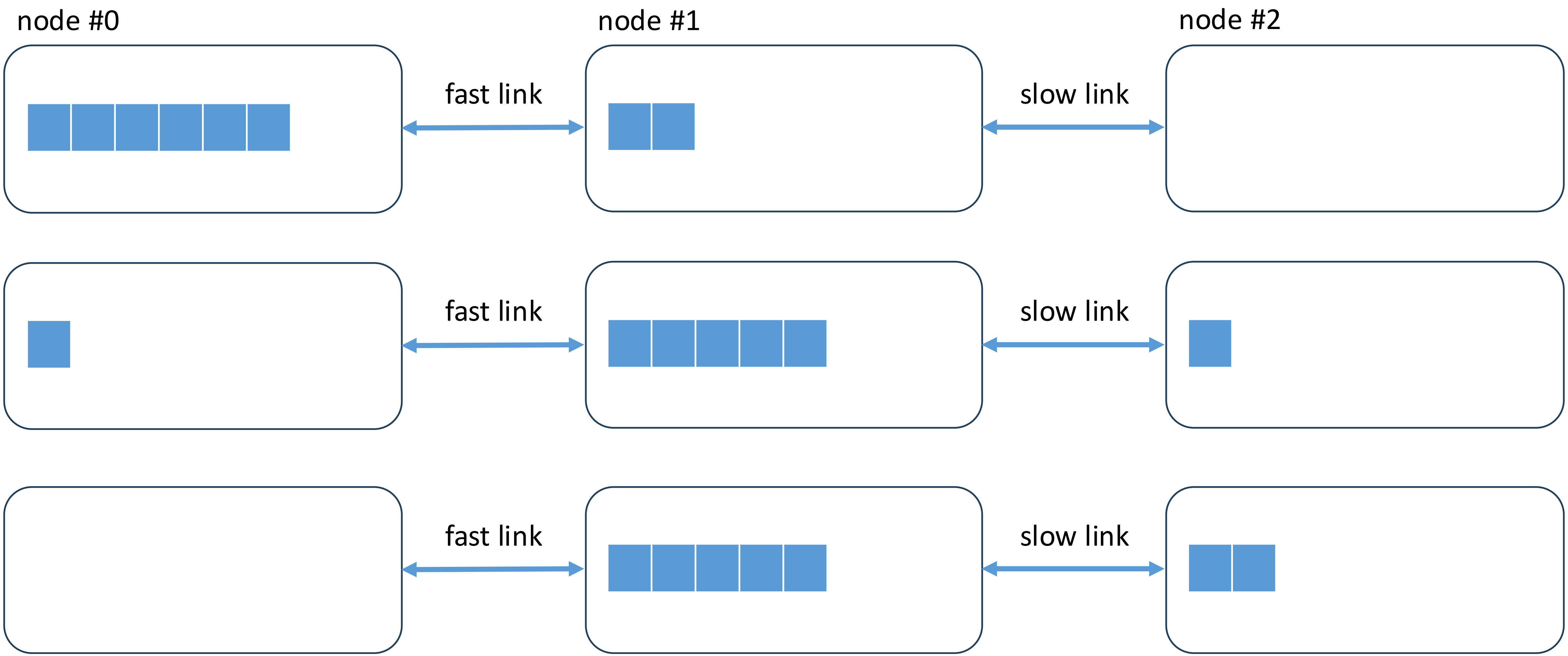


Now permit the queue at node #1 grow longer.

Buffering and bufferbloat

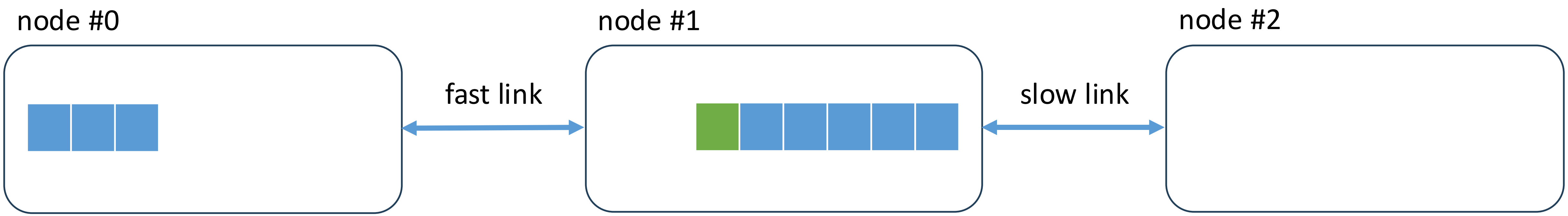


Buffering and bufferbloat



This looks better. Node #0 has released the buffers, and the transfer speed between nodes #1 and #2 adjusts itself immediately to changes in the slow link.

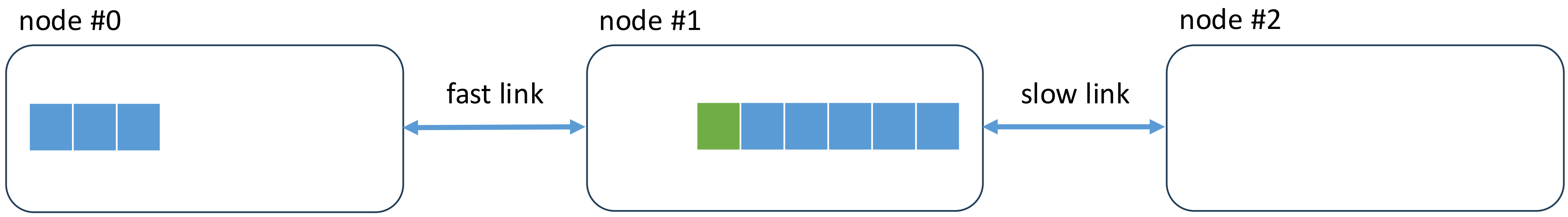
Buffering and bufferbloat



There is problem, though.

Suppose there are two flows from node #0 to node #2, one that needs a lot of bandwidth, and another one that is latency-sensitive. In many cases the packets at node #1 cannot be reordered, and a latency-sensitive flow must wait for packets of a high-bandwidth flow to drain from the queue.

Buffering and bufferbloat



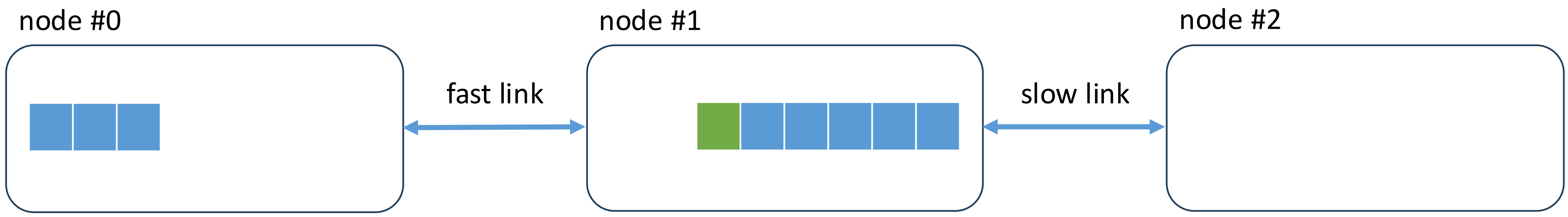
There is problem, though.

Suppose there are two flows from node #0 to node #2, one that needs a lot of bandwidth, and another one that is latency-sensitive. In many cases the packets at node #1 cannot be reordered, and a latency-sensitive flow must wait for packets of a high-bandwidth flow to drain from the queue.

There are many scenarios when packets (more generally, IO requests) can no longer be reordered:

1. packets were transferred to the (hardware) output queue of a network interface,

Buffering and bufferbloat



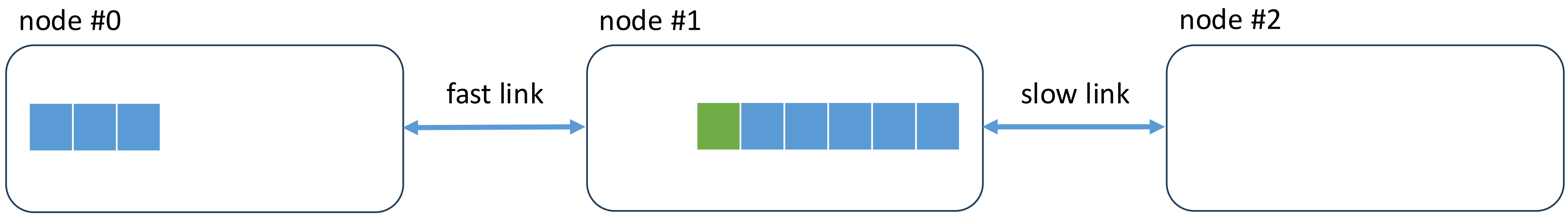
There is problem, though.

Suppose there are two flows from node #0 to node #2, one that needs a lot of bandwidth, and another one that is latency-sensitive. In many cases the packets at node #1 cannot be reordered, and a latency-sensitive flow must wait for packets of a high-bandwidth flow to drain from the queue.

There are many scenarios when packets (more generally, IO requests) can no longer be reordered:

1. packets were transferred to the (hardware) output queue of a network interface,
2. threads of a threadpool entered `preadv() / pwritev()` while doing unbuffered IO,

Buffering and bufferbloat



There is problem, though.

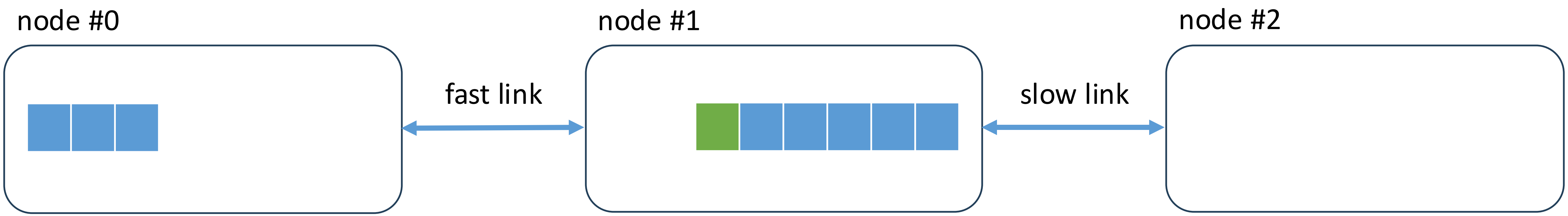
Suppose there are two flows from node #0 to node #2, one that needs a lot of bandwidth, and another one that is latency-sensitive. In many cases the packets at node #1 cannot be reordered, and a latency-sensitive flow must wait for packets of a high-bandwidth flow to drain from the queue.

There are many scenarios when packets (more generally, IO requests) can no longer be reordered:

1. packets were transferred to the (hardware) output queue of a network interface,
2. threads of a threadpool entered `preadv() / pwritev()` while doing unbuffered IO,

Note: this particular scenario is also bad because threads have no control over the scheduling. Even if IO requests go to adjacent disk areas, they may appear randomly ordered to the kernel. Recall that preserving the request order is an important advantage of io_uring.

Buffering and bufferbloat



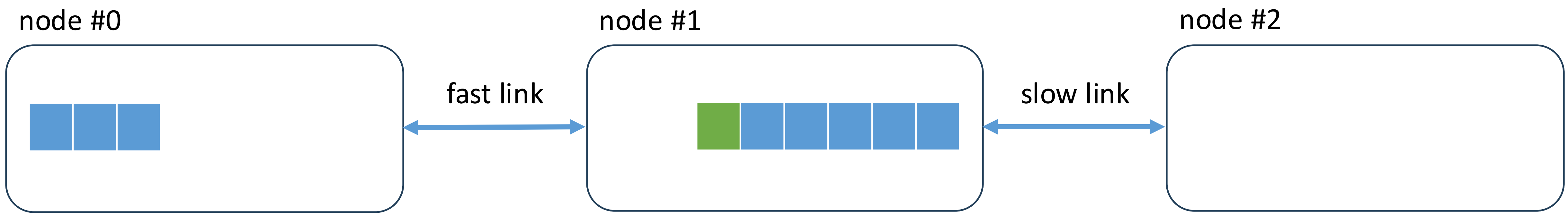
There is problem, though.

Suppose there are two flows from node #0 to node #2, one that needs a lot of bandwidth, and another one that is latency-sensitive. In many cases the packets at node #1 cannot be reordered, and a latency-sensitive flow must wait for packets of a high-bandwidth flow to drain from the queue.

There are many scenarios when packets (more generally, IO requests) can no longer be reordered:

1. packets were transferred to the (hardware) output queue of a network interface,
2. threads of a threadpool entered `preadv() / pwritev()` while doing unbuffered IO,
3. buffered writes dirtied pages in the pagecache,

Buffering and bufferbloat



There is problem, though.

Suppose there are two flows from node #0 to node #2, one that needs a lot of bandwidth, and another one that is latency-sensitive. In many cases the packets at node #1 cannot be reordered, and a latency-sensitive flow must wait for packets of a high-bandwidth flow to drain from the queue.

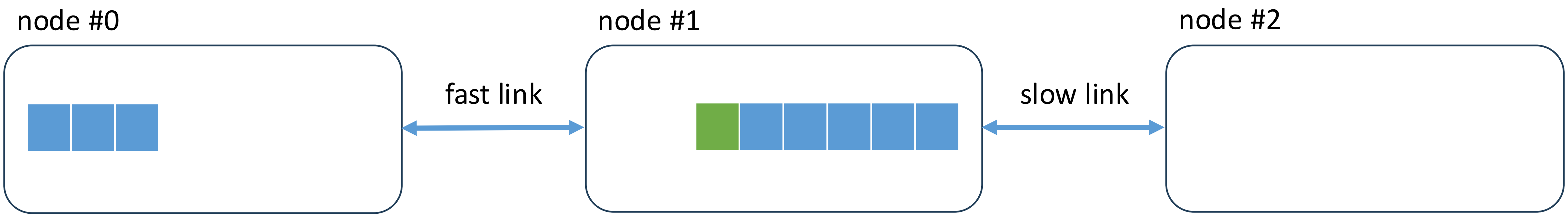
There are many scenarios when packets (more generally, IO requests) can no longer be reordered:

1. packets were transferred to the (hardware) output queue of a network interface,
2. threads of a threadpool entered `preadv() / pwritev()` while doing unbuffered IO,
3. buffered writes dirtied pages in the pagecache,

Note: userspace applications have little control on the order of the writeback. Moreover, the writeback from the page cache is single-threaded, and may become a bottleneck: https://lwn.net/Articles/976856.

See also: `man 2 sync_file_range`

Buffering and bufferbloat



There is problem, though.

Suppose there are two flows from node #0 to node #2, one that needs a lot of bandwidth, and another one that is latency-sensitive. In many cases the packets at node #1 cannot be reordered, and a latency-sensitive flow must wait for packets of a high-bandwidth flow to drain from the queue.

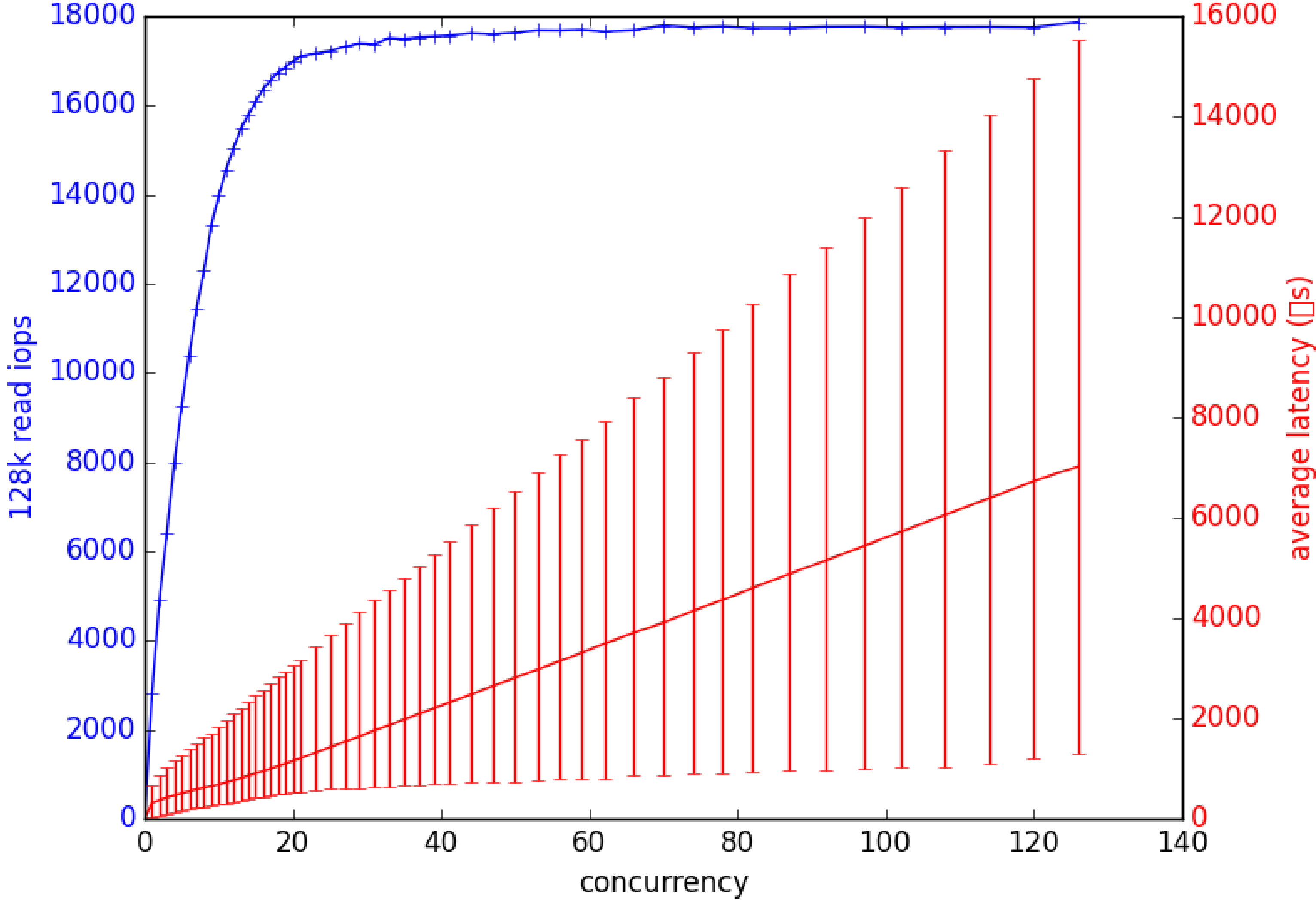
There are many scenarios when packets (more generally, IO requests) can no longer be reordered:

1. packets were transferred to the (hardware) output queue of a network interface,
2. threads of a threadpool entered `preadv() / pwritev()` while doing unbuffered IO,
3. buffered writes dirtied pages in the pagecache,
4. an HTTP request is submitted to golang's standard HTTP client,
5. …

See also: https://lwn.net/Articles/458625, https://queue.acm.org/detail.cfm?id=2209336

More on IO queues

How the IOPS and the request latency depend on the request queue depth:

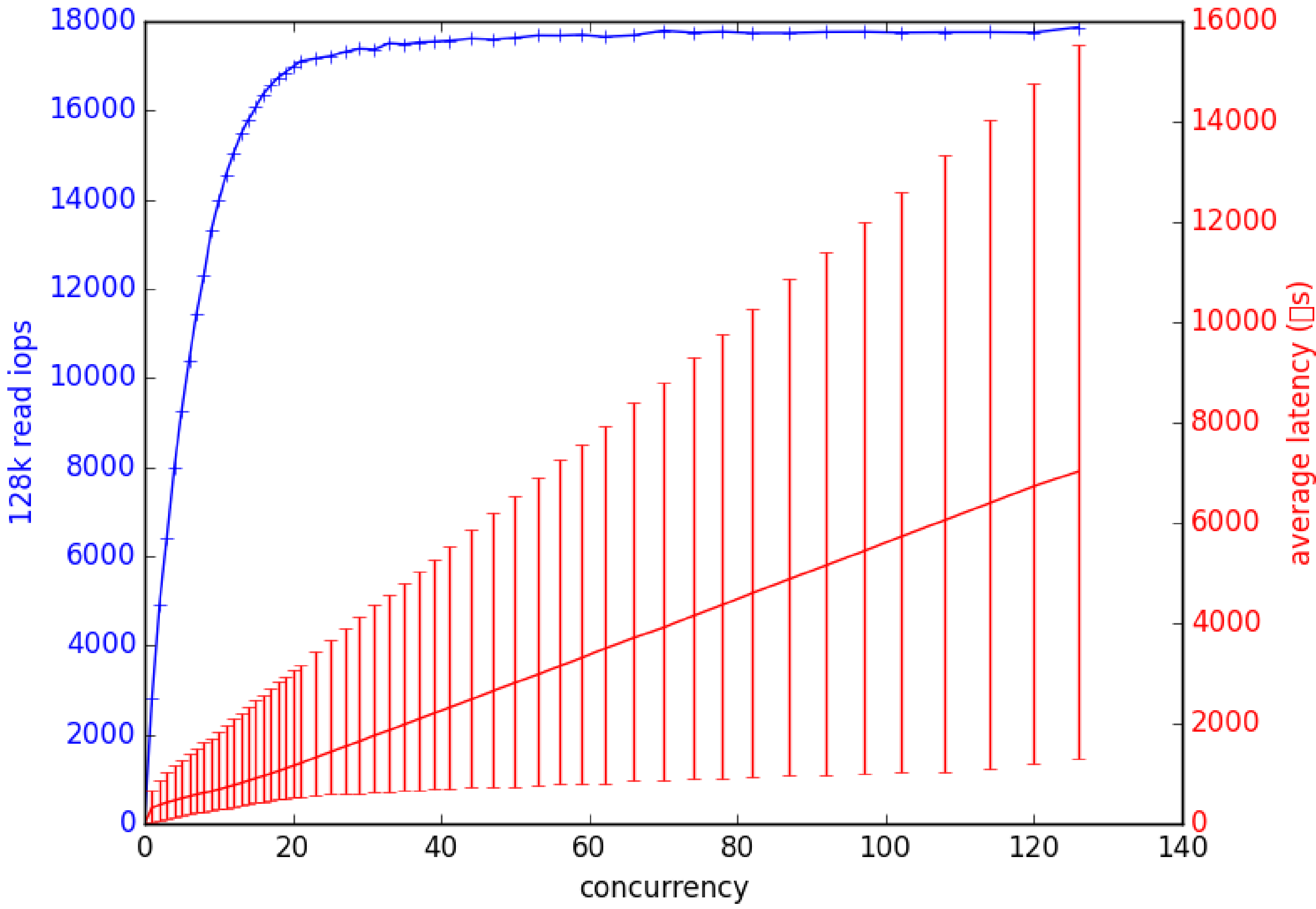


https://www.scylladb.com/2018/04/19/scylla-i-o-scheduler-3

More on IO queues

How the IOPS and the request latency depend on the request queue depth:

GCS exhibits a similar behaviour. Chunk uploads in the range from 24M to 32M are fast and have a predictable latency. Increasing the chunk size decreases the throughput and increases the deviation of request run times.



https://www.scylladb.com/2018/04/19/scylla-i-o-scheduler-3

More on IO queues

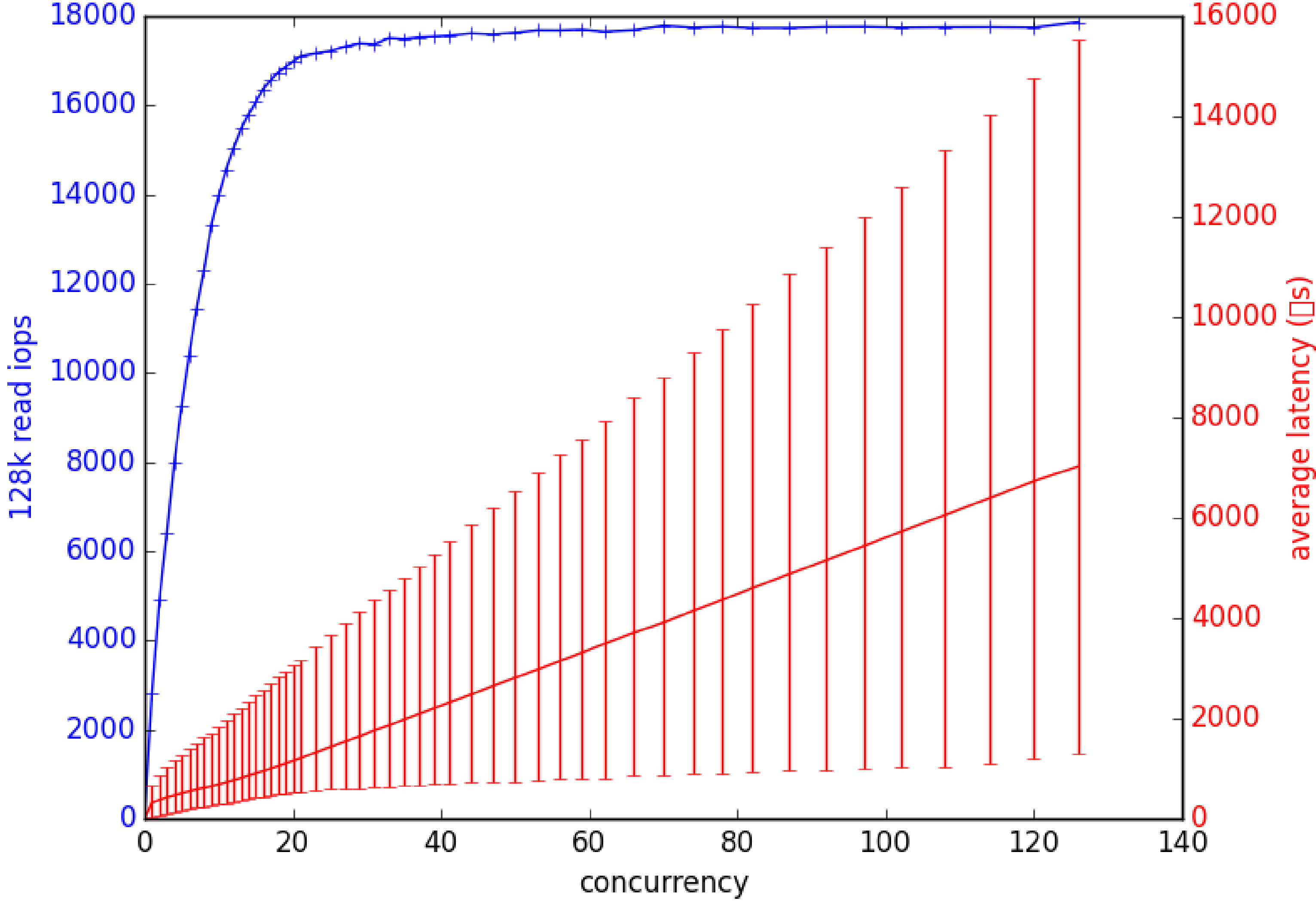
How the IOPS and the request latency depend on the request queue depth:

GCS exhibits a similar behaviour. Chunk uploads in the range from 24M to 32M are fast and have a predictable latency. Increasing the chunk size decreases the throughput and increases the deviation of request run times.

Corollary: there is no need to issue too many concurrent requests or make requests too long. It is preferable to have a user-space queue that an application fully controls and can modify it the way it whishes.

***See also**: "ScyllaDB userspace disk IO scheduler":*

- *https://www.scylladb.com/2016/04/14/io-scheduler-1*
- *https://www.scylladb.com/2016/04/29/io-scheduler-2*
- *https://www.scylladb.com/2018/04/19/scylla-i-o-scheduler-3*



Load shedding

Idea: there is no need to issue too many concurrent requests or make requests too long. It is preferable to have a user-space queue that an application fully controls and can modify it the way it whishes.

This limits the size of egress request queues produced by an application.

How do we handle ingress queues that grow too big?

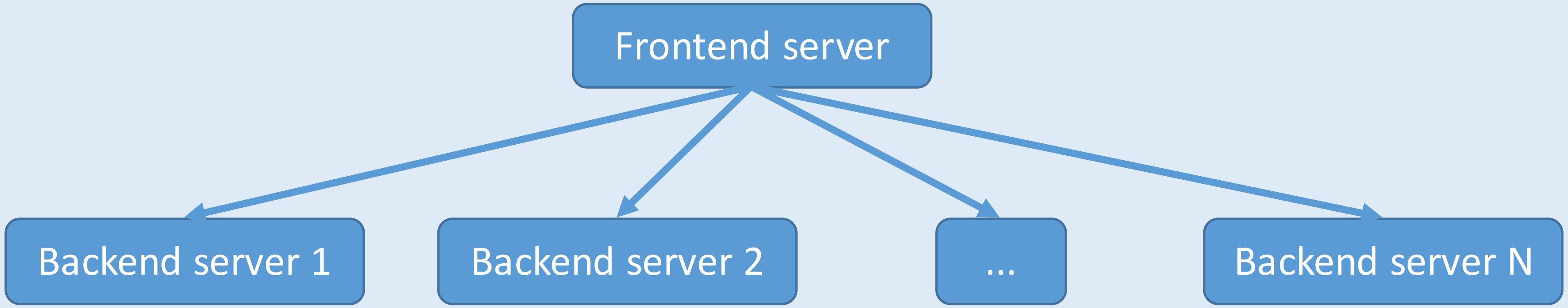
Load shedding

When handling congested queues of IP packets, it suffices to drop some of them randomly. This may be further improved by sending ECNs (Explicit Congestion Notifications). See CoDel and Random Early Drop.

Load shedding

When handling congested queues of IP packets, it suffices to drop some of them randomly. This may be further improved by sending ECNs (Explicit Congestion Notifications). See CoDel and Random Early Drop.

Dropping requests at random in a distributed system may be a poor idea:



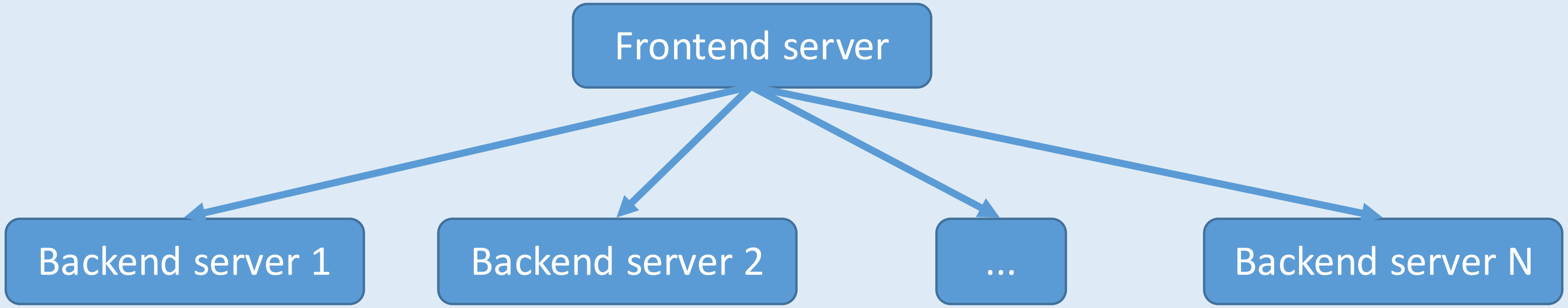
Typically, a request handler issues multiple sub-requests to other services in a distributed system and constructs a response from responses to sub-requests.

Suppose that servers 1, 2, ..., N-1 executed their sub-requests, but server N decided to load-shed its sub-request. Because of this, the whole top-level request can't be served and needs more resources to retry sub-requests. This only leads to increased load on all services within a distributed system.

Load shedding

When handling congested queues of IP packets, it suffices to drop some of them randomly. This may be further improved by sending ECNs (Explicit Congestion Notifications). See CoDel and Random Early Drop.

Dropping requests at random in a distributed system may be a poor idea:

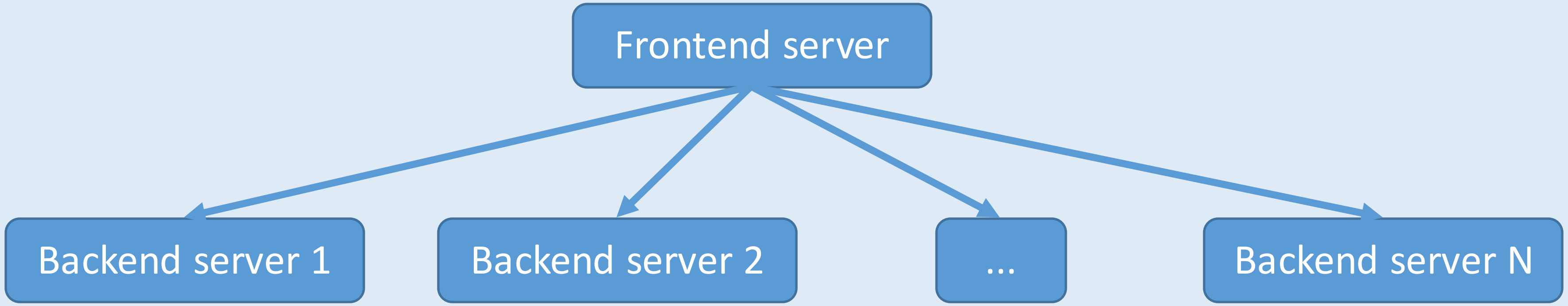


Typically, a request handler issues multiple sub-requests to other services in a distributed system and constructs a response from responses to sub-requests.

Suppose that servers 1, 2, ..., N-1 executed their sub-requests, but server N decided to load-shed its sub-request. Because of this, the whole top-level request can’t be served and needs more resources to retry sub-requests. This only leads to increased load on all services within a distributed system.

Reminder: this is much like tail latencies. The probability of **any** of N servers load-shedding their sub-request grows with N.

Load shedding



Typically, a request handler issues multiple sub-requests to other services in a distributed system and constructs a response from responses to sub-requests.

Suppose that servers 1, 2, ..., N-1 executed their sub-requests, but server N decided to load-shed its sub-request. Because of this, the whole top-level request can't be served and needs more resources to retry sub-requests. This only leads to increased load on all services within a distributed system.

Idea:

1. do not drop requests at random, but have request priorities and drop low-priority requests,
2. prefer to load-shed requests instead of their sub-requests.

Load shedding

Idea:

1. do not drop requests at random, but have request priorities and drop low-priority requests,
2. prefer to load-shed requests instead of their sub-requests.

A naïve solution would be to assign request priorities statically.

However, that is a flawed approach. One cannot invent many of them, and each priority level becomes big enough to cause oscillating behaviour in the system:

- **Quiz**: describe the behaviour.

Load shedding

Idea:
1. do not drop requests at random, but have request priorities and drop low-priority requests,
2. prefer to load-shed requests instead of their sub-requests.

A naïve solution would be to assign request priorities statically.

However, that is a flawed approach. One cannot invent many of them, and each priority level becomes big enough to cause oscillating behaviour in the system:
- once there are many enough requests with priorities <= N, start to drop all requests with priorities >N,
- this removes a large chunk of load on the system so that it is no longer overloaded,
- start accepting requests with priorities >N and become overloaded again.

A much better idea is to choose priority levels at run time. That way we can have a lot of them and make them fine-grained. For example, within every statically allocated priority level we may a user ID as a refinement to the priority level.

See also: https://www.cs.columbia.edu/~ruigu/papers/socc18-final100.pdf