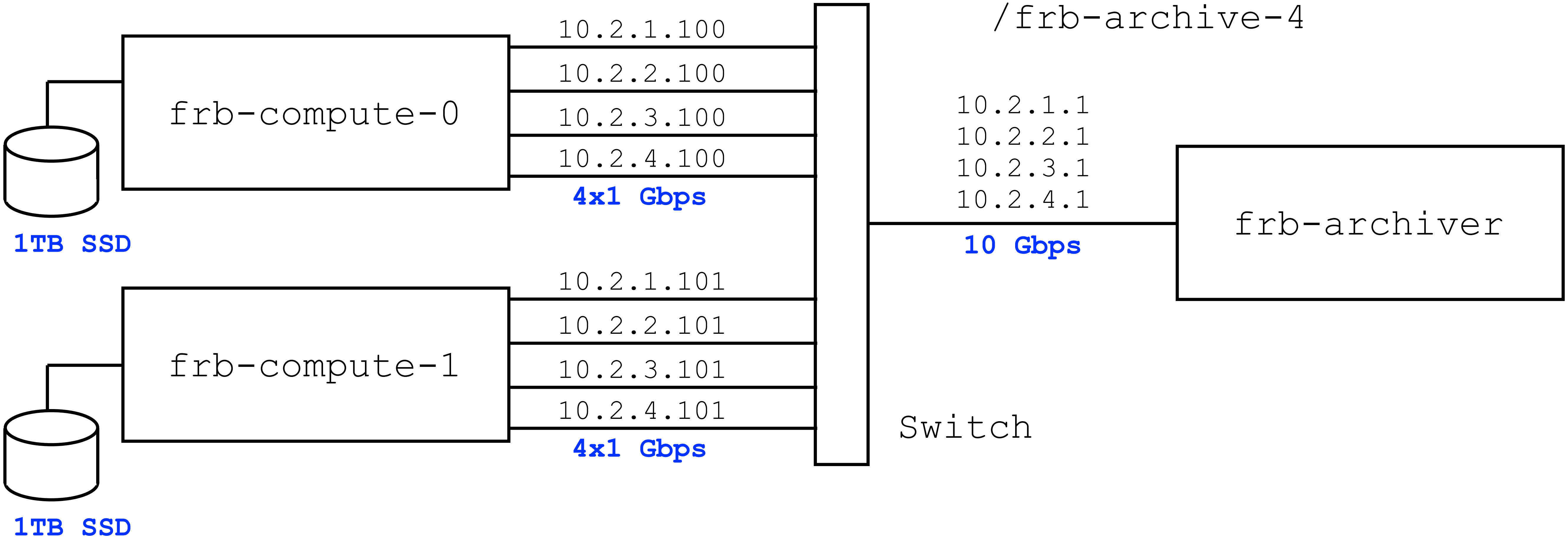
L1 server file-writing code

Kendrick, Dustin

Introduction

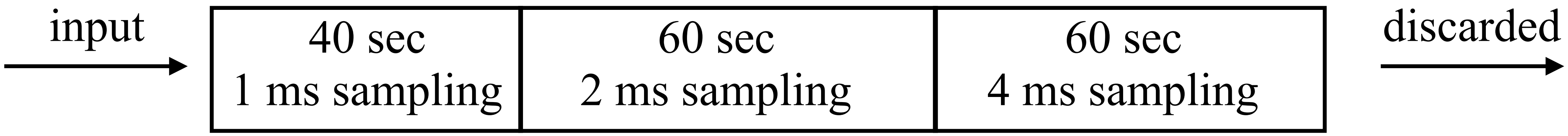
- Each FRB node has four 1 Gbps NIC’s.
- The FRB node “thinks” each NIC is on a different network, but they are physically plugged into the same switch.
- Incoming traffic is balanced by sending four beams to each NIC.
- Outgoing NFS traffic is balanced by making the FRB node “think” the NFS server is four servers:

```
/frb-archive-1
/frb-archive-2
/frb-archive-3
/frb-archive-4
```



Introduction

- Input data rate (per node) is 2.2 Gbps.
- The node buffers its data in a “telescoping” ring buffer.



input

40 sec
1 ms sampling

60 sec
2 ms sampling

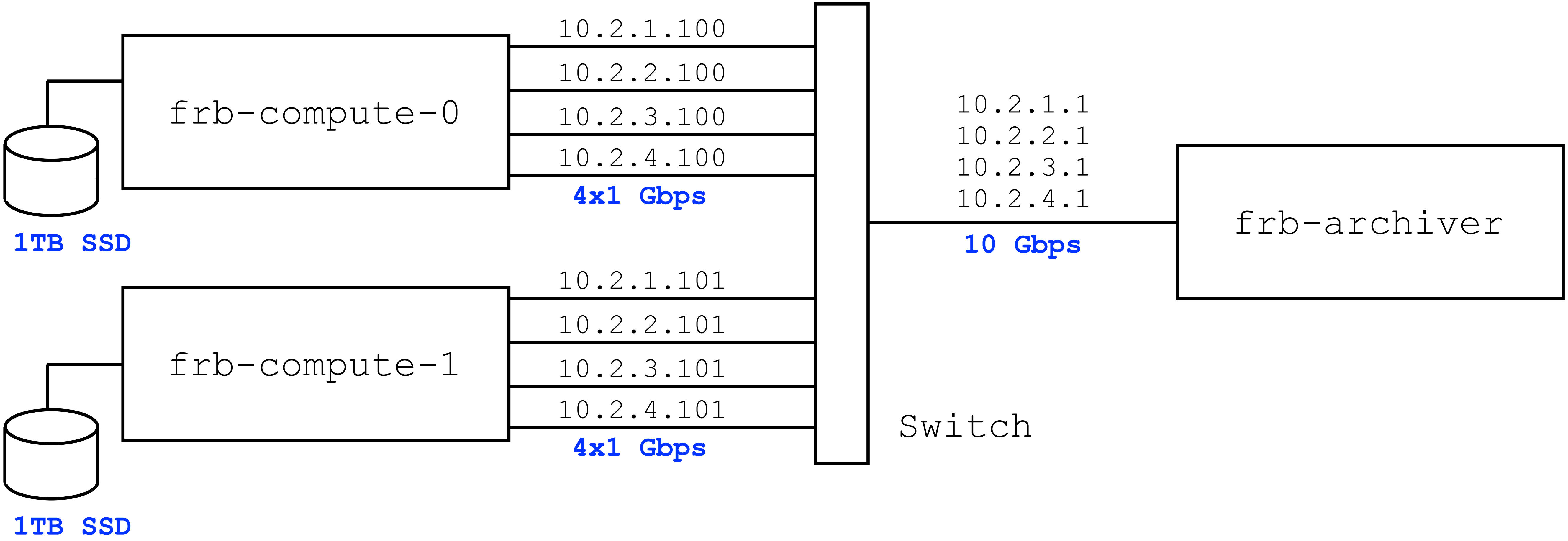
60 sec
4 ms sampling

discarded

- File writes are requested by RPC. When a write is requested for a given beam and time range, the server guarantees that the corresponding data in the ring buffer is preserved in memory, until it can be written to disk.
- If write requests accumulate more quickly than the files can be written, the server will eventually run out of memory. (Desired behaviour: pause dedispersion until there is enough memory. Current behaviour: crash!)

Introduction

- Data can either be written to local SSD or NFS.
- We defined the following goal ("gold standard"). The FRB node should be able to continuously stream one copy of its 2.2 Gbps data stream to SSD, and a parallel copy to disk. This guarantees that the FRB node is never a bottleneck. (The NFS server can still be a bottleneck!)



Network performance tests

Reminder: UDP vs TCP

UDP protocol (packet-based)

- sender chooses transmission speed
- packets are not guaranteed to be delivered
- level of network congestion determines packet drop fraction
- used to send data from L0 to L1.

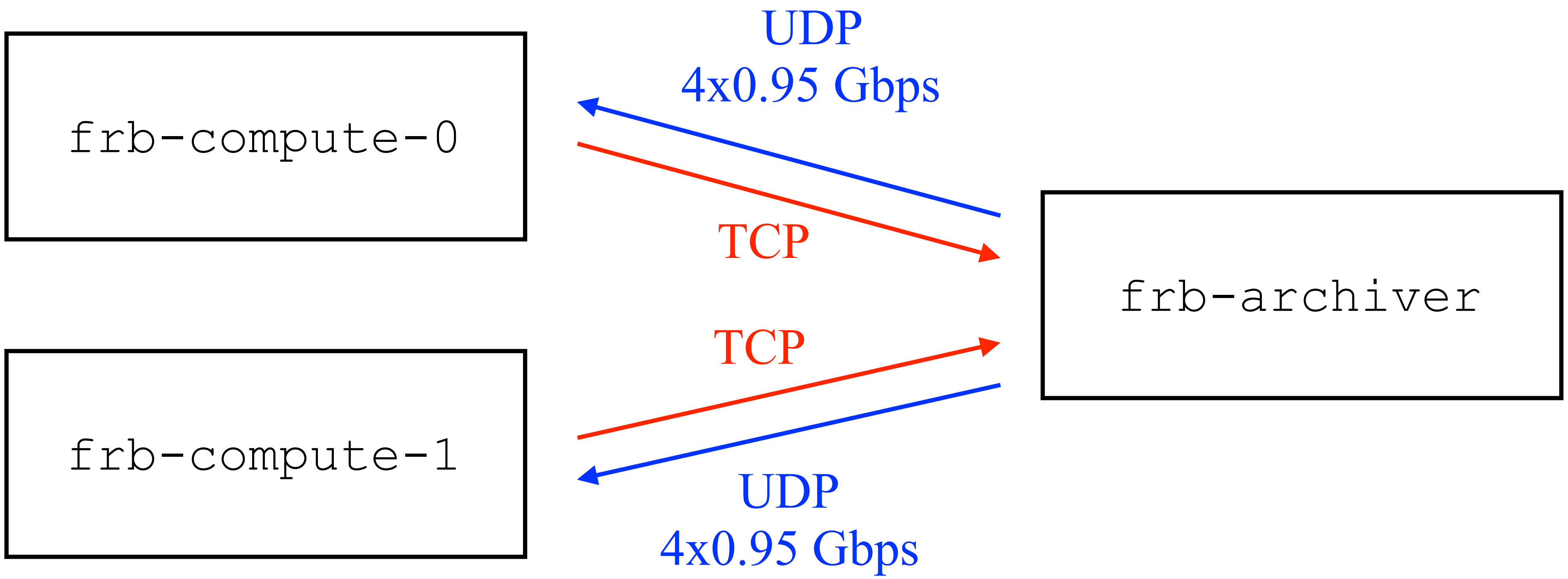
TCP protocol (stream-based)

- stream is guaranteed to be delivered
- level of network congestion determines transmission speed
- used to write data from L1 to file servers.

Network performance tests

Can the nodes write data to a file server at 2.2 Gbps (as part of the "gold standard"), while receiving UDP input data? First consider the following standalone test of the network:

- Send UDP packets from archiver to nodes at 8 x 0.95 Gbps, and measure packet drop fraction.
- Send TCP streams from nodes to archiver, and measure transmission speed

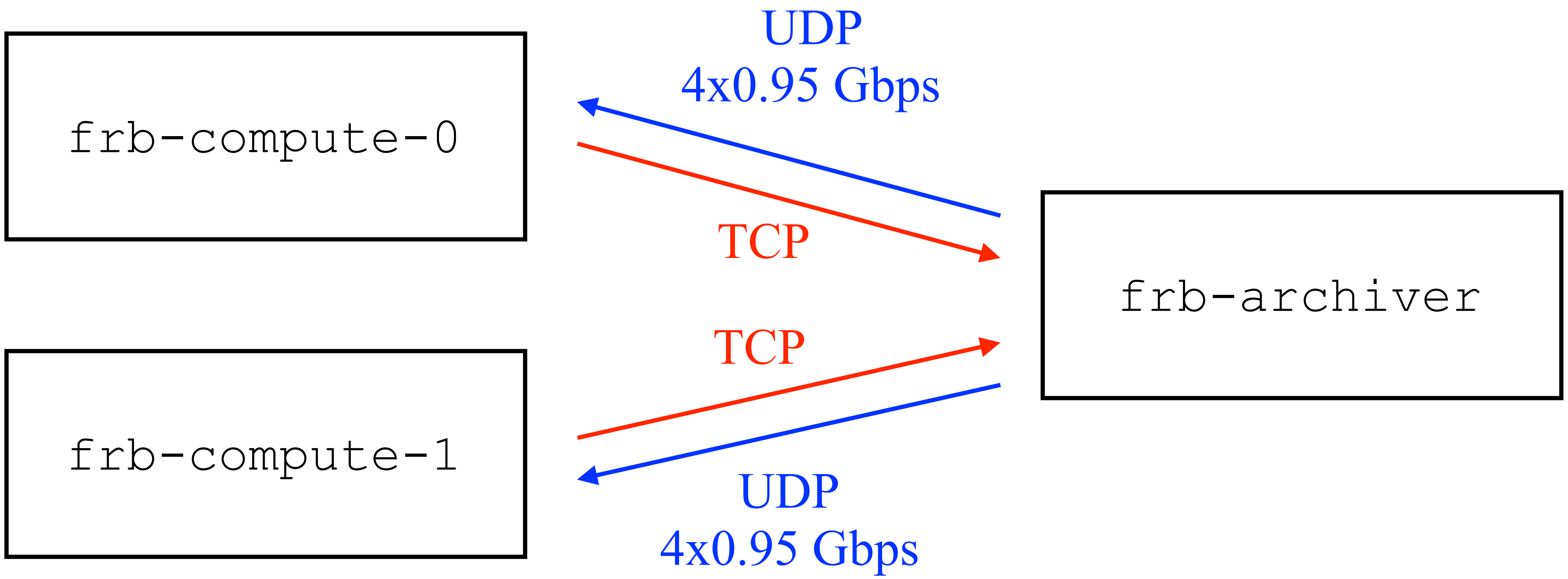


Network performance tests

Result of this test is basically ideal!

- UDP packet loss rate 10^{-5}
- TCP tranmission rate 8 x 0.94 Gbps

… so there is no network bottleneck.



File server performance tests

Next: standalone test of (ZFS) filesystem on file server. Done with "local" threads running on the file server, not over NFS!

Result: it's easy to write data at > 2.2 Gbps if using multiple threads. E.g. with four write threads, get 4 x 1.8 Gbps.

… so no there is no disk bottleneck on the file server.

(Comment: the L1 node has four threads which write to NFS.)

NFS performance tests

Puzzling results here: NFS throughput is significantly lower than either the network throughput or the server filesystem throughput.

- one L1 node, using 1 NIC: 0.7 Gbps
- one L1 node, using 4 NIC's: 4 x 0.47 Gbps
- two L1 nodes, using 2x4 NIC's: 8 x 0.38 Gbps

Strange, since there would seem to be no obstacle to writing our own server-side code which receives data over TCP and streams it to disk… so why can't NFS do it?

Speculate that our NFS server settings are suboptimal somehow..? Maybe we need to learn more about configuring NFS?

SSD performance tests

Each L1 node contains a local 1TB SSD.
Performance tests look great!

- One write thread: 2.8 Gbps (350 MB/s)
- Two write threads: 4 Gbps (500 MB/s).
  This is the theoretical max.

Streaming a full copy of the data (2.2 Gbps) to local SSD
is no problem.

Compression

- It would be nice to implement compression!
- Ballpark estimate: should reduce data size by ~1.5 (entropy calculation assuming Gaussian intensities clipped at 3 sigma)
- Problem: the L1 server thread model is based on the assumption that one dedicated core is be enough for all file I/O.
- In "gold standard" case, actual cost turned out to be ~0.5 cores with compression turned off, or ~3(!) cores with compression on
- Can probably tweak thread model to accommodate, but we decided to deprioritize implementing compression.
- Plan to revisit later. It will help to have some real data, so that we can evaluate tradeoffs between algorithms.

L1 thread model

core 0: dedispersion
core 1: dedispersion
core 2: dedispersion
core 3: dedispersion
core 4: dedispersion
core 5: dedispersion
core 6: dedispersion
core 7: dedispersion
core 8: packet assembly
core 9: file I/O

CPU1

core 10: dedispersion
core 11: dedispersion
core 12: dedispersion
core 13: dedispersion
core 14: dedispersion
core 15: dedispersion
core 16: dedispersion
core 17: dedispersion
core 18: packet assembly
core 19: unused!

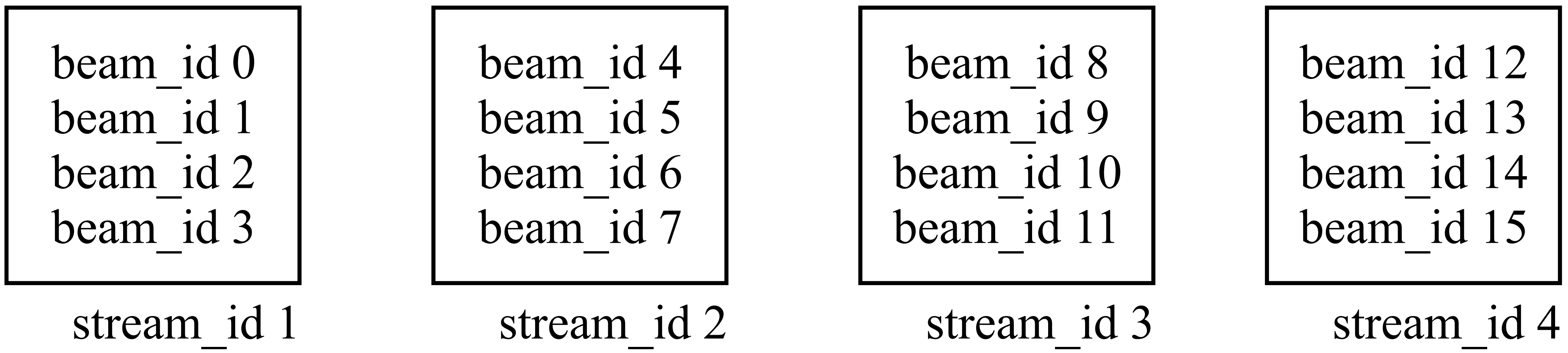
CPU2

L1 file-writing code

- The L1 server uses one I/O thread per “output device”
- We currently define five output devices (4 NIC’s + 1 SSD):

```
/frb-archiver-1
/frb-archiver-2
/frb-archiver-3
/frb-archiver-4
/ssd
```

- As previously mentioned, the L1 server receives 4 beams on each of its 4 NIC’s. Each such group of 4 beams has an internal “stream_id” between 1-4. This detail will be important shortly!



beam_id 0
beam_id 1
beam_id 2
beam_id 3

stream_id 1

beam_id 4
beam_id 5
beam_id 6
beam_id 7

stream_id 2

beam_id 8
beam_id 9
beam_id 10
beam_id 11

stream_id 3

beam_id 12
beam_id 13
beam_id 14
beam_id 15

stream_id 4

L1 file-writing code

- A write_request RPC contains a list of beams and a timestamp range. Output files ("chunks") are always ~16MB in size. A single RPC can trigger many file writes!
- Instead of a filename, the RPC contains a filename "pattern":

```
/ssd/kmsmith/chunk_(BEAM)_(CHUNK)_(NCHUNK).msg
```

- Filenames are derived from the pattern by substitution, e.g.

```
/ssd/kmsmith/chunk_0008_00000137_01.msg
/ssd/kmsmith/chunk_0008_00000138_01.msg
/ssd/kmsmith/chunk_0008_00000139_01.msg
```

- The same pattern can be used for many (or all) write_requests.
- Some useful substitutions (not a complete list, see ch_frb_io.hpp):

```
(BEAM)    -> beam_id
(STREAM)  -> stream_id between 1-4
(CHUNK)   -> chunk index
               (1 chunk = 384*1024 FPGA counts ~ 1 sec)
(NCHUNK)  -> level of downsampling in telescoping ring buffer
               (NCHUNK=1 corresponds to no downsampling)
```

L1 file-writing code

- As previously mentioned, the L1 node "thinks" the NFS server is four different file servers:

```
/frb-archive-1
/frb-archive-2
/frb-archive-3
/frb-archive-4
```

- I/O on each of these filesystems uses a different 1 Gbps NIC, and the L1 node is responsible for load-balancing.
- A convenient way to do this is to use the stream_id in the filename_pattern:

```
/frb-archiver-(STREAM)/kmsmith/chunk_(BEAM)_(CHUNK)_(NCHUNK).msg
```

- This will automatically put beams 0-3 on NIC1, beams 4-7 on NIC2, etc.

L1 file-writing code

The write_chunk RPC in pseudocode:

```
for each ~16MB data chunk:
    expand filename_pattern to filename
    if the same (chunk, filename) pair was written previously:
        return (i.e. do nothing)
    elif the filename already exists on disk:
        fail with an error
    elif the chunk was previously written to a different filename on the same device:
        create a unix hard link
    else:
        create a new file
```

Notes:

- we don’t create new directories (caller must do this)
- writing different chunks to same filename is an error
- writing same chunk to different filenames creates hard links

L1 file-writing code

Pitfall #1: not enough substitutions in the filename_pattern.

```
/ssd/kmsmith/chunk_(BEAM)_(CHUNK)_(NCHUNK).msg
```

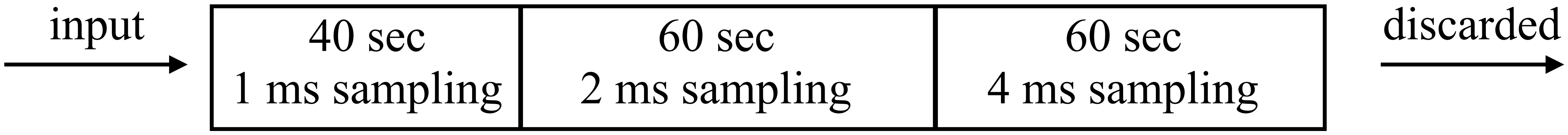
is OK, but

```
/ssd/kmsmith/chunk_(BEAM)_(CHUNK).msg
```

is not OK (can get filename collisions between chunks in different levels of the telescoping ring buffer)

L1 file-writing code

Pitfall #2: writing multiple copies of the data at different levels of downsampling. Suppose we send an RPC which spans the "2ms" part of the telescoping ring buffer.



If the 2ms chunks have previously been written, new copies will not be created (a hard link will be created instead).

However, if the same timestamp range has previously been written at 1ms, a new copy of the data will be created at 2ms.

Can double data rate if not careful.

Summary

- File-writing code is in reasonable shape now.
- Getting good performance (but not quite “gold standard”).
- Limiting factor seems to be NFS. Can we tune the NFS server better?
- Compression not working yet; we’ll revisit this soon.