Part 2:
Data Delivery methods

RCDAQ knows a number of "data delivery" methods – I sometimes call this "data disposal"

The by far most known method is writing to a file.

Interestingly, that was not what happened in PHENIX. Yes, eventually the data ended up in files, but were not directly written by the DAQ.

File writing was also **not** the envisioned main method for sPHENIX, either. Only the advent of the Lustre file system on our buffer boxes made direct file writing by RCDAQ competitive again just in time in 2022.

I'll show the methods to "get data":

- File writing
- Socket writing
- Online monitoring streams

Why is this interesting for us?

With the streaming readout in ePIC, we will need to get data from many DAQ nodes to a central place for soft-triggering

This can work from “localhost” to the LAN to the WAN

The socket-writing protocol that is implemented in RCDAQ and has a number of back-ends to receive the data

It’s an efficient protocol that goes back to my CERN days at the SPS when 10Mbit/s networks were common and state-of-the-art

PHENIX (not sPHENIX) used a back-end that already looks like the “data concentrator” we might need, minus a few small enhancements

First: File writing

Virtually all bench top/test beam setups use direct file writing. Easy, versatile, and (usually) reasonably fast.

The default on a just-started RCDAQ:

```
$ daq_status -l
mlpvm5 -  Stopped
  Filerule:     rcdaq-%08d-%04d.evt
  Logging enabled
```

What's that file rule? It's a c-printf control string that takes two integers, the run number and the file sequence number. The "%08d" makes 8-digit, zero-padded numbers to avoid spaces, here shown in ROOT:

```
root [0] printf ("rcdaq-%08d-%04d.evt\n", 2,  0);
rcdaq-00000002-0000.evt
```

```
$ daq_status -l
mlpvm5 -  running
  Run Number:     2
  Event:          206
  Run Volume:     0.0884323 MB
  Filerule:       rcdaq-%08d-%04d.evt
  Filename:       rcdaq-00000002-0000.evt
```

The sequence number comes in when we roll over the file when a certain size is reached, if selected

File Rules

The file rules can be anything we want, as long as they are generating valid file names.

One typically uses a fully qualified file name

Here shown for an actual sPHENIX component, with a 20GB rollover value set:

```
phnxrc@gl1daq:~$ daq_status -l
gl1daq -  running
  Run Number:     73087
  Event:          15708048
  Run Volume:     30679.8 MB
  Filerule:       /bbox/bbox10/W/GL1/junk/GL1_junk_gl1daq-%08d-%04d.evt
  File rollover:  20GB
  Filename:       /bbox/bbox10/W/GL1/junk/GL1_junk_gl1daq-00073087-0001.evt
```

Run types == pre-canned file rules

The file rule can be changed at any time and is used when the next run starts.

It is usually easier to use “run types”. They are predefined, named file rules that allow you to switch from one to another

```
-- defined Run Types:
         beam - /bbox/bbox10/W/GL1/beam/GL1_beam_gl1daq-%08d-%04d.evt
        calib - /bbox/bbox10/W/GL1/calib/GL1_calib_gl1daq-%08d-%04d.evt
      cosmics - /bbox/bbox10/W/GL1/cosmics/GL1_cosmics_gl1daq-%08d-%04d.evt
       dryrun - /bbox/bbox10/W/GL1/dryrun/GL1_dryrun_gl1daq-%08d-%04d.evt
         junk - /bbox/bbox10/W/GL1/junk/GL1_junk_gl1daq-%08d-%04d.evt
   line_laser - /bbox/bbox10/W/GL1/line_laser/GL1_gl1daq_line_laser-%08d-%04d.evt
     pedestal - /bbox/bbox10/W/GL1/pedestal/GL1_pedestal_gl1daq-%08d-%04d.evt
      physics - /bbox/bbox10/W/GL1/physics/GL1_physics_gl1daq-%08d-%04d.evt
```

The “junk” rule was the one active on the last slide (on Tuesday night with no beam)

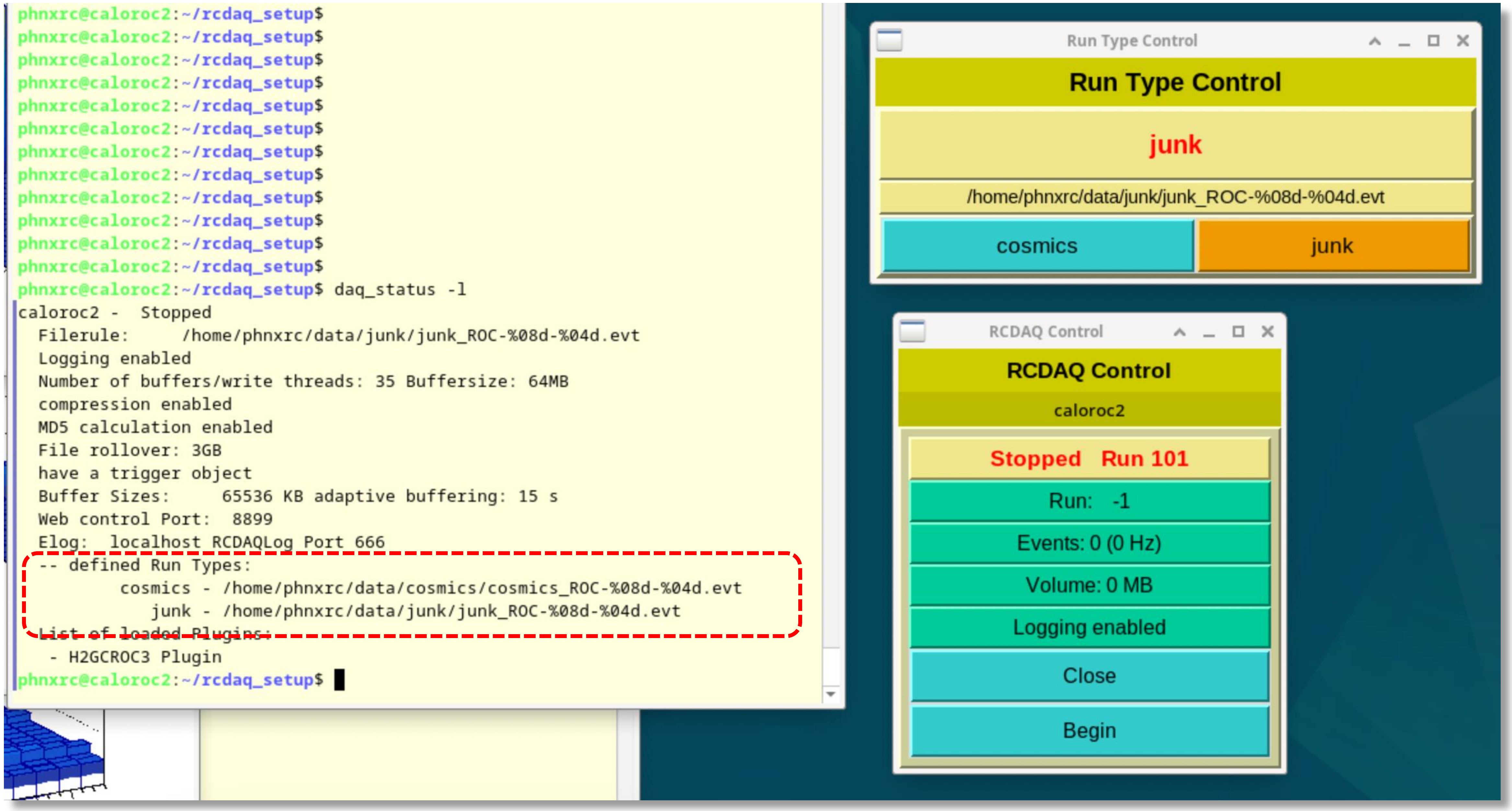
That was a sPHENIX component, let me show the Barrel HCal setup…

The Bldg 510 BHCal setup

Here we have only 2 run types, “junk” and ”cosmics”

Also shown is a “run type chooser” GUI (done by RunControl if one runs this way)

Rollover is set to 3GB here



```
phnxrc@caloroc2:~/rcdaq_setup$
phnxrc@caloroc2:~/rcdaq_setup$
phnxrc@caloroc2:~/rcdaq_setup$
phnxrc@caloroc2:~/rcdaq_setup$
phnxrc@caloroc2:~/rcdaq_setup$
phnxrc@caloroc2:~/rcdaq_setup$
phnxrc@caloroc2:~/rcdaq_setup$
phnxrc@caloroc2:~/rcdaq_setup$
phnxrc@caloroc2:~/rcdaq_setup$
phnxrc@caloroc2:~/rcdaq_setup$
phnxrc@caloroc2:~/rcdaq_setup$
phnxrc@caloroc2:~/rcdaq_setup$
phnxrc@caloroc2:~/rcdaq_setup$ daq_status -l
caloroc2 -  Stopped
  Filerule:     /home/phnxrc/data/junk/junk_ROC-%08d-%04d.evt
  Logging enabled
  Number of buffers/write threads: 35 Buffersize: 64MB
  compression enabled
  MD5 calculation enabled
  File rollover: 3GB
  have a trigger object
  Buffer Sizes:     65536 KB adaptive buffering: 15 s
  Web control Port:  8899
  Elog:  localhost RCDAQLog Port 666
  -- defined Run Types:
          cosmics - /home/phnxrc/data/cosmics/cosmics_ROC-%08d-%04d.evt
             junk - /home/phnxrc/data/junk/junk_ROC-%08d-%04d.evt
  List of loaded Plugins:
   - H2GCROC3 Plugin
phnxrc@caloroc2:~/rcdaq_setup$
```

Run Type Control

Run Type Control

junk

/home/phnxrc/data/junk/junk_ROC-%08d-%04d.evt

RCDAQ Control

RCDAQ Control

caloroc2

Stopped   Run 101

Run: -1

Events: 0 (0 Hz)

Volume: 0 MB

Logging enabled

Let’s move away from files to more interesting things

File writing is just one way to “dispose” of data.

Let’s move to sockets.

Pretty much everything that RCDAQ moves around is called a “buffer”

Why a “super structure”?

Events (or streaming-data chunks) are not large enough to make efficient transfers/writes

We need something bigger…

Depending on setup, we typically stuff between 50 and 2000+ events into a buffer

Buffers, not events – the high-level units in the data format

Buffers are the “moving units” of our data format.

- Buffers are self-contained entities
- Data compression happens at the buffer level
- Data files are just a concatenation of such buffers
- It is possible to mix different types of buffers in a file (as they are independent units), say, compressed and non-compressed buffers
- If file1.evt and file2.evt are valid files, then `cat file1.evt.file2.evt > file3.evt` gives a valid file
- The online monitoring stream hands out buffers. The events inside a buffer are usually contiguous, what the online monitoring often needs
- A buffer can be 4GB in size. That drives the max event size of 4GB – 56 bytes (headers) (we have max event sizes in the few MB range)

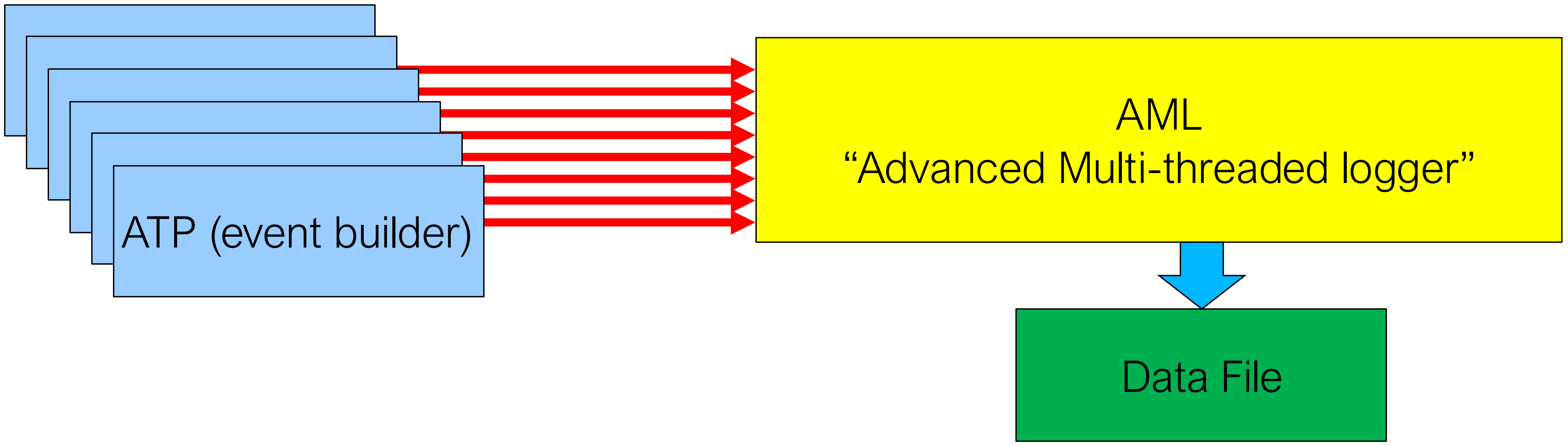
Buffer disposal in PHENIX

RCDAQ can use a more generic method for the buffer disposal

One can specify a host and (optional) port number

This implements a lightweight protocol with a handshake to transfer buffers on a network socket

In **PHENIX** that was the main method to deliver the data:



You are familiar with servers that fork() themselves, such as the sshd, for each incoming connection

The AML, instead of fork()’ing, generates a new thread to deal with a new connection

B/c it is still one process, the AML could write the incoming buffers (with fully built events) from many ATPs into one file

What I had initially planned for sPHENIX

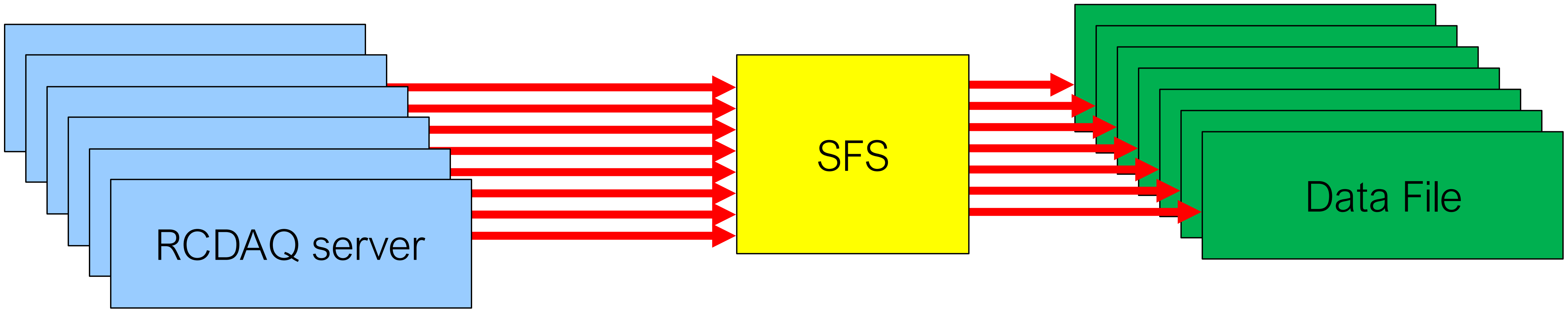
With no event builder, each RCDAQ instance makes its own file, combined offline later

Naively, you'd NFS-mount our buffer boxes' file systems on the RCDAQ machines, open files…

No good! The NFS performance is miles away from where you need it

Enter the sibling of the AML, the "Super-Fast Server" (sfs)

Sending the data via the same protocol as the AML, an RCDAQ->sfs network protocol beats NFS any day



The sfs is a classic "fork()'ing" server, was on track to be the main delivery method…

But then, in 2022, the Lustre file system came along, which is the ultimate high-performance file system

Lustre gives sfs-level performance without the hassle of dealing with many servers, we switched to that instead

The “meat” of the protocol

There are other calls, but this is the actual data transfer sequence

```
int opcode = htonl(CTRL_DATA);
int status = writen(fd, (char *) &opcode, sizeof(int));
if (status)
 {
   // error handling
 }
int len = htonl(total);
status |= writen(fd, (char *) &len, sizeof(int));

if (status)
 {
   // error handling
 }
// now send the actual data
char *p = (char *) bptr;
int sent = writen(fd,p,total);
```

“I want to send you data, ok?”

This many bytes, are you ready?

… and here they come

Then there is an acknowledge handshake, with a sort-of user-defined meaning

In the AML and sfs, the acknowledgement meant not just that the data were received, but that they were actually safe on disk (think of a situation where the disk runs full - no ACK)

FNAL Test beam data delivered directly to the SDCC

In 2017 or so we finessed a sfs-based data delivery from the FermiLab FTBF directly to the SDCC

We tunneled a port to the SDCC with a sfs listening on it…

```
rcdaq_client daq_set_server localhost 5001
```

With 4s beam out of a 60s cycle at the FTBF there was enough time to dispose of the data, no problem

But soon we got out of it –

- The tunnel takes someone's personal account to log in to the SDCC
- We also found that everyone interested in the data for a quick turnaround happened to be at FNAL, so we switched to logging the data there, and transferred later

But it worked great!

The sfs lives on…

The socket delivery method is and will remain an integral part of RCDAQ.

The sfs is still used in a niche for our TPC, for the "line laser" system that makes adjustable tracks any desired way through the gas volume.

Small dedicated DAQ in the IR to record the properties of a given laser pulse

That DAQ has no access to the Lustre systems, so the sfs it is:

```
# we are using the sfs
rcdaq_client daq_set_server bbox6.sphenix.bnl.gov 5001
```

```
phnxrc@tpclasercontrol:~$ daq_status -l
tpclasercontrol -  Stopped
  Filerule:     /data/phnxrc/diode/12S_4D/align7/Run-%08d-%04d.evt
  Logging enabled via remote server bbox6.sphenix.bnl.gov Port 5001
```

```
[phnxrc@bbox6 ~]$ ps ax | grep sfs
3548571 pts/1    S+     3:20 sfs -b eno8303 -p 5001
```

Briefly: the online monitoring stream

It is a socket, TCP-based mechanism similar to the server concept

The difference is that this delivery is designed to be “opportunistic”

Meaning that the client is not allowed to hold up the data taking and must tolerate missing events

It works by RCDAQ maintaining a list of online monitoring clients

They can register and de-register dynamically at any time, arbitrary number of clients

When a buffer is ready to go, RCDAQ traverses the list and sees if any client has expressed interest in this buffer

If not (when that client is still crunching the last buffer or is otherwise slow or has even stopped), we move on – this client will miss that buffer.

What’s in it for ePIC

With the streaming readout in ePIC, we will need to get data from many DAQ nodes to a central place for soft-triggering

This can work from “localhost” to the LAN to the WAN

A “receiver” (similar to the sfs, more likely like the AML) can be developed that receives the various data streams, combines them by Beam Crossing, and takes a decision to pass this crossing on for logging,or not

That sounds a lot like the core of the AML…

And the online monitoring is the same rock-solid protocol, at work in 1008 for 80+ nodes 24/7…