The Event Pool
and
Online Histograming Tutorial

Two Parts:

- Event Pool Mechanics (myself)
- Online QA Histograms (Sergey Panitkin)

Event Pool Mechanics

What is the Event Pool (EVP)?

1) **A Sun 450 Enterprise computer in the DAQ room at the experiment with 4 SPARC CPUs, 1.5 GB RAM and 350 GB of disk running Solaris 2.8 (with a 32 bit kernel) called evp.star.bnl.gov**

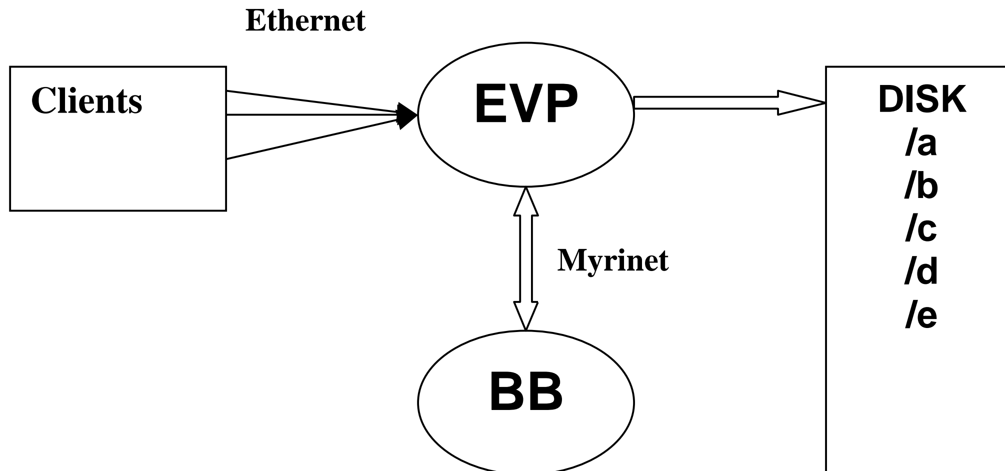
2) **The cached ("pooled") set of events sampled from the DAQ data stream and stored locally on an accessible disk for a short time period ("The Event Pool Store")**

3) **The mechanism of getting "fresh" events served directly from the ongoing run ("The Event Pool Reader").**

4) **A set of libraries which support detector specific raw data unpacking ("The Detector Readers")**

5) **The organization of it all…**

Organization



Store

- A “run” is a directory with the run’s number as the name (i.e. “1234001”)
- An event is a single file within that directory with its sequence number as the name (i.e. “1”, “212”,…)
- Thus:

```
evp.star.bnl.gov                    node name
   /a                               file system
        1234001                     directory
              1                     file/event
              2
              …
        1235012
              1
              …
   /b
   /c
   /d
   /e
```

- All the disks (file systems) are exported READ_ONLY to *.star.bnl.gov via NFS
- Runs may get renamed to "*_DELETE" if the Run Control Operator marks the run as "don't keep" upon end-of-run in which case a cron job will zap them in the night (or so…)
- Runs will get deleted without prior notice when EVP runs out of space…

Requests from the Live Feed

- The clients communicate with a separate task on evp.star via TPC/IP and not directly with DAQ
- The events get stored only if there is a current requestor → no requests == no events stored
- The event requests get pooled so multiple requests may end up getting one single event which will be served to all of them → efficient from DAQ's perspective…
- The requestor only receives the file name of the event from the EVP – the actual data gets transferred via NFS at the clients discretion thus : you must be able to mount evp.star via NFS!
- The requests can be by "type", currently any combination of:
  - TOKEN_0 (i.e. pedestals, RMSes)
  - PHYSICS (trigger command 4)
  - SPECIAL (trigger command not 4)
  - ANY (any of the above)

- There is a set of #includes and a library which does all of that (and more) for Solaris and Linux ➔ libevp.a

evpReader

- A **`"class evpReader"`** is all you need to get events from either: live feed, cached EVP runs (aka directories) or standard DAQ raw data files
- Supported in binary for Linux & Solaris

Digression: code organization

- All the relevant stuff is on daqman.star.bnl.gov in the directory /RTS which is exported via NFS to *.star.bnl.gov → you must have access to this directory if you plan to use the evpReader interface.

Relevant parts:

```
/RTS/include/EVP/evpReader.hh      class declaration
/RTS/lib/SUN/libevp.a
/RTS/lib/LINUX/libevp.a            precompiled libraries
/RTS/src/EVP_READER                sources directory
/RTS/src/EVP_READER/special.C           example code
/RTS/src/EVP_READER/Makefile.special    makes "special"
```

!!! PLEASE LOOK IN evpReader.hh AND special.C !!!

Example usage:

```
#include <EVP/evpReader.hh>

…

class evpReader *evp = new evpReader(NULL) ;
          or
class evpReader *evp = new evpReader(“/evp/a/1234001”) ;
          or
class evpReader *evp = new evpReader(
 “st_pedestal_1234001_raw_001.daq”) ;

FOREVER {       // loop forever

char *mem = evp->get(0,EVP_TYPE_ANY) ;
if(mem == NULL) {                    // event NOT valid!!!
     switch(evp->status) {
     case EVP_STAT_OK :
          // event is not yet available - repeat get!
          continue;       // back to FOREVER
     case EVP_STAT_EVT :
          // event is bad - repeat get and print error msg.
          continue ;      // back to FOREVER
     case EVP_STAT_EOR :
          // end-of-file/run signaled - up to you…
          continue ;      // back to FOREVER
     case EVP_STAT_CRIT :
          // really bad error - shouldn't happen - call me
          exit(-1) ;      // nothing you can do anymore…
     }
}

// mem is valid (non-NULL) and it actually
// points to the beginning of the event. At this stage the
// event is memory mapped to your process'es virtual memory

char *datap = datapReader(mem) ;
// datap now points to the beginning of the DATAP structure
// this step is necessary if you plan to use the new
// detector unpackers/readers
```

```
// an example detector reader/unpacker
int ret = tpcReader(datap) ;

if(ret < 0) {
     // bad event, error in unpacking etc.
} else if (ret == 0) {
     // no TPC in this event
} else {
     // do your stuff here…
}

} // end FOREVER
```

Detector Readers/Unpackers

- Unpack raw DAQ data to detector specific, humanly usable format
- All exist but most of them just check the data headers for consistency only and return the total size in bytes
- Currently implemented: TPC, SVT, FTPC & TRG (parts of it…)
- All xxxReaders use statically allocated storage (malloc/new not used) which gets overwritten (reused) upon every call
- TPC example (con't from above):

```
…
if(ret > 0) {       // see example above…
     // the return value is the length of the whole TPC
     // contribution in bytes

     printf("TPC found with length of %d bytes\n",ret) ;

     for(s=0;s<24;s++) {            // sectors
     for(r=0;r<45;r++) {            // rows
     for(p=0;p<182;p++) {           // pads
     for(cou=0;cou<tpc.counts[s][r][p];cou++) {
          uchar adc8 = tpc.adc[s][r][p] ;
          ushort timebin = tpc.timebins[s][r][p] ;
     }}}}
```

```
        // NOTE: sectors, rows, pads & timebins start from 0
        // NOTE: the ADC data is 8 bit - not 10 bit
}
```

Conclusion

- Look in the code example `"special.C"` as well as the class declaration `"evpReader.hh"` and try to understand the details yourself – it's not so hard, ☺
- Support from me will be very limited in the next few months but I'll try – use the RTS mail server for questions (contact me if you want to be on the maling list…)
- Some detector readers (i.e. `tpcReader`) use a lot of memory at startup (about 300 MB in TPC's case) which may cause bizarre problems under Linux systems which have less than 500 MB (or so…) of RAM+swap configured