

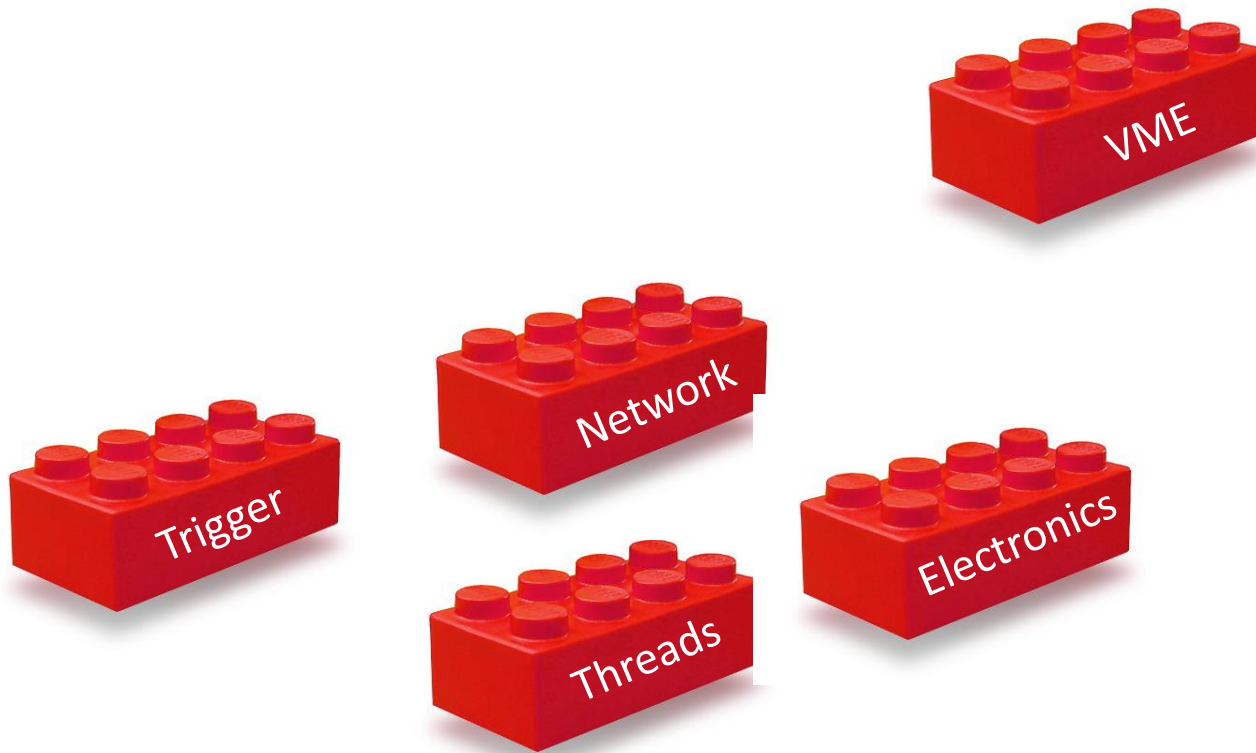


DAQ software

E. Pasqualucci
INFN Roma



You saw many bricks up to now...





... and you will see some cathedrals ...





... but if you want to build a cathedral from bricks you have to start this way ...



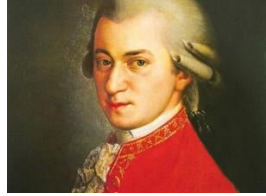


Overview

- Aim of this lecture is
 - Give an overview of a medium-size DAQ
 - Starting from the general picture given by A. Negri
 - Analyze its components
 - Using the concepts introduced by previous lectures
 - Introduce the main concepts of DAQ software
 - As “bricks” to build larger system
 - ... with the help of some pseudo-code ...
 - Give more technical basis
 - For the implementation of larger systems
 - See R. Ferrari’s and F. Pastore’s lectures

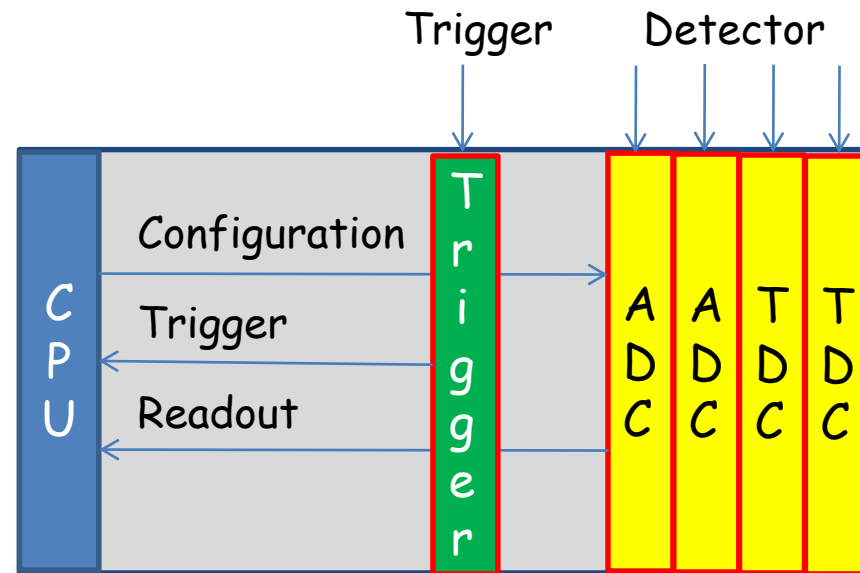


Software components



- Trigger management
- Data read-out
- Event framing and buffering
- Data transmission
- Event building and data storage
- System control and monitoring
- Data sampling and monitoring

Data readout (a simple example)



- Data digitized by VME modules (ADC and TDC)
- Trigger signal received by a trigger module
 - I/O register or interrupt generator
- Data read-out by a Single Board Computer (SBC)



Trigger management

- How to know that new data is available?
 - Interrupt
 - An interrupt is sent by an hardware device
 - The interrupt is
 - Transformed into a software signal
 - Caught by a data acquisition program
 - » Undetermined latency is a potential problem!
 - » Data readout starts
 - Polling
 - Some register in a module is continuously read out
 - Data readout happens when register “signals” new data
- In a synchronous system (the simplest one...)
 - Trigger must also set a busy
 - The reader must reset the busy after read-out completion



Managing interrupts

```
irq_list.list_of_items[i].vector = 0x77;  
irq_list.list_of_items[i].level  = 5;  
irq_list.list_of_items[i].type   = VME_INT_ROAK;  
signum = 42;  
  
ret = VME_InterruptLink(&irq_list, &int_handle);  
ret = VME_InterruptWait(int_handle, timeout, &ir_info);  
ret = VME_InterruptRegisterSignal(int_handle, signum);  
ret = VME_InterruptUnlink(int_handle);
```

Real time programming



- Has to meet operational deadlines from events to system response
 - Implies taking control of typical OS tasks
 - For instance, task scheduling
 - Real time OS offer that features
- Most important feature is predictability
 - Performance is less important than predictability!
- It typically applies when requirements are
 - Reaction time to an interrupt within a certain time interval
 - Complete control of the interplay between applications

Is real-time needed?



- Can be essential in some case
 - May be critical for accelerator control or plasma control
 - Wherever event reaction times are critical
 - And possibly complex calculation is needed
- Not commonly used for data acquisition now
 - Large systems are normally asynchronous
 - Either events are buffered or de-randomized in the HW
 - Performance is usually improved by DMA readout (see M. Joos)
 - Or the main dataflow does not pass through the bus
 - In a small system dead time is normally small
- Drawbacks
 - We loose complete dead time control
 - Event reaction time and process scheduling are left to the OS
 - Increase of latency due to event buffering
 - Affects the buffer size at event building level
 - Normally not a problem in modern DAQ systems



Polling modules

- Loop reading a register containing the latched trigger

```
while (end_loop == 0)
{
    uint16_t *pointer;
    volatile uint16_t trigger;

    pointer = (uint16_t *) (base + 0x80);
    trigger = *pointer;

    if (trigger & 0x200) // look for a bit in the trigger mask
    {
        ... Read event ...
        ... Remove busy ...
    }
    else
        sched_yield (); // if in a multi-process/thread environment
}
```



Polling or interrupt?

- Which method is convenient?
- It depends on the event rate
 - Interrupt
 - Is expensive in terms of response time
 - Typically ($O(1 \mu s)$)
 - Convenient for events at low rate
 - Avoid continuous checks
 - A board can signal internal errors via interrupts
 - Polling
 - Convenient for events at high rate
 - When the probability of finding an event ready is high
 - Does not affect others if scheduler is properly released
 - Can be “calibrated” dynamically with event rate
 - If the input is de-randomized...



The simplest DAQ

- Synchronous readout:
 - The trigger is
 - Auto-vetoed (a busy is asserted by trigger itself)
 - Explicitly re-enabled after data readout
- Additional dead time is generated by the output

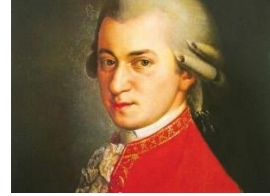
```
// VME interrupt is mapped to SYSUSR1
```

```
static int event = FALSE;  
const int event_available = SIGUSR1;
```

```
// Signal Handler
```

```
void sig_handler (int s)  
{  
    if (s == event_available)  
        event = TRUE;  
}
```

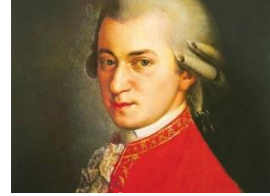
```
event_loop ()  
{  
    while (end_loop == 0) {  
        if (event) {  
            size += read_data (*p);  
            write (fd, ptr, size);  
            busy_reset ();  
            event = FALSE;  
        }  
    }  
}
```

Fragment buffering

- Why buffering?
 - Triggers are uncorrelated
 - Create internal de-randomizers
 - Minimize dead time
 - See Andrea's and Francesca's lecture
 - Optimize the usage of output channels
 - Disk
 - Network
 - Avoid back-pressure due to peaks in data rate
 - Warning!
 - Avoid copies as much as possible
 - Copying memory chunks is an expensive operation
 - Only move pointers!

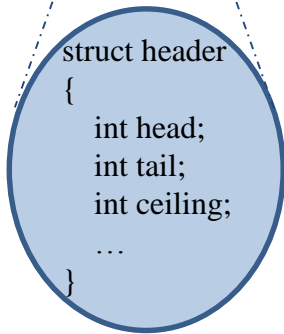
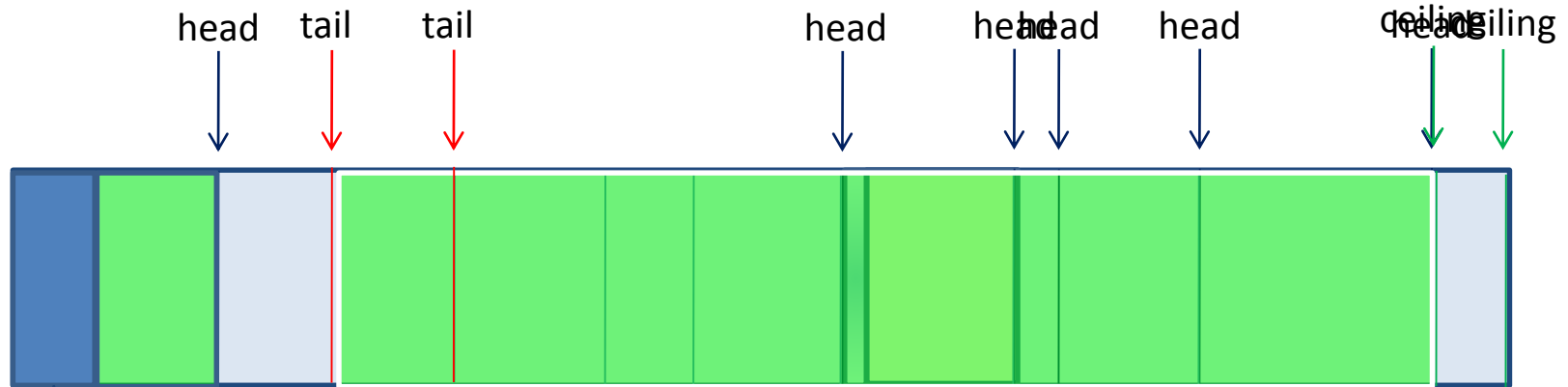
A simple example...



- Ring buffers emulate FIFO
 - A buffer is created in memory
 - Shared memory can be requested to the operating system
 - A “master” creates/destroys the memory and a semaphore
 - A “slave” attaches/detaches the memory
 - Packets (“events”) are
 - Written to the buffer by a writer
 - Read-out by a reader
 - Works in multi-process and multi-thread environment
 - Essential point
 - Avoid multiple copies!
 - If possible, build events directly in buffer memory



Ring buffer



- The two processes/threads can run concurrently
 - Header protection is enough to insure event protection
 - A library can take care of buffer management
 - A simple API is important
 - We introduced protected pointers
 - Shared memories provided by OS
 - Buffer protection (semaphores or mutexes)
 - Buffer and packed headers (managed by the library)

Event buffering example



- Data collector

```
int cid = CircOpen (NULL, Circ_key, size));
while (end_loop == 0) {
    if (event) {
        int maxsize = 512;
        char *ptr; uint32_t *p; uint32_t *words;
        int number = 0, size = 0;

        while ((ptr = CircReserve (cid, number,
                                   maxsize)) == (char *) -1)
            sched_yield ();

        p = (int *) ptr;
        *p++ = crate_number; ++size;
        *p++; words = p; ++size;
        size += read_data (*p);
        *words = size;
        CircValidate (cid, number, ptr,
                      size * sizeof (uint32_t));
        ++number;

        busy_reset ();
        event = FALSE;
    }
    sched_yield ();
}
CircClose (cid);
```

- Data writer

```
int fd, cid;

fd = open (pathname, O_WRONLY | O_CREAT);
cid = CircOpen (NULL, key, 0));

while (end_loop == 0)
{
    char *ptr;

    if ((ptr = CircLocate (cid, &number,
                           &evtsize)) > (char *) 0)
    {
        write (fd, ptr, evtsize);
        CircRelease (cid);
    }

    sched_yield ();
}

CircClose (cid);
close (fd);
```

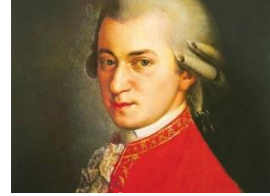
Find next event
Release the scheduler
Reset the busy
Release the buffer

By the way...



- In these examples we were
 - Polling for events in a buffer
 - Polling for buffer descriptor pointers in a queue
 - We could have used
 - Signals to communicate that events were available
 - Handlers to catch signals and start buffer readout
- If a buffer gets full
 - Because:
 - The output link throughput is too small
 - There is a large peak in data rate
 - ⇒ The buffer gets “busy” and generates back-pressure
 - ⇒ Thresholds must be set to accommodate events generated during busy transmission when redirecting data flow
- These concepts are very general...

Event framing

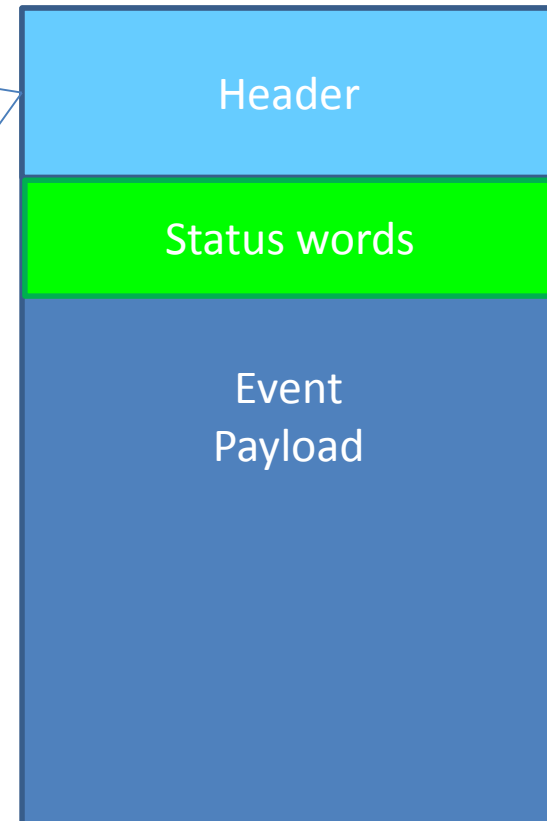


- Fragment header/trailer
- Identify fragments and characteristics
 - Useful for subsequent DAQ processes
 - Event builder and online monitoring tasks
 - Fragment origin is easily identified
 - Can help in identifying sources of problems
 - Can (should) contain a trigger ID for event building
 - Can (should) contain a status word
- Global event frame
 - Give global information on the event
- Very important in networking
 - Though you do not see that
 - See networking lecture

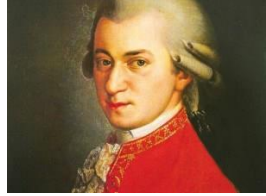


Framing example

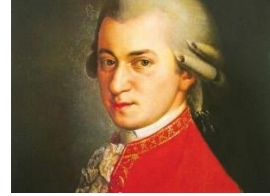
```
typedef struct
{
    u_int startOfHeaderMarker;
    u_int totalFragmentsize;
    u_int headerSize;
    u_int formatVersionNumber;
    u_int sourceIdentifier;
    u_int numberOfStatusElements;
} GenericHeader;
```



What can we do now....



- We are now able to
 - Build a readout (set of) application(s) with
 - An input thread (process)
 - An output thread (process)
 - A de-randomizing buffer
 - Let's elaborate a bit...

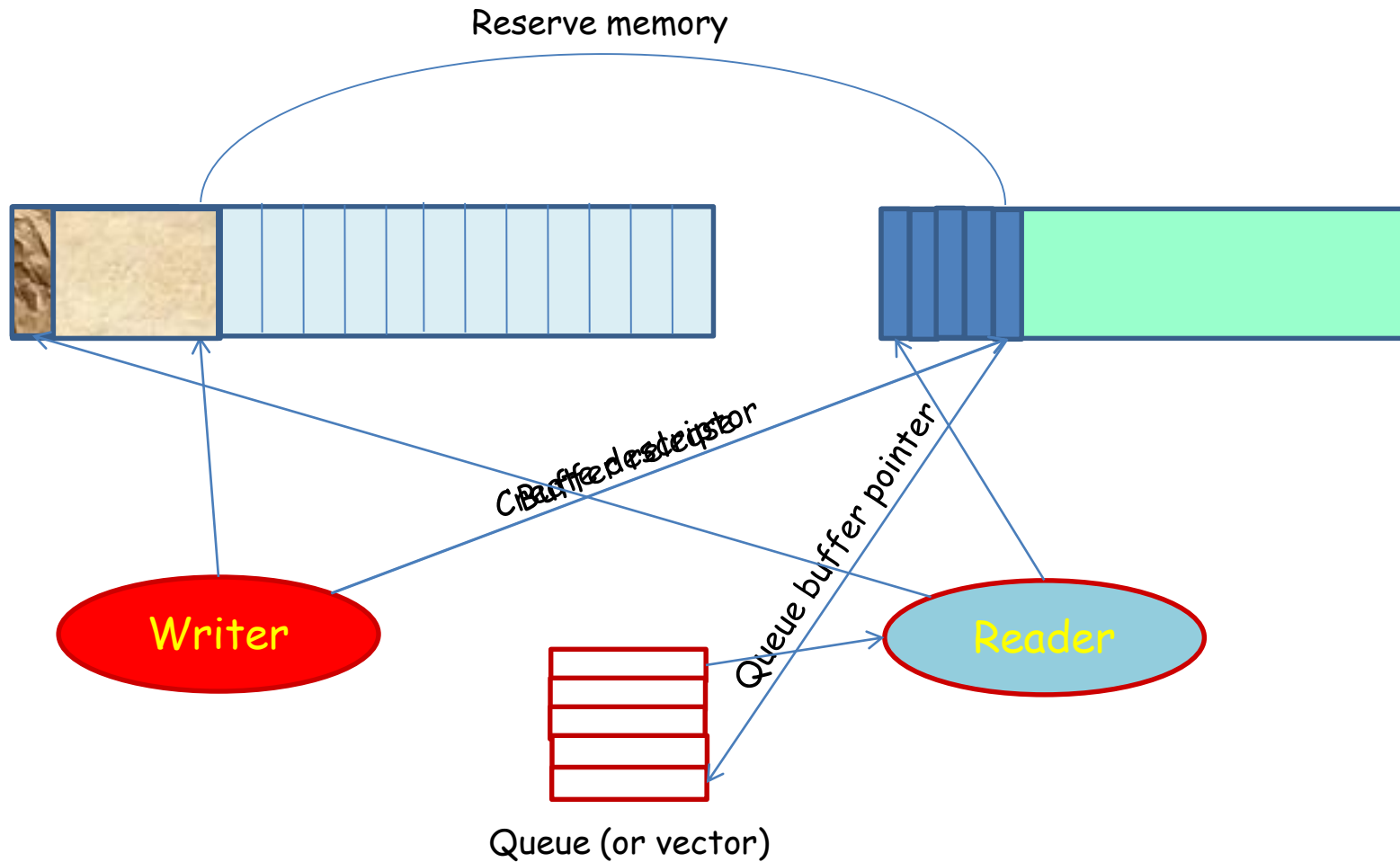


A more general buffer manager

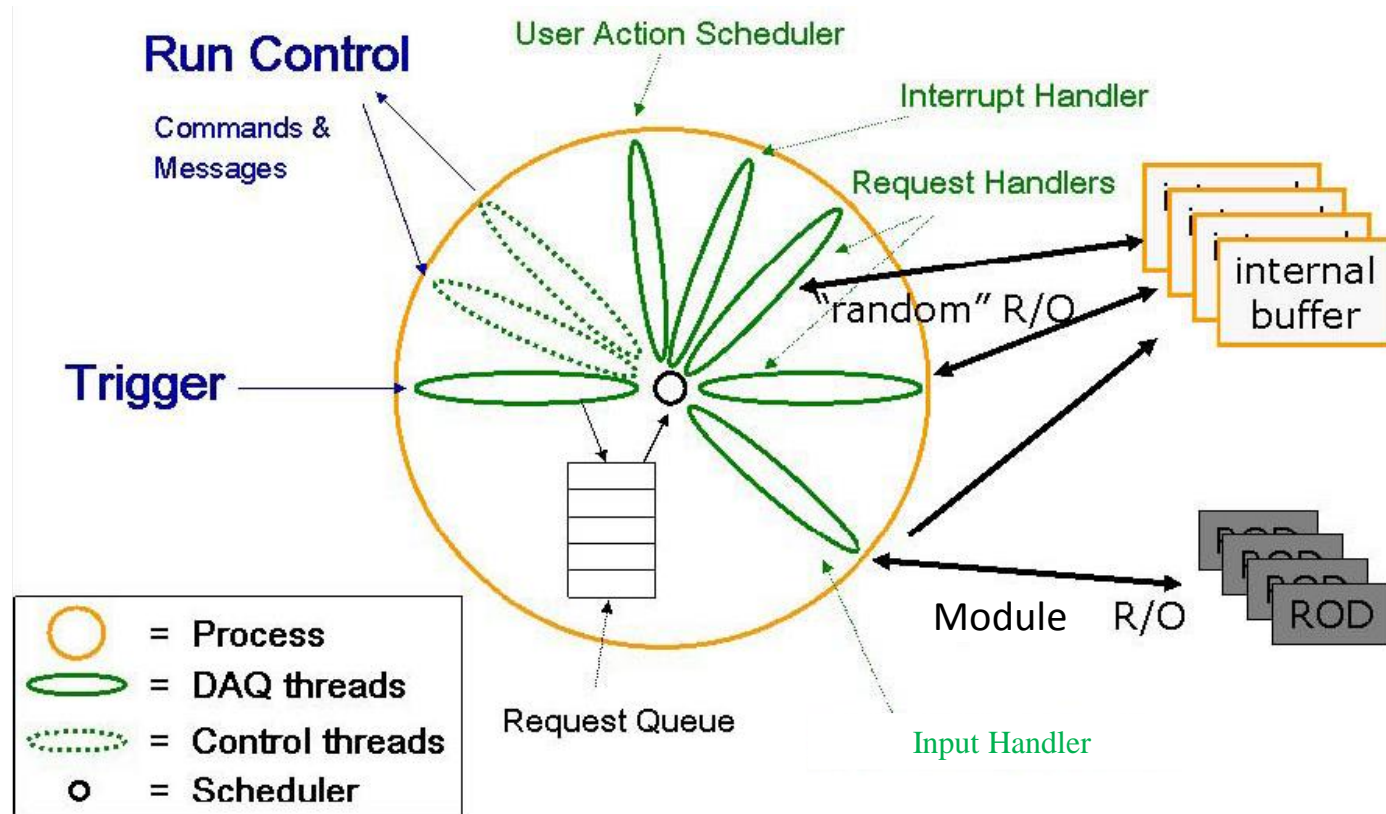
- Same basic idea
 - Use a pre-allocated memory pool to pass “events”
- Paged memory
 - Can be used to minimize pointer arithmetic
 - Convenient if event sizes are comparable
 - At the price of some memory
- Buffer descriptors
 - Built in an on-purpose pre-allocate memory
 - Pointers to descriptors are queued
- Allows any number of input and output threads



A paged memory pool



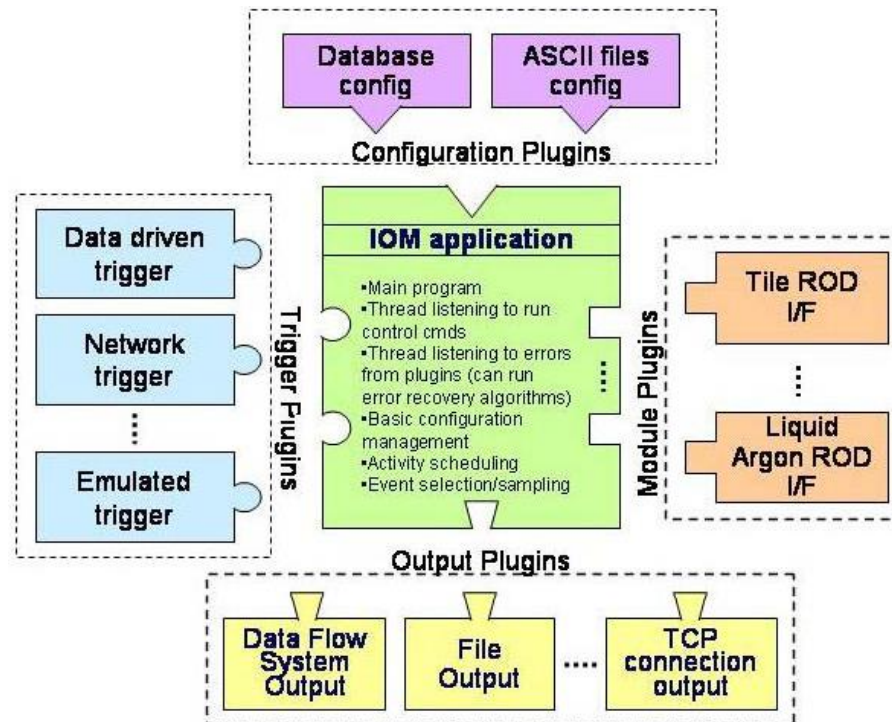
Generic readout application



Configurable applications



- Ambitious idea
 - Support all the systems with a single application
 - Through plug-in mechanism
 - Requires a configuration mechanism
 - You will (not) see an example in exercise 4



Some basic components

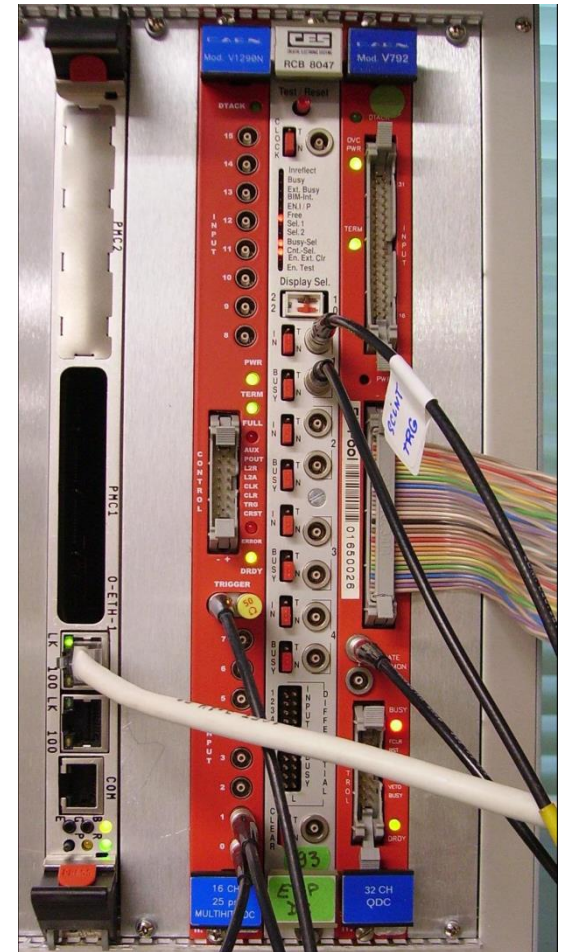


- We introduced basic elements of IPC...
 - Signals and signal catching
 - Shared memories
 - Semaphores (or mutexes)
 - Message queues
- ...and some standard DAQ concepts
 - Trigger management, busy, back-pressure
 - Synchronous vs asynchronous systems
 - Polling vs interrupts
 - Real time programming
 - Event framing
 - Memory management

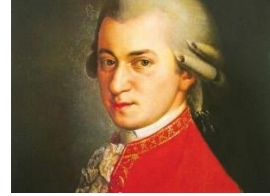


What will you find in the lab?

- Theory at work...
- Exercise 4
 - Simple DAQ with
 - VME crate controller
 - CORBO module
 - Upon trigger reception
 - » Sets busy
 - » Sends a VME interrupt
 - » Latch the trigger in a register
 - QDC
 - TDC

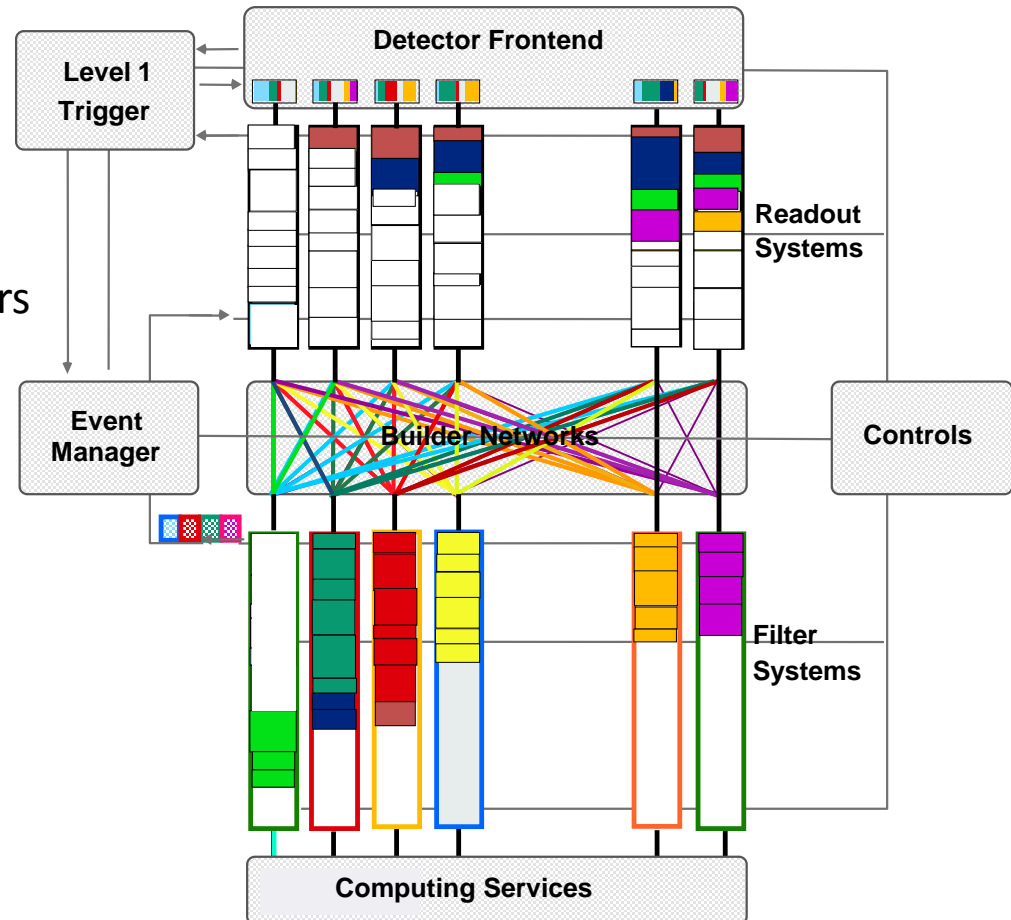




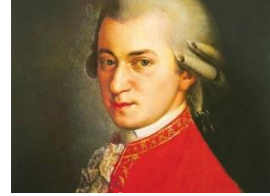


Event building

- Large detectors
 - Sub-detectors data are collected independently
 - Readout network
 - Fast data links
 - Events assembled by event builders
 - From corresponding fragments
 - Custom devices used
 - In FEE
 - In low-level triggers
 - COTS used
 - In high-level triggers
 - In event builder network
- DAQ system
 - data flow & control
 - distributed & asynchronous



Data networks and protocols



- Data transmission
 - Fragments need to be sent to the event builders
 - One or more...
 - Usually done via switched networks
- User-level protocols
 - Provide an abstract layer for data transmission
 - ... so you can ignore the hardware you are using ...
 - ... and the optimizations made in the OS (well, that's not always true) ...
 - See the lecture and exercise on networking
- Most commonly used
 - TCP/IP suite
 - UDP (User Datagram Protocol)
 - Connection-less
 - TCP (Transmission Control Protocol)
 - Connection-based protocol
 - Implements acknowledgment and re-transmission

TCP client/server example



```
struct sockaddr_in sinhim;
sinhim.sin_family      = AF_INET;
sinhim.sin_addr.s_addr = inet_addr (this_host);
sinhim.sin_port        = htons (port);
```

```
if (fd = socket (AF_INET, SOCK_STREAM, 0) < 0)
{ ; // Error ! }
if (connect (fd, (struct sockaddr *)&sinhim,
             sizeof (sinhim)) < 0)
{ ; // Error ! }
```

```
while (running) {
    memcpy ((char *) &wait, (char *) &timeout,
            sizeof (struct timeval));
    if ((nselect = select (nfd, 0, &wfd,
                          0, &wait)) < 0)
    { ; // Error ! }
    else if (nselect) {
        if ((BIT_ISSET (destination, wfd))) {
            count = write (destination, buf, buflen);
            // test count...
            // > 0 (has everything been sent ?)
            // == 0 (error)
            // < 0 we had an interrupt or
            // peer closed connection
        }
    }
}
```

```
close (fd);
```

```
struct sockaddr_in sinme;
sinme.sin_family      = AF_INET;
sinme.sin_addr.s_addr = INADDR_ANY;
sinme.sin_port        = htons (ask_var->port);
```

```
fd = socket (AF_INET, SOCK_STREAM, 0);
bind (fd0, (struct sockaddr *) &sinme,
       sizeof(sinme));
listen (fd0, 5);
```

```
while (n < ns) { // we expect ns connections
    int val = sizeof(this->sinhim);
    if ((fd = accept (fd0,
                      (struct sockaddr *) &sinhim, &val)) > 0) {
        FD_SET (fd, &fds);
        ++ns;
    }
}
```

```
while (running) {
    if ((nselect = select( nfd, (fd_set *) &fds,
                          0, 0, &wait)) [
        count = read (fd, buf_ptr, buflen);
        if (count == 0) {
            close (fd);
            // set FD bit to 0
        }
    }
}
```

```
close (fd0);
```



Data transmission optimization

- See F. Le Goff's lecture
- When you “send” data they are copied to a system buffer
 - Data are sent in fixed-size chunks
- At system level
 - Each endpoint has a buffer to store data that is transmitted over the network
 - TCP stops to send data when available buffer size is 0
 - Back-pressure
 - With UDP we get data loss
 - If buffer space is too small:
 - Increase system buffer (in general possible up to 8 MB)
 - Too large buffers can lead to performance problems
- You will play in lab. 9 with
 - Data transmission
 - Network control

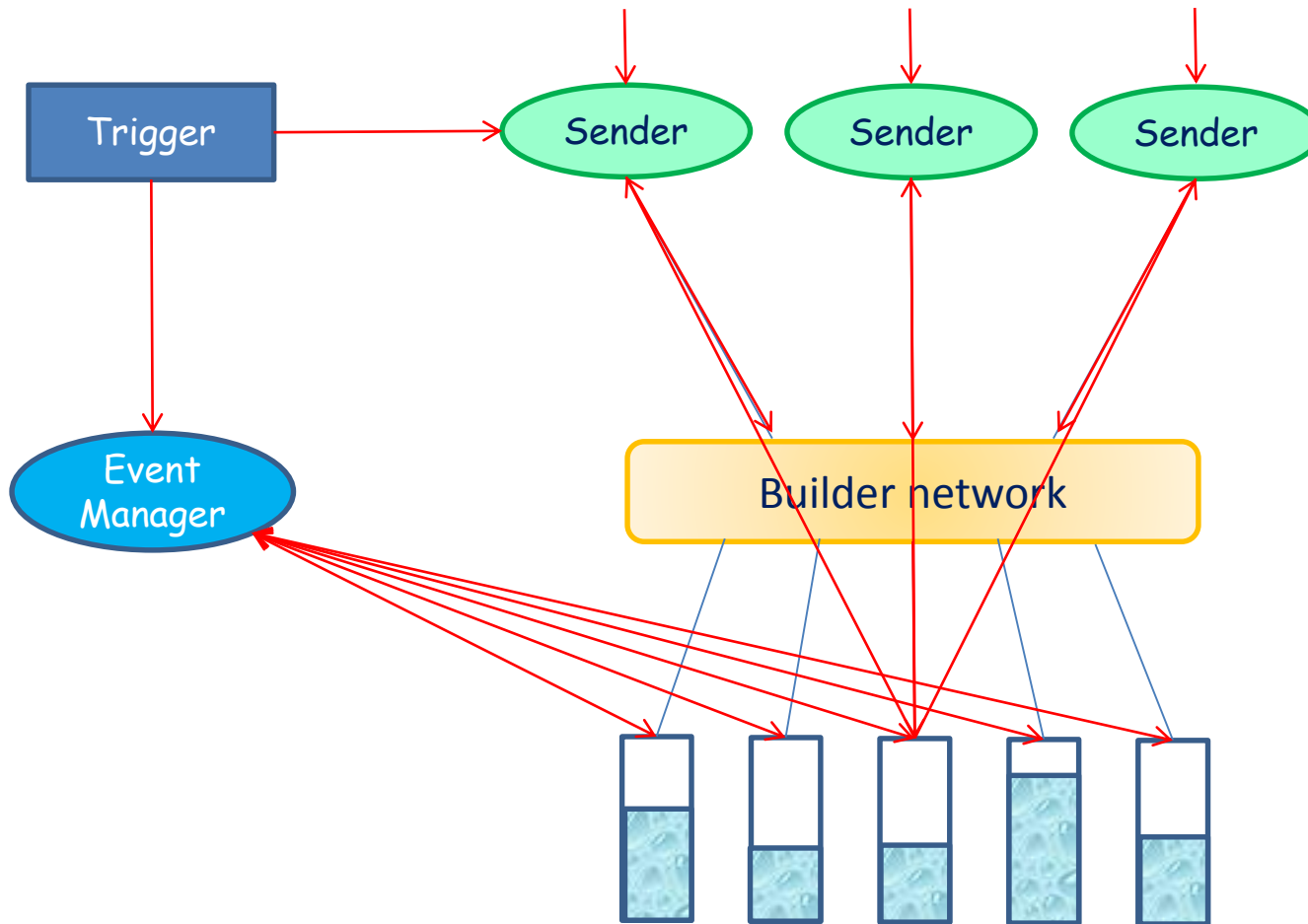
Controlling the data flow



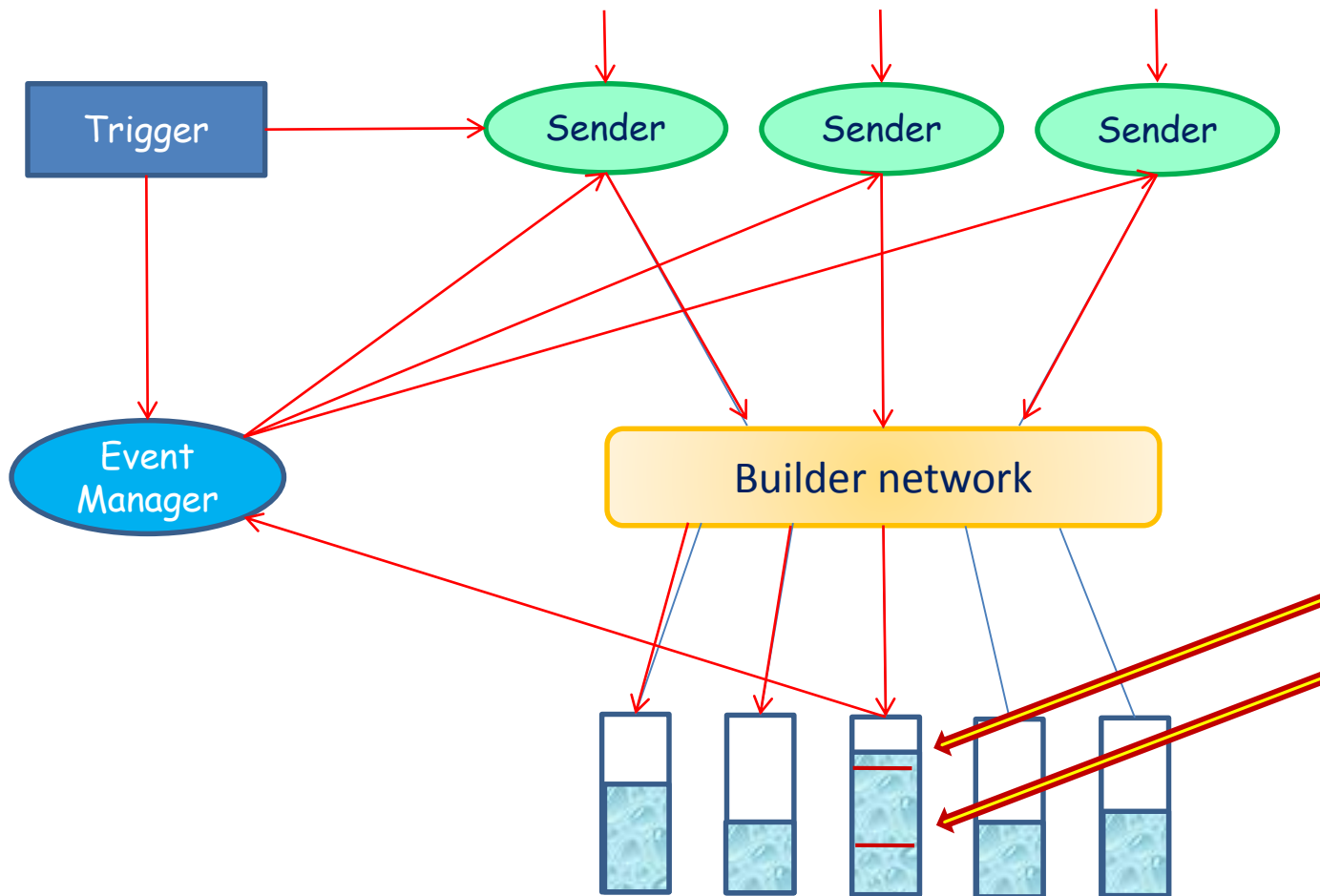
- Throughput optimization
- Avoid dead-time due to back-pressure
 - By avoiding fixed sequences of data destinations
 - Requires knowledge of the EB input buffer state
- EB architectures
 - Push
 - Events are sent as soon as data are available to the sender
 - The sender knows where to send data
 - The simplest algorithm for distribution is the *round-robin*
 - Pull
 - Events are required by a given destination processes
 - Needs an event manager
 - » Though in principle we could build a pull system without manager

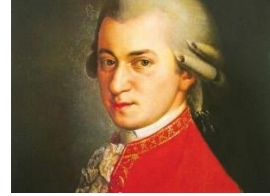


Pull example



Push example





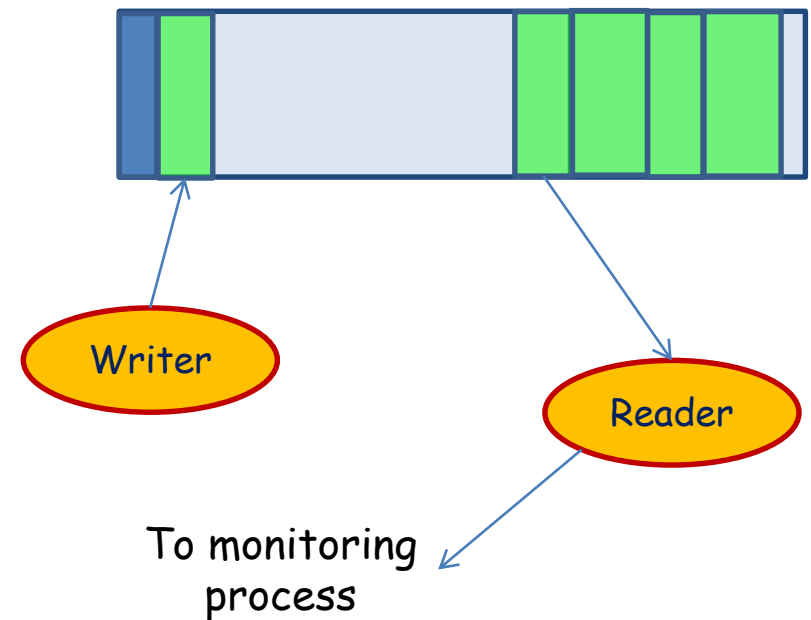
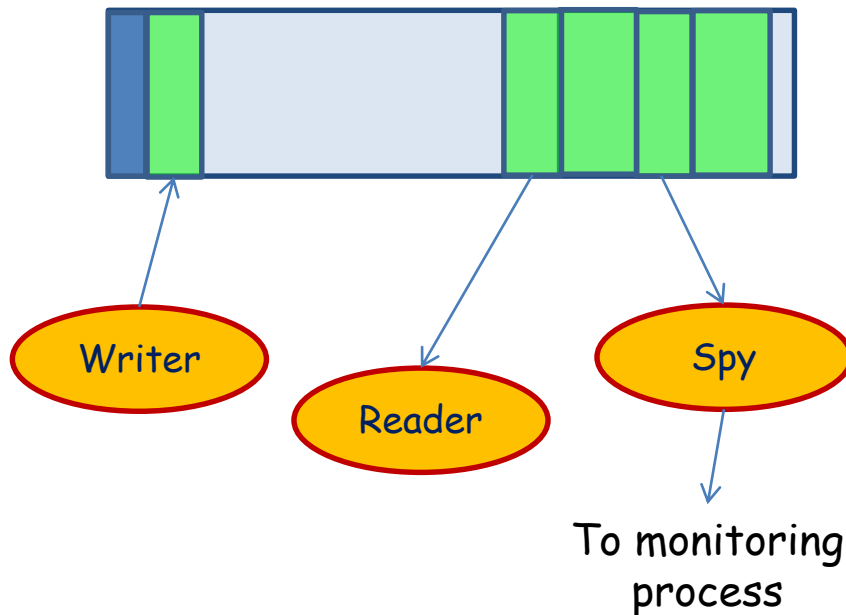
System monitoring

- Two main aspects
 - System operational monitoring
 - Sharing variables through the system
 - Data monitoring
 - Sampling data for monitoring processes
 - Sharing histogram through the system
 - Histogram browsing
 - See also S. Kolos' lecture



Event sampling examples

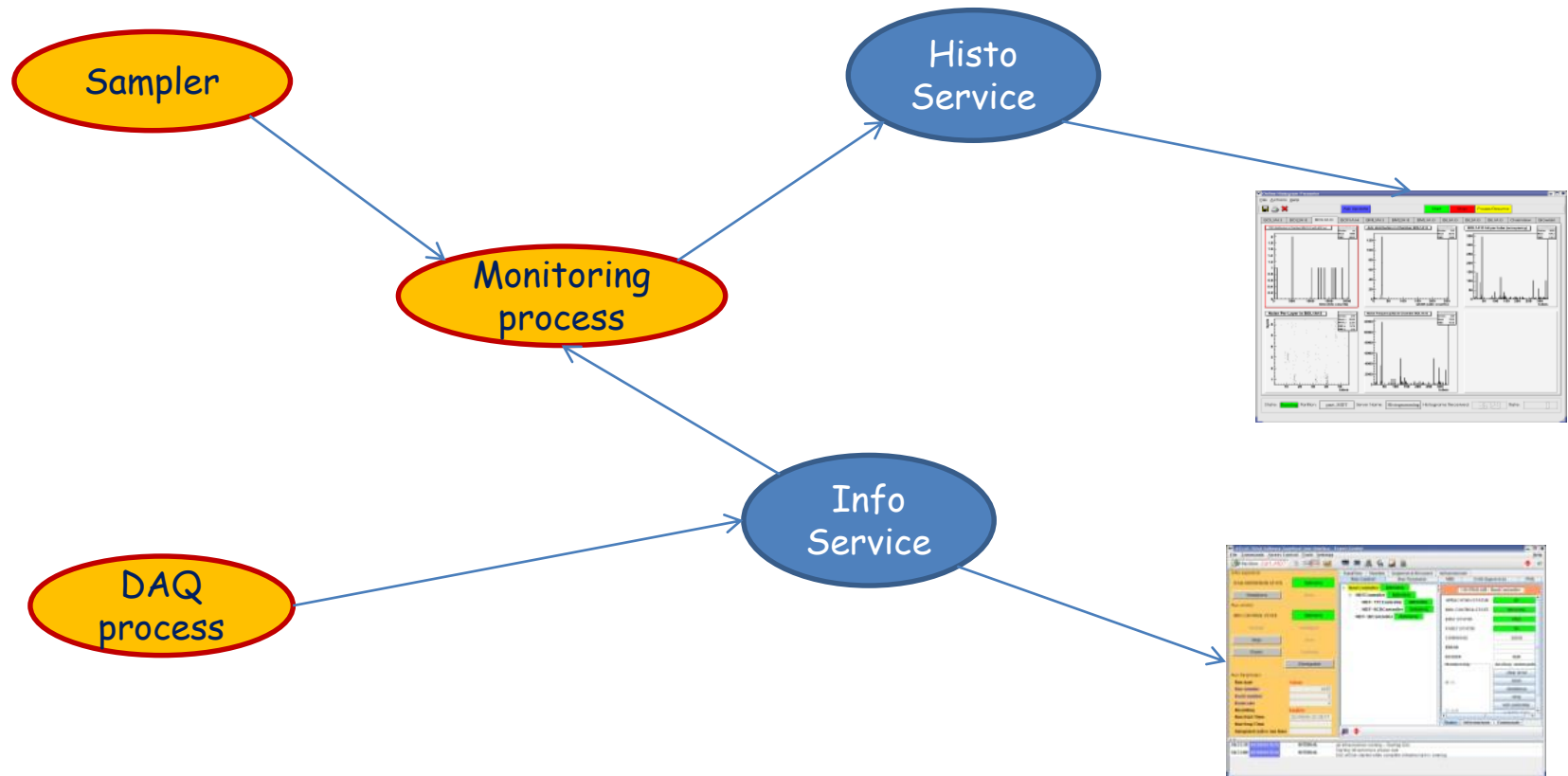
- Spying from buffers
- Sampling on input or output



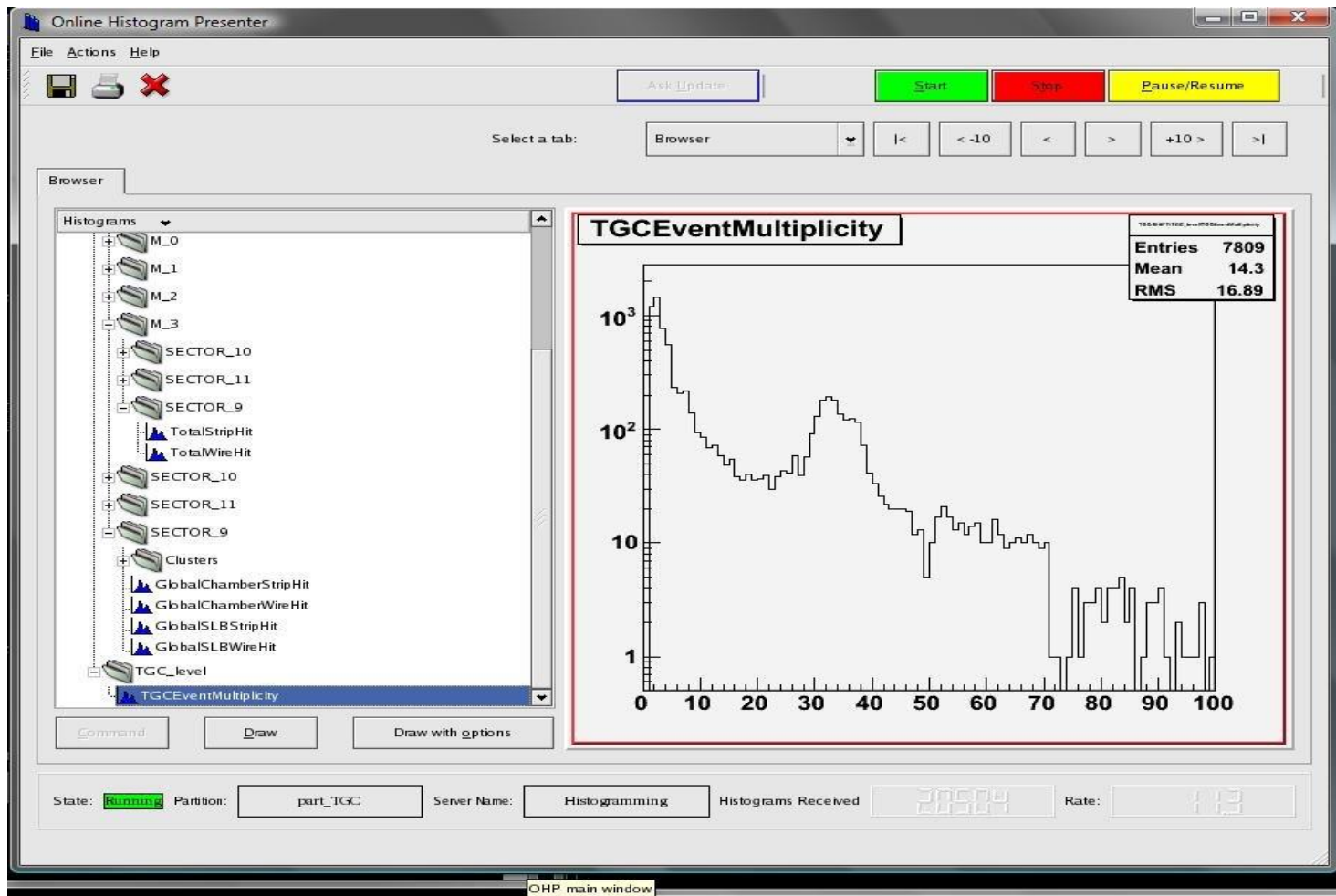
Sampling is always on the “best effort” basis and cannot affect data taking

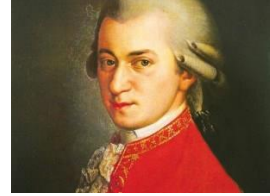


Histogram and variable distribution



Histogram browser



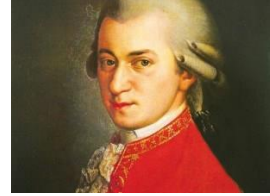


Controlling the system

- Each DAQ component must have
 - A set of well defined states
 - A set of rules to pass from one state to another

⇒ Finite State Machine
- A central process controls the system
 - Run control
 - Implements the state machine
 - Triggers state changes and takes track of components' states
 - Trees of controllers can be used to improve scalability
- A GUI interfaces the user to the Run control
 - ...and various system services...

GUI example



- From exercise 4...
 - ... and Atlas!

ATLAS TDAQ SOFTWARE - Partition part_Scintillator

File Commands Access Control Settings Logging Level Help

Commit & Reload Load Panels

MRS TS DVS ED OMS LM OH

RUN CONTROL STATE: NONE

Run Control Commands:

SHUTDOWN BOOT

TERMINATE INITIALIZE

UNCONFIG CONFIG

STOP START

HOLD TRG RESUME TRG

Beam Stable Warm Start Warm Stop

Run Information & Settings:

Run type: Physics

Run number: 1257947892

Super Master Key:

Detector Mask: 0

Recording: Disabled

Start time: 11-Nov-2009 14:58:12

Stop time: 11-Nov-2009 15:01:53

Total time: 0 h, 3 m, 41 s

Information Counters Settings

Run Control Segments & Resources Dataset Tags

NONE RootController

ABSENT ScintillatorSegment

RootController

HW

PMG

Infrastructure

Infrastructure Advanced

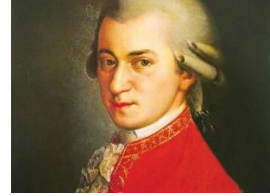
Show Online Segment Find: Match Case Repeats

Subscription criteria: ☒ WARNING ☒ ERROR ☒ FATAL ☐ INFORMATION ☐ Expression

Subscribe

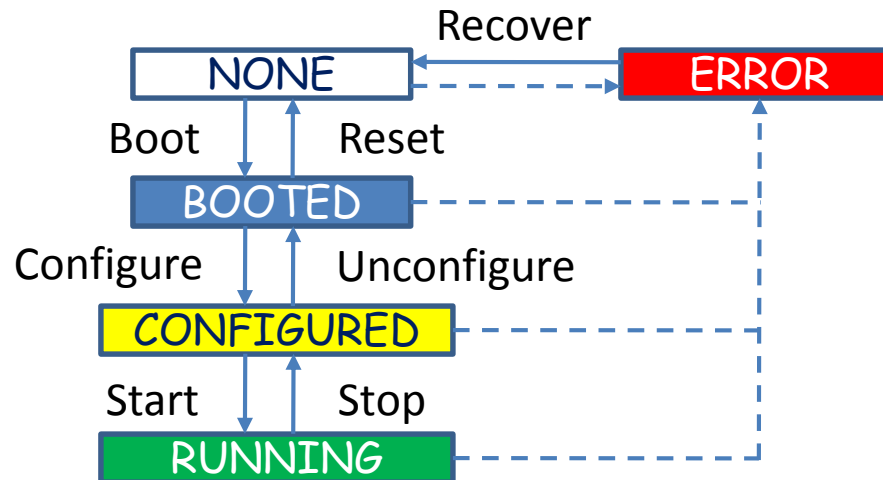
TIME	SEVERITY	APPLICATION	NAME	MESSAGE
15:13:30	INFORMATION	IGUI	INTERNAL	All done! IGUI is going to appear...
15:13:29	INFORMATION	IGUI	INTERNAL	Waiting for the "Dataset Tags" panel to initialize...
15:13:29	INFORMATION	IGUI	INTERNAL	Waiting for the "Segments & Resources" panel to initialize...

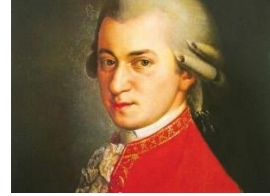
Clear Message format Number of visible rows: 100 Current MRS subscription: WARNING|ERROR|FATAL



Finite State Machines

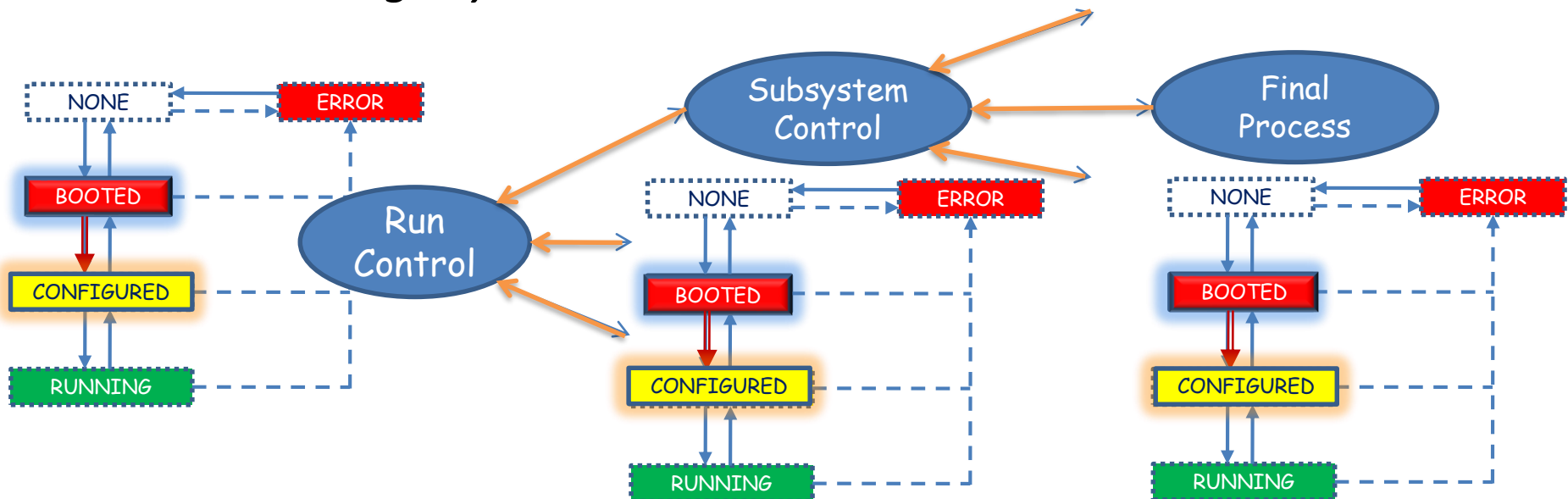
- Models of the behaviors of a system or a complex object, with a limited number of defined conditions or modes
- Finite state machines consist of 4 main elements:
 - States which define behavior and may produce actions
 - State transitions which are movements from one state to another
 - Rules or conditions which must be met to allow a state transition
 - Input events which are either externally or internally generated, which may possibly trigger rules and lead to state transitions



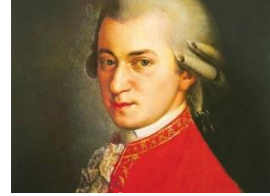


Propagating transitions

- Each component or sub-system is modeled as a FSM
 - The state transition of a component is completed only if all its sub-components completed their own transition
 - State transitions are triggered by commands sent through a *message system*



FSM implementation



- State concept maps on object state concept
 - OO programming is convenient to implement SM
- State transition
 - Usually implemented as callbacks
 - In response to messages
- Remember:
 - Each state **MUST** be well-defined
 - Variables defining the state must have the same values
 - Independently of the state transition
- You will work with a state machine
 - In exercise 12



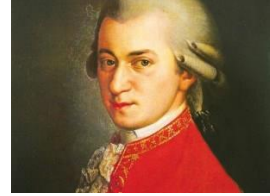
Message system

- Networked IPC
- I will not describe it
 - You see a message system at work in exercise 12
- Many possible implementations
 - From simple TCP packets...
 - ... through (rather exotic) SNMP ...
 - (that's the way many printers are configured...)
 - Very convenient for “economic” implementation
 - Used in the KLOE experiment
 - ... to Object Request Browsers (ORB)
 - Used f.i. by ATLAS



A final remark

- There is no absolute truth
 - Different systems require different optimizations
 - Different requirements imply different design
- System parameters must drive the SW design
 - As for DAQ HW design (see K. Kordas' talk)
 - Examples:
 - An EB may use dynamic buffering
 - Though it is expensive
 - If bandwidth is limited by network throughput
 - React to signals or poll
 - Depends on expected event rate
 - Event framing is important
 - But must not be exaggerated



Thanks for your attention!