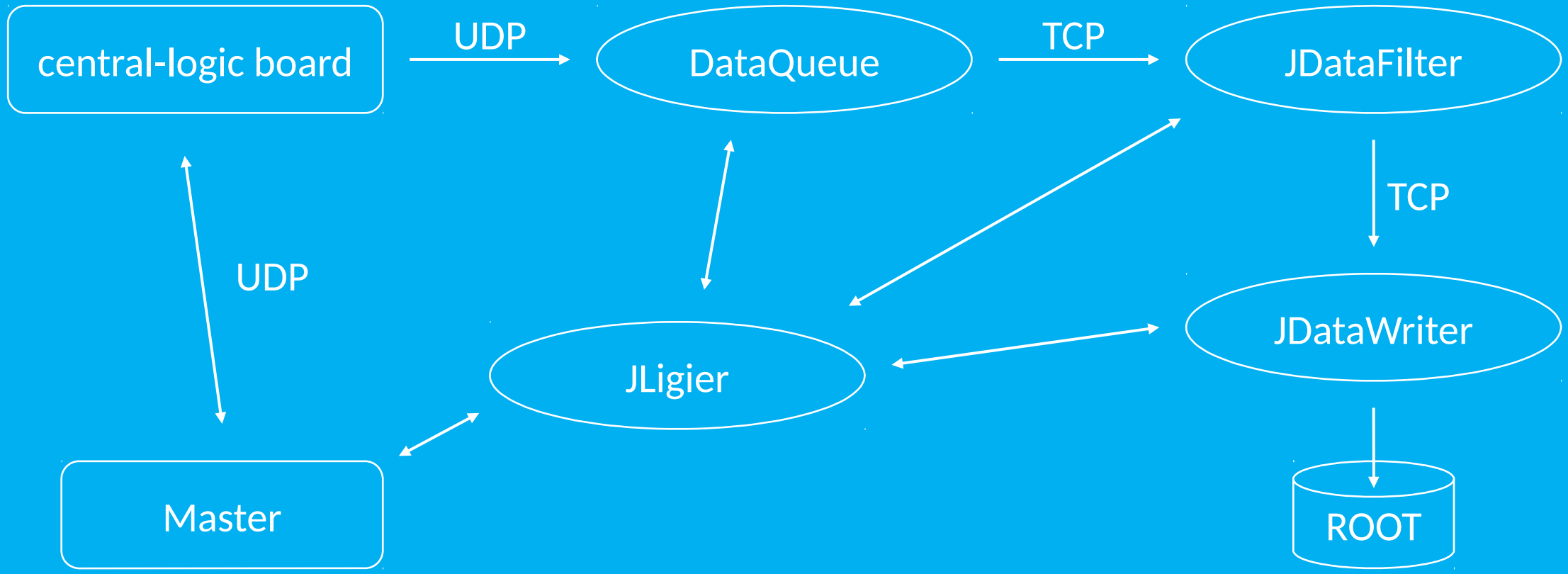
DAQ state machine

M. de Jong

DAQ system in a nutshell



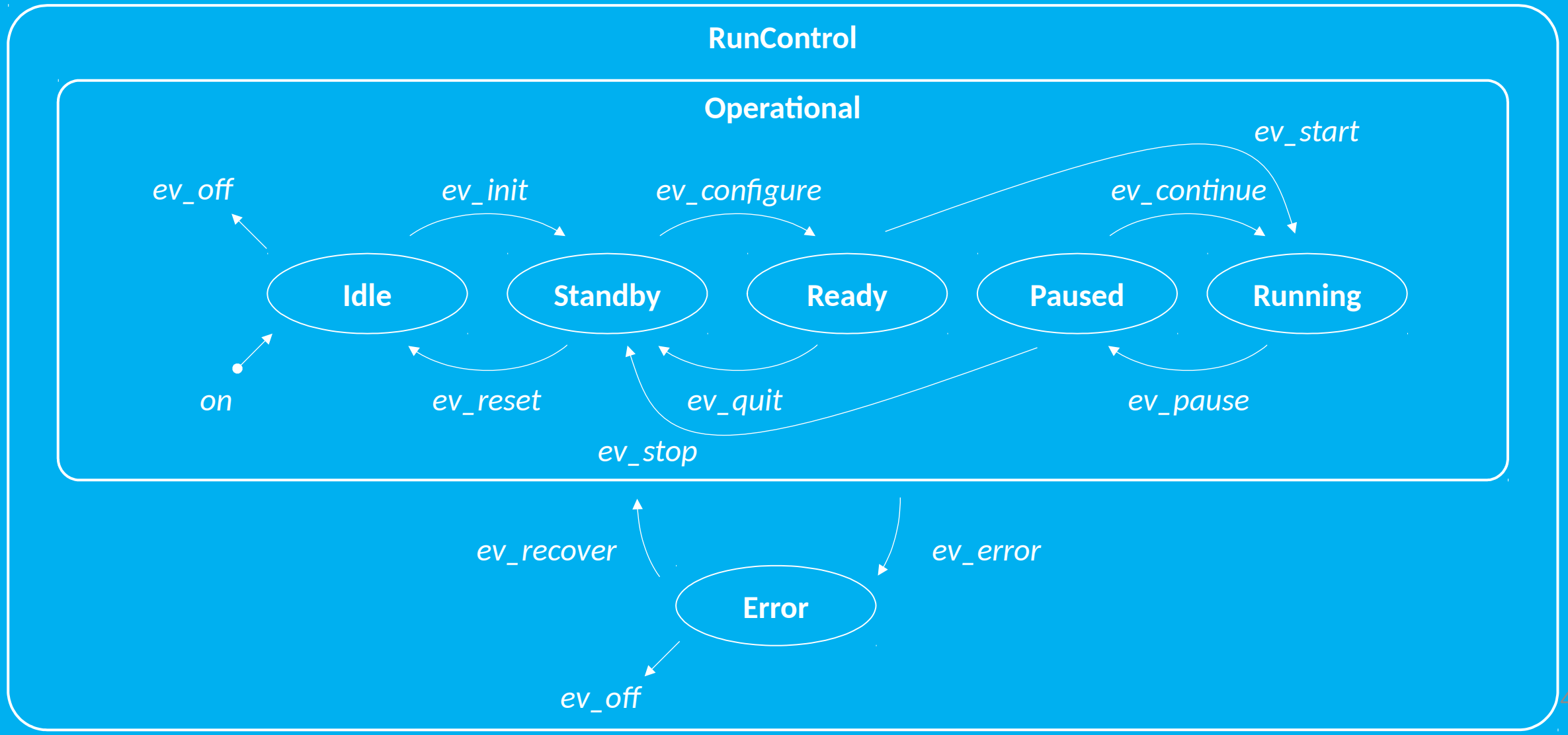
Introduction (1/3)

- There is one unique master
  - e.g. central GUI

- There is a variety of clients¶
  - each client implements same state machine (next slide)
  - each client communicates with master via (central) server
  - each client reports messages to (central) logger

¶ This refers to shore station part of DAQ system.

Introduction (2/3)



Introduction (3/3)

- Protocol for handshaking of state transitions is based on combination of 1) state machine logic, 2) tagged messages and 3) process name
  1) event names are unique by construction
  2) tagged messages are unambiguous by implementation
  3) process names are unique by implementation

JLigier (1/2)

- JLigier is server for inter-process communications
  - protocol based on “tagged” messages
    - messages can contain any data
  - subscription mechanisms
    - “any” receive as much data as possible with given tag
    - “all”receive all data with given tag
  - registration of nick name of process
    - JControlHost::MyId(<nick name>);

    → JLigier broadcasts message “<nick name>” with tag “Born”

    → JLigier broadcasts message “<nick name>” with tag “Died” when process disconnects from JLigier (e.g. terminates)

```
struct JPrefix
{
    char            tag[TAGSIZE ¶];
    long long int   size;
}
```

¶ TAGSIZE = 8;

JLigier (2/2)

- point-to-point connections based on TCP/IP
  - no message will be lost
    - but not specified when message will arrive
- order of messages maintained
  - internal buffers work as FIFOs
- any error is printed to terminal

Implementation (1/5)

- Every event has a corresponding action method
  - ev_XXX → actionXXX(int length, const char* buffer)
    - length = number of bytes in buffer
    - buffer = input data to action
  - enter → actionEnter()
  - ev_off → actionExit()

```
void run()
{
    while (active()) {
        update();
    }
}
```

Implementation (2/5)

```
void update()
{
    JPrefix prefix;

    server->WaitHead(prefix);

    const int length  =  prefix.getSize();
    char* buffer      =  new char[length];

    server->GetFullData(buffer, length);

    update(prefix.getTag(), length, buffer);

    delete [] buffer;
}
```

Implementation (3/5)

- message content
  - <event name>[:<event number>]#[data]
    - optional data are treated as array of bytes
    - data are transferred as-is to corresponding action method
- event table
  - map pair of (<tag>, <event name>) to CHSM event
  - list of accepted tags (included by default)
    - general tag “RC_CMD”
    - unique tag “<IP sub-address¶>/<client name§>”

¶ IP sub-address written in hexadecimal code.

§ client name is specified on command line of application option –u <client name>

Implementation (4/5)

- JDAQClient::update(tag, length, buffer)
  1. parse event name and optional number
  2. look up CHSM event from event table
  3. trigger event

     → call corresponding action method
  4. enter state

     → send reply

Implementation (5/5)

- upon entering state <u>after</u> successful completion of state transition
  - reply message is sent to JLigier
    - tag “RC_REPLY”
    - data “<full name>#<event name>:<event number¶>#<state name§>”
      where <full name> = “DAQ#<IP address†>#<client name>”

¶ From previous request for state transition.

§ complete state name is (for backward compatibility) “Main.RunControl.<state name>.”

† IP address written in hexadecimal code.

Special actions (1/1)

- virtual void JDAQClient::setSelect(JFileDescriptorMask& mask) const;
  - can be used to listen to other file descriptors (e.g. sockets)
  - called before method update()

- virtual void JDAQClient:: actionSelect(const JFileDescriptorMask& mask);
  - can be used to implement actions for other file descriptors
  - called after method update()

Error handling (1/4)

- State machine is either in cluster **Operational** or state **Error**
  - event *ev_error* can be triggered any time
    - exit cluster **Operational**
    - enter state **Error**
  - state **Error** can only be exited by following events
    - *ev_recover* → enter state **Operational.Idle** (i.e. no history¶)
    - *ev_off* → terminates application

¶ It is possible to specify history in CHSM.

Error handling (2/4)

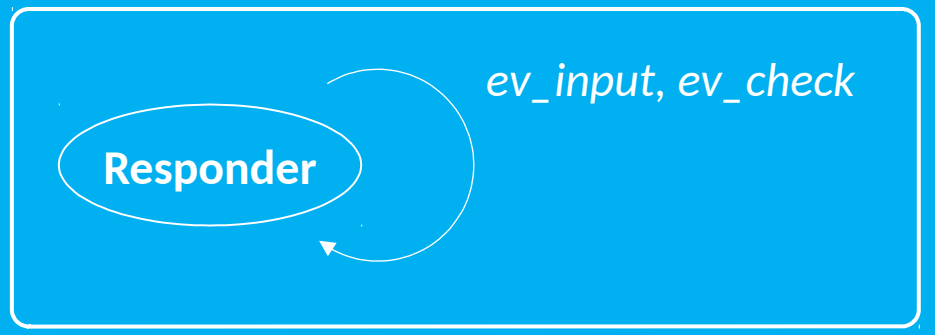
- Exceptions in any actionXXX() method will be caught and trigger event *ev_error*
- Upon entering state **Error**, standard reply message is sent, i.e:
  - tag “RC_REPLY”
  - data “<full name>#ev_error#Main.RunControl.Error”
- Tags to trigger events *ev_recover* or *ev_off*
  - general tag “RC_CMD”
  - unique tag “<IP sub-address>/<client name>”
- Default implementation of corresponding action methods are empty
  - virtual void actionError() {}
  - virtual void actionRecover(int, const char*) {}

Error handling (3/4)

- Following a request for a state transition, either of the following cases will happen:
  a) success
     - tag “RC_REPLY”
     - data“<full name>#<event name>:<event number>#<state name>”
  b) invalid
     - tag “RC_FAIL”
     - data“<full name>#<event name>:<event number>#<state name>”
  c) termination
     - tag “Died”
     - data“<nick name>”
  d) timeout

Error handling (4/4)

- each state machine is also in state **Responder**
- ev_input
  - set debug level, etc.
- ev_check
  - sends message
    - tag “RC_REPLY”
    - data “<full name>#ev_check:<event number>#<state name>”



Specifications (1/2)

- client
  - should receive all requests for state transition from master
    - ✓ subscription to tags “RC_CMD” and <unique tag> is “all”
  - should have unique nick name
    - ✓ nick name equals full name
  - should reply after request for state transition from master within timeout
    - *“AcousticDataFilter may take two minutes to complete actionConfigure()”*

Specifications (2/2)

- master
  - should receive all replies from clients
    - subscription to tags “RC_REPLY”, “RC_FAIL”, “Born” and “Died” should be “all”
  - should maintain state of complete system
    - state transitions of clients should be synchronised
      i.e. all clients are in targeted state before new state transition is triggered

Notifications (1/2)

- JLigier subscription with “any” may result in loss of messages
  - client too slow to process all messages
  - JLigier will not report this as error

- JLigier subscription with “all” may result in congestion
  - client too slow to process all messages
  - JLigier will report this as error

Notifications (2/2)

- client
  - should not need to know state of other clients
    - nonetheless JDataWriter needs handling of cases in which JDataFilter is in different run

- master
  - polling of state is unreliable
    - state of client is given in reply message following request for a state transition
  - use of *ev_check* should be limited to exceptional cases
    - e.g. restart of master