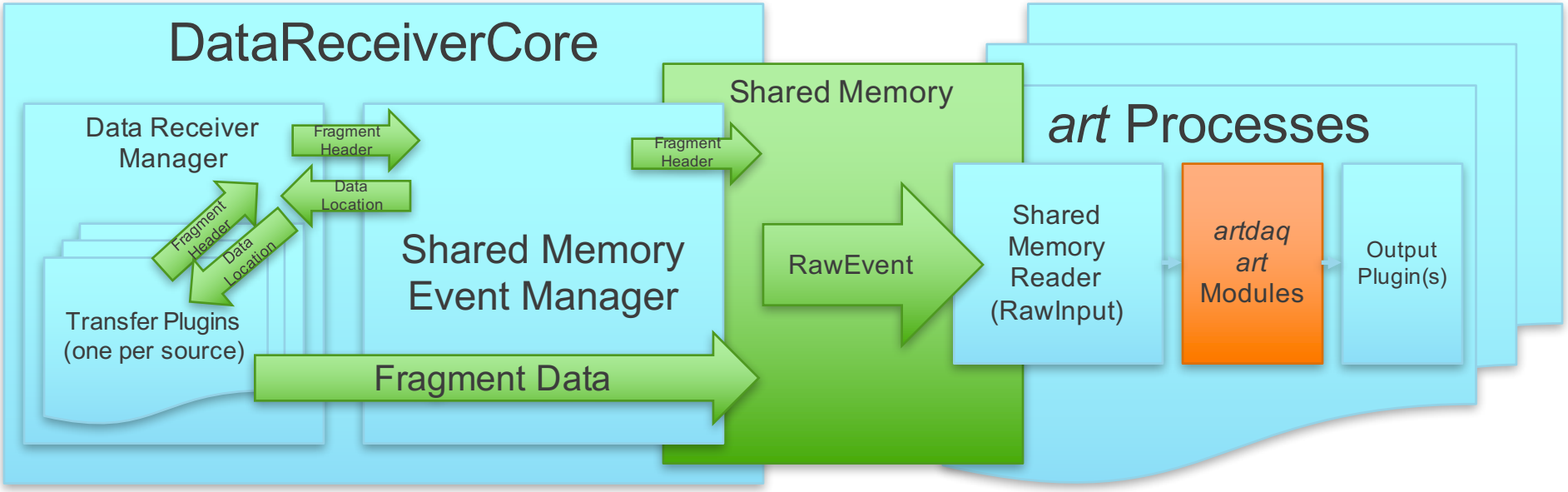
DATAFLOW II: EVENTBUILDER TO DATALOGGER

Wesley Ketchum
(and the *artdaq* team)

EVENTBUILDER APPLICATION

- Application responding to state machine that converts collections of artdaq::Fragments to art::Events
  - *Input: artdaq fragment(s) from BoardReaders*
  - *Output: art::Events to DataLoggers*
- There are no directly user/experiment-written components
  - *Basically all artdaq internal → any changes imply new version of artdaq*
- There are "plugins" for what to do with data
  - *TransferPlugins for data transport*
  - *art modules for data processing and filtering*
  - *Message/metric plugins for what to do with monitoring/logging data*

Diagram of the EventBuilder



EventBuilderCore

- Inherits from DataReceiverCore
  - *Same logic for EventBuilder as DataLogger and Dispatcher*
- DataReceiverCore holds a DataReceiverManager (DRM) and (shared ptr to) SharedMemoryEventManager (SMEM)
  - *SMEM inherits from SharedMemoryManager, and owns/manages the shared memory buffer*
- DRM runs threads for each TransferInput source
- SMEM launches and monitors *art* process, which in turn decide their output module location
  - *Multiple art threads allows distributed data processing and filtering without needing more EventBuilder applications*

DataReceiverManager threads (in brief)

1. Wait (with timeout) for fragment header
   - *Don't exit on timeout … just try again*
2. Check fragment type to handle data vs. control
3. If data, request buffer location from SMEM based on sequence ID, retrying for valid location except in non-reliable mode
4. Collect fragment data into provided location
5. Inform SMEM fragment is completed with writing
6. Increment fragment counting stats
7. Report receiving stats to metric manager

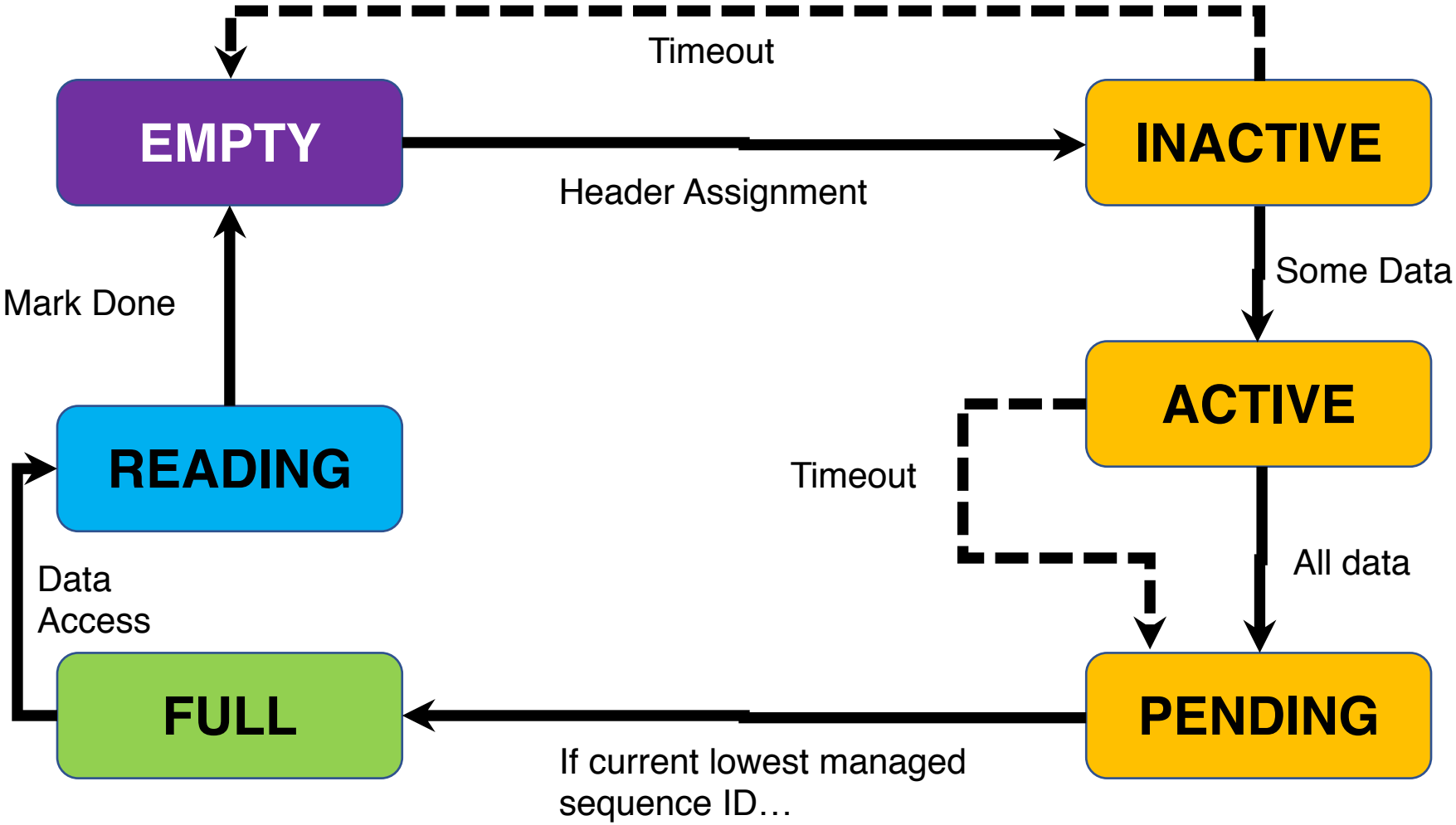
DRM EndOfData

- When DRM gets “EndOfData” fragment, it extracts “EndOfData” count to know sequence IDs it expects
- DRM receiving threads will wait for remaining data
  - *Subject to timeouts*
  - *One could force stop of all sources by sending EndOfData fragment with no more expected fragments*

SharedMemoryEventManager

- Total buffer size determined at initialization, based on max events in buffer, max event size or max fragment size * number of expected fragments per event
- “startRun” called just before start of DRM (at start transition of course) → start of art threads
- Each art thread must be configured in same way, with RawInput_source source module
  - *Avoid potential art processing history bug*
  - *More on RawInput_source later…*
- Each SM event buffer holds flag of state, reader/writer position, and ”last touched time”
  - *SMEM additionally labels buffers being written as inactive, active, or pending*

SharedMemory Buffer State Machine



Multithreaded Access of SharedMemory

- Writing and reading of shared memory is multi-threaded
  - *Multiple sources can write into memory simultaneously*
  - *Multiple art threads can read data from memory simultaneously*
- Mutexes managed per buffer

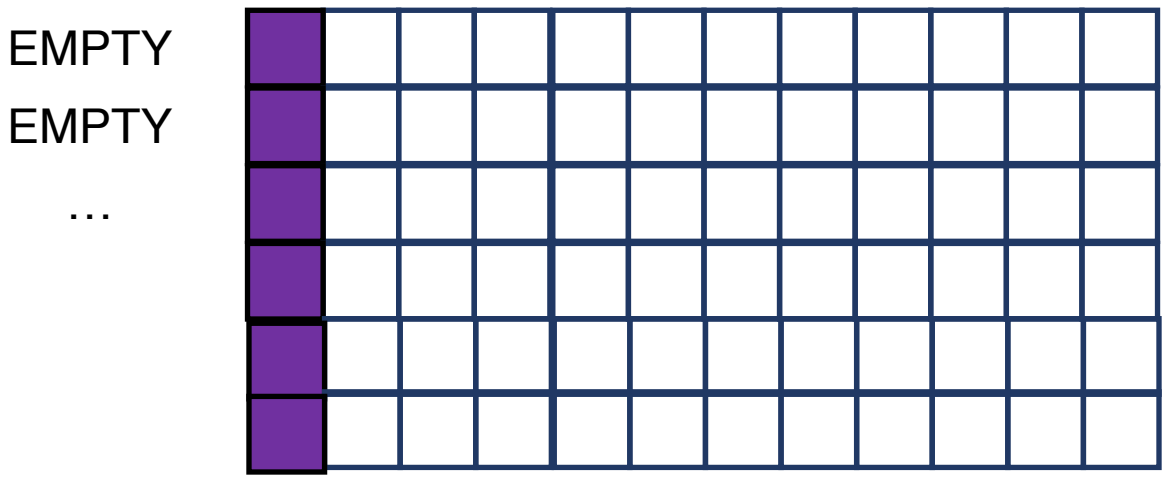
SHAREDMEMORY BUFFER FLOW

- Buffers setup on initialization

Buffer
Buffer
Buffer
Buffer
Buffer
Buffer

SHAREDMEMORY BUFFER FLOW

- All start as empty

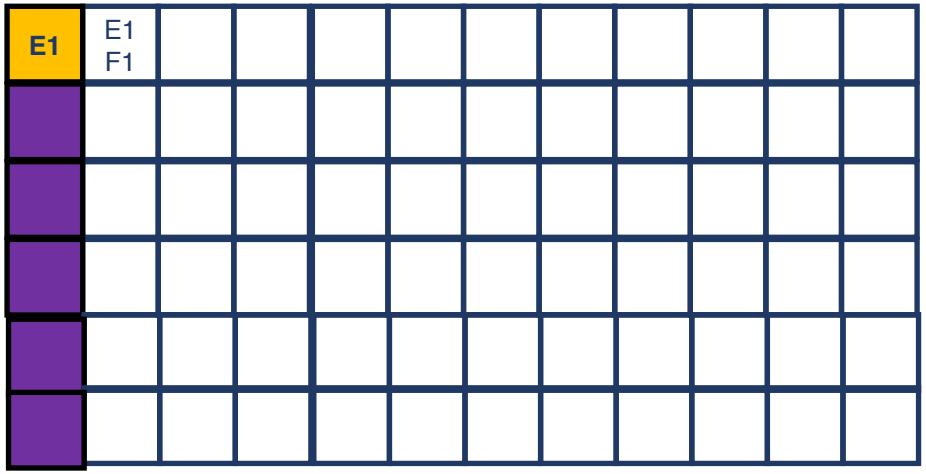


SHAREDMEMORY BUFFER FLOW

- First fragment arrives
  - *Buffer space allocated for that event*
  - *State is “active” since data is there*

WRITING (Active)

EMPTY



SHAREDMEMORY BUFFER FLOW

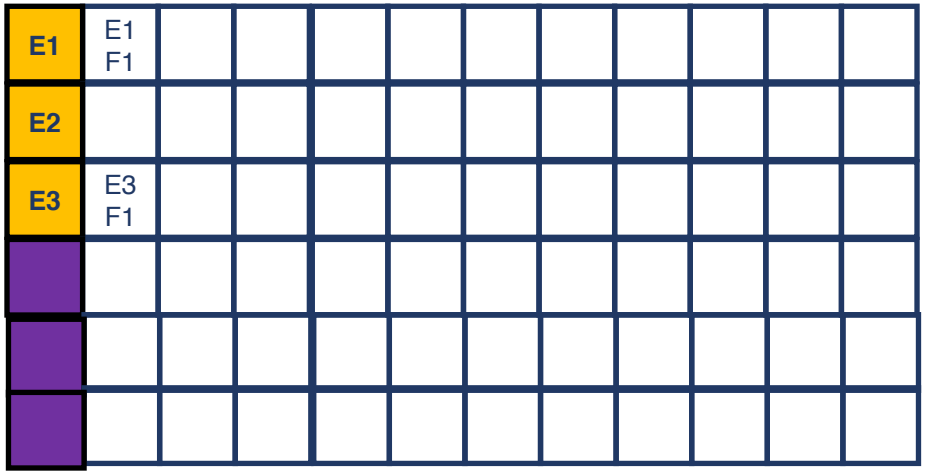
- Fragment from next event arrives
  - *All sequence IDs not previously allocated are assigned*
  - *Buffer without data is “Inactive”*

WRITING (Active)

WRITING (Inactive)

WRITING (Active)

EMPTY



SHAREDMEMORY BUFFER FLOW

- Remaining fragments from first event arrive
  - *Once expected number of fragments detected, buffer marked "Pending"*

WRITING (Pending)

WRITING (Inactive)

WRITING (Active)

EMPTY

E1	E1 F1	E1 F2	E1 F3	E1 F4	E1 F5	E1 F6	E1 F7	E1 F8	E1 F9	E1 FA	E1 FB
E2											
E3	E3 F1										

SHAREDMEMORY BUFFER FLOW

- Fragment state shifts to FULL, as it's the lowest sequence ID in buffer

FULL

WRITING (Inactive)

WRITING (Active)

EMPTY

E1	E1 F1	E1 F2	E1 F3	E1 F4	E1 F5	E1 F6	E1 F7	E1 F8	E1 F9	E1 FA	E1 FB
E2											
E3	E3 F1										

SHAREDMEMORY BUFFER FLOW

- While fragments arrive, FULL buffer can be read by an *art* thread

READING

WRITING (Inactive)

WRITING (Active)

EMPTY

E1	E1 F1	E1 F2	E1 F3	E1 F4	E1 F5	E1 F6	E1 F7	E1 F8	E1 F9	E1 FA	E1 FB
E2											
E3	E3 F1	E3 F2	E3 F3	E3 F4	E3 F5	E3 F6	E3 F7	E3 F8	E3 F9		

SHAREDMEMORY BUFFER FLOW

- Fragments can continue to arrive while we are reading/processing first event

READING	E1	E1 F1	E1 F2	E1 F3	E1 F4	E1 F5	E1 F6	E1 F7	E1 F8	E1 F9	E1 FA	E1 FB
WRITING (Inactive)	E2											
WRITING (Active)	E3	E3 F1	E3 F2	E3 F3	E3 F4	E3 F5	E3 F6	E3 F7	E3 F8	E3 F9		
WRITING (Inactive)	E4											
WRITING (Active)	E5	E5 F1										

SHAREDMEMORY BUFFER FLOW

- Remaining fragments for the “next” event, pushing its state to Pending
  - *Does not immediately move to FULL, as Event 2 is still in buffer*

READING	**E1**	E1 F1	E1 F2	E1 F3	E1 F4	E1 F5	E1 F6	E1 F7	E1 F8	E1 F9	E1 FA	E1 FB
WRITING (Inactive)	**E2**											
WRITING (Pending)	**E3**	E3 F1	E3 F2	E3 F3	E3 F4	E3 F5	E3 F6	E3 F7	E3 F8	E3 F9	E3 FA	E3 FB
WRITING (Inactive)	**E4**											
WRITING (Active)	**E5**	E5 F1										

SHAREDMEMORY BUFFER FLOW

- *art* thread can finish with event and buffer gets marked done …

Mark Done …

WRITING (Inactive)

WRITING (Pending)

WRITING (Inactive)

WRITING (Active)

E1	E1 F1	E1 F2	E1 F3	E1 F4	E1 F5	E1 F6	E1 F7	E1 F8	E1 F9	E1 FA	E1 FB
E2											
E3	E3 F1	E3 F2	E3 F3	E3 F4	E3 F5	E3 F6	E3 F7	E3 F8	E3 F9	E3 FA	E3 FB
E4											
E5	E5 F1										

SHAREDMEMORY BUFFER FLOW

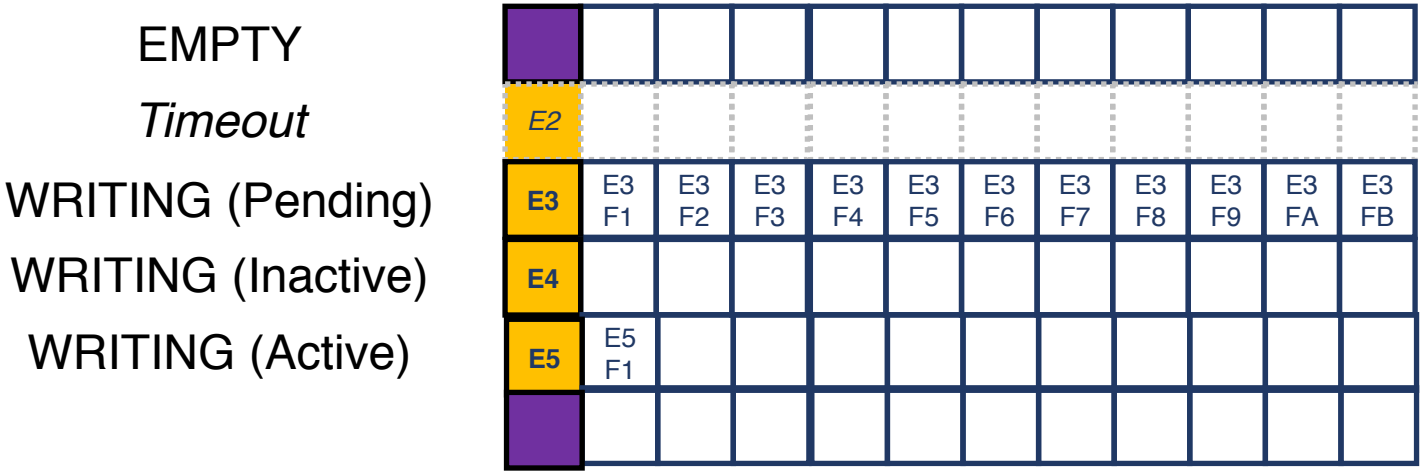
- … and is then marked as EMPTY

EMPTY
WRITING (Inactive)
WRITING (Pending)
WRITING (Inactive)
WRITING (Active)

E2											
E3	E3 F1	E3 F2	E3 F3	E3 F4	E3 F5	E3 F6	E3 F7	E3 F8	E3 F9	E3 FA	E3 FB
E4											
E5	E5 F1										

SHAREDMEMORY BUFFER FLOW

- Buffers in WRITING state can eventually timeout...



SHAREDMEMORY BUFFER FLOW

- …and then revert back to EMPTY...

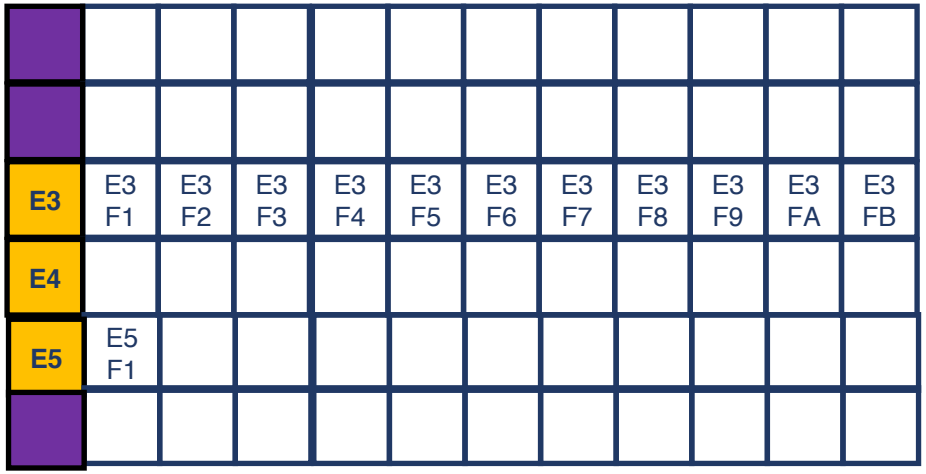
EMPTY

EMPTY

WRITING (Pending)

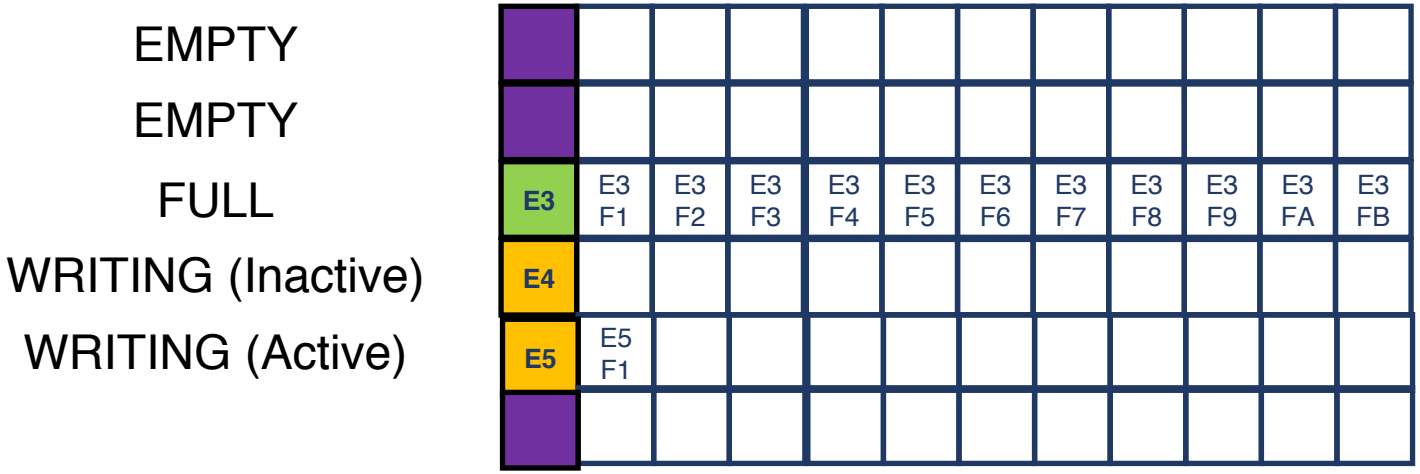
WRITING (Inactive)

WRITING (Active)



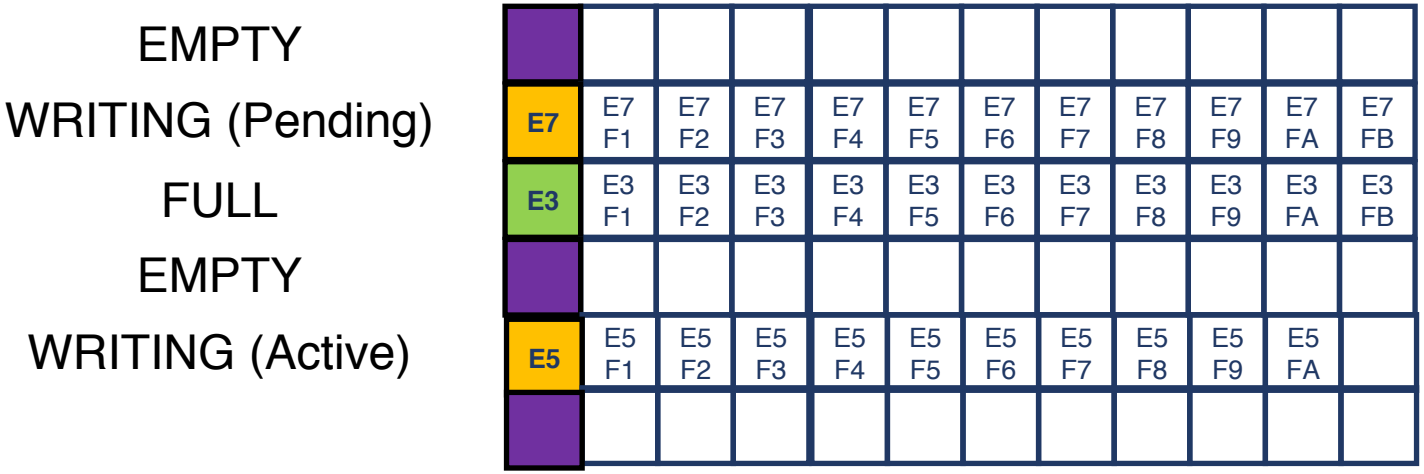
SharedMemory Buffer Flow

- ...which allows the pending buffer to be marked FULL
  - *And then is readable by an art thread*



SHAREDMEMORY BUFFER FLOW

- If a fragment doesn't arrive for a particular event, can also see backlog due to that missing fragment



SHAREDMEMORY BUFFER FLOW

- On timeout, active buffers are pushed forward to pending...

EMPTY											
WRITING (Pending)	E7	E7 F1	E7 F2	E7 F3	E7 F4	E7 F5	E7 F6	E7 F7	E7 F8	E7 F9	E7 FA	E7 FB
FULL	E3	E3 F1	E3 F2	E3 F3	E3 F4	E3 F5	E3 F6	E3 F7	E3 F8	E3 F9	E3 FA	E3 FB
EMPTY											
WRITING (Pending)	E5	E5 F1	E5 F2	E5 F3	E5 F4	E5 F5	E5 F6	E5 F7	E5 F8	E5 F9	E5 FA	

SHAREDMEMORY BUFFER FLOW

- ...which allows them to be marked as FULL ...
  - *Note: pending and FULL buffers will not accept new fragments, so incoming missing fragment would be ignored*

EMPTY											
WRITING (Pending)	E7	E7 F1	E7 F2	E7 F3	E7 F4	E7 F5	E7 F6	E7 F7	E7 F8	E7 F9	E7 FA	E7 FB
FULL	E3	E3 F1	E3 F2	E3 F3	E3 F4	E3 F5	E3 F6	E3 F7	E3 F8	E3 F9	E3 FA	E3 FB
EMPTY											
FULL	E5	E5 F1	E5 F2	E5 F3	E5 F4	E5 F5	E5 F6	E5 F7	E5 F8	E5 F9	E5 FA	

SHAREDMEMORY BUFFER FLOW

- ...which will then eventually clear the backlog

EMPTY											
FULL	E7	E7 F1	E7 F2	E7 F3	E7 F4	E7 F5	E7 F6	E7 F7	E7 F8	E7 F9	E7 FA	E7 FB
FULL	E3	E3 F1	E3 F2	E3 F3	E3 F4	E3 F5	E3 F6	E3 F7	E3 F8	E3 F9	E3 FA	E3 FB
EMPTY											
FULL	E5	E5 F1	E5 F2	E5 F3	E5 F4	E5 F5	E5 F6	E5 F7	E5 F8	E5 F9	E5 FA	

SHAREDMEMORY BUFFER FLOW

- When enough events come in (and we haven't read events fast enough) the buffer can start to fill...

WRITIING (Inactive)	E8										
FULL	E7	E7 F1	E7 F2	E7 F3	E7 F4	E7 F5	E7 F6	E7 F7	E7 F8	E7 F9	E7 FA	E7 FB
FULL	E3	E3 F1	E3 F2	E3 F3	E3 F4	E3 F5	E3 F6	E3 F7	E3 F8	E3 F9	E3 FA	E3 FB
WRITING (Active)	E9	E9 F1									
FULL	E5	E5 F1	E5 F2	E5 F3	E5 F4	E5 F5	E5 F6	E5 F7	E5 F8	E5 F9	E5 FA	

SHAREDMEMORY BUFFER FLOW

- ...and so if the next event (here event 11) comes in, we overwrite the lowest inactive buffer

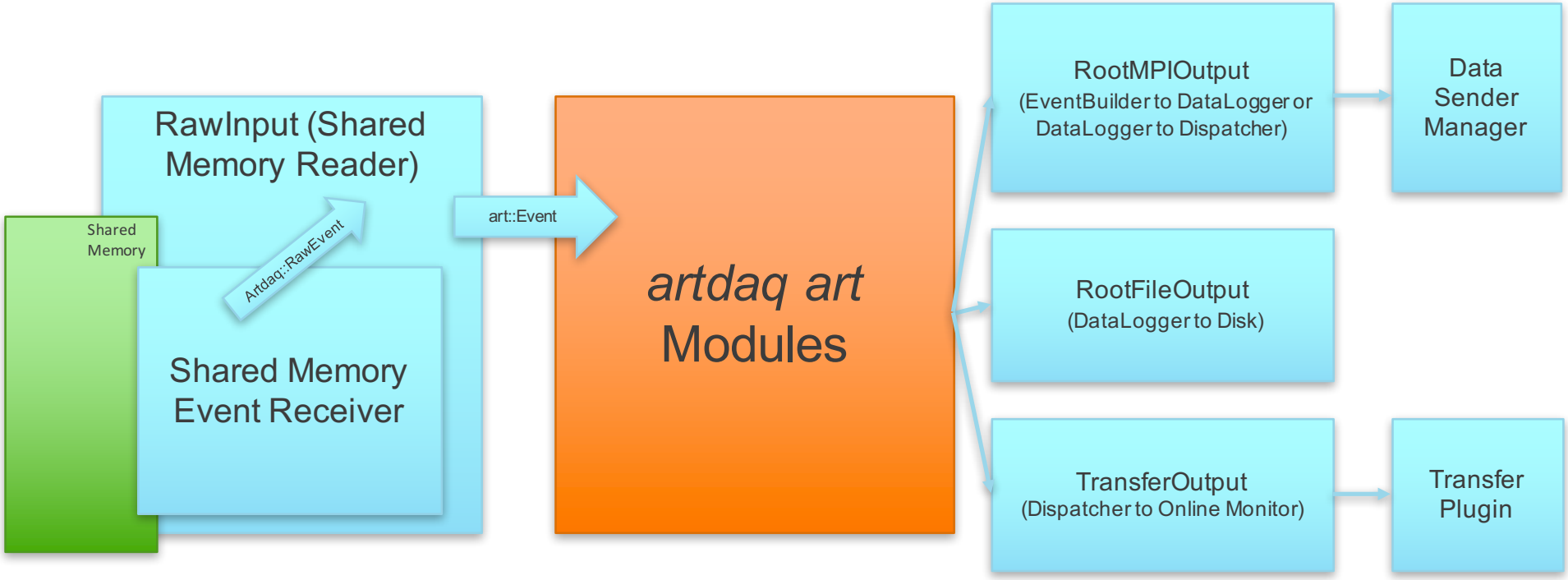
Overwrite…											
	E8										
FULL	E7	E7 F1	E7 F2	E7 F3	E7 F4	E7 F5	E7 F6	E7 F7	E7 F8	E7 F9	E7 FA	E7 FB
FULL	E3	E3 F1	E3 F2	E3 F3	E3 F4	E3 F5	E3 F6	E3 F7	E3 F8	E3 F9	E3 FA	E3 FB
WRITING (Active)	E9	E9 F1										
FULL	E5	E5 F1	E5 F2	E5 F3	E5 F4	E5 F5	E5 F6	E5 F7	E5 F8	E5 F9	E5 FA	
WRITING (Inactive)	EA											

SHAREDMEMORY BUFFER FLOW

- ...and fill with the new event
  - *Option, by default, is to allow for overwriting FULL buffers (but not READING buffers) if there are no Inactive buffers left*

WRITIING (Active)	**EB**	EB F1										
FULL	**E7**	E7 F1	E7 F2	E7 F3	E7 F4	E7 F5	E7 F6	E7 F7	E7 F8	E7 F9	E7 FA	E7 FB
FULL	**E3**	E3 F1	E3 F2	E3 F3	E3 F4	E3 F5	E3 F6	E3 F7	E3 F8	E3 F9	E3 FA	E3 FB
WRITING (Active)	**E9**	E9 F1										
FULL	**E5**	E5 F1	E5 F2	E5 F3	E5 F4	E5 F5	E5 F6	E5 F7	E5 F8	E5 F9	E5 FA	
WRITING (Inactive)	**EA**											

DIAGRAM OF THE ART SIDE



What happens in art, generally

- **Data comes in via "input source"**
  - Normal offline art jobs: this is a ROOT file with art::Events
  - "EmptyEvent" used for simulation generation
  - ***Dedicated input source for reading in from shared memory: SharedMemoryReader source***
    - *Takes in raw data and creates art::Event*
- **Producer/Filter modules run in "trigger path" over art::Events**
  - Multiple trigger paths allowed without running modules twice
- **Events passing trigger paths sent to output modules**
  - All trigger paths must finish first
  - Multiple output modules allowed

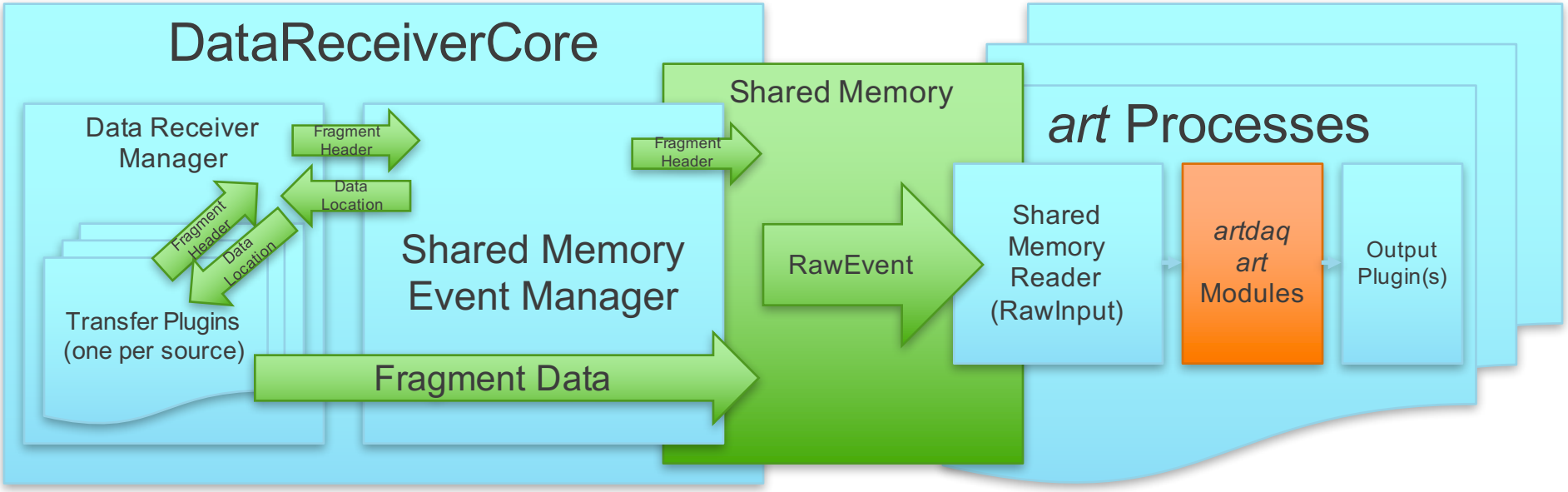
SharedMemoryReader Source

- Produces collections of artdaq::Fragments in art::Event
  - *Artdaq::Fragments have instance labels of fragment type or "ContainerXXX" type*
- readNext routine checks for data indefinitely
  - *There is a timeout, but by default resume back in loop automatically after a timeout*
- Returns and quits art thread if
  - *No resume after timeout (and we timeout)*
  - *Get an event with no fragments*
  - *First fragment type is EndOfData*
- Handles art needs with EndOfRun and EndOfSubrun fragments too

ROOTMPIOUTPUT MODULE

- Output module used for handling sending art::Events between *artdaq* processes
  - *EventBuilder→DataLogger, DataLogger→Dispatcher*
- Two main parts
  - *Handling and packaging of the art::Event into a binary blob inside an artdaq::Fragment*
  - *The data transport, done via the NetMonTransportService*
- NetMonTransportService is global scope element in art to handle data transport (incoming and outgoing)
  - *Contains DataSenderManager for outgoing, with TransferInterface plugins (only TPC and SHMEM work right now)*
  - *Contains SharedMemoryReader piece for incoming*
  - *All ROOTMPIOutput modules in an art job will share the NetMonTransportService*
  - *Note: DataSenderManager is same as in BRs, meaning it can use a RoutingMaster as well*

DIAGRAM OF THE DATALOGGER



DATALOGGERCORE

- Same logic for EventBuilder as DataLogger
  - *DataReceiverCore holds a DataReceiverManager and SharedMemoryEventManager (and buffer)*
- DRM runs threads for each TransferInput source
- SMEM launches and monitors *art* process, which in turn decide their output module location
- SMEM handling/flow works in same way as EventBuilder
  - *Simplified that there is now only 1 fragment per event, containing the fully built event*
- Input source for art threads reads from shared memory and simply copies/unpacks the fragment data payload into new art::Event
  - *Specifying all the necessary art internals when it does*

RootOutput Module

- art already provides an output module to write a ROOT file: RootOutputModule
- Options for …
  - *ROOT compression level*
  - *When to create new files (per n events, per subrun, max file size, etc.)*
  - *Objects to drop (not write to file)*
  - *Trigger paths from which to include events (default is all events at end of all paths)*