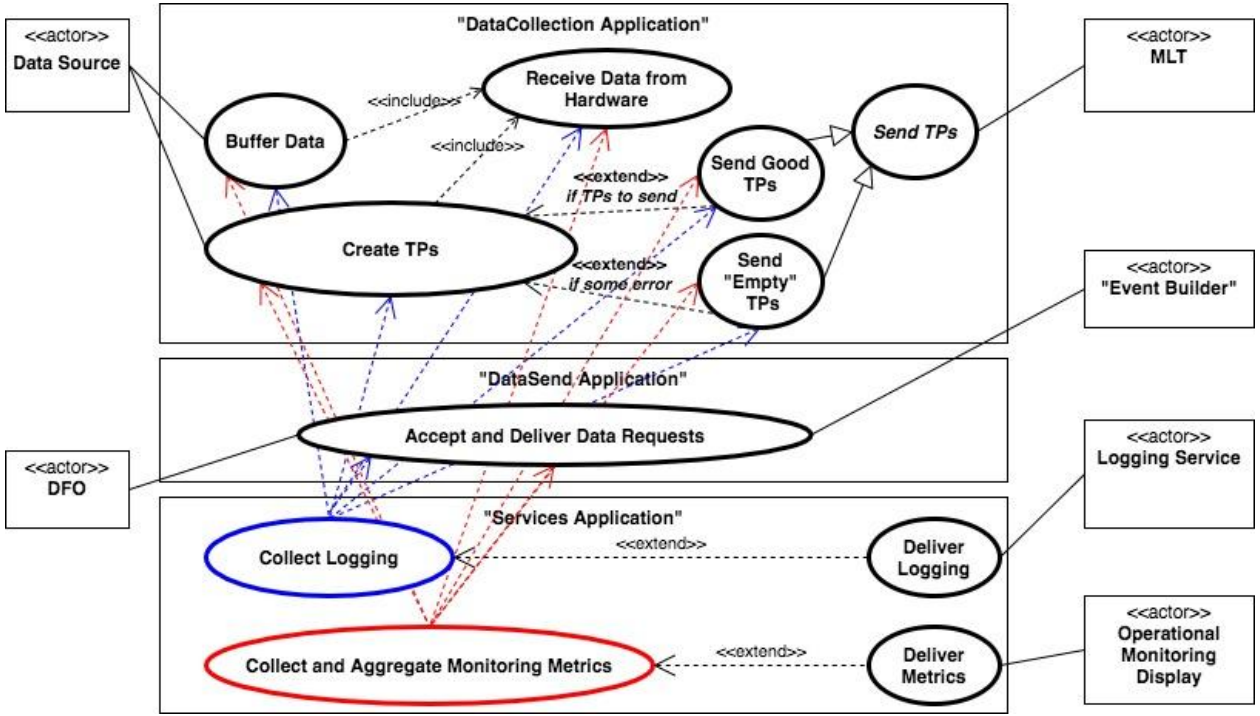
Further BoardReader Use Case Thoughts

Wes Ketchum

12 Feb 2020

From yesterday



Next steps...

Try to think explicitly how one would write user modules, and configure them to run a DAQ system

Still focus on *BoardReader* application where we do two basic things

1. Read data from hardware and store it in a local buffer
2. Upon data requests, find data in buffer and deliver data payload

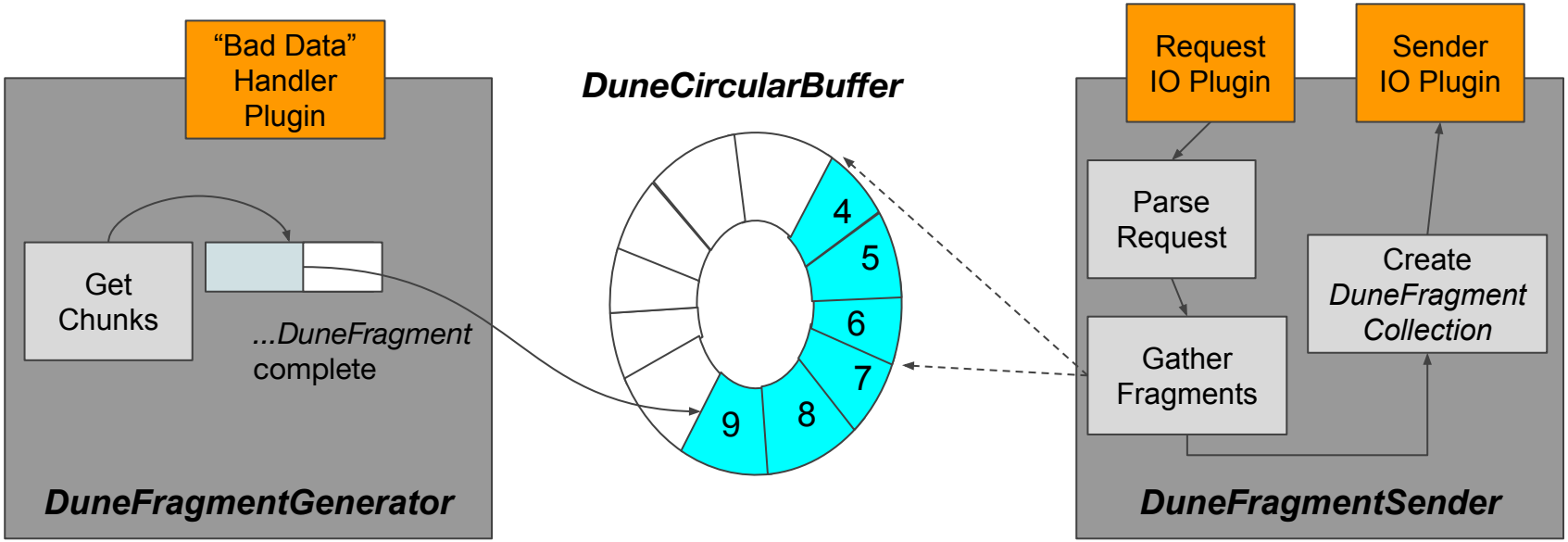
Sorry, no UML diagrams today

~~Alice~~Alessandro's Design Approach

Let's have two modules:

1. *DuneFragmentGenerator*
   - Receive chunks of data from the hardware and put them in some local memory
   - After each chunk, check to see if you have a complete *DuneFragment* of data
   - If you do, *create* that fragment, and put it into a *DuneCircularBuffer*---a circular buffer for *DuneFragment*s
2. *DuneFragmentSender*
   - Receive requests for *DuneFragment*s in some window of time
   - Go to the *DuneCircularBuffer* and find the appropriate *DuneFragment*s and create a *DuneFragmentCollection*
   - Send the *DuneFragmentCollection* out to the desired location

A pictorial view...



Few quick notes

- Alessandro rightfully makes use of common utilities that define request and data sending formats or interface to error/inhibit handling if we encounter bad format…
- ... *and* uses plugins so he can defer the exact implementation to runtime
  - There are other service-level plugins not mentioned here for certain: metrics, logging,

- Alessandro (or someone else) will have to also write the interface *DuneCircularBuffer*
  - Which must of course do appropriate locking/atomicity/etc, have concept of readers/writers, etc.

Alessandro's Configuration

```
dune_gen:{
      type: DuneFragmentGenerator
      data_in: none #it's hardcoded in there
      data_out: dune_circbuf
      err_handler_plugin: {}
      … }
dune_cirbuf: {
      type: DuneCircularBuffer
      data_writer: dune_gen
      data_readers: [ dune_fragsend ]
      … }
dune_fragsend: {
      type: DuneFragmentSender
      data_in: dune_circbuf
      data_out: @local::tcp_dg_sender
      req_plugin: {}
      … }
```

```
module_path:
[ dune_gen --> dune_circbuf --> dune_fragsend ]
```

~~Bob~~Bonnie's Design Approach

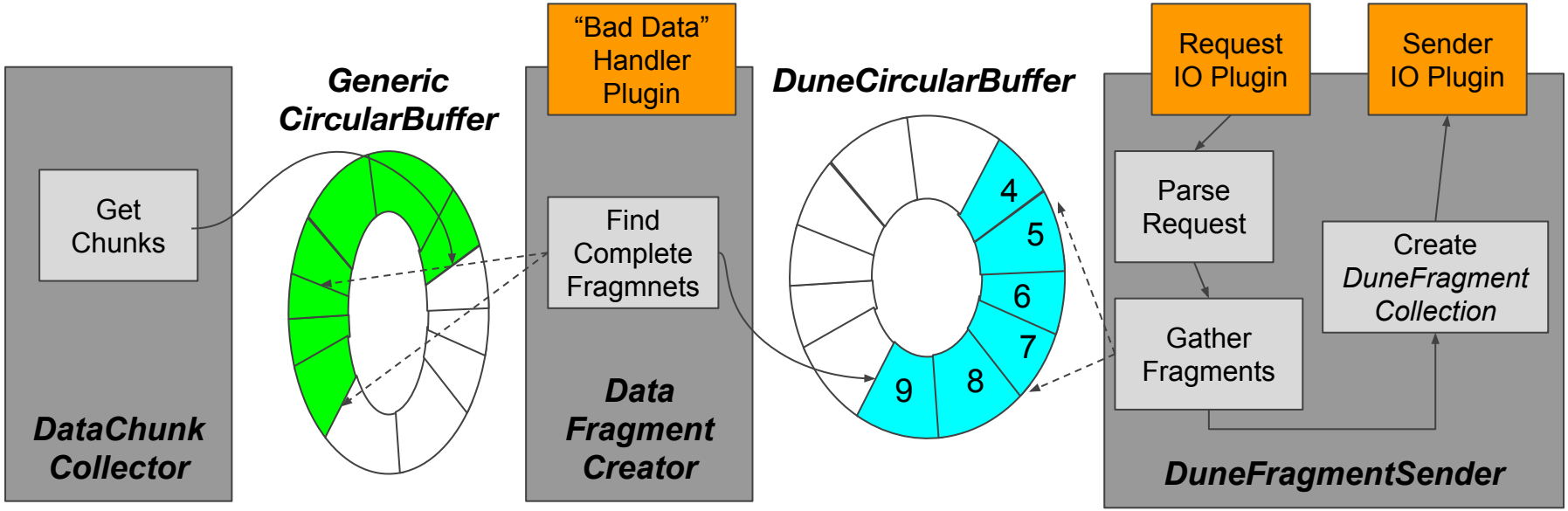
Bonnie likes Alessandro's *DataFragmentSender* module, but is worried about the *DataFragmentGenerator*

> *Isn't there a potentially slow, blocking condition between getting successive chunks, where you check to see if you have all the data to make a fragment?*

Instead, Bonnie decides to develop and use two other modules

1. *DataChunkCollector*
   - Collect chunks from the hardware and put them immediately in a *GenericCircularBuffer*
2. *DataFragmentCreator*
   - Read raw bytes from the *GenericCircularBuffer* and create *DuneDataFragments* (asynchronous to filling it), and put them into a *DuneCircularBuffer*

Bonnie's picture



Bonnie's Configuration

```
chunk_col:{
      type: DataChunkCollector
      data_in: none
      data_out: gen_circbuf }
gen_circbuf:{
      type: GenericCicularBuffer
      data_writer: chunk_col
      data_readers: [ dune_gen ] }
dune_gen:{
      type: DuneFragmentCreator
      data_in: gen_circbuf
      data_out: dune_circbuf
      err_handler_plugin: {}
      … }
dune_cirbuf: {
      type: DuneCircularBuffer
      data_writer: dune_gen
      data_readers: [ dune_fragsend ]
      … }
```

```
dune_fragsend: {
      type: DuneFragmentSender
      data_in: dune_circbuf
      data_out: @local::tcp_dg_sender
      req_plugin: {}
      … }
```

```
module_path:
[ chunk_col     -->
  gen_circbuf   -->
  dune_gen      -->
  dune_circbuf  -->
  dune_fragsend ]
```

Few quick notes

- There's a choice here to make this two modules, rather than a multi-threaded single module
  - Is it the right choice? What defines the right choice?

- We now have two circular buffer types that need to do the same things but with different base data types: these could probably be combined into a templated circular buffer container class…
  - But how would that be manifested in configuration? Maybe that would just be too confusing?
  - Regardless, we need to watch out we don't use the wrong one/type! Who will check that and how in config?

~~Carol~~Karol's Design Approach

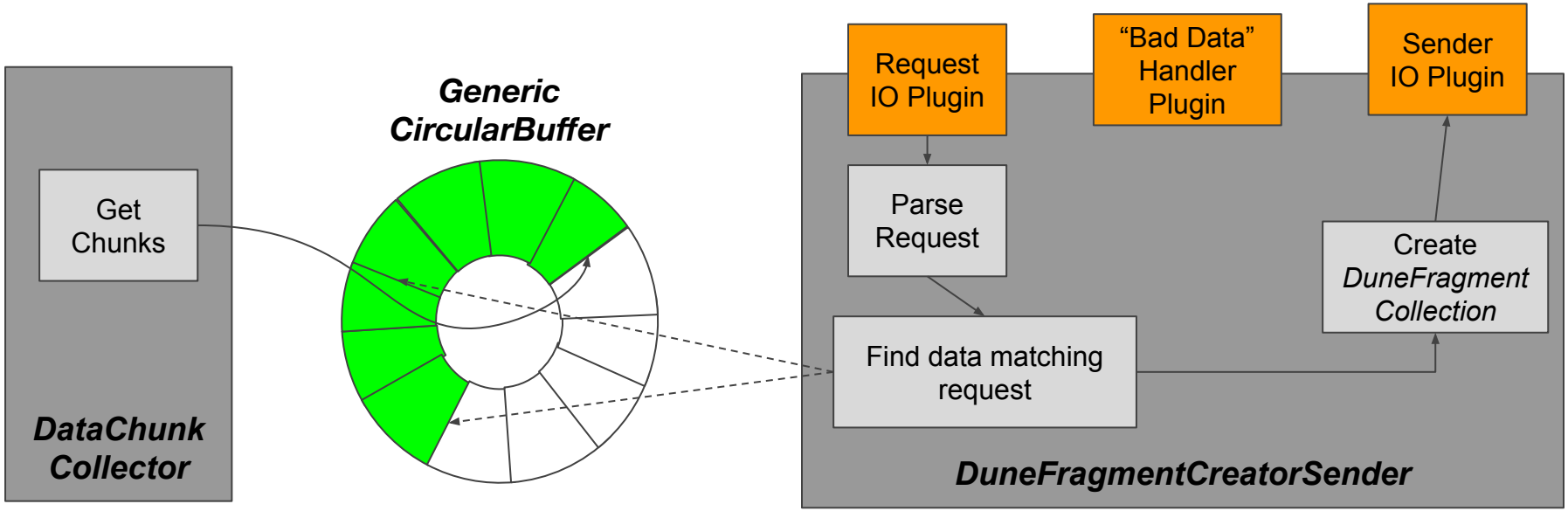
Karol agrees that unencumbered chunk collection is the way to go, but then wonders why we need this in-between *DataFragment* creation thing...

> *Aren't there lots of unnecessary memory copies? Aren't we wasting a lot of time processing data that will never be requested? If data is well formatted, can't we just traverse the bytes and find/package our desired data on the fly?*

So, Karol is now thinking of a new sender module:

- *DuneFragmentCreatorSender*
  - Upon request for data, traverse a *GenericCircularBuffer* using an overlay library to determine what data falls inside the desired timewindow, and package/send that data accordingly

Karol's picture



Karol's Configuration

```
chunk_col:{
      type: DataChunkCollector
      data_in: none
      data_out: gen_circbuf }

gen_circbuf:{
      type: GenericCicularBuffer
      data_writer: chunk_col
      data_readers: [ dune_fragsend ] }

dune_fragsend: {
      type: DuneFragmentCreatorSender
      data_in: gen_circbuf
      data_out: @local::tcp_dg_sender
      req_plugin: {}
      err_handler_plugin: {}
      … }
```

```
module_path:
[ chunk_col     -->
  gen_circbuf   -->
  dune_fragsend ]
```

In the end, thoughts?

- Three (and more) ways of constructing user modules to accomplish the same set of tasks
  - Not obvious that one is better than the other: depends on available resources and circumstances
  - This is good: I think we wanted to open up possibilities to developers
- Understanding the types in interfaces is necessary to ensuring a good configuration → many module changes will likely involve ‘slight’ interface changes
  - Especially early, we may have fewer single drop-in replacements, and more re-engineered ‘sub-chains’ of modules
- Regardless of a *ModuleManager* or not, we need methods to help people determine the types of interfaces expected by modules
  - Auto-generation of human-readable documentation via doxygen?
  - Lookup dictionaries/tables (machine-readable) created at compile-time to be used in config checking?
  - Automated checks on repository pull requests to make sure developers provide necessary details?