



art 3

Kyle J. Knoepfel
art stakeholders meeting
31 May 2018



Disclaimer

- “art 3” means the **first** released version of art with the major number 3 (e.g. 3.00.00). It does not refer to all versions with the major number 3.
- This talk is an introduction to *art 3*.
- It does not discuss all aspects of *art 3*.
- It is not intended to be formal documentation for *art 3*.
- It may contain mistakes.
- Some interface may yet change (not a lot though).

Outline

- Opening remarks
- *art* transitions and path processing
 - Consequences
- *art 3* introduction
 - Command-line invocation
 - Guarantees and limitations
 - Kinds of modules
 - Illustrations
 - Module interface
 - Services
- General breaking changes to legacy modules and to legacy services
- Guidance moving to *art 3*
- Next steps/down the road

September 2017 – first *art* MT forum meeting

Approaching the design

- The design of a multi-threaded framework should be based on fundamental principles, **not** on the limitations of external dependencies.
 - The relevant questions are:
 - In what contexts does multi-threading make sense?
 - In what contexts does multi-threading not make sense?
 - Not, when *can* we do multi-threading and when *can we not*.
 - The implementation must accommodate any limitations, not cater to them.
- We have striven for a balance between complexity and efficiency:
 - Our preference is to have a slightly less efficient, easier-to-understand system than a slightly more efficient, difficult-to-understand system.

September 2017 – first *art* MT forum meeting

Approaching the design

- The design of a multi-threaded framework should be based on fundamental principles, **not** on the limitations of external dependencies.

User interface

- *“No framework code is so precious that it must be saved; nothing is untouchable. Our users' code is precious, and we should break as little of it as possible.”*
- To the extent possible, the user interface should reflect only physically meaningful concepts—i.e. those concepts that are already known to *art* users: `Results`, `Run`, `SubRun`, and `Event`. It should not reflect implementation details.
 - Exceptions to this could be for those developing services or other more-expert facilities than modules.

September 2017 – first *art* MT forum meeting

Approaching the design

- The design of a multi-threaded framework should be based on fundamental principles, **not** on the limitations of external dependencies.

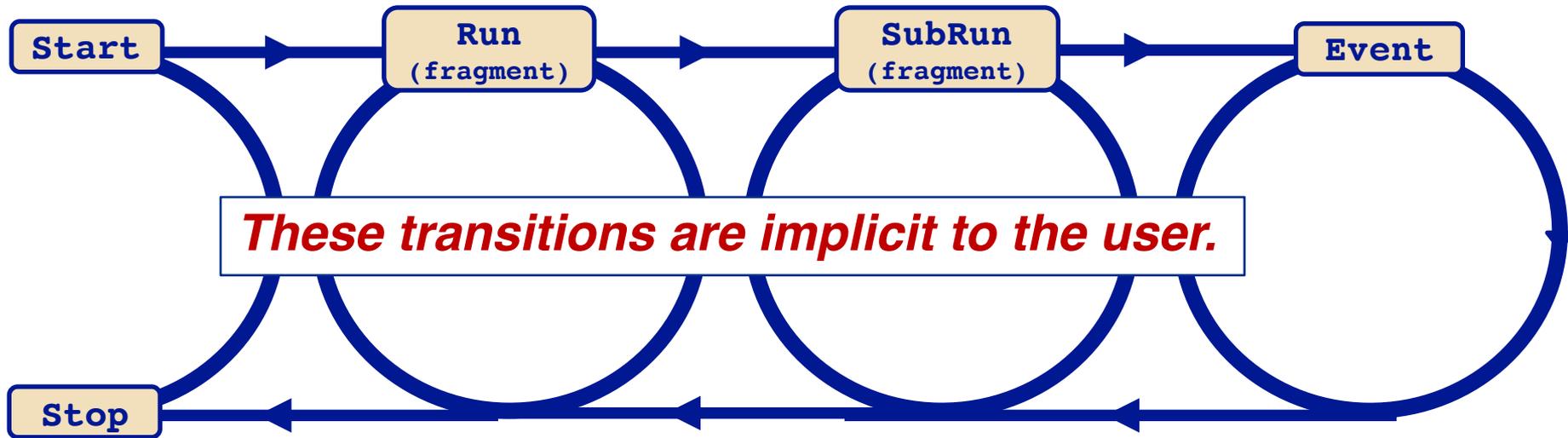
User interface

Nine months later, these principles have held:

- The design is based on fundamental principles, not limitations of external libraries.
- There are known inefficiencies that have been tolerated for the sake of simplicity and to avoid premature optimization. They can be improved upon in future releases.
- Except for issues directly coupled to MT execution, legacy modules and configurations will continue to work without modification. Typically, only rebuilding will be required.

Allowed transitions

- *art* is designed to process a hierarchy of data-containment levels:
 - *Run* \supset *SubRun* \supset *Event*
- *art* users expect the framework to respect this hierarchy
- The allowed transitions by the framework are thus:



Processing a data-containment level (e.g. Event)

- The order in which modules are executed for a Run, SubRun, or Event is determined by the **path declarations** in the configuration file.

```
physics: {  
  
  producers: {  
    makeHits: {...}  
    makeShowers: {...}  
    produceG4Steps: {...}  
  }  
  analyzers: {  
    plotHits: {...}  
  }  
  
  hitPath: [makeHits, makeShowers]  
  geomPath: [produceG4Steps]  
  analyzePath: [plotHits]  
}
```

Module declarations

Path declarations

Processing a data-containment level (e.g. Event)

- The order in which modules are executed for a Run, SubRun, or Event is determined by the **path declarations** in the configuration file.

```
physics: {  
  producers: {  
    makeHits: {...}  
    makeShowers: {...}  
    produceG4Steps: {...}  
  }  
  analyzers: {  
    plotHits: {...}  
  }  
}
```

Trigger path	hitPath: [makeHits, makeShowers]
Trigger path	geomPath: [produceG4Steps]
End path	analyzePath: [plotHits]

Processing a data-containment level (e.g. Event)

- The order in which modules are executed for a Run, SubRun, or Event is determined by the **path declarations** in the configuration file.

```
physics: {  
  producers: {  
    makeHits: {...}  
    makeShowers: {...}  
    produceG4Steps: {...}  
  }  
  analyzers: {  
    plotHits: {...}  
  }  
}
```

Trigger path	hitPath: [makeHits, makeShowers]
Trigger path	geomPath: [produceG4Steps]
End path	analyzePath: [plotHits]

- The order in which *trigger paths* are executed is unspecified (current *art*).
- In MT *art* trigger paths will be executed simultaneously.
- Modules in a trigger path are executed in the order specified.
- End paths are always processed after all trigger paths.
- A module is executed once per event.

Processing a data-containment level (e.g. Event)

- The order in which modules are executed for a Run, SubRun, or Event is determined by the **path declarations** in the configuration file.

```
physics: {  
  producers: {  
    makeHits: {...}  
    makeShowers: {...}  
    produceG4Steps: {...}  
  }  
  analyzers: {  
    plotHits: {...}  
  }  
}
```

Trigger path	hitPath: [makeHits, makeShowers]
Trigger path	geomPath: [produceG4Steps]
End path	analyzePath: [plotHits]

- The order in which *trigger paths* are executed is unspecified (current *art*).
- In MT *art* trigger paths will be executed simultaneously.
- Modules in a trigger path are executed in the order specified.
- End paths are always processed after all trigger paths.

Heeding these facts is essential for successful use of *art 3*.

Consequences of *art*'s guarantees

- Modules on one trigger path may not consume products created by modules that are not on that same path.

Consequences of *art*'s guarantees

- Modules on one trigger path may not consume products created by modules that are not on that same path.
- The following is a configuration error (heuristically):

```
physics: {  
  producers: {  
    p1: { produces: ["int", ""] }  
    p2: { consumes: ["int", "p1::current_process"] }  
  }  
  tp1: [p1]  
  tp2: [p2]  
}
```

Consequences of *art*'s guarantees

- Modules on one trigger path may not consume products created by modules that are not on that same path.
- The following is also a configuration error (heuristically):

```
physics: {  
  producers: {  
    p1: { produces: ["int", ""] }  
    p2: { produces: ["int", "instanceName"] }  
    readThenMake: {  
      consumesMany: ["int"] // calls getMany  
    }  
  }  
  tp1: [p1, readThenMake]  
  tp2: [p2, readThenMake]  
}
```

Consequences of *art*'s guarantees

- Modules on one trigger path may not consume products created by modules that are not on that same path.
- The following is also a configuration error (heuristically):

```
physics: {
```

art 3 catches these errors if you use the consumes interface.

```
Module readThenMake on paths tp1, tp2 depends on  
Module p2 on path tp2
```

```
consumes: {  
  // catches geometry  
}  
}  
tp1: [p1, readThenMake]  
tp2: [p2, readThenMake]  
}
```

art 3

Multi-threaded event-processing

- *art 3* supports concurrent processing of events.
 - The number of events to process concurrently is specified by the **number of schedules**
 - The user can optionally specify the number of threads.
- The user ***opts in*** to concurrent processing.

Multi-threaded event-processing

- *art* 3 supports concurrent processing of events.
 - The number of events to process concurrently is specified by the **number of schedules**
 - The user can optionally specify the number of threads.
- The user ***opts in*** to concurrent processing.

(nSch, nThr)	Command
(1, 1)	<code>art -c <config> ...</code>
(1, 1)	<code>art -c <config> -j 1 ...</code>
(4, 4)	<code>art -c <config> -j 4 ...</code>
(nproc, nproc)	<code>art -c <config> -j 0 ...</code>
(1, 4)	<code>art -c <config> --nschedules 1 --nthreads 4 ...</code>

- In a grid environment, number of threads is limited to the number of CPUs configured for the HTCondor slot (*art* adjusts the number of threads).

art 3 guarantees

- Processing of an event happens on one and only one schedule.
- For a given trigger path, modules are processed in the order specified.
- A module shared among paths will be processed only once per event.
- Product insertion into the event is thread-safe.
- Product retrieval from the event is thread-safe.
- Provenance retrieval from the event is thread-safe.
- All modules and services provided by *art* are thread-safe.
 - For `TFileService`, the user is required to specify additional serialization.

art 3 limitations— *Primum non nocere* (first, to do no harm)

- Only events within the same SubRun are processed concurrently.
- Analyzers and output modules do not run concurrently.
- MixFilter modules are legacy modules.
- Secondary input-file reading is allowed only for 1 schedule and 1 thread.
- TFileService file-switching is allowed only for 1 schedule and 1 thread.

Kinds of modules in *art* 3

- *art* guarantees that any currently-existing modules (to within some interface changes) will be usable in a multi-threaded execution of *art*.
 - No multi-threading benefits will be realized with such “legacy” modules
- To take advantage of *art*'s multi-threading capabilities, users will need to choose the kind of module they use:
 - **Shared module**: sees all events—calls can be serialized or asynchronous.
 - **Replicated module**: for a configured module, one copy of that module is created per schedule—each module copy sees one event at a time. Use if moving to a concurrent, shared module is not feasible.

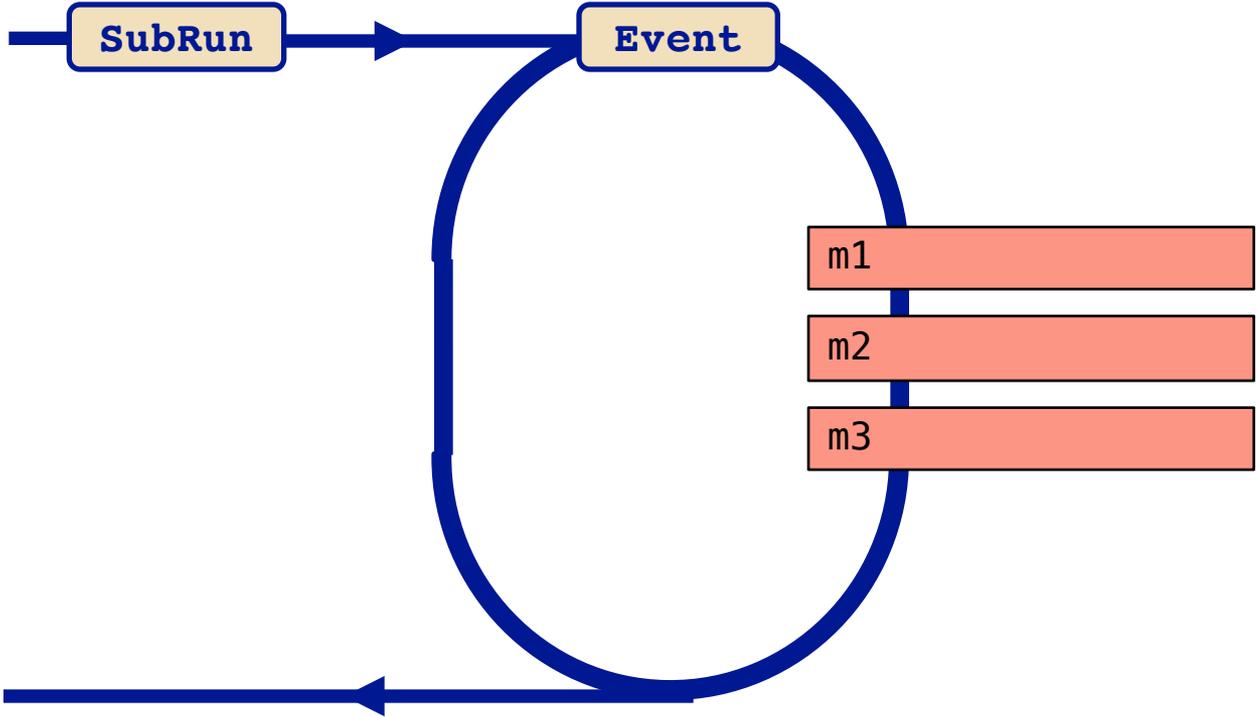
Time structure for calling modules

Single schedule (current *art*)



Time structure for calling modules

Single schedule (current *art*)

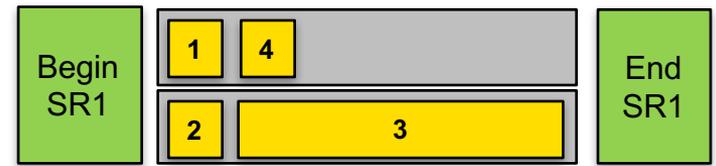


Shared modules

Modules shared across schedules

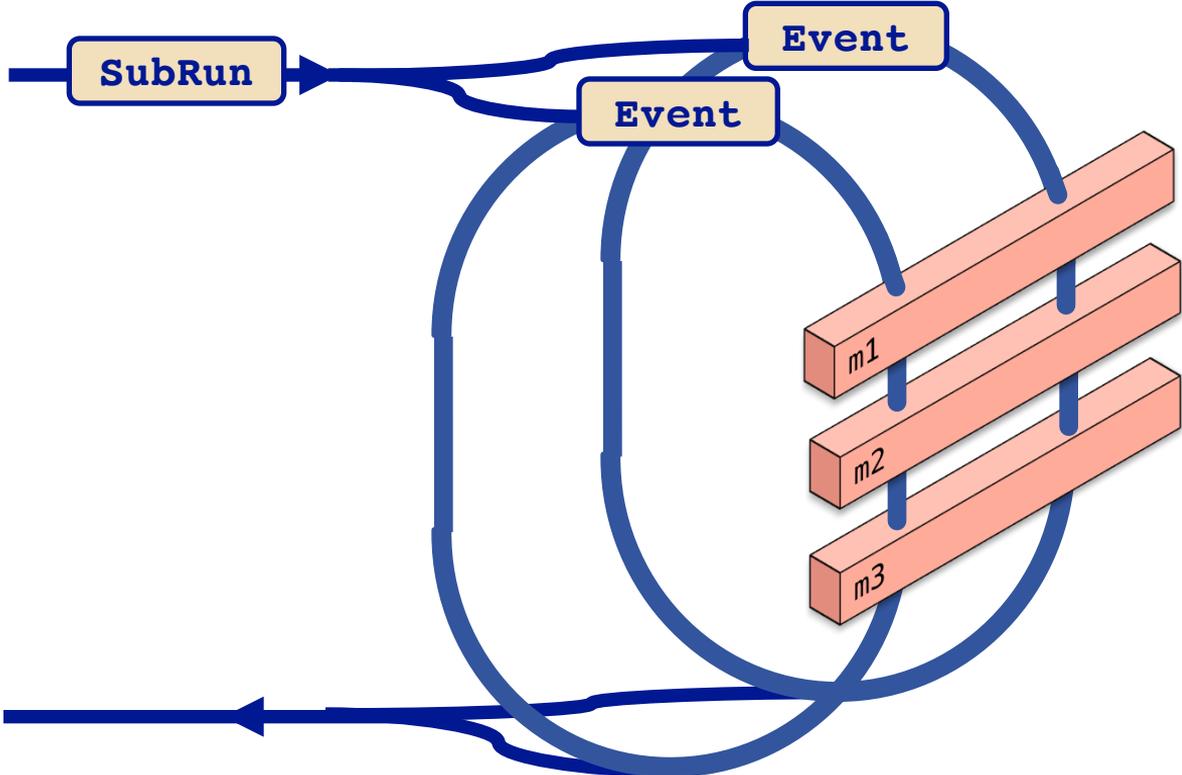
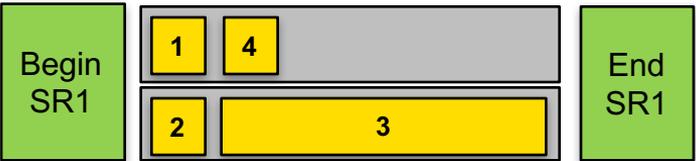
Time structure for calling modules

Multiple schedules (*art* 3.0)



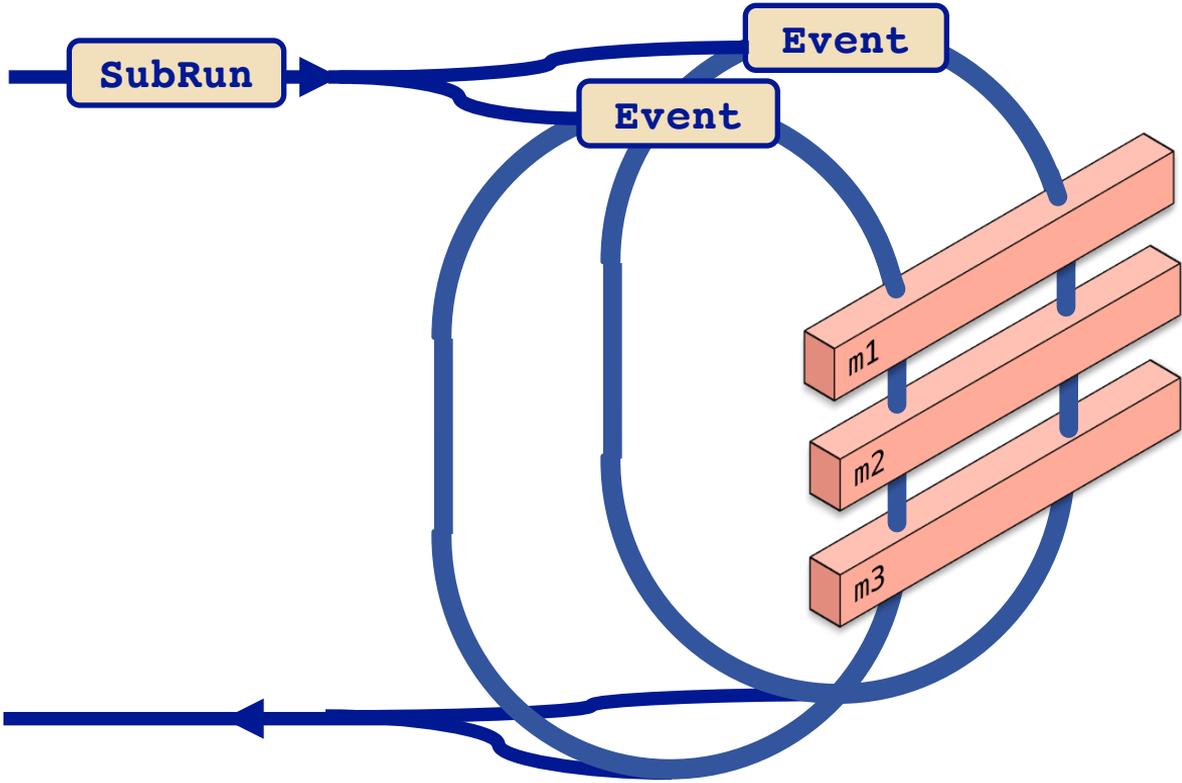
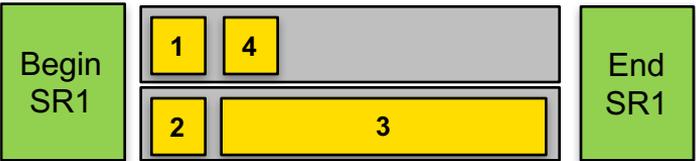
Time structure for calling modules

Multiple schedules (art 3.0)



Time structure for calling modules

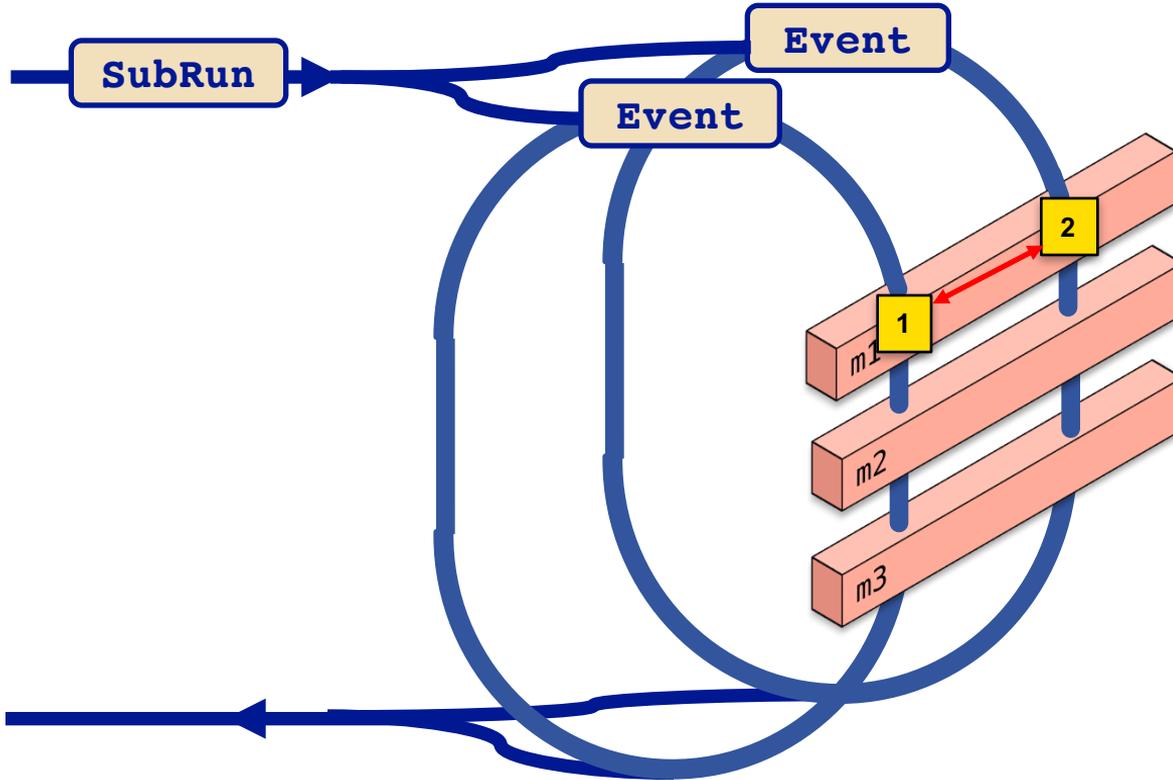
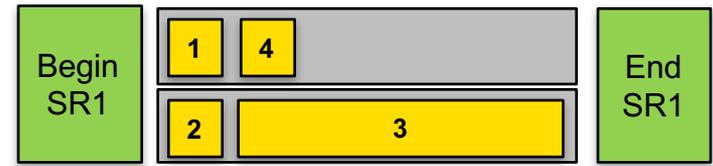
Multiple schedules (*art* 3.0)



Data races are now possible.

Time structure for calling modules

Multiple schedules (*art* 3.0)



If the state of one of the modules is updated when simultaneously processing two events, there can be a data race.

What are some ways to handle this?

Use a “legacy” module

```
class HistMaker : public art::EDProducer {
public:
  explicit HistMaker(Parameters const& p) : EDProducer{p}
  {}

  void produce(Event& e) override {} // Called serially wrt. all
                                     // serialized modules
};
```

- Legacy modules imply maximum serialization.
 - Legacy modules cannot be run in parallel with any other legacy modules or any serialized shared modules.
- With *art 3*, any new modules should not be legacy modules.
- The better solution is to use a `SharedModule`, which can be serialized only wrt itself.

Use a shared module

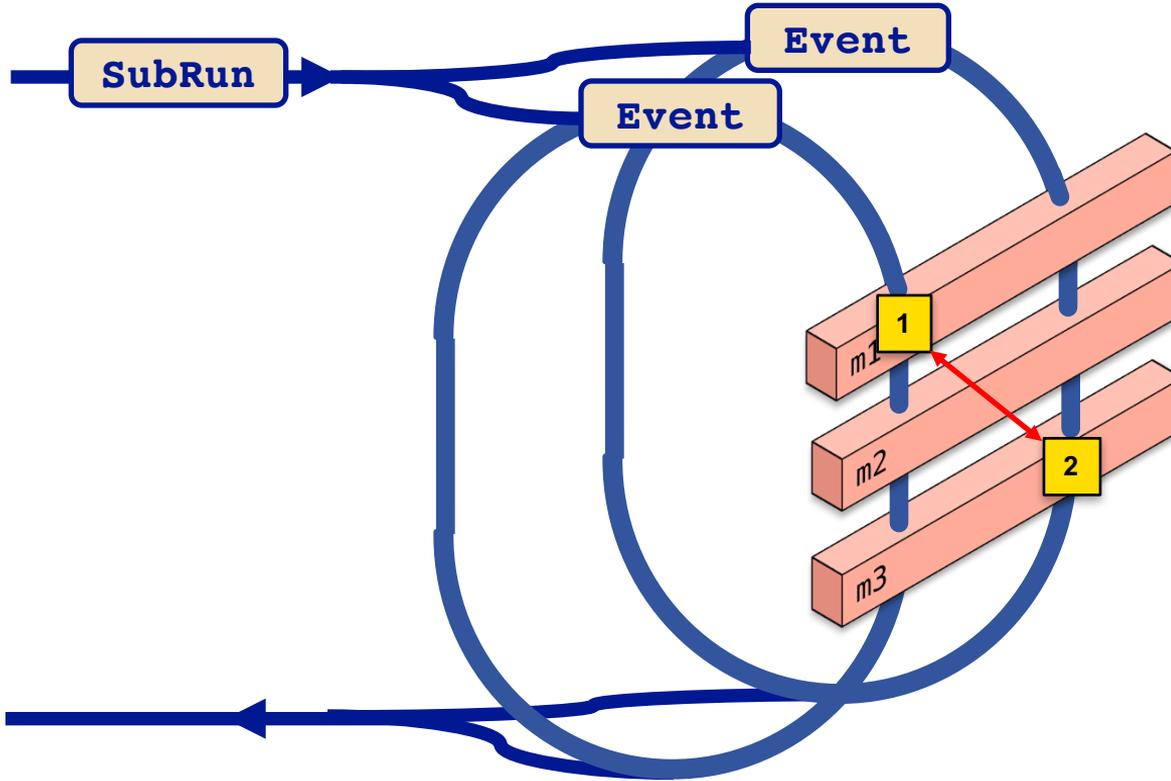
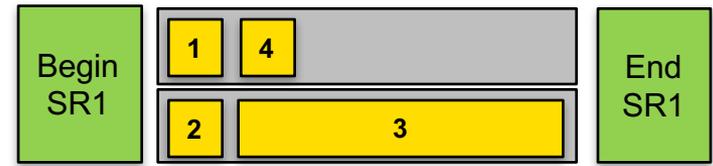
```
class HistMaker : public art::SharedProducer {
public:
    explicit HistMaker(Parameters const& p) : SharedProducer{p}
    {
        serialize<InEvent>(); // Declaration to process
                             // one event at a time.
    }

    void produce(Event&) override {} // Called serially wrt. itself
};
```

- But there can be other data race problems.

Time structure for calling modules

Multiple schedules (*art* 3.0)



If two modules are processing different events at the same time, but they are using a common resource, there can be a data race.

How do we avoid such a data race?

Serialized module due to shared resource

```
class Fitter : public art::shared::Producer {  
public:  
    explicit Fitter(Parameters const& p)
```

Suppose you want to call `TCollection::(Set|Get)CurrentCollection`
First step: please don't. This is only illustrating a thread-unsafe interface.

```
    // Called serially wrt. other modules that use TCollection  
    void produce(Event& e) override {}  
};
```

Serialized module due to shared resource

```
class Fitter : public art::SharedProducer {
public:
    explicit Fitter(Parameters const& p) : SharedProducer{p}
    {
        serialize<InEvent>("TCollection"); // Declare the common resource
    }

    // Called serially wrt. other modules that use TCollection
    void produce(Event& e) override {}
};
```

- We are working on a way to standardize the arguments to `serialize`.

If you can guarantee no data races...

```
class HitMaker : public art::SharedProducer {
public:
    explicit HitMaker(Parameters const& p) : SharedProducer{p}
    {
        async<InEvent>();
    }

    void produce(Event& e) override {} // Called asynchronously
};
```

Replicated modules

One module per schedule

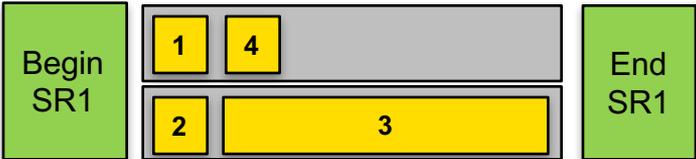
Replicated modules

One module per schedule

- Sometimes the easiest way to gain multi-threading benefits is to replicate modules across schedules—avoids data races from sharing a module.

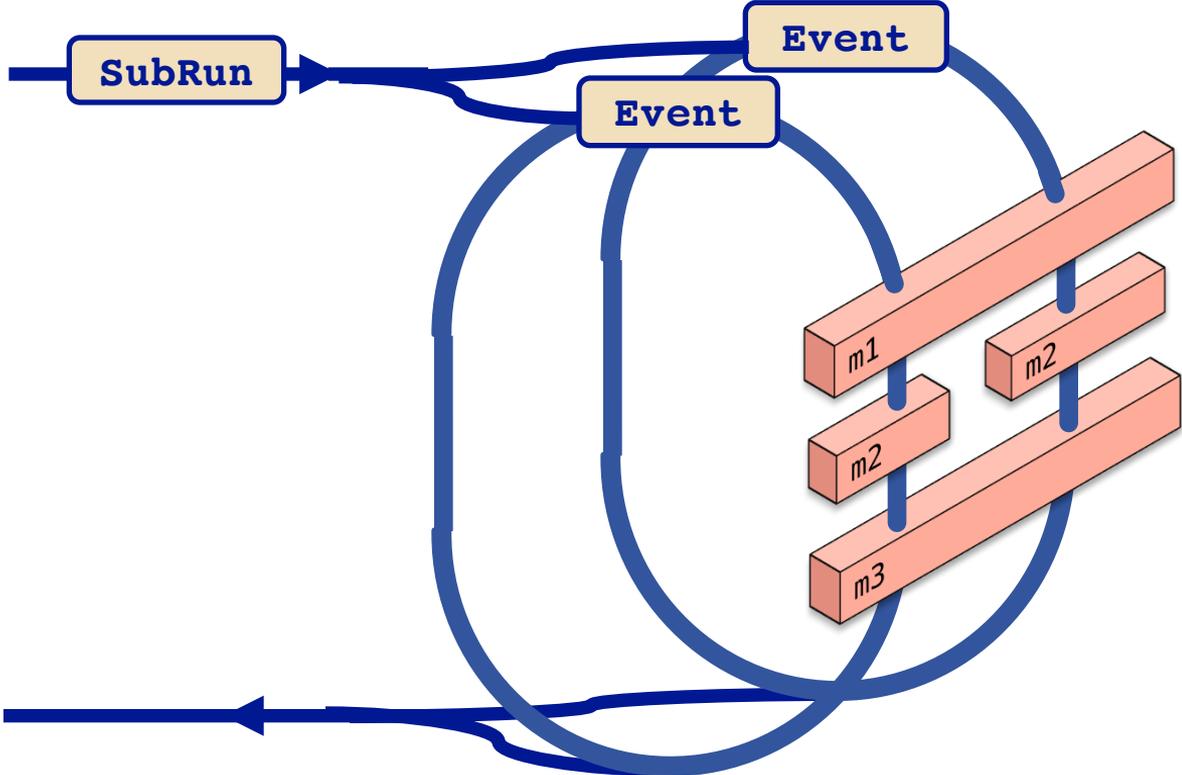
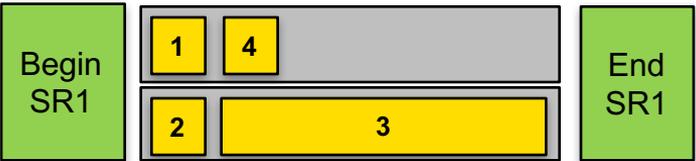
Time structure for calling modules

Multiple schedules (*art* 3.0)



Time structure for calling modules

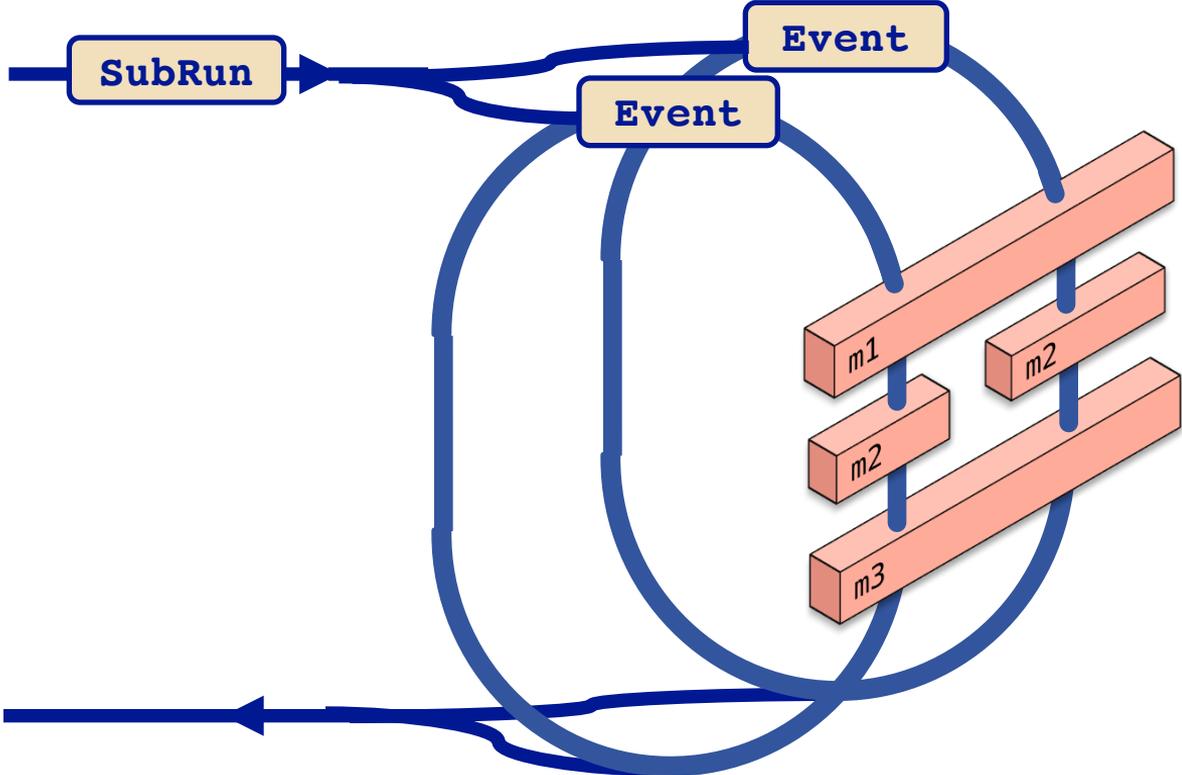
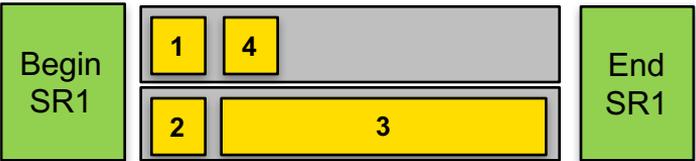
Multiple schedules (*art* 3.0)



Multiple copies of configured module **m2** avoids data-races wrt. **m2** data members.

Time structure for calling modules

Multiple schedules (*art* 3.0)



Multiple copies of configured module **m2** avoids data-races wrt. **m2** data members.

Consequence: each module copy does not see all events.

Replicated producer

```
class Accumulator : public art::ReplicatedProducer {
public:
    explicit Accumulator(Parameters const& p)
        : ReplicatedProducer{p}
    {}

    // Each module copy sees one event at a time
    void produce(Event& e) override;
};
```

- Do not use a replicated producer if you need to use a shared resource.
- For *art* 3.0, replicated modules cannot produce Run and SubRun data products.

Producer virtual member functions

EDProducer	SharedProducer	ReplicatedProducer
void beginJob()	void beginJob(Services const&)	void beginJob(Services const&)
void beginRun(Run&)	void beginRun(Run&, Services const&)	void beginRun(Run const& , Services const&)
void beginSubRun(SubRun&)	void beginSubRun(SubRun&, Services const&)	void beginSubRun(SubRun const& , Services const&)
void produce(Event&)	void produce(Event&, Services const&)	void produce(Event&, Services const&)
void endSubRun(SubRun&)	void endSubRun(SubRun&, Services const&)	void endSubRun(SubRun const& , Services const&)
void endRun(Run&)	void endRun(Run&, Services const&)	void endRun(Run const& , Services const&)
void endJob()	void endJob(Services const&)	void endJob(Services const&)

- A produce override is required; all others are optional.

Filter virtual member functions

EDFilter	SharedFilter	ReplicatedFilter
<code>void beginJob()</code>	<code>void beginJob(Services const&)</code>	<code>void beginJob(Services const&)</code>
<code>bool beginRun(Run&)</code>	<code>void beginRun(Run&, Services const&)</code>	<code>void beginRun(Run const&, Services const&)</code>
<code>bool beginSubRun(SubRun&)</code>	<code>void beginSubRun(SubRun&, Services const&)</code>	<code>void beginSubRun(SubRun const&, Services const&)</code>
<code>bool filter(Event&)</code>	<code>bool filter(Event&, Services const&)</code>	<code>bool filter(Event&, Services const&)</code>
<code>bool endSubRun(SubRun&)</code>	<code>void endSubRun(SubRun&, Services const&)</code>	<code>void endSubRun(SubRun const&, Services const&)</code>
<code>bool endRun(Run&)</code>	<code>void endRun(Run&, Services const&)</code>	<code>void endRun(Run const&, Services const&)</code>
<code>void endJob()</code>	<code>void endJob(Services const&)</code>	<code>void endJob(Services const&)</code>

- A filter override is required; all others are optional.

Analyzer virtual member functions

EDAnalyzer	SharedAnalyzer	ReplicatedAnalyzer
<code>void beginJob()</code>	<code>void beginJob(Services const&)</code>	<code>void beginJob(Services const&)</code>
<code>void beginRun(Run const&)</code>	<code>void beginRun(Run const&, Services const&)</code>	<code>void beginRun(Run const&, Services const&)</code>
<code>void beginSubRun(SubRun const&)</code>	<code>void beginSubRun(SubRun const&, Services const&)</code>	<code>void beginSubRun(SubRun const&, Services const&)</code>
<code>void analyze(Event const&)</code>	<code>void analyze(Event const&, Services const&)</code>	<code>void analyze(Event const&, Services const&)</code>
<code>void endSubRun(SubRun const&)</code>	<code>void endSubRun(SubRun const&, Services const&)</code>	<code>void endSubRun(SubRun const&, Services const&)</code>
<code>void endRun(Run const&)</code>	<code>void endRun(Run const&, Services const&)</code>	<code>void endRun(Run const&, Services const&)</code>
<code>void endJob()</code>	<code>void endJob(Services const&)</code>	<code>void endJob(Services const&)</code>

- An analyze override is required; all others are optional.

What is the Services type?

“O art::ServiceHandle<T>{}, thou time is short.”
- Anonymous

- Until now, users have been able to create ServiceHandles from anywhere.
- With *art 3*, this pattern is changing.
- The recommended pattern will be for *art* users to create service handles from the passed-in Services object.

```
void HitMaker::produce(Event&, Services const& services)
{
    ServiceHandle<Calib> calibH = services.getHandle<Calib>();
}
```

- This will eventually allow for replicated services, akin to replicated modules.
- ServiceHandles can still be constructed anywhere, but that will eventually change.

Breaking changes to legacy modules and to services

Breaking changes for legacy modules

- For producers and filters that call `createEngine`, you must explicitly call the non-default constructor for `EDProducer` and `EDFilter`.

```
// art 2
RNGProducer(Parameters const& p)
  : dist_{createEngine(p().seed())}
{}

// art 3
RNGProducer(Parameters const& p)
  : art::EDProducer{p} // must specify base class c'tor
  , dist_{createEngine(p().seed())}
{}

```

- This change will become necessary for all `EDProducer` and `EDFilter` modules in a later version of *art*.
- All shared and replicated modules require calling similar base class constructors.

Breaking changes for services

- Through *art 2*, `RandomNumberGenerator` has had a concept of the “current” module being processed:

```
ServiceHandle<RNG>{}->getEngine();  
ServiceHandle<RNG>{}->getEngine("the_other_one");
```

- In *art 3*, there is no longer any “current” module. The equivalent interface is:

```
ServiceHandle<RNG> rng{};  
rng->getEngine(scheduleID, moduleLabel);  
rng->getEngine(scheduleID, moduleLabel, "the_other_one");
```

- If all engines are retrieved using the `createEngine` interface, then `getEngine` can be removed, and the correct engine can be given by reference to the functions that need it.
 - Direct access to the `RandomNumberGenerator` service is no longer needed.

Breaking changes for services

- Service callback signature changes:

Signal	art 2	art 3
sPreSourceEvent	void()	void(ScheduleContext)
sPostSourceEvent	void(Event const&)	void(Event const&, ScheduleContext)
sPreProcessEvent	void(Event const&)	void(Event const&, ScheduleContext)
sPostProcessEvent	void(Event const&)	void(Event const&, ScheduleContext)
sPreWriteEvent	void(ModuleDescription const&)	void(ModuleContext const&)
sPostWriteEvent	void(ModuleDescription const&)	void(ModuleContext const&)
sPreProcessPath	void(string const&)	void(PathContext const&)
sPostProcessPath	void(string const&, HLTPathStatus const&)	void(PathContext const&, HLTPathStatus const&)
sPreModule*	void(ModuleDescription const&)	void(ModuleContext const&)
sPostModule*	void(ModuleDescription const&)	void(ModuleContext const&)

Breaking changes for services

- Services must be thread-safe.

ROOT and MT

- ROOT's thread-safety flag **has** been enabled by *art*.
 - Allows (e.g.) multiple ROOT files to be opened in parallel.
- ROOT's implicit MT flag **has not** been enabled by *art*.
- All interactions *art* has with ROOT are serialized.
 - Input-file reading
 - Output-file writing
 - To use TFileService, you must use a shared module that calls the appropriate `serialize` function.

Guidance moving to *art* 3

- **Solve workflow issues first.**
 - You might have thread-safe modules and services.
 - If you're relying on illegal path configurations, you'll run into product dependency errors.

Guidance moving to *art* 3

- **Solve workflow issues first.**
 - You might have thread-safe modules and services.
 - If you're relying on illegal path configurations, you'll run into product dependency errors.

Recompile/rerun jobs with 1 schedule/1 thread
(default)

Add consumes statements to modules
(use `-M` program option for help)

Recompile/rerun jobs with 1 schedule/1 thread
and use `--errorOnMissingConsumes`

Recompile/rerun jobs with more than 1
schedule/1 thread

Guidance moving to *art* 3

- **Solve workflow issues first.**
 - You might have thread-safe modules and services.
 - If you're relying on illegal path configurations, you'll run into product dependency errors.
- **Determine what kind of module you need.**
 - Producer, filter, or analyzer?
 - Do you need to create (Sub)Run products?
 - Do you need to see every event?
 - Do you need to call an external library that is not thread-safe?
 - Do you have mutable data members for which operations are not thread-safe?

Guidance moving to *art* 3

- **Solve workflow issues first.**
 - You might have thread-safe modules and services.
 - If you're relying on illegal path configurations, you'll run into product dependency errors.
- **Determine what kind of module you need.**
 - Producer, filter, or analyzer?
 - Do you need to create (Sub)Run products?
 - Do you need to see every event?
 - Do you need to call an external library that is not thread-safe?
 - Do you have mutable data members for which operations are not thread-safe?
- We are working to provide guidance in dealing with such issues.
- **Contact us.**

Next steps/down the road

Next steps

- Expect *art-3* tag in next few days.
- Documentation!
- Will work with Mu2e to demonstrate scalability.
- The SciSoft team's direct involvement with LArSoft means that we will help LArSoft as well.
- If you are interested in upgrading your code to benefit from *art 3*, please contact us.

art 3.01

- Only C++17 builds provided
- `EDProducer` and `EDFilter` default constructors will be deprecated
- Global `errorOnFailureToPut` parameter/program-option will be deprecated

art 3.02

- *art* will begin using C++17 features
- `EDProducer` and `EDFilter` default constructors will be removed
- Global `errorOnFailureToPut` parameter/program-option will be removed
- `RandomNumberGenerator::getEngine` member function will be deprecated

art 3.xy

- Many other issues we have in the books that have taken a back seat
 - Improve exception format
 - Update stored SAM metadata
 - Disentangling *art* from unnecessary ROOT dependencies
 - etc.