Getting Started with DDS

[In C++, Java and Scala]

Angelo CORSARO, Ph.D.
Chief Technology Officer
OMG DDS Sig Co-Chair
PrismTech
angelo.corsaro@prismtech.com

OpenSplice | DDS

General Information

- This tutorial will get you started with DDS. At the end of this course you should have a firm grip of DDS concepts and the capacity of designing and writing DDS applications
- The tutorial will be highly interactive and provide plenty of examples and live demonstrations
- The tutorial will cover the new C++ and Java API
- The tutorial will also introduce you into distributed functional programming with **Scala** and **DDS**

Outline

- Background
- DDS Basics
- Data Reader/Writer Caches
- DDS Quality of Service
- Data & State Selectors
- Advanced Topics in DDS
- Concluding Remarks

OpenSplice | DDS

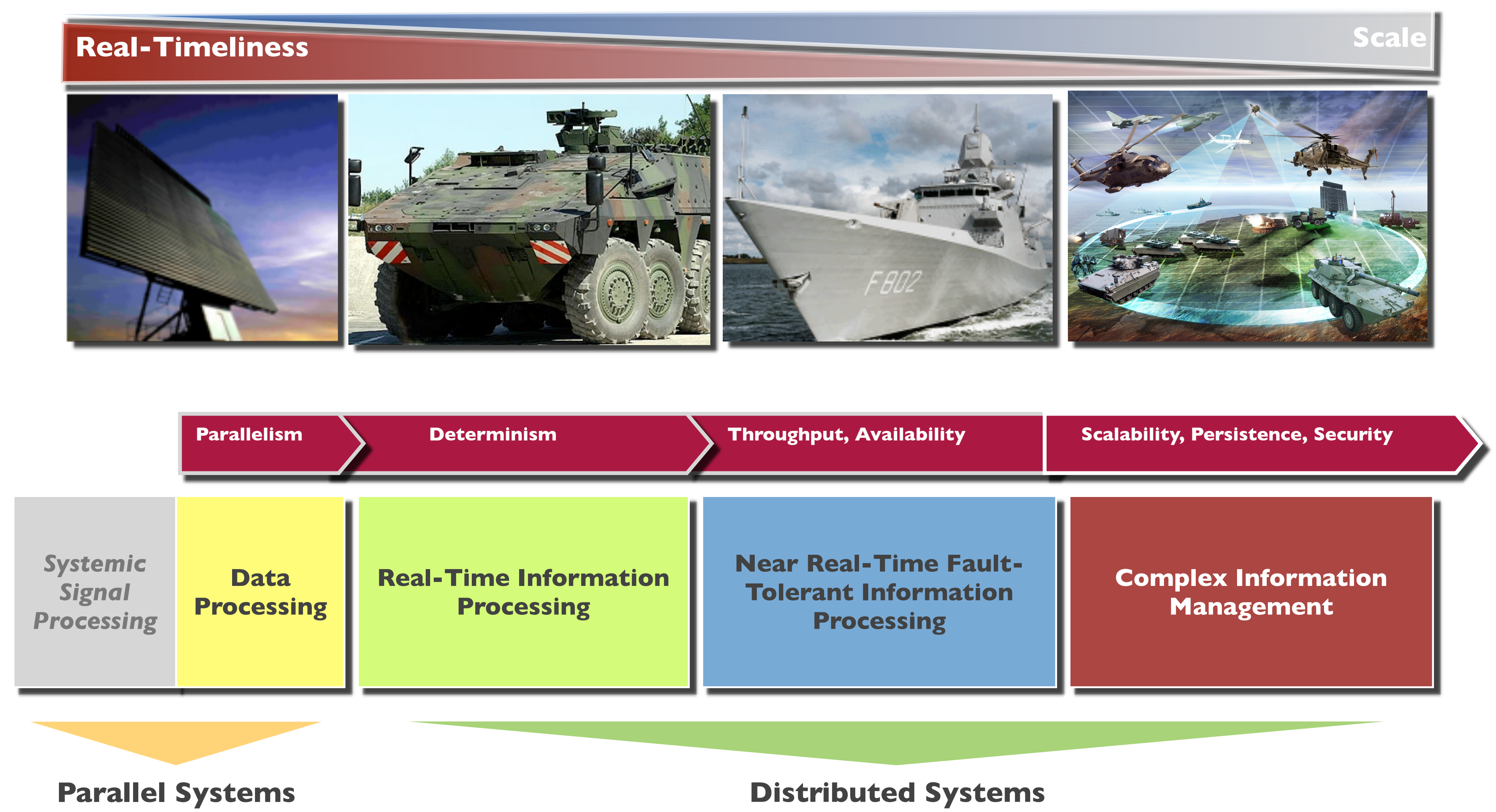
Background

Data Distribution Service

For Real-Time Systems

- Introduced in 2004 to address the **Data Distribution challenges** faced by a wide class of **Defense and Aerospace Applications**

- Key requirement for the standard were to deliver very **high and predictable performance** while scaling from embedded to ultra-large-scale deployments

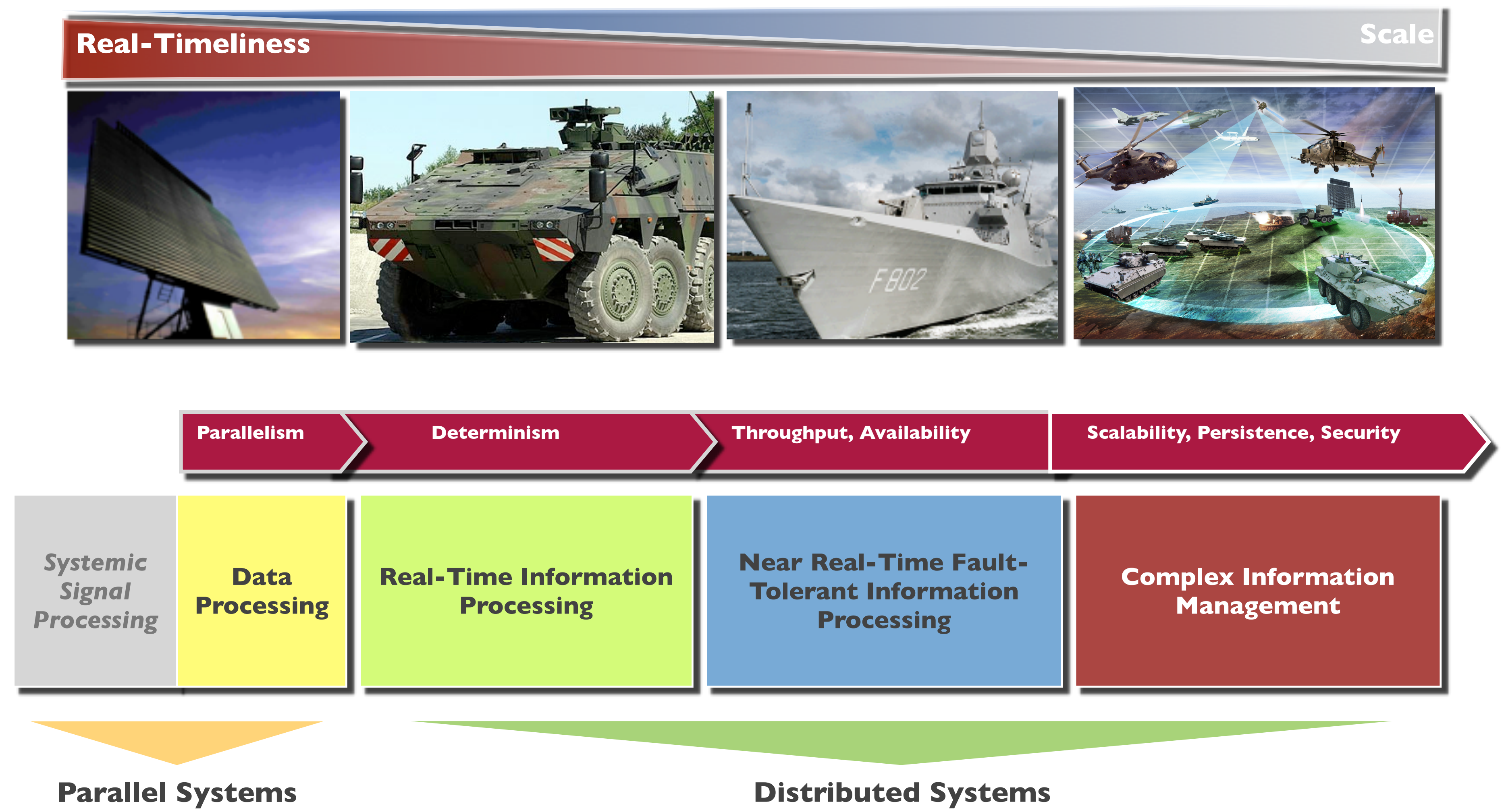


OpenSplice | DDS

Data Distribution Service

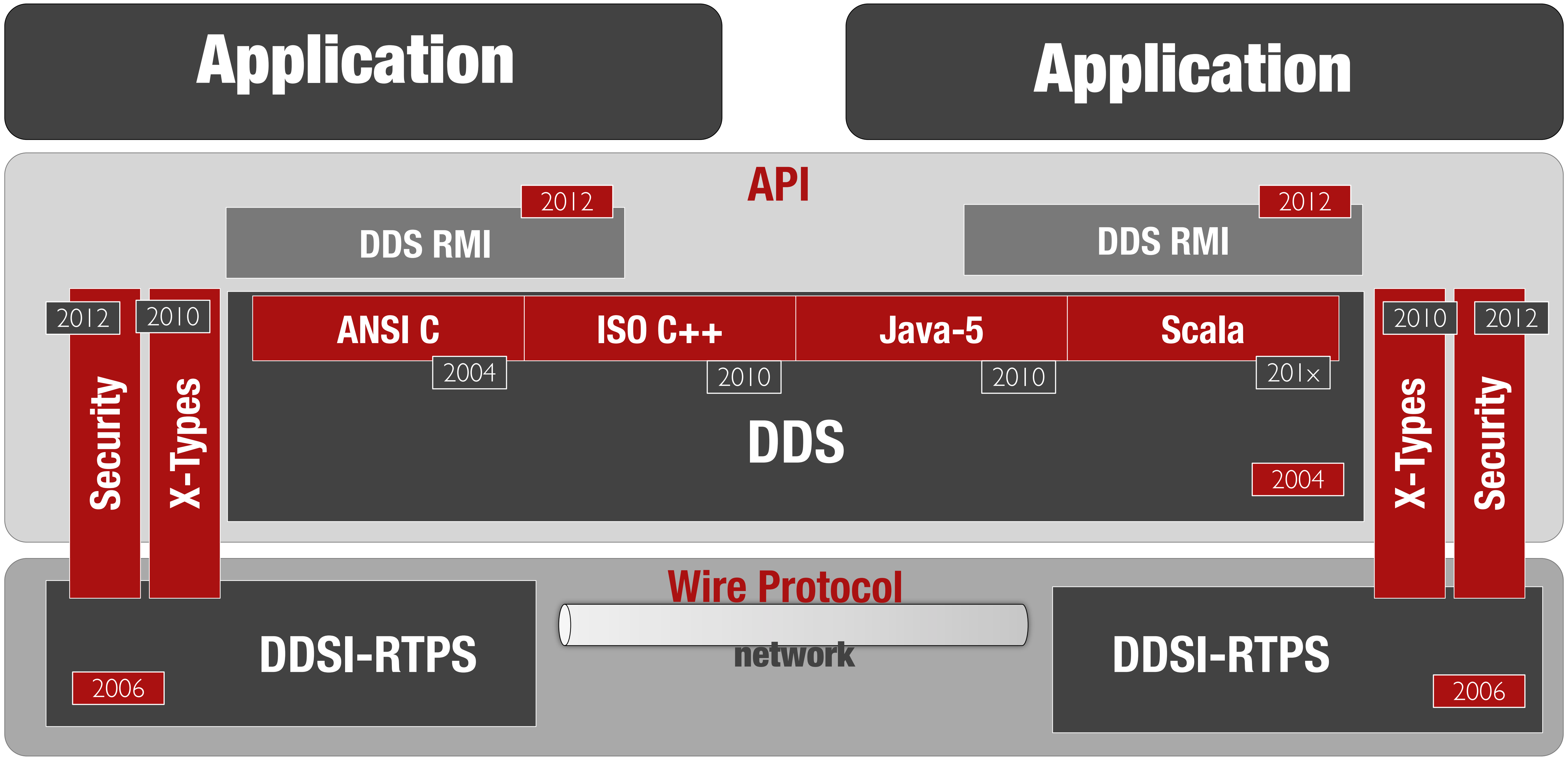
For Real-Time Systems

- **Recommended** by key administration **worldwide**, e.g. DoD, MoD, EUROCAE, etc.
- **Widely adopted** across several different domains, e.g., Automated Trading, Simulations, SCADA, Telemetry, etc.



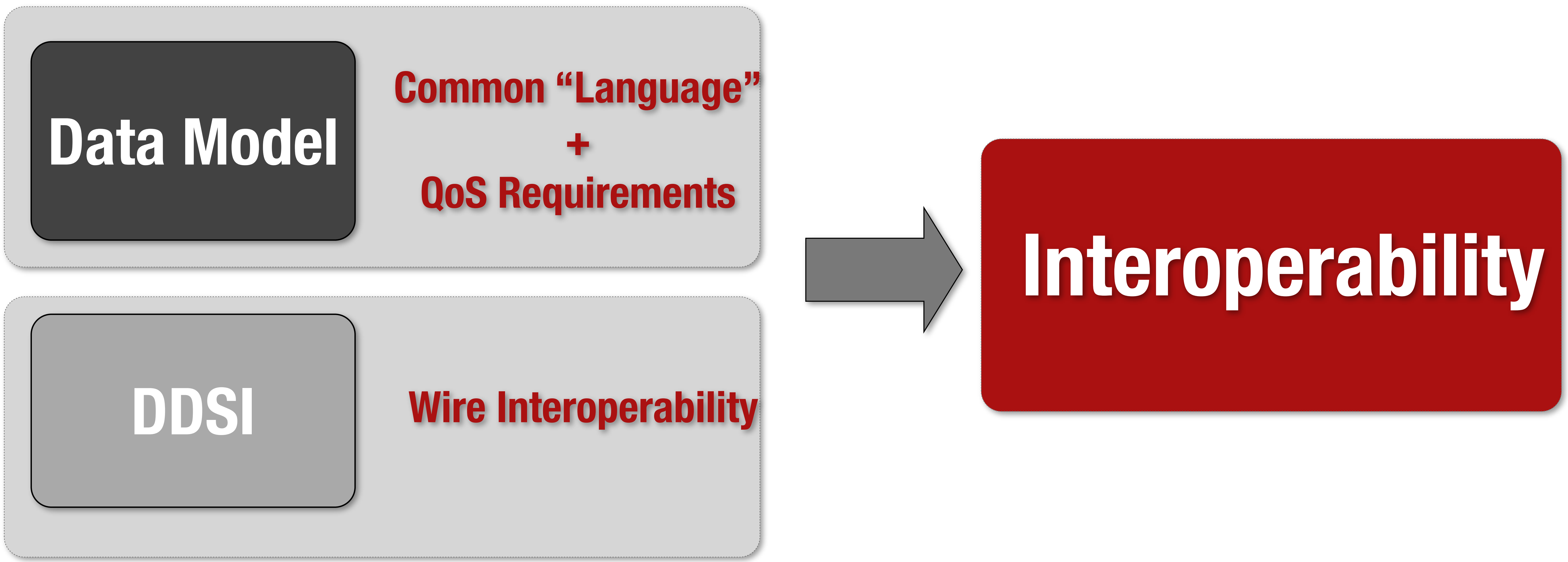
OpenSplice | DDS

DDS Standard Ecosystem

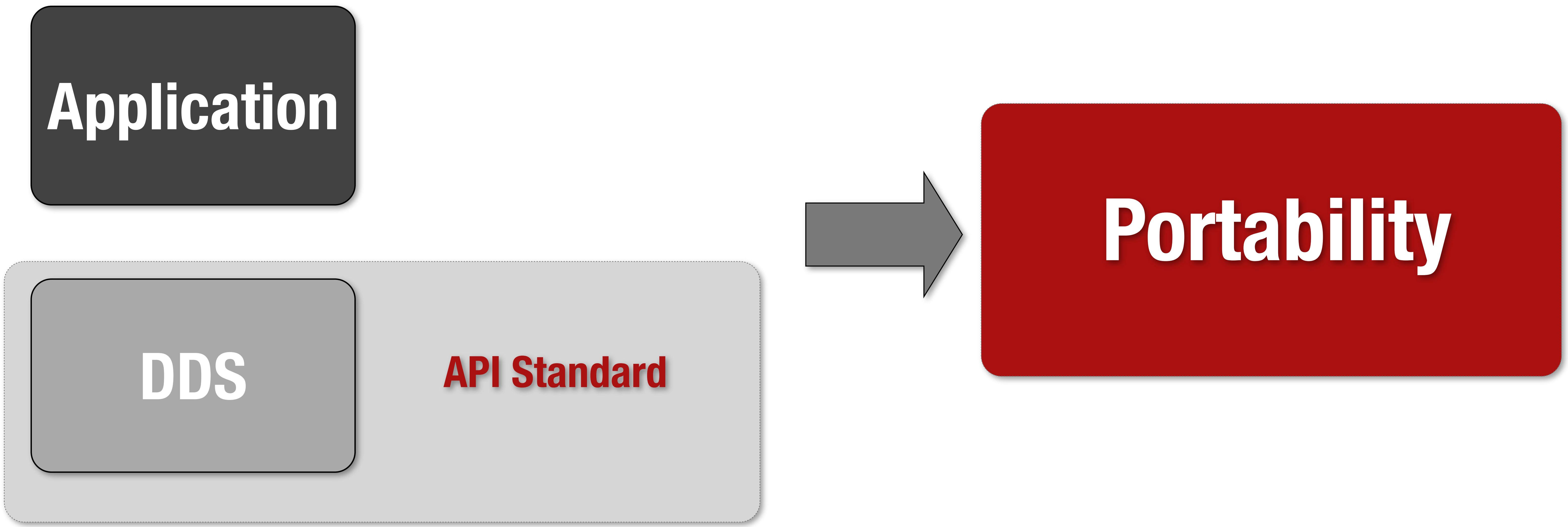


OpenSplice | DDS

Standards: What For?



Standards: What For?

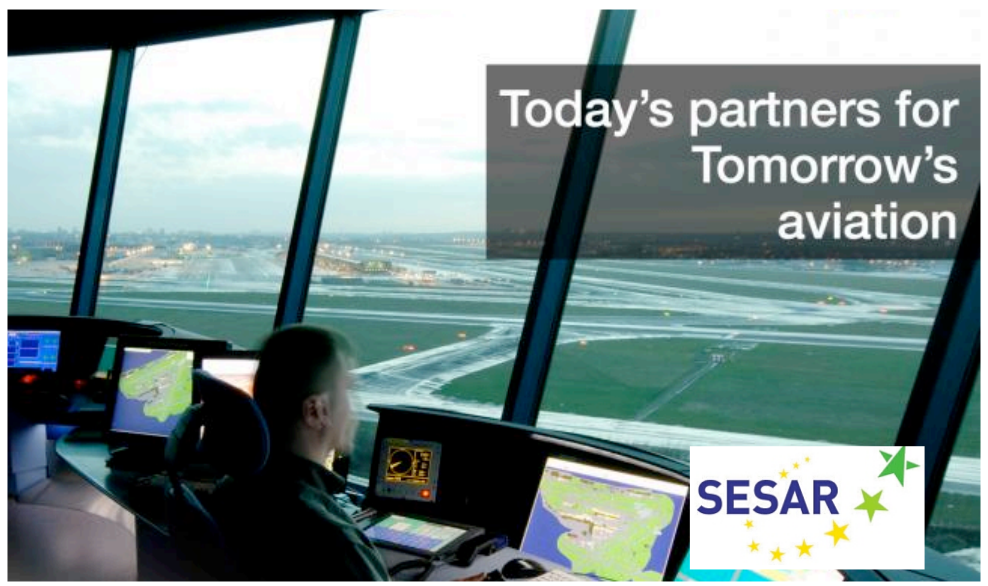


Defense and Aerospace

Integrated Modular Vetronics

Training & Simulation Systems

Naval Combat Systems



Air Traffic Control & Management

Unmanned Air Vehicles

Aerospace Applications

OpenSplice | DDS

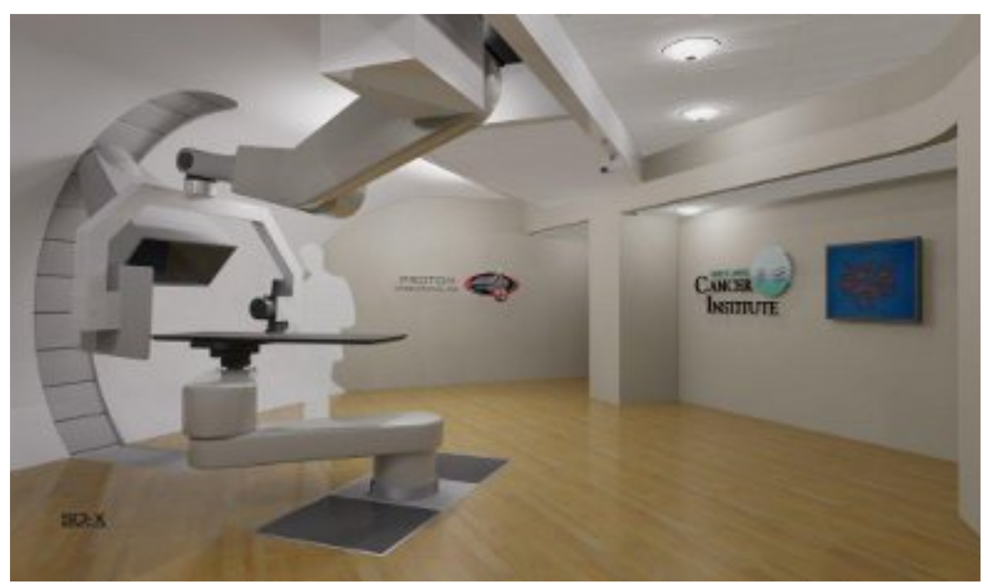
Commercial Applications

Agricultural Vehicle Systems

Large Scale SCADA Systems

Smart Cities

Train Control Systems



Complex Medical Devices

High Frequency Auto-Trading

OpenSplice | DDS

OpenSplice | DDS

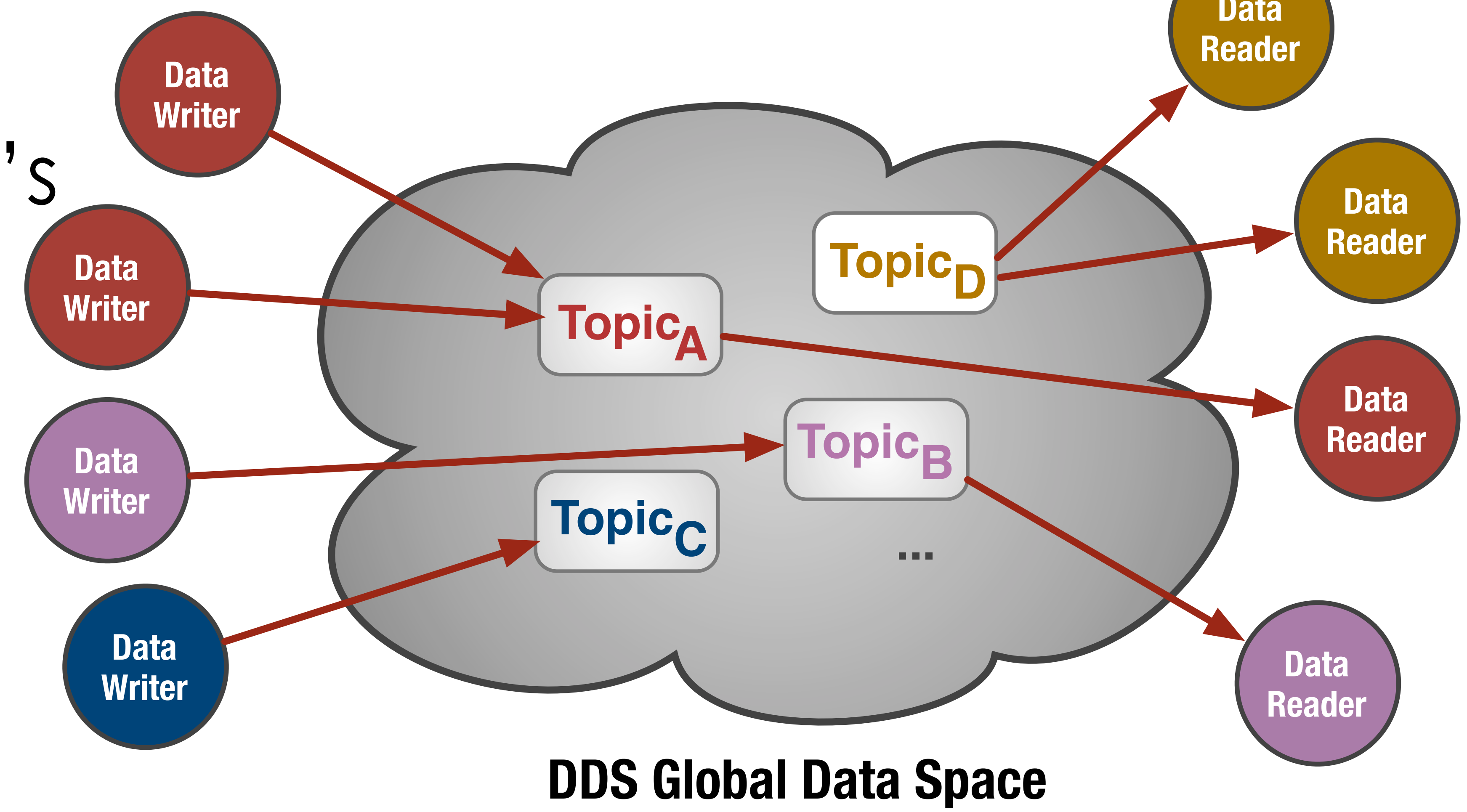
DDS
BaSics

Data Distribution Service

For Real-Time Systems

DDS provides a Topic-Based Publish/ Subscribe abstraction based on:

- **Topics**: data distribution subject's
- **DataWriters**: data producers
- **DataReaders**: data consumers

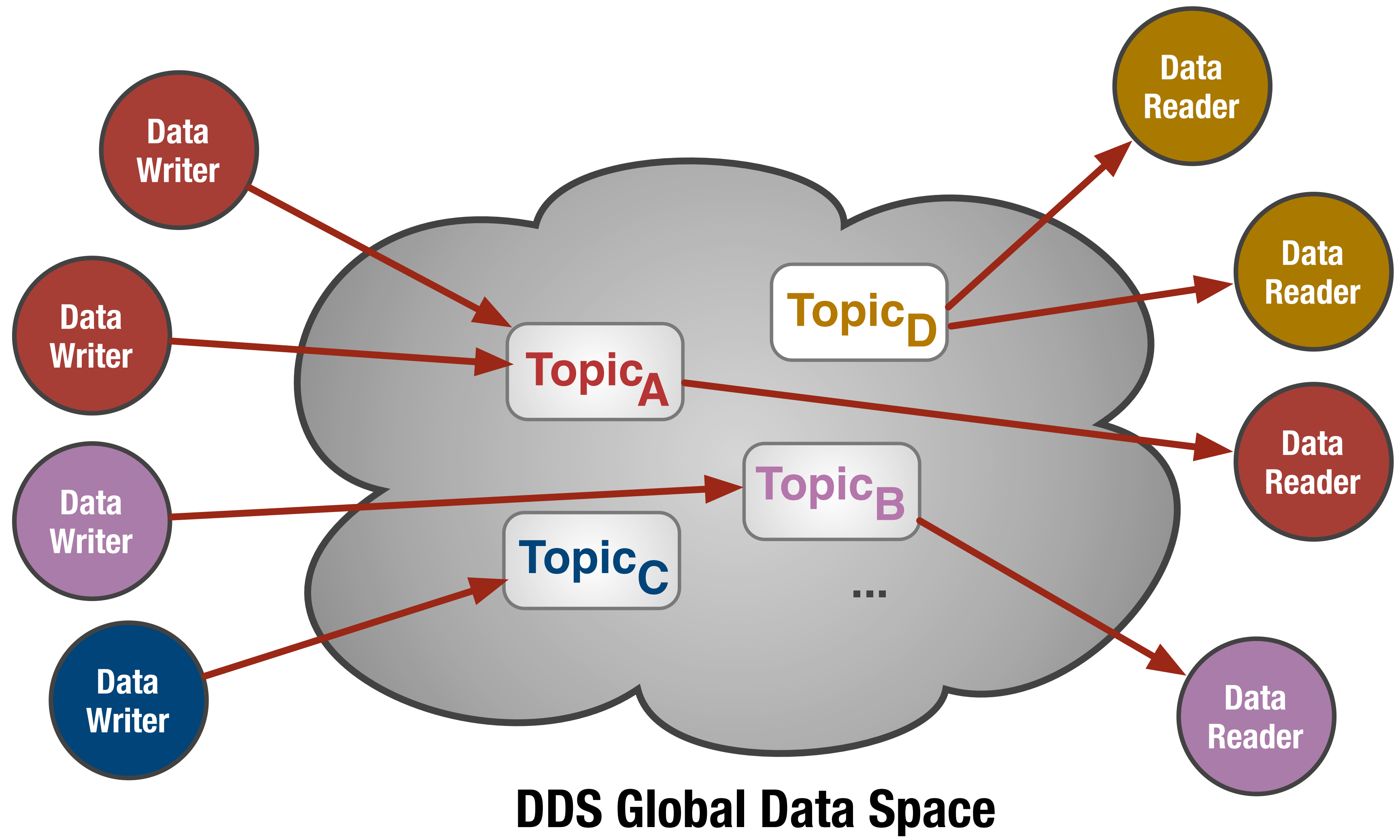


OpenSplice | DDS

Data Distribution Service

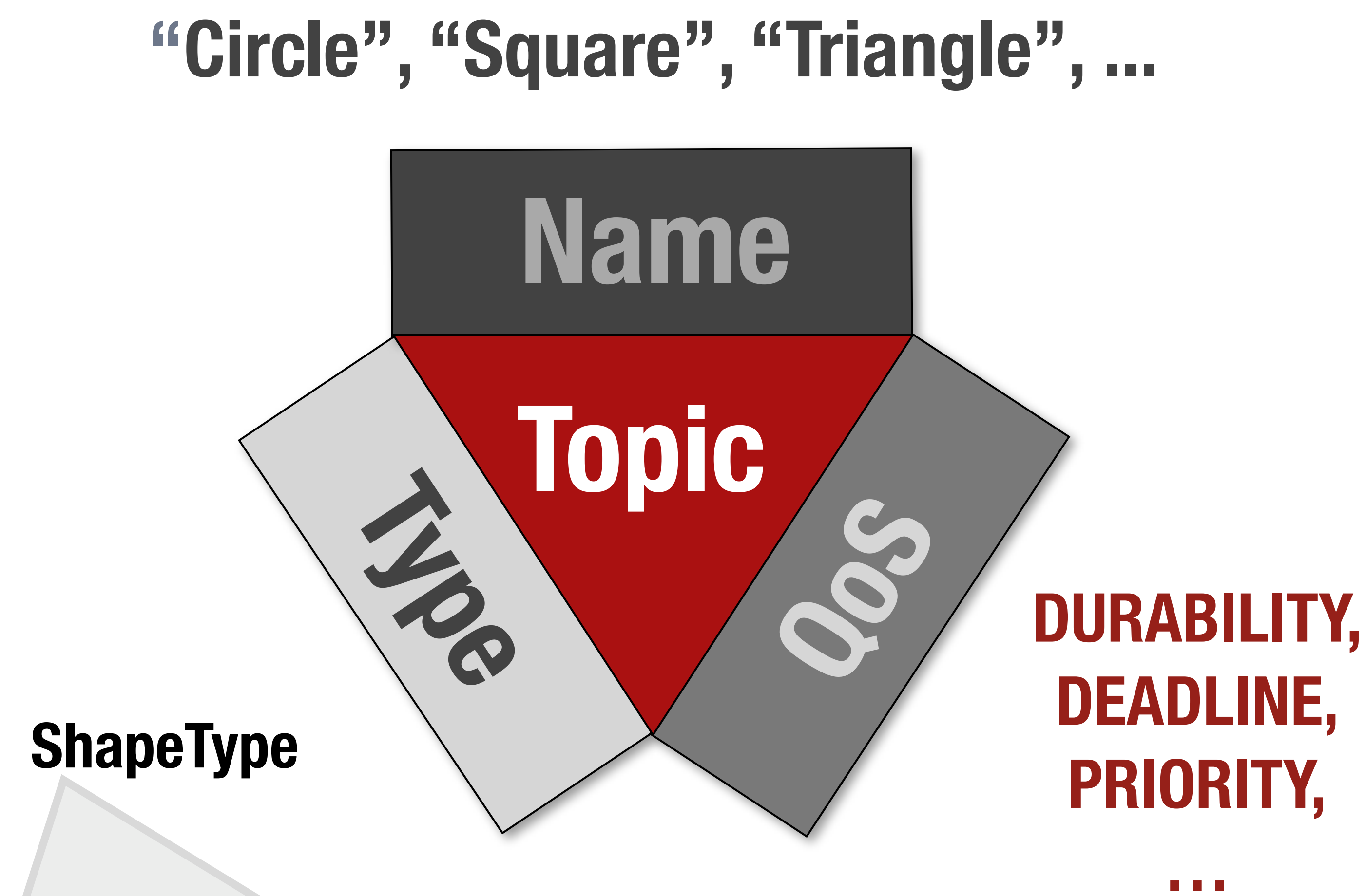
For Real-Time Systems

- DataWriters and DataReaders are automatically and dynamically matched by the DDS **Dynamic Discovery**
- A rich set of **QoS** allows to **control existential, temporal, and spatial properties of data**



DDS Topics

- A **Topic** defines a **class of streams**
- A **Topic** has associated a **unique name**, a **user defined extensible type** and a **set of QoS** policies
- QoS Policies capture the Topic non-functional invariants
- Topics can be discovered or locally defined

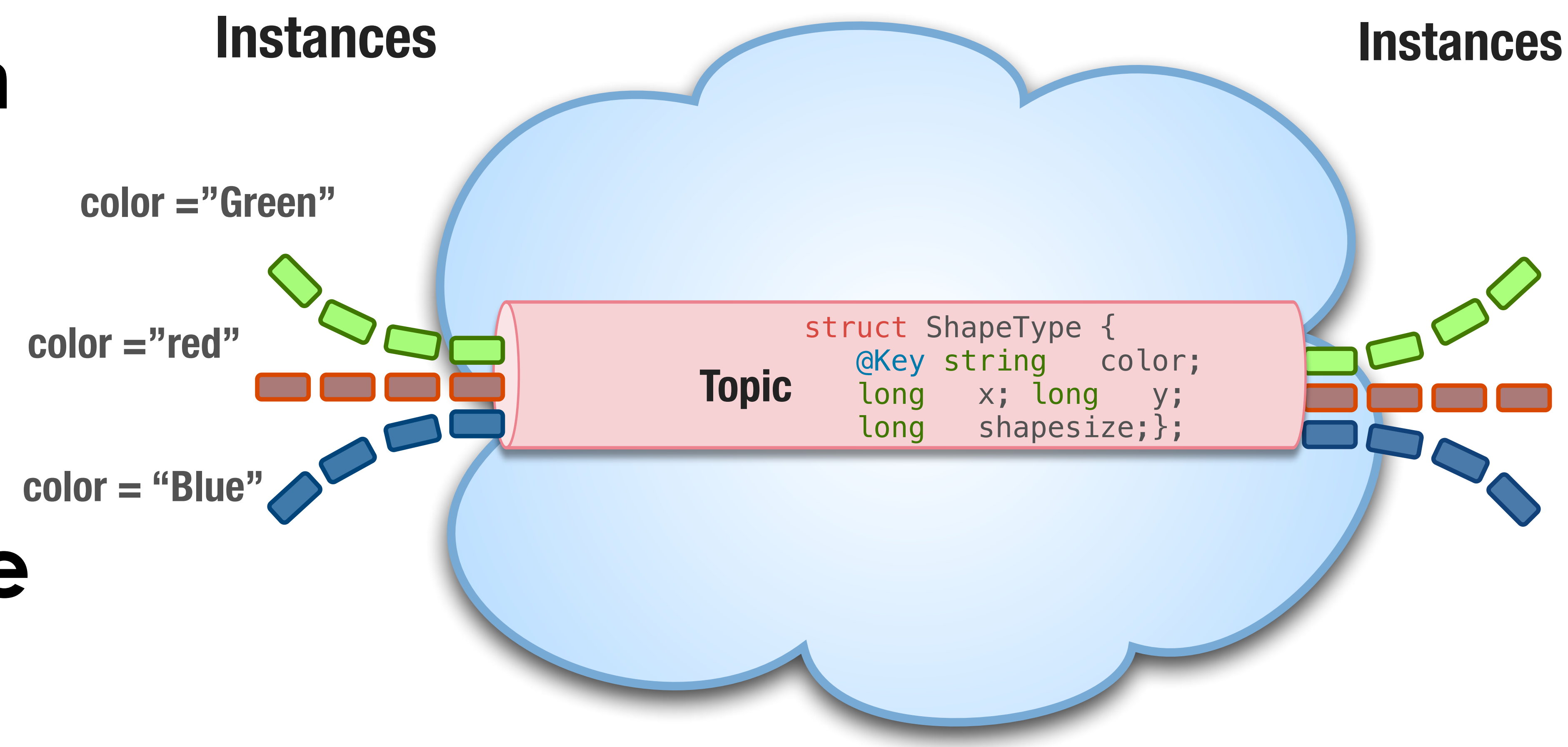



```
struct ShapeType {
   @Key
   string   color;
   long   x;
   long   y;
   long   shapesize;
};
```

OpenSplice | DDS

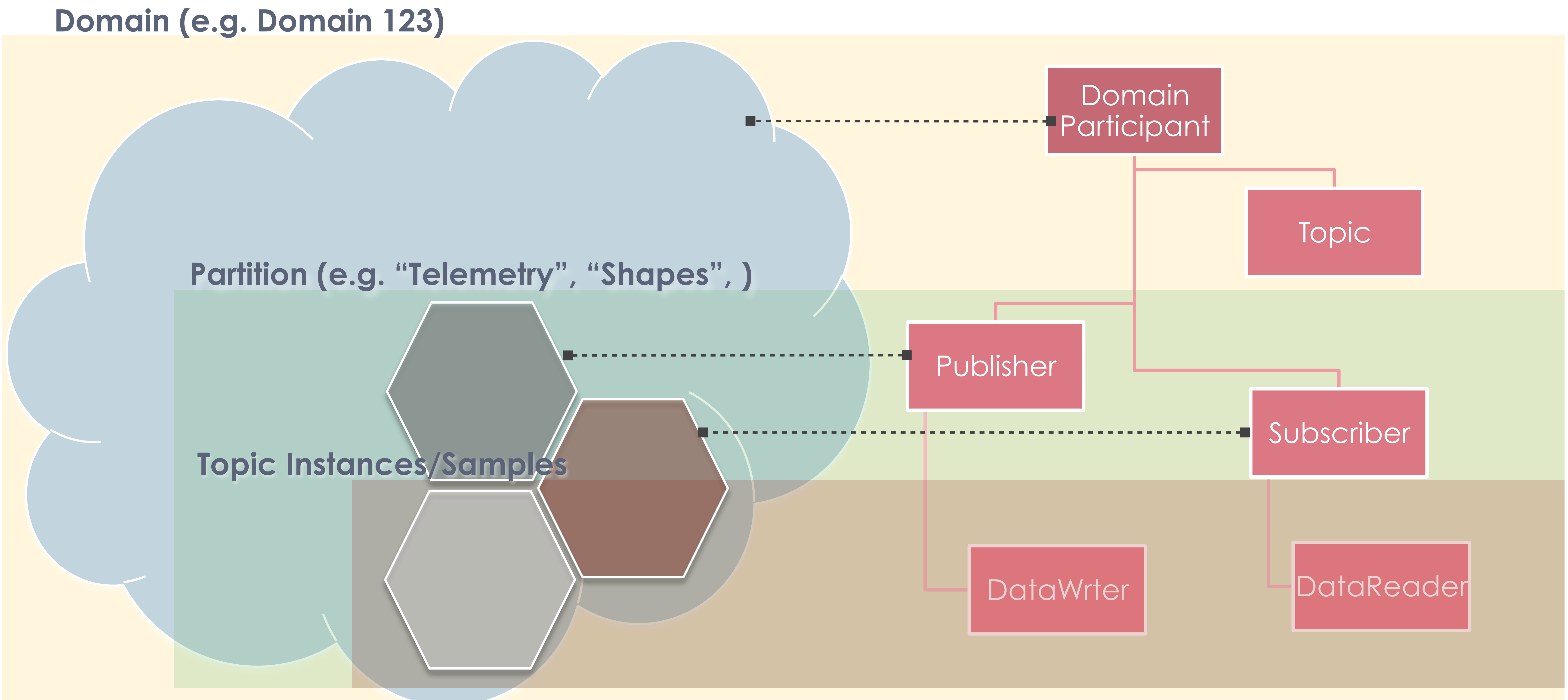
Topic Instances

- Each unique **key value** identifies a unique **stream of data**
- DDS not only demultiplexes "streams" but provides also **lifecycle information**
- A DDS DataWriter can write multiple instances



OpenSplice | DDS

Anatomy of a DDS Application

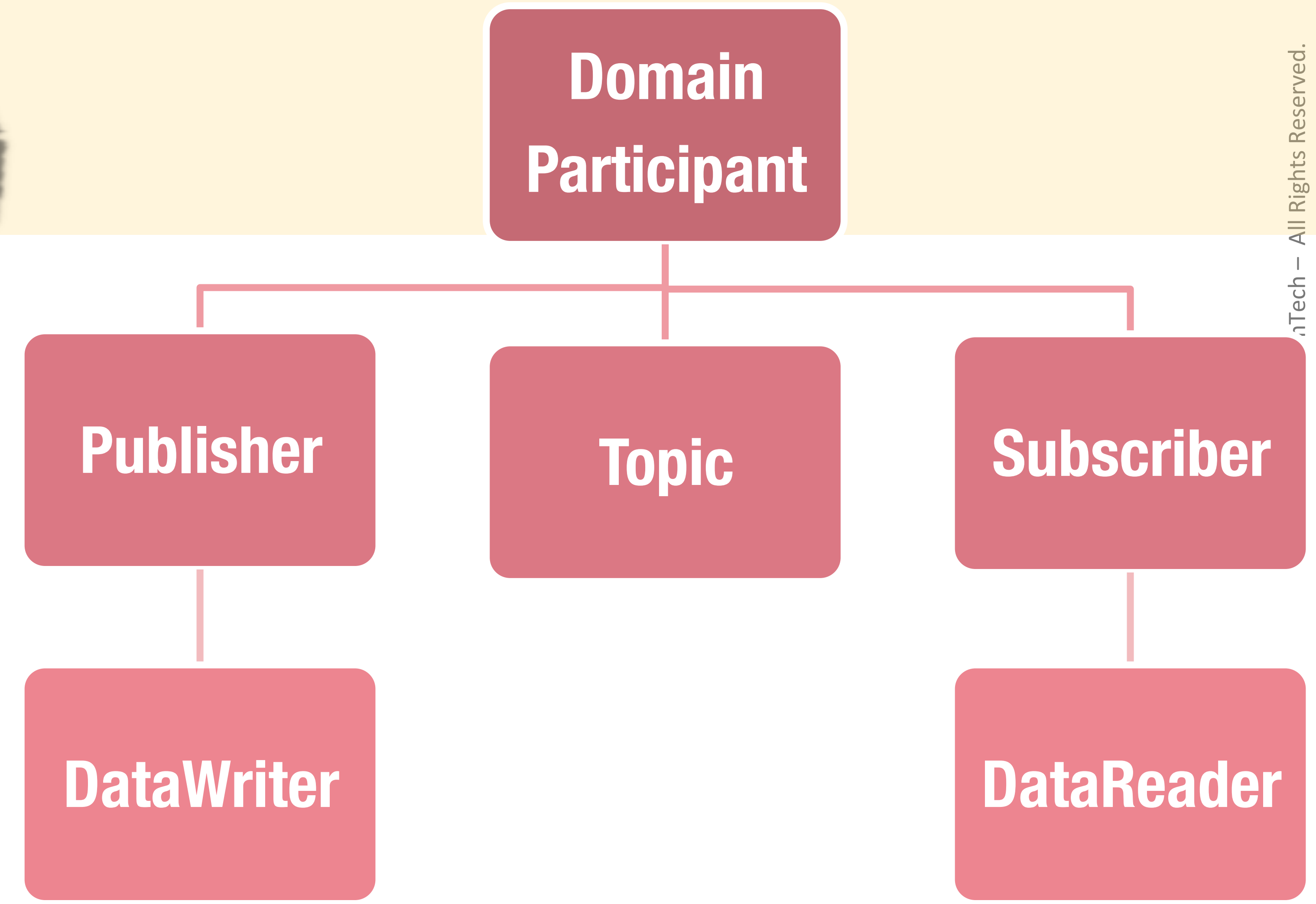


Anatomy of a DDS Application

[Scala API]

Domain

```
val dp = DomainParticipant(domainId)
```



OpenSplice | DDS

Anatomy of a DDS Application

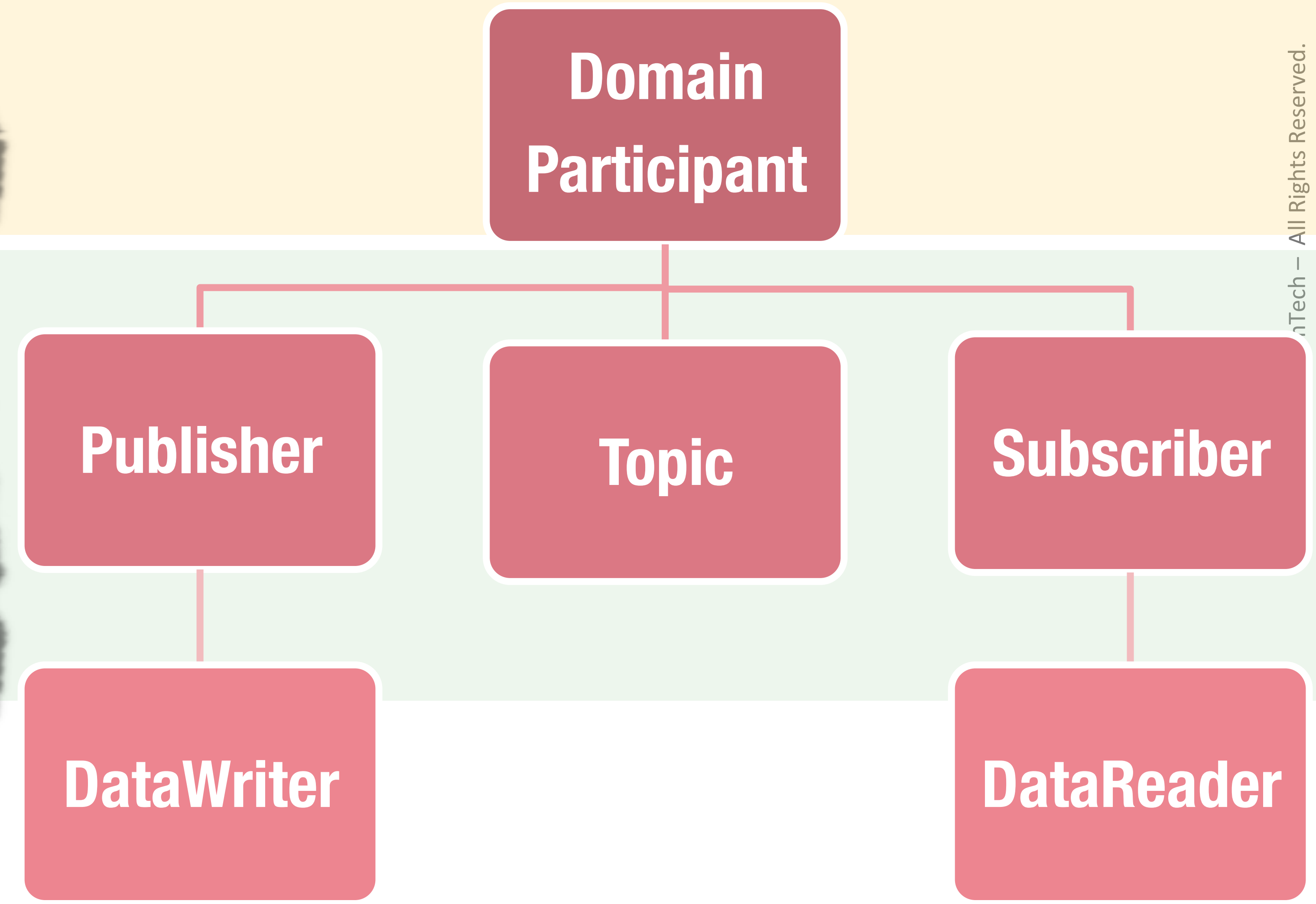
[Scala API]

Domain

```scala
val dp = DomainParticipant(domainId)
```

Session

```scala
// Create a Topic
val topic = Topic[ShapeType](dp, “Circle”)
// Create a Publisher / Subscriber
val pub = Publisher(dp)
val sub = Subscriber(dp)
```



OpenSplice | DDS

Anatomy of a DDS Application

[Scala API]

Domain

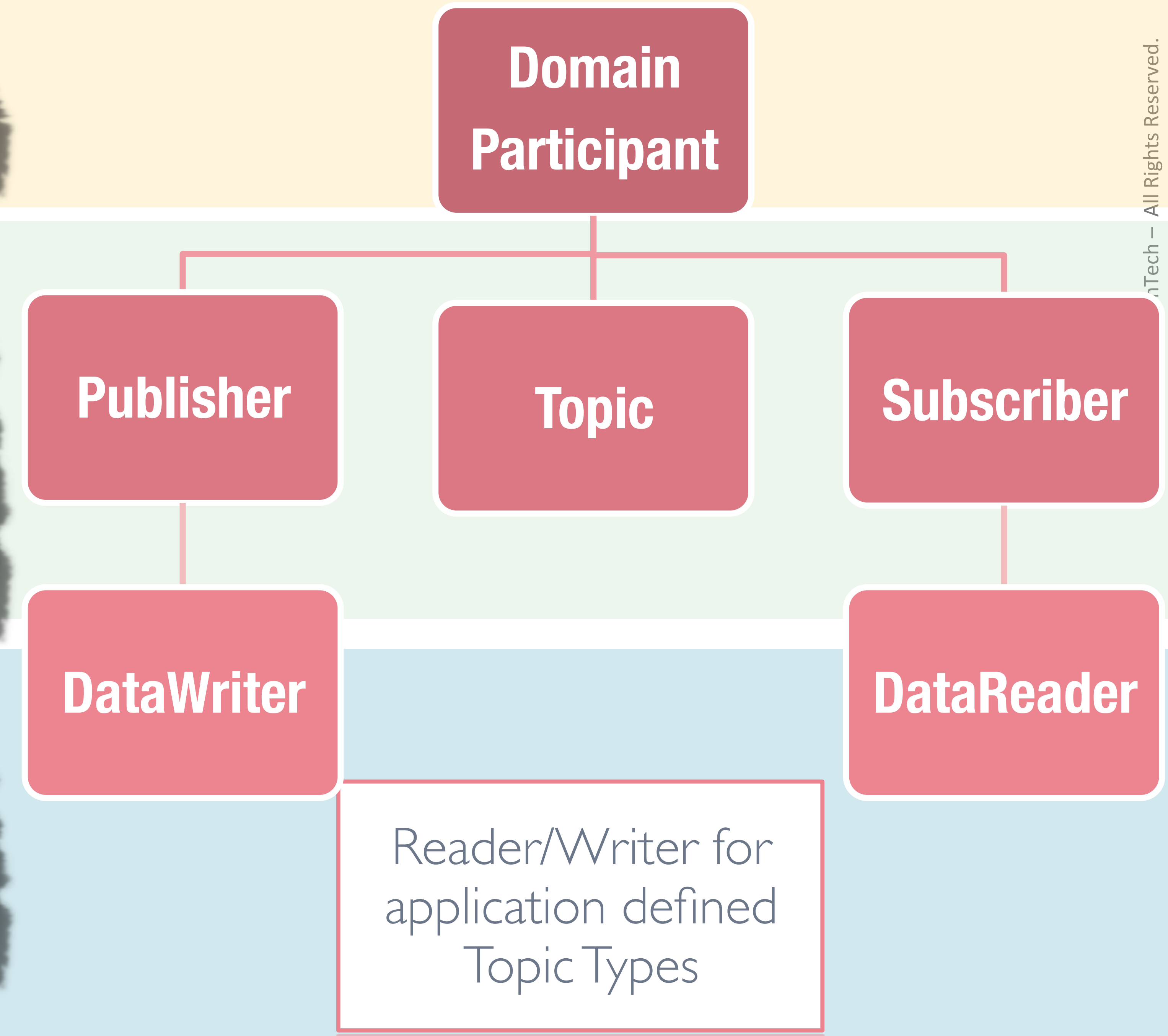
```
val dp = DomainParticipant(domainId)
```

Session

```
// Create a Topic
val topic = Topic[ShapeType](dp, “Circle”)
// Create a Publisher / Subscriber
val pub = Publisher(dp)
val sub = Subscriber(dp)
```

Reader/Writers for User Defined for Types

```
// Create a DataWriter/DataWriter
val writer = DataWriter[ShapeType](pub, topic)
val reader = DataReader[ShapeType](sub, topic)
```



OpenSplice | DDS

Anatomy of a DDS Application

[Scala API]

Domain

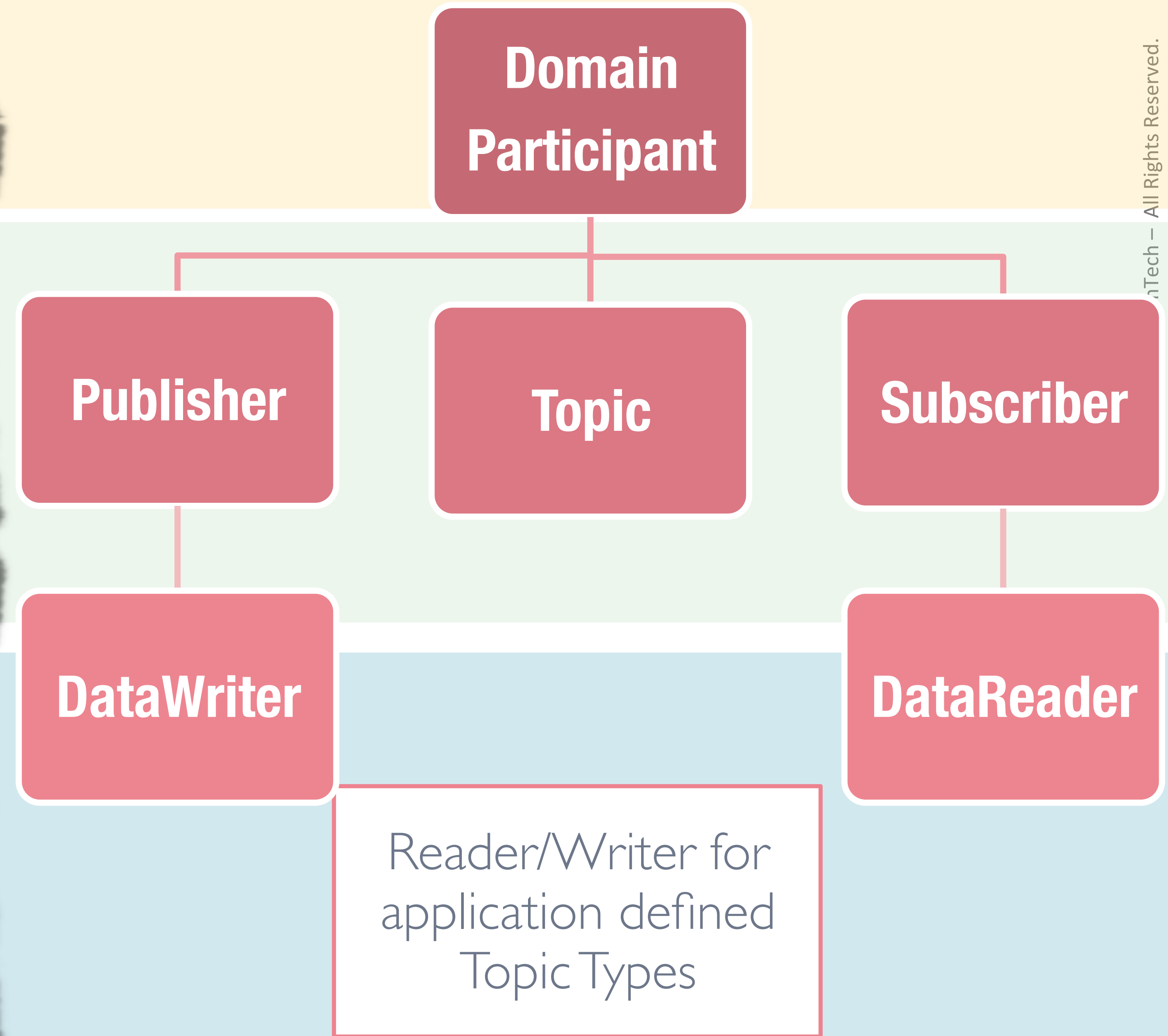
```scala
val dp = DomainParticipant(domainId)
```

Session

```scala
// Create a Topic
val topic = Topic[ShapeType](dp, “Circle”)
// Create a Publisher / Subscriber
val pub = Publisher(dp)
val sub = Subscriber(dp)
```

Reader/Writers for User Defined for Types

```scala
// Write data
val data = new ShapeType(“RED”, 131, 107, 75)
writer write data
// But you can also write like this...
writer ! data

// Read new data and print it on the screen
(reader read) foreach (prinln)
```



OpenSplice | DDS

Anatomy of a DDS Application

[DDS C++ API 2010]

Domain

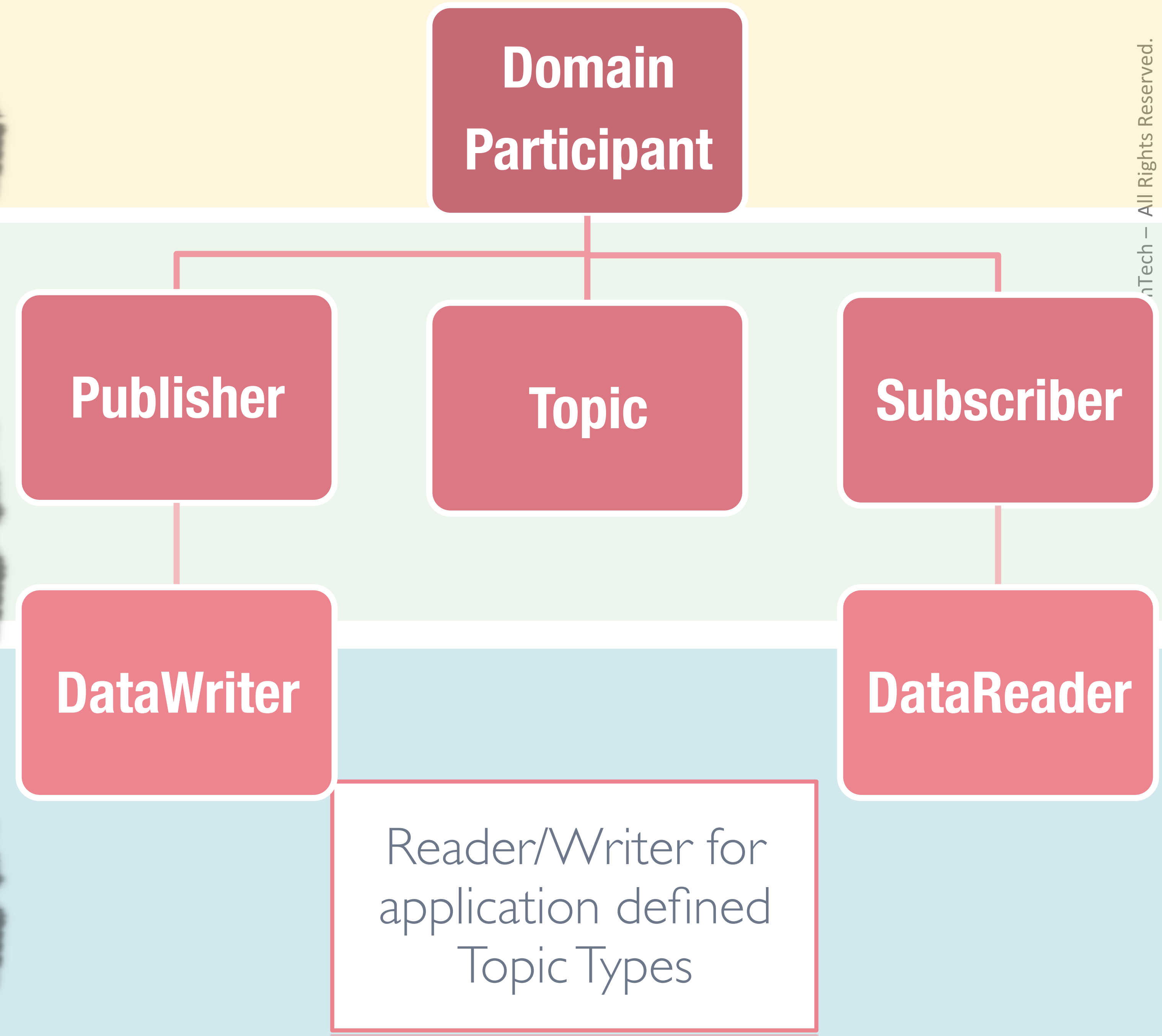
```
auto dp = DomainParticipant(domainId);
```

Session

```
// Create a Topic
auto topic = Topic<ShapeType>(dp, "Circle")
// Create a Publisher / Subscriber
auto pub = Publisher(dp)
auto sub = Subscriber(dp)
```

Reader/Writers for User Defined for Types

```
// Create a DataWriter/DataWriter
auto writer = DataWriter<ShapeType>(pub, topic);
auto reader = DataReader<ShapeType>(sub, topic);
```



Anatomy of a DDS Application

[DDS C++ API 2010]

Domain

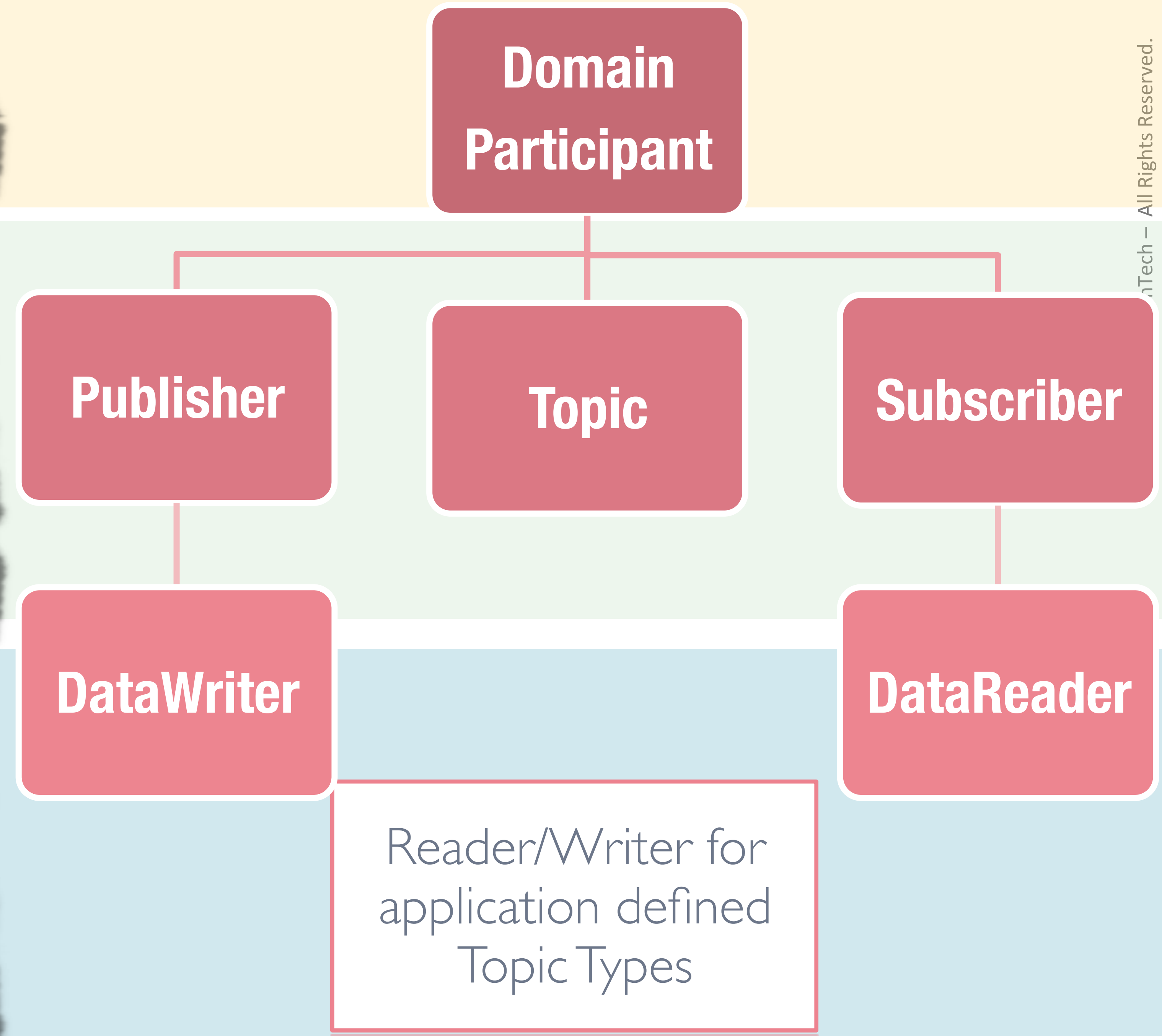
```
auto dp = DomainParticipant(domainId);
```

Session

```
// Create a Topic
auto topic = Topic<ShapeType>(dp, “Circle”)
// Create a Publisher / Subscriber
auto pub = Publisher(dp)
auto sub = Subscriber(dp)
```

Reader/Writers for User Defined for Types

```
// Write data
writer.write(ShapeType(“RED”, 131, 107, 89));
// But you can also write like this...
writer << ShapeType(“RED”, 131, 107, 89);

// Read new data (loaned)
auto data = reader.read();
```



OpenSplice | DDS

Anatomy of a DDS Application

[DDS Java 5 API]

Domain

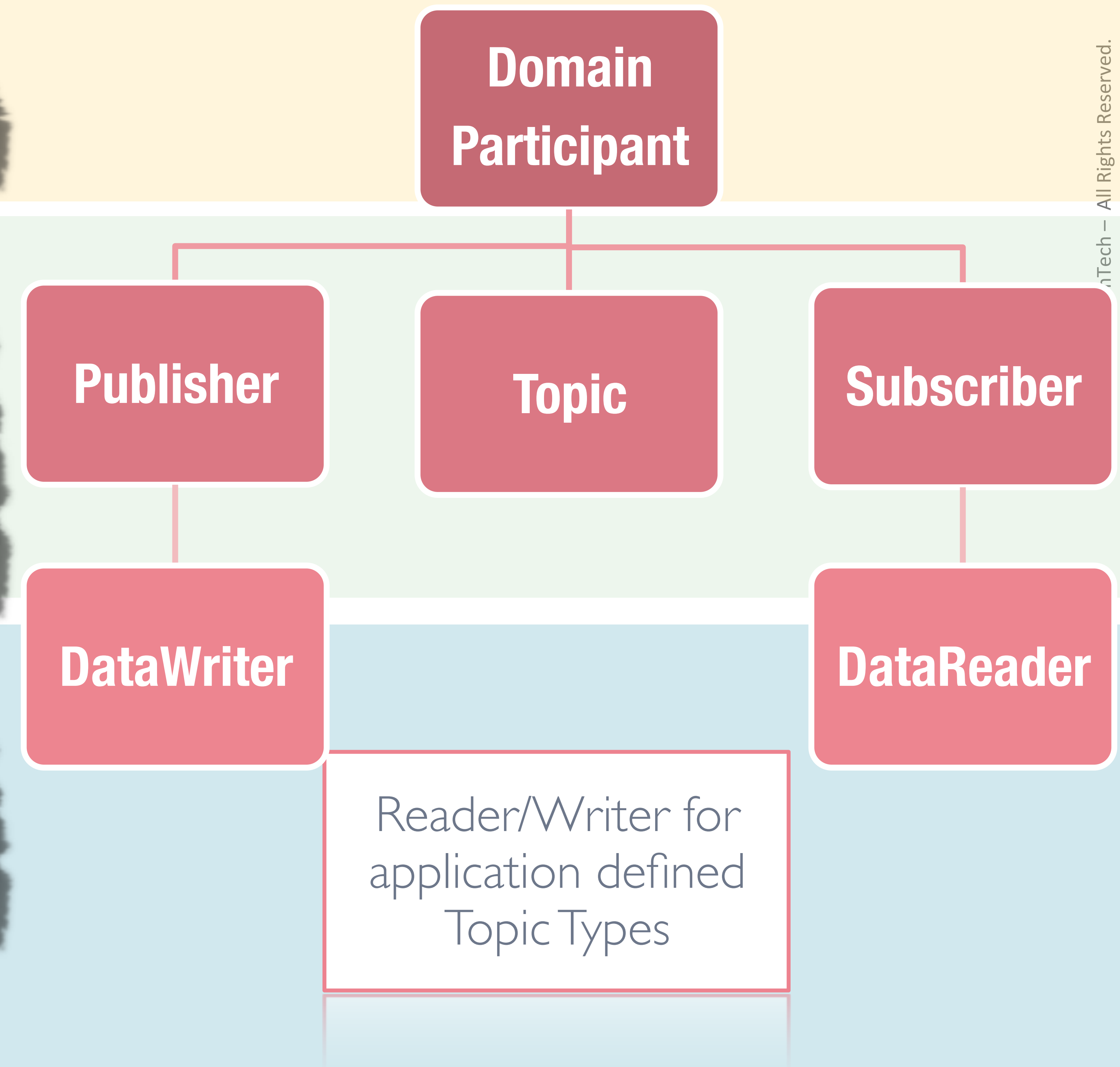
```
auto dp = DomainParticipant(domainId);
```

Session

```
// Create a Topic
val topic = Topic<ShapeType>(dp, "Circle")
// Create a Publisher / Subscriber
val pub = Publisher(dp)
val sub = Subscriber(dp)
```

Reader/Writers for User Defined for Types

```
// Create a DataWriter/DataWriter
auto writer = DataWriter<ShapeType>(pub, topic);
auto reader = DataReader<ShapeType>(sub, topic);
```



OpenSplice | DDS

Anatomy of a DDS Application

[DDS Java 5 API]

Domain

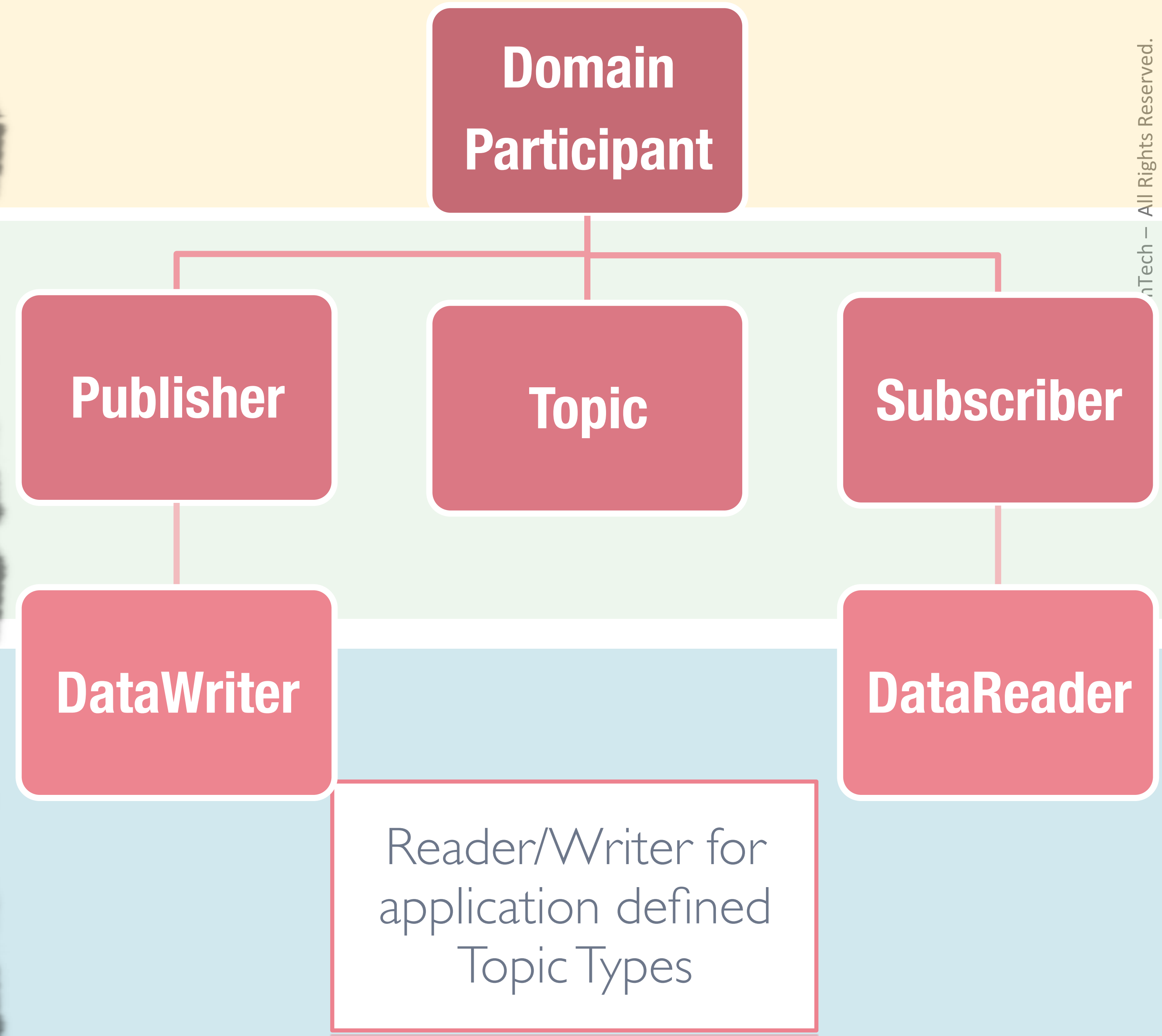
```
Do dp = DomainParticipant(domainId);
```

Session

```
// Create a Topic
val topic = Topic<ShapeType>(dp, “Circle”)
// Create a Publisher / Subscriber
val pub = Publisher(dp)
val sub = Subscriber(dp)
```

Reader/Writers for User Defined for Types

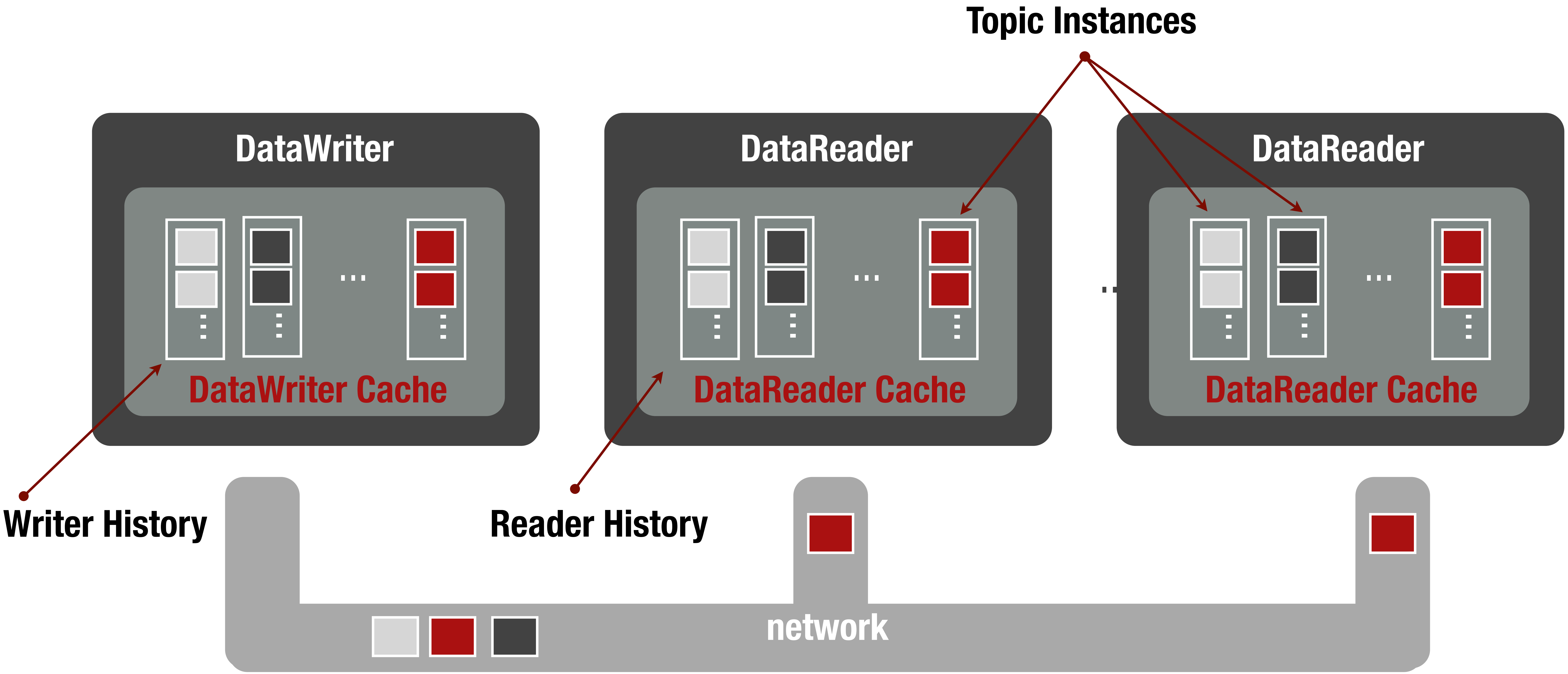
```
// Write data
writer.write(ShapeType(“RED”, 131, 107, 89));
// But you can also write like this...
writer << ShapeType(“RED”, 131, 107, 89);

// Read new data (loaned)
auto data = reader.read();
```



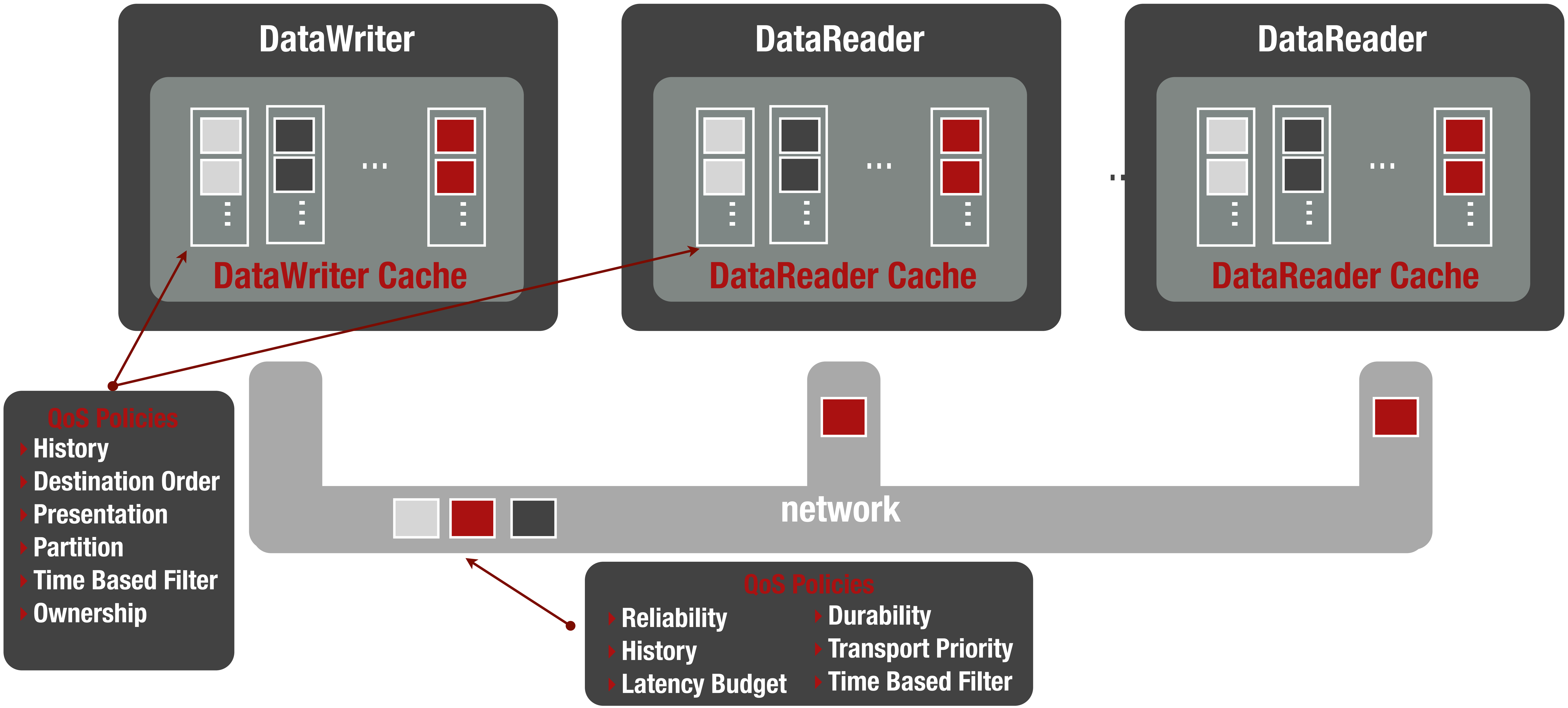
OpenSplice | DDS

Data Reader/Writer Caches

DDS Model

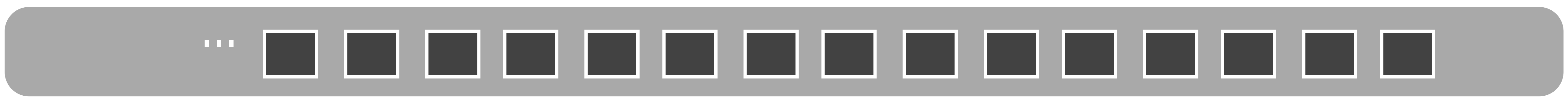

Topic Instances
DataWriter
DataWriter Cache
DataReader
DataReader Cache
DataReader
DataReader Cache
Writer History
Reader History
network

DDS Model



DataWriter
DataWriter Cache
DataReader
DataReader Cache
DataReader
DataReader Cache
network
QoS Policies
History
Destination Order
Presentation
Partition
Time Based Filter
Ownership
QoS Policies
Reliability
History
Latency Budget
Durability
Transport Priority
Time Based Filter

Dynamic View of a Stream



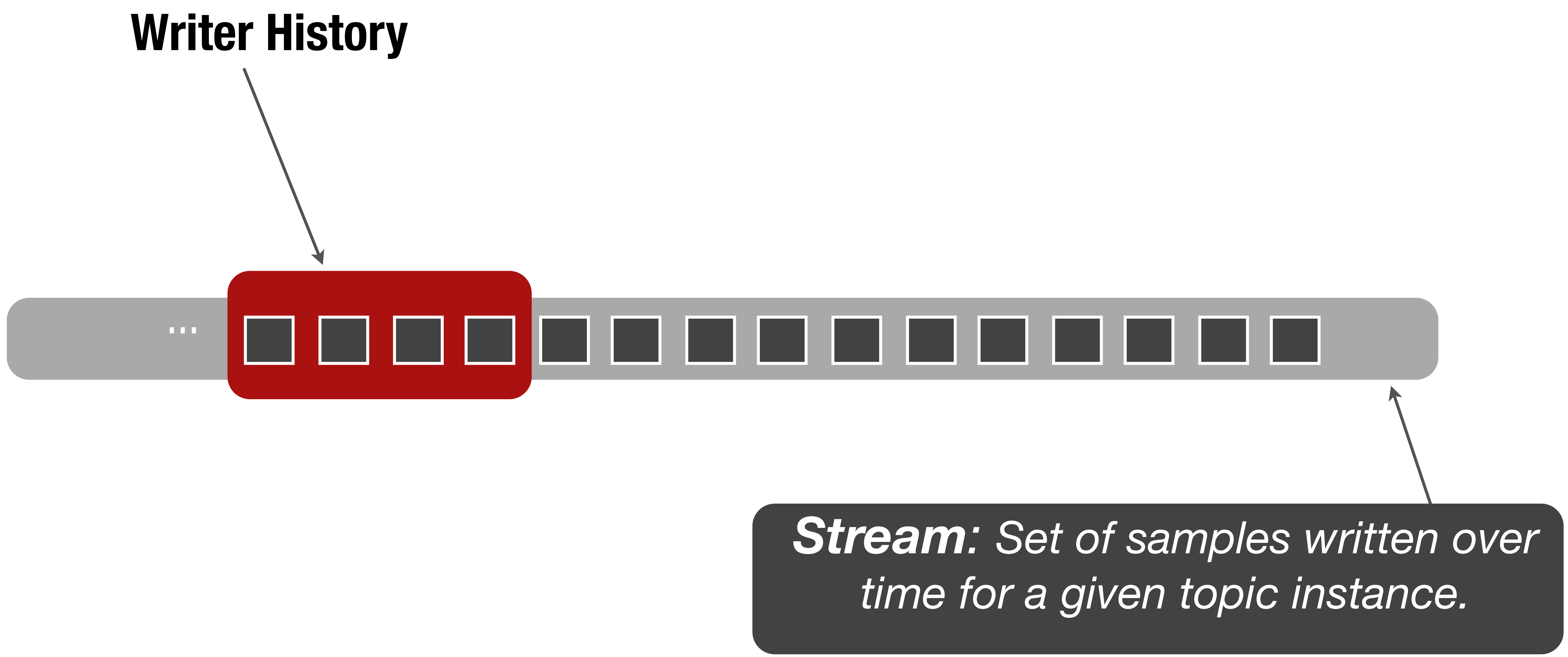
Stream: *Set of samples written over time for a given topic instance.*

OpenSplice | DDS

Dynamic View of a Stream

Assumptions:

Reader History = KeepLast (n)
WriterHistory = KeepLast (m)

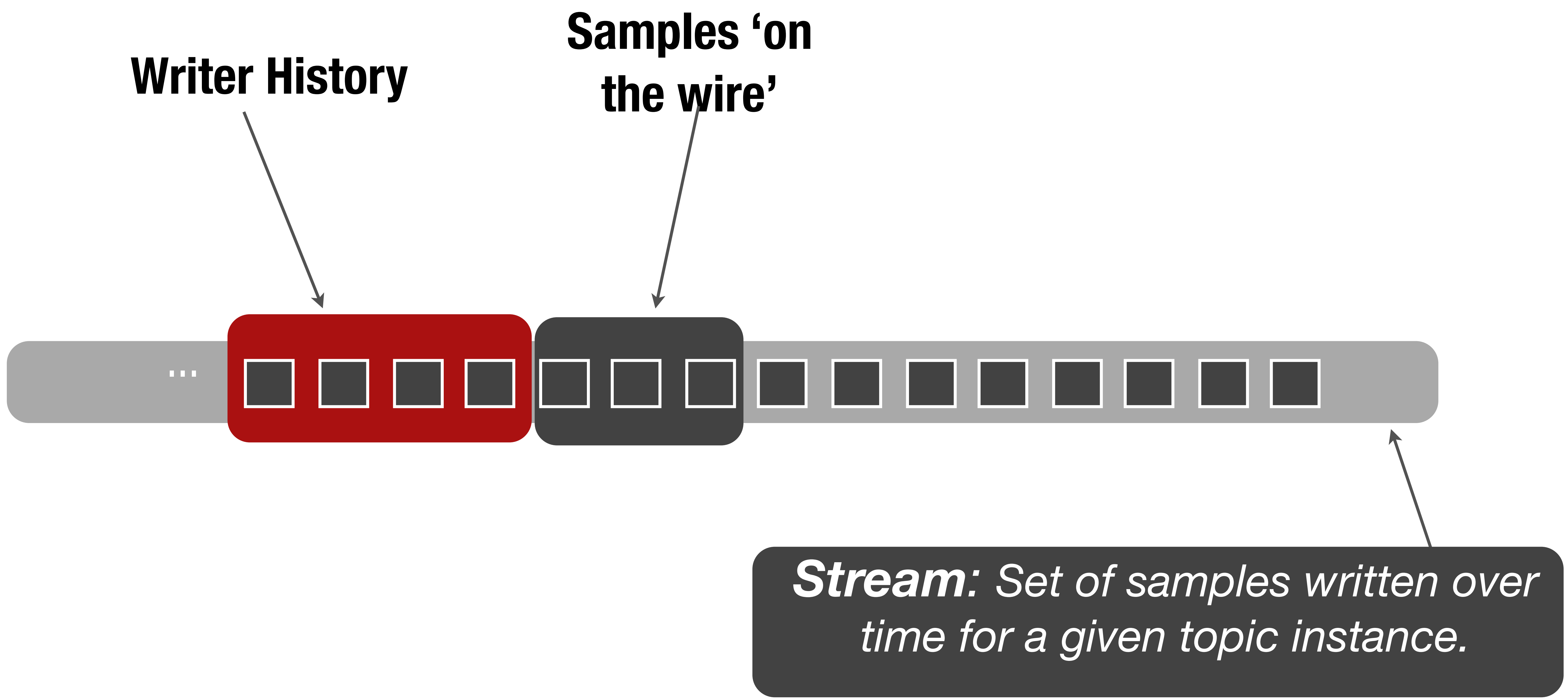


OpenSplice | DDS

Dynamic View of a Stream

Assumptions:

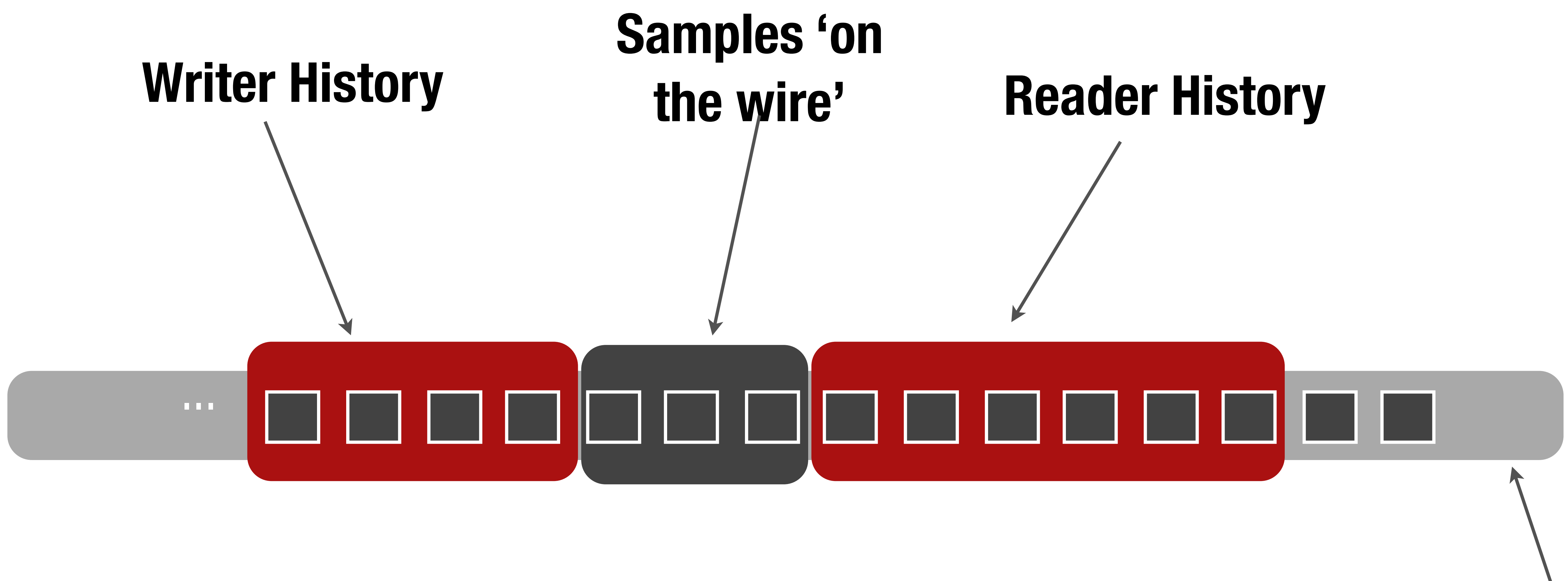
Reader History = KeepLast (n)
WriterHistory = KeepLast (m)



Dynamic View of a Stream

Assumptions:

Reader History = KeepLast (n)
WriterHistory = KeepLast (m)

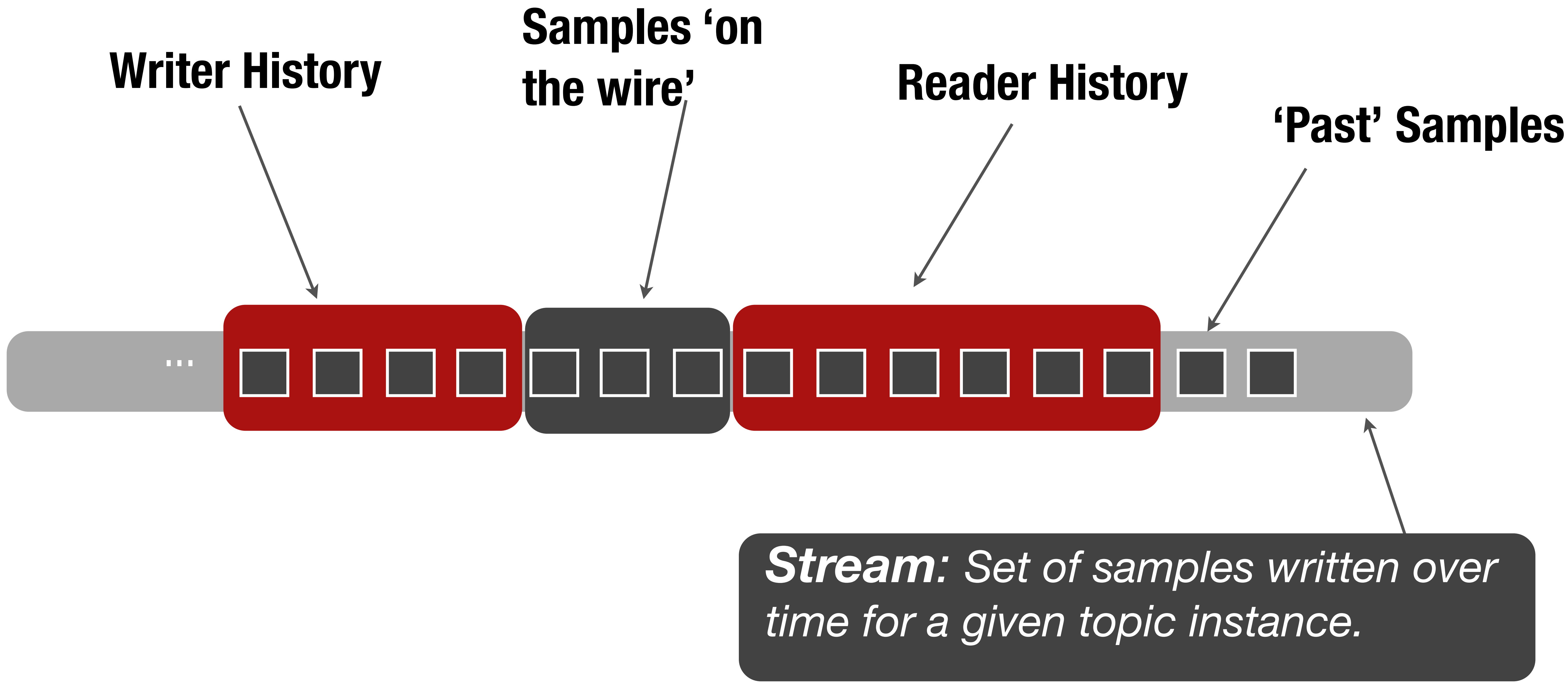


***Stream**: Set of samples written over time for a given topic instance.*

Dynamic View of a Stream

Assumptions:

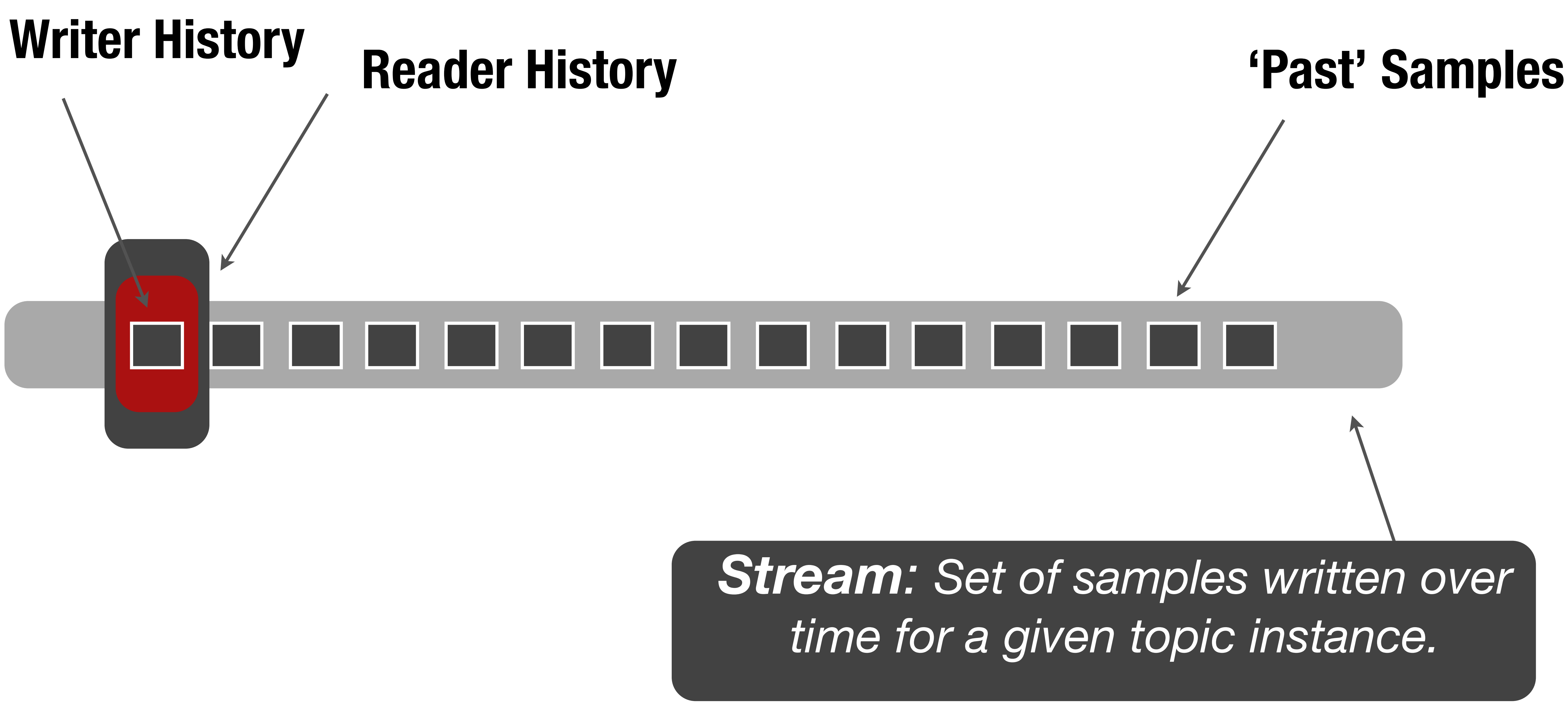
Reader History = KeepLast (n)
WriterHistory = KeepLast (m)



Eventual View of a Stream

Assumptions (Default Settings):

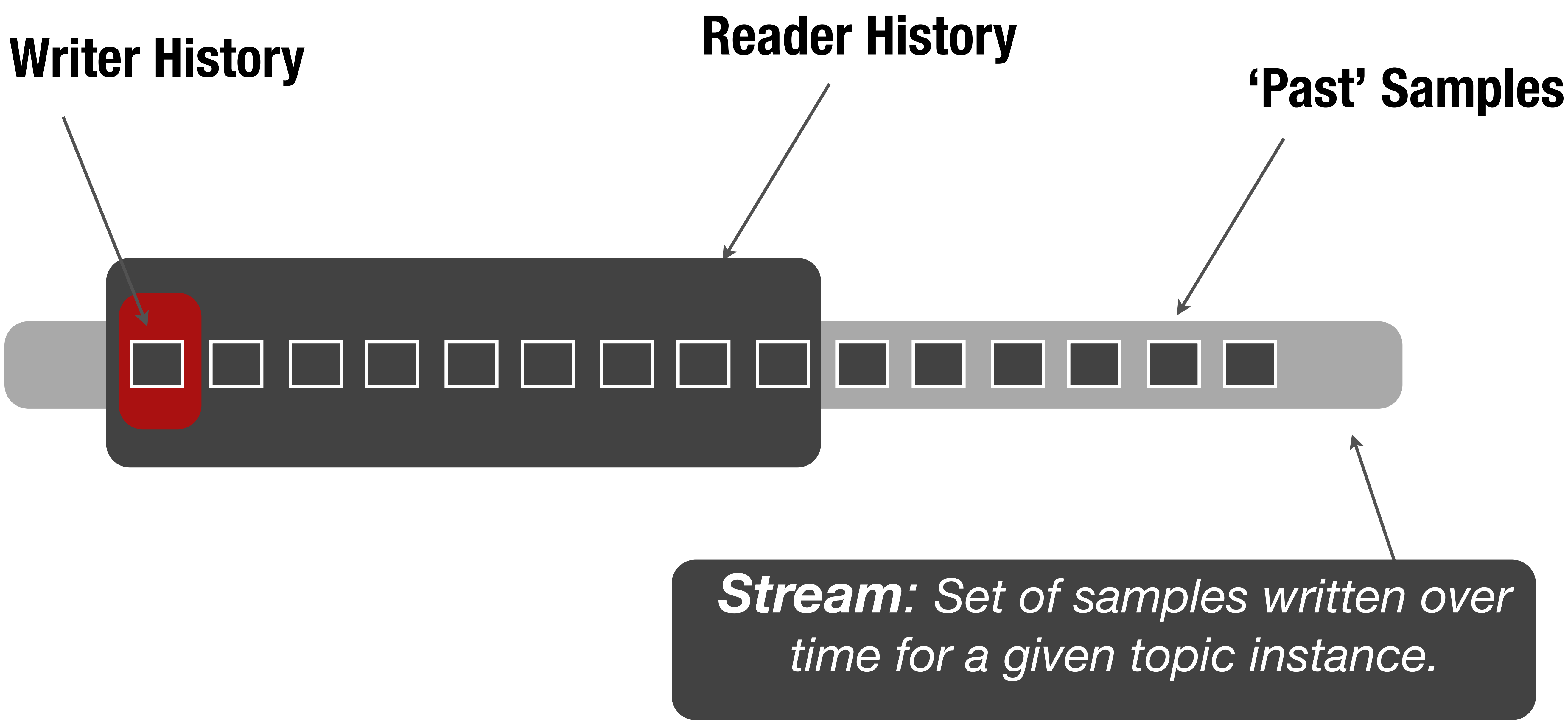
Reader History = KeepLast (1)
WriterHistory = KeepLast (1)



Eventual View of a Stream

Assumptions:

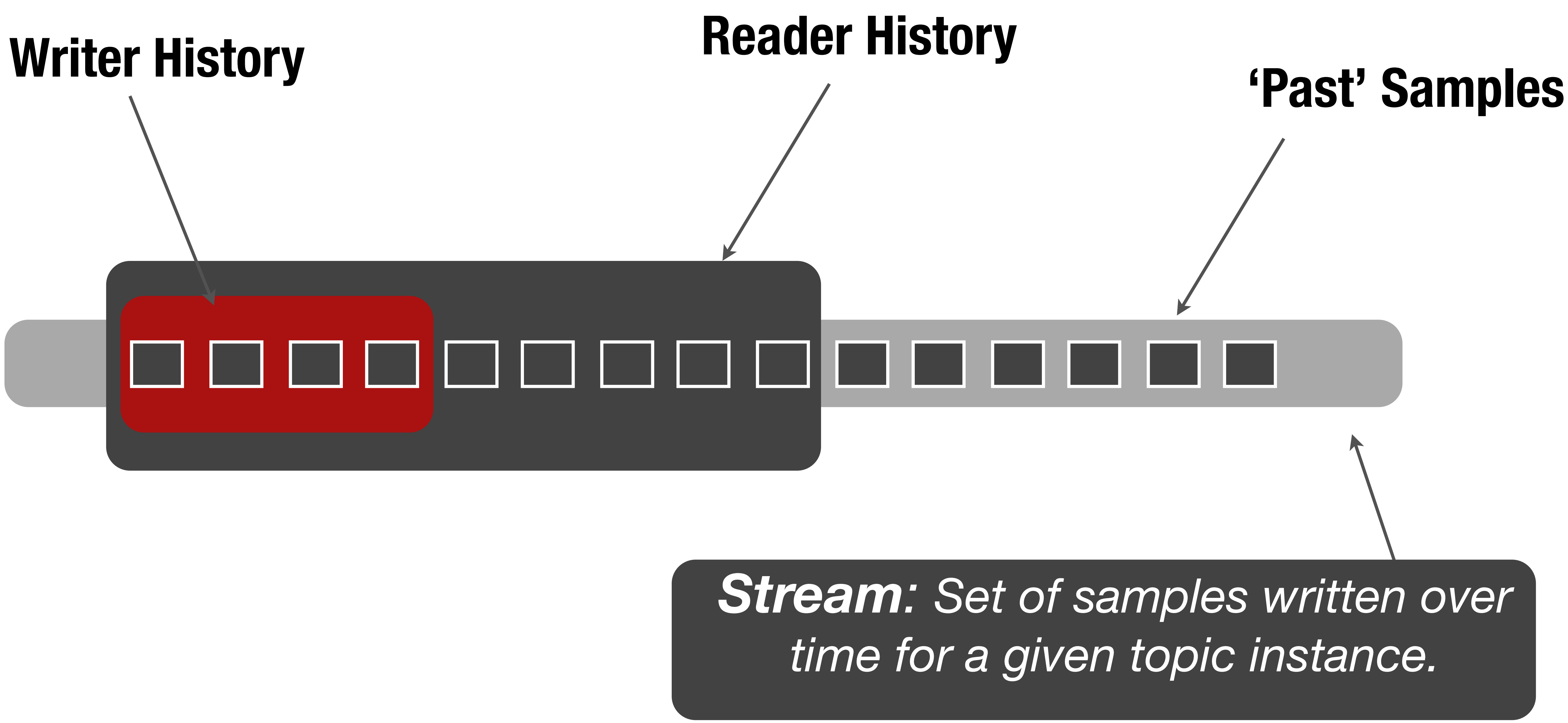
Reader History = KeepLast (n) with n > 1
WriterHistory = KeepLast (1)



Eventual View of a Stream

Assumptions:

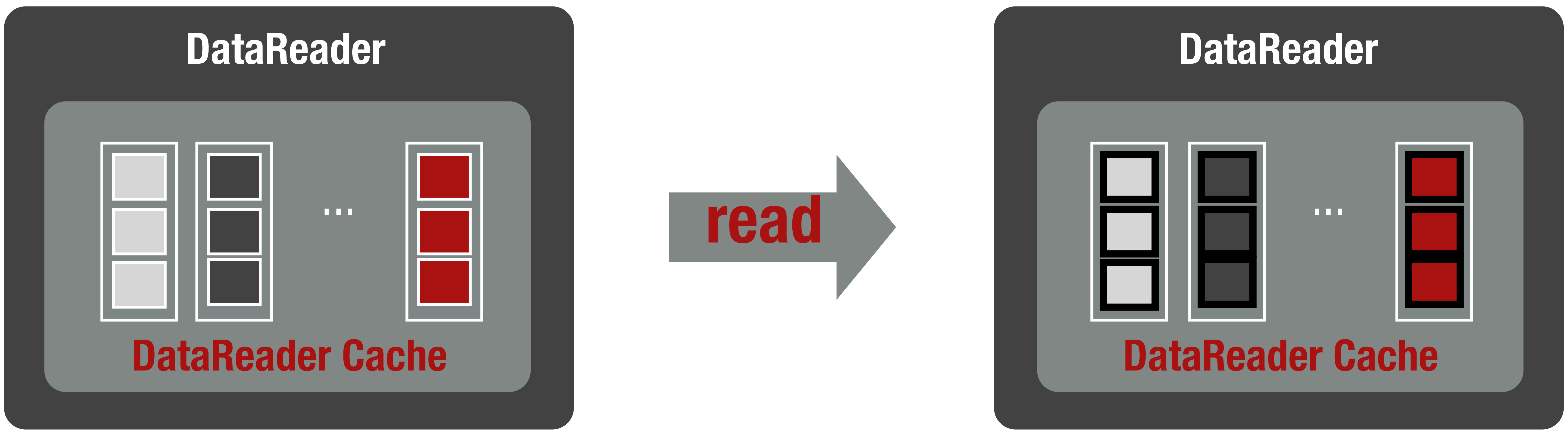
Reader History = KeepLast (n) with n > 1
WriterHistory = KeepLast (m) with n > m > 1



OpenSplice | DDS

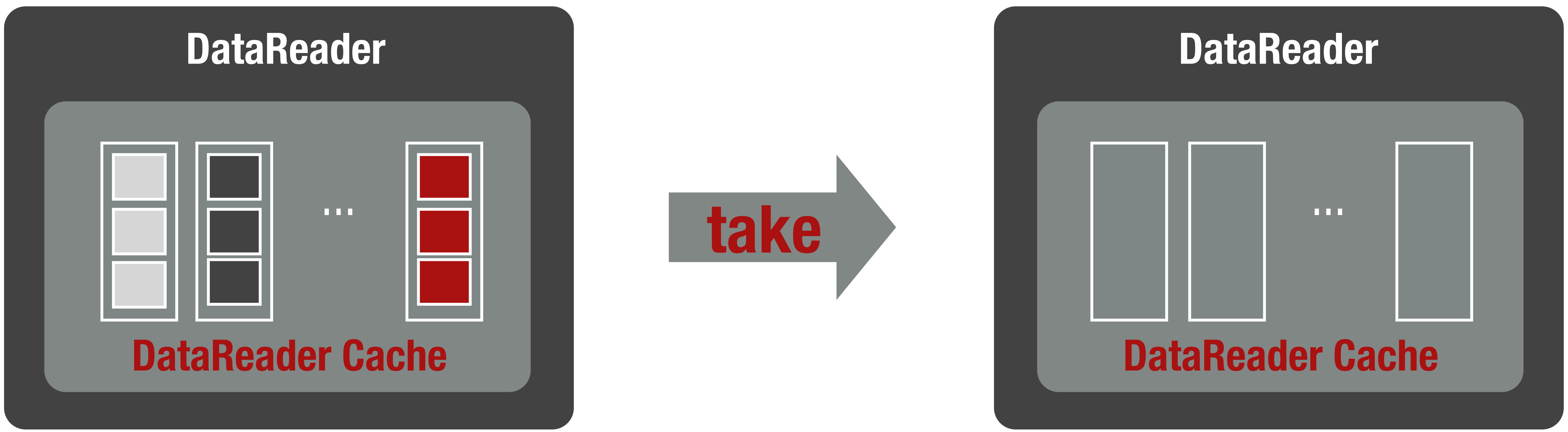
Reading Data Samples

- Samples can be read from the Data Reader History Cache
- The action of reading a sample is non-destructive. Samples are not removed from the cache



Taking Data Samples

- Samples can be taken from the Data Reader History Cache
- The action of taking a sample is destructive. Samples are removed from the cache



OpenSplice | DDS

Read vs. Take

- The **read** operation should always be access the latest know value for topics that represent **distributed state**
- The **take** operation should be used to get the last notification from a topic that represent an **event**

OpenSplice | DDS

Eventual Consistency

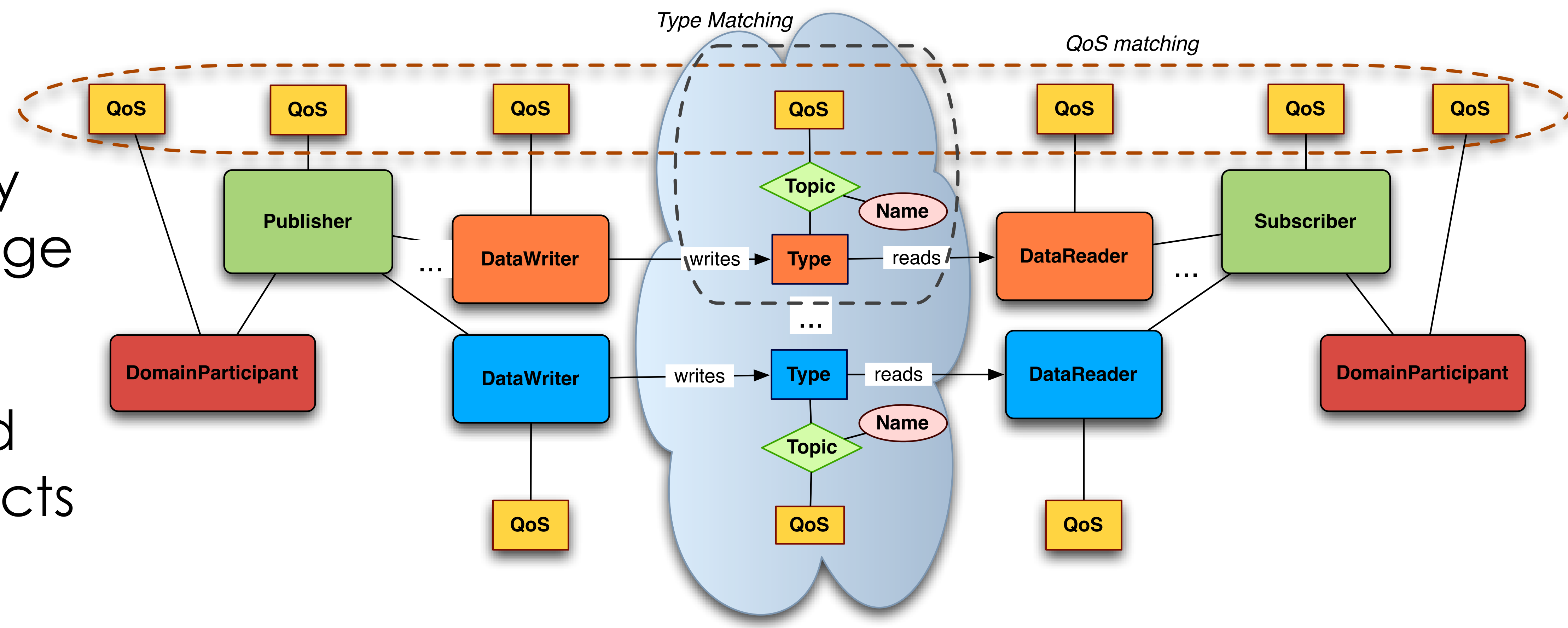
- DDS caches provide eventual consistency semantics
- This means that a **read** will see the effect of a preceding **write** eventually
- Furthermore, given a data-writer that is currently matching **N** readers, we can think of DDS as providing eventual consistency with **W=0** and **R=1**
  - **W:** the number of Acks expected in order to return from a **write**
  - **R:** the number of sources from which a read access data

OpenSplice | DDS

QoS

QoS Model

- QoS-Policies control local and end-to-end properties of DDS entities
- Local properties controlled by QoS are related resource usage
- End-to-end properties controlled by QoS are related to temporal and spatial aspects of data distribution
- Some QoS-Policies are matched based on a **Request vs. Offered** Model thus QoS-enforcement



OpenSplice | DDS

QoS Policies

[**T:** Topic] [**DR:** DataReader] [**DW:** DataWriter] [**P:** Publisher] [**S:** Subscriber] [**DP:** Domain Participant]

QoS Policy	Applicability	RxO	Modifiable	
USER_DATA	DP, DR, DW	N	Y	**Configuration**
TOPIC_DATA	T	N	Y	
GROUP_DATA	P, S	N	Y	
DURABILITY	T, DR, DW	Y	N	**Data Availability**
DURABILITY SERVICE	T, DW	N	N	
HISTORY	T, DR, DW	N	N	
PRESENTATION	P, S	Y	N	**Data Delivery**
RELIABILITY	T, DR, DW	Y	N	
PARTITION	P, S	N	Y	
DESTINATION ORDER	T, DR, DW	Y	N	
LIFESPAN	T, DW	N	Y	

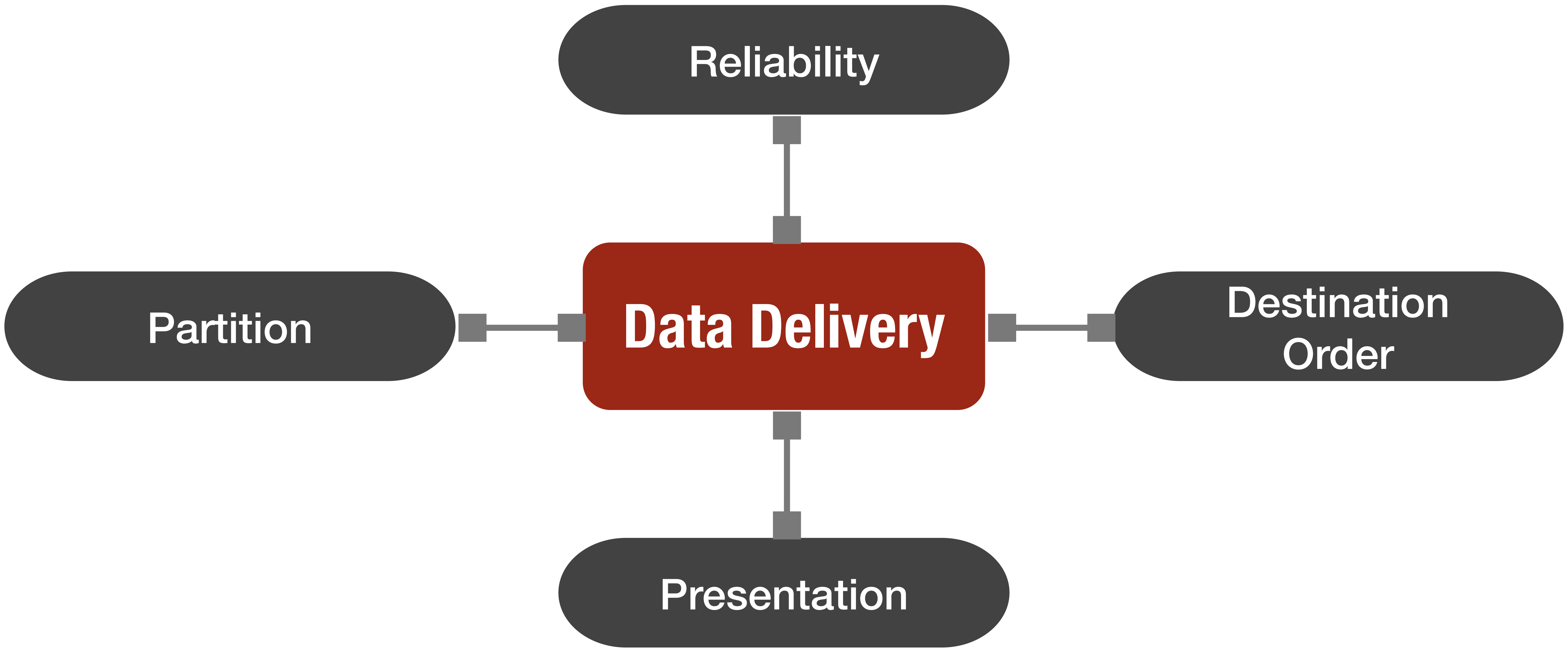
QoS Policies

[**T:** Topic] [**DR:** DataReader] [**DW:** DataWriter] [**P:** Publisher] [**S:** Subscriber] [**DP:** Domain Participant]

QoS Policy	Applicability	RxO	Modifiable	
DEADLINE	T, DR, DW	Y	Y	**Temporal/ Importance Characteristics**
LATENCY BUDGET	T, DR, DW	Y	Y	
TRANSPORT PRIORITY	T, DW	N	Y	
TIME BASED FILTER	DR	N	Y	
OWNERSHIP	T, DR, DW	Y	N	**Replication**
OWNERSHIP STRENGTH	DW	N	Y	
LIVELINESS	T, DR, DW	Y	N	**Fault-Detection**

OpenSplice | DDS

Data Delivery



Reliability QoS Policy

QoS Policy	Applicability	RxO	Modifiable
RELIABILITY	T, DR, DW	Y	N

The Reliability Policy controls the level of guarantee offered by the DDS in delivering data to subscribers

- **Reliable**. In steady-state, and no data writer crashes, the middleware guarantees that all samples in the DataWriter history will eventually be delivered to all the DataReader

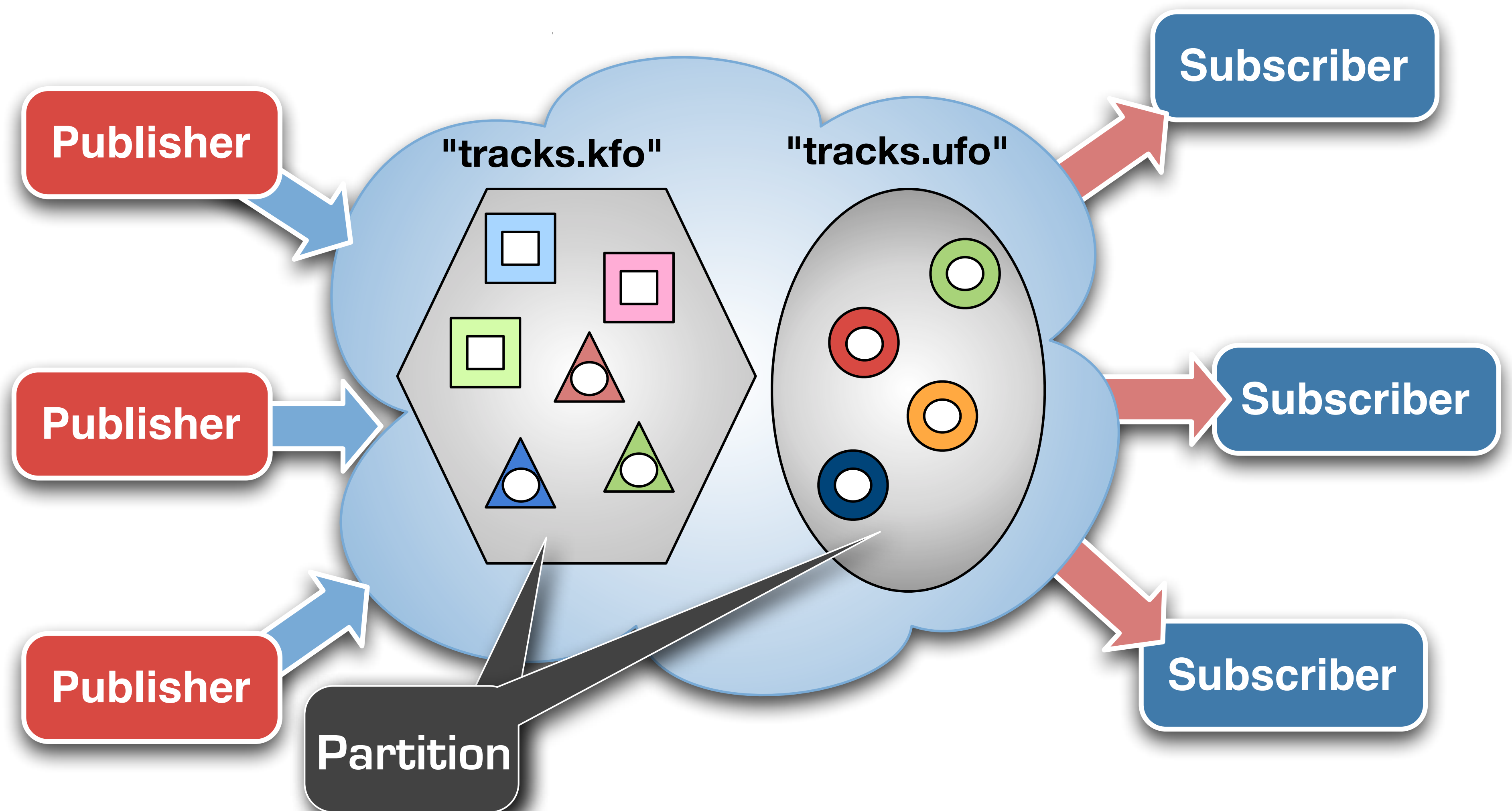
- **Best Effort**. Indicates that it is acceptable to not retry propagation of any samples

OpenSplice | DDS

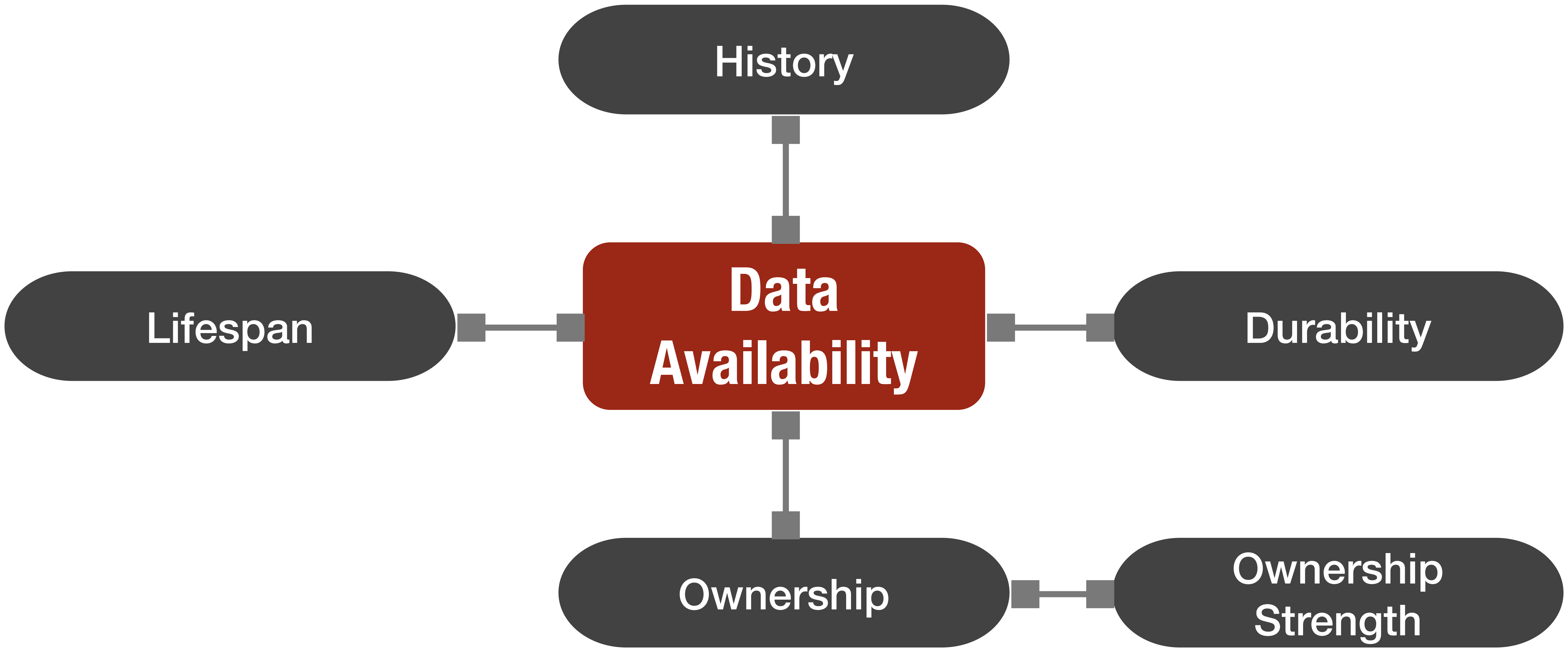
Partition QoS Policy

QoS Policy	Applicability	RxO	Modifiable
PARTITION	P, S	N	Y

- The Partition QoS Policy can be used as subjects for organizing the flows of data
- The Partition QoS Policy is used to connect Publishers/Subscribers to a Partitions' List which might also contain wildcards, e.g. tracks.*
- Topics instances are published and subscribed across one or more Partitions



Data Availability



OpenSplice | DDS

Durability QoS Policy

QoS Policy	Applicability	RxO	Modifiable
DURABILITY	T, DR, DW	Y	N

The DURABILITY QoS controls the data availability w.r.t. late joiners, specifically the DDS provides the following variants:

- **Volatile.** No need to keep data instances for late joining data readers
- **Transient Local.** Data instance availability for late joining data reader is tied to the data writer availability
- **Transient.** Data instance availability outlives the data writer
- **Persistent.** Data instance availability outlives system restarts

OpenSplice | DDS

History QoS Policy

QoS Policy	Applicability	RxO	Modifiable
HISTORY	T, DR, DW	N	N

For DataWriters, the HISTORY QoS policy controls the amount of data that can be made available to late joining DataReaders under TRANSIENT_LOCAL Durability

For DataReader, the HISTORY QoS policy controls how many samples will be kept on the reader cache

- **Keep Last.** DDS will keep the most recent "depth" samples of each instance of data identified by its key
- **Keep All.** The DDS keep all the samples of each instance of data identified by its key -- up to reaching some configurable resource limits

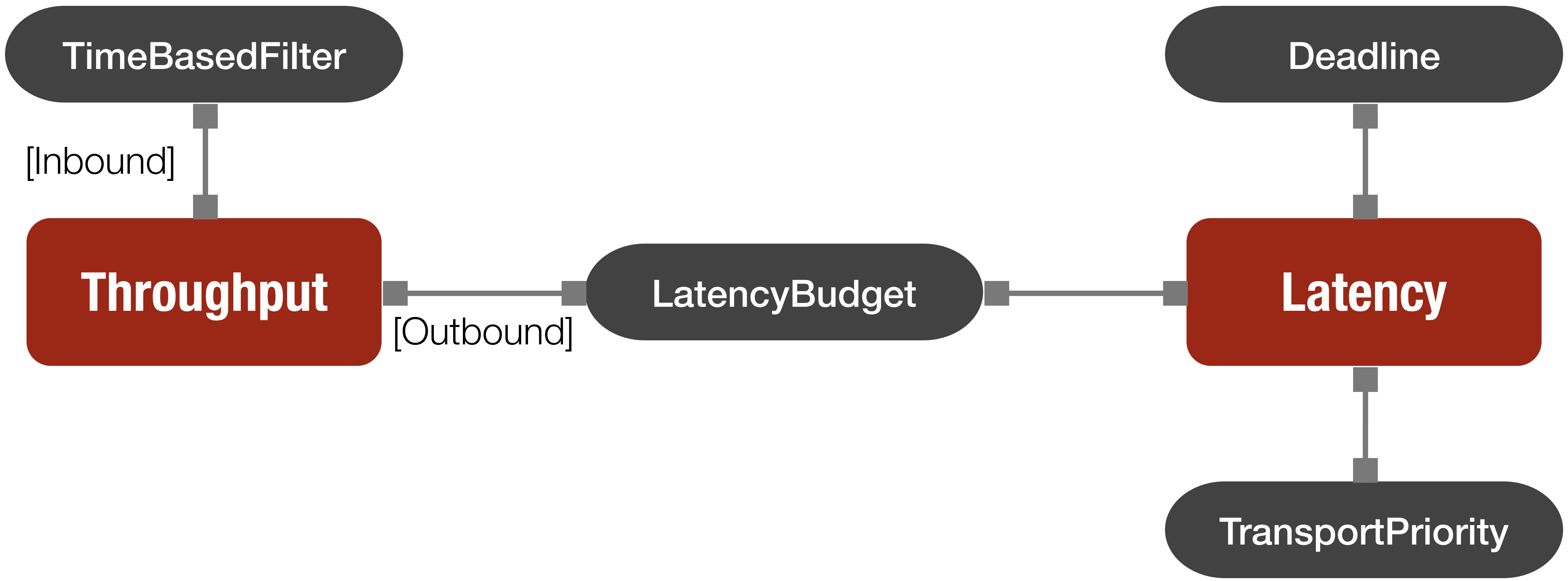
Ownership QoS Policy

QoS Policy	Applicability	RxO	Modifiable
OWNERSHIP	T, DR, DW	Y	N
STRENGTH	DW	N	Y

Availability of data producers can be controlled via two QoS Policies

- OWNERSHIP (SHARED vs. EXCLUSIVE)
- OWNERSHIP STRENGTH
- Instances of exclusively owned Topics can be modified (are owned) by the higher strength writer
- Writer strength is used to coordinate replicated writers

Temporal Properties

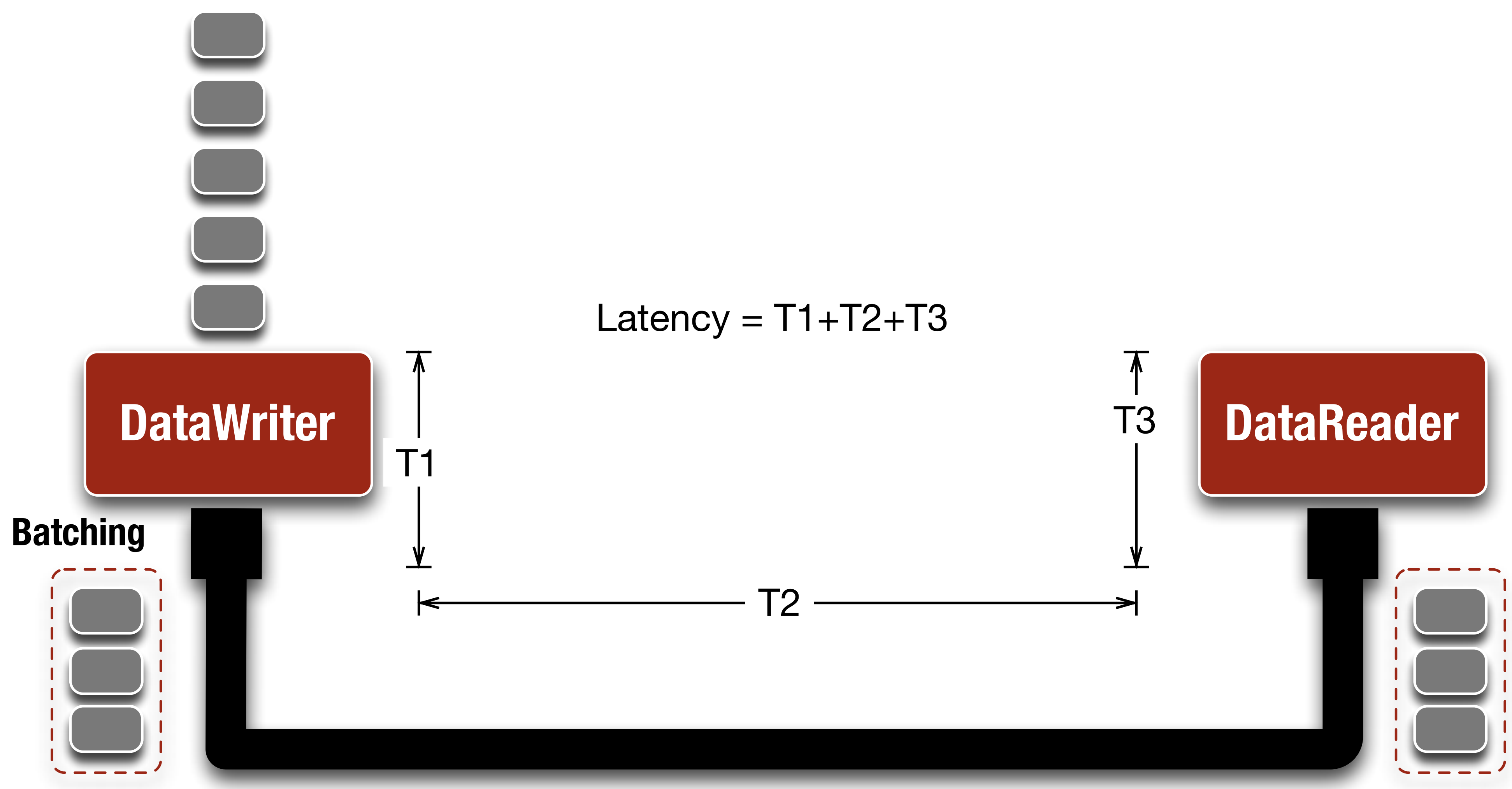


Latency Budget QoS Policy

QoS Policy	Applicability	RxO	Modifiable
LATENCY BUDGET	T, DR, DW	Y	Y

- The LATENCY_BUDGET QoS policy specifies the maximum acceptable delay from the time the data is written until the data is inserted in the receiver's application-cache

- A non-zero latency-budget allows a DDS implementation to batch samples and improve CPU/ Network utilization

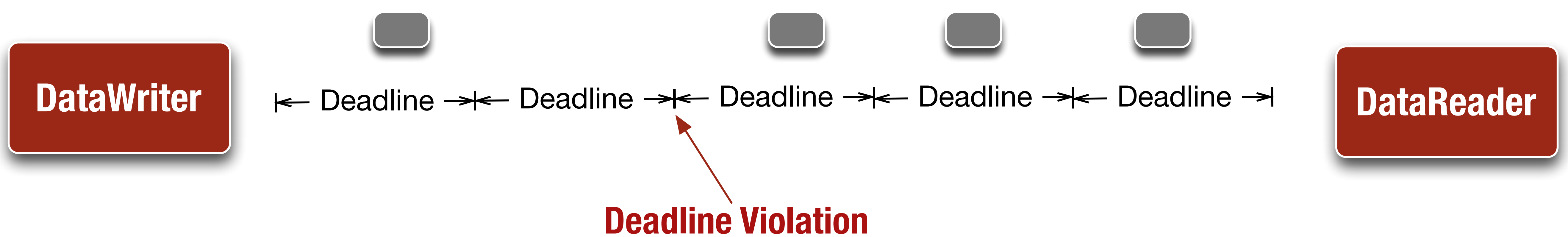


OpenSplice | DDS

Deadline QoS Policy

QoS Policy	Applicability	RxO	Modifiable
DEADLINE	T, DR, DW	Y	Y

- The DEADLINE QoS policy allows to define the maximum inter-arrival time between data samples
- DataWriter indicates that the application commits to write a new value at least once every deadline period
- DataReaders are notified by the DDS when the DEADLINE QoS contract is violated



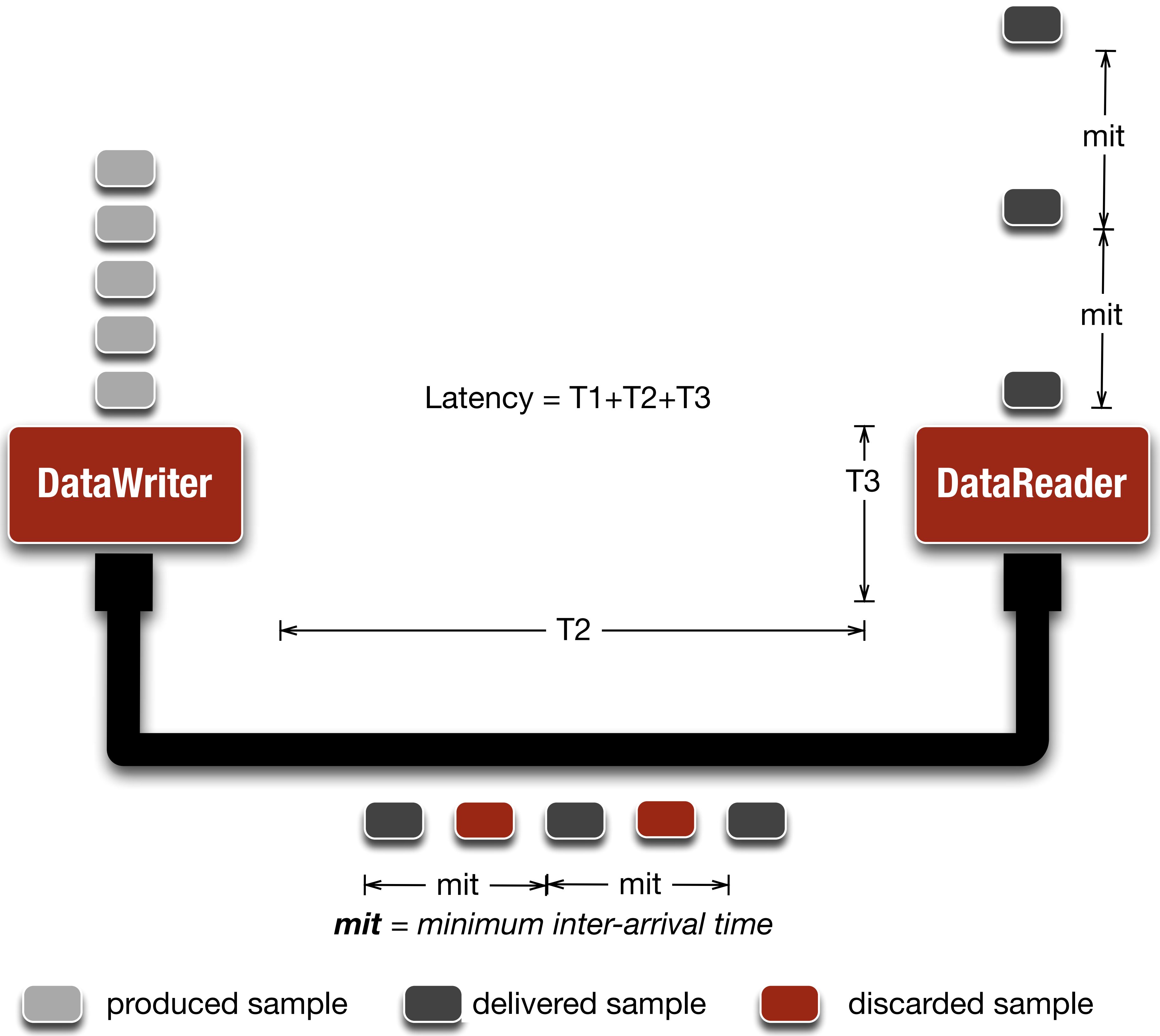
Transport Priority QoS Policy

QoS Policy	Applicability	RxO	Modifiable
TRANSPORT PRIORITY	T, DW	N	Y

- The TRANSPORT_PRIORITY QoS policy is a hint to the infrastructure as to how to set the priority of the underlying transport used to send the data.

Time-Based Filter QoS Policy

QoS Policy	Applicability	RxO	Modifiable
TIME BASED FILTER	DR	N	Y



- The Time Based Filter allows to control the throughput at which data is received by a data reader
- Samples produced more often than the minimum inter-arrival time are not delivered to the data reader

Setting QoS Policies

C++

```
// Setting Partition QoS-Policy on Publisher
qos::PublisherQos pubQos;
pubQos << policy::Partition("Partition");
Publisher pub(dp, pubQoS);

// Setting various QoS-Policy on a Topic
qos::TopicQos tqos;
tqos << policy::Reliability::Reliable()
     << policy::Durability::Transient()
     << policy::History::KeepLast(5);

Topic<VehicleDynamics> topic(dp,"Partition", tqos);
```

OpenSplice | DDS

Data Selectors

Read Styles

The new API supports two read styles

- User-Provided Buffers read
- Loaned Buffers read

User-Provided Buffers Read

```
// --- Forward Iterators: --- //
template <typename SamplesFWIterator, typename InfoFWIterator>
uint32_t
read(SamplesFWIterator sfit,
     InfoFWIterator ifit,
     size_t max_samples);


// --- Back-Inserting Iterators: --- //
template <typename SamplesBIIterator, typename InfoBIIterator>
uint32_t
read(SamplesBIIterator sbit,
     InfoBIIterator ibit);
```

OpenSplice | DDS

Example

```
uint32_t max_size = 10;
std::vector<ShapeType> data(max_size);
std::vector<DDS::SampleInfo> info(max_size);

uint32_t len =
   dr.read(data.begin(), info.begin(), max_size);

for (uint32_t i = 0; i < len; ++i)
    std::cout << data[i] << std::endl;
```

OpenSplice | DDS

Loaned Buffers read

```
dds::sub::LoanedSamples<T> read();
```

```
template <typename T,
          template <typename Q> class DELEGATE>
class dds::sub::LoanedSamples :
    public dds::core::Value< DELEGATE<T> >
{
public:
    typedef T DataType;
    typedef Sample<DataType> SampleType;

public:
    /* Snipped... */
    };

public:
    const Iterator begin() const;

    const Iterator end() const;

public:
   // explicitly return loan
    void return_loan();

};
```

OpenSplice | DDS

Cherry Picking in DDS

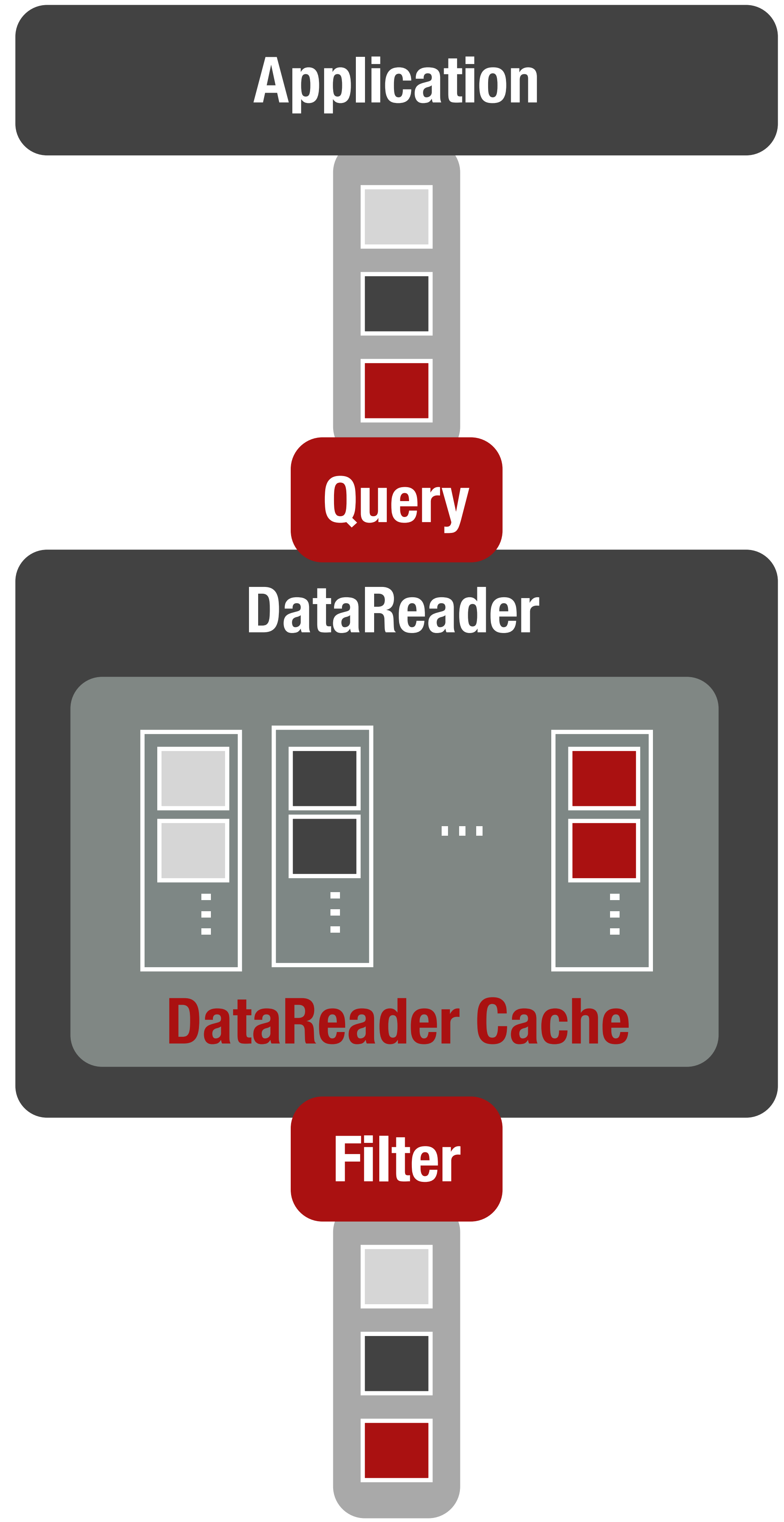
- DDS provides some very flexible mechanisms for selecting the data to be read:
  - Data Content
  - Data Status
- These mechanisms are composable

OpenSplice | DDS

Content-Based Data Selection

Filters and Queries

- DDS **Filters** allow to **control what gets into** a DataReader cache
- DDS **Queries** allow to **control what gets out** of a DataReader cache
- Filters are defined by means of **ContentFilteredTopics**
- Queries operate in conjunction with **read** operations
- Filters and Queries are expressed as SQL where clauses



OpenSplice | DDS

Filters
[Scala API]

```
struct ShapeType {
   @Key
   string   color;
   long   x;
   long   y;
   long   shapesize;
};
```

```
/**
 *  NOTE: The Scala API if not provided with DP/Sub/Pub assumes
 *  default domains and default partition.
 **/
// Create a Topic
val topic = Topic[ShapeType](“Circle”)

// Define filter expression and parameters
val query = Query(“x < %0 AND y < %1”, List(“200”, “300”))

// Define content filtered topic
val cftopic =
  ContentFilteredTopic[ShapeType](“Circle”, topic, query)

// Create a DataReader for the content-filtered Topic
val reader =
  DataReader[ShapeType](cftopic)
```

OpenSplice | DDS

Query
[DDS C++ API 2010]

```
struct ShapeType {
    @Key
    string   color;
    long    x;
    long    y;
    long    shapesize;
};
```

```
// Define the query and the parameters

std::vector<std::string> p;
p.push_back("100");
p.push_back("100");
dds::core::Query q("x < %0 AND y < %1", p.begin(), p.end());

auto data = reader
   .selector()
       .filter_content(q)
   .read();
```

OpenSplice | DDS

OpenSplice | DDS

Instances

- DDS provides a **very efficient way** of **reading** data belonging to a specific Topic **Instance**
- Obviously, one could use queries to match the key's value, but this is not as efficient as the special purpose **instance selector**

```
// C++
auto data = reader
   .selector()
       .instance(handle)
   .read();

// Scala
val data = reader read(handle)
```

State-Based Selection

Sample, Instance, and View State

- The samples included in the DataReader cache have associated some meta-information which, among other things, describes the status of the sample and its associated stream/instance
- The **Sample State (READ, NOT_READ)** allows to distinguish between new samples and samples that have already been read
- The **View State (NEW, NOT_NEW)** allows to distinguish a new instance from an existing one
- The **Intance State (ALIVE, NOT_ALIVE_DISPOSED, NOT_ALIVE_NO_WRITERS)** allows to track the life-cycle transitions of the instance to which a sample belongs

State Selector in Action

C++

```cpp
// Read only new samples
auto data = reader
   .selector()
      .filter_state(status::DataState::new_data())
   .read()

// Read any samples from live instances
auto data = reader
   .selector()
      .filter_state(status::DataState::any_data())
   .read()
```

Scala

```scala
// Read only new samples
val data = reader read

// Read any samples from live instances
val data = reader read(SampleSelector.AnyData)
```

Putting all Together

- Selectors can be composed in a flexible and expressive manner

C++

```
auto data = reader
      .selector()
         .instance(handle)
         .filter_state(status::DataState::new_data())
         .filter_content(q)
       .read();
```

OpenSplice | DDS

Communication Statuses

DataReader	SAMPLE_REJECTED	A (received) sample has been rejected.
	LIVELINESS_CHANGED	The liveliness of one or more ***DataWriter*** that were writing instances read through the ***DataReader*** has changed. Some ***DataWriter*** have become "active" or "inactive."
	REQUESTED_DEADLINE_MISSED	The deadline that the ***DataReader*** was expecting through its ***QosPolicy*** DEADLINE was not respected for a specific instance.
	REQUESTED_INCOMPATIBLE_QOS	A ***QosPolicy*** value was incompatible with what is offered.
	DATA_AVAILABLE	New information is available.
	SAMPLE_LOST	A sample has been lost (never received).
	SUBSCRIPTION_MATCHED	The *DataReader* has found a *DataWriter* that matches the *Topic* and has compatible QoS, or has ceased to be matched with a ***DataWriter*** that was previously considered to be matched.
DataWriter	LIVELINESS_LOST	The liveliness that the ***DataWriter*** has committed through its ***QosPolicy*** LIVELINESS was not respected; thus ***DataReader*** entities will consider the ***DataWriter*** as no longer "active."
	OFFERED_DEADLINE_MISSED	The deadline that the ***DataWriter*** has committed through its ***QosPolicy*** DEADLINE was not respected for a specific instance.
	OFFERED_INCOMPATIBLE_QOS	A ***QosPolicy*** value was incompatible with what was requested.
	PUBLICATION_MATCHED	The *DataWriter* has found *DataReader* that matches the *Topic* and has compatible QoS, or has ceased to be matched with a *DataReader* that was previously considered to be matched.

OpenSplice | DDS

Liveliness Changed Status

LivelinessChangedStatus	Attribute meaning.
`alive_count`	The total number of currently active DataWriters that write the Topic read by the DataReader. This count increases when a newly matched DataWriter asserts its liveliness for the first time or when a DataWriter previously considered to be not alive reasserts its liveliness. The count decreases when a DataWriter considered alive fails to assert its liveliness and becomes not alive, whether because it was deleted normally or for some other reason.
`not_alive_count`	The total count of currently DataWriters that write the Topic read by the DataReader that are no longer asserting their liveliness. This count increases when a DataWriter considered alive fails to assert its liveliness and becomes not alive for some reason other than the normal deletion of that DataWriter. It decreases when a previously not alive DataWriter either reasserts its liveliness or is deleted normally.
`alive_count_change`	The change in the alive_count since the last time the listener was called or the status was read.
`not_alive_count_change`	The change in the not_alive_count since the last time the listener was called or the status was read.
`last_publication_handle`	Handle to the last DataWriter whose change in liveliness caused this status to change.

OpenSplice | DDS

Let’s Experiment!

Advanced Topics

Advanced Topics

- Depending on time and attendees interest, I'll be covering a set of advanced topics such as:
  - Distributed State and Events
  - Advanced Distributed Algorithms with DDS, such as Leader Election, Mutual Exclusion, etc.

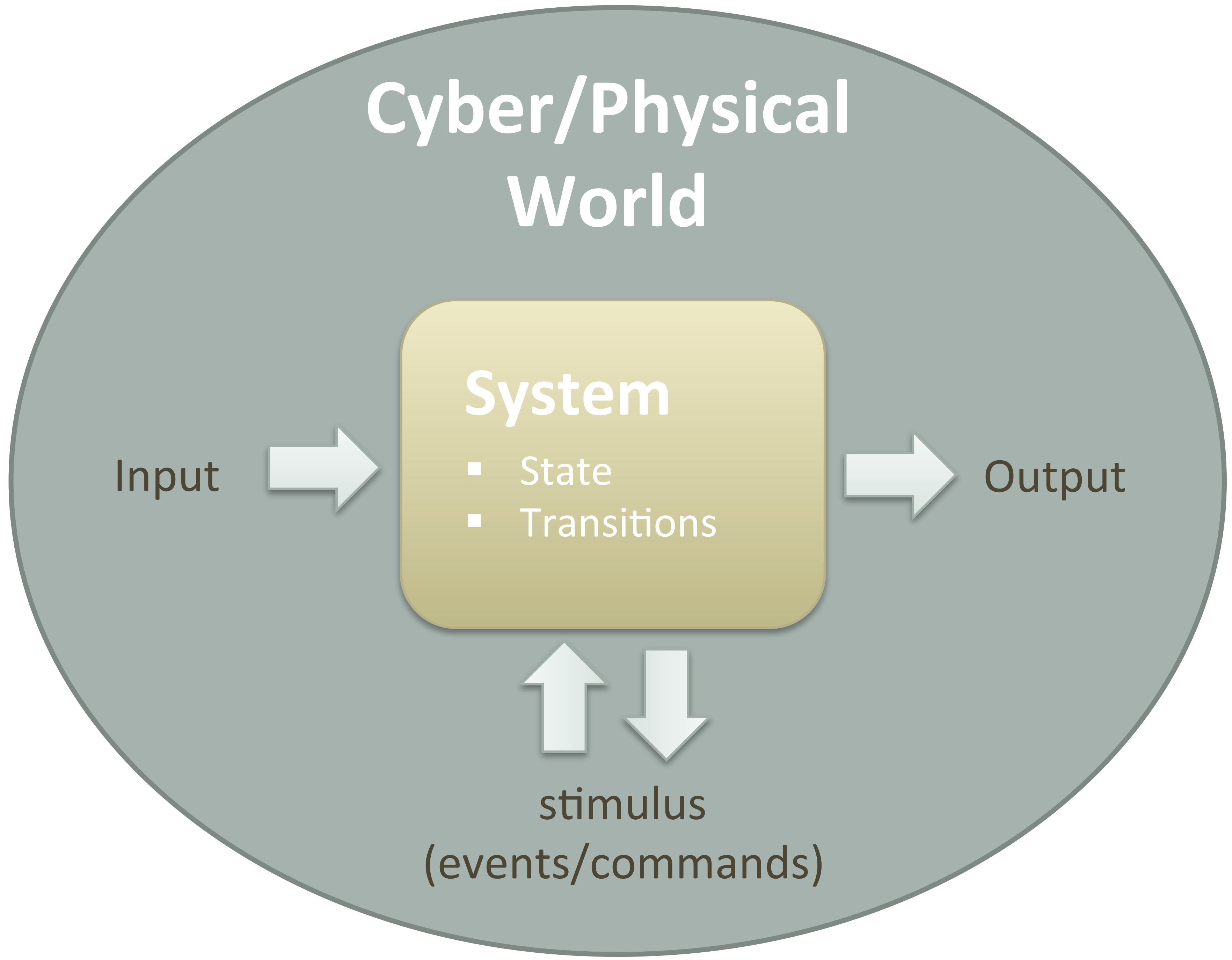
OpenSplice | DDS

Distributed Events
vs.
Distributed State

Foundations

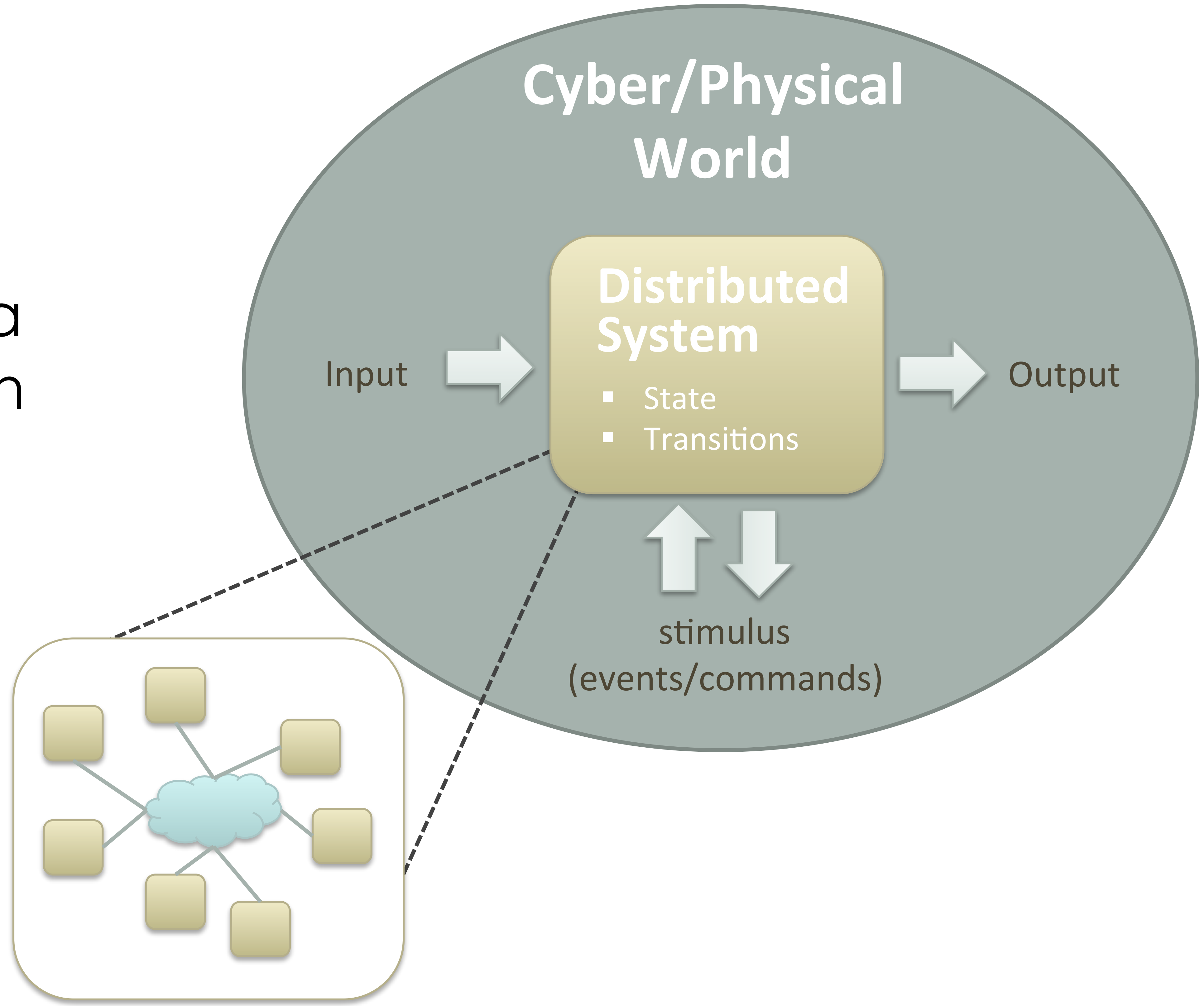
Defining a System

- A set of interacting or interdependent parts forming an integrated whole



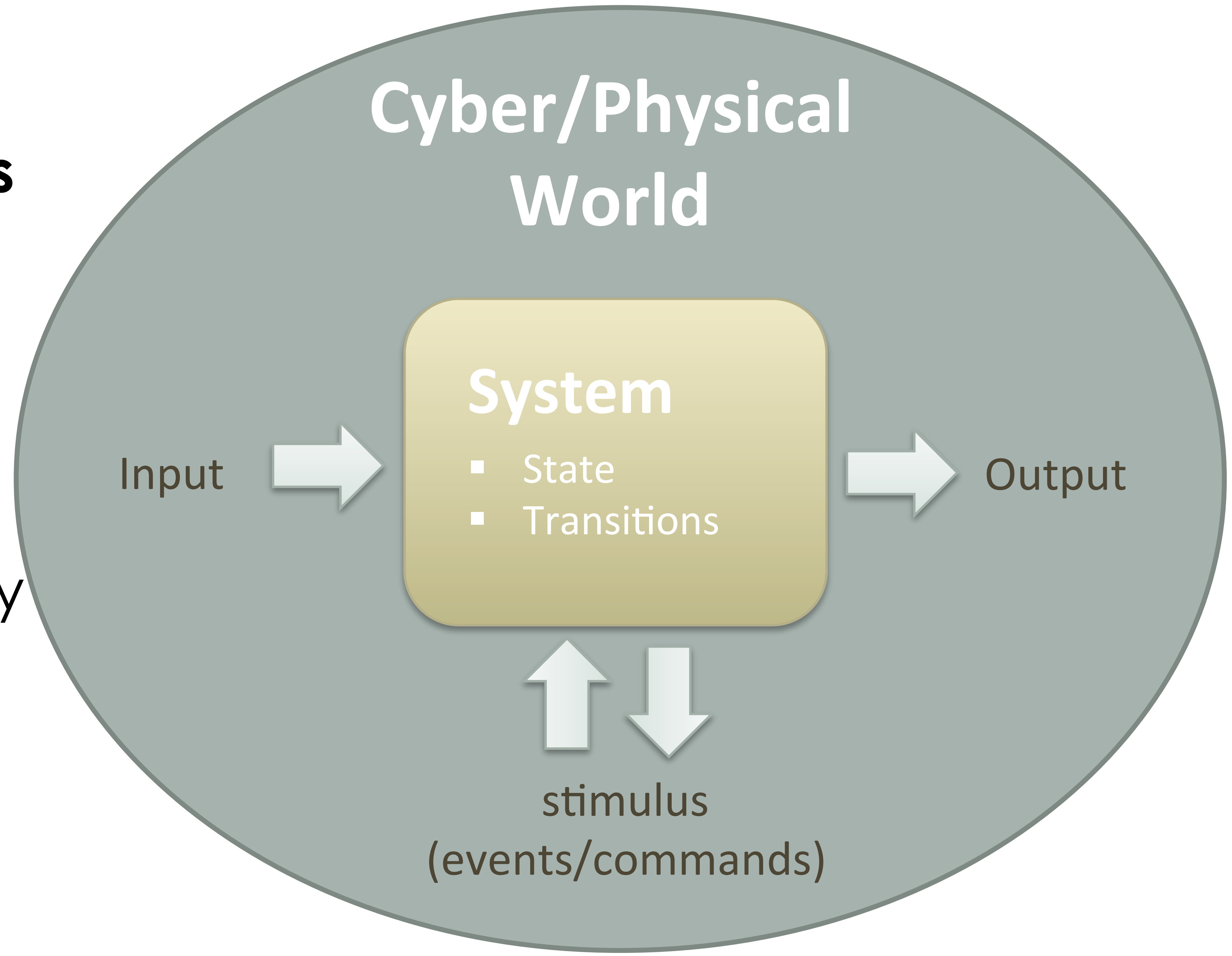
Defining a Distributed System

- A Distributed System is a System whose parts can only interact by communicating over a network



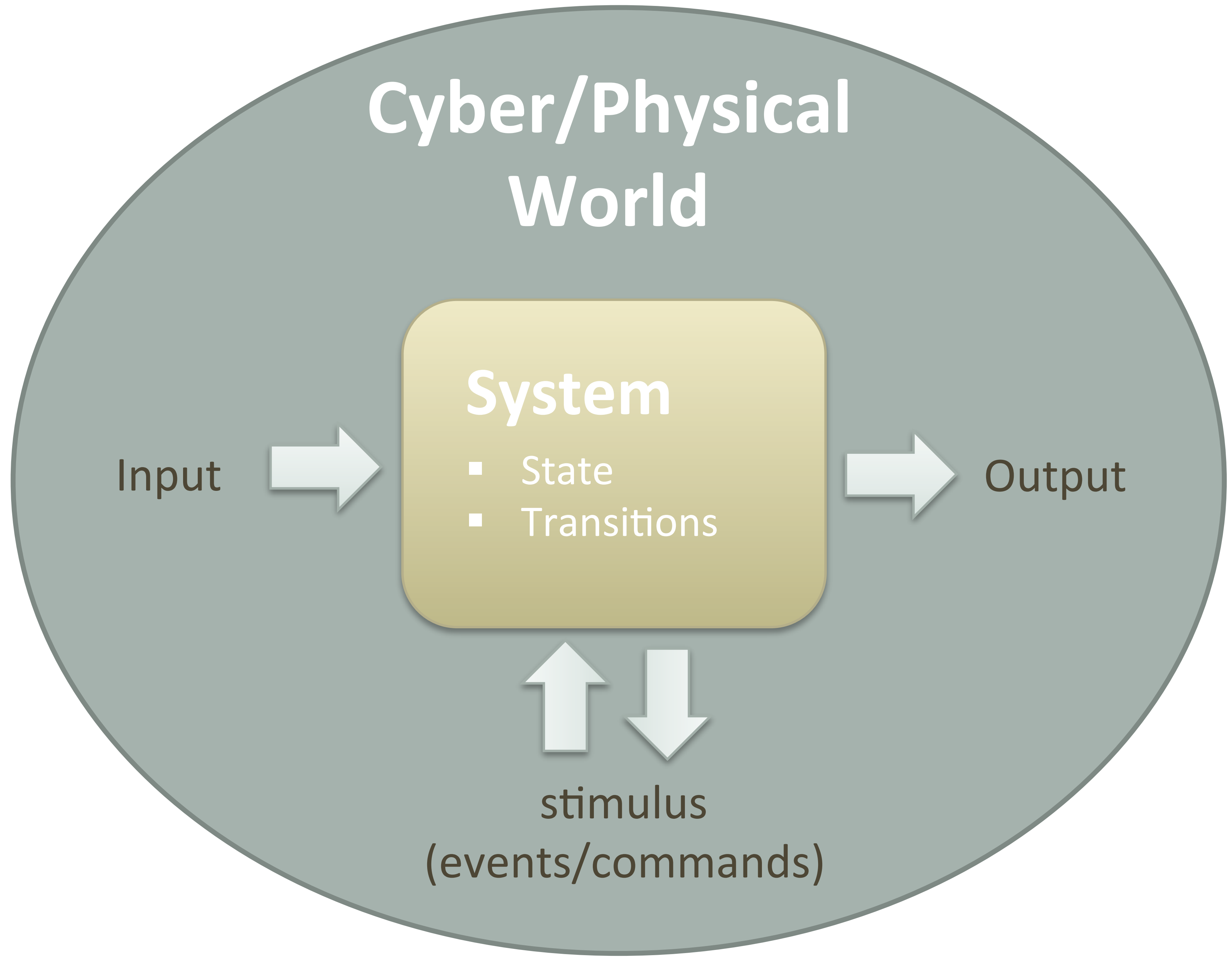
State in a Distributed System

- The State of a distributed system is the **collection of the states of its parts plus the stimulus currently propagating** through the system
- As Distributed Systems don't share memory, one problem to address is how to access arbitrary subsets of its state (or of its parts)
- The other problem is the consistency of this state...



Stimulus in a Distributed System

- Internal and Environmental Stimuli in a distributed system are used to:
  - **evolve the system state** (commands, i.e. do something)
  - **notify particular condition** on the state (events, i.e. something happened)

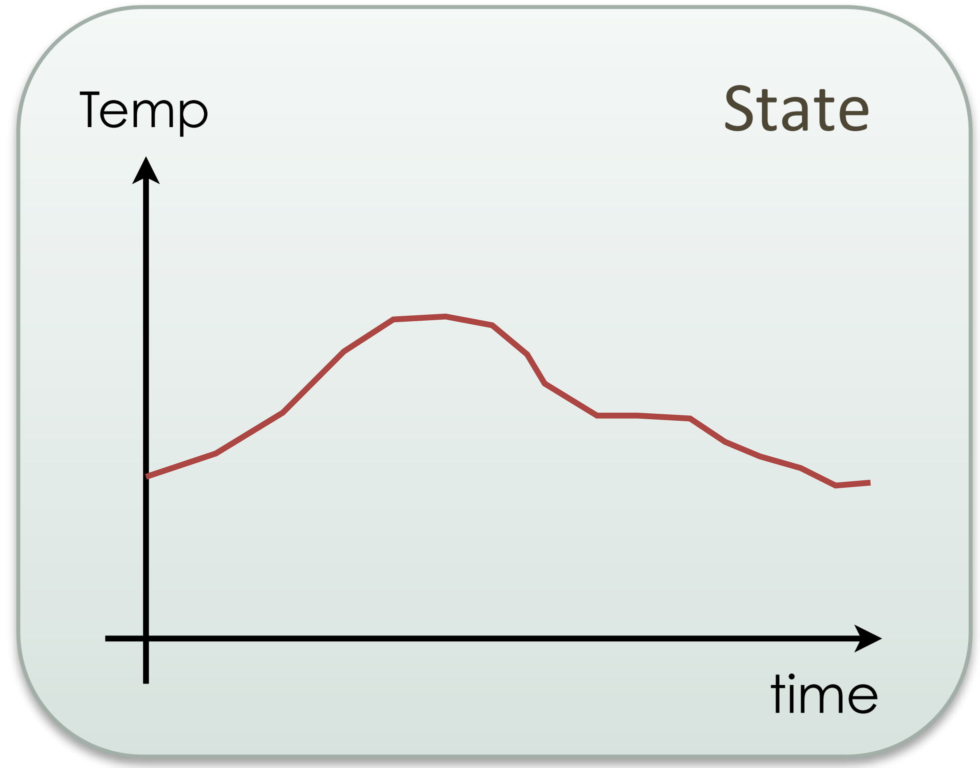


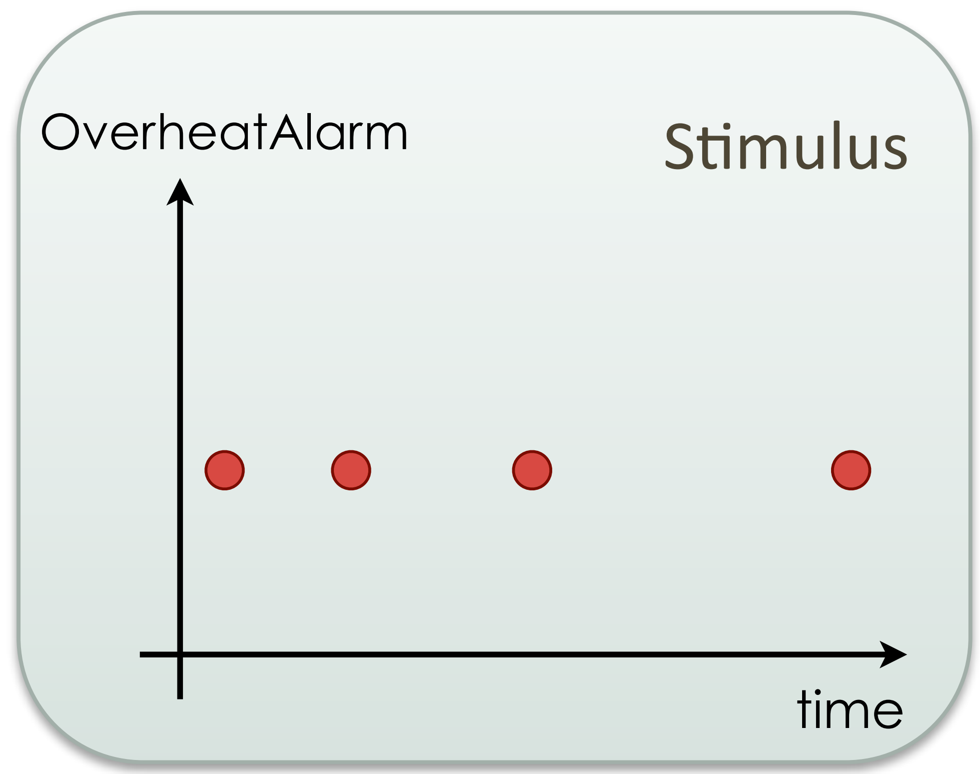
OpenSplice | DDS

State vs Stimulus

- The state of a system is always defined to have a value

- A Stimulus only exists at a particular point in time





OpenSplice | DDS

State and Events in DDS

State in DDS

Distributed State with DDS

- The "public" state of the elements making the distributed system can easily be captured via topic definitions
- Representing state with topics is more a matter of discipline w.r.t. to the QoS being used and the way in which it is accessed

OpenSplice | DDS

State's DDS QoS

Topics representing state should have the following QoS Settings

- RELIABILITY = **RELIABLE**
- HISTORY = **KEEP_LAST(1)**
- DURABILITY = **(TRANSIENT | PERSISTENT)**
- OWNERSHIP = **EXCLUSIVE**
- DESTINATION_ORDER = **SOURCE_TIMESTAMP**

OpenSplice | DDS

Soft-State's DDS QoS

Topics representing soft-state, meaning state that is periodically updated, should have the following QoS Settings

- RELIABILITY = **BEST_EFFORT**
- HISTORY = **KEEP_LAST(1)**
- DURABILITY = **VOLATILE**
- OWNERSHIP = **EXCLUSIVE**
- DESTINATION_ORDER = **SOURCE_TIMESTAMP**

Accessing State in DDS

- The `DataReader.read` operation should be used to access topics representing state
  - This ensures that the last value for the state will be kept in DDS and will be readable again and again
- The DataReader data should be accessed with the following flags:
  - ANY_SAMPLE_STATE
  - ALIVE_INSTANCE_STATE
  - ANY_VIEW_STATE

Example [1/3]

- A Robot Position in 2D is an example of state
- Let's assume that the Robot only update position when it moves
- Topic Type:

```
struct RobotPosition {
   @key
   long rid;
   long x;
   long y;
};
```

OpenSplice | DDS

Example [2/3]

- The Topic and DataReader would be constructed as follows

```
// Create Topic Qos
val tQos =
  TopicQos() <= KeepLastHistory(1)
             <= Reliable()
             <= TransientDurability()
             <= ExclusiveOwnership()
             <= SourceTimestamp();
// Create Topic
val rpt = Topic[RobotPosition](“RobotPosition”,topicQos)

// Create DataReader
val rpdr = DataReader[RobotPosition](rpt, DataReaderQos(tqos))
```

OpenSplice | DDS

Example [3/3]

- Data can be read as follows

```
// Read data
val data = rpdr.read(ReadState.AllData)

// Or specific to Escalier
val data = rpdr.history
```

OpenSplice | DDS

Events in DDS

Distributed Events with DDS

- Events raised by a distributed system can be easily captured via topic definitions
- Representing events with topics is more a matter of discipline w.r.t. to the QoS being used and the way in which it is accessed
- Event topics are often keyless

OpenSplice | DDS

Events' DDS QoS

Events should have the following QoS Settings

- RELIABILITY = **RELIABLE**
- HISTORY = **KEEP_ALL**
- DURABILITY = **VOLATILE**
- OWNERSHIP = **SHARED**
- DESTINATION_ORDER = **SOURCE_TIMESTAMP**

Events' DDS QoS

Events should have the following QoS Settings

- RELIABILITY = **RELIABLE**
- HISTORY = **KEEP_ALL**
- DURABILITY = **VOLATILE**
- OWNERSHIP = **SHARED**
- DESTINATION_ORDER = **SOURCE_TIMESTAMP**

OpenSplice | DDS

Accessing Events in DDS

- The `DataReader.take` operation should be used to access topics representing events
  - This ensures that the DDS cache is always freed by available events
- The DataReader data should be accessed with the following flags:
  - NEW_SAMPLE_STATE
  - ALIVE_INSTANCE_STATE
  - ANY_VIEW_STATE

OpenSplice | DDS

Example [1/3]

- A CollisionEvent could be raised by a Robot when it is colliding (or about to collide) with something

- Topic Type:

```
struct CollisionEvent {
   long detectingRobotId;
   long collidingRobotId;
   long xe;
   long ye;
};
```

OpenSplice | DDS

Example [2/3]

- The Topic and DataReader would be constructed as follows

```
// Create Topic Qos
val tQos =
  TopicQos() <= KeepAll()
             <= Reliable()
             <= VolatileDurability()
             <= SharedOwnership()
             <= SourceTimestamp();
// Create Topic
val cet = Topic[CollisionEvent](“CollisionEvent”,topicQos)

// Create DataReader
val cedr = DataReader[CollisionEvent](cet, DataReaderQos(tqos))
```

OpenSplice | DDS

Example [3/3]

- Data can be read as follows

```
// Take data
val data = cedr.take()
```

OpenSplice | DDS

Distributed Mutex

OpenSplice | DDS

Lamport's Distributed Mutex

- A relatively simple Distributed Mutex Algorithm was proposed by Leslie Lamport as an example application of Lamport's Logical Clocks
- The basic protocol (with Agrawala optimization) works as follows (sketched):
  - When a process needs to enter a critical section sends a MUTEX request by tagging it with its current logical clock
  - The process obtains the Mutex only when he has received ACKs from all the other process in the group
  - When process receives a Mutex requests he sends an ACK only if he has not an outstanding Mutex request timestamped with a smaller logical clock

Mutex Abstraction

- A base class defines the Mutex Protocol
- The Mutex companion uses dependency injection to decide which concrete mutex implementation to use

```
abstract class Mutex  {
  def acquire()

  def release()

}
```

OpenSplice | DDS

Foundation Abstractions

- The mutual exclusion algorithm requires essentially:
  - FIFO communication channels between group members
  - Logical Clocks
  - MutexRequest and MutexAck Messages

These needs, have now to be translated in terms of topic types, topics, readers/writers and QoS Settings

Topic Types

- For implementing the Mutual Exclusion Algorithm it is sufficient to define the following topic types:

```
struct TLogicalClock {
    long ts;
    long mid;
};
#pragma keylist LogicalClock mid


struct TAck {
   long amid; // acknowledged member-id
   LogicalClock ts;
};
#pragma keylist TAck ts.mid
```

OpenSplice | DDS

Topics

We need essentially two topics:

- One topic for representing the Mutex Requests, and
- Another topic for representing Acks

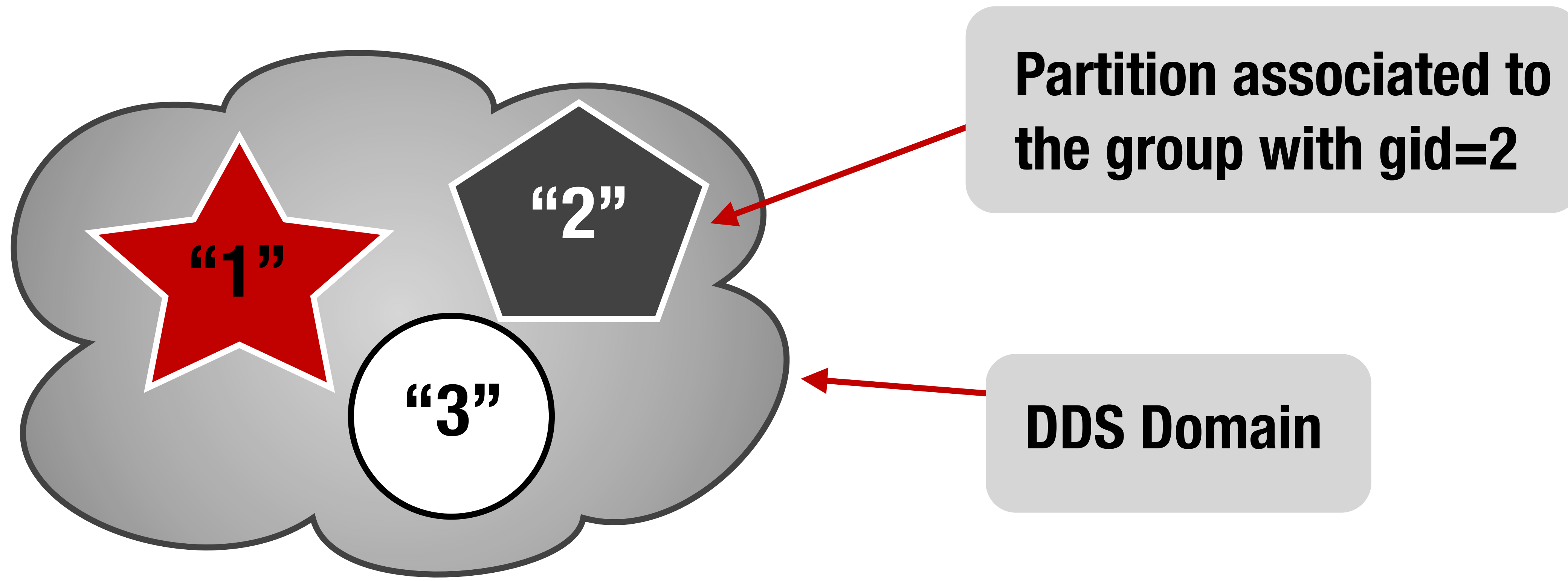
This leads us to:

- **Topic**(**name** = MutexRequest, **type** = TLogicalClock, **QoS** = {Reliability.Reliable, History.KeepAll})
- **Topic**(**name** = MutexAck, **type** = TAck, **QoS** = {Reliability.Reliable, History.KeepAll})

OpenSplice | DDS

Distinguishing Groups

- To distinguish between members belonging to different groups we introduce a group-id **gid** that is used to uniquely identify a group
- At a DDS-level, the **gid** is used to **name** the **partition** in which all the group related traffic will take place



Show me the Code!

- All the **algorithms** presented were **implemented** using **DDS** and **Scala**
- Specifically we've used the **OpenSplice Escalier** language mapping for Scala
- The resulting library has been baptized "**dada**" (DDS Advanced Distributed Algorithms) and is available under LGPL-v3

LCMutex

- The LCMutex is one of the possible Mutex protocol, implementing the Agrawala variation of the classical Lamport's Algorithm

```
class LCMutex(val mid: Int, val gid: Int, val n: Int)(implicit val logger: Logger) extends Mutex {

  private var group = Group(gid)
  private var ts = LogicalClock(0, mid)
  private var receivedAcks = new AtomicLong(0)

  private var pendingRequests = new SynchronizedPriorityQueue[LogicalClock]()
  private var myRequest =  LogicalClock.Infinite

  private val reqDW =
   DataWriter[TLogicalClock](LCMutex.groupPublisher(gid), LCMutex.mutexRequestTopic, LCMutex.dwQos)
  private val reqDR =
   DataReader[TLogicalClock](LCMutex.groupSubscriber(gid), LCMutex.mutexRequestTopic, LCMutex.drQos)

  private val ackDW =
   DataWriter[TAck](LCMutex.groupPublisher(gid), LCMutex.mutexAckTopic, LCMutex.dwQos)
  private val ackDR =
   DataReader[TAck](LCMutex.groupSubscriber(gid), LCMutex.mutexAckTopic, LCMutex.drQos)

  private val ackSemaphore = new Semaphore(0)
```

OpenSplice | DDS

LCMutex.acquire

```
def acquire() {
    ts = ts.inc()
    myRequest = ts
    reqDW ! myRequest
    ackSemaphore.acquire()
  }
```

Notice that as the LCMutex is single-threaded we can't issue concurrent acquire.

OpenSplice | DDS

LCMutex.release

```
def release() {
    myRequest = LogicalClock.Infinite
    (pendingRequests dequeueAll) foreach { req =>
      ts = ts inc()
      ackDW ! new TAck(req.id, ts)
    }
```

Notice that as the LCMutex is single-threaded we can't issue a new request before we release.

OpenSplice | DDS

LCMutex.onACK

```
ackDR.reactions += {
    case DataAvailable(dr) => {
      // Count only the ACK for us
      val acks = ((ackDR take) filter (_.amid == mid))
      val k = acks.length
      // Set the local clock to the max (tsi, tsj) + 1
      synchronized {
        val maxTs = math.max(ts.ts, (acks map (_.ts.ts)).max) + 1
        ts = LogicalClock(maxTs, ts.id)
      }
      val ra = receivedAcks.addAndGet(k)
      val groupSize = group.size
      // If received sufficient many ACKs we can enter our Mutex!
      if (ra == groupSize - 1) {
        receivedAcks.set(0)
        ackSemaphore.release()
      }
    }
  }
```

OpenSplice | DDS

LCMutex.onReq

```
reqDR.reactions += {
   case DataAvailable(dr) => {
     val requests = (reqDR take) filter (_.mid != mid)
     synchronized {
       val maxTs = math.max((requests map (_.ts)).max, ts.ts) + 1
       ts = LogicalClock(maxTs, ts.id)
     }
     requests foreach (r => {
       if (r < myRequest) {
         ts = ts inc()
         val ack = new TAck(r.mid, ts)
         ackDW ! ack
       }
       else {
         (pendingRequests find (_ == r)).getOrElse( {
           pendingRequests.enqueue(r)
           r
         })

       }
     })
   }
 }
```

OpenSplice | DDS

Dealing with Faults...

How to deal with Faults?

- The algorithm presented here intentionally ignores failures to keep the presentation simple
- The failure of a single group member would violate progress
- It is not hard to extend the algorithm to deal with failures, especially under the assumption of eventual synchrony
- If you want to learn more attend the following RTWS-2012 presentation this Thursday:

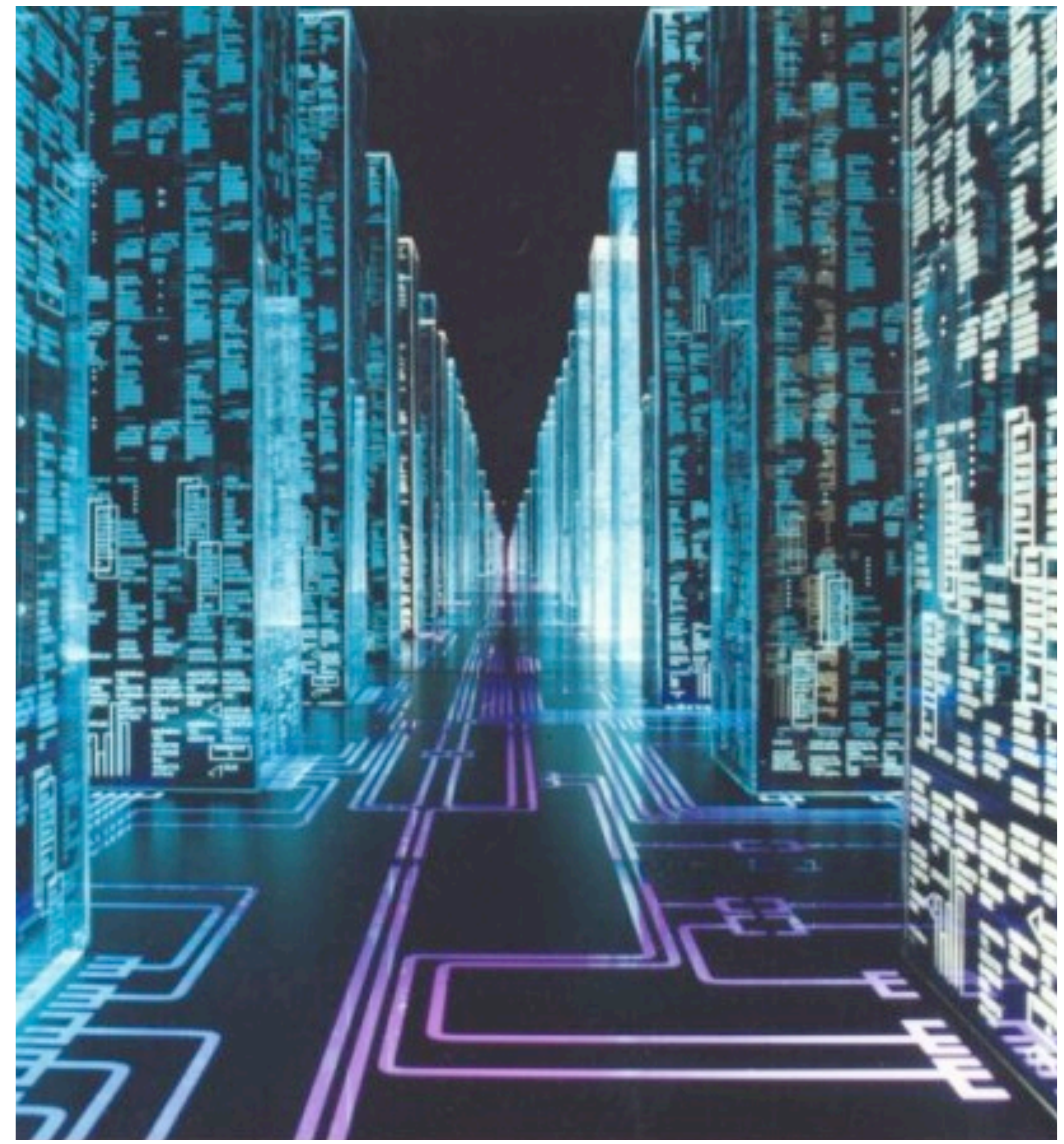
> **Classical Distributed Algorithms with DDS**
> *Sara Tucci-Piergiovanni, Research Engineer, CEA LIST*
> *Angelo Corsaro, Chief Technology Officer, PrismTech*

OpenSplice | DDS

Concluding Remarks

Concluding Remarks

- The **DDS** provides a **powerful and feature-rich topic-based publish/subscribe** abstraction
- This technology is **widely used in mission and business critical systems** and it being swiftly adopted in **data-centric/big-data** systems
- Differently from what some people think, **DDS is very simple** to get-started with
- Very good Open Source implementation are available... **Good Hacking!**



OpenSplice | DDS

References

dada

- **D**DS-based **A**dvanced **D**istributed **A**lgorithms Toolkit
- Open Source
- github.com/kydos/dada

OpenSplice | DDS

- #1 OMG DDS Implementation
- Open Source
- www.opensplice.org

Scala

- Fastest growing JVM Language
- Open Source
- www.scala-lang.org

Escalier

- Scala API for OpenSplice DDS
- Open Source
- github.com/kydos/escalier

SimD [C++]

- Simple C++ API for DDS
- Open Source
- code.google.com/p/simd-cxx

SimD [Java]

- DDS-PSM-Java for OpenSplice DDS
- Open Source
- github.com/kydos/simd-java

DDS-PSM-Cxx 2010

- DDS-PSM-Cxx API Standard
- Open Source
- github.com/kydos/dds-psm-cxx

:: Connect with Us ::

OpenSplice | DDS

- opensplice.com
- forums.opensplice.org
- opensplice.org
- opensplicedds@prismtech.com

twitter

- @acorsaro
- @prismtech

YouTube

- youtube.com/opensplicetube

slideshare

- slideshare.net/angelo.corsaro

PRISMTECH

- crc@prismtech.com
- sales@prismtech.com

OpenSplice | DDS

Appendix

Stepping into Scala

Angelo CORSARO, Ph.D.
Chief Technology Officer
OMG DDS Sig Co-Chair
PrismTech
angelo.corsaro@prismtech.com

OpenSplice | DDS

What is Scala

- Scala (pronounced Skah-lah) stands for “Scalable language”
- It is a language that carefully and creatively blends Object Oriented and Functional constructs into a statically typed language with sophisticated type inference
- Scala targets the JVM and .NET runtime and is 100% compatible with Java

OpenSplice | DDS

Why Should You Care?

- Scala is simple to write, extremely compact and easy to understand
- Scala is strongly typed with a structural type system
- Scala is an extensible language (many construct are build in the standard library)
- Scala makes it easy to design Domain Specific Language

Case Study: Complex Numbers

Complex Numbers

- To explore some of the nice features of Scala, let's see how we might design a Complex number class
- What we expect to be able to do is all mathematical operations between complex numbers as well as scalar multiplications and division
  - [(1+i2)+2*(3-i5)*(i4)]/(1+i3)
  - ~(1+i2) [conjugate]
  - !(3+i4) [Modulo]

Constructor

- Scala allows to implicitly create constructors and attributes starting from a simple argument list associated with the class declaration

```
class Complex(val re: Float, val im: Float)
```

OpenSplice | DDS

In Java

```java
public class Complex {

   private final float re;
   private final float im;

   public Complex(float re, float im) {
      this.re = re;
      this.im = im;
   }
   public Complex(float f) {
      this.re = f;
      this.im = 0;
   }

   public float re() { return re;}

   public float im() { return im;}
```

OpenSplice | DDS

Methods

- Everything in Scala is a method even operators
- Methods name can be symbols such as *, !, +, etc.

```
def + (c: Complex) : Complex = Complex(re + c.re, im + c.im)
```

or, taking advantage of type inference....

```
def + (c: Complex) = Complex(re + c.re, im + c.im)
```

In Java...

```java
public Complex add(Complex that) {
     return new Complex(this.re() + that.re(),
                        this.im() + that.im());
  }
```

As a result...

Scala

```scala
val result = Complex(1,2) + Complex(2,3)
```

Java

```java
Complex c1 = new Complex(1, 2);
Complex c2 = new Complex (3,4);
Complex c3 = c1.add(c2);
```

or...

```java
Complex c3 = (new Complex(1, 2).add(new Complex (3,4));
```

OpenSplice | DDS

Companion Object

- Scala does not have the concept of static methods/attributes
- On the other hand it provides built-in support for Singletons, which are specified with the "object" keyword as opposed to the "class" keyword

The companion object, is the object associated with a class, which shares the same name and provides typically helper methods

```
object Complex {
   def apply(real: Float, img: Float) = new Complex(real, img)

   def apply(real: Float) = new Complex(real, 0)

   implicit def floatToReComplex (f: Float) = new ReComplex(f)

   implicit def intToReComplex(i : Int) = new ReComplex(i)
}
```

OpenSplice | DDS

“apply” Magic

- When an instance of a class is followed by parentheses with a list of zero or more parameters, the compiler invokes the apply method for that instance
- This is true for an object with a defined apply method (such as a companion object), as well as an instance of a class that defines an apply method

```
val result = Complex(1,2)
```

is the same as....

```
val result = Complex.apply(1,2)
```

OpenSplice | DDS

Negation and Scalar Multiplication

- In order to design a Complex class that is well integrated in our type system we should be able to support the following cases:
  - -(a+ib)
  - c*(a+ib)
  - (a+ib)*c
- How can we supporting something like -(a+ib) and c*(a+ib)?

OpenSplice | DDS

Scala Unary Operators

- Scala allows to define unary operators for the following method identifiers +, -, !, ~

```
def unary_-() = Complex(-re, -im)

def unary_!() = Math.sqrt(re*re + im*im)

def unary_~() = Complex(re, -im)
```

as a result we can write:

```
val result = -Complex(1,2) + ~Complex(2,3)
```

OpenSplice | DDS

Scala Implicit Conversions

- The expression: `val c3 = 3*Complex(5, 7)`
- Is equivalent to:

```
val c3 = 3.*(Complex(5, 7))
```

- Yet, the method to multiply a Integer to a Complex is not present in the Scala Int class
- What can we do to make the trick?

OpenSplice | DDS

Scala Implicit Conversions

- Scala does not support Open Classes, thus allowing to add new methods to existing classes
- Yet Scala supports implicit conversions that can be used to achieve the same result
- Lets see how...

OpenSplice | DDS

Scala Implicit Conversion

```
object Complex {
   implicit def floatToReComplex (f: Float) = new ReComplex(f)

   implicit def intToReComplex(i : Int) = new ReComplex(i)
}


class ReComplex(re: Float) {

   def * (that: Complex) = Complex(re*that.re,re*that.im)

}
```

OpenSplice | DDS

The Result is...

```
val c3 = 3*Complex(5, 7)
```

is converted automatically into:

```
val c3 = ReComplex(3).*(Complex(5, 7))
```

Putting it all together

```scala
case class Complex(val re: Float, val im: Float) {

    // Binary Operators
    def + (c: Complex) = Complex(re + c.re, im + c.im)
    def - (c: Complex) = Complex(re - c.re, im - c.im)

    def * (f: Float) = Complex(f*re, f*im)
    def * (c: Complex) = Complex((re*c.re) - (im*c.im),
                                 ((re*c.im) + (im*c.re)))

    def / (c: Complex) = {
        val d = c.re*c.re + c.im*c.im
        Complex(((re*c.re) + (im + c.im))/d,
              ((im*c.re) - (re*c.im))/d )
    }

   // Unary Operators
    def unary_-() = Complex(-re, -im)
    def unary_!() = Math.sqrt(re*re + im*im)
    def unary_~() = Complex(re, -im)

    // Formatting
    override def toString() : String  = {
        if (im > 0) re + "+i" + im
        else if (im < 0) re + "-i" + (-im)
        else re.toString

    }
}
```

OpenSplice | DDS

Functions, Closures and Currying

Functions

- Scala has first-class functions
- Functions can be defined and called, but equally functions can be defined as unnamed literals and passed as values

```
def inc(x: Int) = x + 1
inc(5)
```

```
val vinc = (x: Int) => x+1
vinc(5)
```

Notice once again the uniform access principle

Playing with Functions

```scala
val list = List(1,2,3,4,5,6,7,8,9)
val g5 = list.filter((x: Int) => x > 5)
g5: List[Int] = List(6, 7, 8, 9)
```

Or with placeholder syntax

```scala
val list = List(1,2,3,4,5,6,7,8,9)
val g5 = list.filter(_ > 5)
g5: List[Int] = List(6, 7, 8, 9)
```

OpenSplice | DDS

Closures

- Scala allows you to define functions which include **free variables** meaning variables whose value is not bound to the parameter list
- Free variable are resolved at runtime considering the closure of visible variable
- Example:

```
def mean(e : Array[Float]) : Float = {
      var sum = 0.0F
      e.foreach((x: Int) => sum += x)
      return sum/e.length
   }
```

```
def mean(e : Array[Float]) : Float = {
             var sum = 0.0F
          e.foreach(sum += _)
          return sum/e.length
                   }
```

OpenSplice | DDS

Currying

- Scala provides support for curried functions which are applied to multiple argument lists, instead of just one
- Currying is the mechanism Scala provides for introducing new control abstraction

```
def curriedSum(x: Int)(y: Int) = x + y

curriedSum(1) {
   3 +5
}
```

OpenSplice | DDS

Traits

Traits

- Scala supports single inheritance from classes but can mix-in multiple traits
- A trait is the unit of code reuse for Scala. It encapsulate methods and field definitions
- Traits usually expect a class to implement an abstract method, which constitutes the “narrow” interface that allows to implement a rich behaviour
- Traits are also very useful for **dependency injection**

OpenSplice | DDS

Ordered Complex Numbers

- Our complex numbers are not comparable
- Let's assume that we wanted to make them comparable, and lets supposed that we define the total order as based on the module of the complex number
- How can we implement this behavior?

OpenSplice | DDS

Ordered Trait

- The Ordered[T] traits encapsulates the set of methods that allow to define a total ordering over a type
- All the behaviour is defined in terms of an abstract method, namely “compare”
- Classes that mix-in this trait have to implement the “compare” method

```
class Complex(val re: Float, val im: Float) extends
Ordering[Complex] {
def compare(x: Complex, y: Complex) = {
      if (x == y) 0
      else if (!x > !y) 1
      else -1
   }
                          }
```

OpenSplice | DDS

Case Classes
&
Pattern Matching

Case Classes and Pattern Matching

- Case Classes and Pattern Matching are twin constructs that are pretty useful when dealing with tree-like recursive data structures
- These constructs allow to match patterns in an expression and reconstruct the object graph that makes it up
- Lets see an example...

OpenSplice | DDS

Case Classes and Pattern

```
abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Float) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String, left: Expr, right: Expr)
```

```
def simplifyExpr(expr: Expr) : Expr = expr match {
   case UnOp("-", UnOp("-", e)) => e
   case BinOp("+", e, Number("0")) => e
   case BinOp("*", e, Number("1")) => e
   case _ => expr
}
```

OpenSplice | DDS

Type Parametrization

Type Parametrization

- Scala provides support for type parametrization and makes it available for both classes as well as traits

```
trait Queue[T] {
   def head: T
   def tail: Queue[T]
   def append(x: T) : Queue[T]
}
```

- Scala allows to annotate the parametrized type to control the resulting type variance

OpenSplice | DDS

Type Variance

- If S <: T is Queue[S] <: Queue[T]?
- By default Scala makes generic types nonvariant. This behaviour can be changed using the following annotations:
- Queue[+T] indicates that the the sub-typing is covariant in the parameter T
- Queue[-T] indicates that the the sub-typing is contravariant in the parameter T