DDS

Data Distribution Service

Advanced Tutorial

Gerardo Pardo-Castellote, Ph.D.

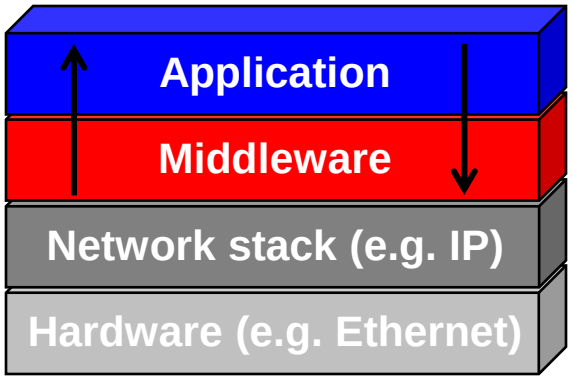
Real-Time Innovations, Inc.

http://www.rti.com

DDS Advanced Tutorial

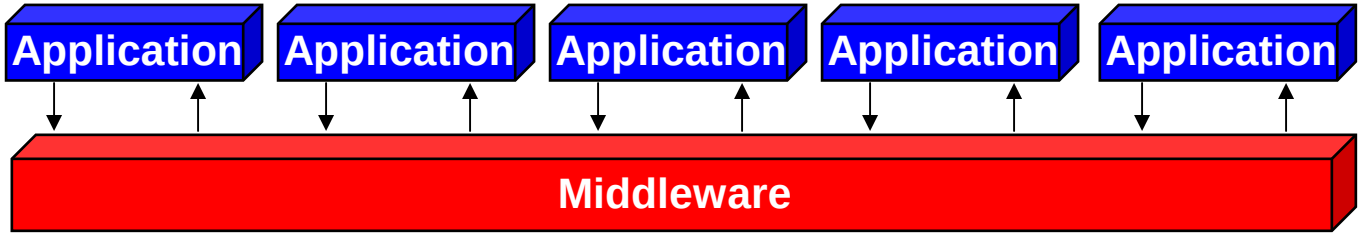
- Background
- Communication model
- Concept Demo
- DDS Entities
- Listeners, Conditions, WaitSets
- Quality of Service
- Keys and instances

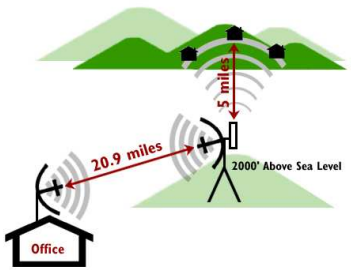
Middleware



Network middleware: A library between the operating system and the application

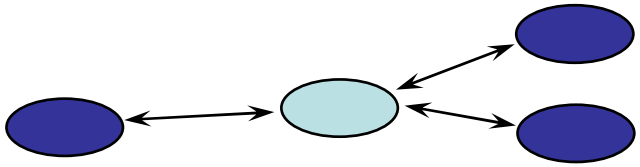
It insulates application from the raw network and provides an easier way to communicate





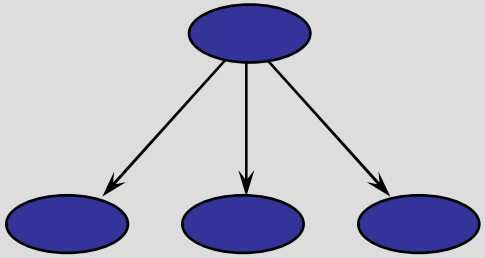
Middleware Information Models

Point-to-Point
Telephone, TCP
Simple, high-bandwidth
Leads to stove-pipe systems

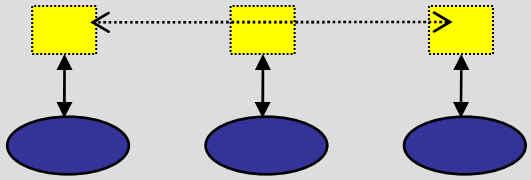


Client-Server
File systems, Database, RPC, CORBA, DCOM
Good if information is naturally centralized
Single point failure, performance bottlenecks

DDS



Publish/Subscribe Messaging
Magazines, Newspaper, TV
Excels at *many-to-many communication*
Excels at distributing *time-critical information*



Replicated Data
Libraries, Distributed databases
Excels at data-mining and analysis

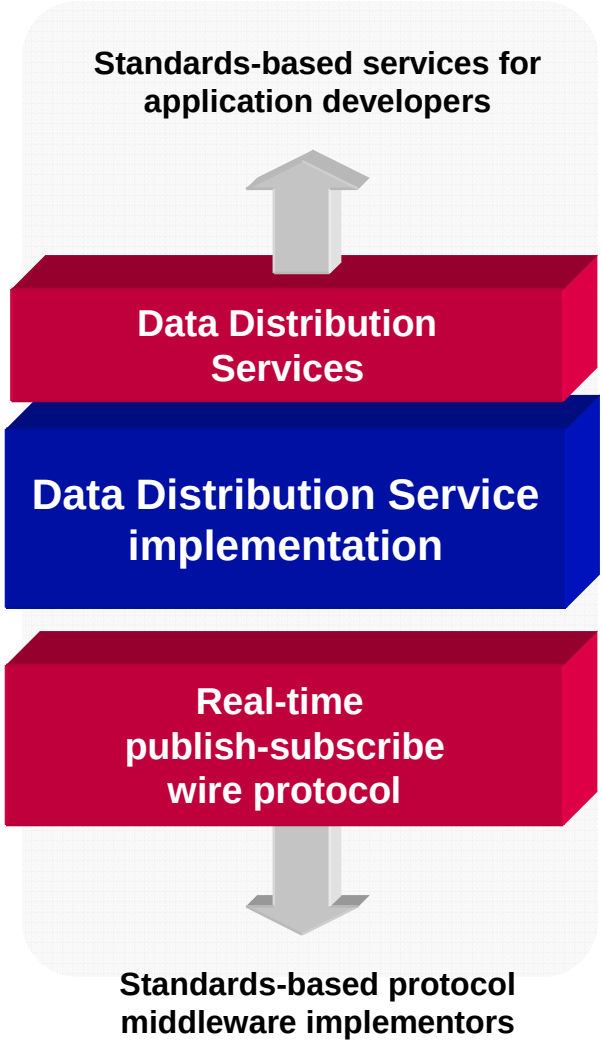
Data Distribution Service – Standard

- **Standard needed to be created to support data-critical applications**
- **DDS API specification**
  - *Finalized June 2004*
  - *Defines middleware API and services for Data-Centric Publish-Subscribe communication focused on distributed real-time systems*
  - *Version 1.2 adopted in June 2006*
- **DDS Interoperability Protocol**
  - *Approved June 2006*
  - *Defines wire protocol uses by DDS implementations to communicate with each other*

DDS Interoperability: Wire Protocol

DDS-RTPS Interoperability Wire Protocol

- RTPS = Real-Time Publish Subscribe
- Joint submission of RTI and THALES
- Adopted by OMG in June 2006
- Protocol tailored to needs of DDS systems
  - ***Supports unreliable transports, multicast, message fragmentation, efficient filtering at the source, etc.***
- Extensible -> Will support extensions while remaining interoperable



OMG Middleware standards

CORBA

Distributed object

- Client/server
- Remote method calls
- Reliable transport

Best for

- Remote command processing
- File transfer
- Synchronous transactions

DDS

Distributed data

- Publish/subscribe
- Multicast data
- Configurable QoS

Best for

- Quick dissemination to many nodes
- Dynamic nets
- Flexible delivery requirements

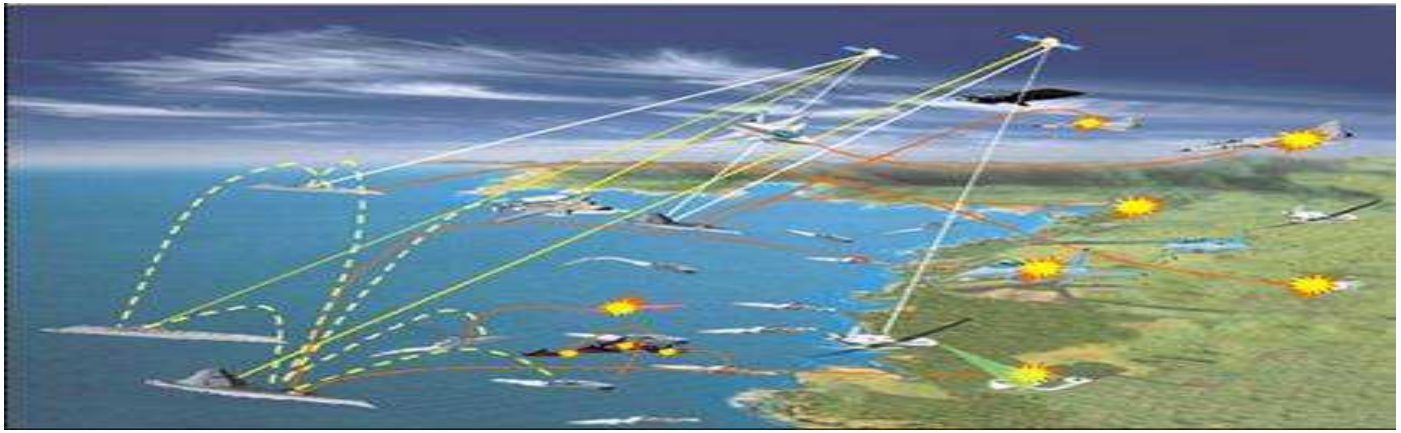
DDS and CORBA address different needs

Complex systems often need both…

The net-centric vision

- *Vision for "net-centric applications"*
- *Total access to information for real-time applications*
- *This vision is enabled by the internet and related network technologies*
- *Challenge:*

  "Provide the right information at the right place at the right time… no matter what."



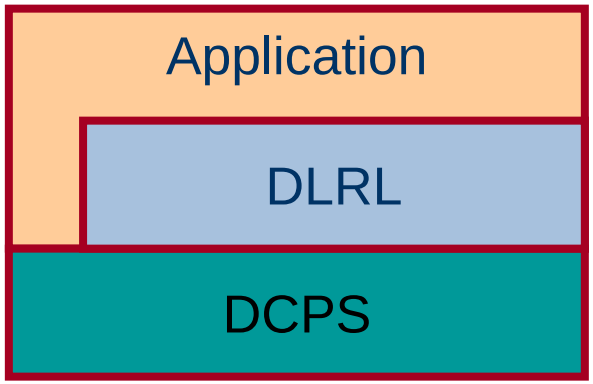
What is DDS

DCPS = Data Centric Publish_Subscribe

- Purpose: Distribute the data

DLRL = Data Local Reconstruction Layer

- Purpose: provide an object-based model to access data ‘as if’ it was local



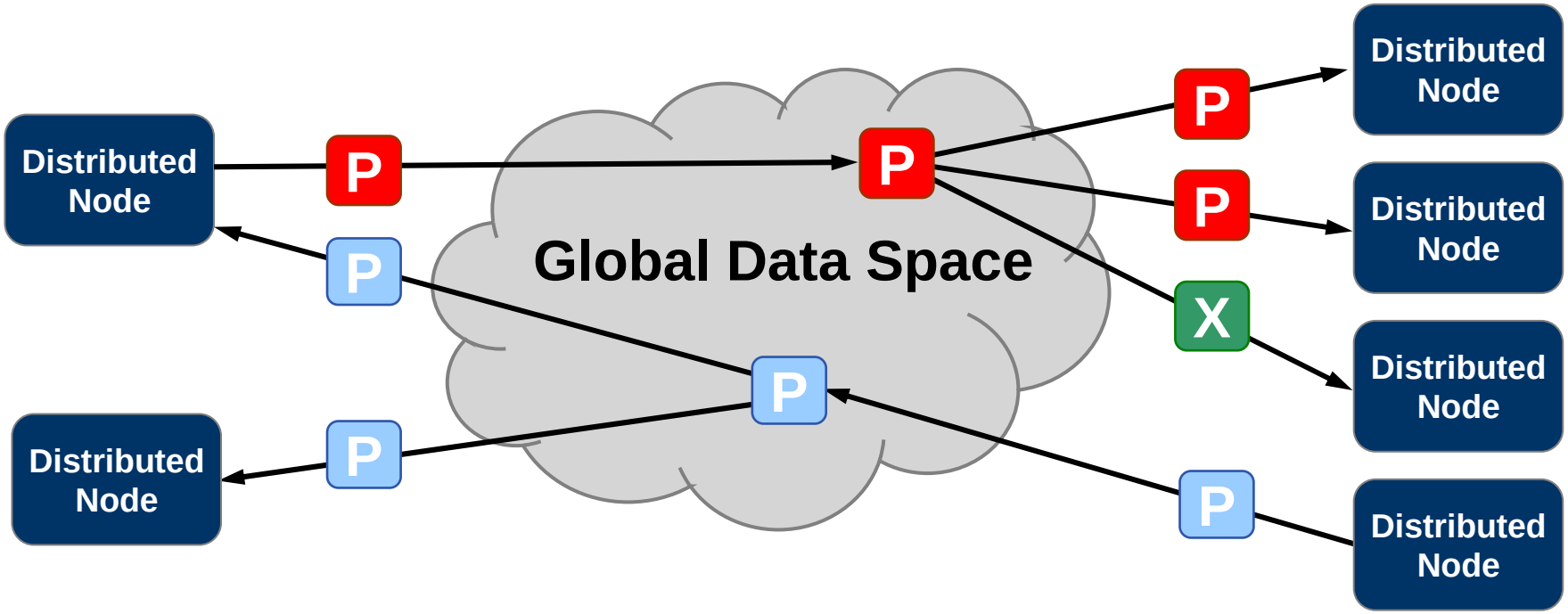
DDS Advanced Tutorial

- Background
- → Communication model
- Concept Demo
- DDS Entities
- Listeners, Conditions, WaitSets
- Quality of Service
- Keys and instances

DDS

Provides a "Global Data Space" that is accessible to all interested applications.

- Data objects addressed by ***domainId***, ***Topic*** and ***Key***
- Subscriptions are ***decoupled*** from Publications
- Contracts established by means of ***QoS***
- Automatic ***discovery*** and configuration

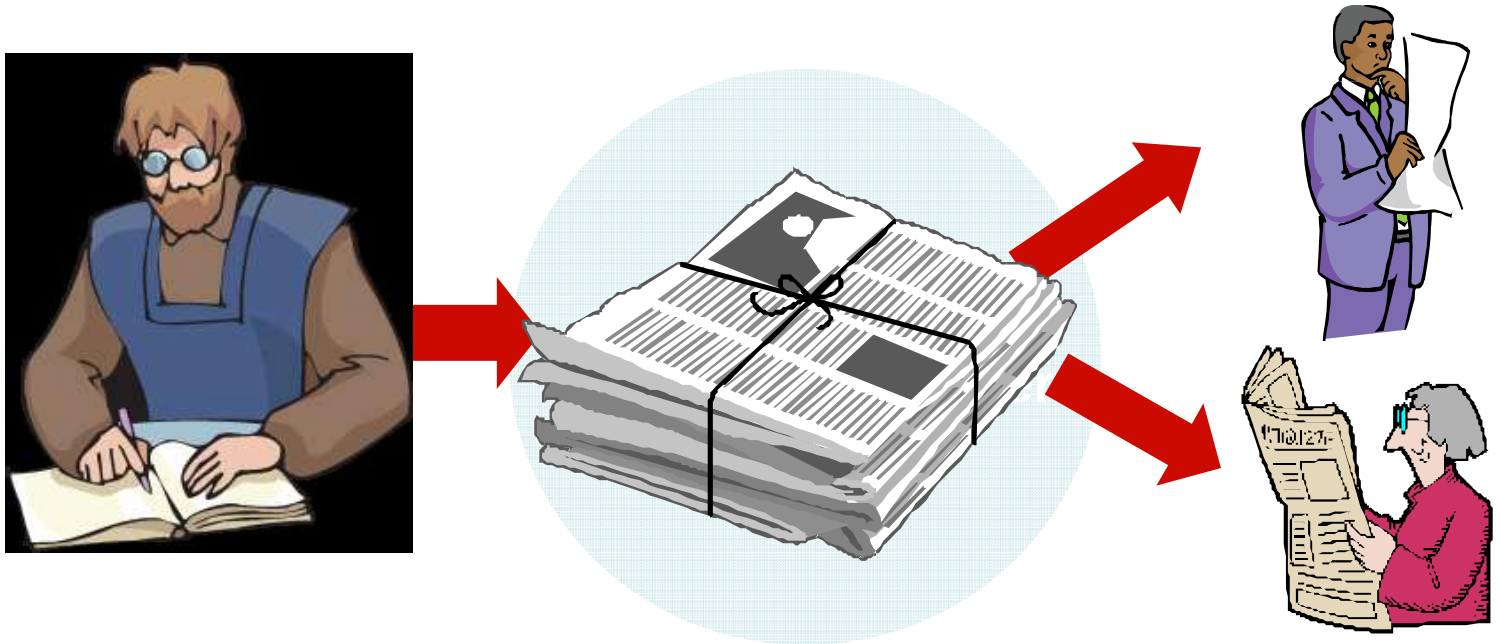


Publish Subscribe Model

- ***Efficient mechanism for data communications***

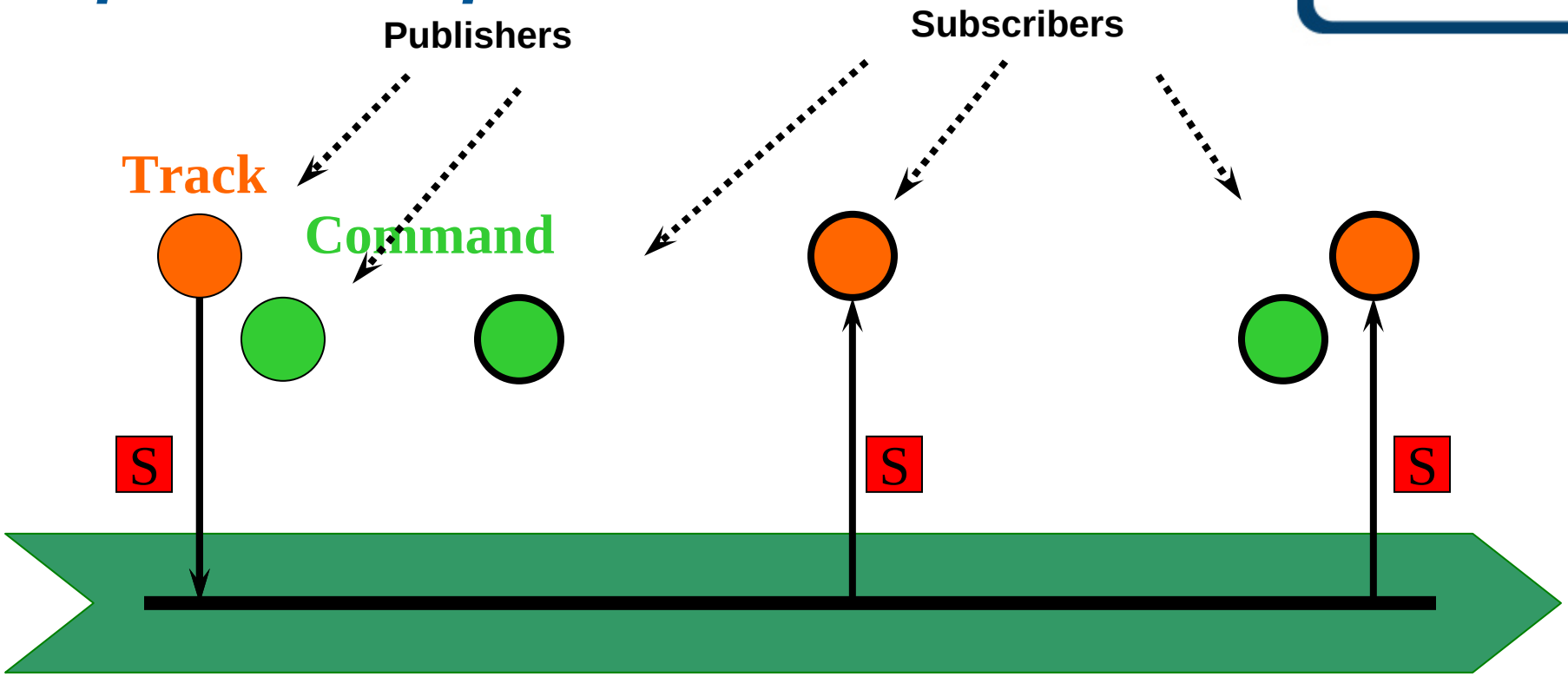
Reporter does not need to know where subscribers live.

Subscribers do not need to know where reporter lives.

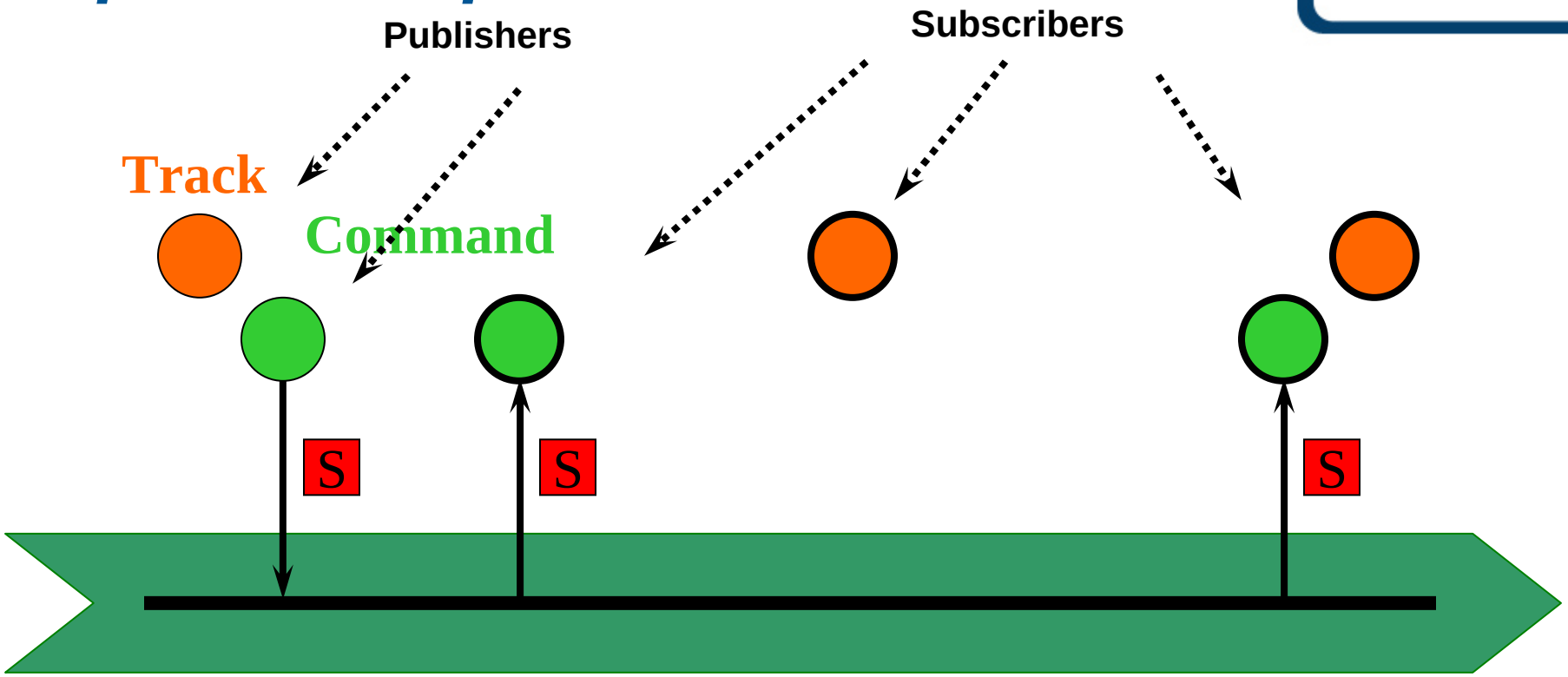


Data Producer **Middleware** **Consumers**

Topic-based publish-subscribe



Topic-based publish-subscribe



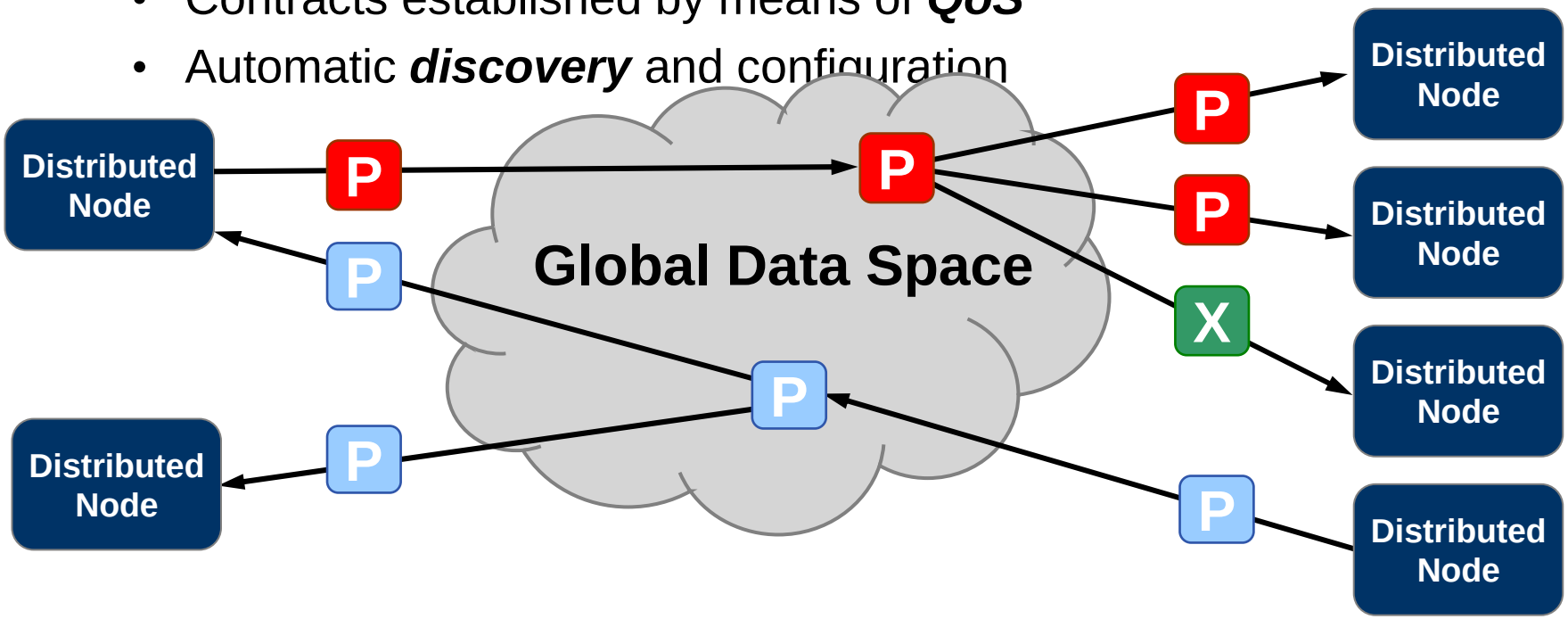
Publish-subscribe allows infrastructure to prepare itself…

… Such that when the data is written it is directly sent to the subscribers

DDS

Provides a "Global Data Space" that is accessible to all interested applications.

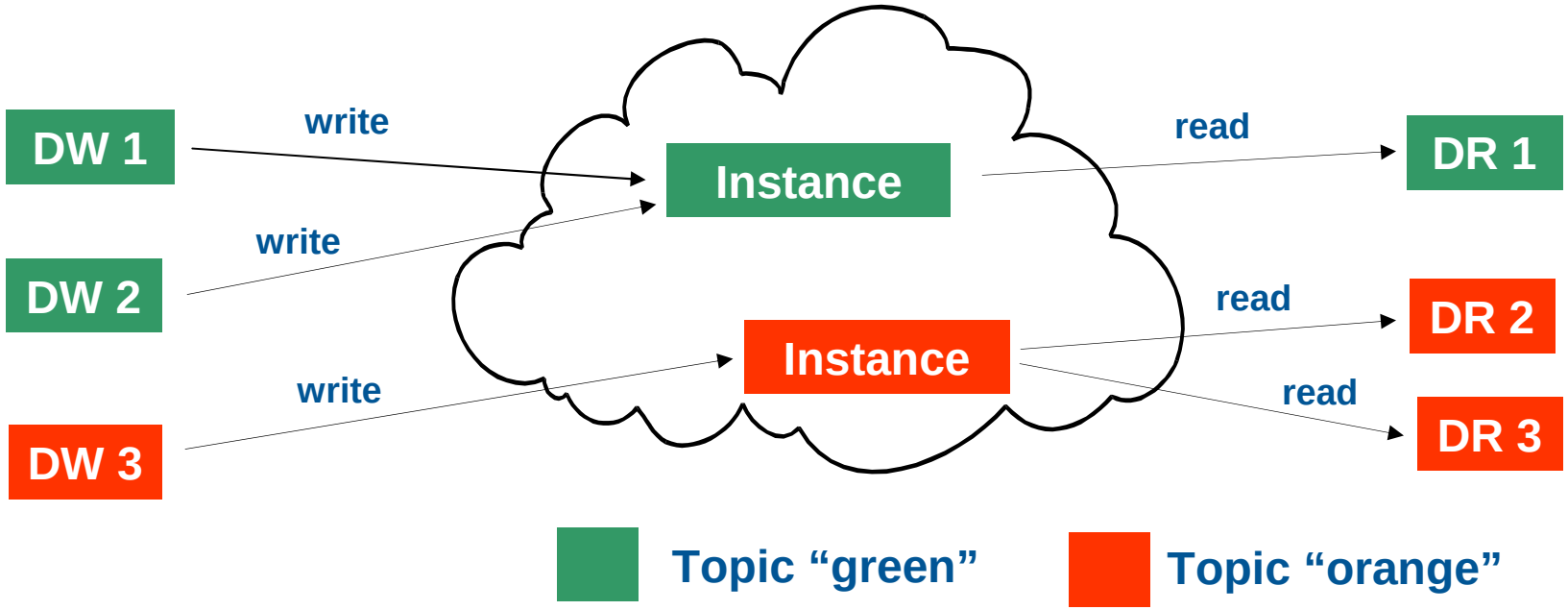
- Data objects addressed by ***domainId***, ***Topic*** and ***Key***
- Subscriptions are ***decoupled*** from Publications
- Contracts established by means of ***QoS***
- Automatic ***discovery*** and configuration



Example without keys

When not using keys:

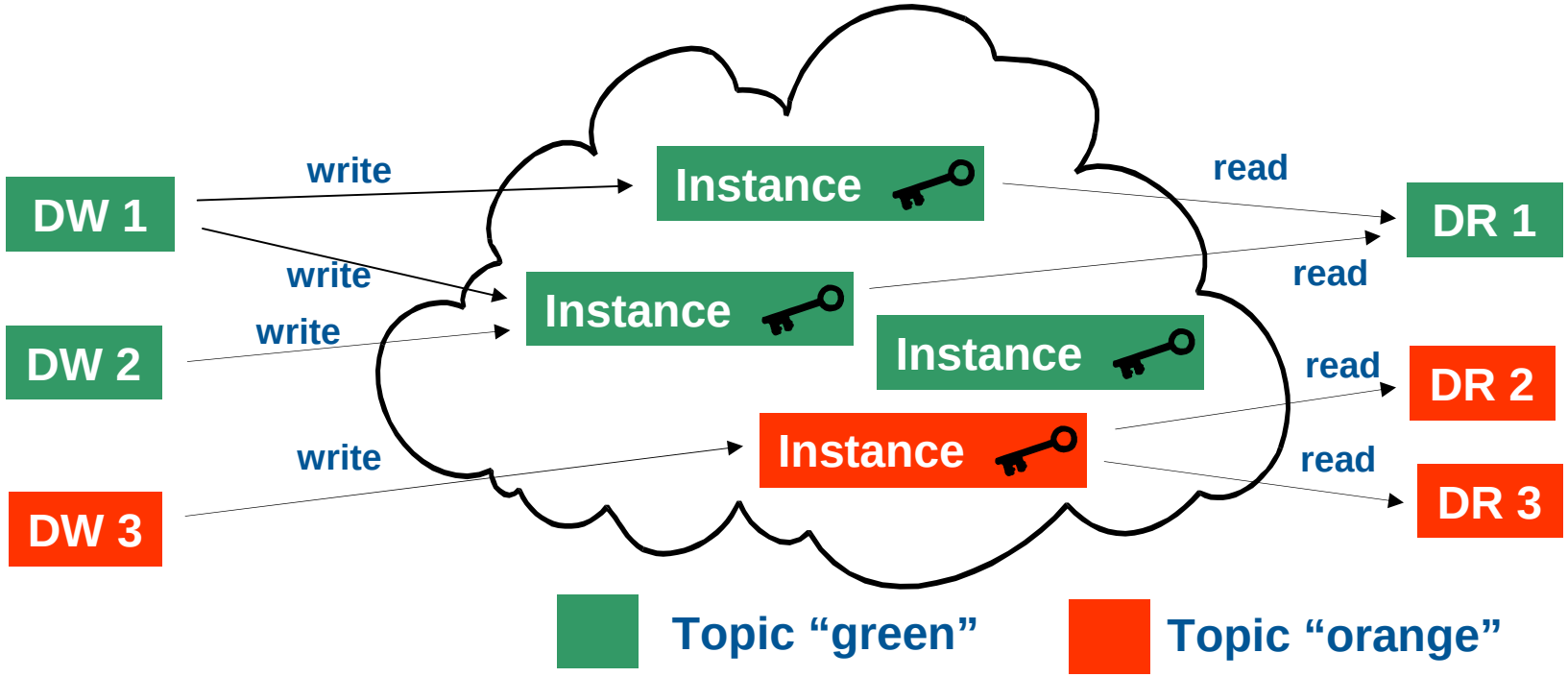
- Each topic corresponds to a *single* data instance.
- A DataWriter associated with a topic can write to the instance corresponding to that topic.
- Multiple DataWriters may write to the same instance.
- A DataReader specifies the topic (instance) it wants to receive updates from.



Example with keys

Address in Global Data Space = (domain_id, Topic, Key)

- Each topic corresponds to a *multiple* data instances
- Each DataWriter can write to multiple instances of a *single* topic
- Multiple DataWriters may write to the same instance
- Each DataReader can receive updates from *multiple* instances of a *single* topic
- Multiple DRs may read from the same instances

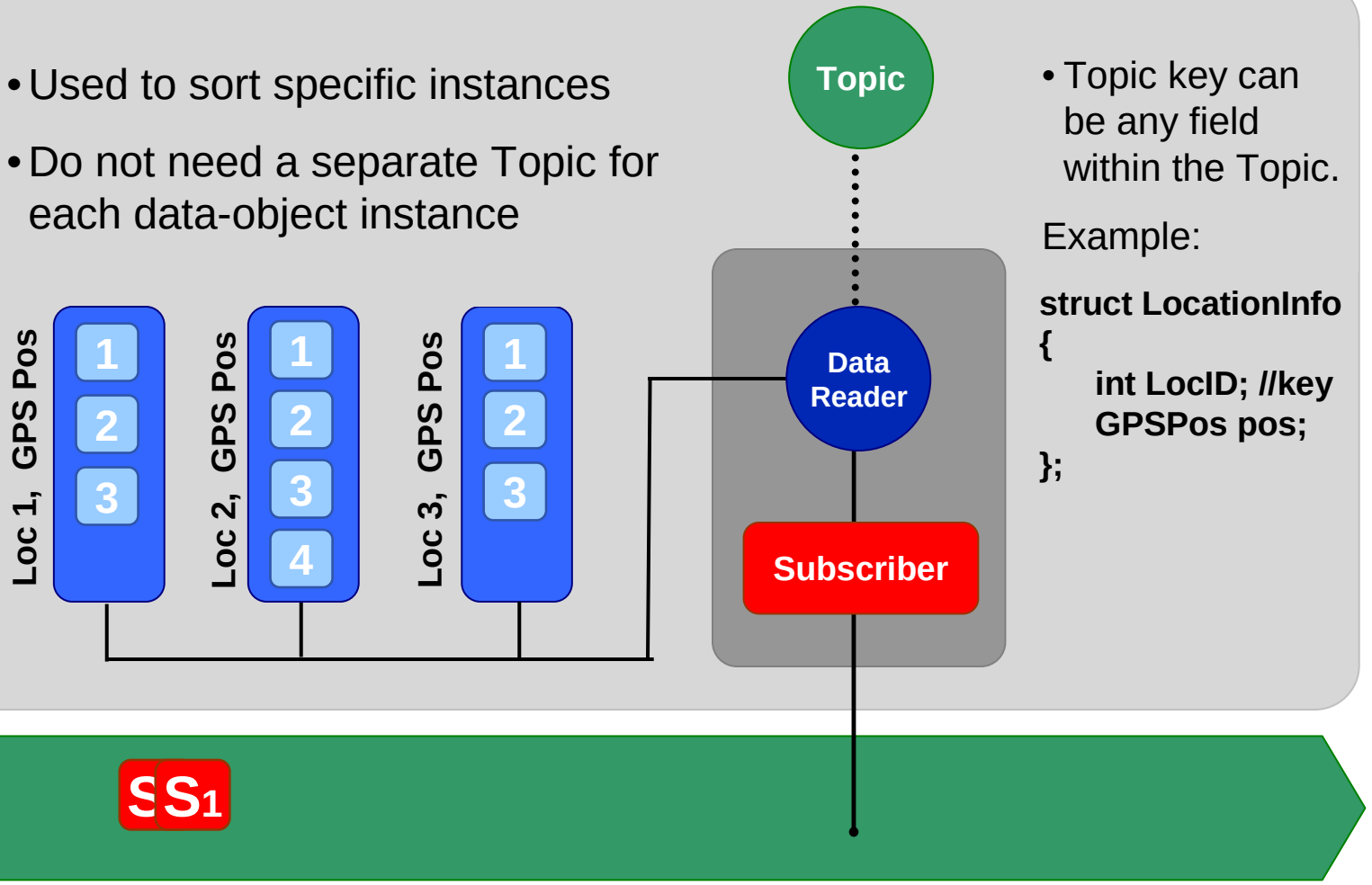


Data object addressing: Keys

Address in Global Data Space = (domain_id, Topic, Key)

Multiple instances of the same topic

- Used to sort specific instances
- Do not need a separate Topic for each data-object instance



- Topic key can be any field within the Topic.

Example:

```
struct LocationInfo
{
    int LocID; //key
    GPSPos pos;
};
```

DDS Advanced Tutorial

Background

Communication model

→ Concept Demo

DDS Entities

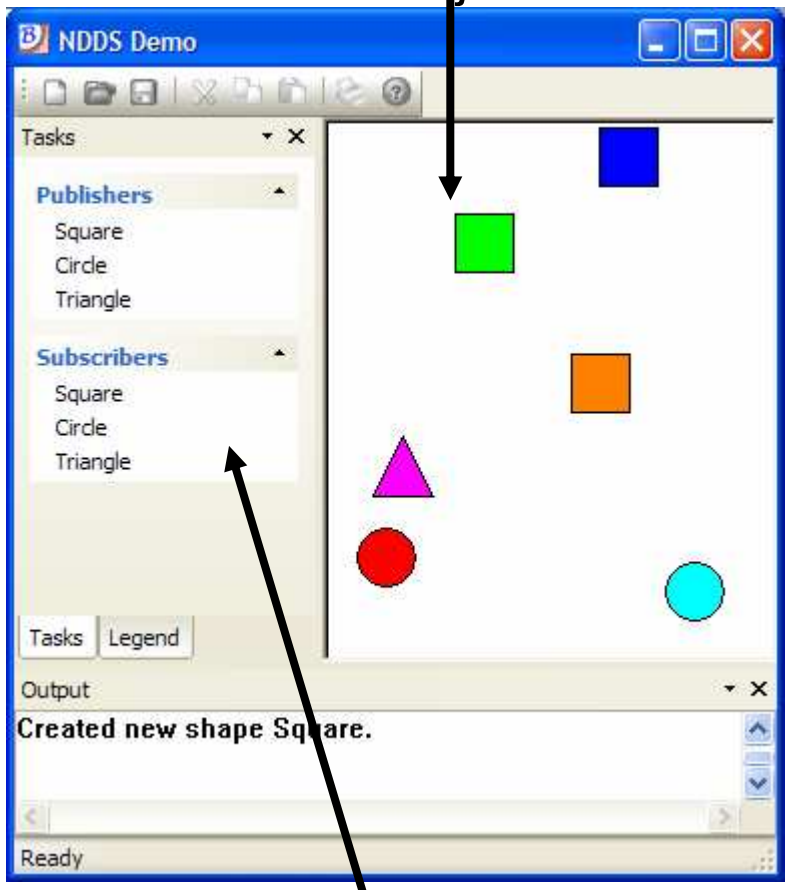
Listeners, Conditions, WaitSets

Quality of Service

Keys and instances

Demo GUI

Display Area:
Shows state of objects



Control Area:
Allows selection of objects and QoS

Topics

- Square, Circle, Triangle
- Attributes

IDL data types

- Shape (color, x, y, size)
  - ***Color is instance Key***
- Attributes
  - ***Shape & color used for key***

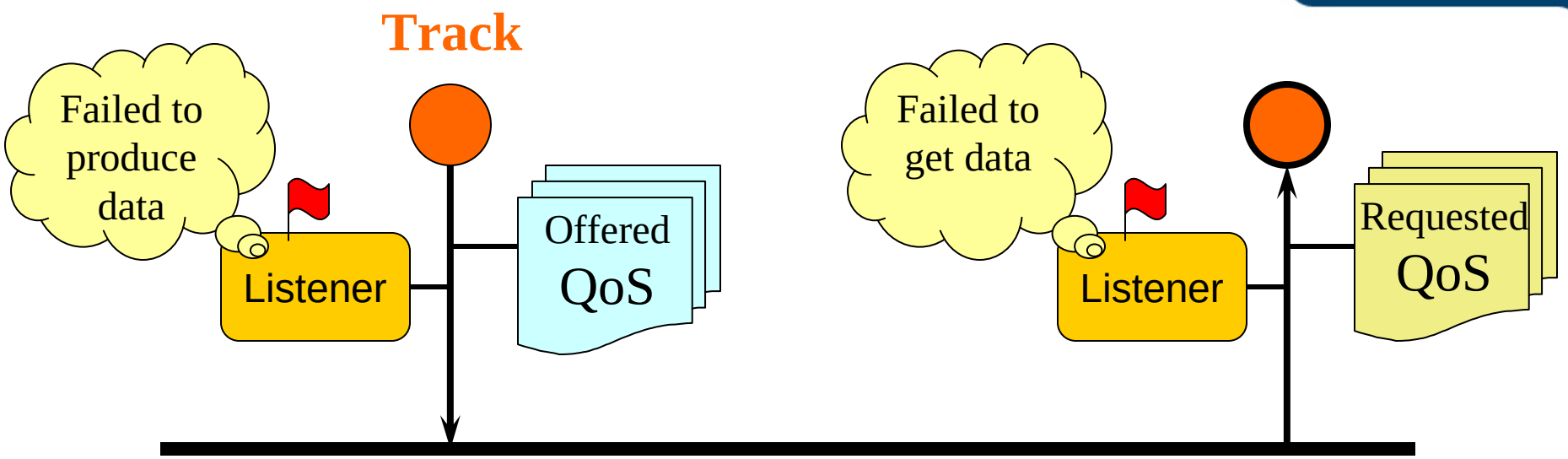
QoS

- Deadline, Liveliness
- Reliability, Durability
- History, Partition
- Ownership

DDS Advanced Tutorial

- Background
- Communication model
- Concept Demo
- → DDS Entities
- Listeners, Conditions, WaitSets
- Quality of Service
- Keys and instances

DDS communications model



Publisher declares information it has and specifies the Topic

- … and the offered QoS contract
- … and an associated listener to be alerted of any significant status changes

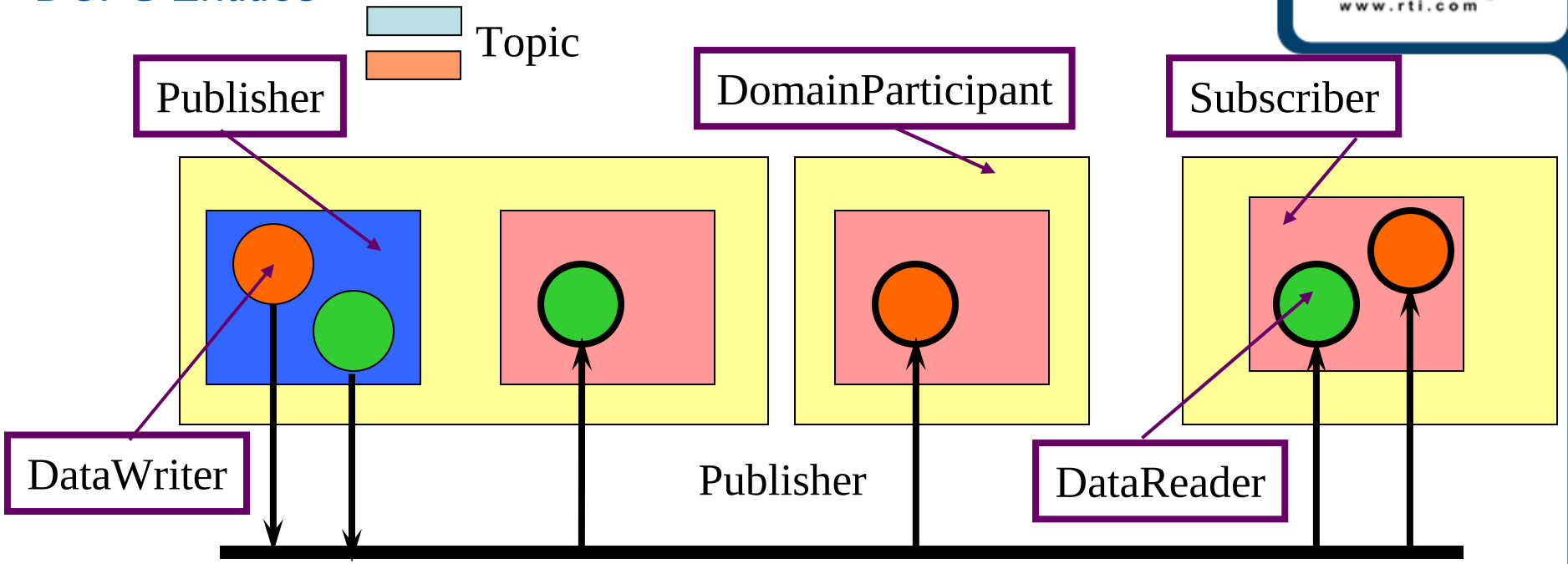
Subscriber declares information it wants and specifies the Topic

- … and the requested QoS contract
- … and an associated listener to be alerted of any significant status changes

DDS automatically discovers publishers and subscribers

- DDS ensures QoS matching and alerts of inconsistencies

DCPS Entities



DomainParticipant ~ ***Represents participation of the application in the communication collective***

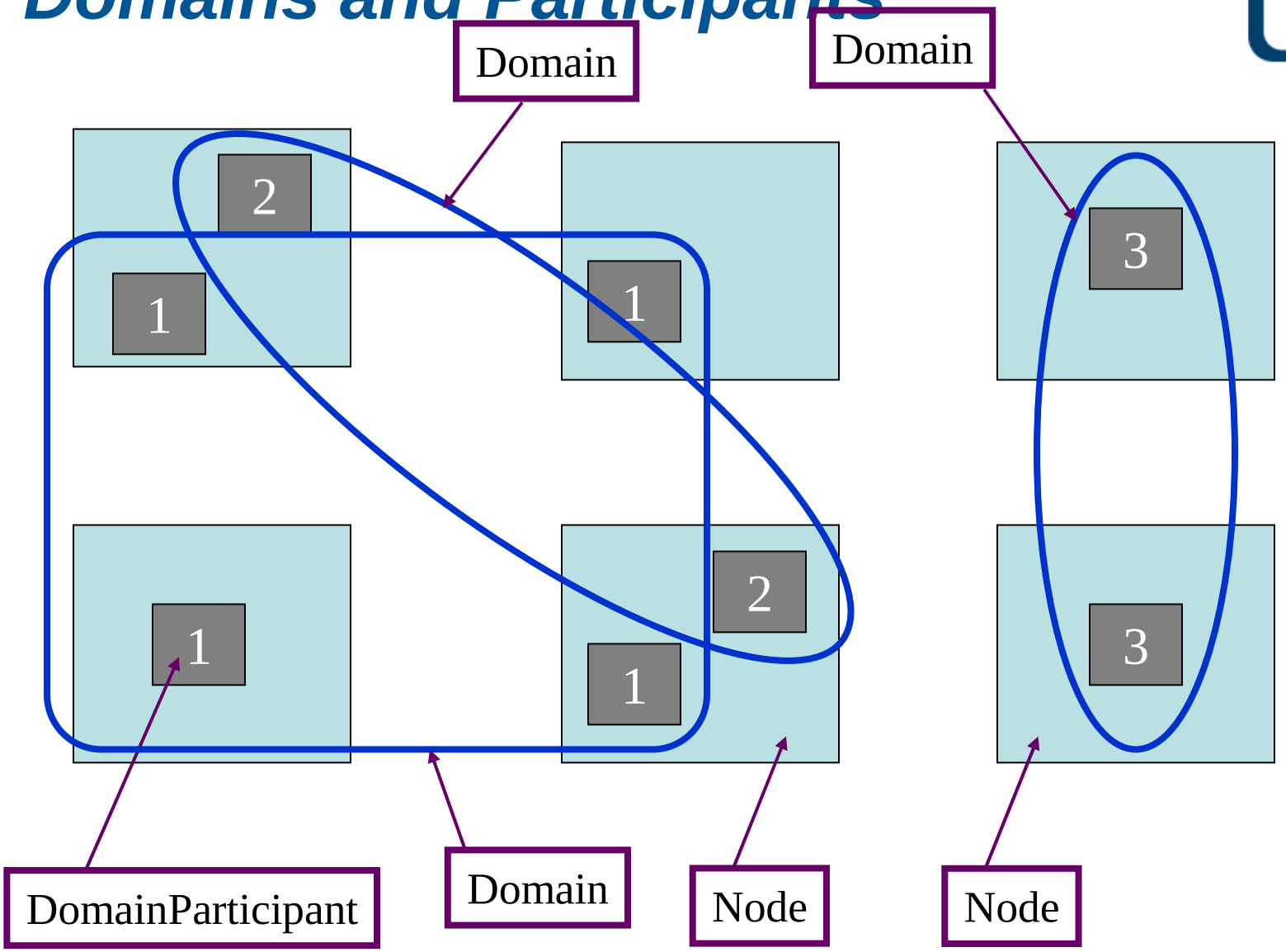
DataWriter ~ ***Accessor to write typed data on a particular*** *Topic*

Publisher ~ ***Aggregation of DataWriter objects. Responsible for disseminating information.***

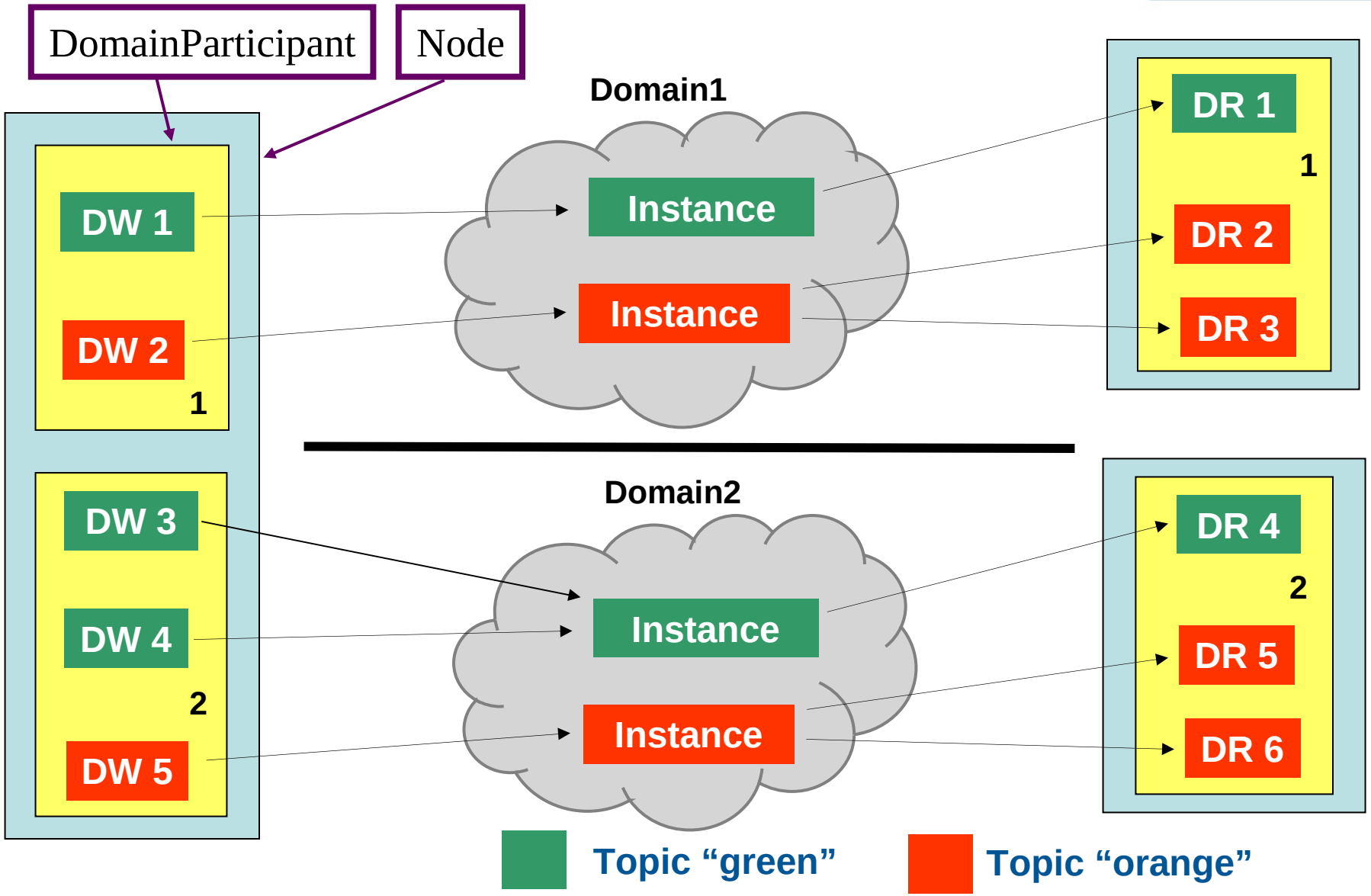
DataReader ~ ***Accessor to read typed data regarding a specific*** *Topic*

Subscriber ~ ***Aggregation of DataReader objects. Responsible for receiving information***

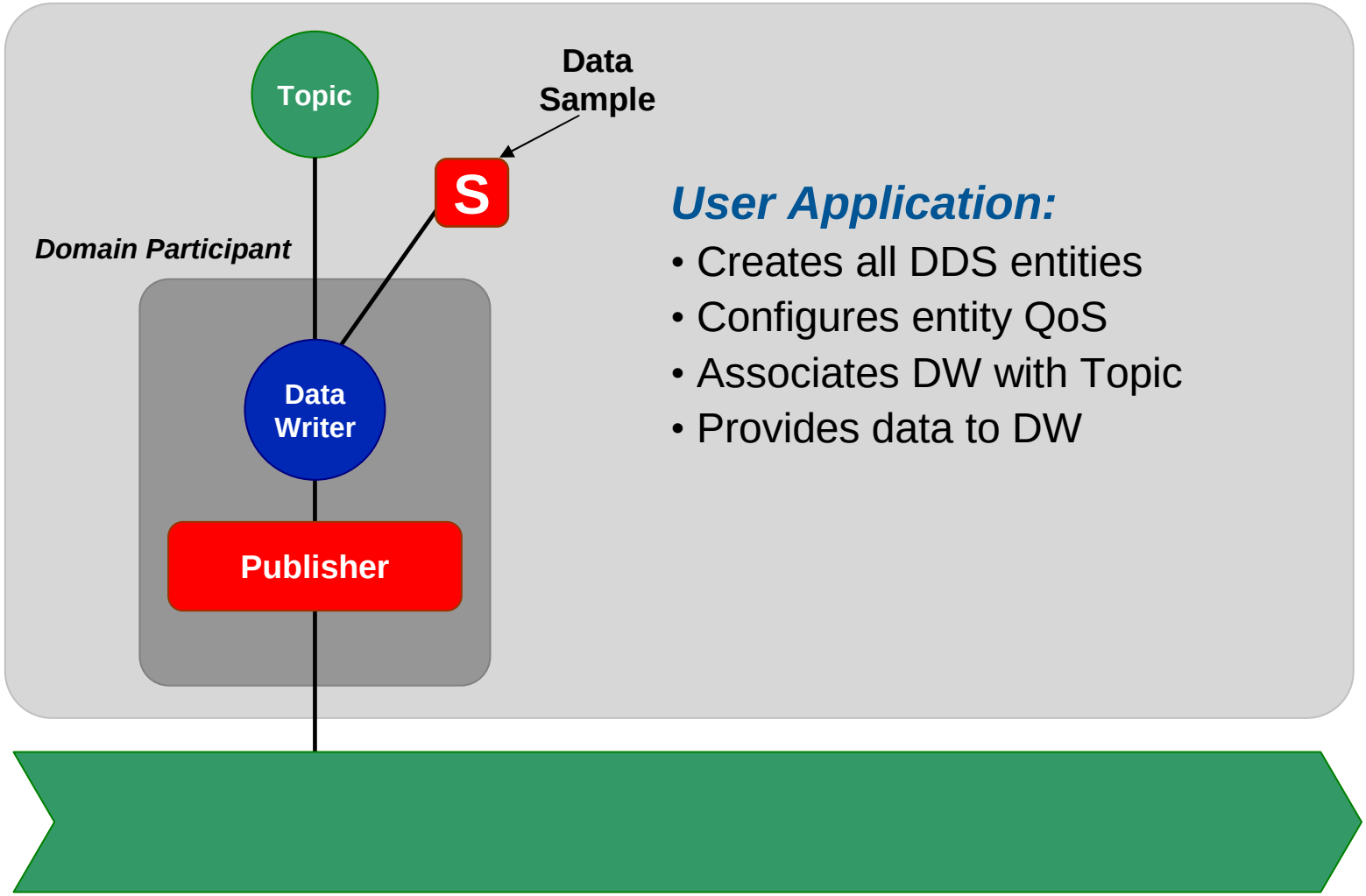
Domains and Participants



DDS



DDS Publication



User Application:

- Creates all DDS entities
- Configures entity QoS
- Associates DW with Topic
- Provides data to DW

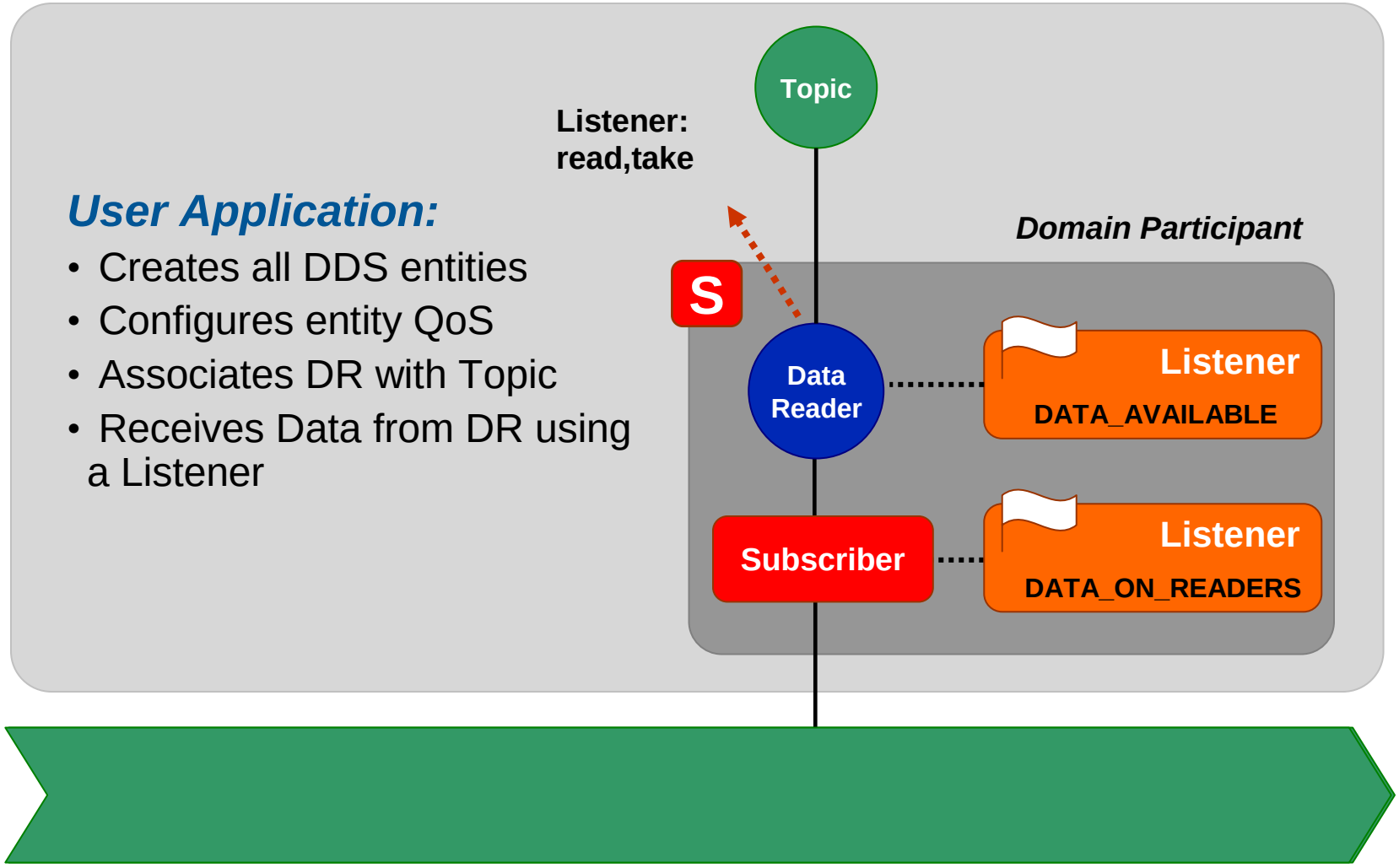
Example: Publication

```
Publisher publisher = domain->create_publisher(
        publisher_qos,
        publisher_listener);

Topic topic = domain->create_topic(
        “Track”, “TrackStruct”,
        topic_qos, topic_listener);

DataWriter writer = publisher->create_datawriter(
        topic, writer_qos, writer_listener);
TrackStructDataWriter twriter =
      TrackStructDataWriter::narrow(writer);

TrackStruct my_track;
twriter->write(&my_track);
```

DDS Subscription Listener

User Application:

- Creates all DDS entities
- Configures entity QoS
- Associates DR with Topic
- Receives Data from DR using a Listener



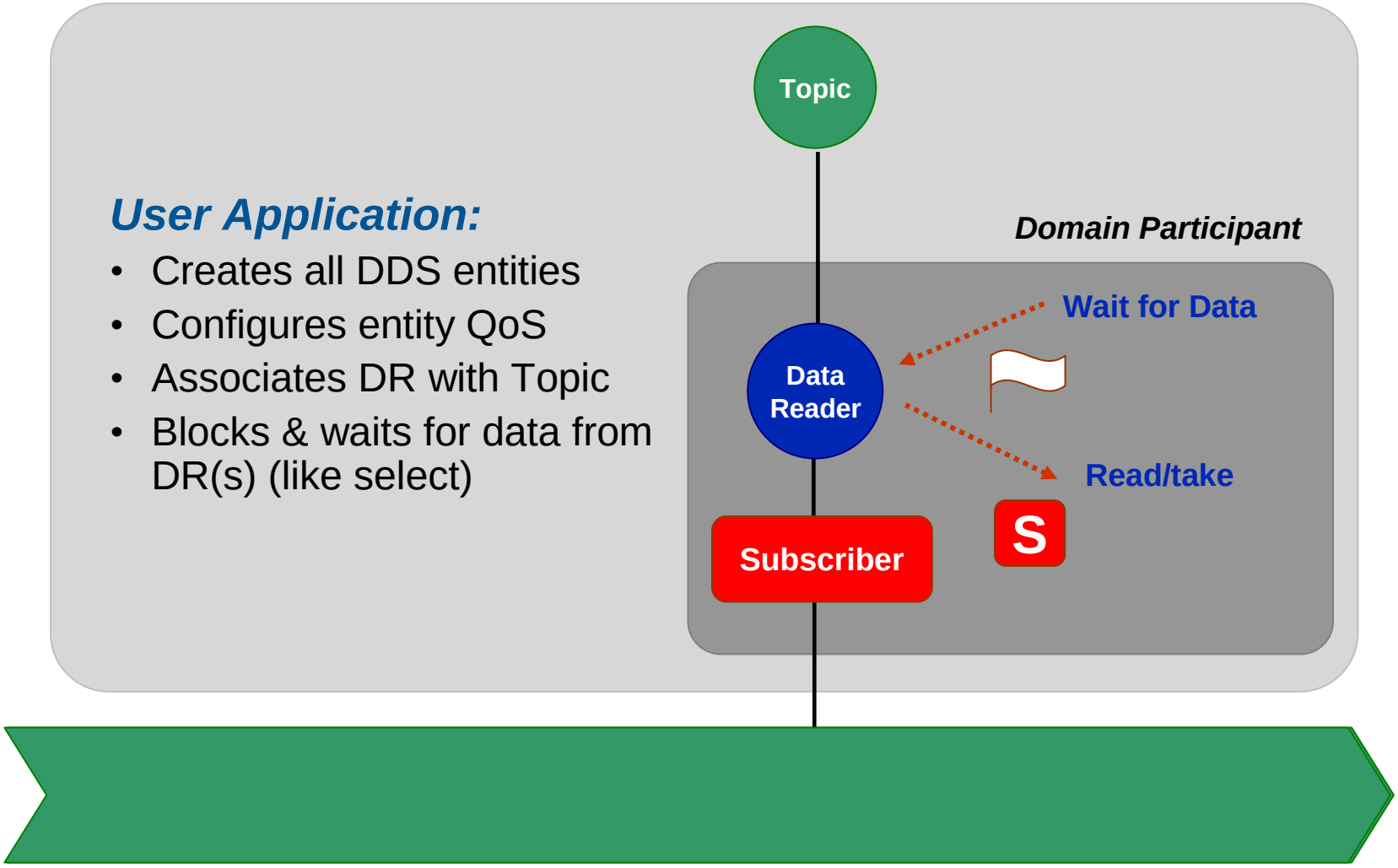
Example: Subscription

```
Subscriber  subs = domain->create_subscriber(
       subscriber_qos, subscriber_listener);

Topic topic = domain->create_topic(
       “Track”, “TrackStruct”,
       topic_qos, topic_listener);

DataReader reader = subscriber->create_datareader(
       topic, reader_qos, reader_listener);

// Use listener-based or wait-based access
```

How to get data (listener-based)

```
Listener listener = new MyListener();
reader->set_listener(listener);

MyListener::on_data_available( DataReader reader )
{
    TrackStructSeq received_data;
    SampleInfoSeq sample_info;
    TrackStructDataReader treader =
        TrackStructDataReader::narrow(reader);

    treader->take( received_data,
                   sample_info, …)

    // Use received_data
}
```

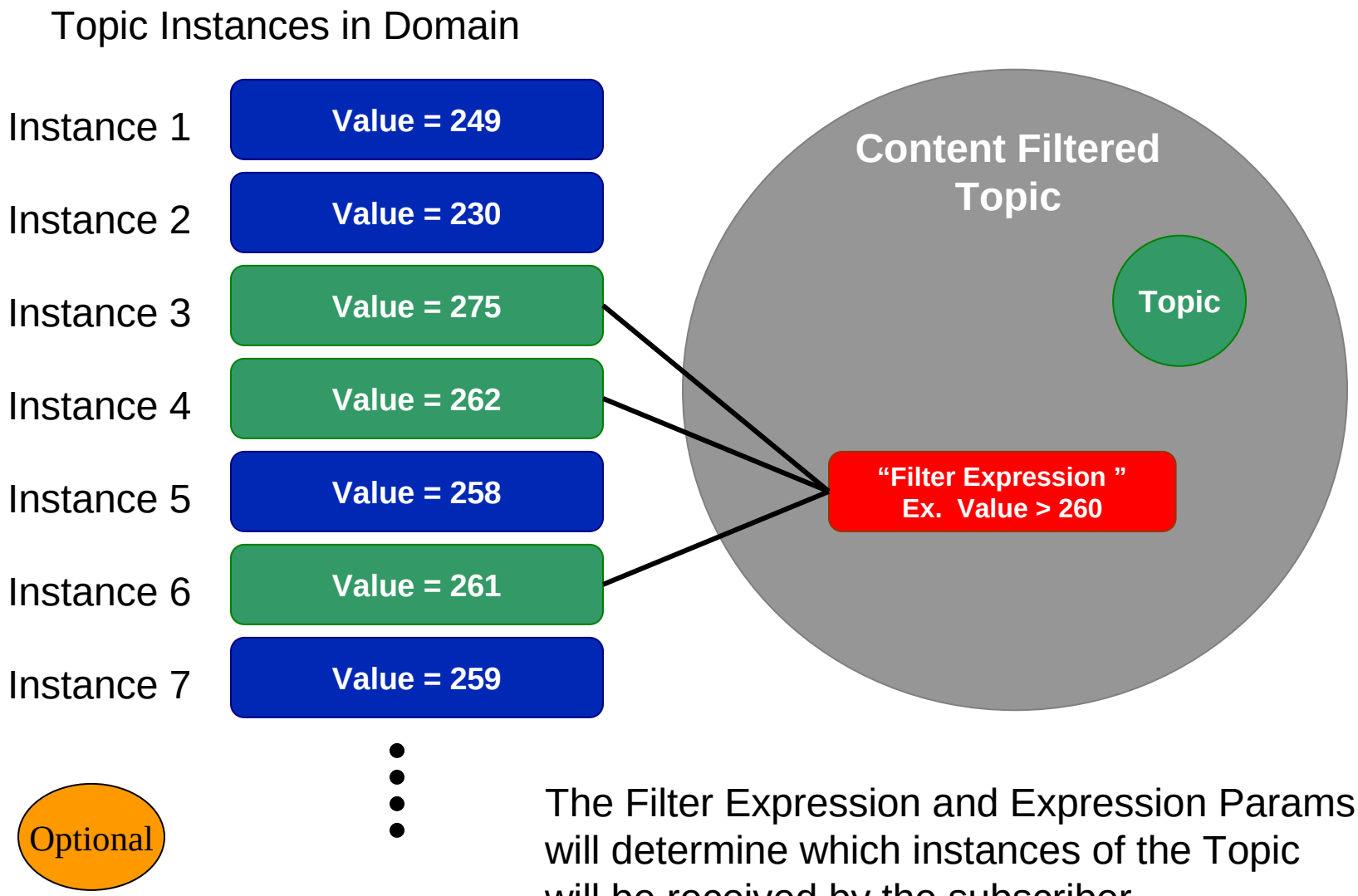
DDS Subscription Wait-Set



How to get data (wait-based)

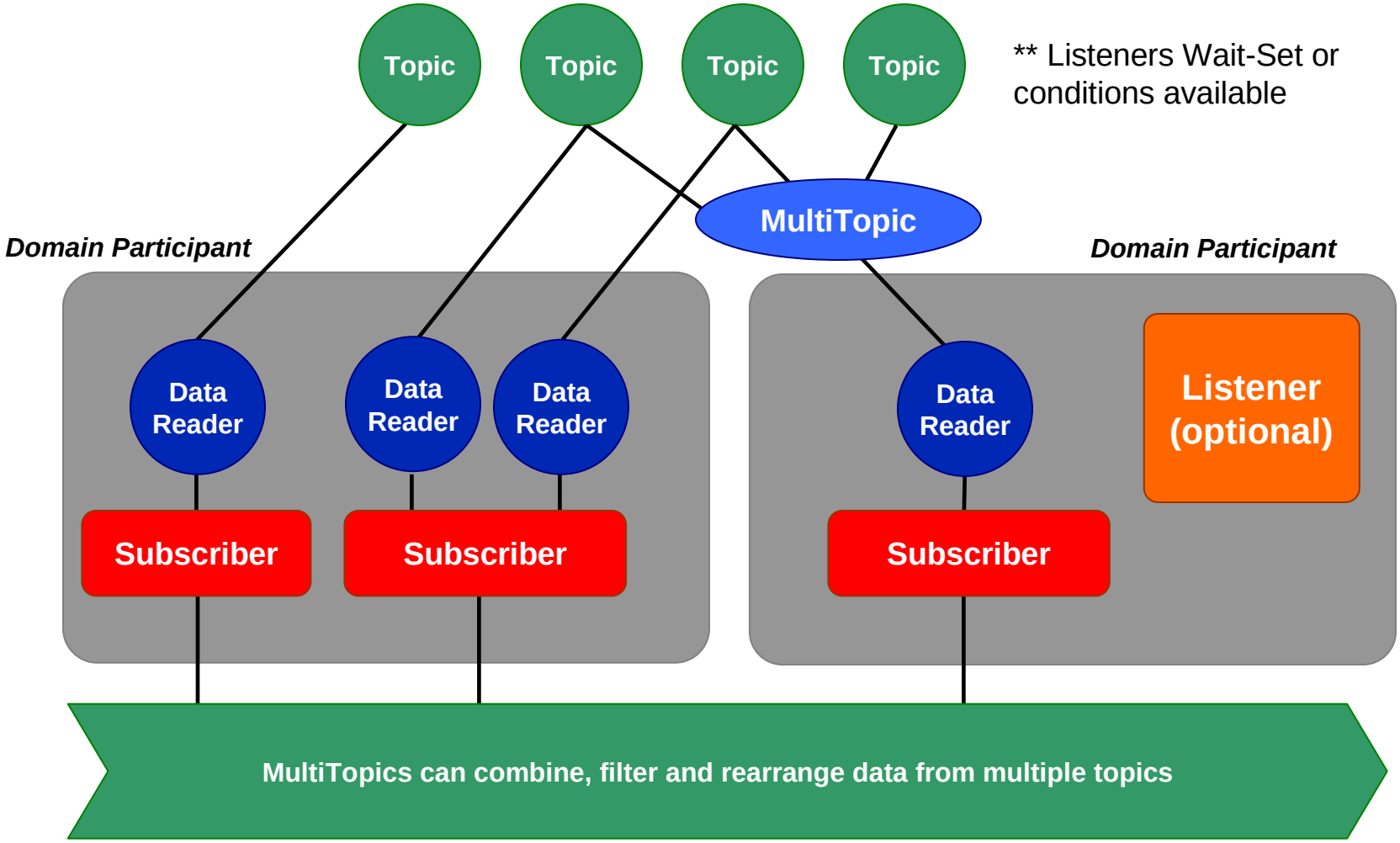
```
Condition  foo_condition =
   treader->create_readcondition(…);

waitset->add_condition(foo_condition);

ConditionSeq active_conditions;
waitset->wait(active_conditions, timeout);
…
FooSeq received_data;
SampleInfoSeq sample_info;

treader->take_w_condition(received_data,
                          sample_info,
                          foo_condition);
// Use received_data
```

DDS Content Filtered Topics



The Filter Expression and Expression Params will determine which instances of the Topic will be received by the subscriber.

DDS Subscription Objects (MultiTopic)



DDS Advanced Tutorial

Background

Communication model

Concept Demo

DDS Entities

→ Listeners, Conditions, WaitSets

Quality of Service

Keys and instances

Listeners, Conditions & WaitSets

Middleware must notify user application of relevant events

- Arrival of data
- QoS violations
- Discovery of relevant entities
- These events may be detected asynchronously by the middleware
- … Same issue arises with POSIX signals…

DDS allows the application a choice:

- Either get notified asynchronously using a Listener
- Or wait synchronously using a WaitSet

Both approaches are unified using STATUS changes

Status changes

DDS defines

- A set of enumerated STATUS
- The statuses relevant to each kind of DDS Entity

STATUS	Entity
INCONSISTENT_TOPIC	Topic
DATA_ON_READERS	Subscriber
LIVELINESS_CHANGED	DataReader
REQUESTED_DEADLINE_MISSED	DataReader
RUQESTED_INCOMPATIBLE_QOS	DataReader
DATA_AVAILABLE	DataReader
SAMPLE_LOST	DataReader
SAMPLE_REJECTED	DataReader
SUBSCRIPTION_MATCHED	DataReader
LIVELINESS_LOST	DataWriter
OFFERED_INCOMPATIBLE_QOS	DataWriter
OFFERED_DEADLINE_MISSED	DataWriter
PUBLICATION_MATCHED	DataWriter

A DDS entity maintains a value for each STATUS

```
struct LivelinessChangedStatus {
    long alive_count;
    long not_alive_count;
    long alive_count_change;
    long not_alive_count_change;
}
```

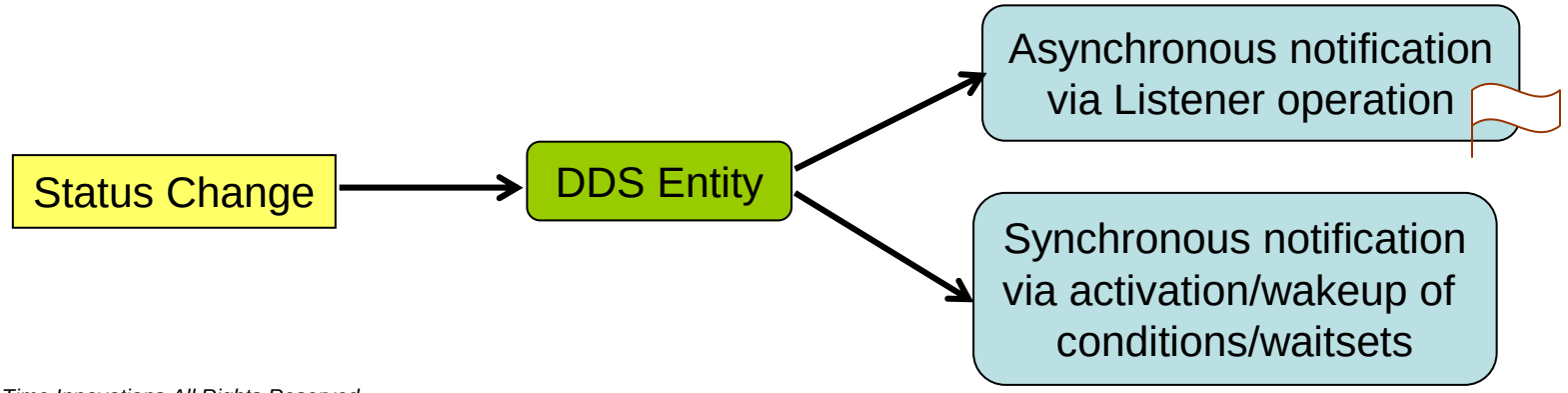
Listeners & Condition duality

A StatusCondition can be selectively activated to respond to any subset of the statuses

An application can wait changes in sets of Status Conditions using a WaitSet

Each time the value of a STATUS changes DDS

- Calls the corresponding Listener operation
- Wakes up any threads waiting on a related status change



Listeners, Conditions and Statuses

A DDS Entity is associated with

- A listener of the proper kind (if activated)
- A StatusCondition (if activated)

The Listener for an Entity has a separate operation for each of the relevant statuses

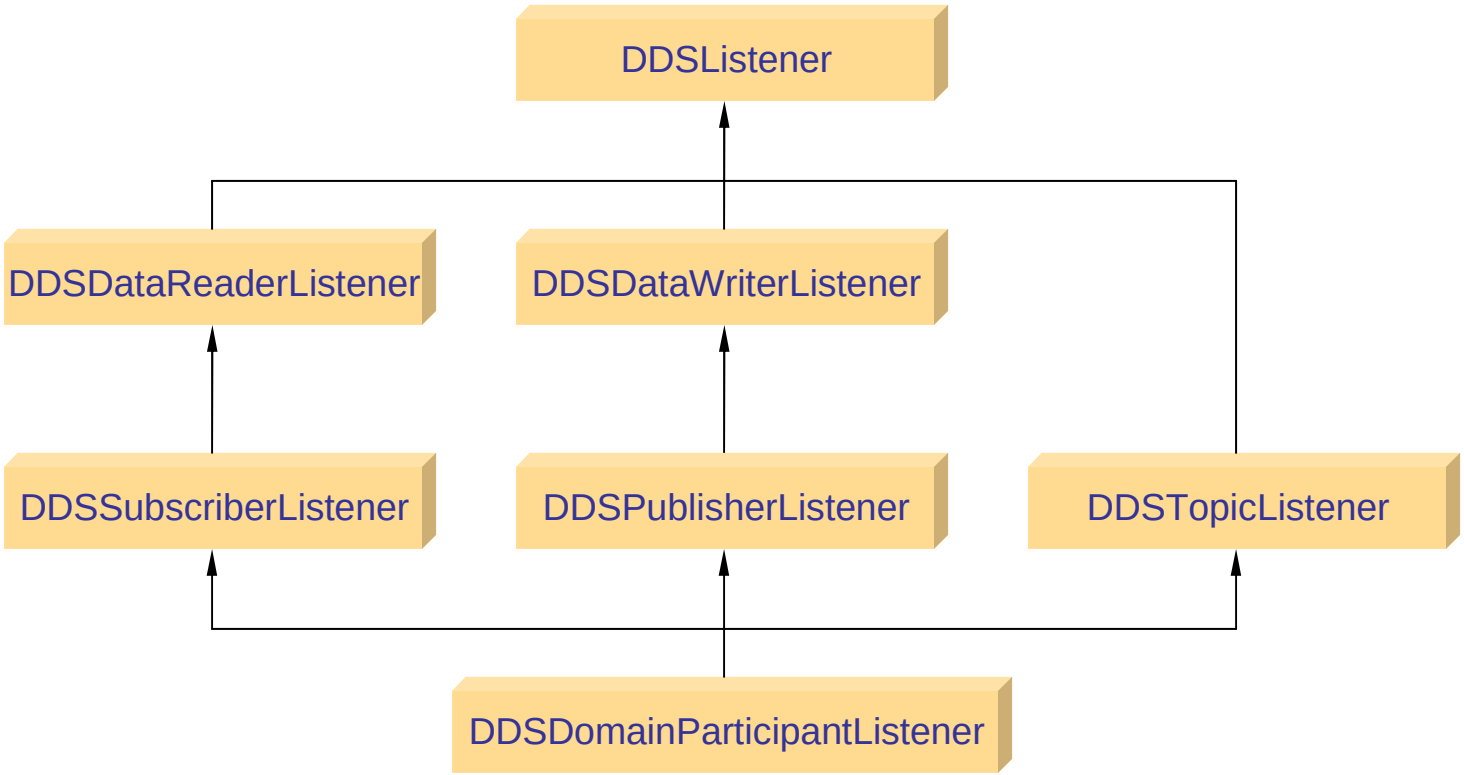
STATUS	Entity	Listener operation
INCONSISTENT_TOPIC	Topic	on_inconsistent_topic
DATA_ON_READERS	Subscriber	on_data_on_readers
LIVELINESS_CHANGED	DataReader	on_liveliness_changed
REQUESTED_DEADLINE_MISSED	DataReader	on_requested_deadline_missed
RUQESTED_INCOMPATIBLE_QOS	DataReader	on_requested_incompatible_qos
DATA_AVAILABLE	DataReader	on_data_available
SAMPLE_LOST	DataReader	on_sample_lost
SAMPLE_REJECTED	DataReader	on_sample_rejected
SUBSCRIPTION_MATCHED	DataReader	on_subscription_matched
LIVELINESS_LOST	DataWriter	on_liveliness_lost
OFFERED_INCOMPATIBLE_QOS	DataWriter	on_offered_incompatible_qos
OFFERED_DEADLINE_MISSED	DataWriter	on_offered_deadline_missed
PUBLICATION_MATCHED	DataWriter	on_publication_matched

DDS Middleware Listeners

DDS provides listeners to monitor delivery-related events and to notify application when these events occur.

- Domain Participant
- Topic
- Publisher
- Data Writer
- Subscriber
- Data Reader

Inheritance Diagram



DDS Middleware Listener Steps

Step 1: Derive from DDS Listener

```
class MyDataListener : public DDSDataReaderListener
{ etc. }
```

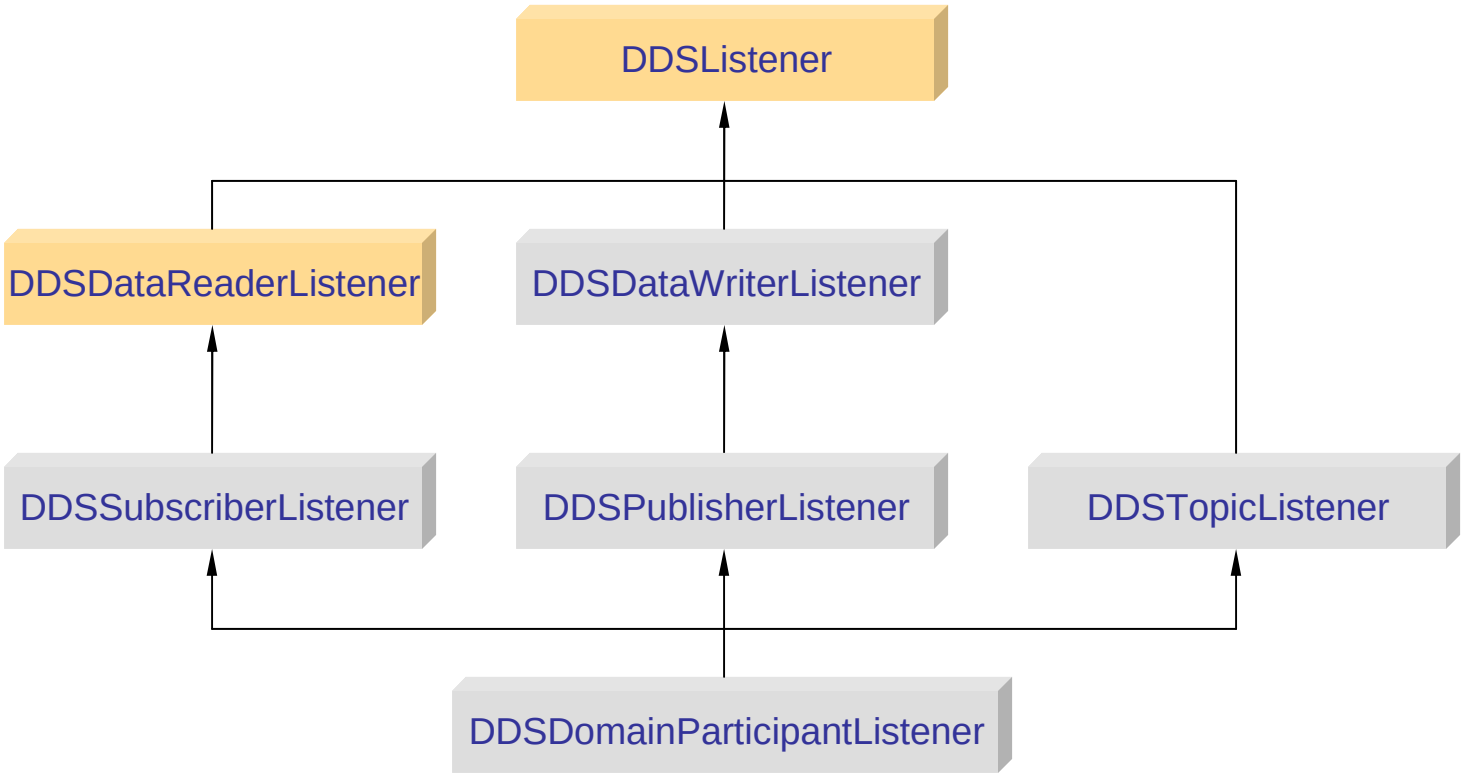
Step 2: Implement listener methods

```
void MyDataListener::on_data_available(DDSDataReader* reader)
{ etc. }
```

Step 3: Register listener in application

```
MyDataListener*  listener = new MyDataListener();
reader→set_listener(listener, DDS_StatusKindMask mask);
```

Part 1 – DataReader Listener



DataReader Listener

Events that DataReader Listener handles

- on_data_available( )
  - *new data is available to DataReader*
- on_requested_deadline_missed( )
  - *deadline has passed without new data arriving*
  - *includes status info on how many deadlines missed*
- on_liveliness_changed( )
  - *new DataWriter appeared | existing DataWriter disappeared*

DataReader Listener

Events that DataReader Listener handles (cont'd)

- on_requested_incompatible_qos( )
  - ***DataWriter found with incompatible QoS***
  - ***Includes which QoS policies are incompatible***
- on_sample_rejected( )
  - ***DataReader cannot place incoming sample into its queue***
  - ***Includes status of which resource limits were exceeded***

DataReader Listener

Events that DataReader Listener handle (cont'd)

- on_sample_lost( )
  - ***DataReader has detected that it has "lost" a sample***
- on_subscription_match()
  - ***DataReader has discovered a match with remote DataWriter***
  - ***Includes***
    - cumulative & incremental number matching DataWriters
    - handle to latest matching DataWriter

Other Listeners

DataWriter Listener

- on_offered_deadline_missed()
- on_offered_incompatible_qos()
- on_liveliness_lost()
- on_publication_match()

Topic Listener

- on_inconsistent_topic()

More Listeners

Subscriber Listener

- on_data_on_readers()
- Any DataReader method/mask not caught by DataReaderListener

Publisher Listener

- Any DataWriter method/mask not caught by DataWriterListener

DomainParticipant Listener

- Any DataWriter, DataReader, Publication, Subscription, or Topic method/mask not caught by the contained entity's listeners.

Read Communication Status callback order

DDS looks for listeners in the following order, will call only one

on_data_on_readers() on Subscriber

on_data_on_readers() on DomainParticipant

Can call notify_datareaders() to trigger on_data_available() calls

on_data_available() on DataReader

on_data_available() on Subscriber

on_data_available() on DomainParticipant

Entity Status callback order

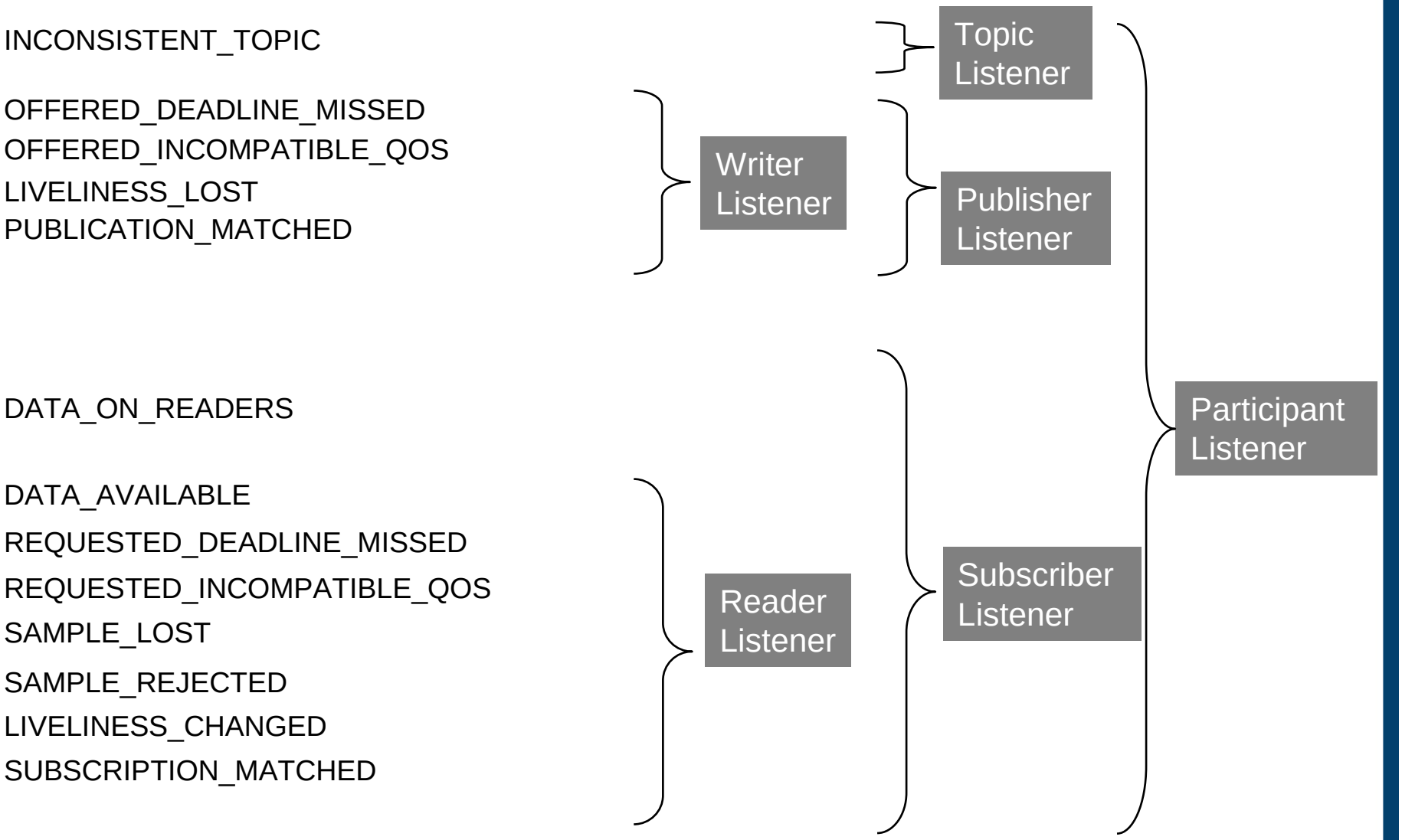
DDS looks for status mask listeners in the following order, will call only one

Reader/Writer (na for Topic Status)

Publisher/Subscriber /Topic (For Topic Status)

Participant

DDS Middleware Events Summary



Listeners, Conditions & WaitSets

Middleware must notify user application of relevant events

- Arrival of data, QoS violations
- Discovery of relevant entities
- These events may be detected asynchronously by the middleware

DDS allows the application a choice:

- Either get notified asynchronously using a Listener
- Or wait synchronously using a WaitSet

Both approaches are unified using STATUS changes

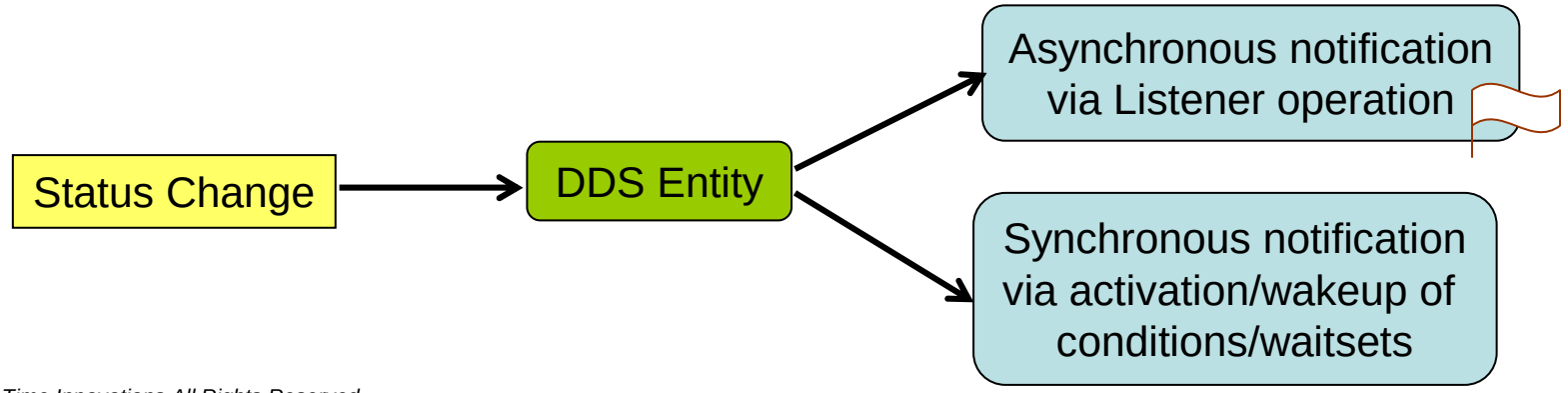
Listeners & Condition duality

A StatusCondition can be selectively activated to respond to any subset of the statuses

An application can wait changes in sets of Status Conditions using a WaitSet

Each time the value of a STATUS changes DDS

- Calls the corresponding Listener operation
- Wakes up any threads waiting on a related status change



DDS Advanced Tutorial

Background

Communication model

Concept Demo

DDS Entities

Listeners, Conditions, WaitSets

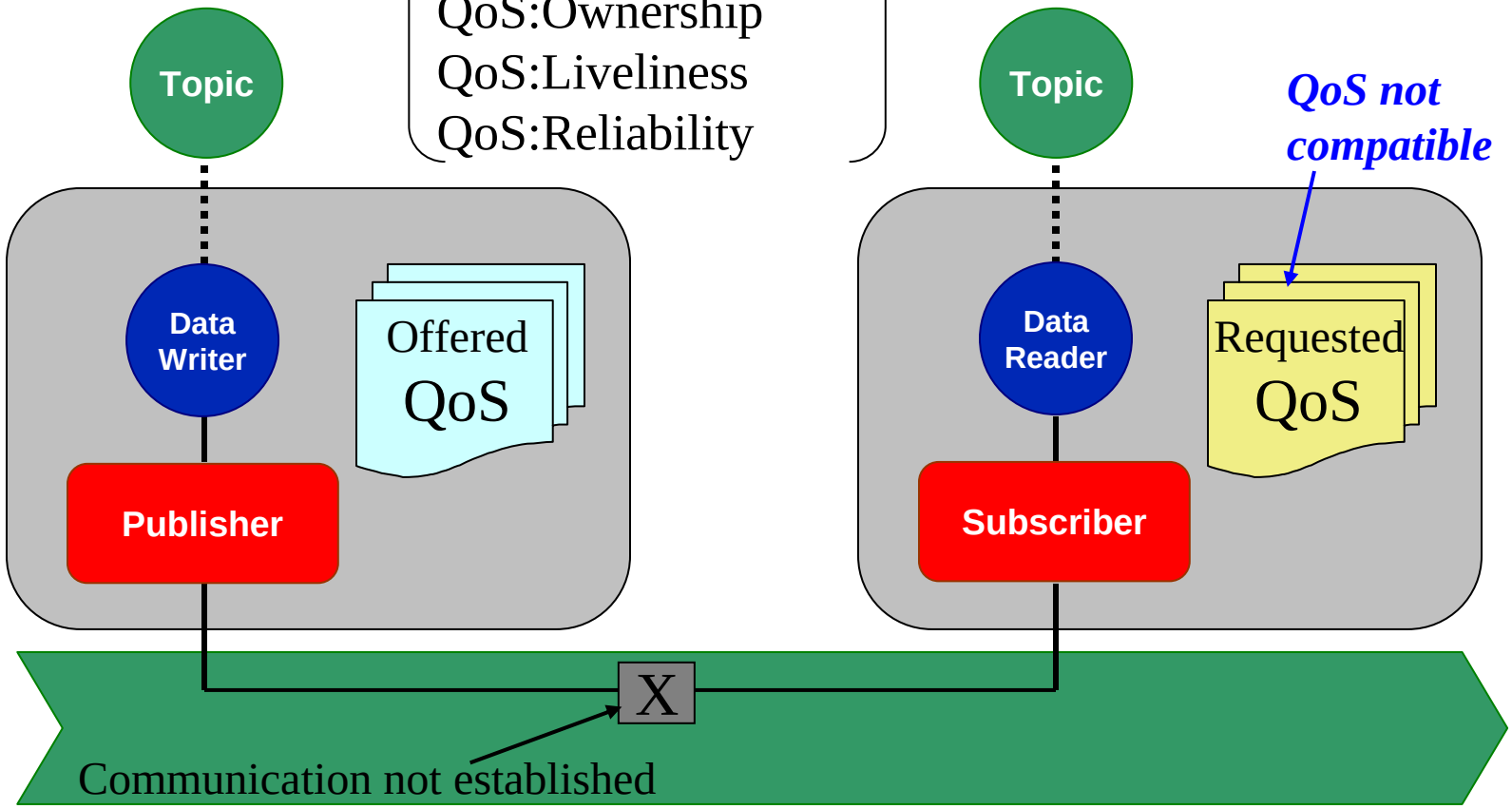
→ Quality of Service

Keys and instances

QoS Contract "Request / Offered"

QoS:Durability
QoS:Presentation
QoS:Deadline
QoS:Latency_Budget
QoS:Ownership
QoS:Liveliness
QoS:Reliability

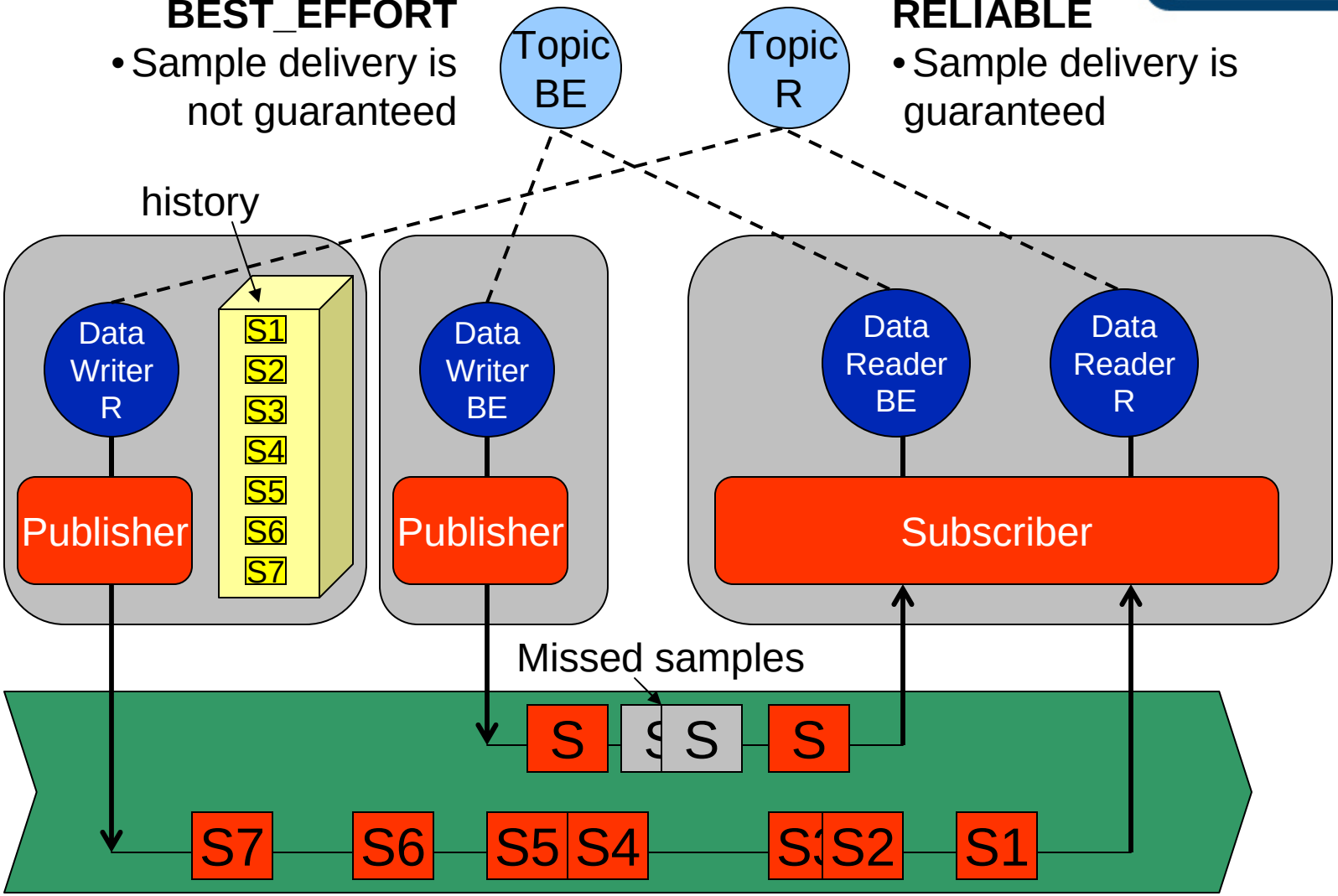
QoS Request / Offered:
Ensure that the compatible QoS parameters are set.



QoS: Reliability

BEST_EFFORT
- Sample delivery is not guaranteed

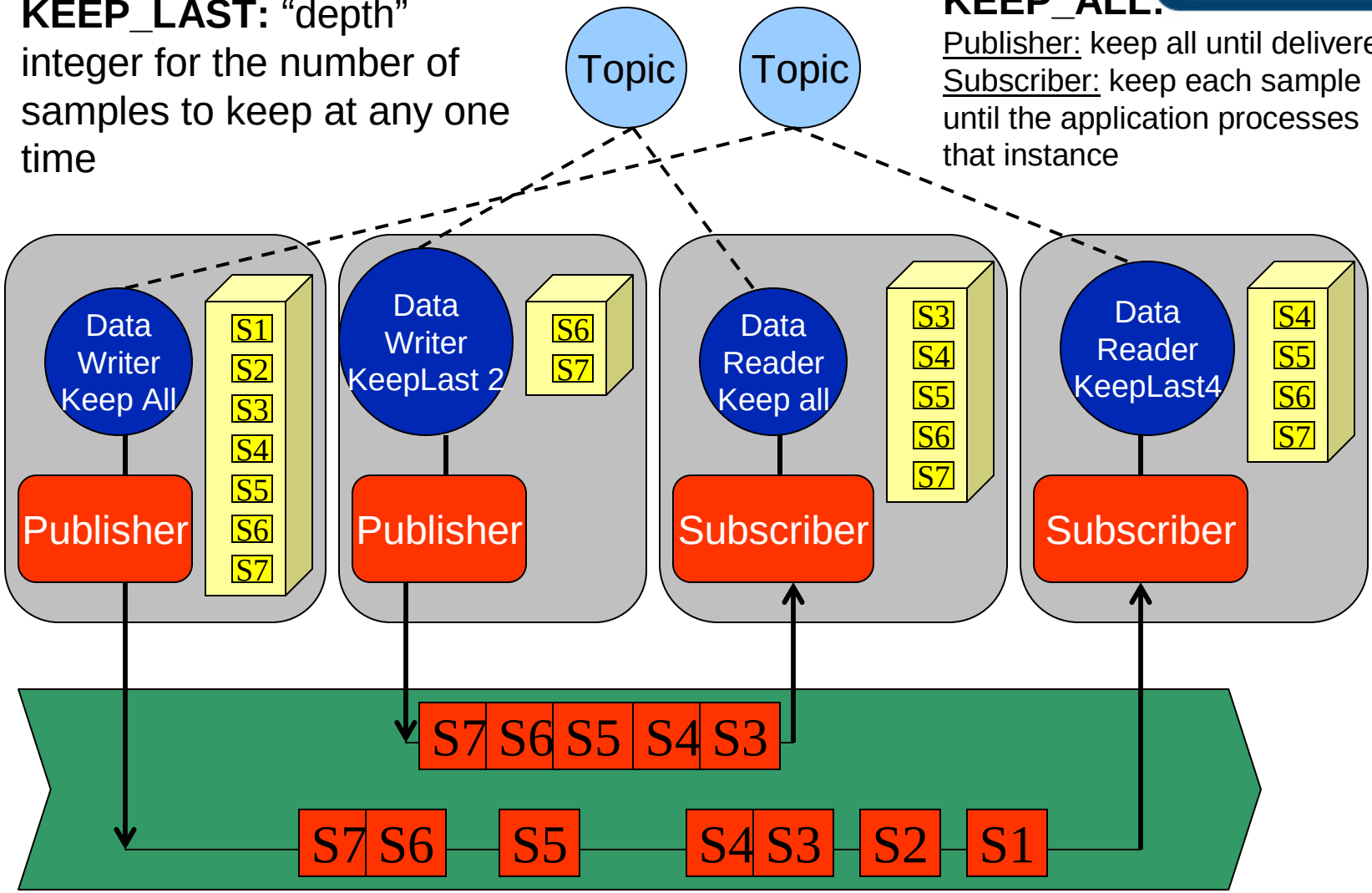
RELIABLE
- Sample delivery is guaranteed



QoS: History: Last x or All

KEEP_LAST: "depth" integer for the number of samples to keep at any one time

KEEP_ALL:
Publisher: keep all until delivered
Subscriber: keep each sample until the application processes that instance



State propagation

System state

- Information needed to describe future behavior of the system
  - ***System evolution defined by state and future inputs.***
- Minimalist representation of past inputs to the system

State variables

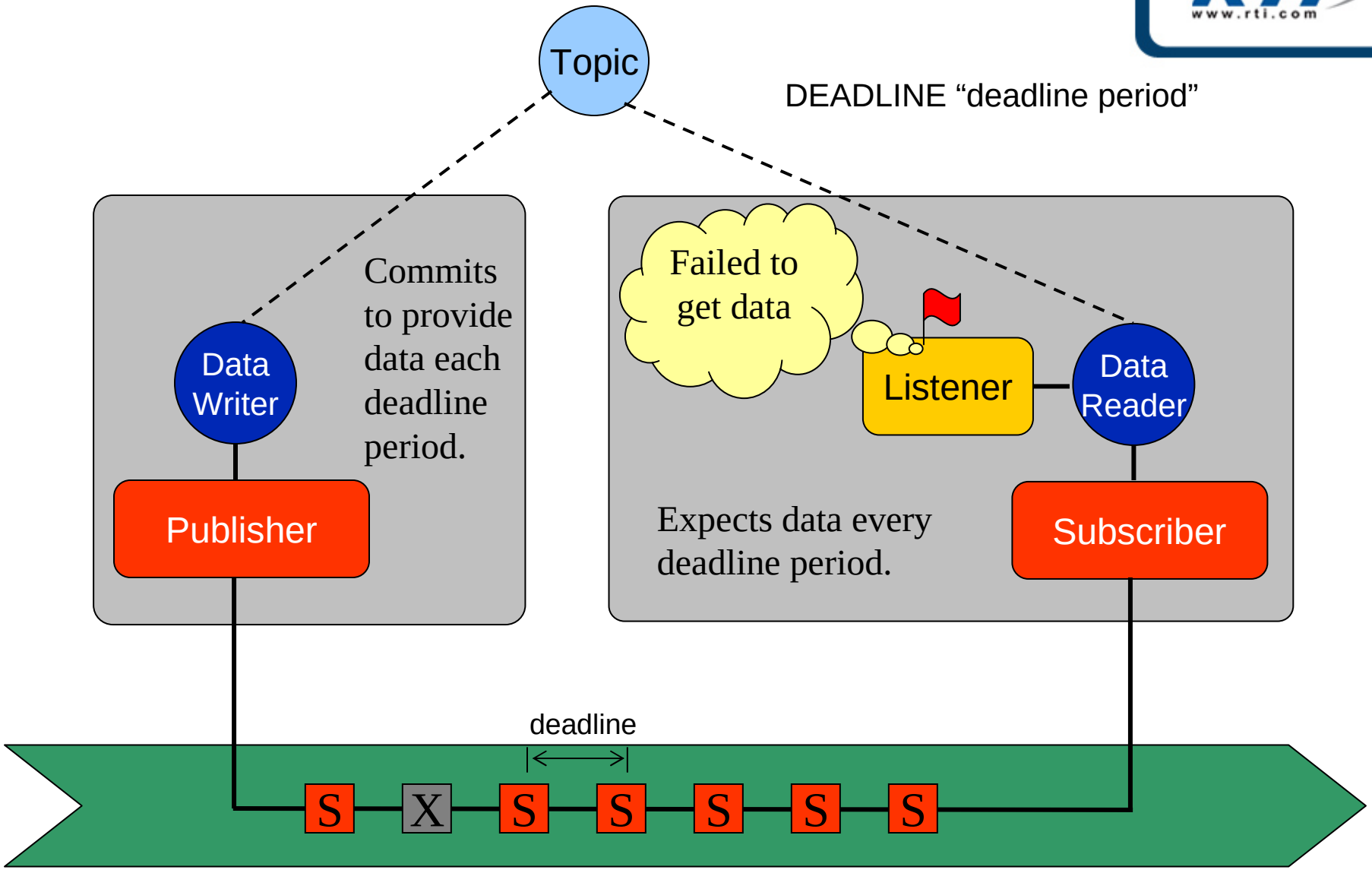
- Set of data-objects whose value codifies the state of the system

Relationship with DDS

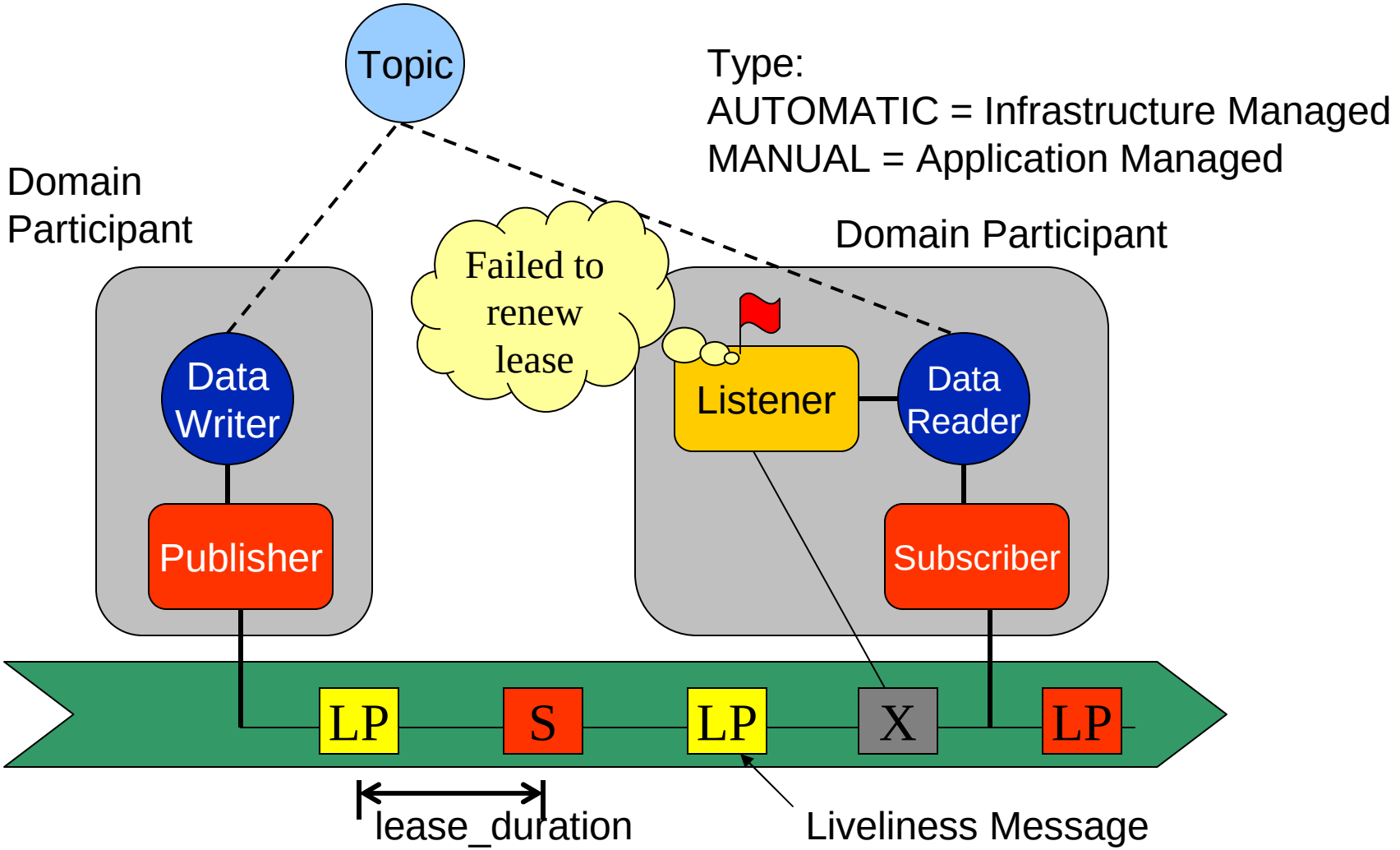
- DDS well suited to propagate and replicate state
- Topic+key can be used to represent state variables
- KEEP_LAST history QoS exactly matches semantics of state-variable propagation

Significance: Key ingredient for fault-tolerance and also present in many RT applications

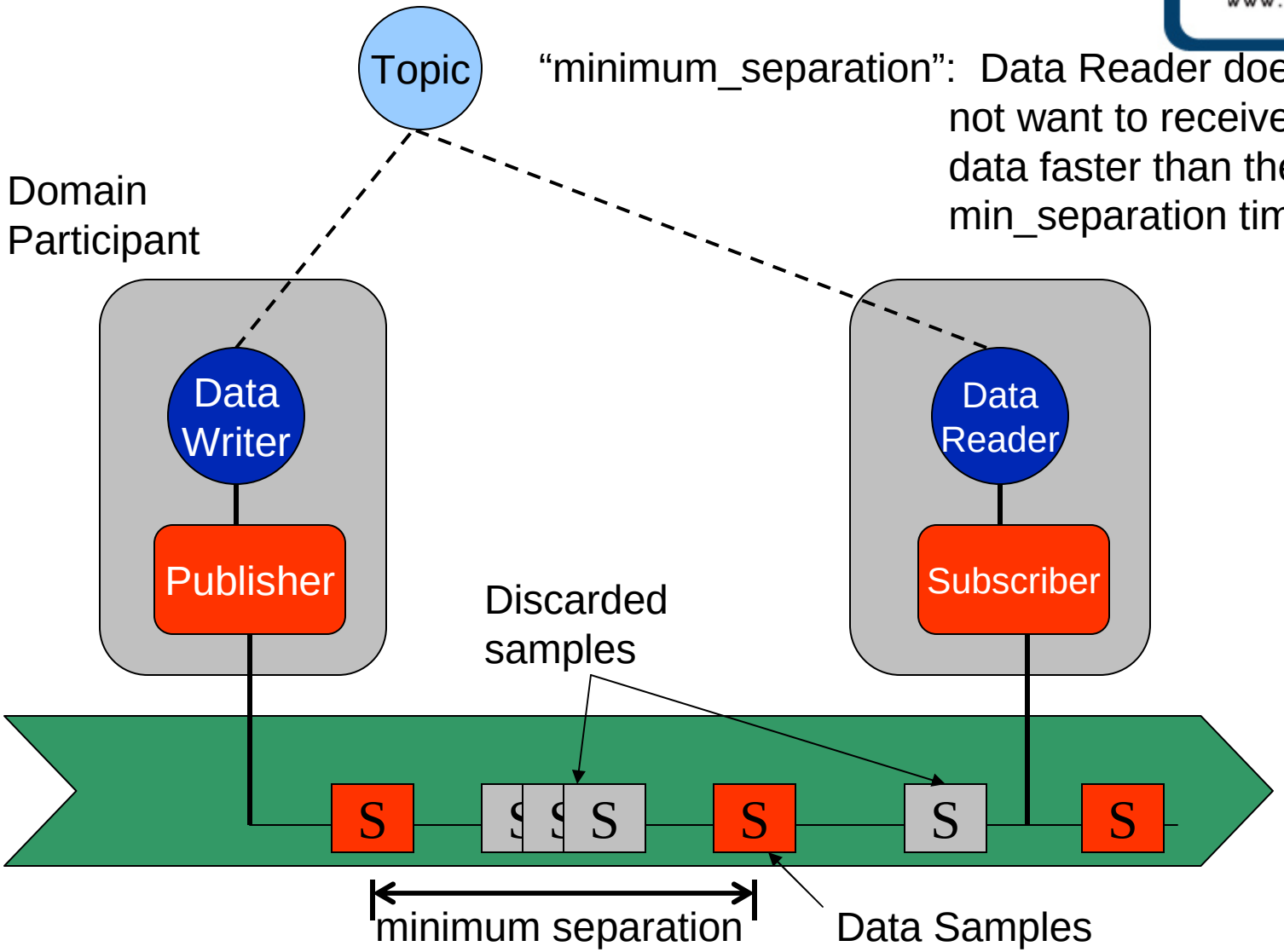
QoS: Deadline



QoS: Liveliness – Type, Duration



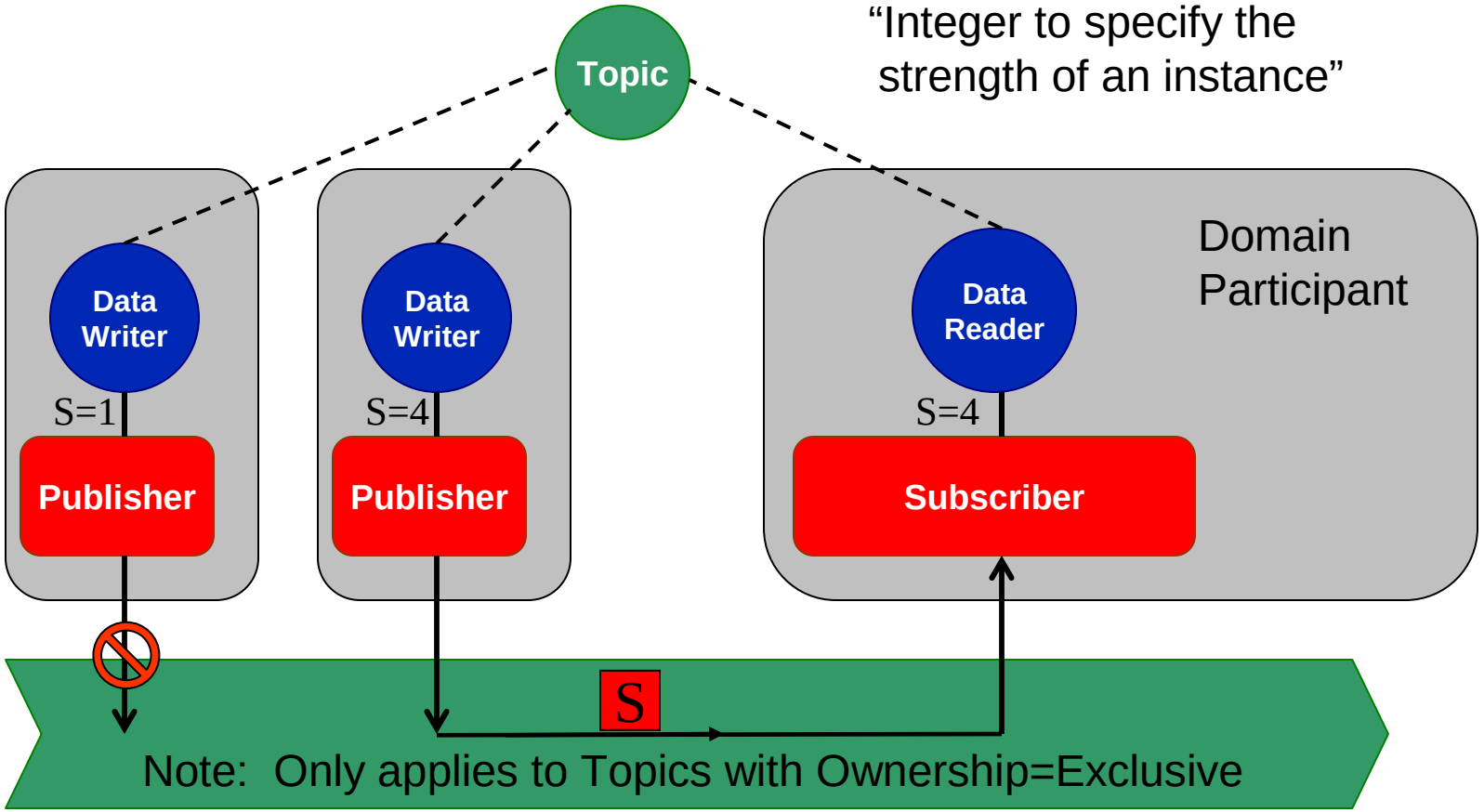
QoS: Time_Based_Filter



QoS: Ownership_Strength

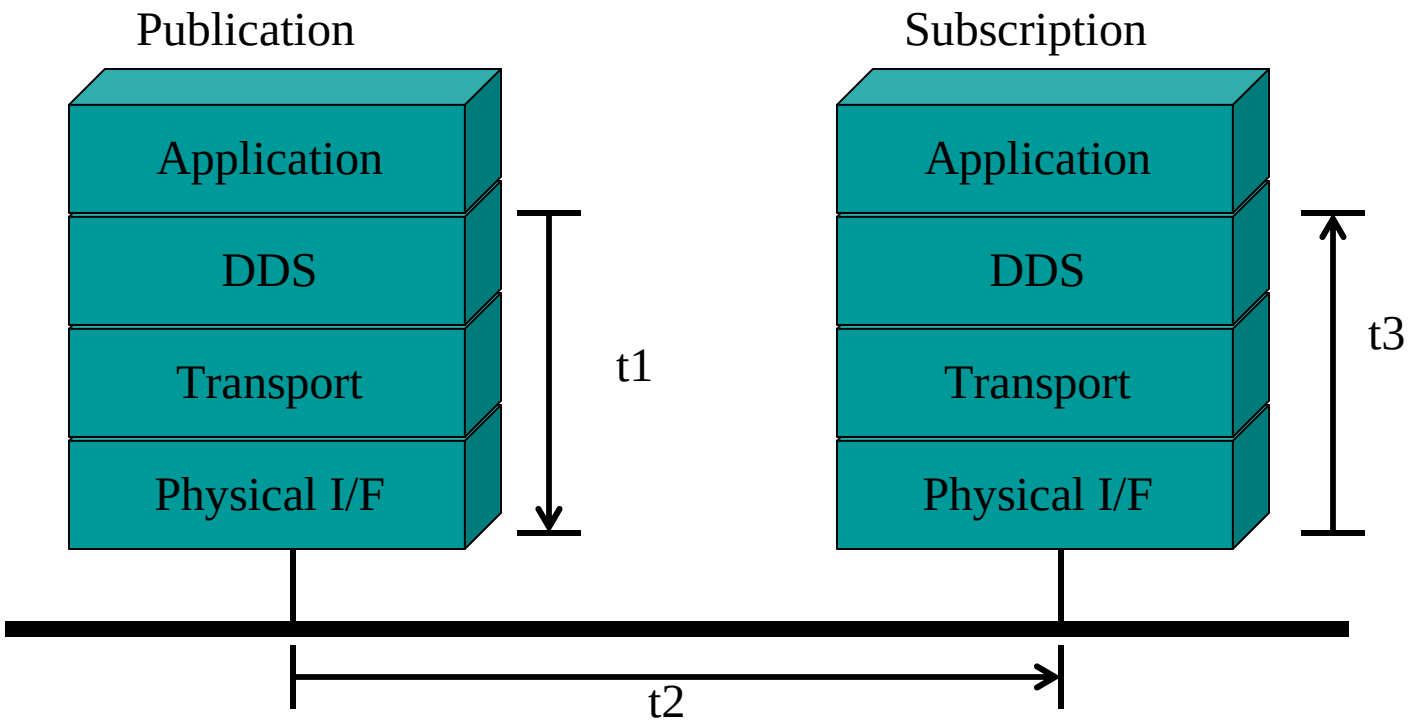
Ownership Strength: Specifies which writer is allowed to update the values of data-objects

OWNERSHIP_STRENGTH
"Integer to specify the strength of an instance"



QoS: Latency_Budget

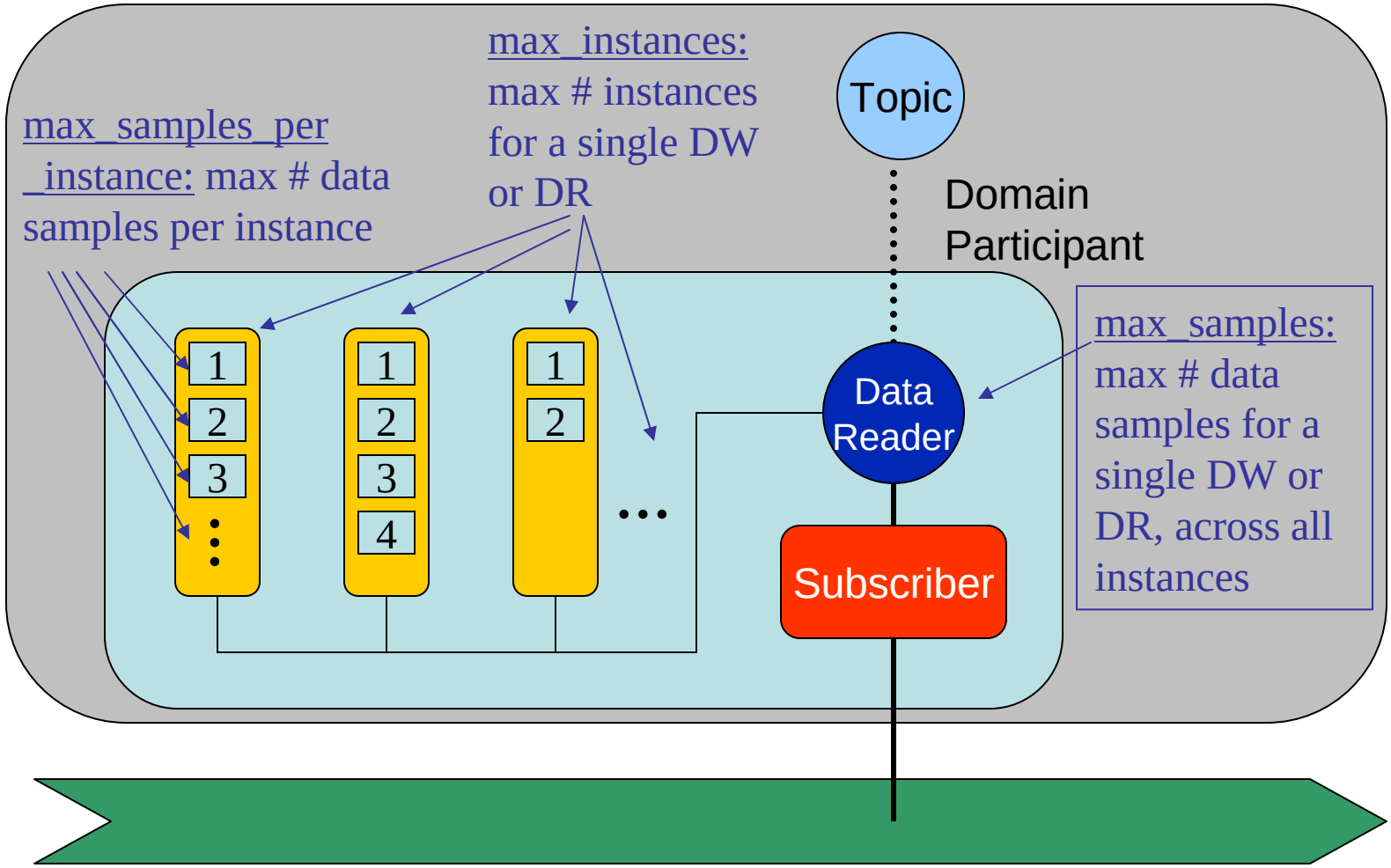
- Intended to provide time-critical information to the publisher for framework tuning where possible.
- Will not prevent data transmission and receipt.



Latency = t1 + t2 + t3

QoS: Resource_Limits

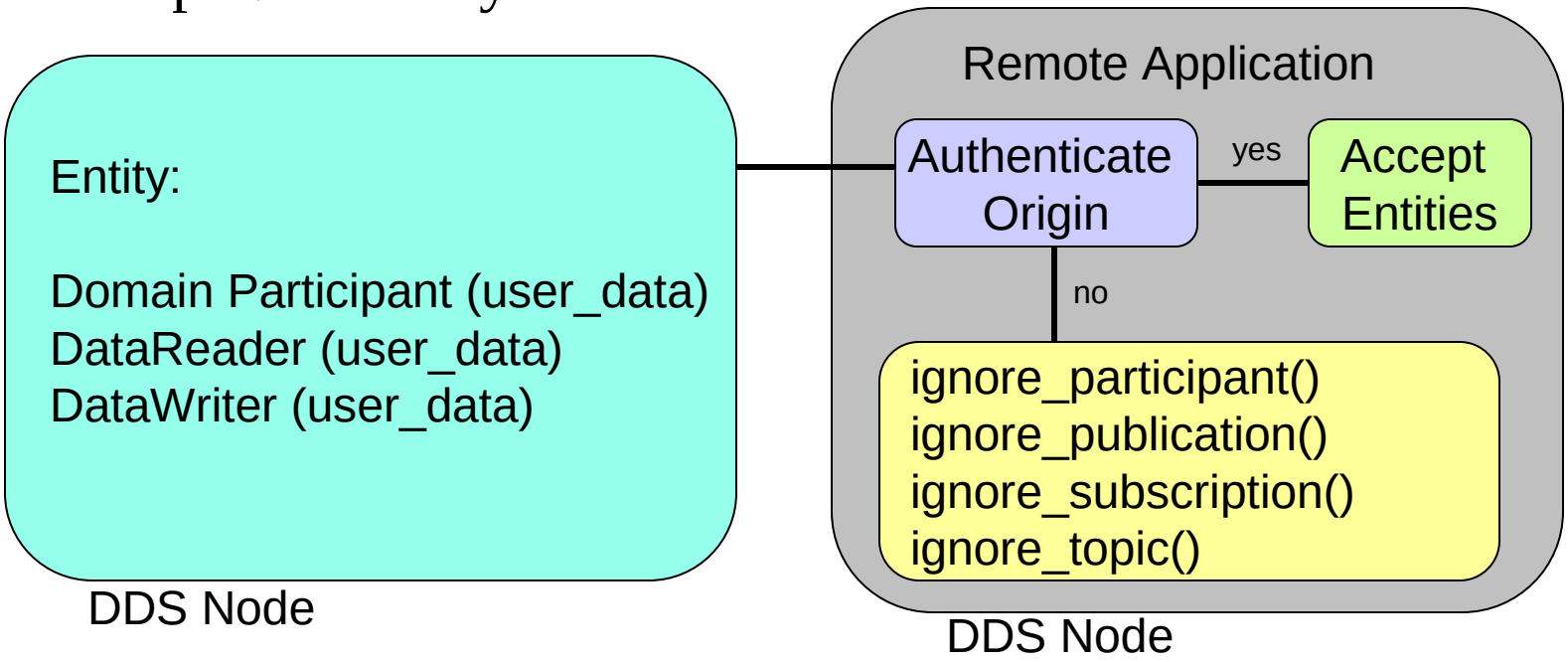
Specifies the resources that the Service can consume to meet requested QoS



QoS: USER_DATA

Definition: User-defined portion of Topic metadata

Example: Security Authentication

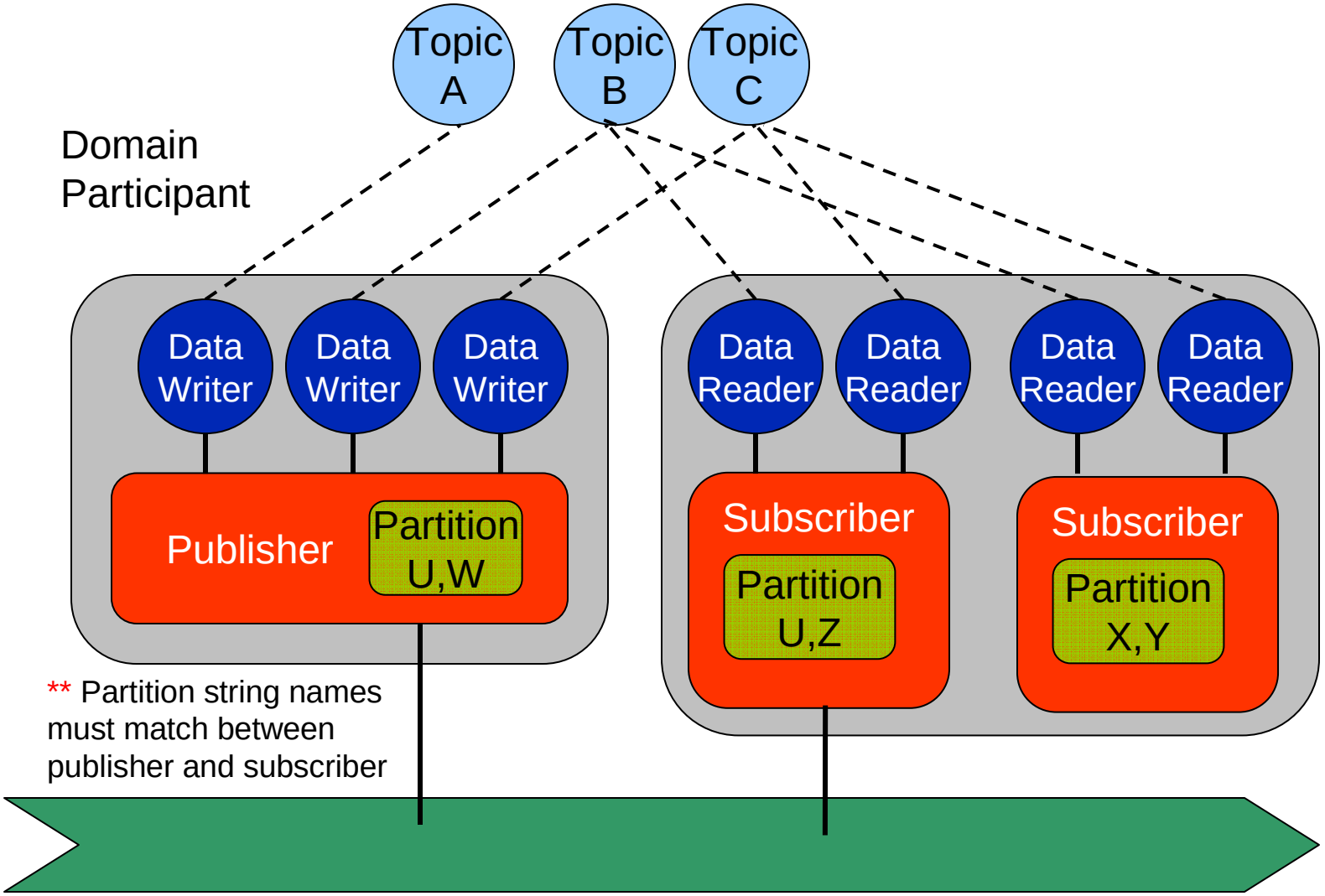


User data can be used to authenticate an origination entity.

Note: USER_DATA is contained within the DDS metadata.

QoS: Partition

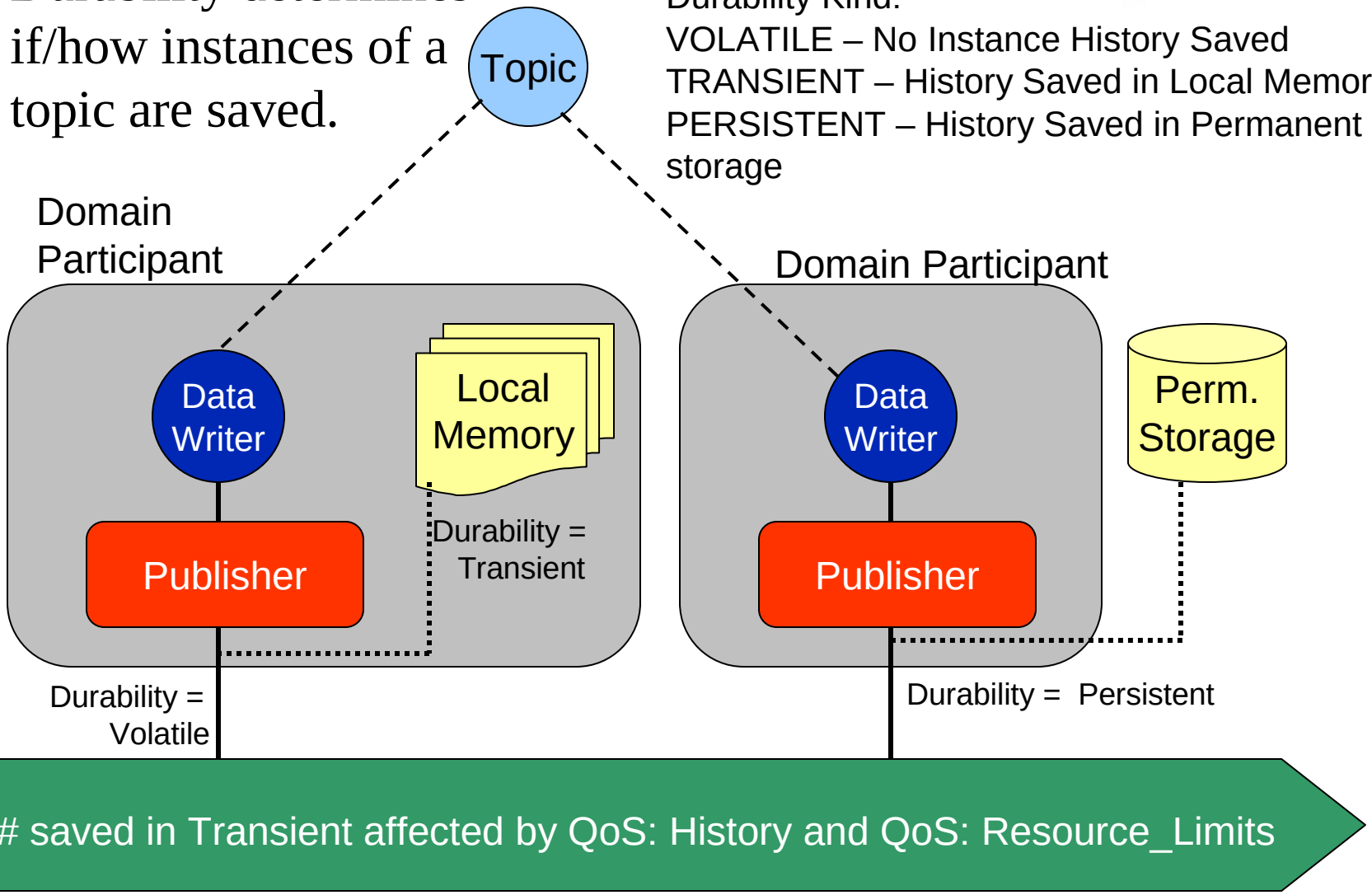
Partition: Logical “namespace” for topics



QoS: Durability

Durability determines if/how instances of a topic are saved.

Durability Kind:
VOLATILE – No Instance History Saved
TRANSIENT – History Saved in Local Memory
PERSISTENT – History Saved in Permanent storage



QoS: Presentation

- **Governs how related data-instance changes are presented to the subscribing application.**
- **Type: Coherent Access and Ordered Access**
  - Coherent access: All changes (as defined by the Scope) are presented together.
  - Ordered access: All changes (as defined by the Scope) are presented in the same order in which they occurred.
- **Scope: Instance, Topic, or Group**
  - Instance: The scope is a single data instance change. Changes to one instance are not affected by changes to other instances or topics.
  - Topic: The scope is all instances by a single Data Writer.
  - Group: The scope is all instances by Data Writers in the same Subscriber.

QoS: Quality of Service (1/2)

QoS Policy	Concerns	RxO	Changeable
DEADLINE	T,DR,DW	YES	YES
LATENCY BUDGET	T,DR,DW	YES	YES
READER DATA LIFECYCLE	DR	N/A	YES
WRITER DATA LIFECYCLE	DW	N/A	YES
TRANSPORT PRIORITY	T,DW	N/A	YES
LIFESPAN	T,DW	N/A	YES
LIVELINESS	T,DR,DW	YES	NO
TIME BASED FILTER	DR	N/A	YES
RELIABILITY	T,DR,DW	YES	NO
DESTINATION ORDER	T,DR	NO	NO

QoS: Quality of Service (2/2)

QoS Policy	Concerns	RxO	Changeable
USER DATA	DP,DR,DW	NO	YES
TOPIC DATA	T	NO	YES
GROUP DATA	P,S	NO	YES
ENTITY FACTORY	DP, P, S	NO	YES
PRESENTATION	P,S	YES	NO
OWNERSHIP	T	YES	NO
OWNERSHIP STRENGTH	DW	N/A	YES
PARTITION	P,S	NO	YES
DURABILITY	T,DR,DW	YES	NO
HISTORY	T,DR,DW	NO	NO
RESOURCE LIMITS	T,DR,DW	NO	NO

DDS Advanced Tutorial

- Background
- Communication model
- Concept Demo
- DDS Entities
- Listeners, Conditions, WaitSets
- Quality of Service
- → Keys and instances

Keys

Definition:

- DDS uses *keys* to uniquely identify each data instance in the system.
- DDS Data Writers (DW) can update multiple instances of a given topic. Similarly, Data Readers (DR) can receive updates from multiple instances of a given topic.

Why use keys?

- Avoids proliferation of topics
- A single DW can update multiple data instances
- Simplifies data distribution in a dynamic system where instances come and go

Use cases

Radar tracks

- Subscribe to tracks using single topic “tracks”
- Each airplane represented by separate instance
- Airplane (dis)appearance maps to instance lifecycle
- No need to know number of instances beforehand

Dynamic discovery

- Receive all entity info using single topic
- Instance key maps to entity GUID

In general, any application interested in lifecycle and updates for unknown number of instances

- Stock price updates, warehouse package tracking, ...

Keys API support

DataWriter API

- register_instance / unregister_instance
- write
- dispose
- get_key_value

DataReader API

- read / read_instance / read_next_instance
- take / take_instance / take_next_instance
- get_key_value
- lookup_instance

Instance state

For each instance, DDS maintains an instance state:

- ALIVE : there are live DWs writing this instance
- NOT_ALIVE_DISPOSED: a DW explicitly disposed the instance. If ownership QoS == exclusive, only owner can dispose an instance
- NOT_ALIVE_NO_WRITERS : DR has concluded there are no more writers writing this instance

Usage:

- Instance state available as part of sample info
- Detect disposed instances
- Detect a specific instance lost its writers

View state

For each instance, DDS maintains a view state:

- NEW_VIEW_STATE: this is the first time the DR accesses samples of this instance
- NOT_NEW_VIEW_STATE: the DR has already accessed samples of this instance

Usage:

- Detect new instances in the system
- Restrict reading to updates of known instances only

Selecting what samples to read

```
read/take (FooSeq &received_data,
        DDS_SampleInfoSeq &info_seq,
        long max_samples,
        DDS_SampleStateMask
    sample_states,
        DDS_ViewStateMask view_states,
        DDS_InstanceStateMask
    instance_states)
```

- view_states mask: only data samples matching one of these view states will be returned
- instance_states mask: only data samples matching one of these instance states will be returned

Keys and Instances

Details

→ Examples

Qos Perspective

Example: on_data_available()

```
void TrackListener::on_data_available(DDSDataReader*
    reader)
{
   TrackDataReader *Track_reader = NULL;
   TrackSeq data_seq;
   DDS_SampleInfoSeq info_seq;
   DDS_ReturnCode_t retcode;
   Track MyTrack;
   int i;

   Track_reader = TrackDataReader::narrow(reader);

   retcode = Track_reader->take(
      data_seq, info_seq, DDS_LENGTH_UNLIMITED,
      DDS_ANY_SAMPLE_STATE,  DDS_ANY_VIEW_STATE,
      DDS_ANY_INSTANCE_STATE);
```

Example: on_data_available()

Checking *view_state*

```
for (i = 0; i < data_seq.length(); ++i) {

        //Check flags
        switch (info_seq[i].view_state)
        {
          case DDS_NEW_VIEW_STATE:
              printf ("DDS_NEW_VIEW_STATE\n");
              break;

          case DDS_NOT_NEW_VIEW_STATE:
              printf ("DDS_NOT_NEW_VIEW_STATE\n");
              break;
        }
```

Example: on_data_available()

Checking *instance_state*

```
switch (info_seq[i].instance_state)
        {
          case DDS_ALIVE_INSTANCE_STATE:

             printf ("DDS_ALIVE_INSTANCE_STATE\n");
             break;

          case DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE:
            returnCode = Track_reader->get_key_value(
                                MyTrack,
                                info_seq[i].instance_handle
                                );

            printf
("DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE
                    for key -> %d\n",MyTrack.code );
            break;
        }
```

Example: on_data_available()

Finish check of *instance_state*, **then get** *valid_data*

```
case DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE:
    returnCode = Track_reader->get_key_value(
                                    MyTrack,
                                    info_seq[i].instance_handle
                                    );

    printf ("DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE
            for key -> %d\n",MyTrack.code);
    break;
  } //End Case for instance state

  if (info_seq[i].valid_data) {
     TrackTypeSupport::print_data(&data_seq[i]);
  }
 } //end for

 retcode = Track_reader->return_loan(data_seq, info_seq);
} // end on_data_available()
```

Keys and Instances

- Details
- Examples
- → Qos Perspective

Resource Limits QoS

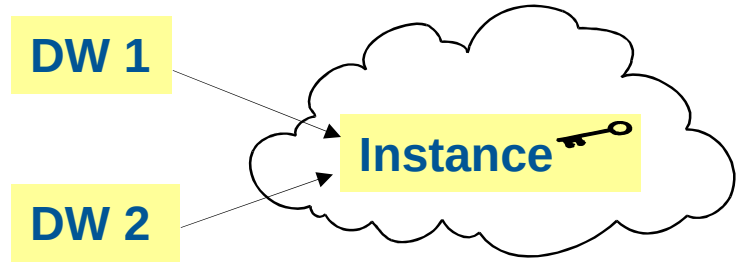
Specifies the resources available to the service (T, DW, DR)

- max_samples <unlimited>
  - ***total queue size* across all instances**
  - ***hard, physical limit***
- max_instances <unlimited>
  - ***maximum number of instances allowed***
  - ***logical queues***
- max_samples_per_instance <unlimited>
  - ***maximum size of logical queue for each instance***
  - ***Enables fairness across instances: no single instance can take over complete queue***

Ownership QoS

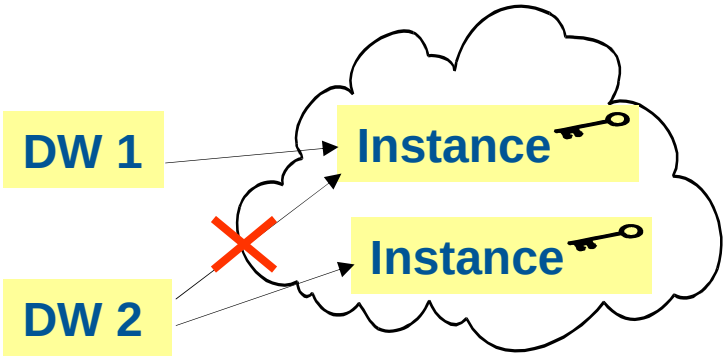
Specifies whether multiple DWs are allowed to modify the same instance of data (T,DR,DW)

- <*Shared*> – multiple Data Writers can update same sample instance
- *Exclusive* – only single Data Writer can update sample instance (i.e. owns the instance). To own an instance, DW must be alive, register/write to instance and have highest strength.



Strength N/A

Shared



Strength 2 < Strength 1

Exclusive

Ownership Strength QoS

Specifies relative strength among multiple Data Writers (DW)

- Highest strength DW **owns** the instance (if exclusive ownership) -> on instance-by-instance basis!
- Ownership can be lost due to:
  - ***Higher strength DW***
  - ***Owner looses liveliness or missed deadline***
  - ***Owner unregisters instance***
- Only the owner can dispose an instance.

Deadline QoS

Deadline indicates maximum time allowed to elapse before new data sample is sent or received (T, DR, DW)

- Deadline applies to ***each instance*** written or read.
- Allows each DW to declare at least how fast it will update ***each instance*** it writes to.
- Allows each DR to declare at least how fast it needs to receive updates for ***each instance*** it is reading.

Example on_requested_deadline_missed()

```
void TrackListener::on_requested_deadline_missed(
        DDSDataReader* reader,
        const DDS_RequestedDeadlineMissedStatus& status)
{
    DDS_ReturnCode_t returnCode;
    TrackDataReader *Track_reader = NULL;
    Track MyTrack;
    MyTrack.code = -1;

    DDS_InstanceHandle_t myInstanceHandle = status.last_instance_handle;

    Track_reader = TrackDataReader::narrow(reader);
    returnCode = Track_reader->get_key_value(
                                        MyTrack,
                                        myInstanceHandle
                                        );

    printf("ERROR Requested Deadline missed on Key %d\n",MyTrack.code);
}
```

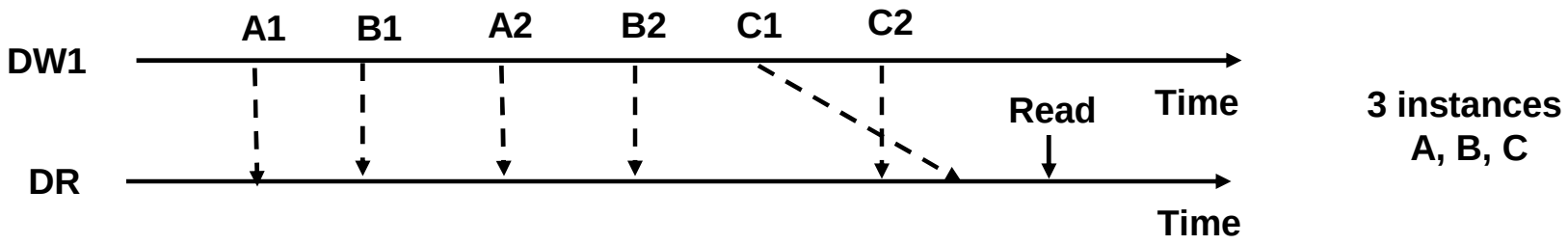
Delivery : Presentation QoS

Governs how "related" sample updates are presented to subscribing application

- Access SCOPE defines the extent of relation
  - <Instance> ***– spans a single instance***
  - *Topic* ***– spans all instances within same DW***
  - *Group****– all instances belonging to DWs within same Publisher***
- TYPE defines how data should be presented
  - *DDS_Boolean coherent_access* <false>
    - If TRUE, changes (within SCOPE) are presented together - publisher defines coherence
  - *DDS_Boolean ordered_access* <false>
    - If TRUE, changes (within SCOPE) are presented in the order they occurred on the DW

ordering

Presentation QoS: Example



- Ordered_access = TRUE
  - Access_scope = Instance: must maintain order for each instance.
    - A1, B1, A2, B2, C1, C2 or
    - A1, A2, B1, B2, C1, C2 or
    - B1, B2, A1, A2, C1, C2 are all valid sequences to return to the reader.
  - Access_scope = topic: must maintain order across instances.
    - A1, B1, A2, B2, C1, C2 is the only option

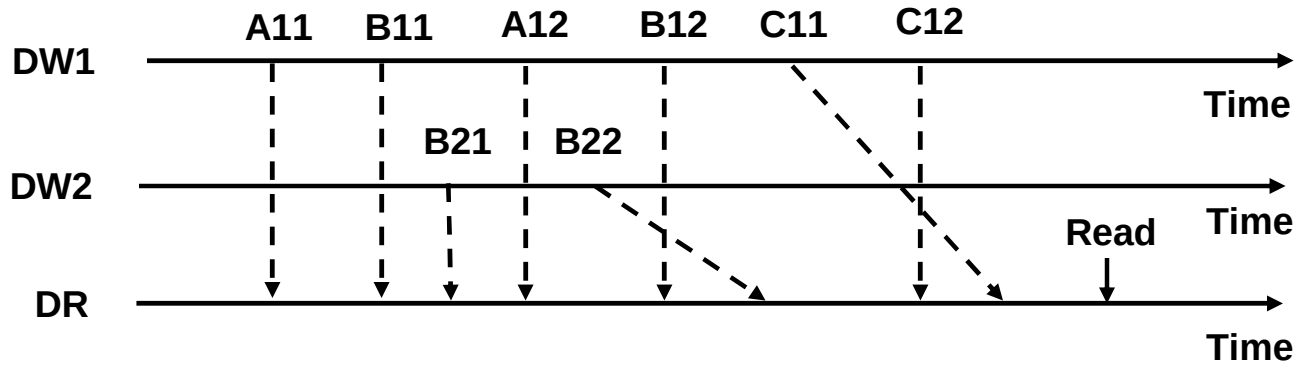
- Ordered_acces = FALSE → can return samples in any order
  - A1, B1, A2, B2, C1, C2

Destination Order QoS

Specifies sample ordering when multiple DWs share ownership of same data instance (T,DR,DW)

- enum **kind**
  - **<By Reception Timestamp> –** ***last sample received at at destination is kept***
  - **By Source Timestamp –** ***sample with latest origin timestamp is kept***
- Usage:
  - ***Determine ordering of data when sent from multiple sources to multiple destinations***
  - ***The final value of each instance received by all DRs are guarantee to be the same.***
  - ***No guarantee of the samples in the history, because out-of-order samples are discarded and not stored in the reader queue.***

Destination order QoS: Example



3 instances
A, B, C

Ownership shared
between DW1 & DW2

Ordered-access = TRUE, Access_scope = Instance, destination_order = by_source

A11, A12, B11, B21, B22, B12, C11, C12

A11, B11, B21, A12, B22, B12, C11, C12 are all valid sequences to return to DR

Ordered-access = TRUE, Access_scope = Instance, destination_order = by_reception

A11, A12, B11, B21, B12, B22, C11, C12 → C's order does not change!! (from single DW)

Ordered-access = TRUE, Access_scope = Topic, destination_order = by_source

A11, B11, B21, A12, B22, B12, C11, C12

Ordered-access = TRUE, Access_scope =Topic, destination_order = by_reception

A11, B11, B21, A12, B12, B22, C11, C12

History QoS

Specifies* how many *data samples should be archived (T, DR, DW)

- enum **history.kind** =
  - **<Keep Last> – *store last* depth *samples of data***
  - **Keep All – *store all samples of data (up to available resource limits)***
- **history.depth**
  - ***< 1 > number of samples to be kept***

The above settings apply on an instance-by-instance basis!

Writer Data Lifecycle QoS

Specifies whether Data Writer should also dispose its instances when they are unregistered (DW)

- DDS_Boolean **autodispose_unregistered_instances**
  - **<TRUE>**
  - ***FALSE – data remains available, not automatically disposed***

- Usage:
  - ***Auto-removal of instances when they are unregistered.***
    - E.g: unregister_instance()
  - ***Auto-removal of instances when their DW is deleted.***
  - ***Saves having to explicitly dispose instances.***

Reader Data Lifecycle QoS

Specifies how long a DR must retain information regarding instances that have the instance state NOT_ALIVE_NO_WRITERS (DR)

- DDS_Duration_t **autopurge_nowriter_samples_delay**
  - **<Infinity>**
- DDS_Duration_t **autopurge_disposed_samples_delay**
  - **<Infinity>**
- Action:
  - ***After duration expires, instance information and any untaken samples are purged from receive queue***
- Usage:
  - ***Auto-removal of data and info for instances that have indicated instance state. Prevents having to take all samples to free up resources.***

Conclusion

Summary

DDS targets applications that need to distribute data in a real-time environment

DDS is highly configurable by QoS settings

DDS provides a shared "global data space"

- Any application can publish data it has
- Any application can subscribe to data it needs
- Automatic discovery
- Facilities for fault tolerance
- Heterogeneous systems easily accommodated