



DDS – Advanced Tutorial *Using QoS to Solve Real-World Problems*

OMG Real-Time & Embedded Workshop
July 9-12, Arlington, VA

Gordon A. Hunt
Chief Applications Engineer, RTI

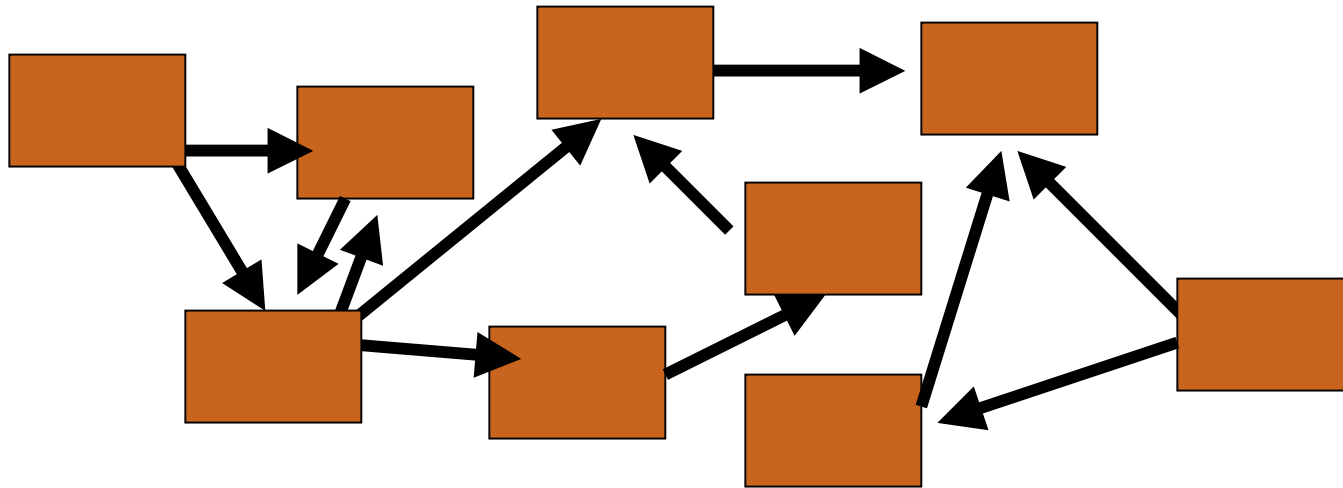
Agenda

- DDS Short Review
 - Publish/Subscribe & Data-Centric Design
- DDS QoS Use-Cases
 - QoS and Sending Data
 - QoS and State Information
 - QoS with Alarms and Events
 - QoS with Discovery
 - Sample Application Requirements mapped to QoS
- In Depth on Keys, and other topics

Uniqueness of Distributed Systems

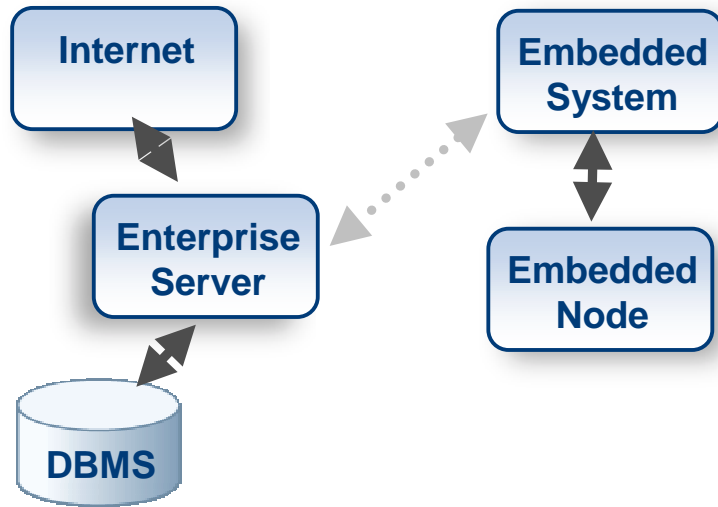
- The fundamental challenge: Getting the right data to the right place at the right time
- Office and enterprise networks
 - Sharing files
 - Transactions
- Real-time systems
 - Find and disseminate information quickly to many nodes
 - Meet strict delivery requirements
 - Control exactly when & where information is delivered

Distributed System Challenges

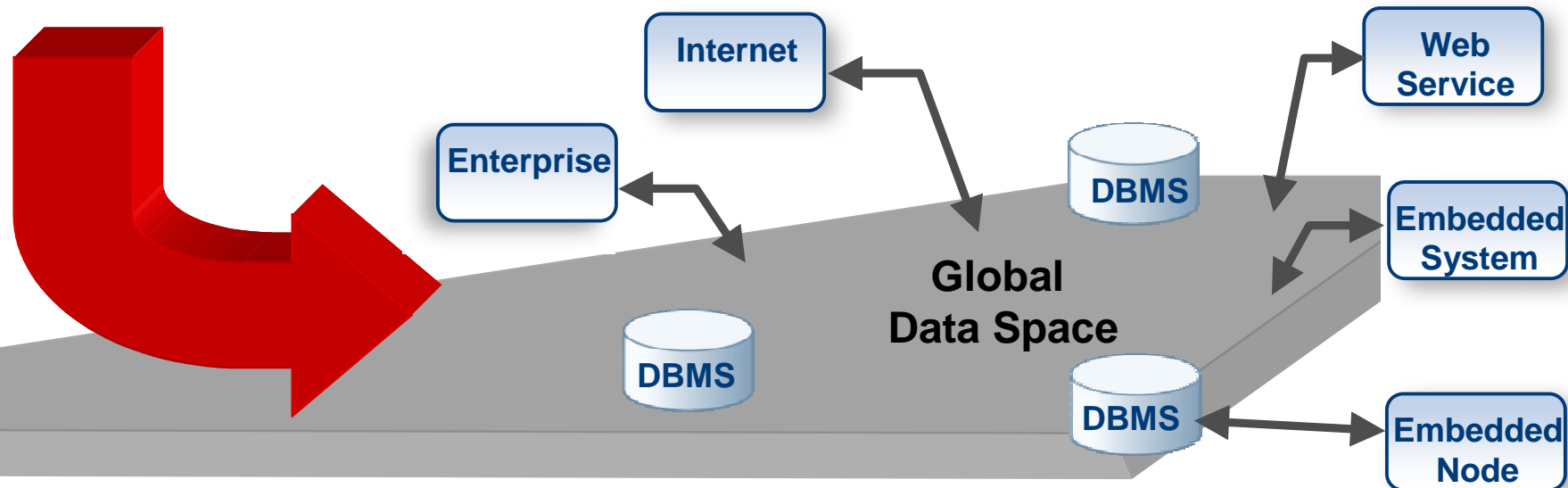


- Many more issues!
 - Dynamic node configuration
 - Congestion
 - Start up sequence
 - Node failure
 - Transmission failures
 - Can't even define “what time is it”?
- Networks are too complex to be “real time”
 - But...we still need to work with a real-time system...how?

Pervasive Data

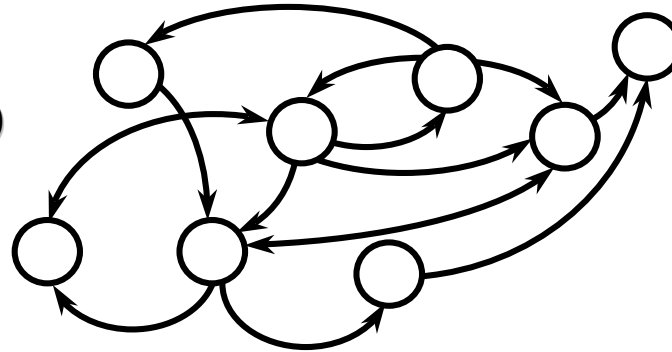


- Today's newly-connected distributed systems are just the beginning
- Pervasive data will drive the vast distributed applications of the future

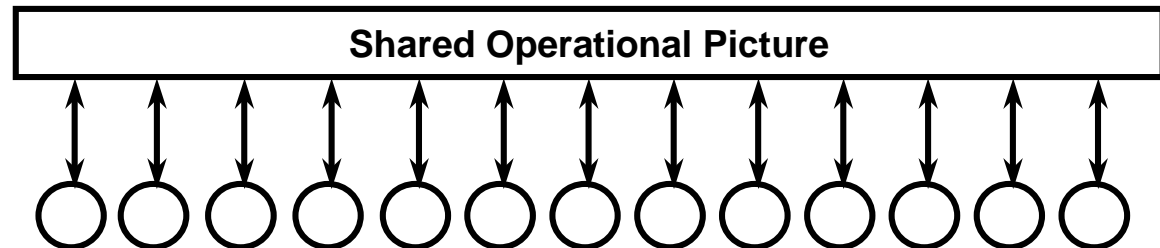


Solution requires Shift to Data-Oriented Design

NOT THIS:
(connection-oriented)



BUT THIS:

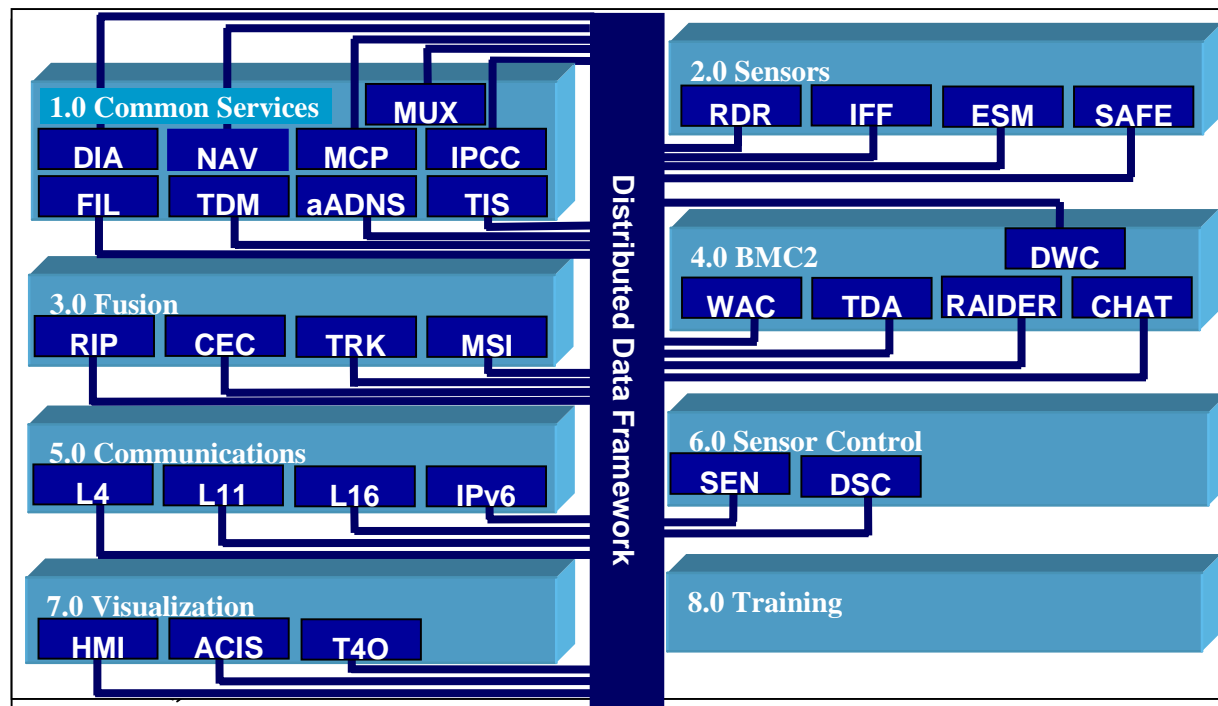


○ = System Components

Use of Information/Data –Oriented design for Pub-Sub applications avoids creating stovepipe systems

Data-Oriented and Distribution Middleware

- Architecture can evolve over time and assimilate new capabilities
- Information flow is not pre-determined



UNCLASSIFIED

Data-Centric Test

- Think about a recent project...
- How much of the code is managing and manipulating data?
 - Defining, initializing, and managing data structures
 - Gathering, aggregating, and transforming data
 - Using data
- How much? Greater than 50% likely...
 - Now ask, “What data model did you use?”

Data-Centric Methodology

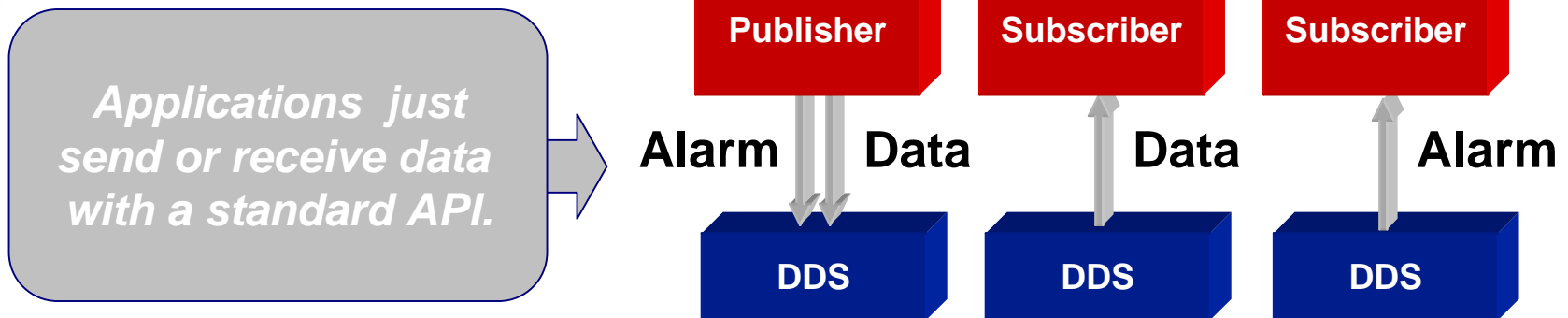
- Tenets of Data Oriented Programming
 - Expose the data
 - Hide the code
 - Separate data and code
 - Data-handling and data-processing
 - Code generated from Interfaces
 - Loosely coupled
 - Data separate from technology



Data-Oriented = Data-Centric = Net-Centric

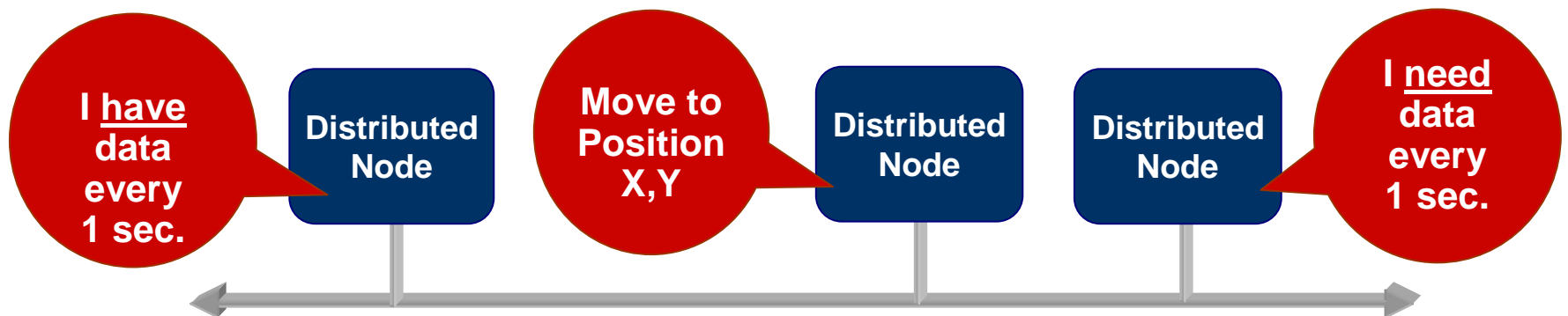
Why DDS?

- Data-Centric Communications
 - Just declare your intent to publish or receive data.
 - No need to make a special request for every piece of data.



Why DDS?

- Data distribution with minimal overhead
 - No reply confirmation needed
 - Very flexible QoS on a per-data-stream basis
 - Determinism vs. reliability
 - Cyclic \leftrightarrow acyclic messages
 - Bandwidth control
 - Fault tolerance (esp. over unreliable media)



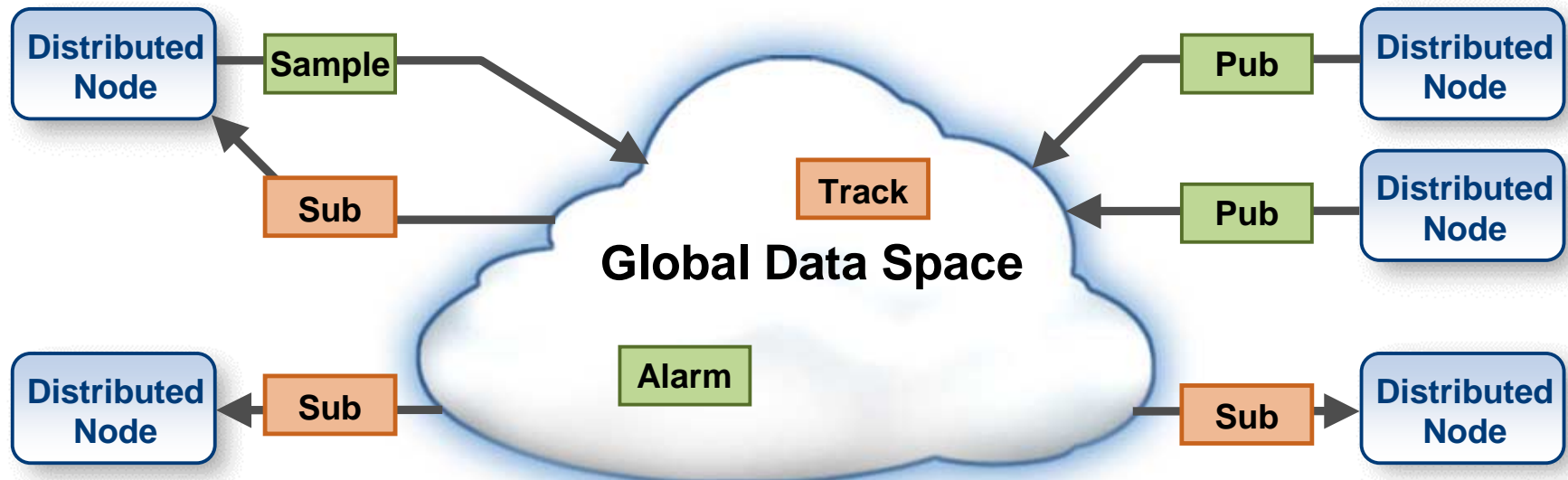
Why DDS?

Decoupling

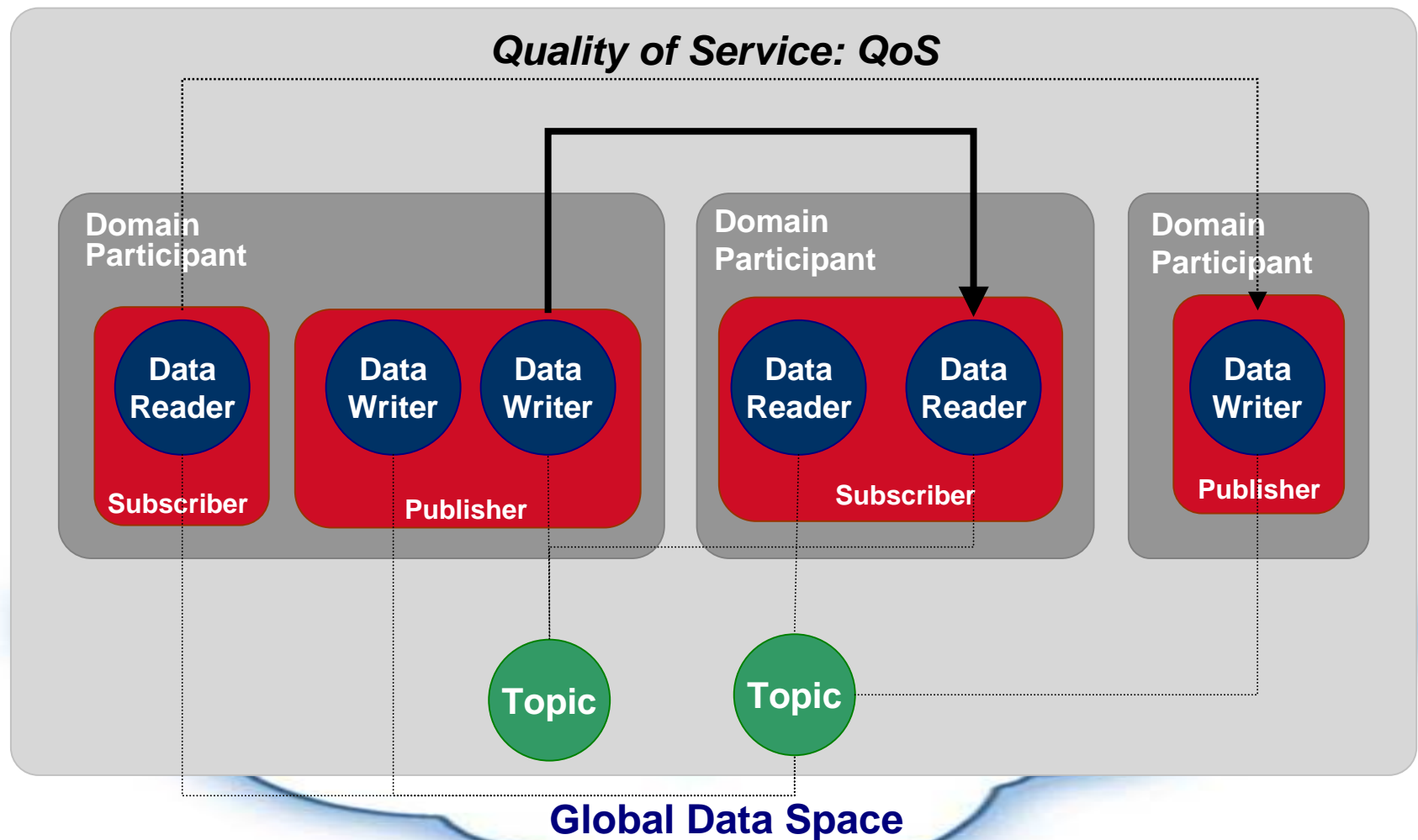
- Location: reduce dependencies
- Redundancy: multiple readers & writers
- Time: data when you want it
- Platform: Connect any set of systems

Benefits

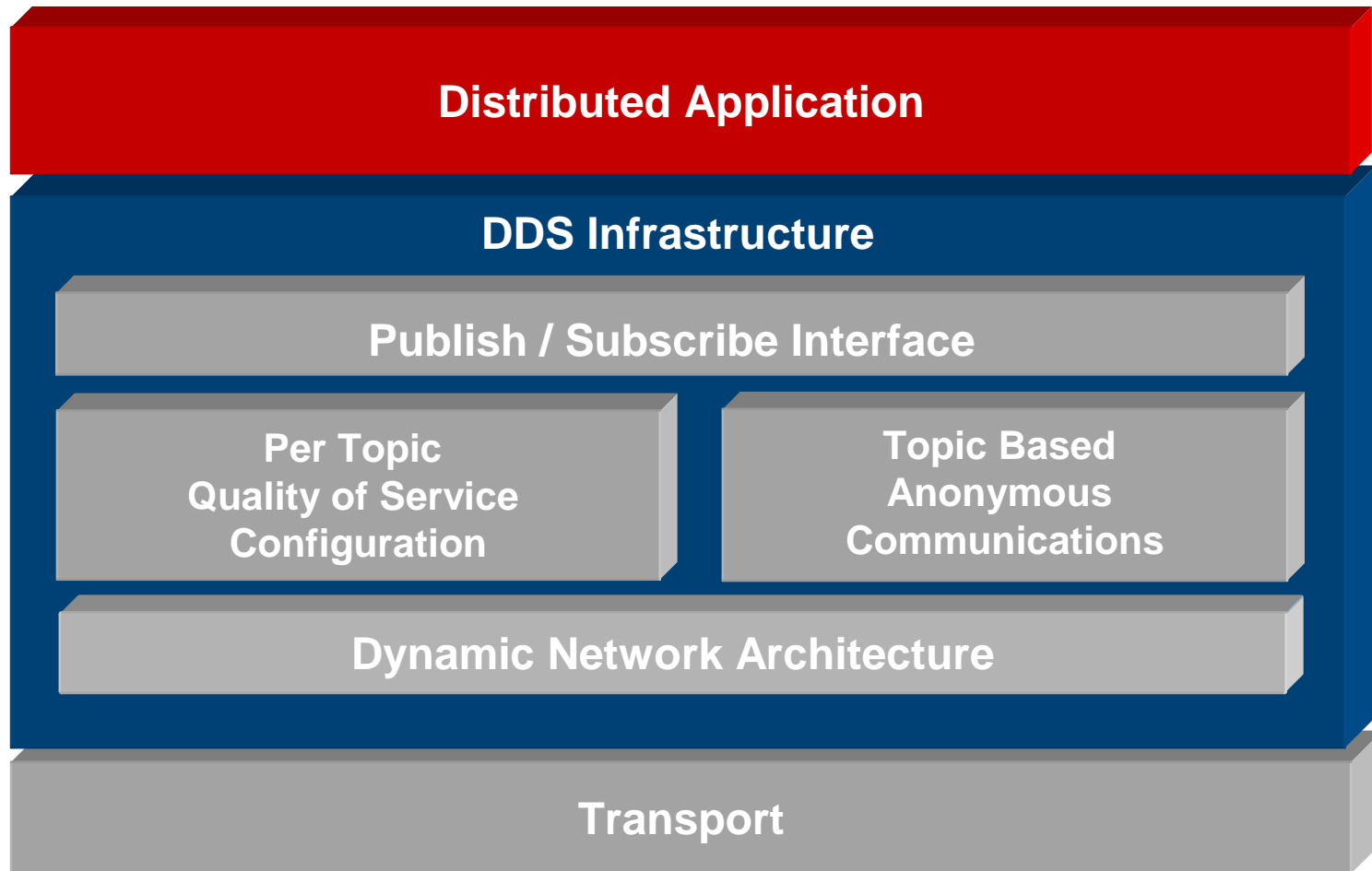
- Modular structure
- Flexibility
- Power



DDS Infrastructure

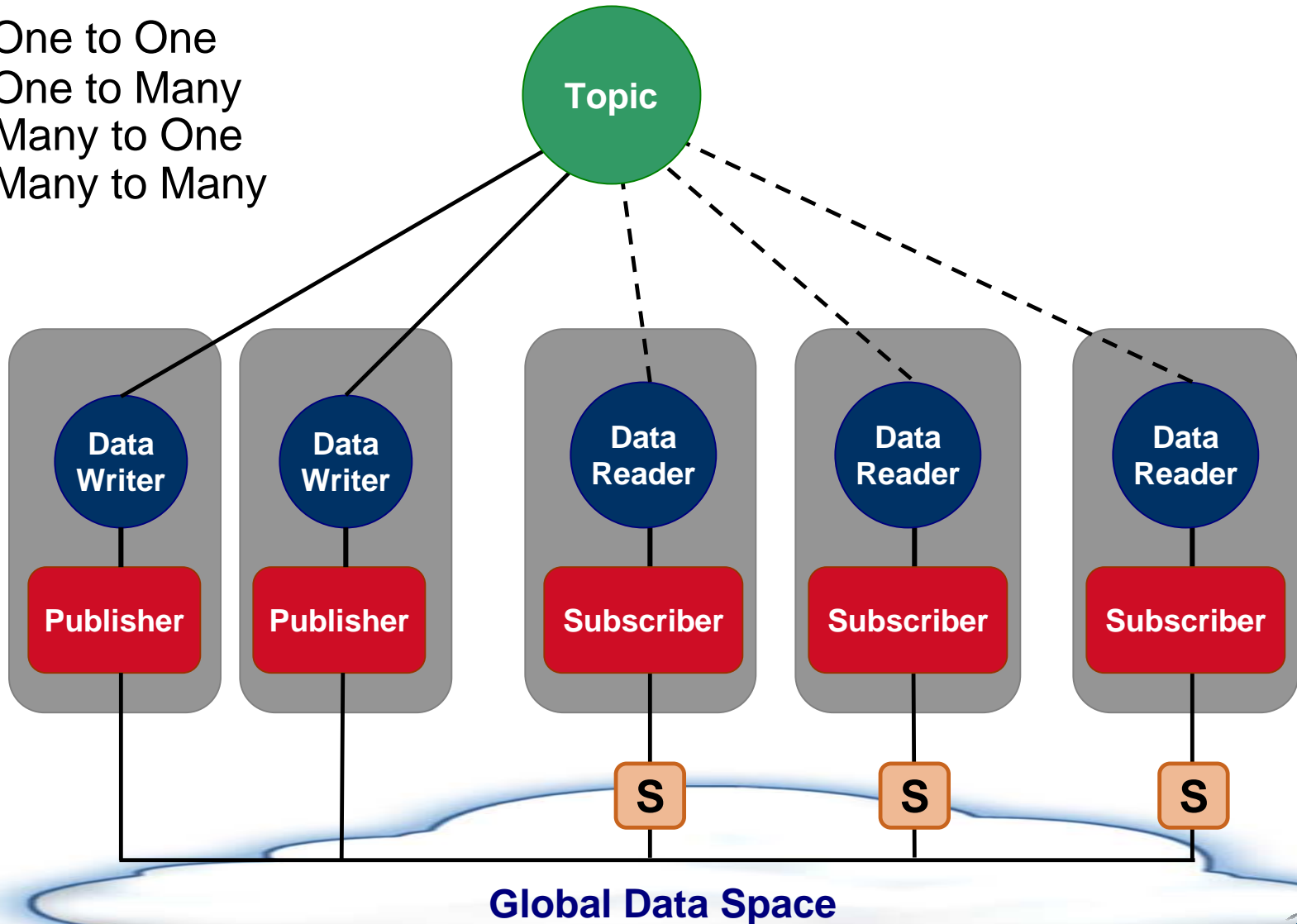


The DDS Infrastructure



DDS: Pub/Sub Scenarios – Loosely Coupled

One to One
One to Many
Many to One
Many to Many



QoS: Quality of Service

QoS Policy		
Volatility	DURABILITY	User QoS
	HISTORY	
	READER DATA LIFECYCLE	
	WRITER DATA LIFECYCLE	
Infrastructure	LIFESPAN	Presentation
	ENTITY FACTORY	
	RESOURCE LIMITS	Redundancy
	RELIABILITY	
Delivery	TIME BASED FILTER	Transport
	DEADLINE	
	CONTENT FILTERS	

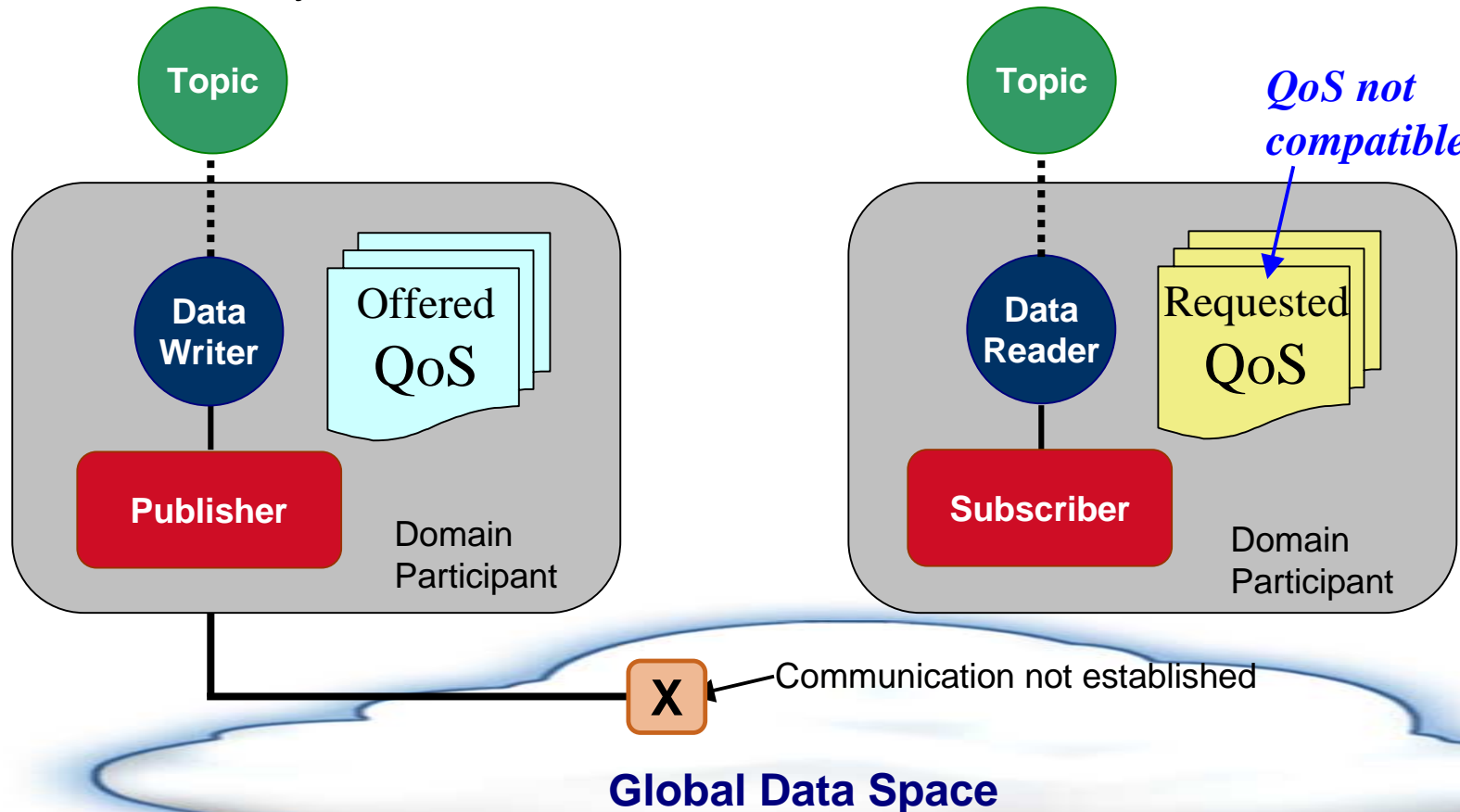
QoS Policy		
	USER DATA	User QoS
	TOPIC DATA	
	GROUP DATA	
	PARTITION	Presentation
	PRESENTATION	
	DESTINATION ORDER	Redundancy
	OWNERSHIP	
	OWNERSHIP STRENGTH	Transport
	LIVELINESS	
	LATENCY BUDGET	
	TRANSPORT PRIORITY	

QoS Contract “Request / Offered”

QoS:Latency_Budget
QoS:Ownership
QoS:Liveliness
QoS:Reliability

QoS:Durability
QoS:Presentation
QoS:Deadline
...

Ensure that the compatible QoS parameters are set.



QoS: Quality of Service

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

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

Using QoS to Send Data

- Continuous Data
 - Constantly updating data
 - Many-to-many delivery
 - Sensor data, last value is best
 - Seamless failover
- State Information
 - Occasionally changing persistent data
 - Recipients need latest and greatest
- Alarms & Events
 - Asynchronous messages
 - Need confirmation of delivery

Using QoS to Send Data

- Continuous Data
 - Constantly updating data – **best-effort**
 - Many-to-many delivery – **keys, multicast**
 - Sensor data, last value is best – **keep-last**
 - Seamless failover – **ownership, deadline**
- State Information
 - Occasionally changing persistent data – **durability**
 - Recipients need latest and greatest – **history**
- Alarms & Events
 - Asynchronous messages – **liveliness**
 - Need confirmation of delivery – **reliability**

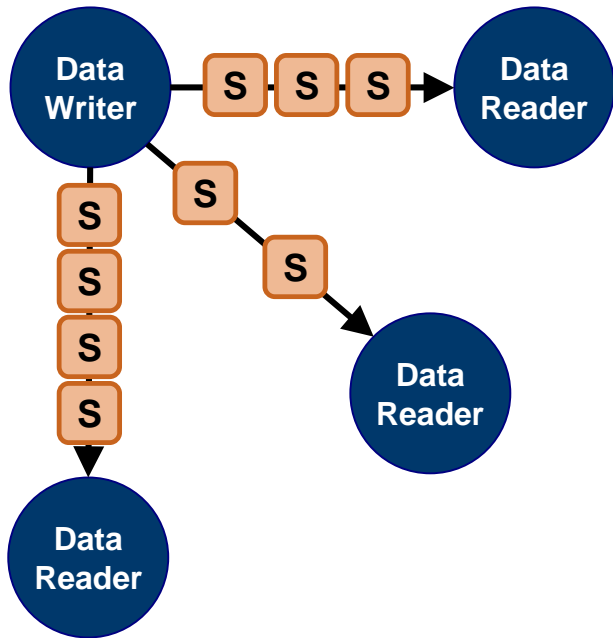
Design Patterns – Continuous Data

- Scenarios
 - Multiple Writers constantly updating the topic “Data”
 - Many-to-many delivery
 - Desired to have separate instances of “Data”
- Example
 - Track updates, vehicle position, stock quotes, temperatures,...

Quality of Service

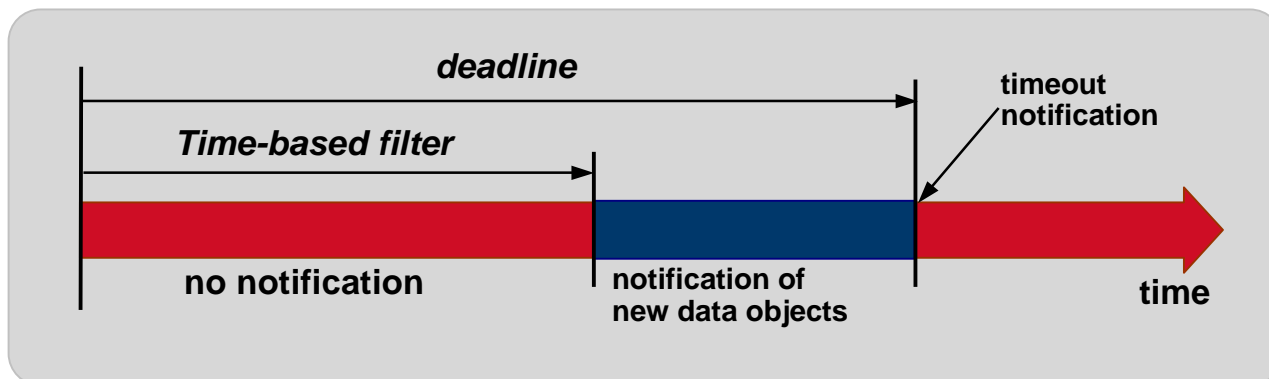
- Best Effort
 - Delivery mechanism can be specified on a per-Reader basis
- Multicast
 - Leverage efficiencies in transports on a per-Reader basis
- Keys
 - Controls scalability, enabling single topic with multiple instances

Best Effort Communications



QoS: Reliability = Best Effort

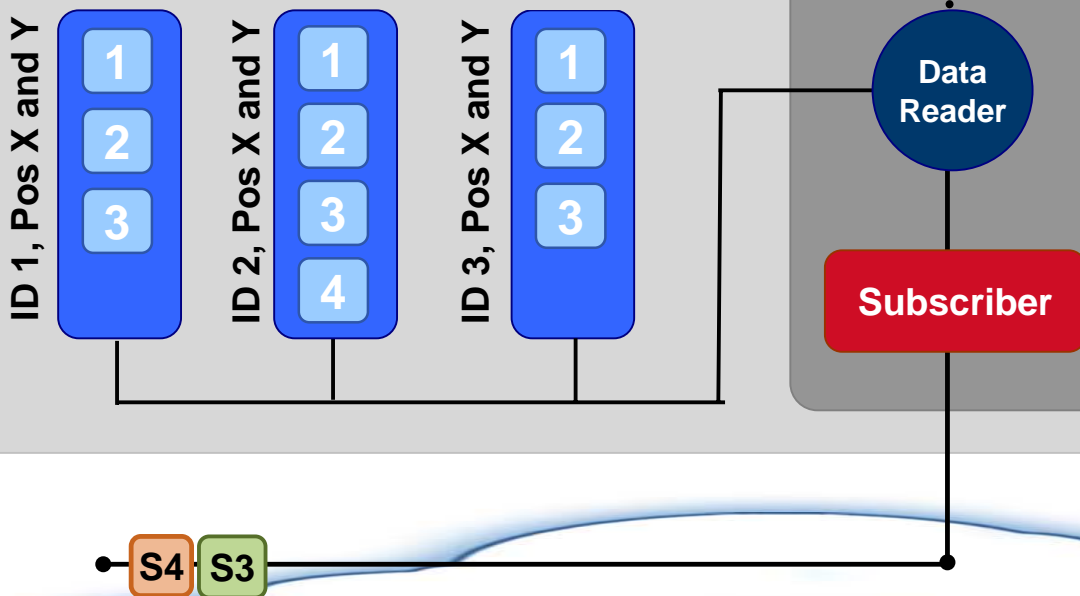
- Low latency and overhead
- Samples are sent without any feedback
- Deadline specifies determinism
- Time-Based Filter controls bandwidth



Keys (data-object scalability)

- Multiple instances of the same topic

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



- Topic key can be any data-type within the Topic.

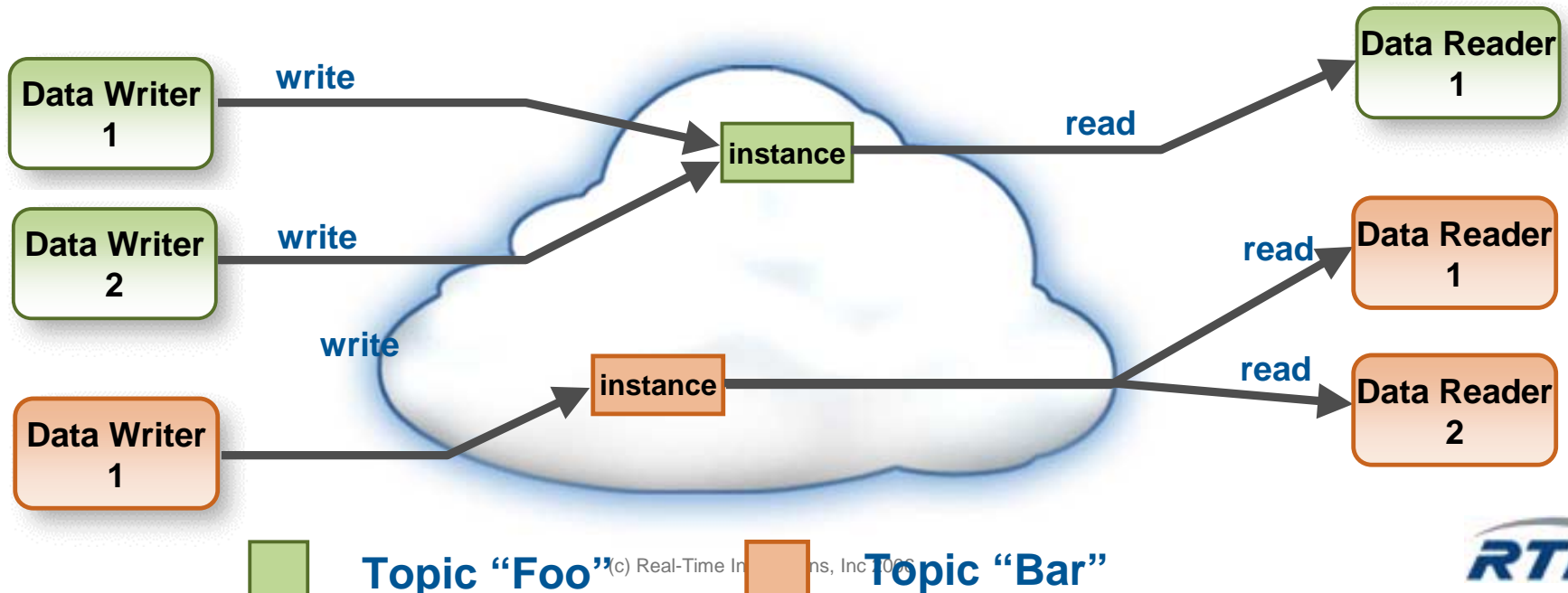
Example:

```
long ID //@key  
float temperature;  
long status;
```


Global Data Space

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.

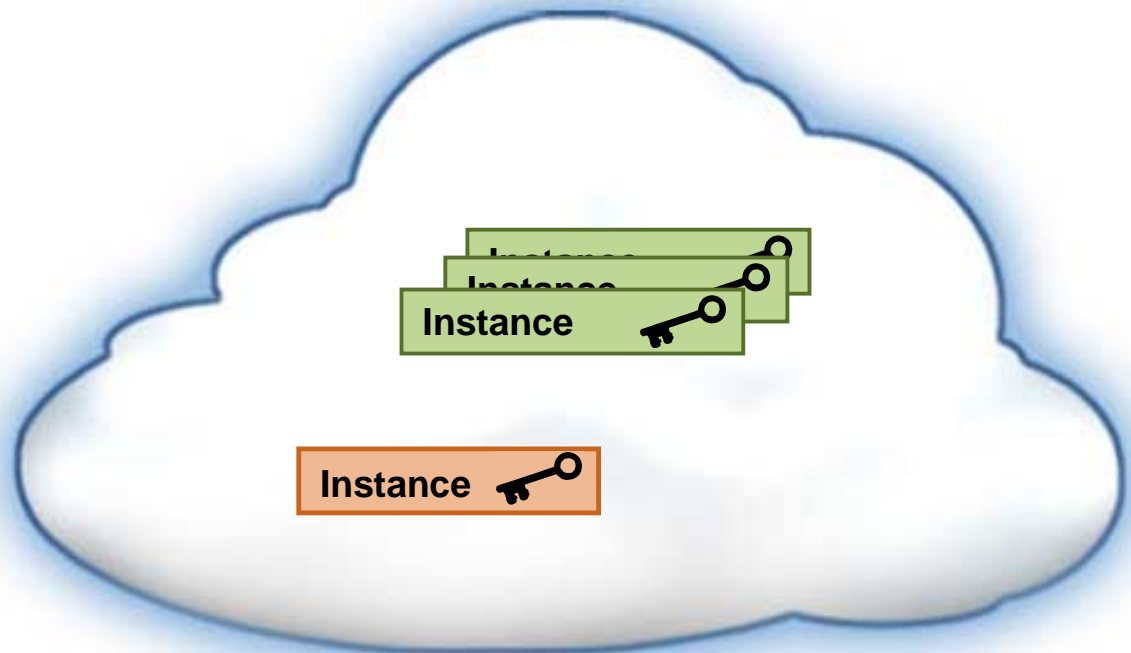


Example with Keys

- When using keys:
 - The system may contain multiple instances of a given topic, with each instance uniquely identified by a key. 

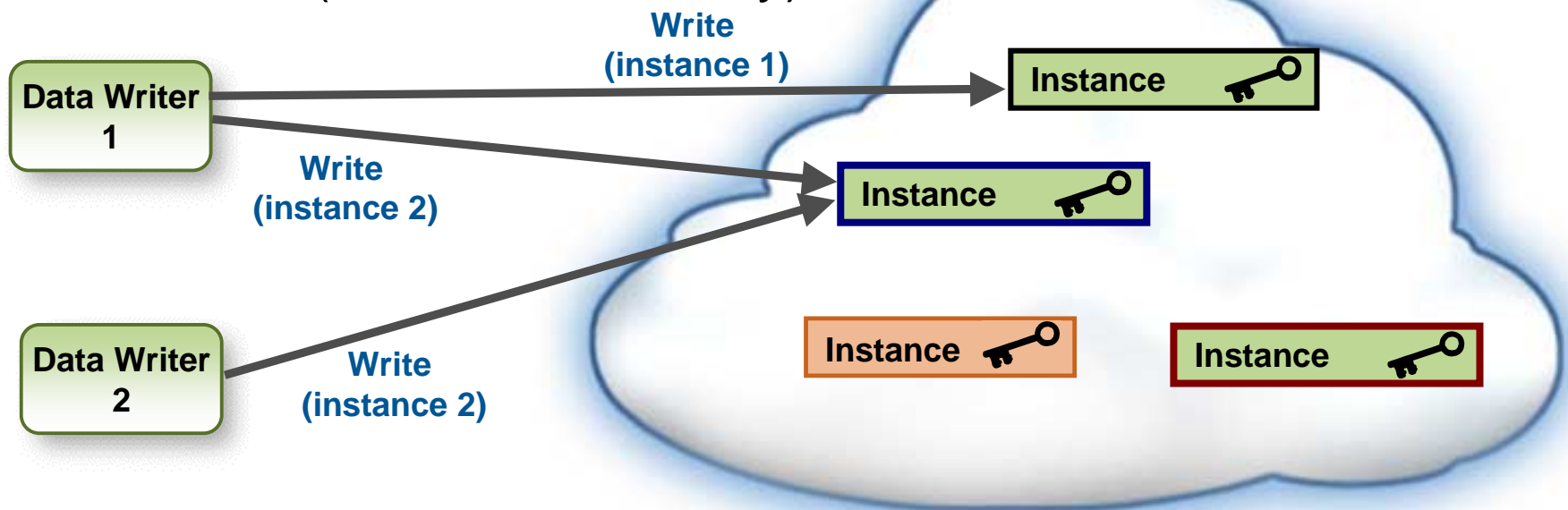
 Topic “Foo”

 Topic “Bar”



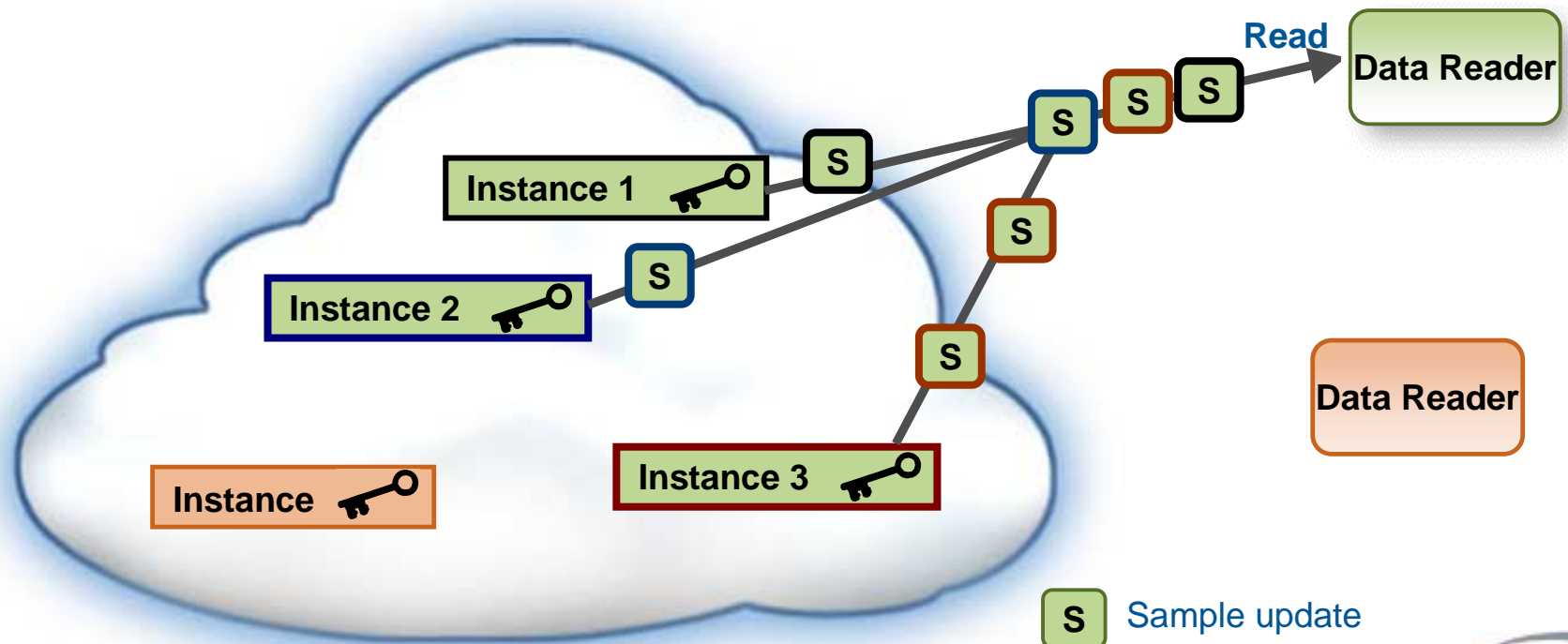
Example with keys (DataWriter)

- When using keys:
 - Each DW can write to multiple instances of a single topic
 - Multiple DWs may write to same instance (i.e. with same key)



Example with keys (DataReader)

- When using keys:
 - Each DR can receive updates from multiple instances of a single topic
 - Multiple DRs may read from the same instances



Design Patterns – Continuous Data

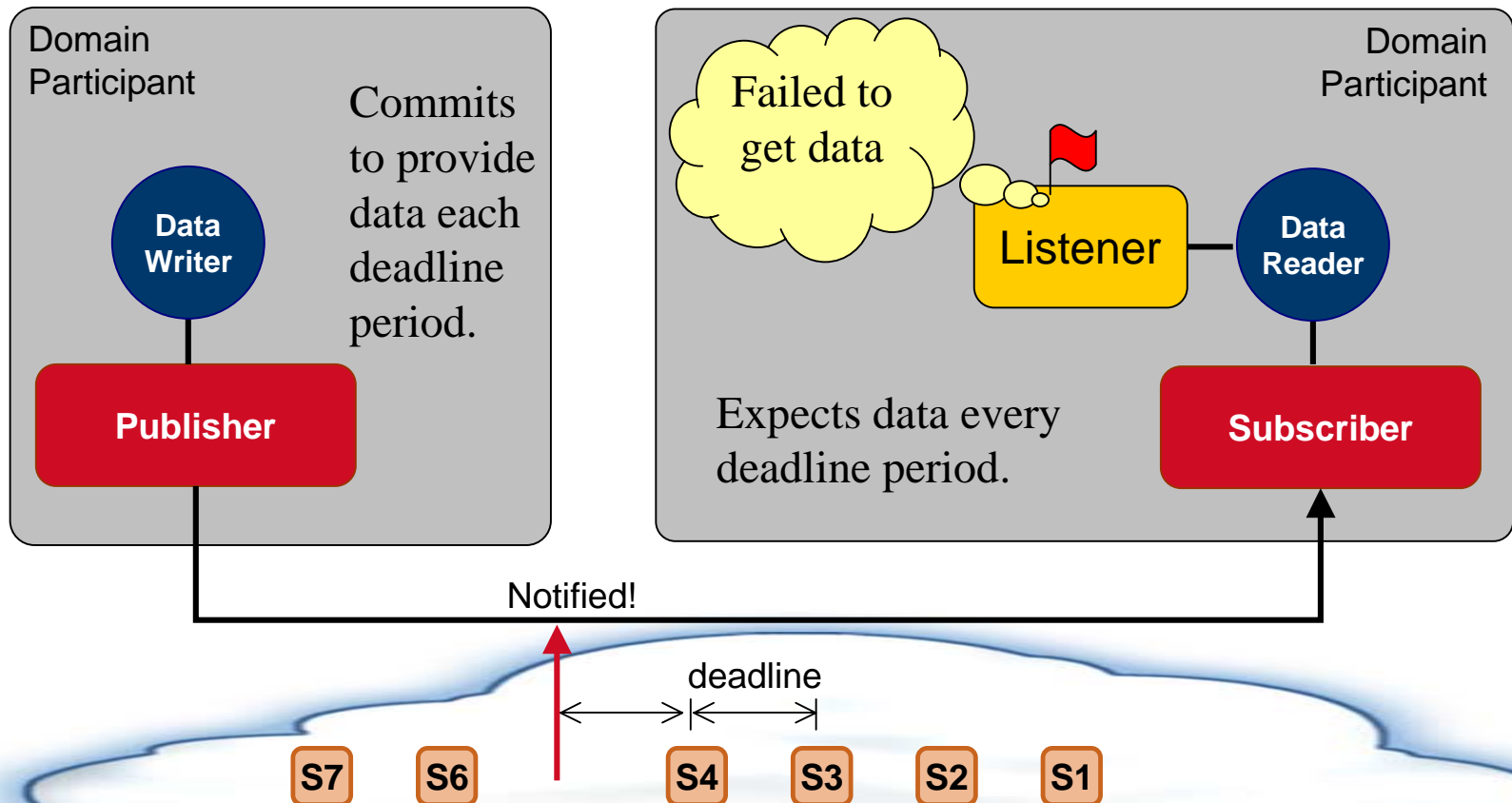
- Scenarios (Part 2)
 - A DataReader need to know when a writer or instance ‘fails’
 - Some DataReaders need to limit their updates
 - Only one DataWriter can update a specific instance
- Examples
 - Two radars monitoring the same Track, multiple sources of weather data, manual override of UAV flight control,...
 - High level display doesn’t need all real-time data

Quality of Service

- Deadline
 - Can indicate health and send/receive time requirements
- Time Based Filter
 - Readers can control their own data delivery rate
- Ownership
 - Controls exclusivity of writing to an instance

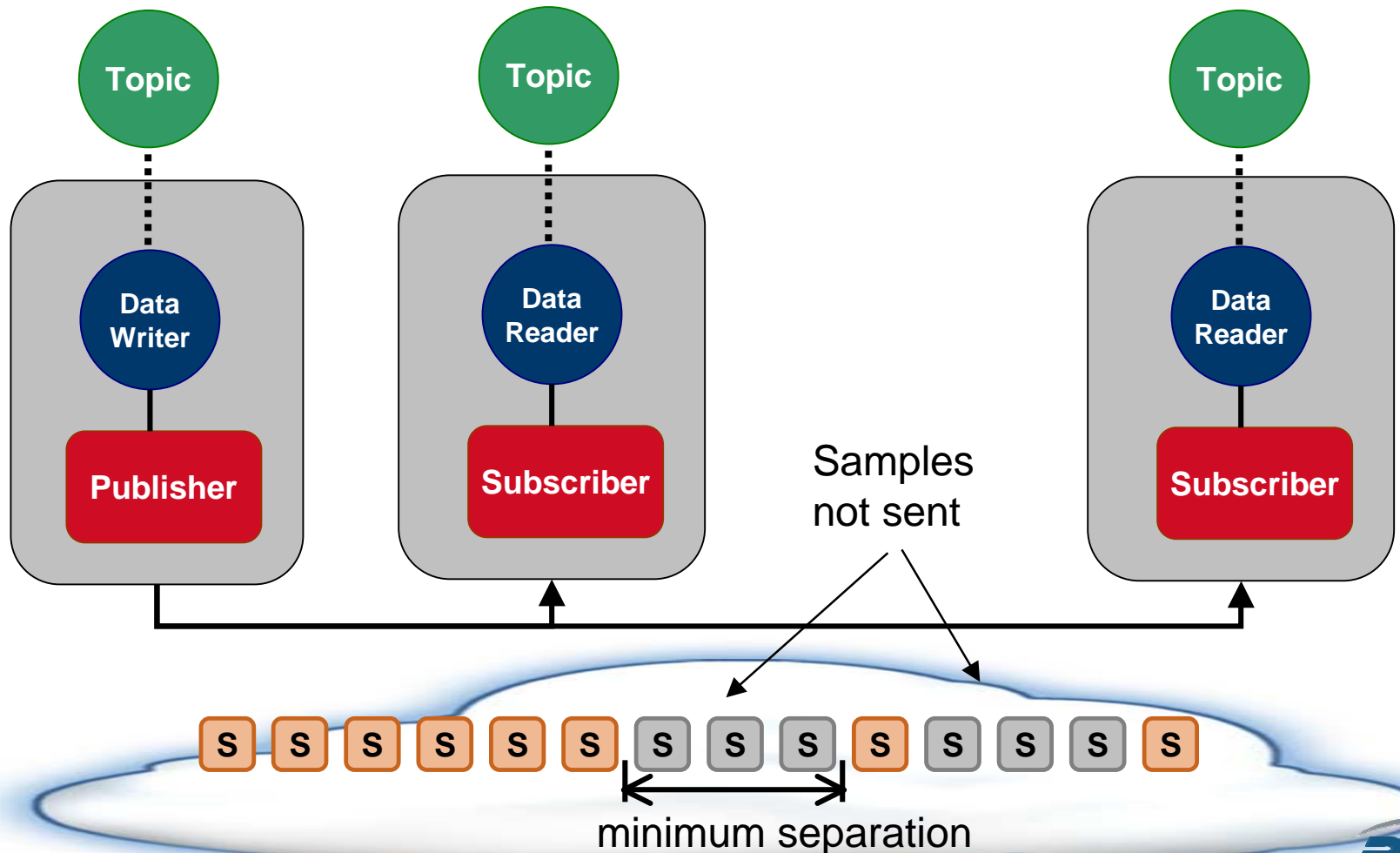
QoS: Deadline

Specified as a “period”, following request/offered design pattern



QoS: Time Based Filter

“minimum_separation”: DataReader does not want to receive data faster than the min_separation time

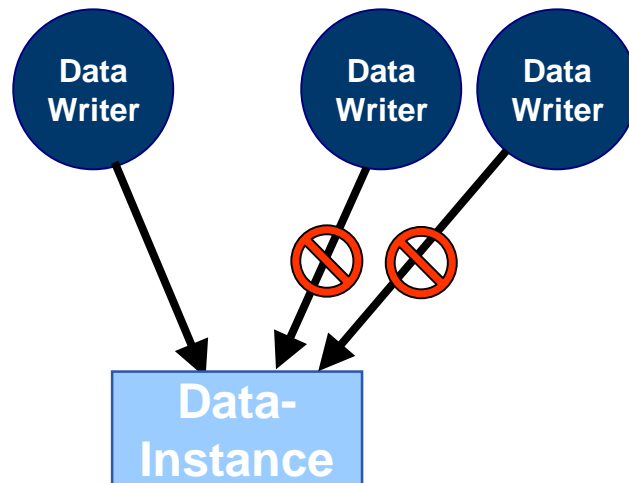


QoS: Ownership

Specifies whether more than one DataWriter can update the same instance of a data-object

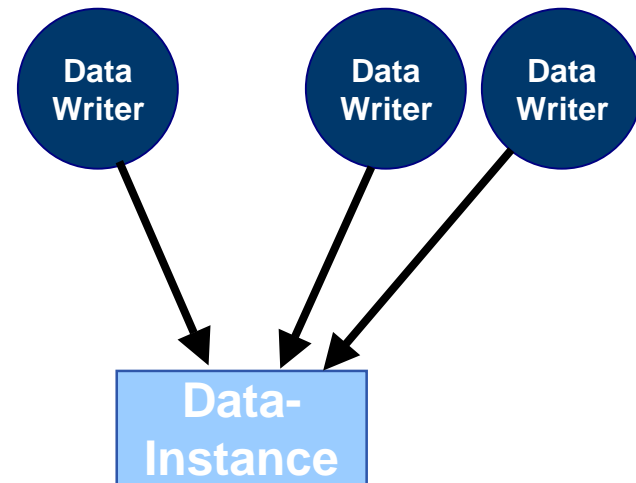
Ownership = EXCLUSIVE

“Only highest-strength data writer can update each data-instance”



Ownership = SHARED

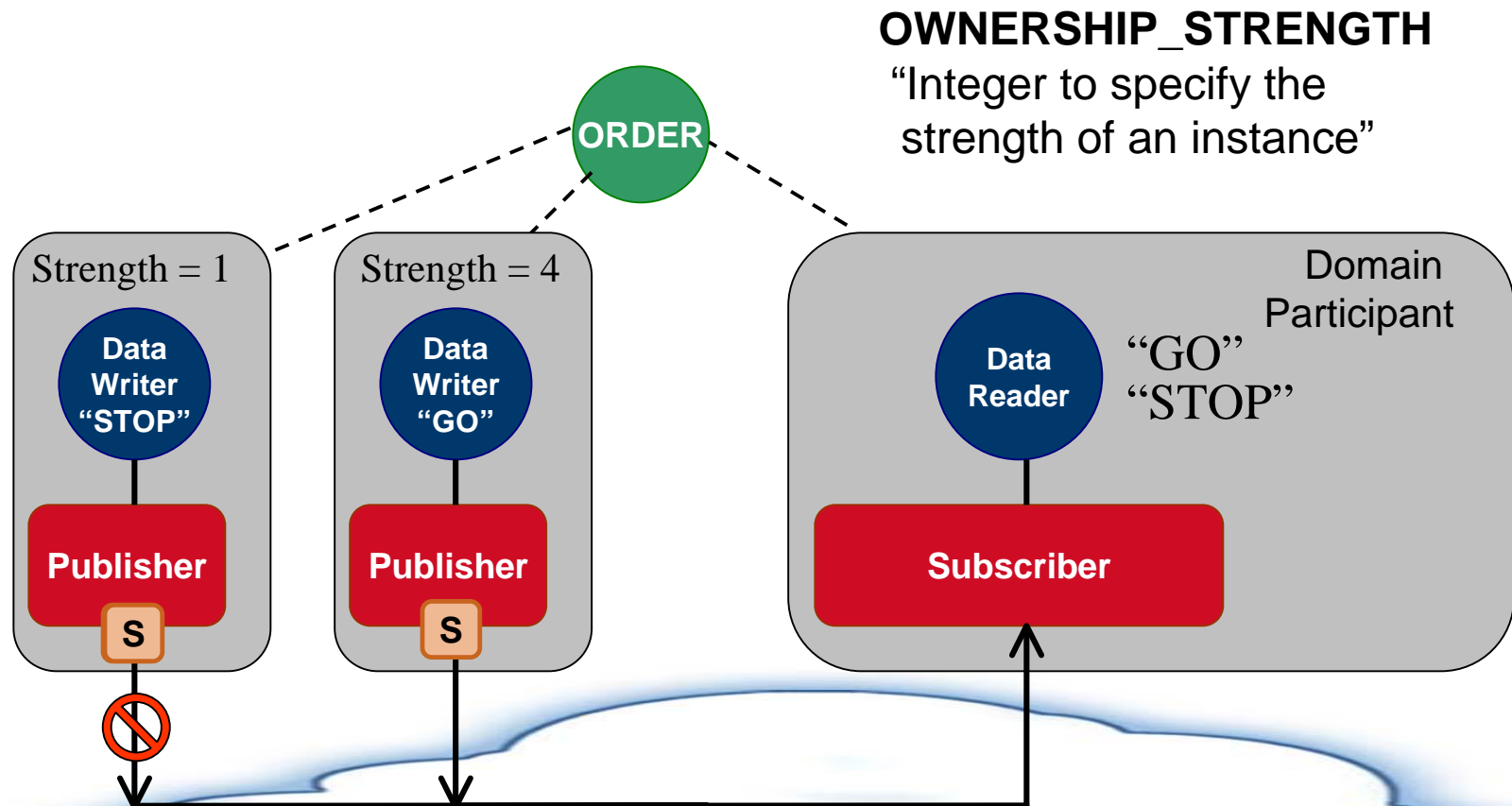
“All data-writers can each update data-instance”



Provides fast, robust, transparent replacement for fail-over and/or take-over.

QoS: Ownership Strength

Specifies which DataWriter is allowed to update the values of data-objects



Note: Only applies to Topics with Ownership = Exclusive

Design Patterns – State Information

- Scenarios
 - One DataWriter of “Data”, multiple DataReaders
 - Late joining DataReaders should get last set of “Data”
 - Every DataReader should get all issued data
- Examples
 - Arrival and departure time for trains, current aircraft flight plans, operational system configuration and mode

Quality of Service

- History (Keep Last)
 - Controls how much data need to be available
- Reliability
 - Ensures that data is delivered
- Durability & Lifespan
 - Controls persistence, how and where data is stored

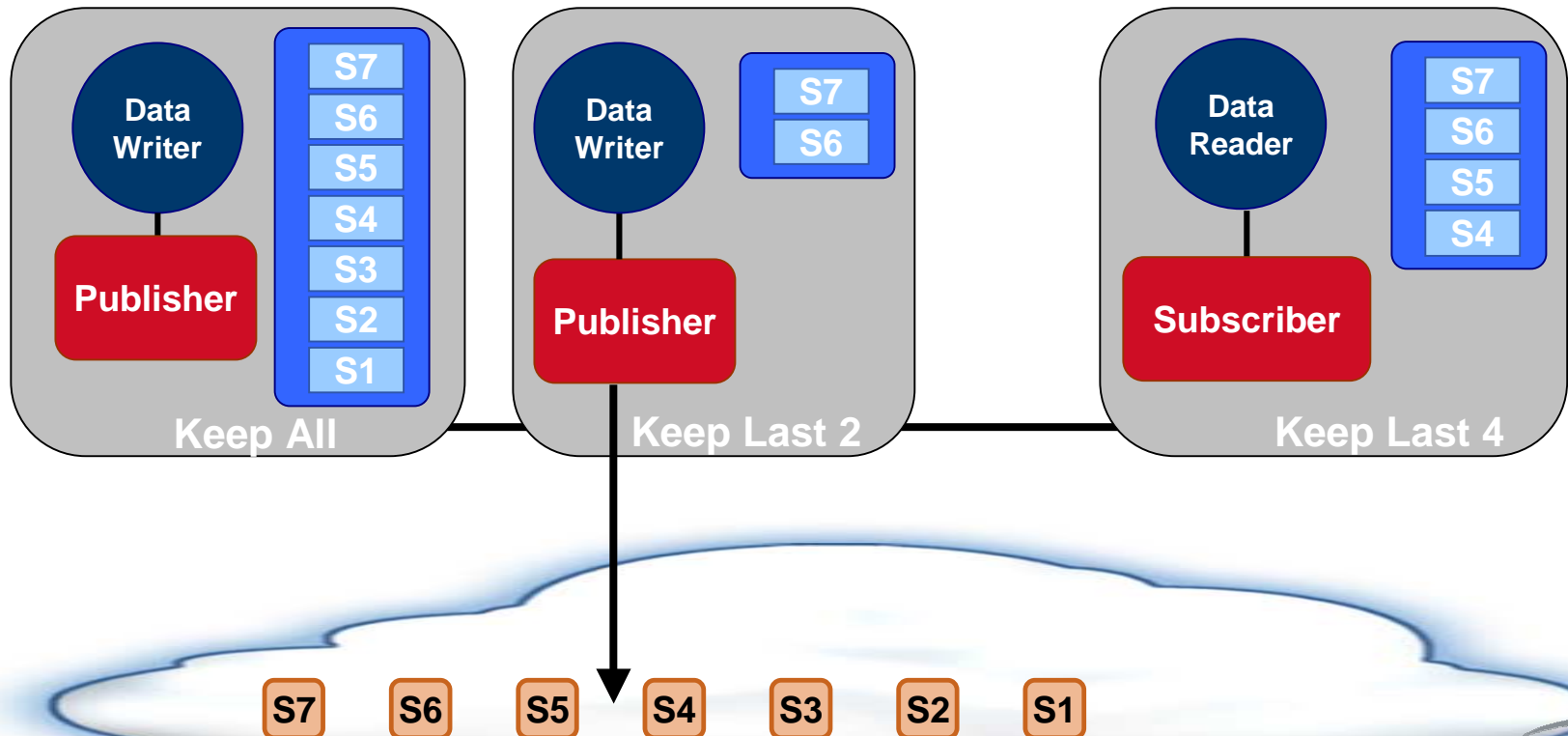
QoS: History – Last x or All

KEEP_ALL:

Publisher: keep all until delivered

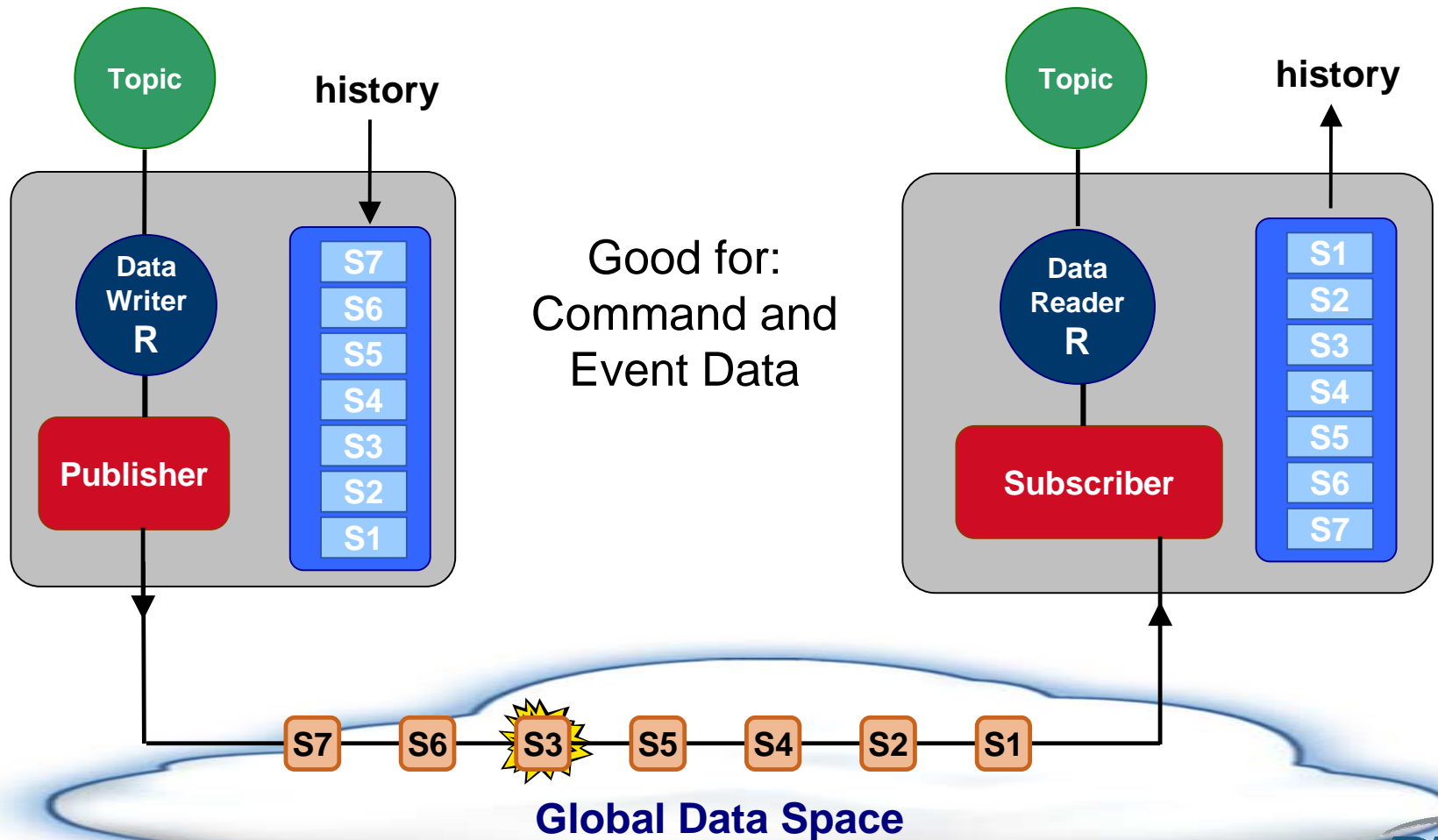
Subscriber: keep each sample until the application processes that instance

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



Reliable Communications

Reliability QoS: Ordered Instance delivery is guaranteed

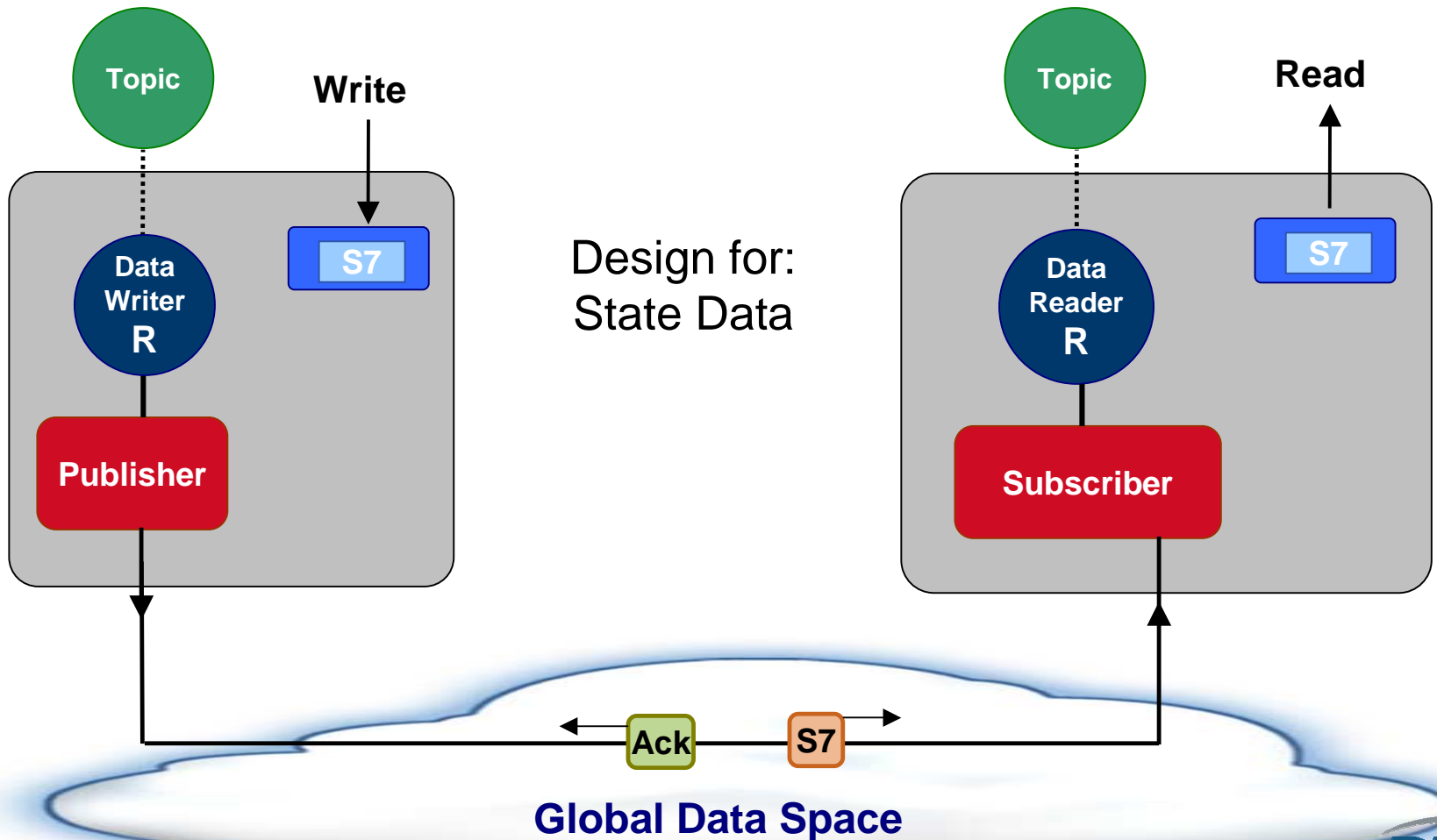


“Just the Right Amount” of Reliability

- Not all Scenarios are the same
 - Streaming high-rate data, many Readers
 - Asynchronous, infrequent critical commands
 - Bursty and low/high amounts of data
 - Unicast or Multicast delivery
- Example QoS
 - HISTORY: KEEP_ALL or KEEP_LAST
 - max_blocking_time: Writer queue full?
 - Nobody gets the data verses someone loses data
 - RTPS properties: how/when to HB

Reliable Communications

Reliability QoS: Ordered Instance delivery is guaranteed



QoS: Durability

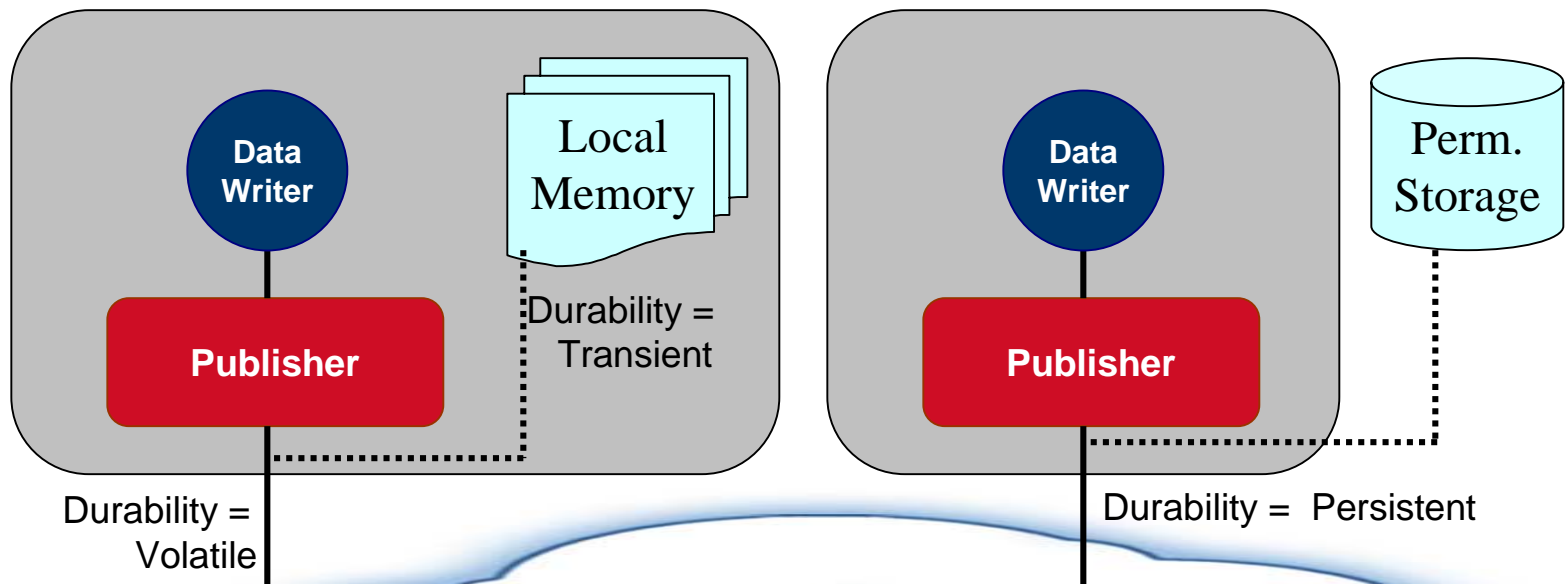
Durability Kind:

VOLATILE – No Instance History Saved

TRANSIENT – History Saved in Local Memory

PERSISTENT – History Saved in Permanent storage

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

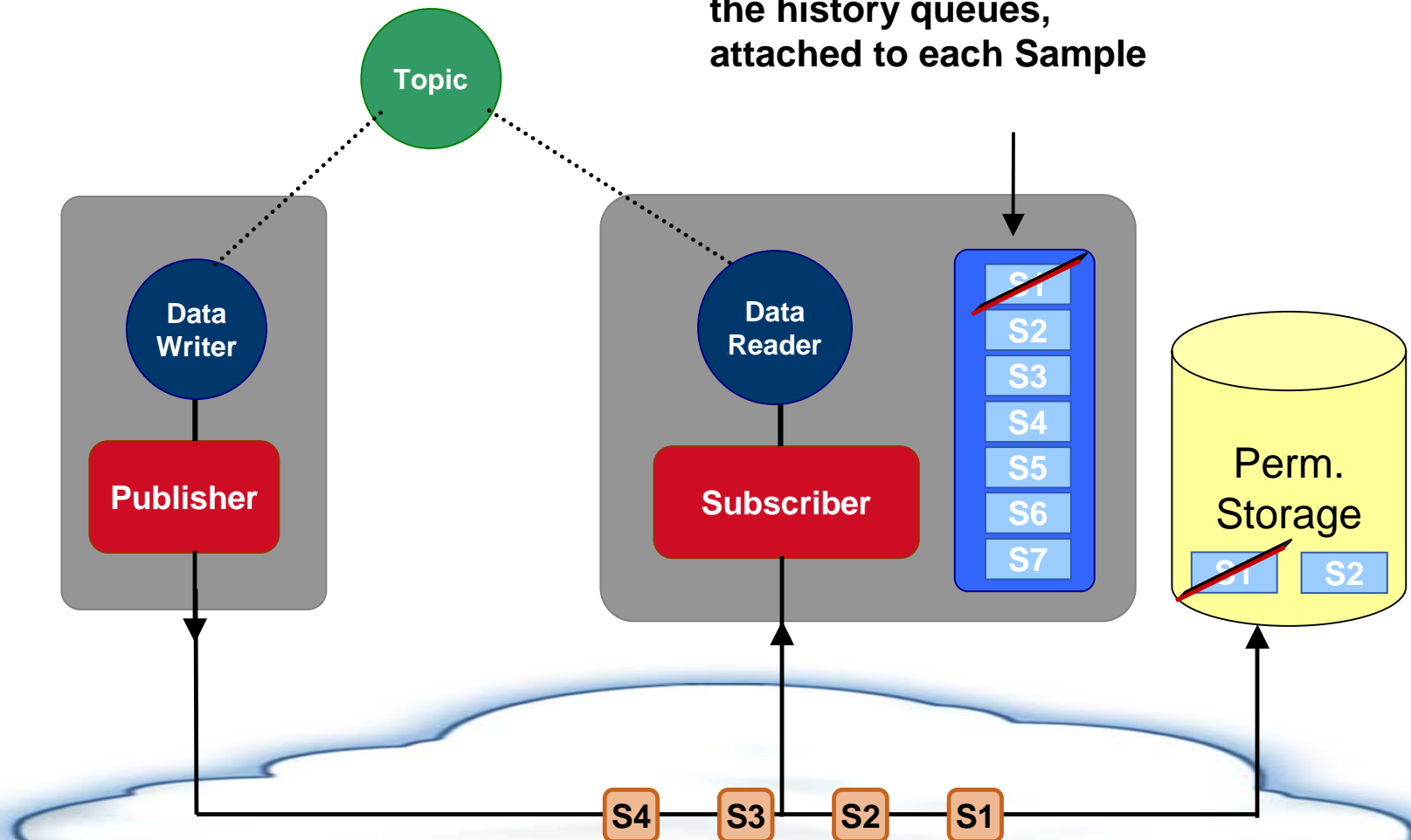


saved in Transient affected by QoS:
History and QoS: Resource_Limits

QoS: Lifespan

*User can set **duration***

Manages samples in the history queues, attached to each Sample



“How Much and Where” to Persist

- More about the HISTORY QoS Setting
 - Synchronous data is simply a function of the rate
 - Asynchronous?
- If DURABILITY kind is *persistent*
 - DURABILITY_SERVICE QoS helps the DataWriter tell the service how to operate
- LIFESPAN can bound needed resources for durable data

Design Patterns – Alarms & Events

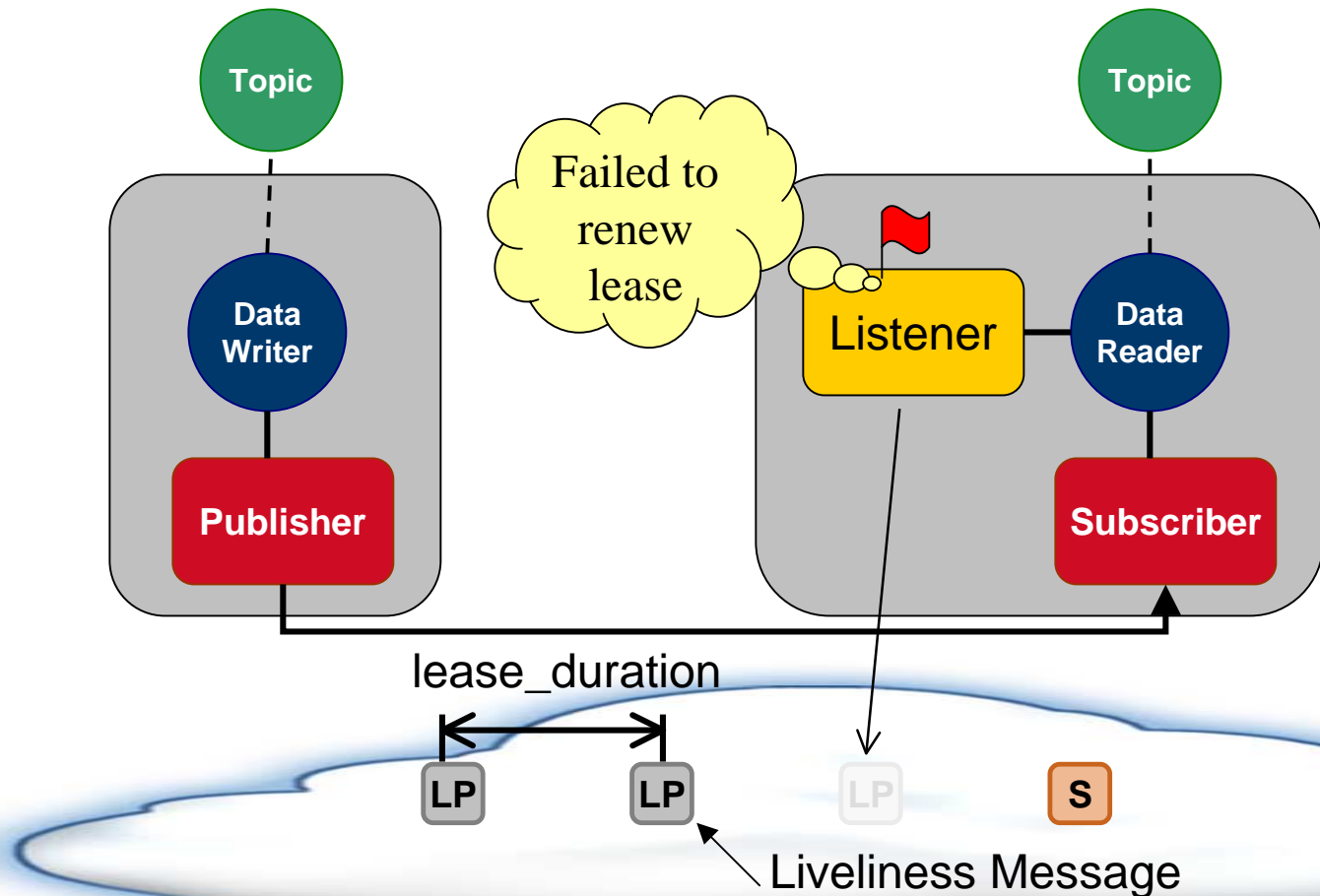
- Scenarios
 - All issues of data need to be received
 - Asynchronous messages
 - Possible only one authorized 'Event' Writer
- Examples
 - Emergency notifications, boiler over heating alarm, calling the elevator, smoke detectors,...

Quality of Service

- Liveliness
 - Need to know if a DataWriter is present independent of receipt of data
- Reliability, History (Keep All)
- Ownership
- Lifespan

QoS: Liveliness – Type and Duration

Type: Controls who is responsible for issues of 'liveliness packets'
AUTOMATIC = Infrastructure Managed
MANUAL = Application Managed



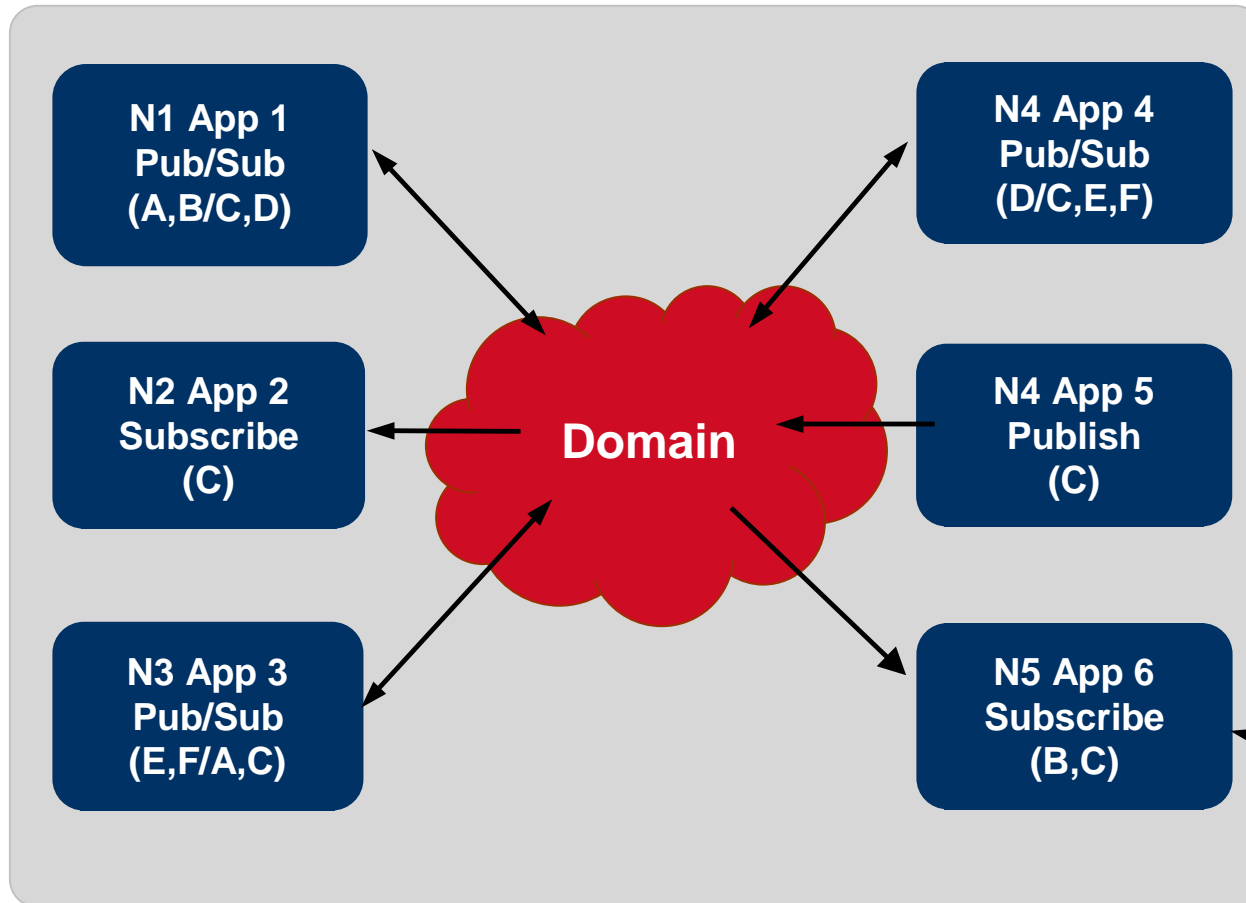
Design Patterns – Architecture

- Discovery
 - Control the scope of the system
 - Sub-modules may be using same topic names
 - Manage statically or dynamically who finds whom
- Authentication
 - Not all Writers of a specific topic authorized
 - Additional Attribute data distributed with a Topic
- Data Selection
 - Receive only a subset of a Topic's instances

Design Patterns – Architecture

- Discovery – domains, partitions
 - Control the scope of the system
 - Sub-modules may be using same topic names
 - Manage statically or dynamically who finds whom
- Authentication – user-data, built-in topics
 - Not all Writers of a specific topic authorized
 - Additional Attribute data distributed with a Topic
- Data Selection – read/take semantics
 - Receive only a subset of a Topic's instances

Domain and Domain Participants

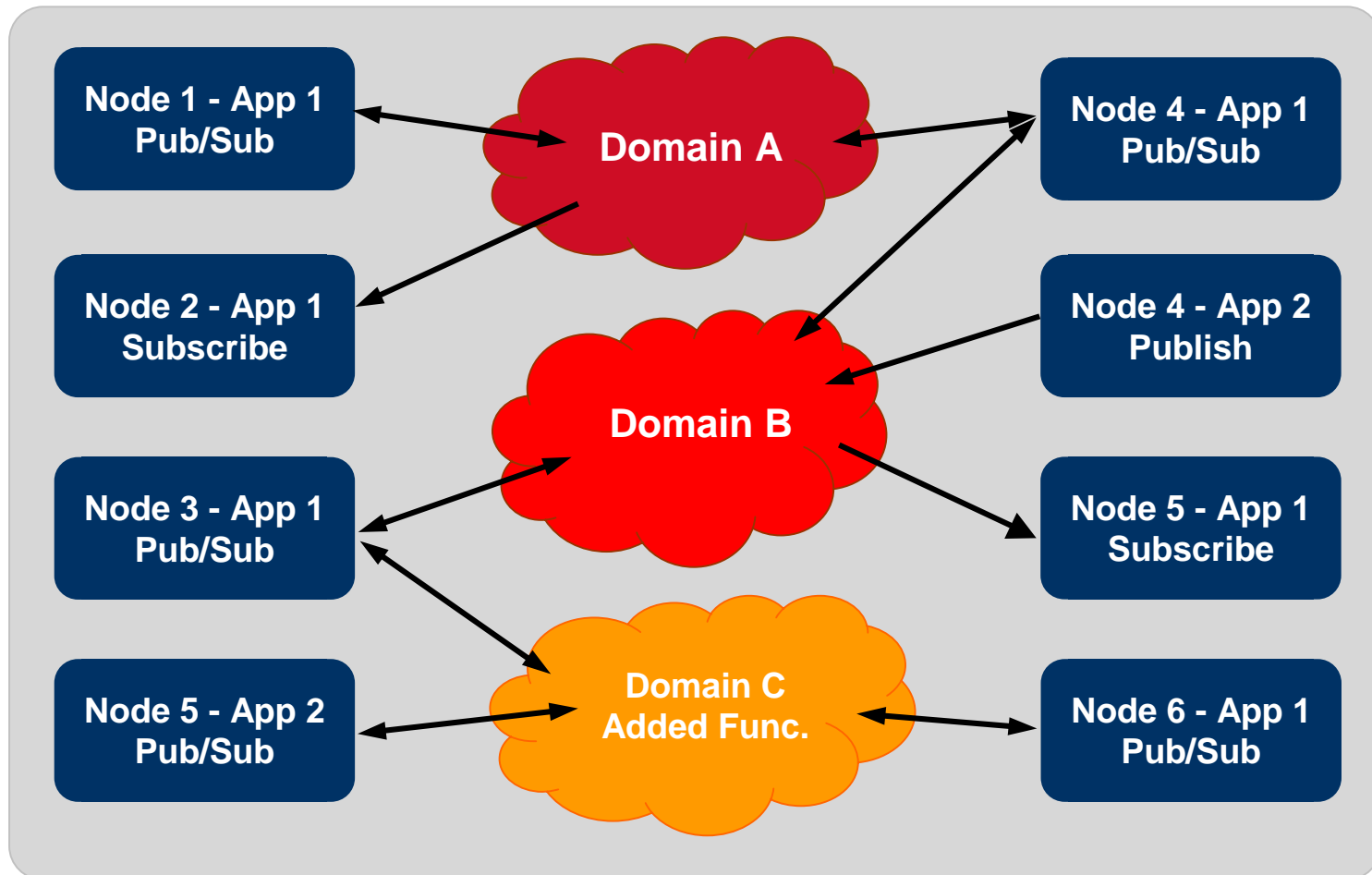


- Container for applications that want to communicate
- Applications can join or leave a domain in any order
- New Applications are “Auto-Discovered”
- An application that has joined a domain is also called a “Domain Participant”

Single ‘Domain’ System

Domain and Domain Participants

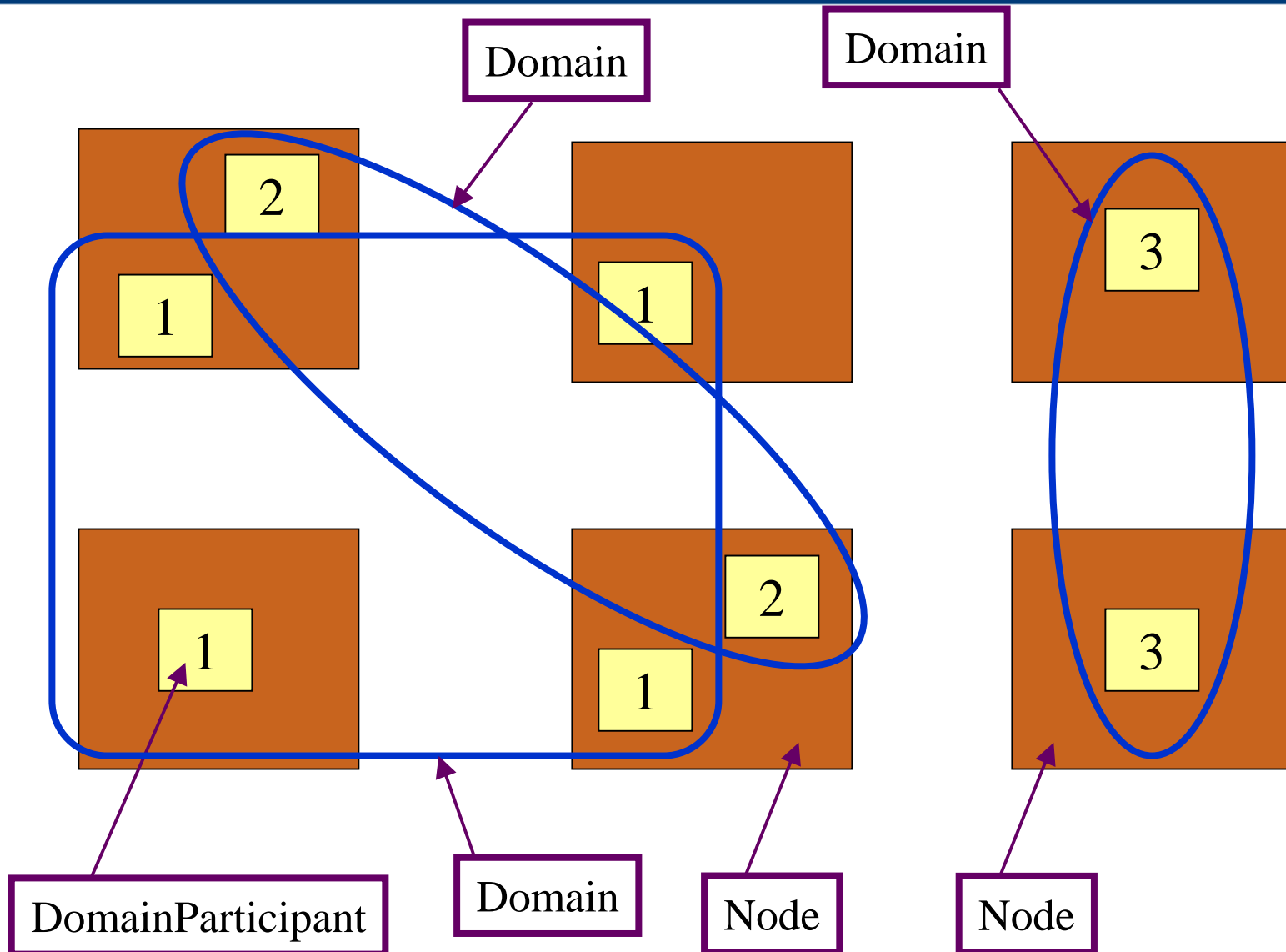
Using Multiple domains for Scalability, Modularity & Isolation



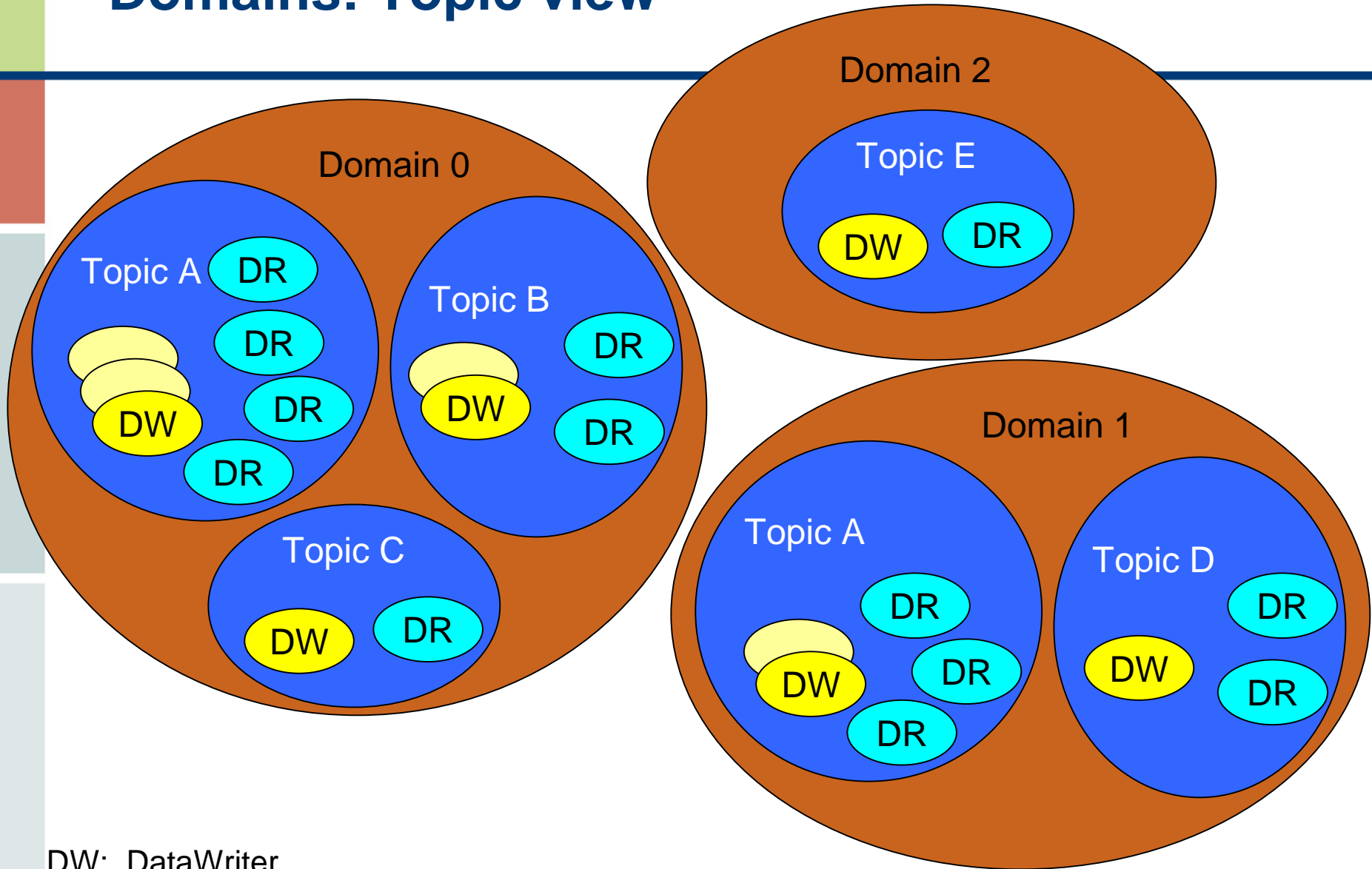
Multiple Domain System

(c) Real-Time Innovations, Inc 2006

Domains and Participants



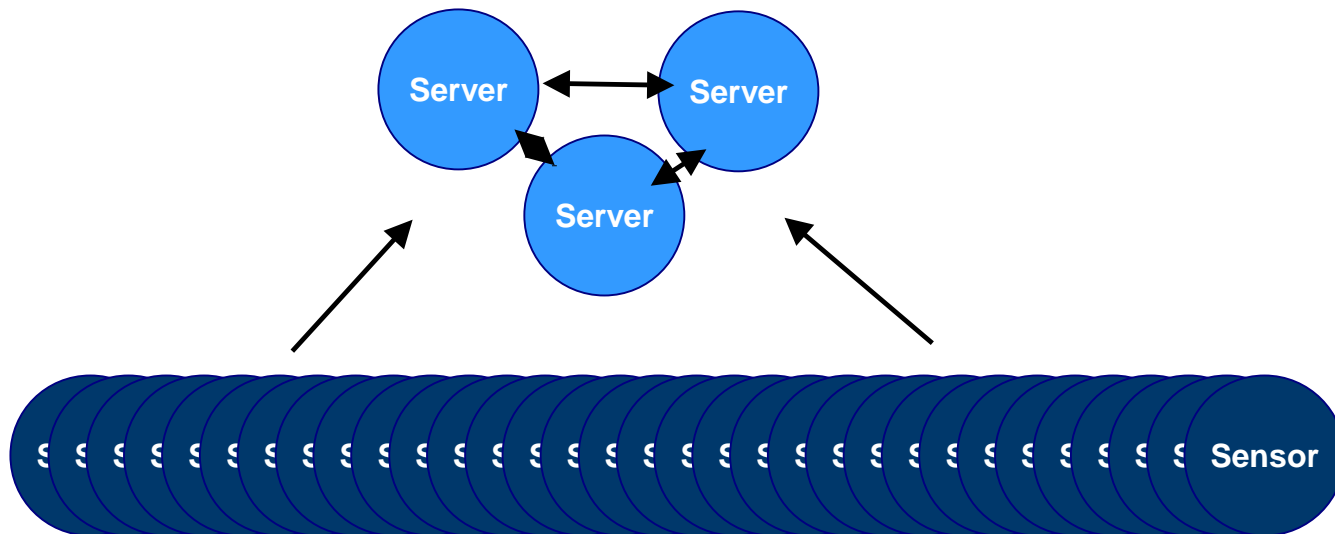
Domains: Topic view



DW: DataWriter
DR: DataReader

“Managing the Scope” of a Domain

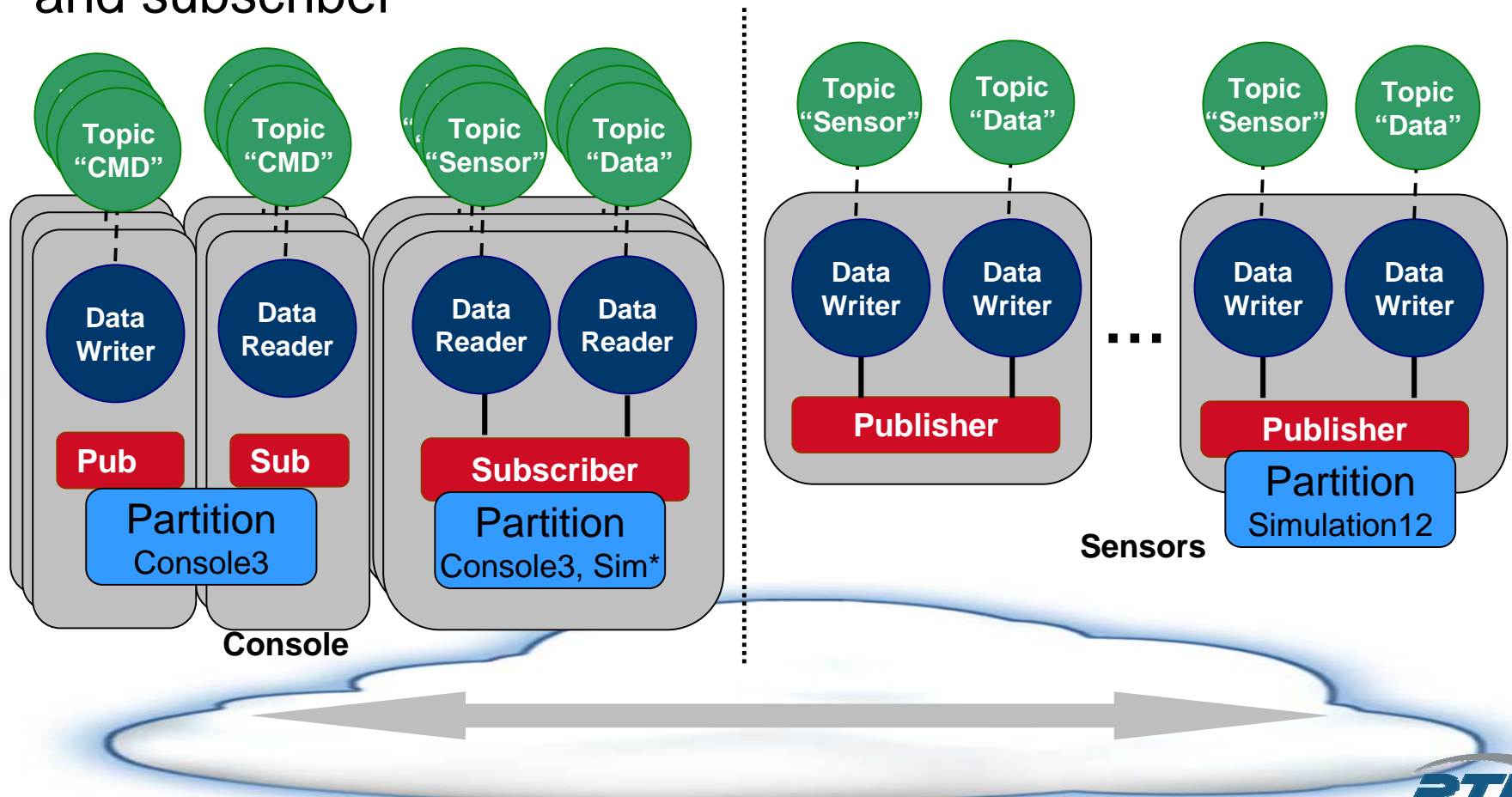
- Participants don't need to be symmetric
- Recent adoption of RTPS protocol allows
 - Specification of 'peers'
 - Multicast receive addresses



QoS: Partition

Partitions are a logical “namespace” for topics

****** Partition string names must match between publisher and subscriber



Design Patterns – Controlled Data Access

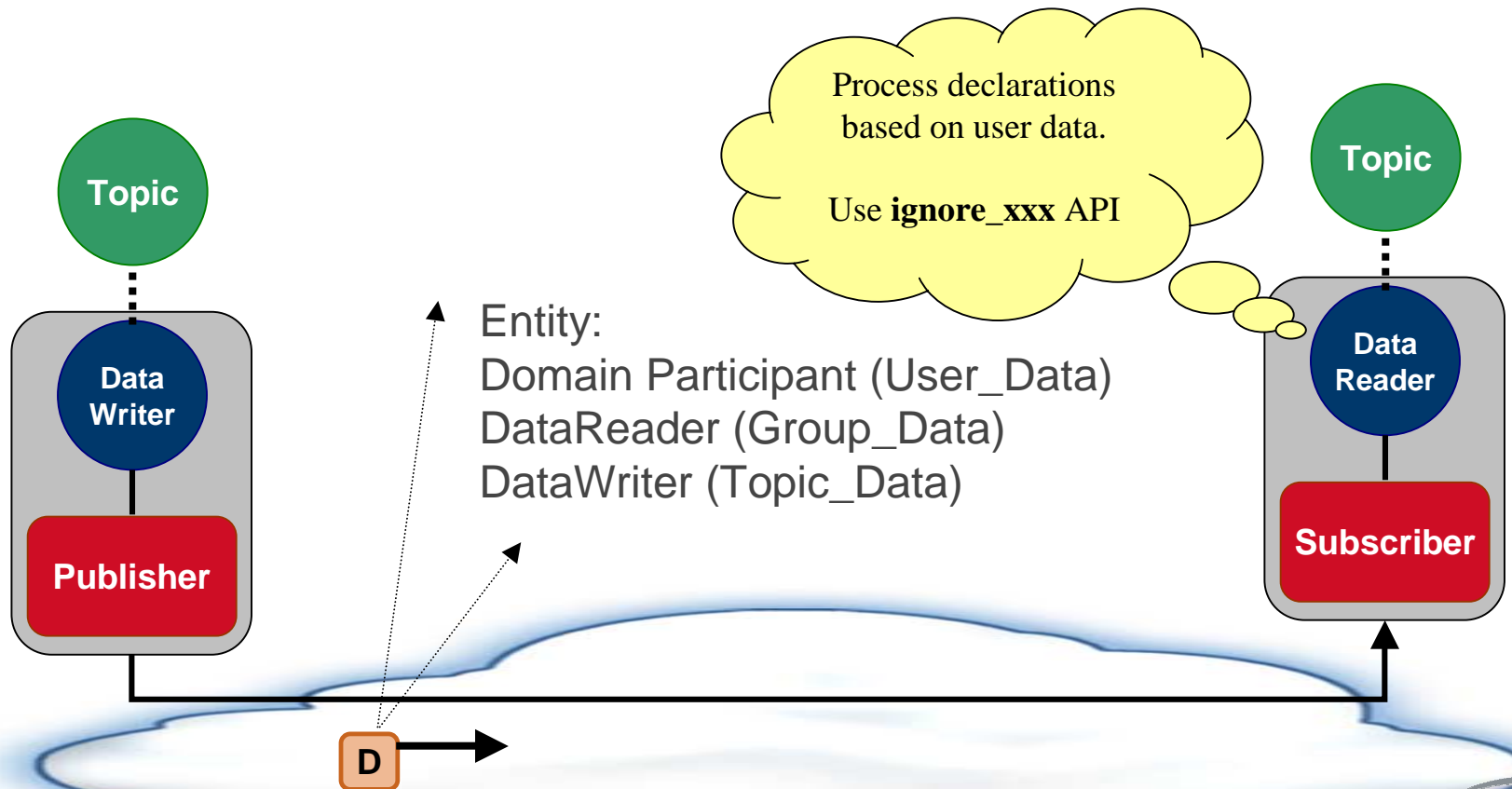
- Scenarios
 - Limited access to certain data by node or application or entity or user ID,...
 - Exchange meta-data for authentication and identification
 - Execution of identically configured systems
- Examples
 - Command authorization, configuration control, Separation of simulation and operational data
 - Separation of simulation and operational data
 - Isolation of data from multiple system instances (e.g. each tank will manage and see its own data, despite using the same Topics)

Quality of Service

- Domains
 - A separate global data space for each Domain
- Partitions
 - Dynamic association of Readers and Writers (grouping)
- User Data
 - Exchange Reader and Writer meta-data and with Discovery
- Ignore API
 - Permanent denial of read/write access

QoS: User Data

- USER_DATA is contained within the DDS metadata.
- User data could be used to authenticate an origination entity or pass additional attribute information



Design Patterns – Hot-swap and failover

- Scenarios
 - Primary / Secondary are duplicate machines
 - Running duplicate applications
 - Primary handles full load until failure
 - Need to selectively override Topics
- Examples
 - Flight systems, high reliability systems, back-up data center for 911 call centers, know that the train operator is still awake, ...

Quality of Service

- Liveliness
 - Need to know if a DataWriter is present independent of receipt of data
- Deadline
 - Need to refresh every instance continuously
- Ownership
 - Switch primary/secondary data sources

Design Patterns – Filter by Data Content

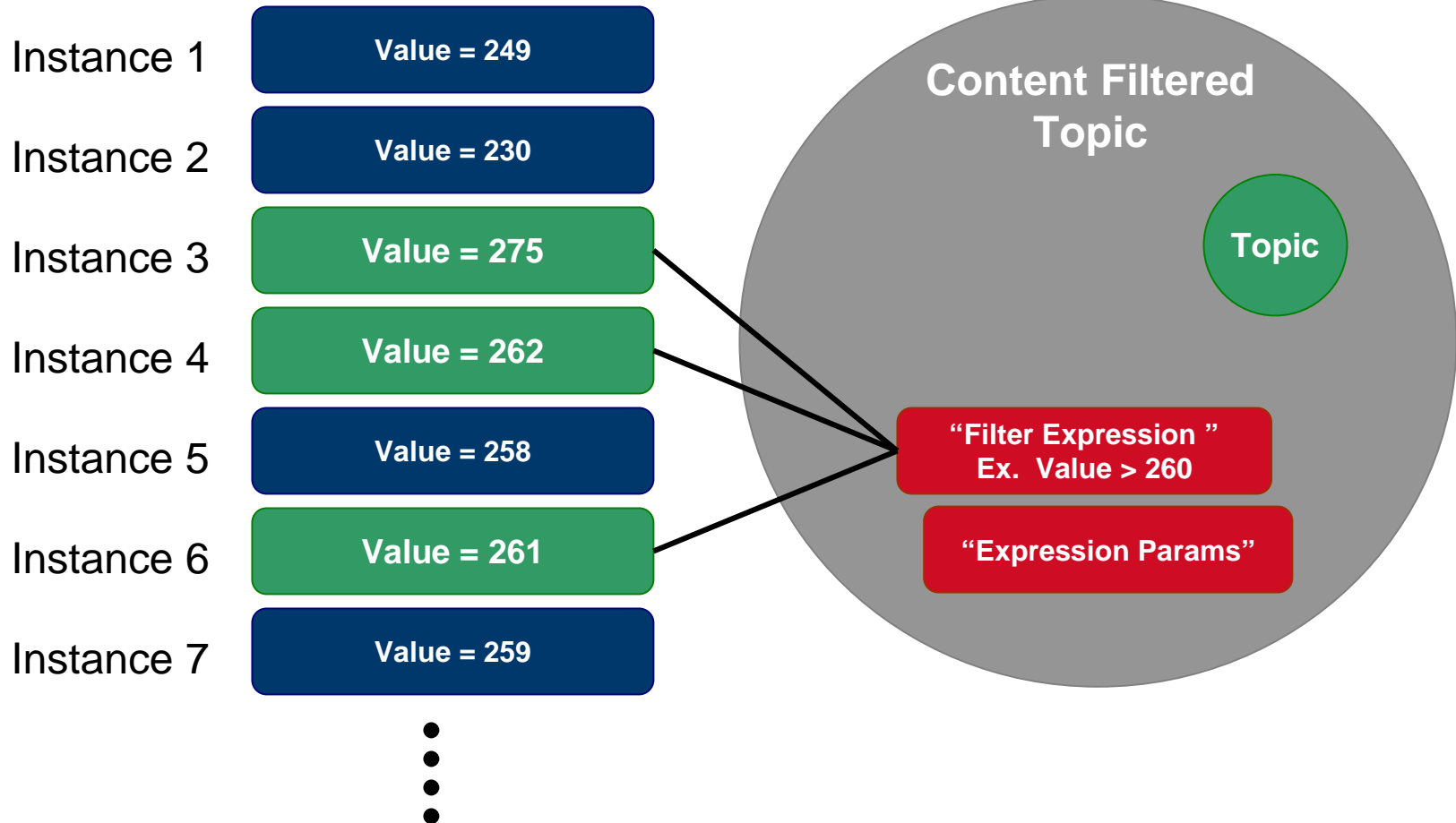
- Scenarios
 - Receive only ‘interesting’ Data
 - Send a message to a specific receiver(s)
- Examples
 - Monitor aircraft within region of interest, display only hostile tracks, monitor objects near my current ship location,....
 - Response to authorization gas pump sale request

Quality of Service/API

- Content Filtered Topics
 - SQL filter language with changeable filter parameters
 - Encode receiverID in message send
- read/take() with a QueryCondition
 - By specific instance or lifecycle states

Content Filtered Topics

Topic Instances in Domain



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

Filter by Data “Context”

- Order of the written samples is important
 - Instance 12, 3, 7, 12, 4
 - QoS: PRESENTATION scope
- Time of the written samples is important
 - DataWriter 2’s sample was written after DataWriter 3
 - QoS: DESTINATION_ORDER
- The set of data is important
 - Must get all instance updates or none
 - QoS: PRESENTATION coherent changes

Putting it All Together

Scenario: Fire-control radar system, gets updates from multiple sensors up to 500x/sec

- Requirements

- Radars

- Multiple radars, may track same objects, but some higher performance/accuracy than others

- Fire control

- Needs every possible update quickly; last is best
- Needs to know when a track is lost

- Display console

- Can only display 10Hz updates

- Logger

- Must record all information in a database for later analysis

- Quality of Service Settings

- Radars

- Key for each track
- Ownership= Exclusive; Assign strength by accuracy
 - (each object tracked by best radar that sees it)
- Publishers offer reliability

- Fire Control

- Maximize determinism
- Best efforts; get every sample
- Use key state/deadline to know when objects lost

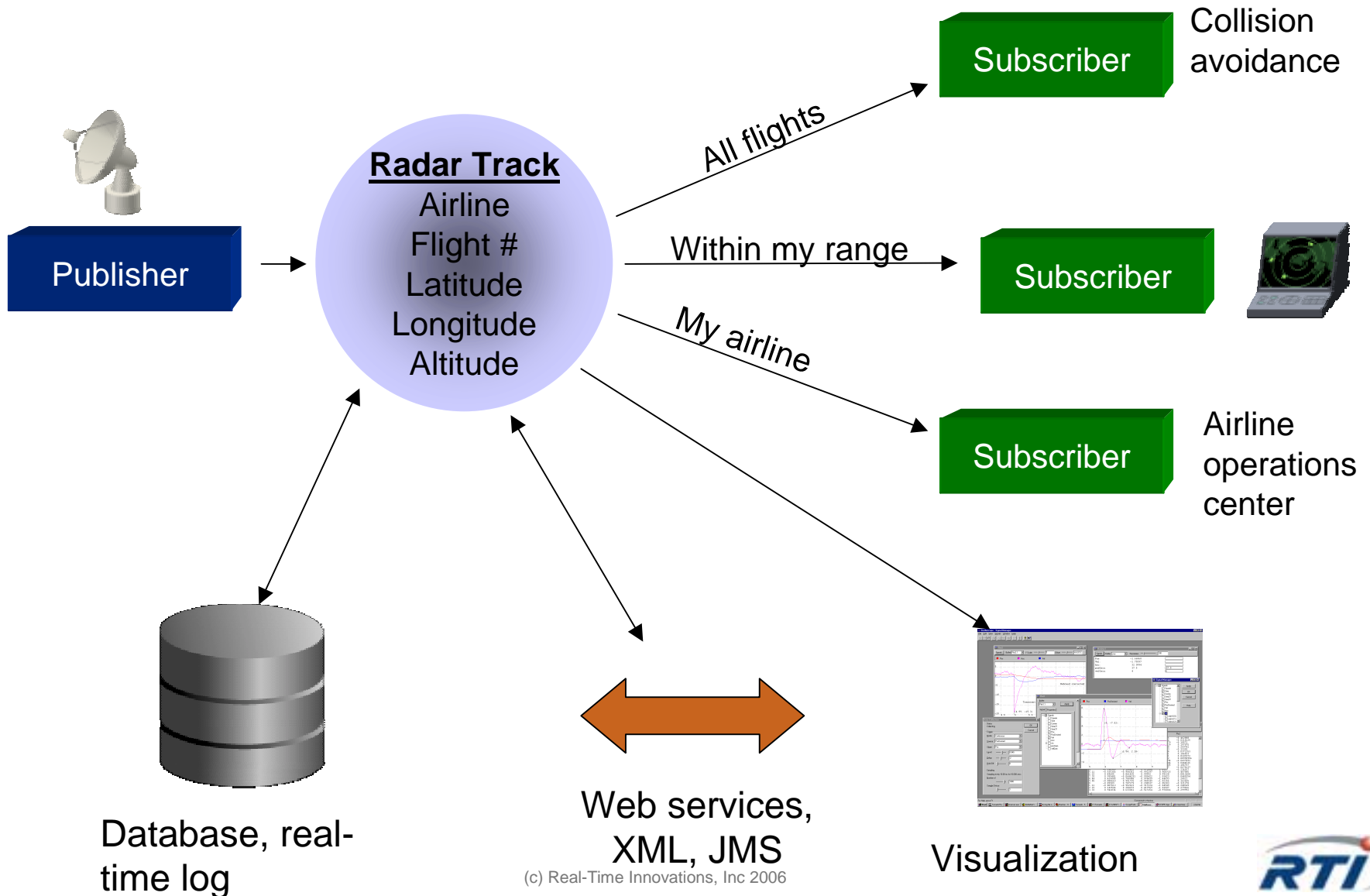
- Display console

- Don't waste bandwidth
- Best efforts
- Time-based filter at 0.1sec

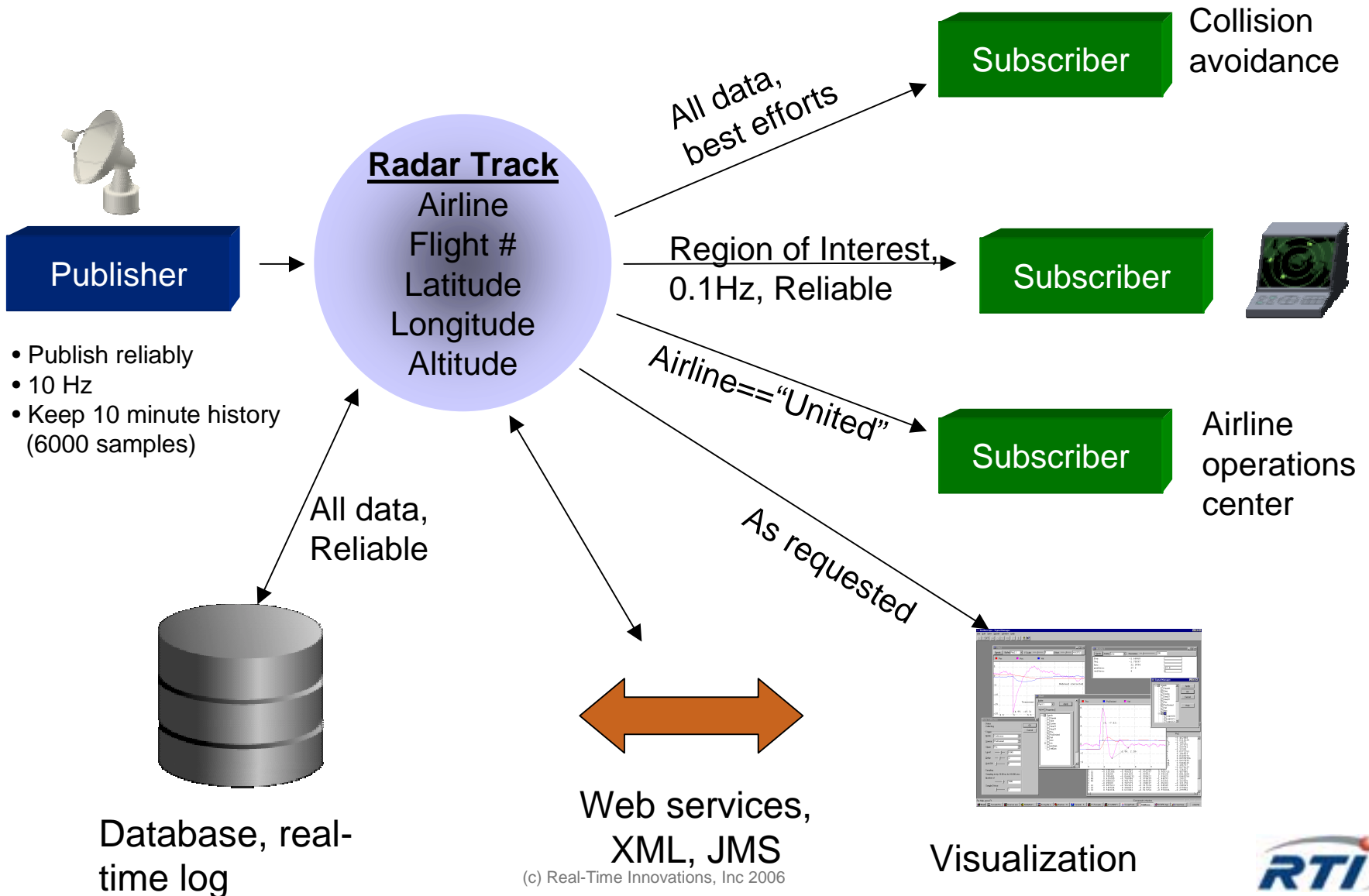
- Logger

- Reliable subscription

Application Example: Air Traffic Control



Application Example: Air Traffic Control



Air Traffic Control Example

- Requirements
 - Radar track manager
 - Coordinates several radars
 - Creates a unified model of airspace, publishes tracks
 - Publishes at 10Hz
 - Consoles
 - Display tracks from track manager
 - Hand off planes between consoles as they change cells
 - Only show planes in region of interest
 - Alarm if track not updated 1x/sec
 - Need to update newly-joined consoles (e.g. on reboot) with previous up to 6000, but no more than 10 mins of data
 - Publish commands; must fail over console to backup
- Quality of Service Settings
 - Radar track manager
 - Subscribe to radars
 - Publish tracks; offer deadline of 200ms for each track (5Hz)
 - Make tracks durable, with 6000-deep history (ten minutes)
 - Set lifespan on topic to 10 mins
 - Consoles
 - Subscribe to tracks in region using content-based filter
 - Overlap regions with secondary console(s); secondaries publish commands at lower strength
 - Subscribe with deadline of 1sec
 - If fails; alarm in 1 sec

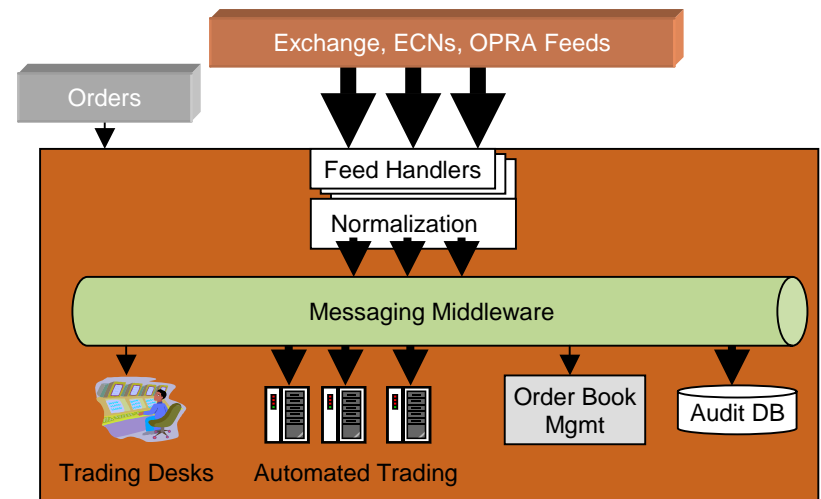
Application Example: Financial Trading

- Requirements

- Feed handlers
 - Take direct market data
 - Publish 750k msgs/sec
- Trading desks
 - Watch only “interesting” symbols
 - Can handle only 2Hz
- Automated trading
 - Watch for transient price differences in specific symbols
 - 300 machines watching various symbols/aspects of market
- Audit database
 - Save everything

- Quality of Service Settings

- Feed handlers
 - Offer multicast, reliable
- Trading desks
 - Subscribe at low bandwidth with time-based filters
- Automated trading
 - Partition market data to divide load



Application Example

Scenario: Alarm Handler. The system must display alarms from any of many subsystems

- Requirements

- Subsystems
 - Capable of raising alarms
- Alarm handler
 - Must process alarms
 - Must know if disconnected from any subsystem

- Quality of Service Settings

- Subsystems
 - Publish asynchronously (no deadline QoS)
 - Offer liveness
- Alarm handler
 - Subscribe to alarms from subsystem reliably
 - Request liveness notification



Real-Time Middleware

Keys and Instances

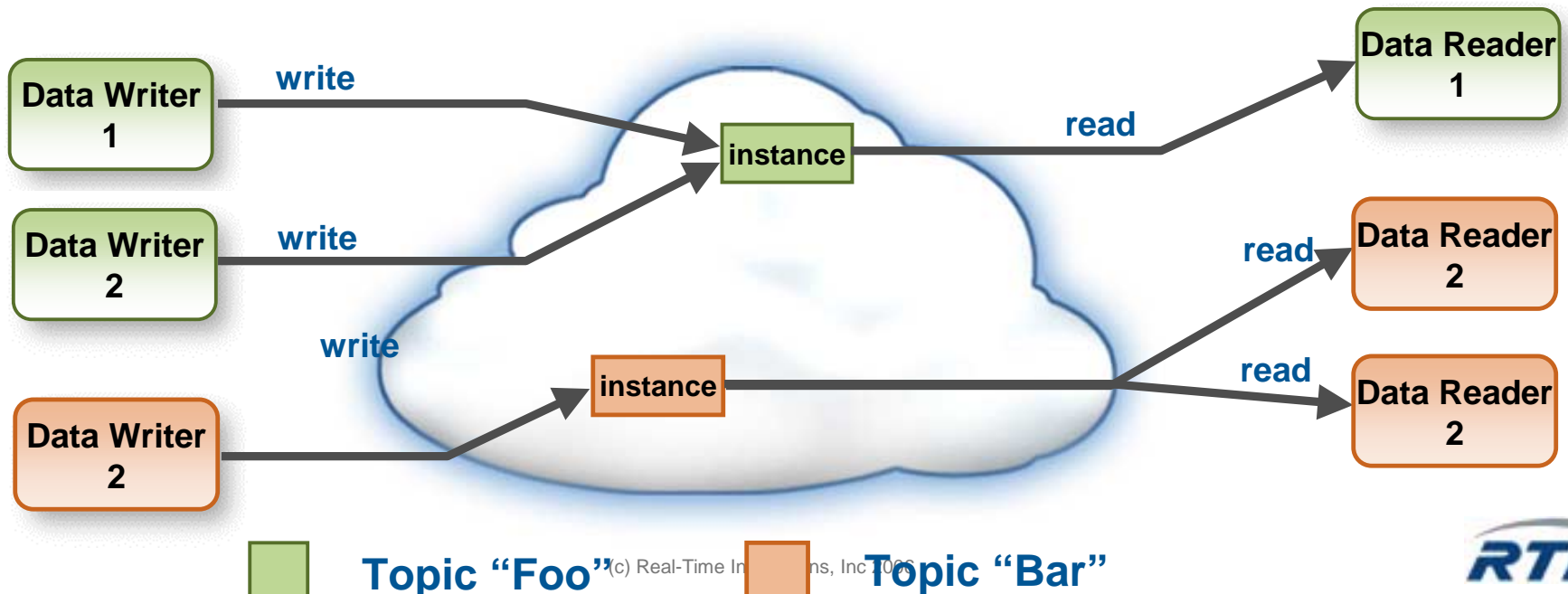
Details

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

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.

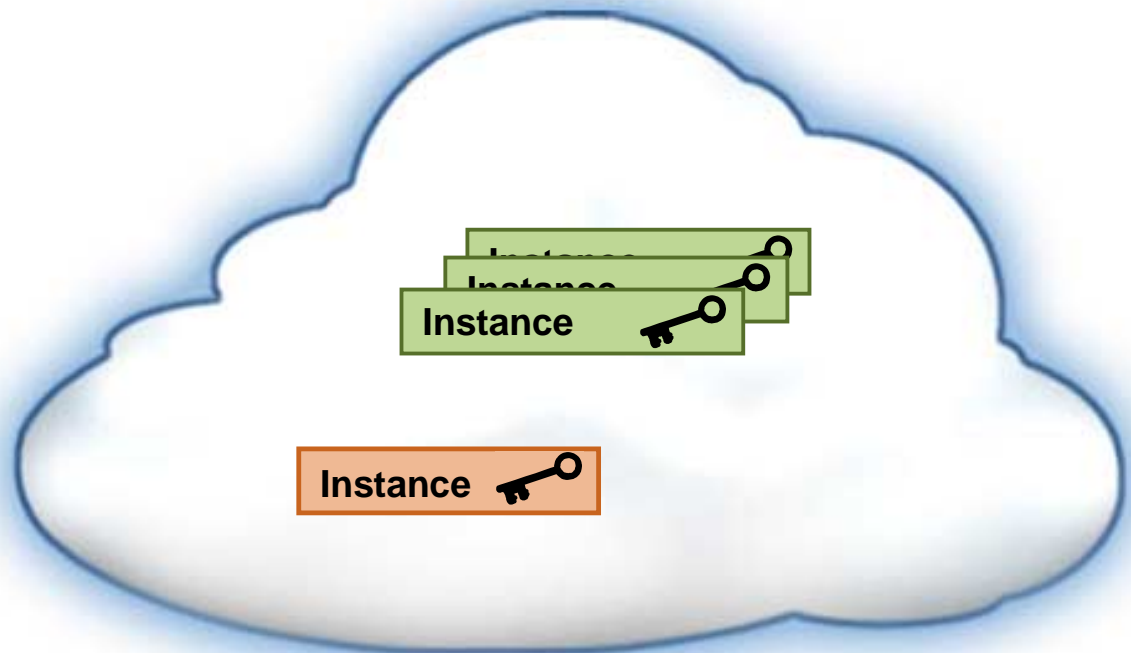


Example with Keys

- When using keys:
 - The system may contain multiple instances of a given topic, with each instance uniquely identified by a key. 

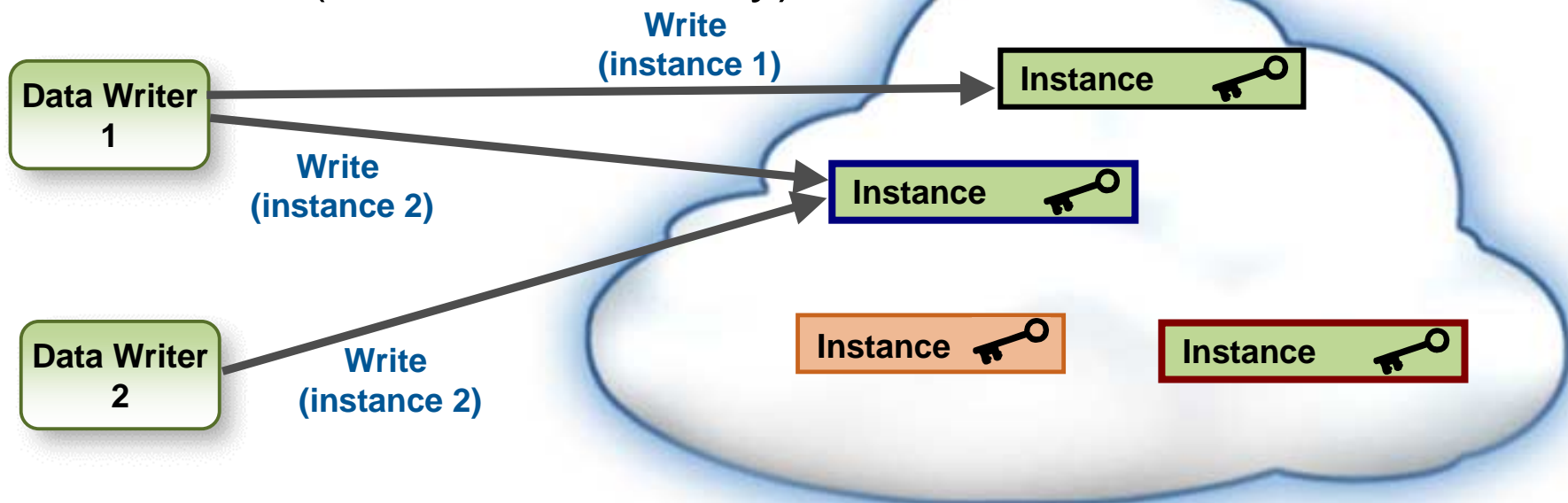
 Topic “Foo”

 Topic “Bar”



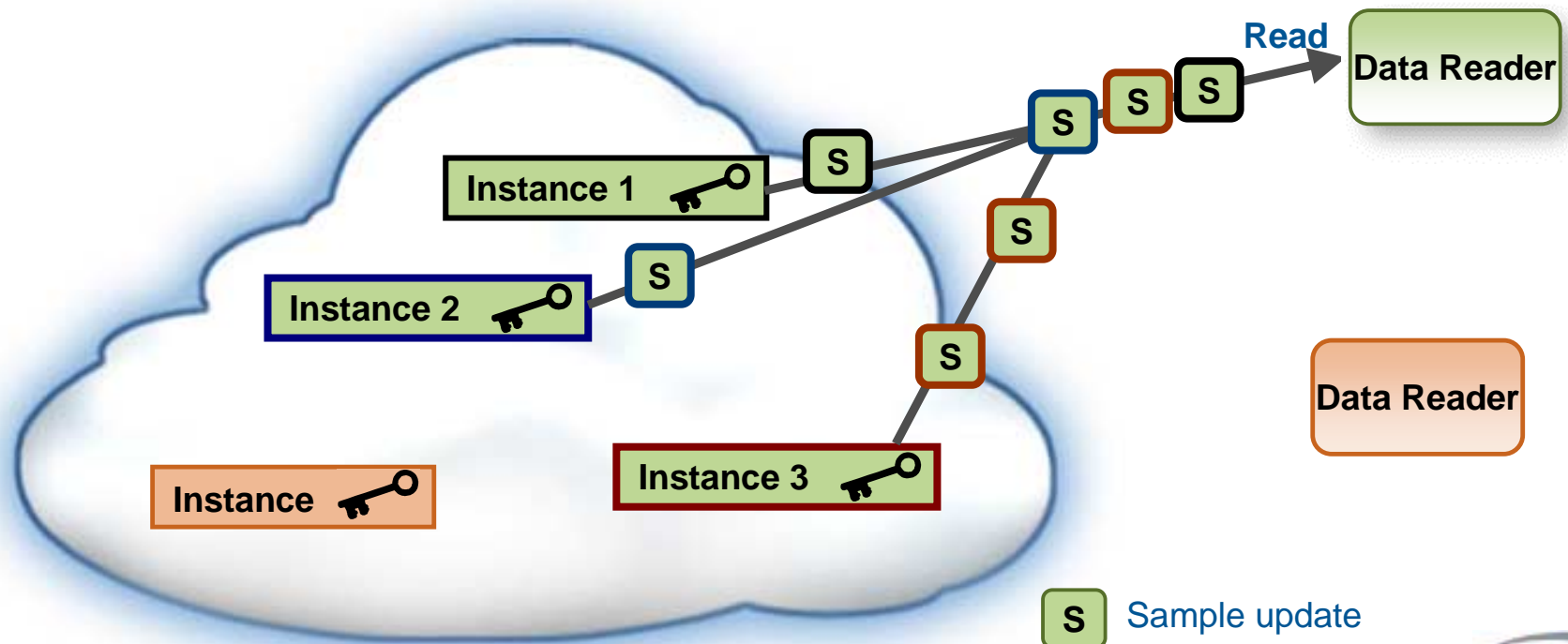
Example with keys (DataWriter)

- When using keys:
 - Each DW can write to multiple instances of a single topic
 - Multiple DWs may write to same instance (i.e. with same key)



Example with keys (DataReader)

- When using keys:
 - Each DR can receive updates from multiple instances of a single topic
 - Multiple DRs may read from the same instances



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

Keys and Instances: Instance State

- For each instance
 - 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

Keys and Instances: View State

- For each instance
 - 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

Keys and Instances: Sample State

- For each sample received
 - READ_SAMPLE_STATE: indicates that the DR has already accessed that sample by means of a read or take operation.
 - NOT_READ_SAMPLE_STATE: indicates that the DR has not accessed that sample before.
- Usage:
 - Pull only new samples from the queue when accessing data
 - Enable the use of the DataReader's queues as a convenient data store

Keys API support

- DataWriter API
 - register_instance() / unregister_instance()
 - lookup_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()

Resource Limits QoS

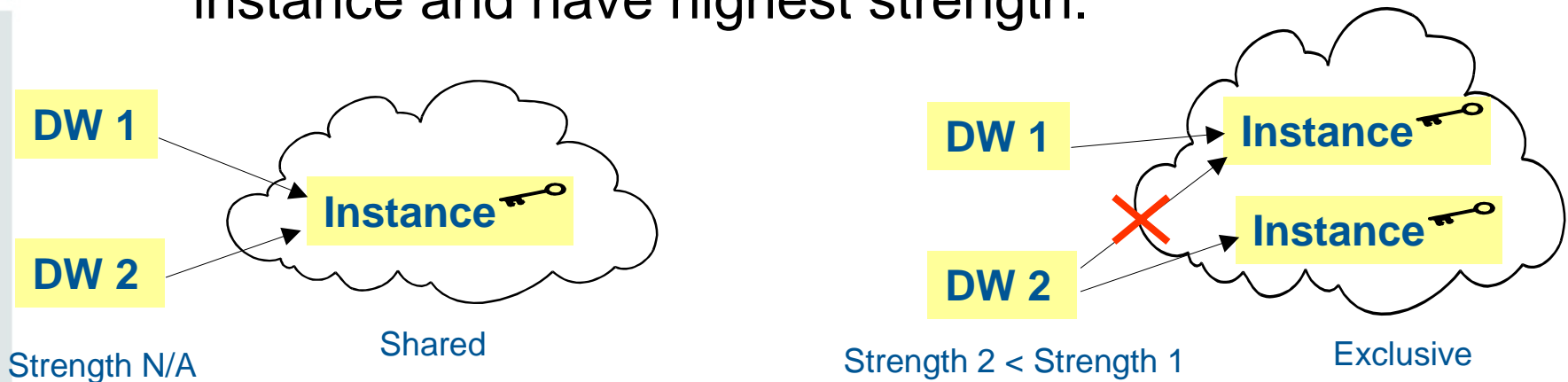
Specifies the resources available to the service (T, DW, DR)

- 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.



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 loses liveness 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.

Delivery : Presentation QoS

Governs how “related” sample updates are presented to subscribing application

- Access SCOPE defines the extent of relation
 - **<Instance>** – The relative order of samples sent by a DataWriter is only preserved on a per-instance basis
 - **Topic** – The relative order of samples sent by a DataWriter is preserved for all instances (ie Keys)
 - **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



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 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 guaranteed 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.

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.



Real-Time Middleware

Reliable Pub-Sub

Details

Reliable Publish-Subscribe

Goal: provides reliability layer (on ANY transport)

- Optimizes bandwidth usage
 - lightweight ACK/NACK mechanism
 - minimal number of re-sends
- Minimizes latency
 - dropped issues detected in timely fashion
 - disruptions on one “channel” does not hold up delivery on other “channels”
- Maximizes flexibility
 - trade off reliability for timing-determinism on per subscription basis
 - tunable QoS parameters to achieve just the “right amount” of reliability for each pub/sub pair`

Reliable Communications

- DataWriter must “offer” reliability and each DataReader can “request” reliable delivery
- Best-Effort vs. Reliable
 - Same entities, additional QoS and overhead associated
- Key differences:
 - Reliable properties
 - queue management, flow control
 - timeliness of detecting dropped issues
 - Listener to monitor events
 - Status indicators to convey delivery status

Reliability Mechanism

- Positive acknowledgement (ACK)
 - Used to confirm receipt of an issue
- Negative acknowledgement (NACK)
 - Used to request a missing issue
- Speculative caching
 - DataReader will store out-of-sequence issues so that DataWriter will not have to resend later
- Ordered delivery
 - DataReader will deliver issues to application in correct order



Conditions and WaitSets

Another way to get your data.

Overview

- WaitSets

- Can create many different WaitSets
- WaitSets block, with a timeout, user thread(s) waiting for certain conditions to be true
- Each WaitSet can have any number of Conditions attached to it, for any number of entities

- Conditions

- StatusCondition – triggered by the entity statuses
- ReadCondition – triggered by the presence of data
- GuardCondition – triggered by user-defined condition

Conditions – General Implementation

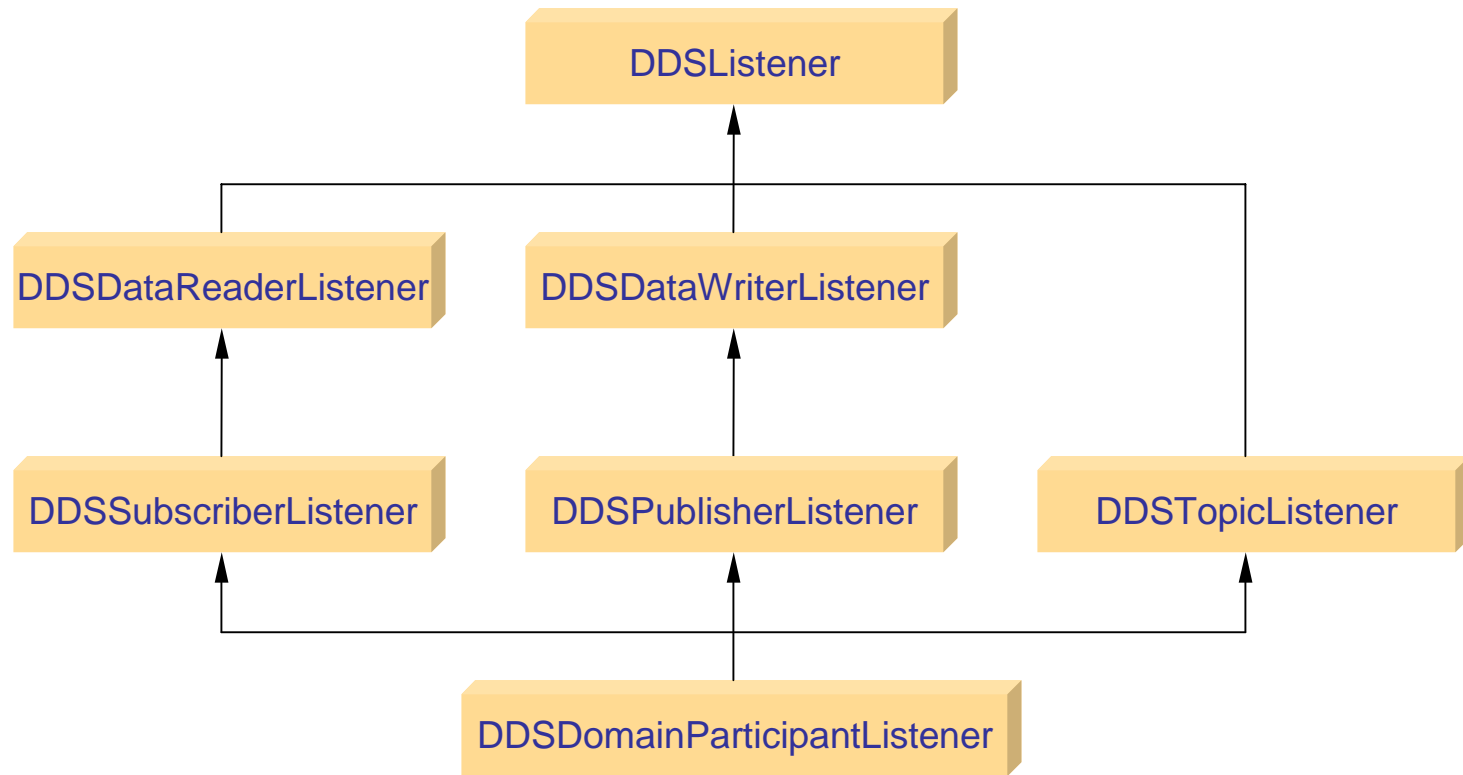
- Represented as a boolean value of its trigger
 - Can check at any time
- Maintains a list of WaitSets to which it is attached
- When a Condition becomes true
 - Iterate through all WaitSets
 - Update the “active-ness” of the Condition in the WaitSet’s condition list
 - Unblock the WaitSet’s semaphore
- User needs to keep track of each condition



Listeners

Details

Inheritance Diagram



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_matched()
 - 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_matched()`
- 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.

Plain Communication Status callback order

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

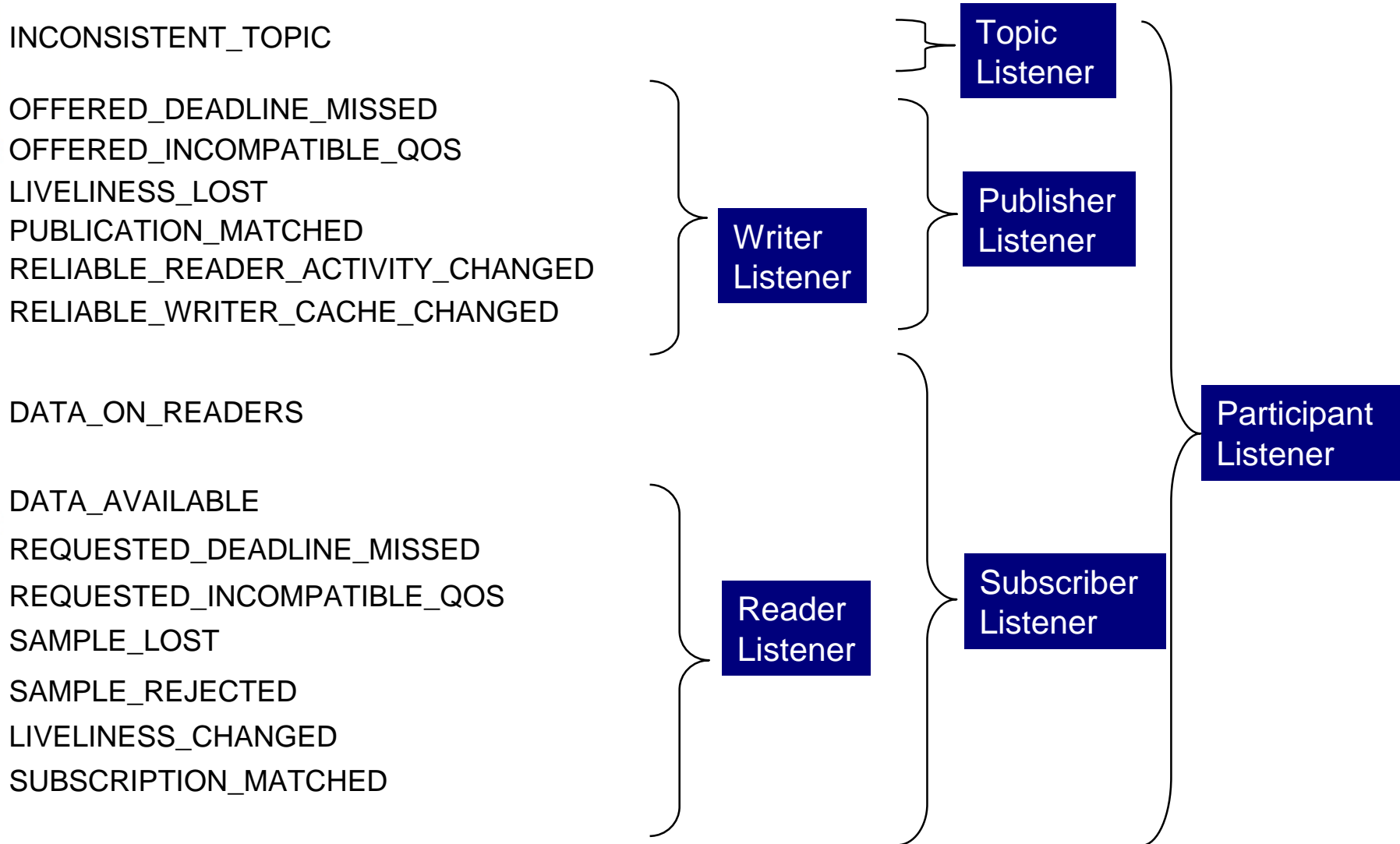
1. Reader/Writer (na for Topic Status)
2. Publisher/Subscriber /Topic (For Topic Status)
3. Participant

Read Communication Status callback order

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

1. `on_data_on_readers()` on Subscriber
2. `on_data_on_readers()` on DomainParticipant
Can call `notify_datareaders()` to trigger `on_data_available()` calls
3. `on_data_available()` on DataReader
4. `on_data_available()` on Subscriber
5. `on_data_available()` on DomainParticipant

RTI 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
 - And/Or wait synchronously using a WaitSet
- Both approaches are unified using STATUS changes



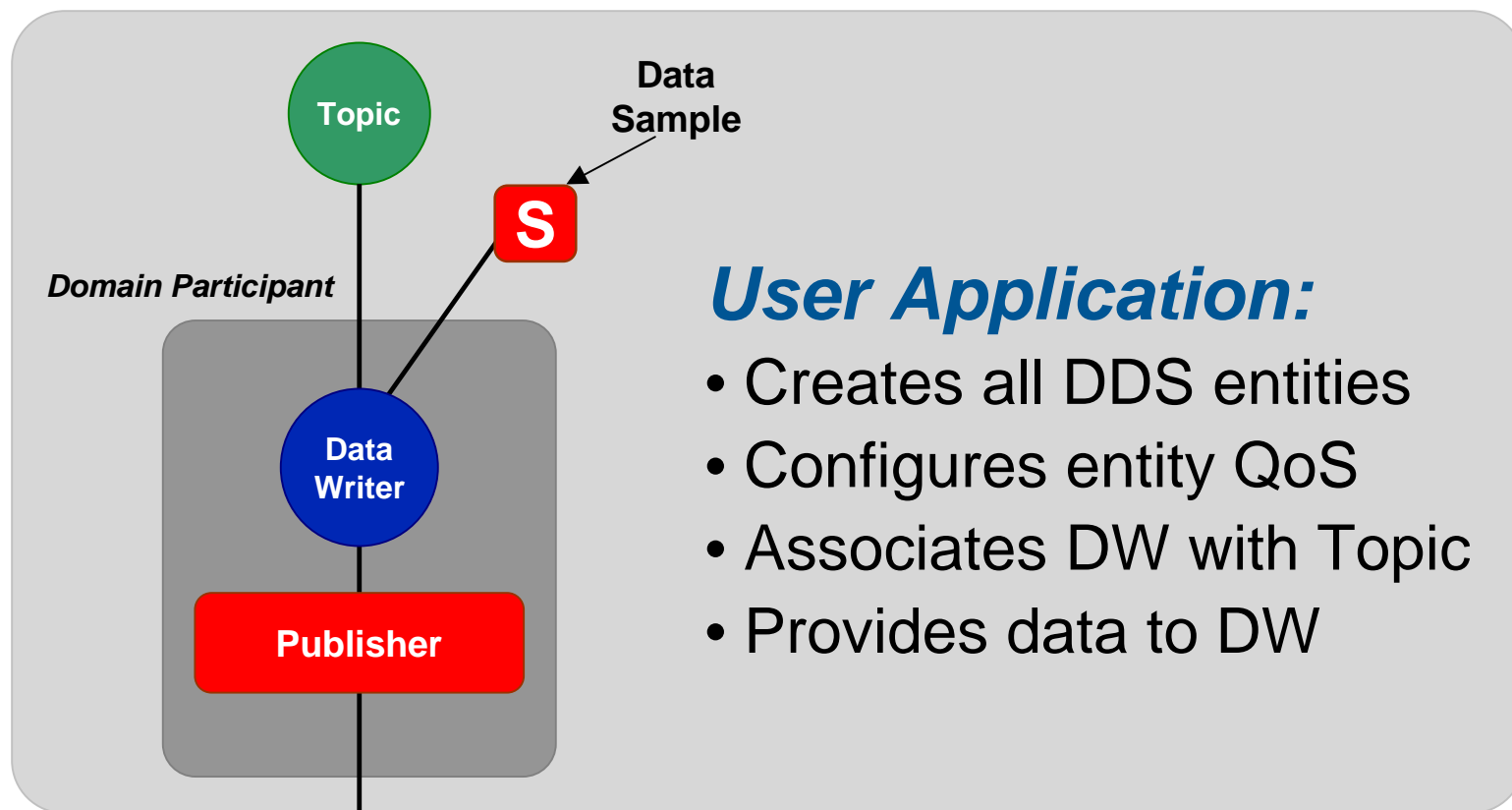
Content Filtered Topics



Real-Time Middleware

Sending and Receiving

Sending Data (Publication)



User Application:

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

Example: Publication

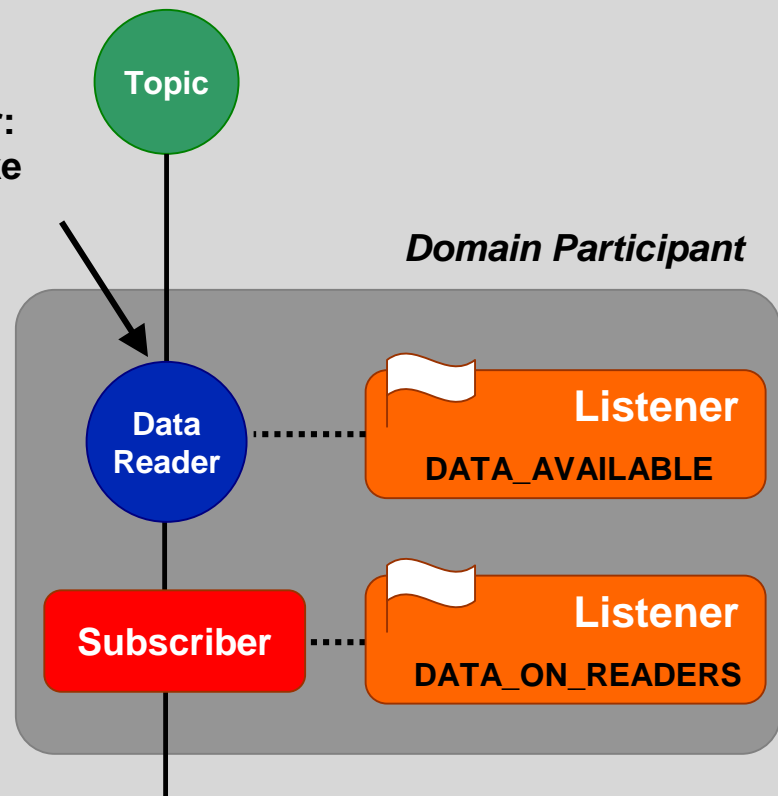
```
DDSPublisher *publisher = domain->create_publisher(  
    publisher_qos,  
    publisher_listener, publisher_listener_mask);  
  
DDSTopic *topic = domain->create_topic(  
    "Track", "TrackStruct",  
    topic_qos, topic_listener, mask);  
  
DDSDataWriter *writer = publisher->create_datawriter(  
    topic, writer_qos, writer_listener, mask);  
  
TrackStruct my_track;  
  
writer->write(&my_track);
```

Receiving Data (Subscription)

User Application:

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

Listener:
read,take



Example: Subscription

```
DDSSubscriber *subs = domain->create_subscriber(  
    subscriber_qos, subscriber_listener, mask);
```

```
DDSTopic *topic = domain->create_topic(  
    "Track", "TrackStruct",  
    topic_qos, topic_listener, mask);
```

```
DDSDataReader *reader = subscriber->create_datareader(  
    topic, reader_qos, reader_listener, mask);
```

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

How to get data (listener-based)

```
class MyListener : DataReaderListener {  
    virtual void on_data_available(DDSDataReader *reader);  
}
```

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

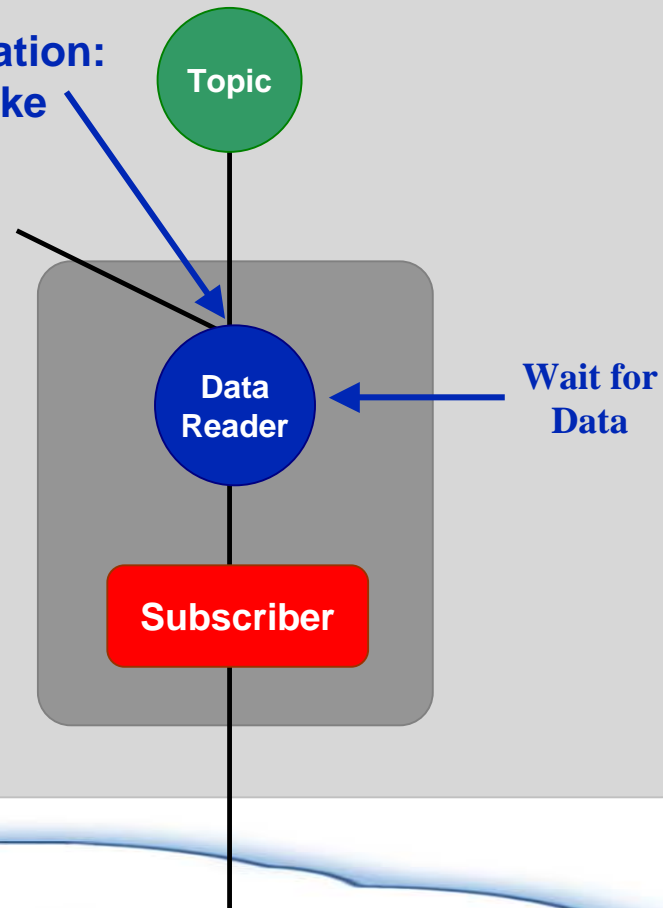
```
MyListener::on_data_available( DataReader reader )  
{  
    FooSeq received_data;  
    SampleInfoSeq sample_info;  
  
    reader->take( &received_data, &sample_info, ...)  
  
    // Use received_data  
}
```

Subscription Wait-Set

User Application:

- Creates all DDS entities
- Configures entity QoS
- Associates DR with Topic
- Blocks & waits for data from DR(s) (like select)

Application:
read,take



How to get data (wait-based)

```
DDSReadCondition *foo_condition =
    reader->create_readcondition(...);

waitset->attach_condition(foo_condition);

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

reader->take_w_condition(&received_data,
                        &sample_info,
                        foo_condition);

// Use received_data
```