DDS

EXPLAINED

Angelo Corsaro, PhD

Chief Technology Officer

ADLINK Tech. Inc.

angelo.corsaro@adlinktech.com

ADLINK

Leading EDGE COMPUTING

ADLINK
Leading EDGE COMPUTING

What is DDS?

DDS in 131 Characters

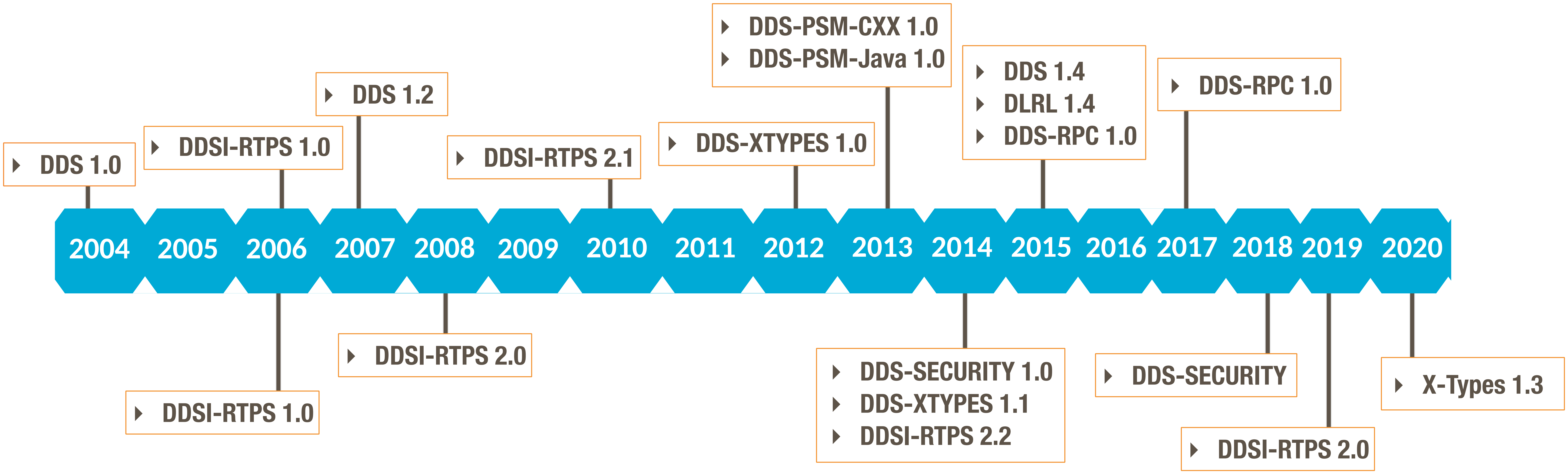
DDS is a standard technology for ubiquitous, interoperable, secure, platform independent, and real-time **data sharing** across network connected devices

ADLINK Leading EDGE COMPUTING

The DDS Standard

Relevant Standards

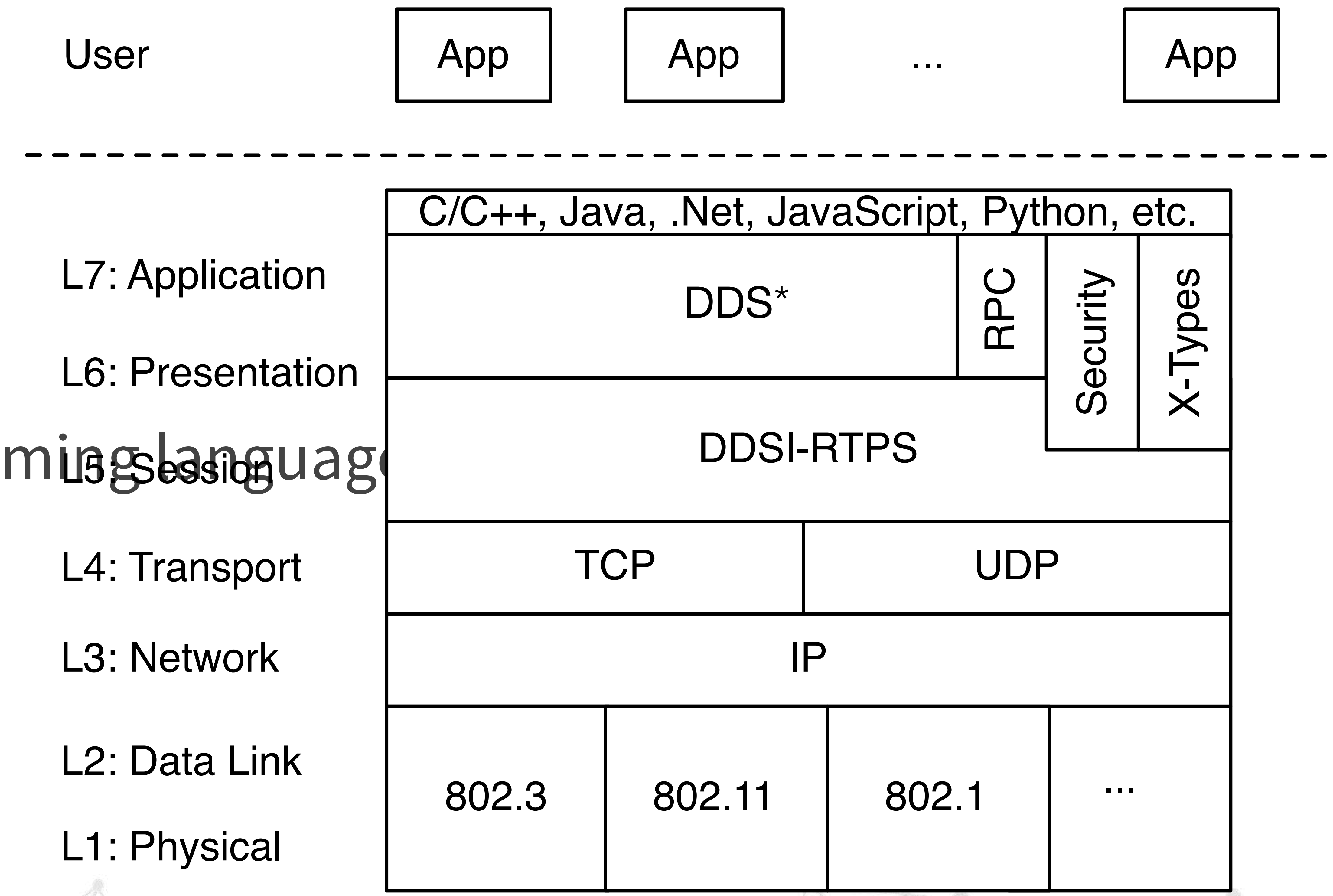
DDS Evolution



DDS Standards

Data Distribution Service (DDS)

Defines a high level API for programming language independent data sharing



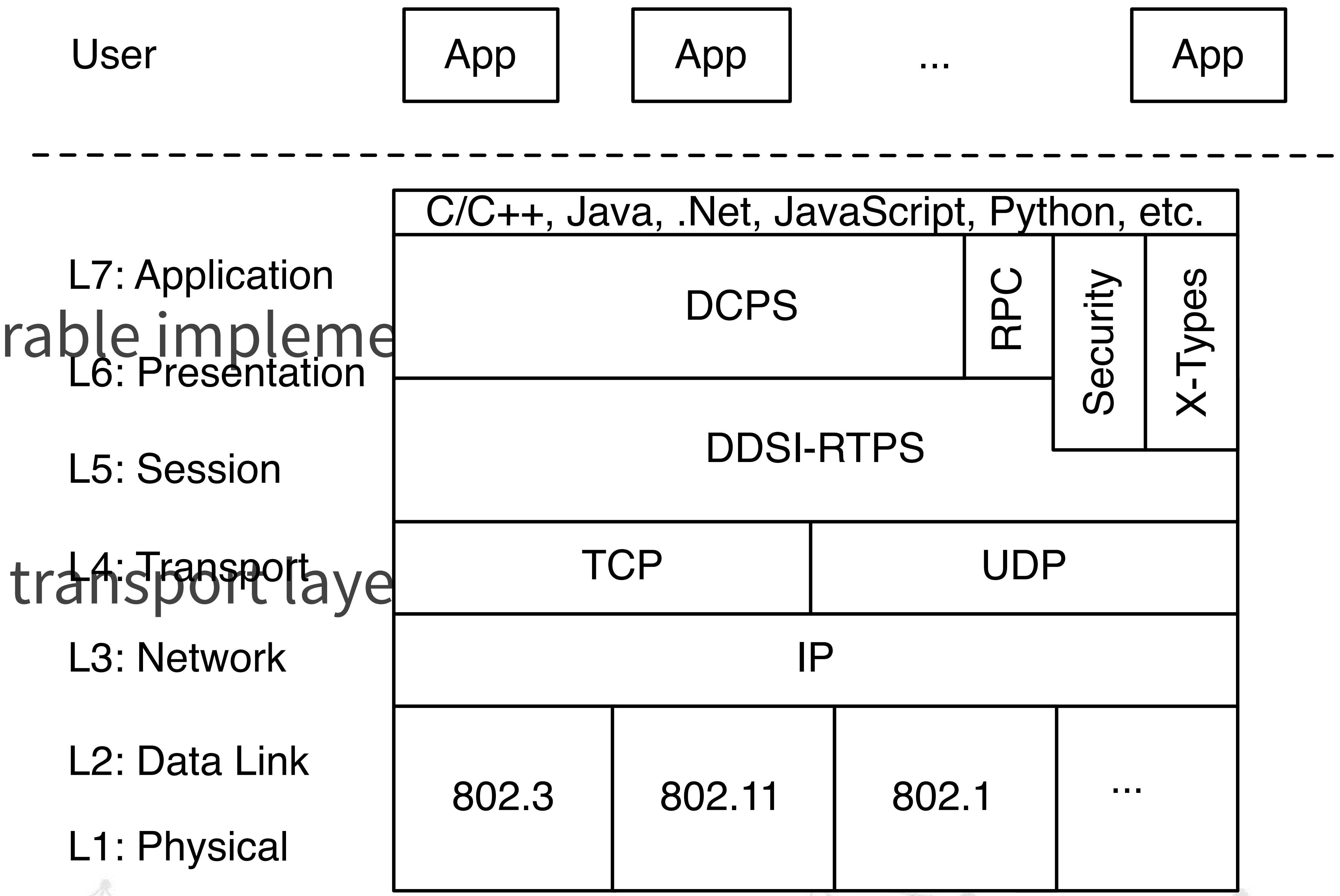
(*) This used to be called DCPS as originally the DDS standard was composed by two layers, DCPS (Data Centric Publish/Subscribe and DLRL (Data Local Reconstruction Layer).

DDS Standards

DDS Interoperability Protocol (DDSI-RTPS)

Defines a wire protocol for interoperable impleme
abstractions.

This protocol assumes a best-effort transport laye
provided by DDSI.



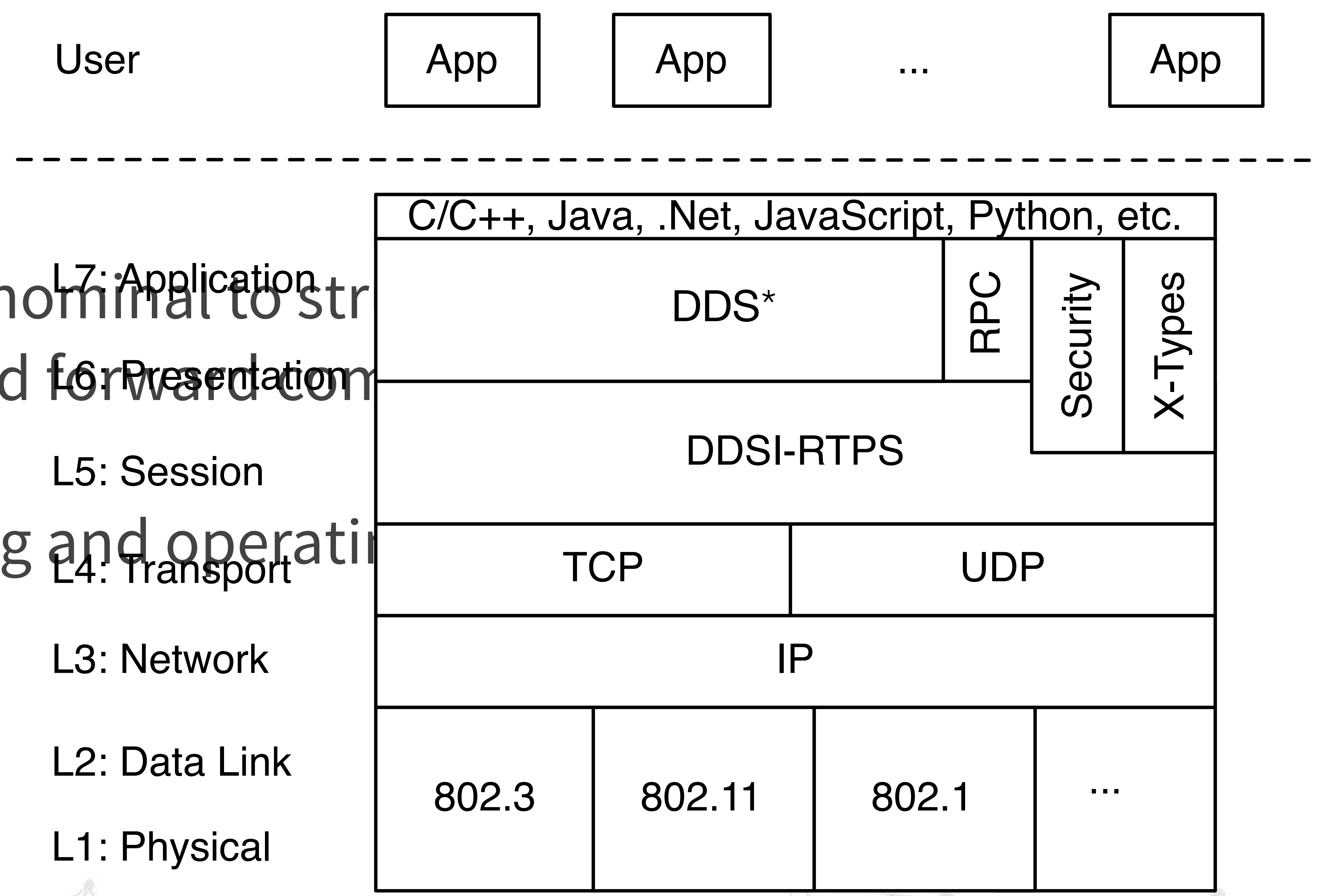
(*) *This used to be called DCPS as originally the DDS standard was composed by two layers, DCPS (Data Centric Publish/Subscribe and DLRL (Data Local Reconstruction Layer).*

DDS Standards

eXtensible Types (DDS-XTypes)

Extends the DDS type system from nominal to str
very good support for evolutions and forward com

Defines APIs for dynamically defining and operati

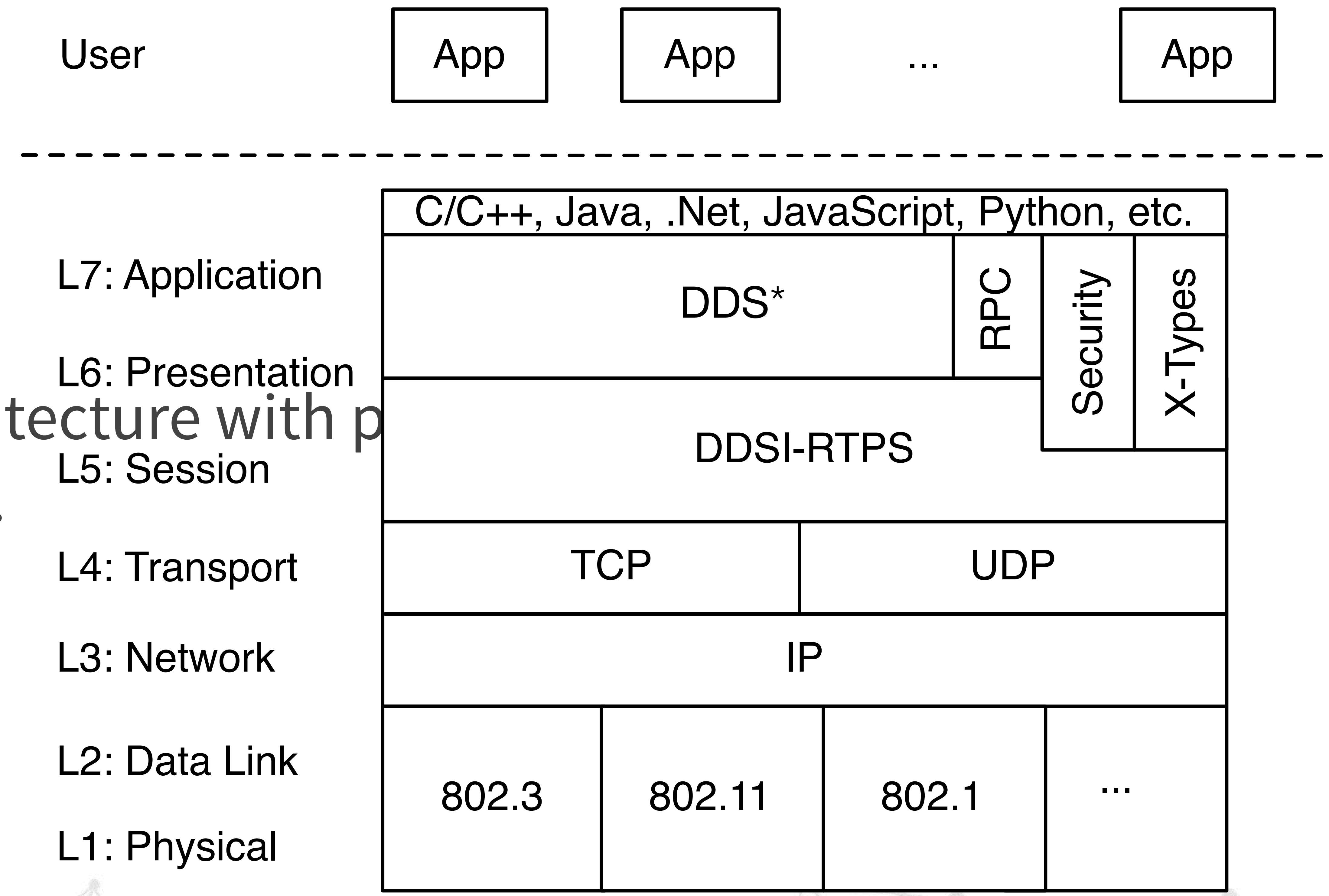


(*) This used to be called DCPS as originally the DDS standard was composed by two layers, DCPS (Data Centric Publish/Subscribe and DLRL (Data Local Reconstruction Layer).

DDS Standards

Security (DDS-Security)

Defines a data-centric security architecture with p
Access Control, Crypto and Logging.

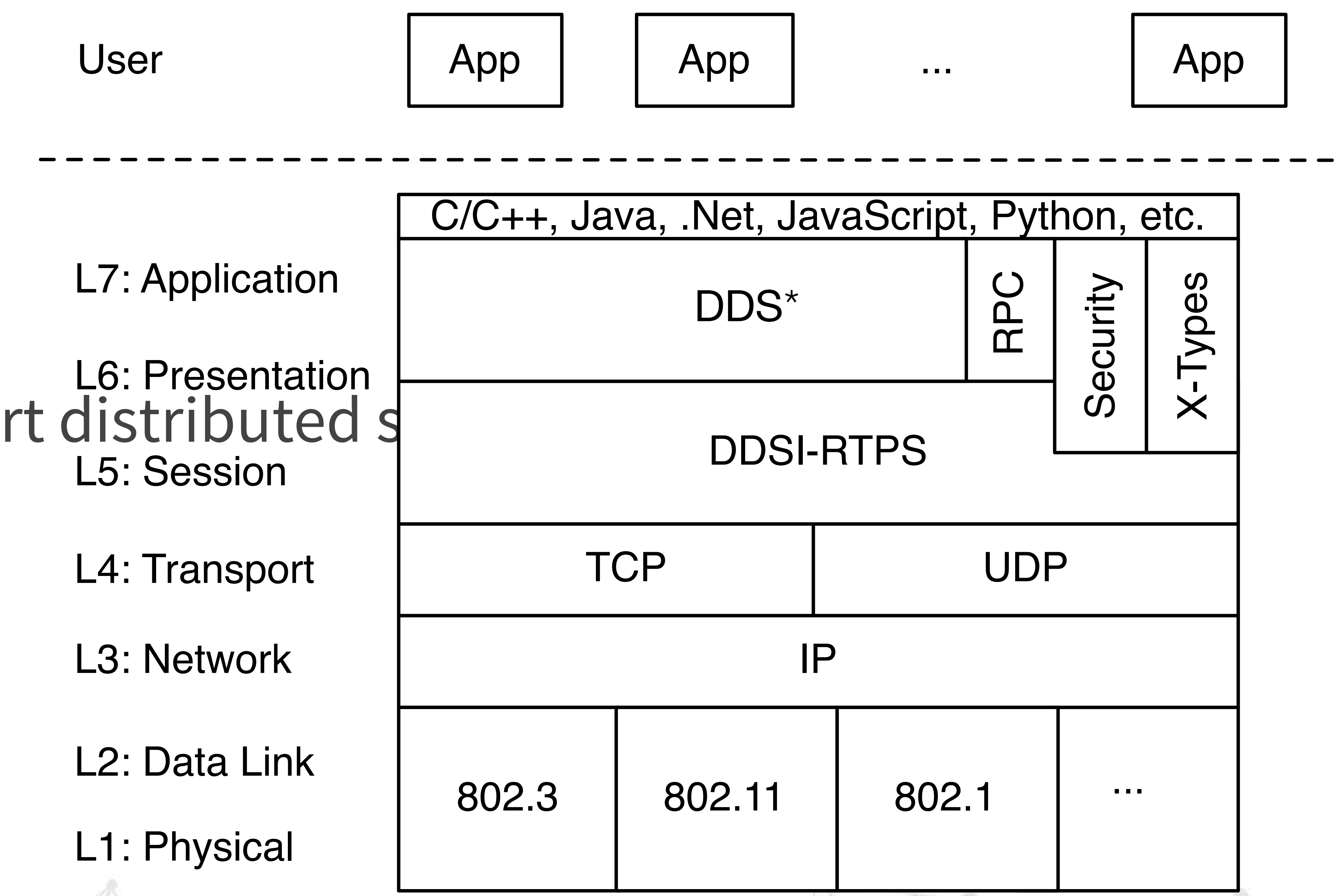


(*) This used to be called DCPS as originally the DDS standard was composed by two layers, DCPS (Data Centric Publish/Subscribe and DLRL (Data Local Reconstruction Layer).

DDS Standards

Remote Procedure Calls (DDS-RPC)

Extends DDS abstractions to support distributed s
remote operation invocations.



(*) This used to be called DCPS as originally the DDS standard was composed by two layers, DCPS (Data Centric Publish/Subscribe and DLRL (Data Local Reconstruction Layer).

Who Uses DDS?

AUTONOMOUS VEHICLES

DDS is used for data sharing within and across the vehicle.

The environment is highly heterogeneous and requires dynamic pairing along with coordination of fast moving vehicles

SMART CITIES

DDS is used as the integration technology for data sources and sinks.

DDS is also often used as a control plane to control and provision equipment

SMART GRID

DDS is used to integrate and normalise data sharing among the various elements of a smart grid at scale

Duke's Energy COW showed how only with DDS it was possible to distribute the phase alignment signal at scale with the required 20ms periodicity

SMART GREEN HOUSES

DDS leveraged to virtualise I/O and provide better decoupling between I/O, Control and Management functions of the system

COMBAT MANAGEMENT SYSTEMS

DDS is used at the core of an incredible number of Naval Management System

In this context DDS distributes soft and hard real-time sensor and actuator data

NASA LAUNCH SYSTEMS

The NASA Kennedy Space Centre uses DDS to collect the Shuttle Launch System Telemetry.

DDS streams over 400.000 Msgs/sec

In comparison, world-wide, Twitter deals with less than 7000 msgs/sec

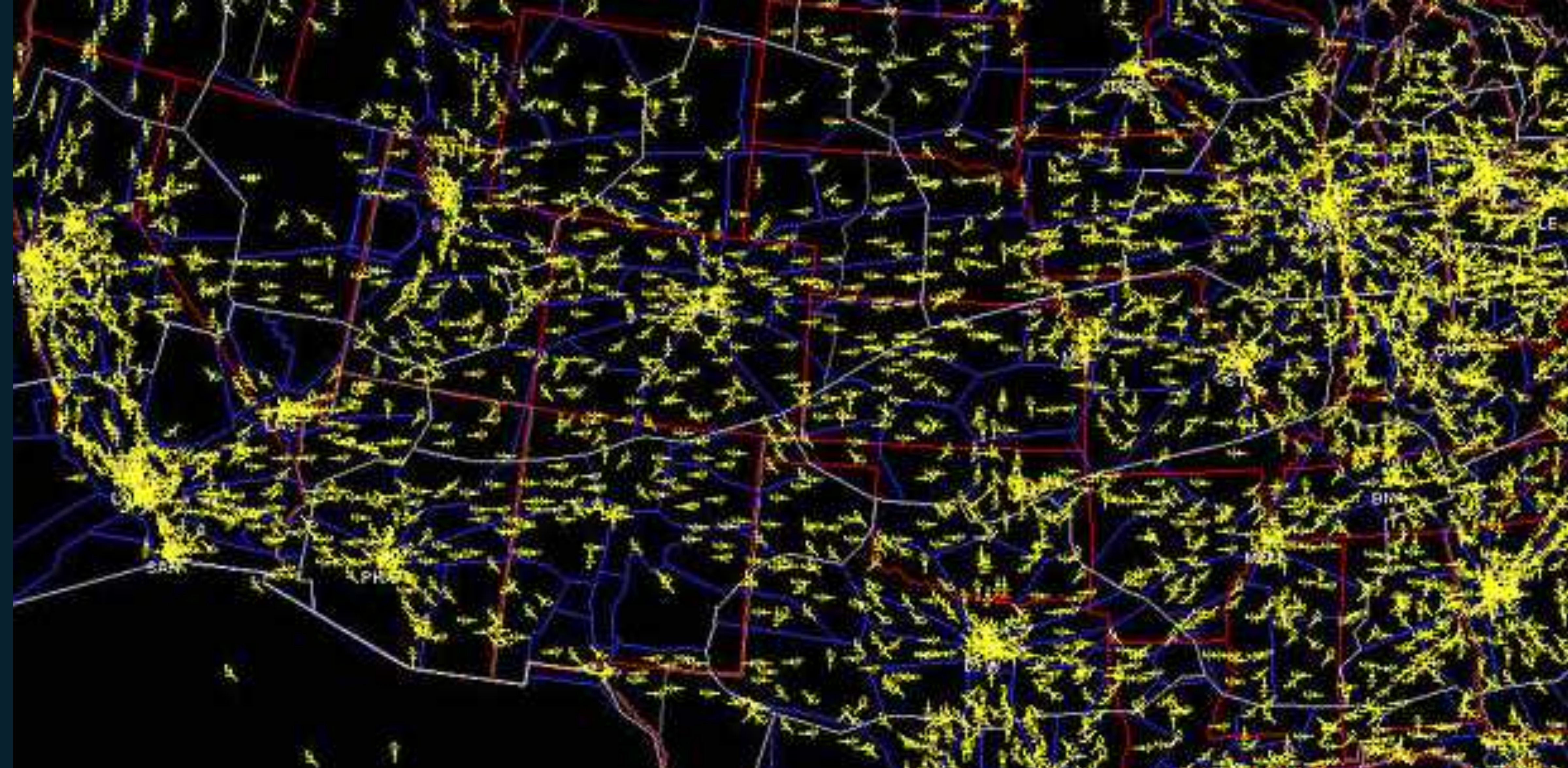
AIR TRAFFIC CONTROL

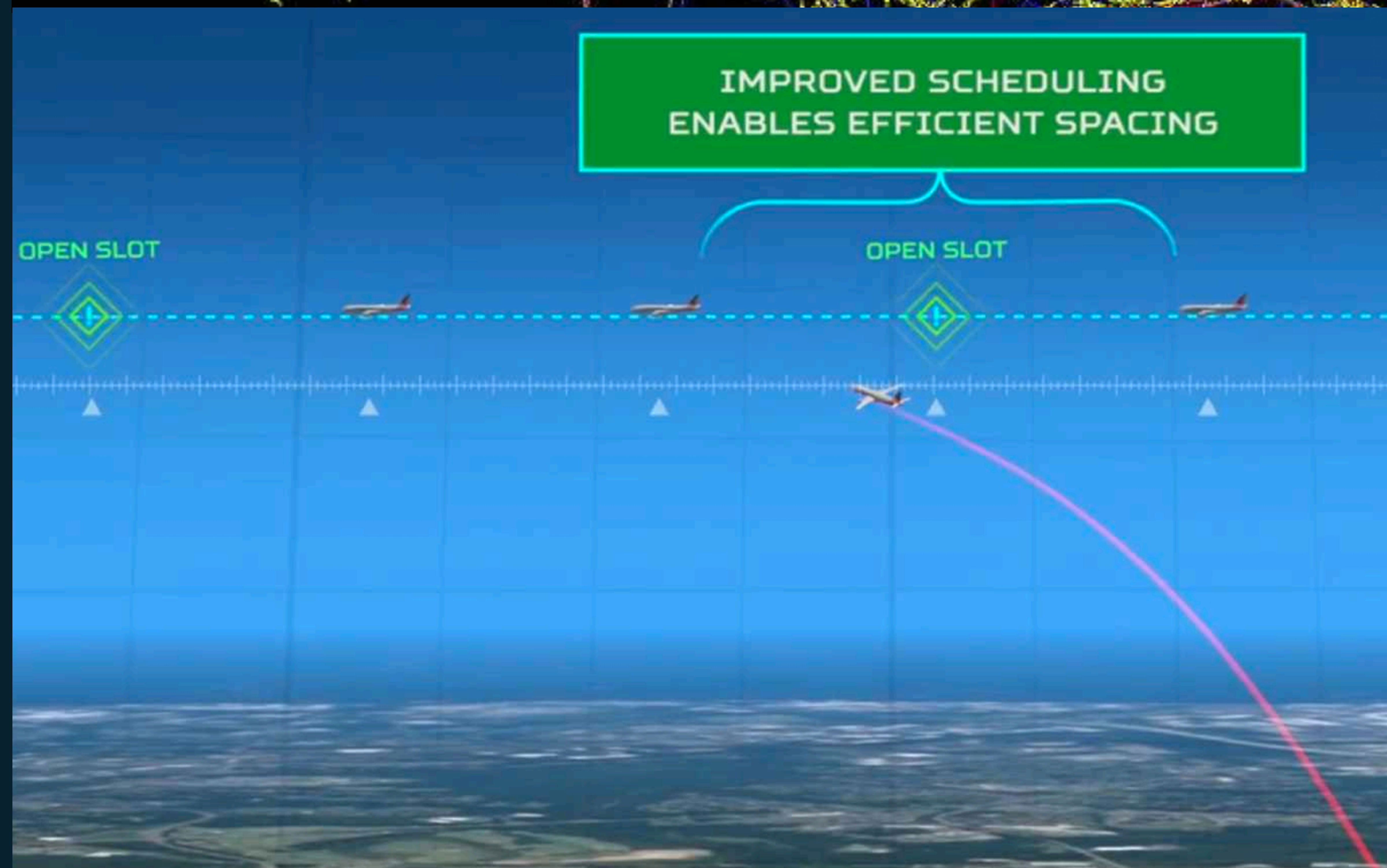
DDS is used to share Flight Data within and across Air Traffic Control Centres.

These applications have extremely high dependability constraints.

NASA'S SMART NAS

The Shadow Mode Assessment Using Realistic Technologies for the National Airspace System (SMART-NAS) Project is an air traffic management simulation capability to explore the NAS integration of alternative concepts, technologies and architectures.





UNMANNED AIR VEHICLES

DDS is used in UAV in-flight mission management system to distribute several thousands of sensor targets

SIMULATORS

Airbus Helicopters use DDS for their full-function (level D) helicopter simulators

All data in the system is managed in DDS, with applications converting to the ARINC429 bus used by the avionics

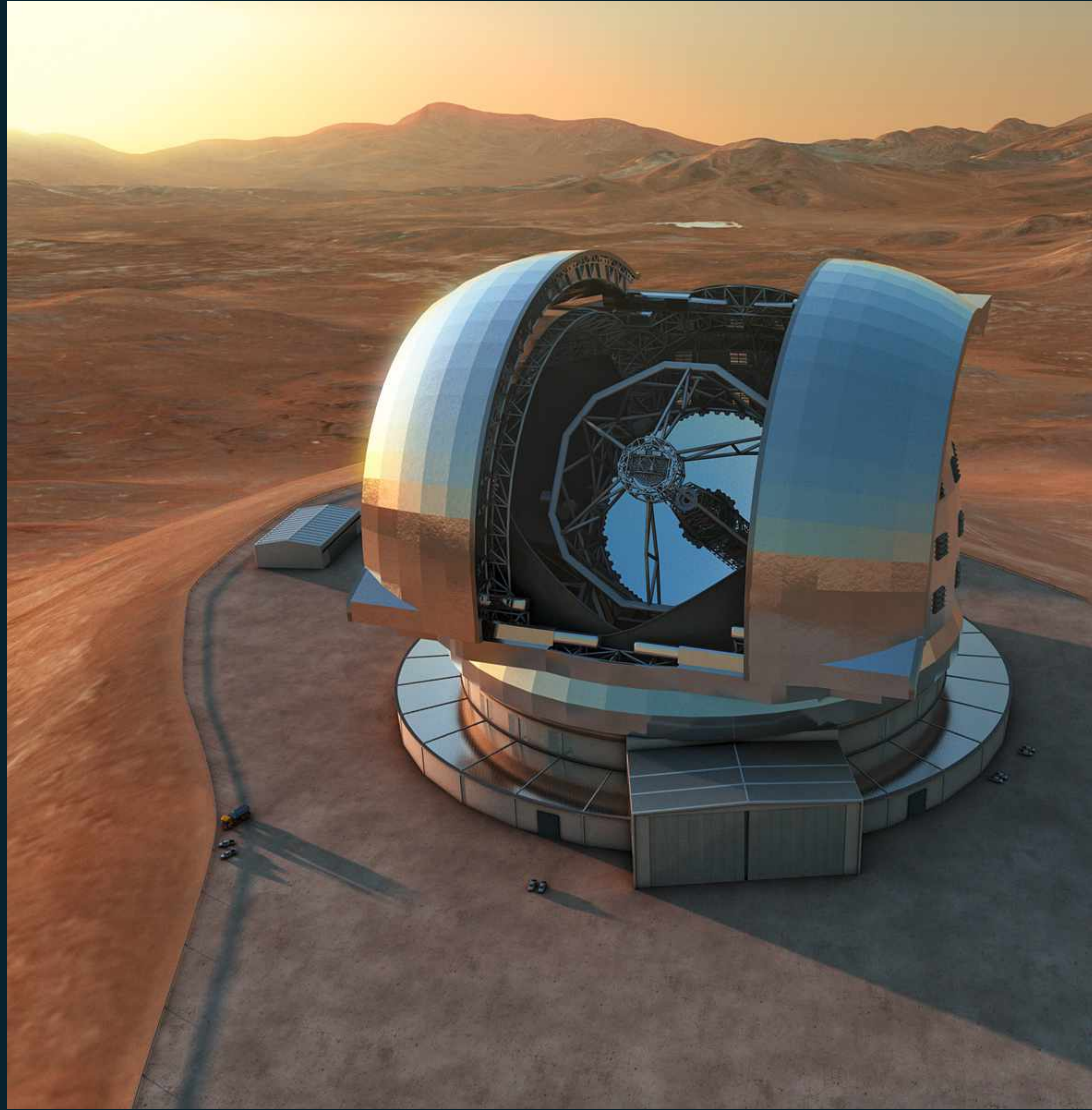
SMART FACTORY

DDS is used in Smart Factories to provide horizontal and vertical data integration across the traditional SCADA layers.

EXTREMELY LARGE TELESCOPE (ELT)

DDS used to control the 100.000 mirrors that make up ELT's optics.

These mirrors are adjusted 100 times per second



MEDICAL DEVICES

DDS is used inside several medical devices to share data between the various stages of data acquisition, processing and visualisation

ROBOTICS

DDS is heavily used for data sharing in Robotics and is today at the heart of the Robot Operating System (ROS)

Why are these Applications Using DDS?

DDS PROVIDES AN EXTREMELY HIGH LEVEL AND POWERFUL ABSTRACTIONS ALONG WITH A ROCK SOLID INFRASTRUCTURE TO BUILD HIGHLY MODULAR AND DISTRIBUTED SYSTEMS

DDS MAKES IT MUCH EASIER TO SOLVE SOME VERY HARD DISTRIBUTED SYSTEM PROBLEMS, SUCH AS FAULT-TOLERANCE, SCALABILITY AND ASYMMETRY

DDS IS LIKE A UNIVERSAL GLUE THAT ALLOWS TO SEAL TOGETHER HIGHLY HETEROGENEOUS ENVIRONMENTS WHILE MAINTAINING A SINGLE, ELEGANT AND EFFICIENT ABSTRACTION

Platform Indeper nt

DDS implementation are av a incred numbers of platforms, including ente desktop, embedded, real-time, mobile, and we

Polyglot

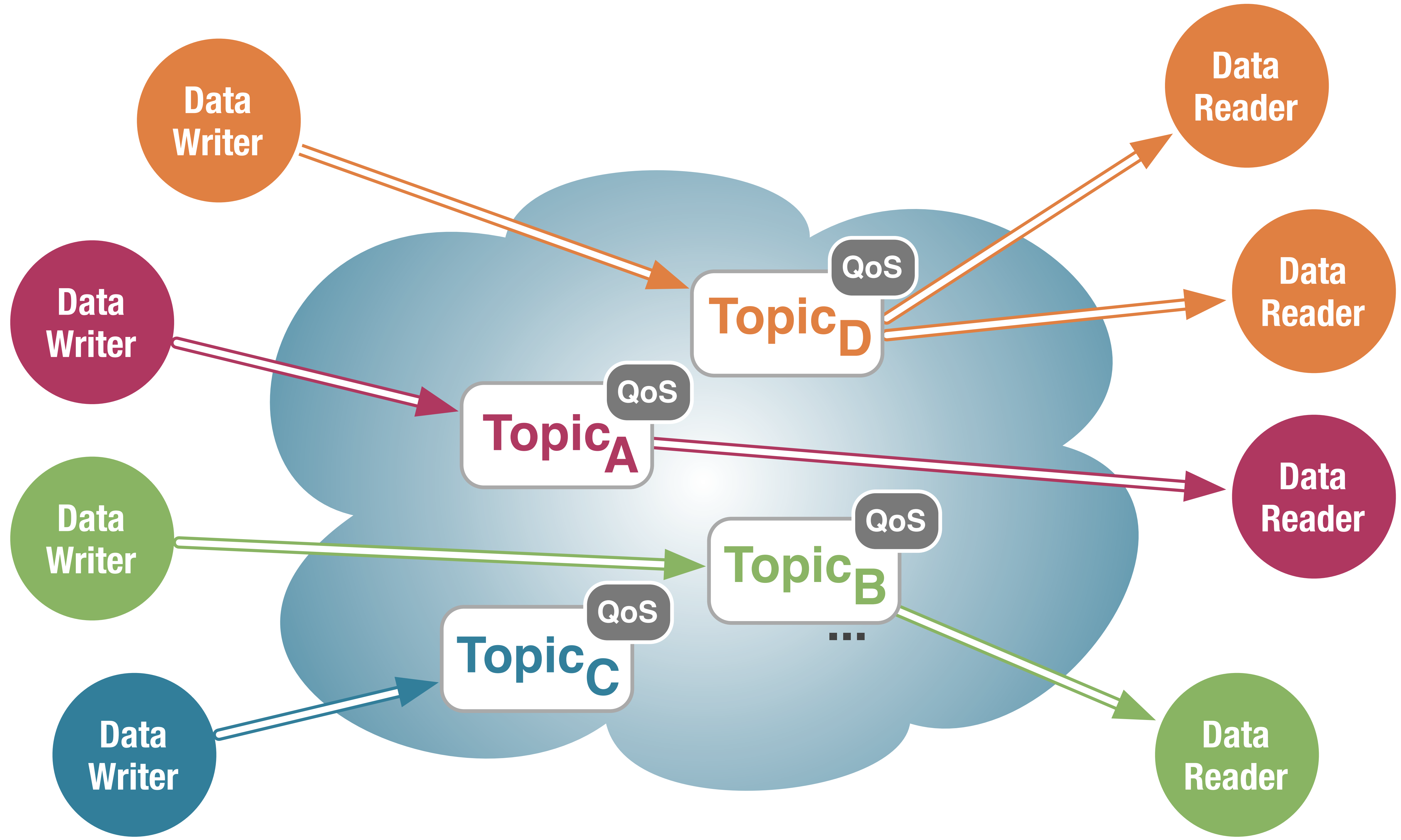
DDS applications can be written in your favourite programming language.

Interoperability across languages is taken ca

DDS: Foundations

Virtualised Data Space

Applications can **autonomously** and **asynchronously read** and **write** data enjoying **spatial** and **temporal decoupling**

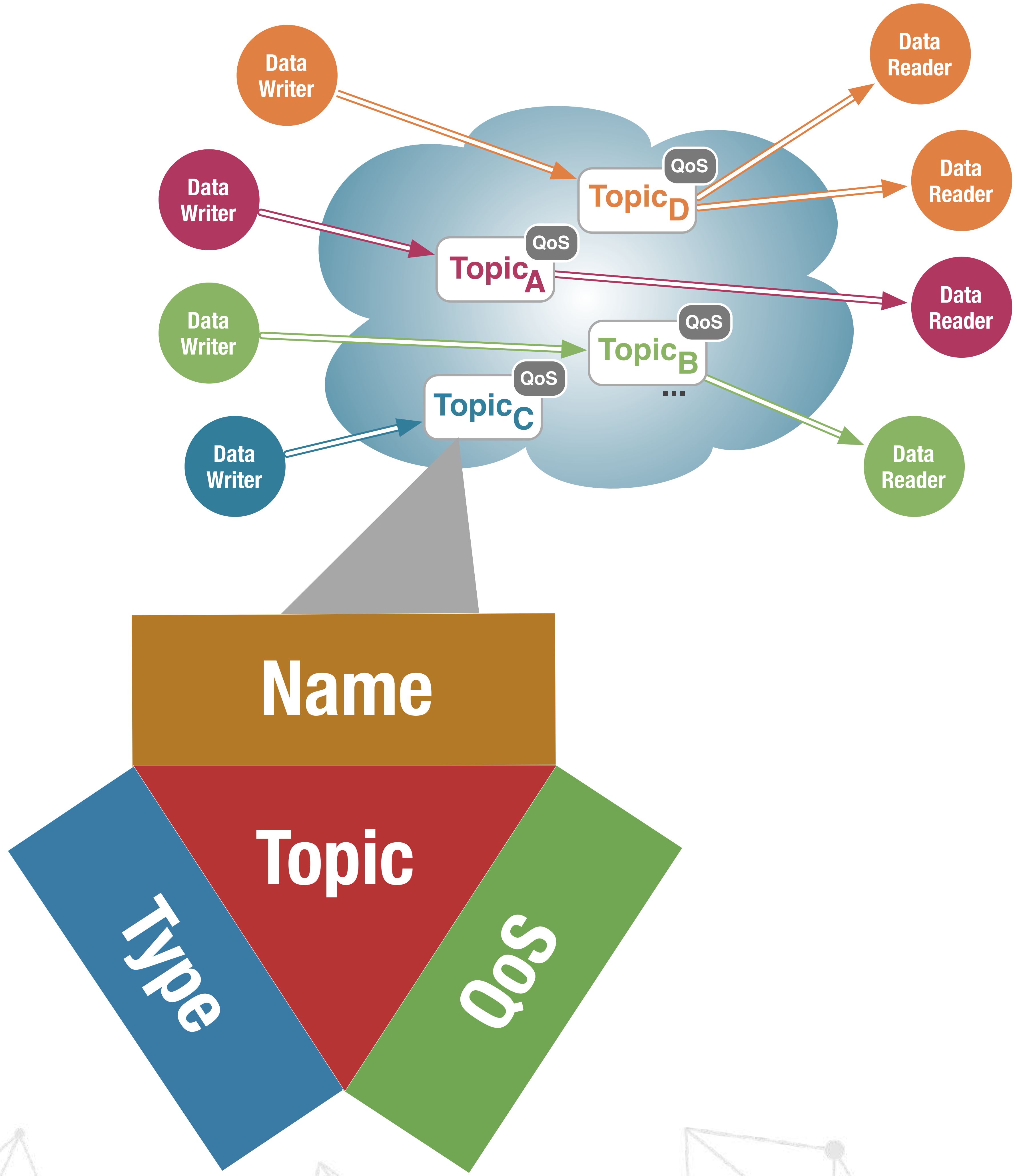


DDS Global Data Space

Topic

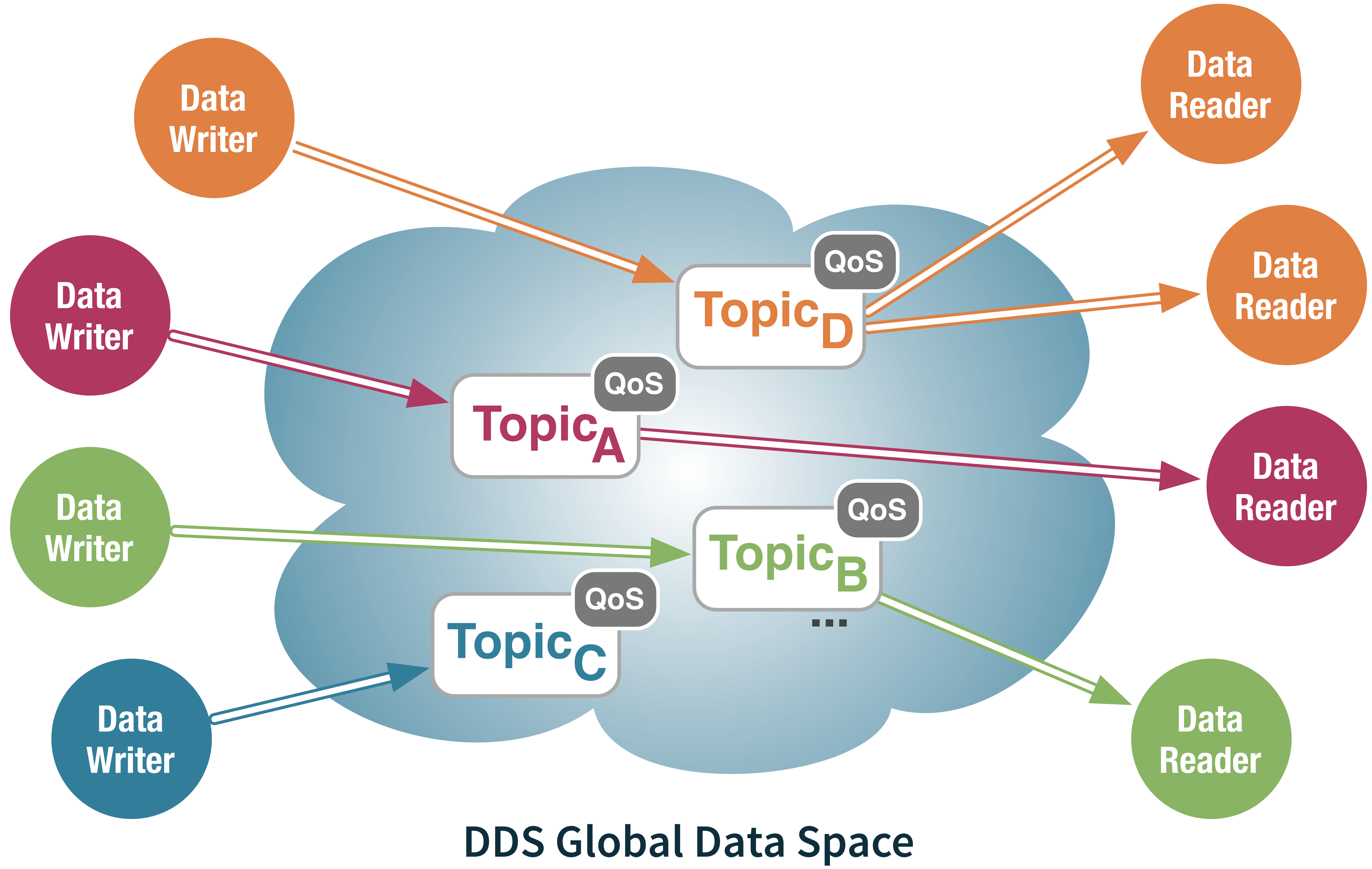
A domain-wide information's class

A **Topic** defined by means of a <name, type, qos>



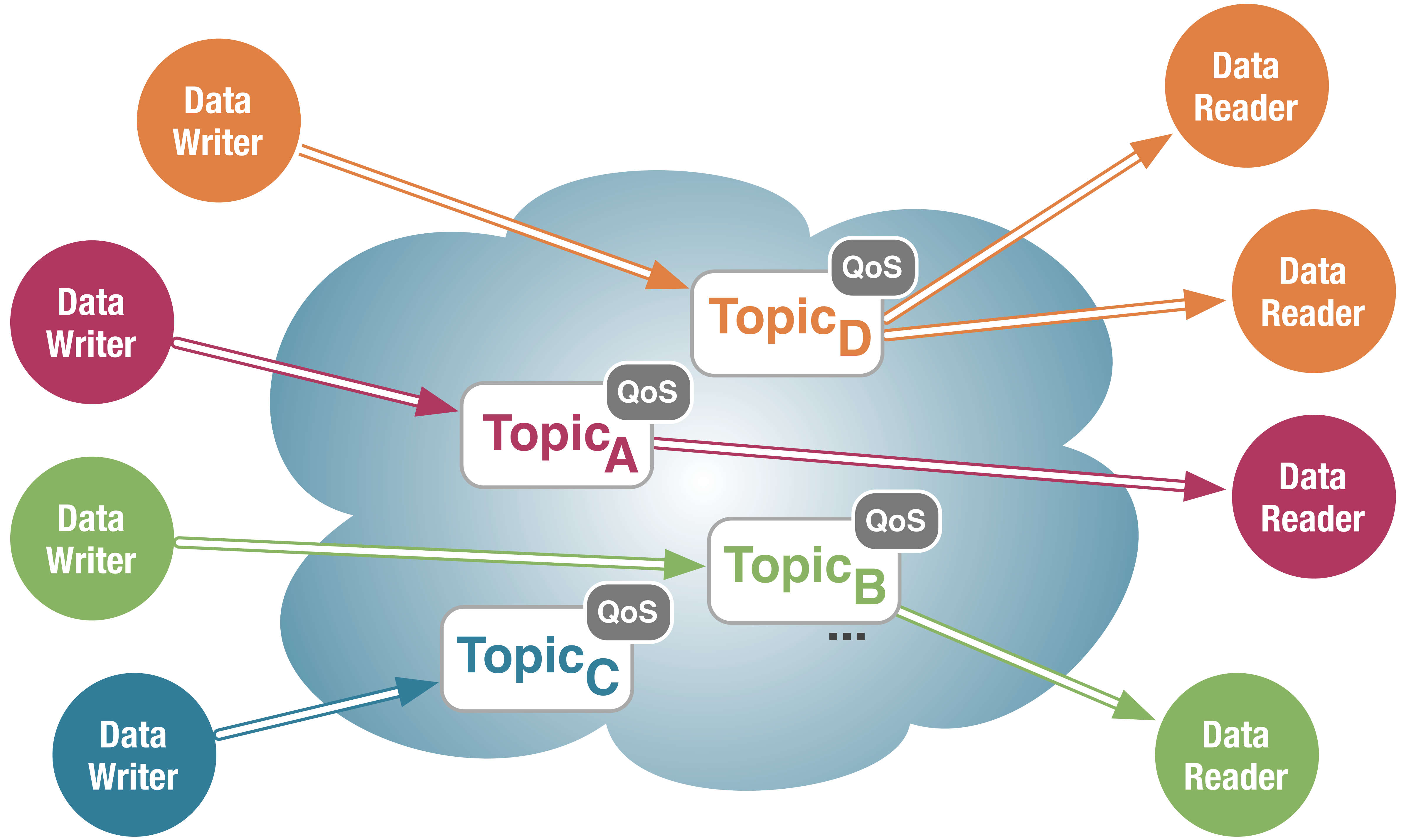
QoS Enabled

QoS policies allow to express **temporal** and **availability constraints** for data



Dynamic Discovery

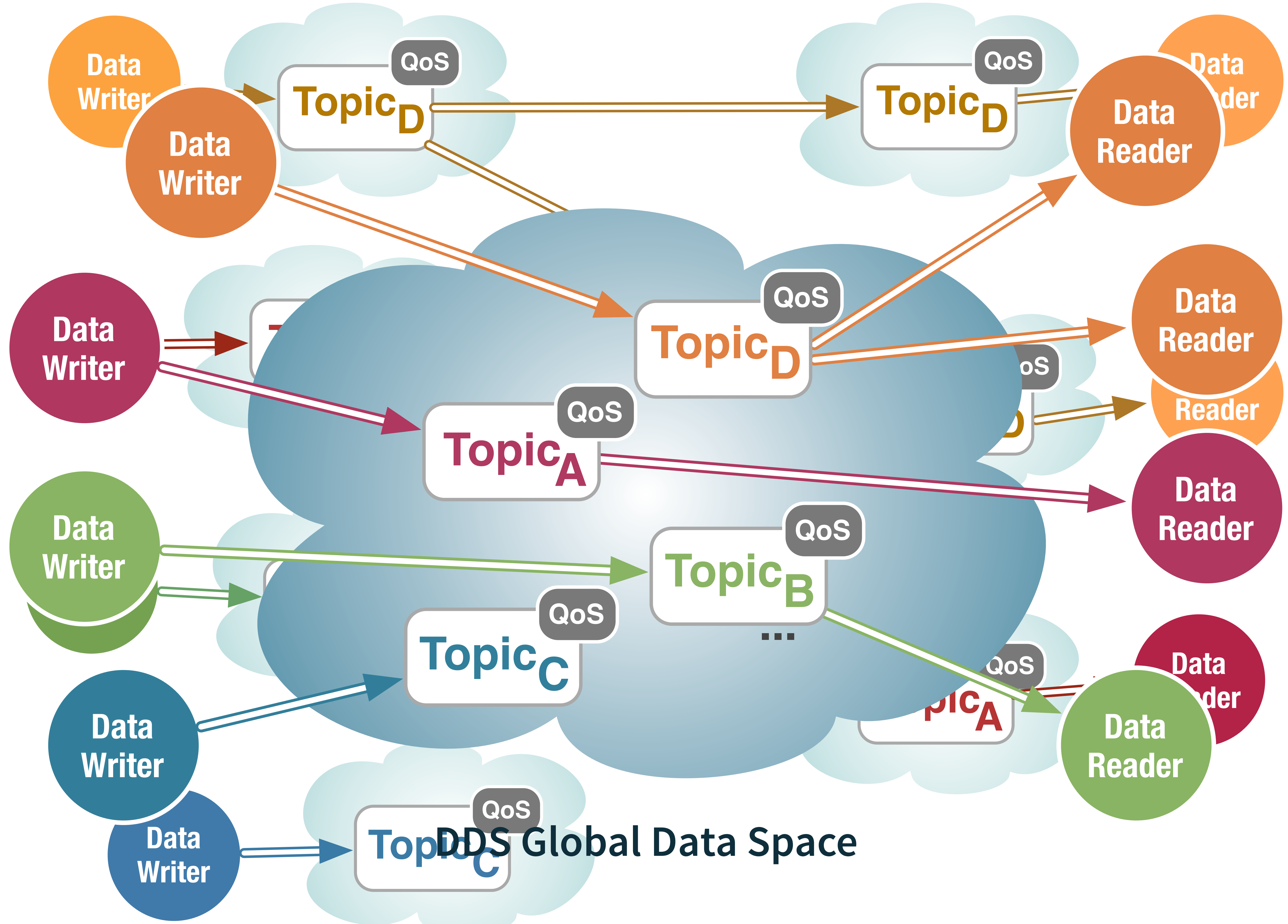
Built-in dynamic discovery **isolates applications from network topology** and **connectivity** details



DDS Global Data Space

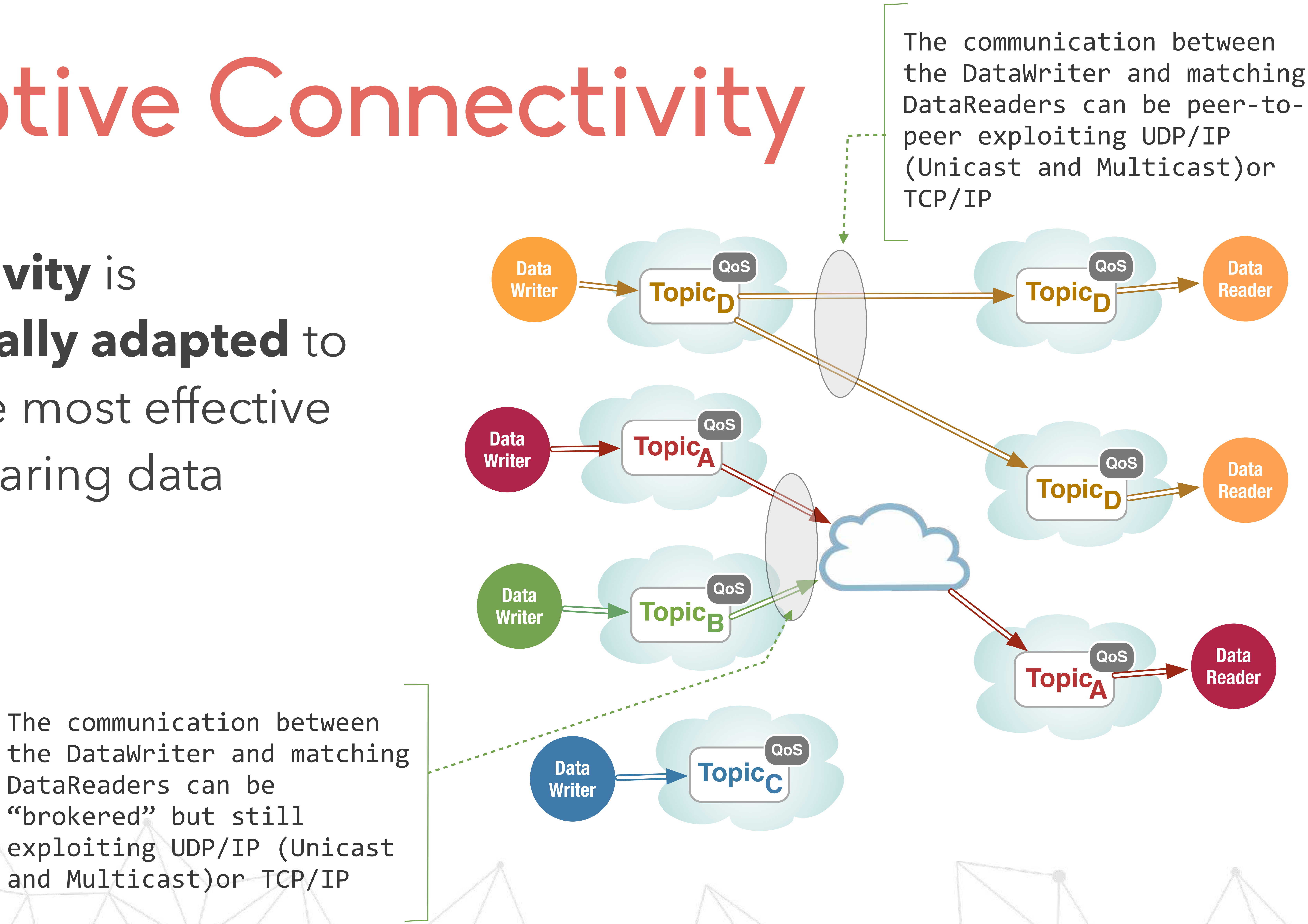
Decentralised Data-Space

No single point of failure or bottleneck



Adaptive Connectivity

Connectivity is **dynamically adapted** to chose the most effective way of sharing data



Lab 1
First App

```cpp
using namespace dds::core;
using namespace dds::core::policy;
using namespace dds::topic;
using namespace dds::pub;
using namespace dds::sub;

using namespace org::eclipse;

int
main(int argc, char* argv[])
{

  try {
    // Parse the command line args
    tspub_options opt = parse_tspub_args(argc, argv);

    // Create the domain participant
    dds::domain::DomainParticipant dp(cyclonedds::domain::default_id());

    // Initialize random number generation with a seed
    srandom(clock());

    dds::topic::qos::TopicQos tqos =
      dp.default_topic_qos() << LatencyBudget(Duration(2,0)) << Deadline(Duration(4,0));

    auto topic = Topic<tutorial::TempSensorType>(dp, "TempSensorTopic", tqos);
    auto pub = Publisher(dp);
    auto dw = DataWriter<tutorial::TempSensorType>(pub, topic, tqos);
```

```
[variant] Loaded new set of variants
[kit] Successfully loaded 1 kits from /Users/kydos/.local/share/CMakeTools/cmake-tools-kits.json
[main] Configuring folder: dds-tutorial
```

Cyclone DDS

Eclipse Cyclone DDS was born with the ambition of developing the **best DDS implementation ever**.

We are leveraging the **Open Source** Ecosystem to facilitate **user-driven innovation** and mitigate some of the downsides of Open Standards with Open Source and Open Innovation

We are **working with the community** to **establish** the **Go-To DDS implementation** for all captive markets

https://github.com/eclipse-cyclonedds/cyclonedds

Writing Data in C++

```cpp
#include <dds/dds.hpp>

int main(int, char**) {

  dds::domain::DomainParticipant dp(0);
  dds::topic::Topic<Meter> topic("SmartMeter");
  dds::pub::Publisher pub(dp);
  dds::pub::DataWriter<Meter> dw(pub, topic);

  while (!done) {
     auto value = readMeter()
     dw.write(value);
     std::this_thread::sleep_for(SAMPLING_PERIOD);
  }

  return 0;
}
```

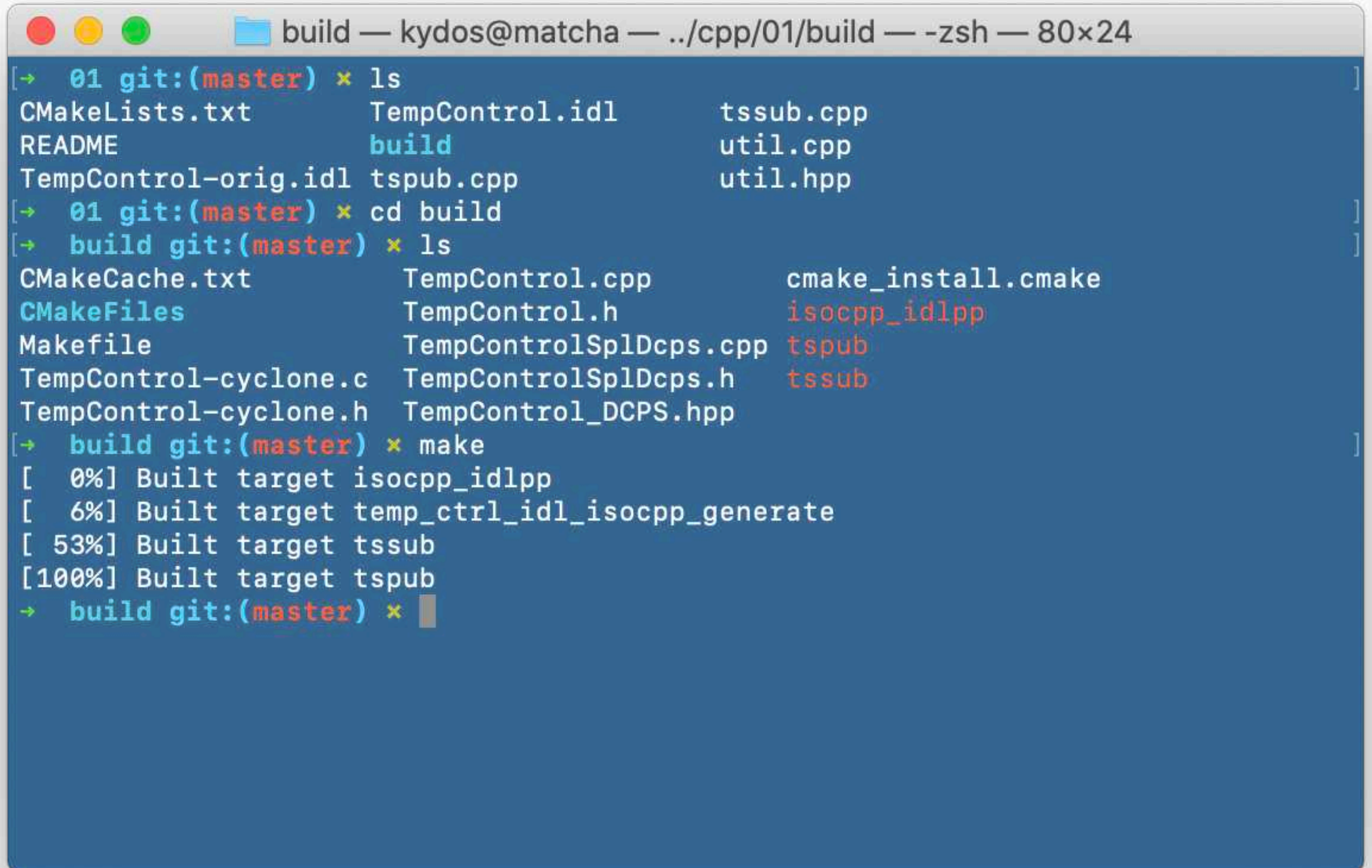
```
enum UtilityKind {
    ELECTRICITY,
    GAS,
    WATER
 };

 struct Meter {
    string sn;
    UtilityKind utility;
    float reading;
    float error;
 };
 #pragma keylist Meter sn
```

Reading Data in C++

```cpp
#include <dds/dds.hpp>

int main(int, char**) {

  dds::domain::DomainParticipant dp(0);
  dds::topic::Topic<Meter> topic("SmartMeter");
  dds::sub::Subscriber sub(dp);
  dds::sub::DataReader<Meter> dr(dp, topic);

  auto samples = dr.read();
  std::for_each(samples.begin(),
                samples.end(),
                [](Sample<Meter>& sample) {
                     std::cout << sample.data() << std::endl;
                 });

   return 0;
}
```

```
enum UtilityKind {
    ELECTRICITY,
    GAS,
    WATER
 };

 struct Meter {
    string sn;
    UtilityKind utility;
    float reading;
    float error;
 };
 #pragma keylist Meter sn
```

Console Time!



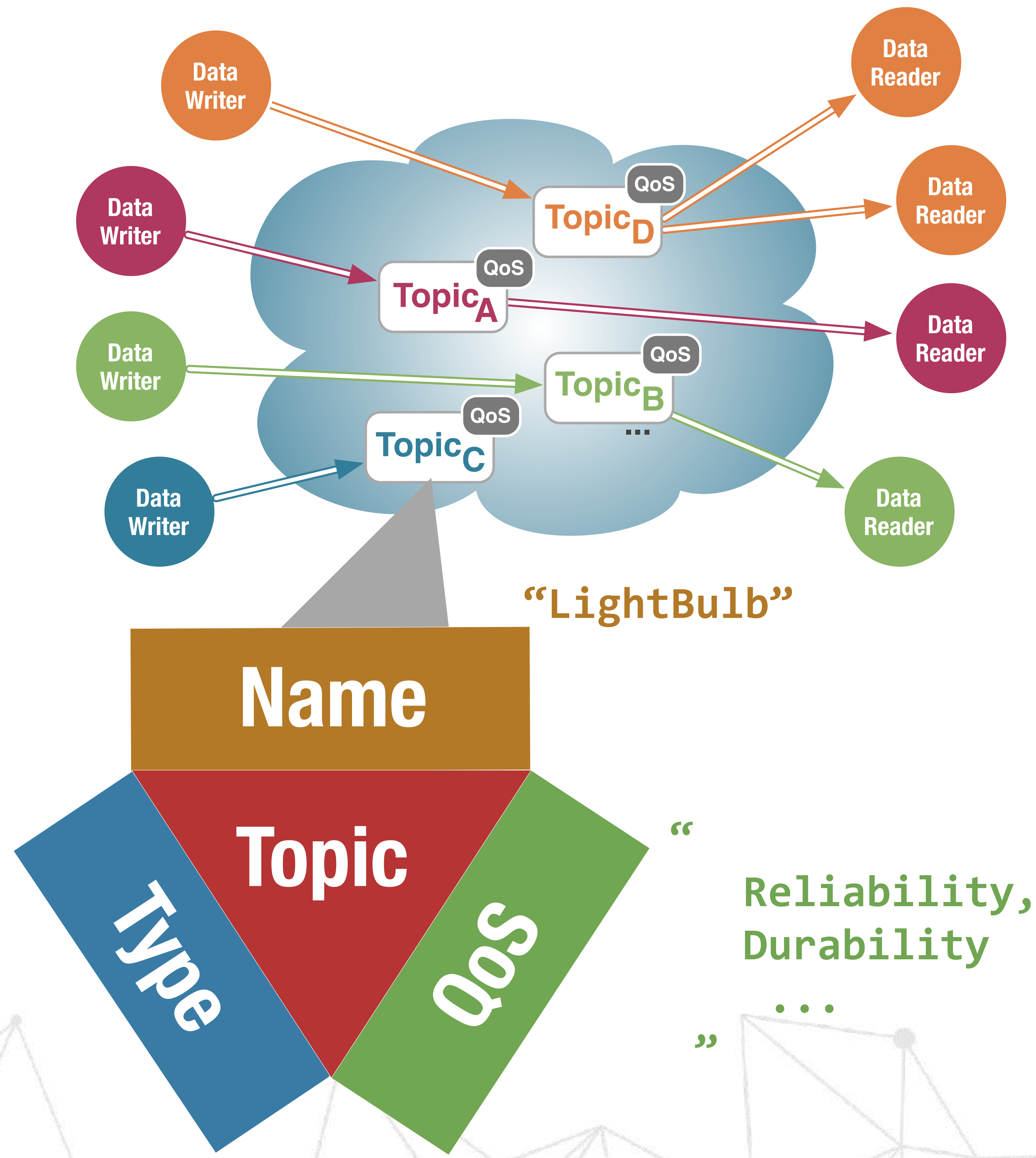
DDS:
Going Deeper

Topics

DDS data streams are defined by means of **Topics**

A **Topic** represented is by means of a <name, type, qos>

```
struct LightBulbState {
    string sn;
    float luminosity;
    long hue;
    boolean on;
};
#pragma keylist LightBulgState sn
```

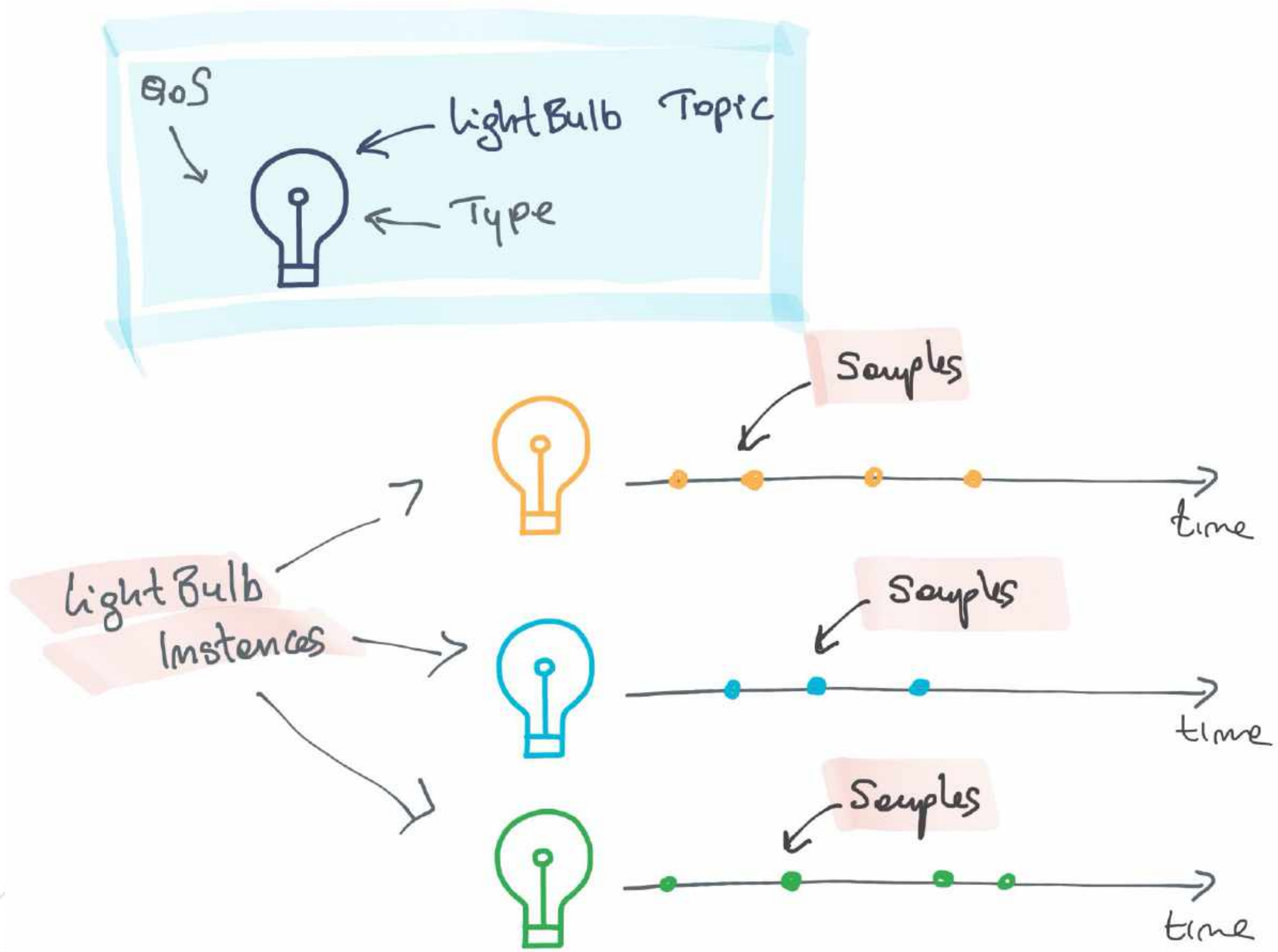


Topic Instances

Topic may **mark some** of their associated type **attributes** as **key-fields**

Each unique key value (tuple of key attributes) identifi es a Topic Instance. Each Topic Instance has associated a FIFO ordered stream of samples

DDS provides useful **instance life-cycle management** and samples demultiplexing



Topics and Relations

A **topic** cans be seen as defining a **relation**

```
struct LightBulbState {
   string sn;
   float luminosity;
   long hue;
   boolean on;
};
#pragma keylist LightBulgState sn
```

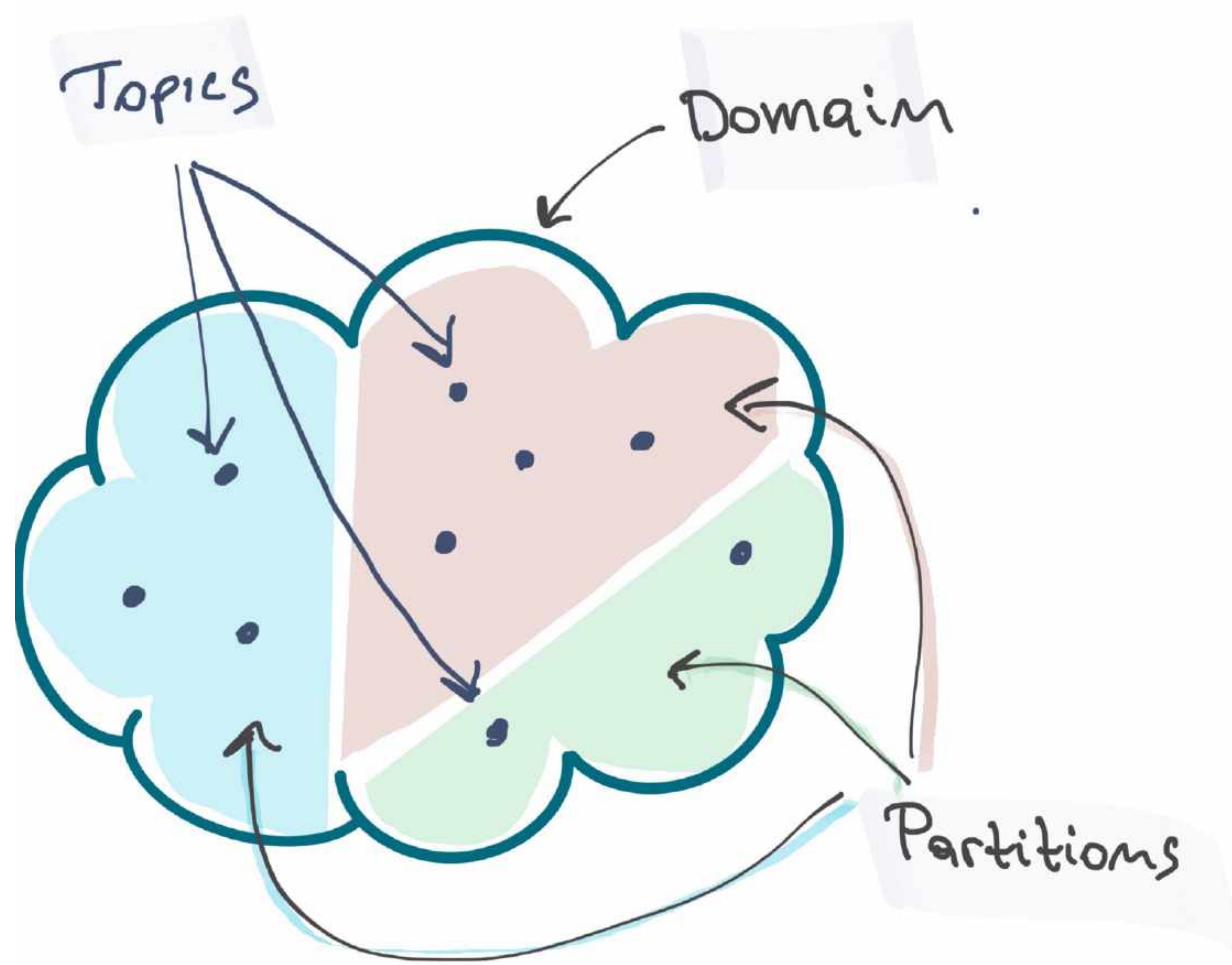
sn	luminosity	hue	on
a123-21ef	0.5	12750	TRUE
600d-caf3	0.8	46920	FALSE
1234-c001	0.75	25500	TRUE

Information Scopes

DDS information lives within a **domain**

A domain can be thought as organised in **partitions**

Samples belonging to a given **Topic Instance** are read/written from/in one or more **partitions**



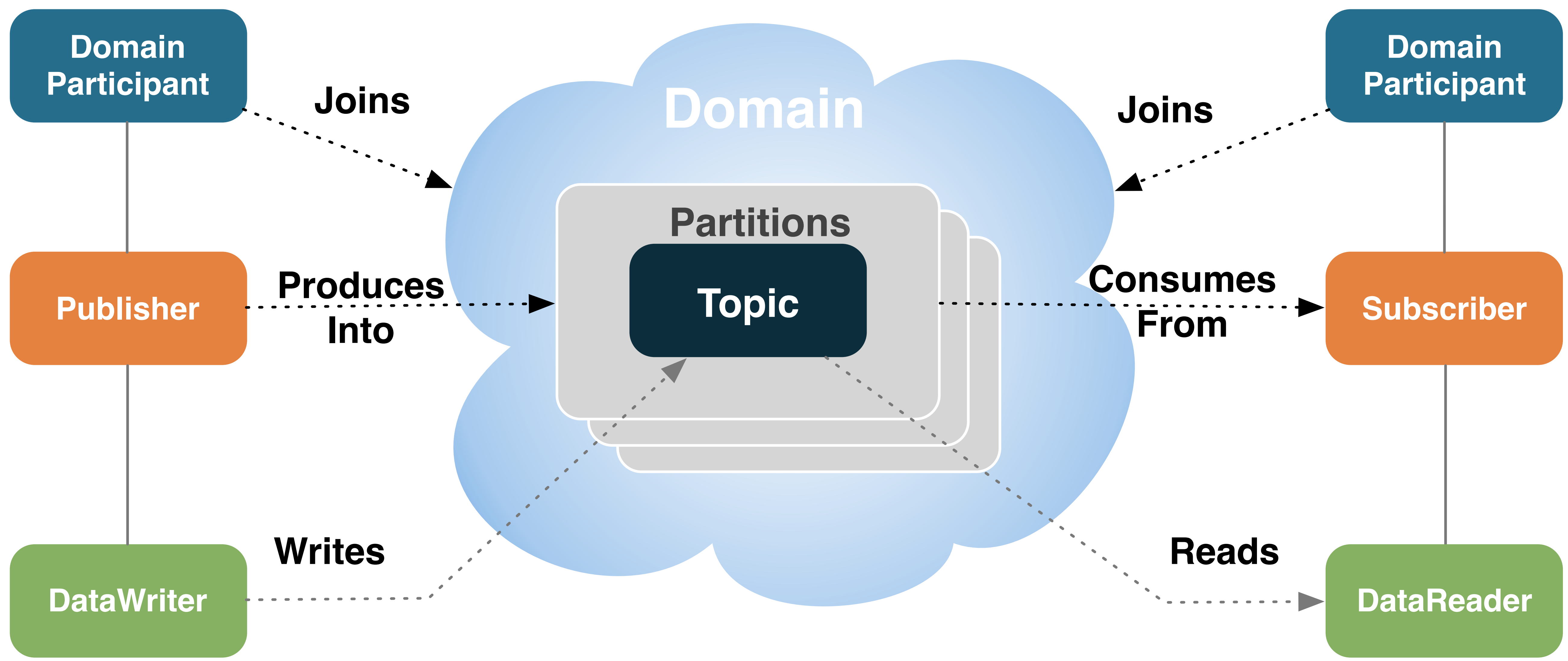
DDS Entities

DDS Entities

DDS provides three different entities to control **where** and **what** data is read/written

The **DomainParticipant**, **Publisher** and **Subscriber** relate to the "**where**"

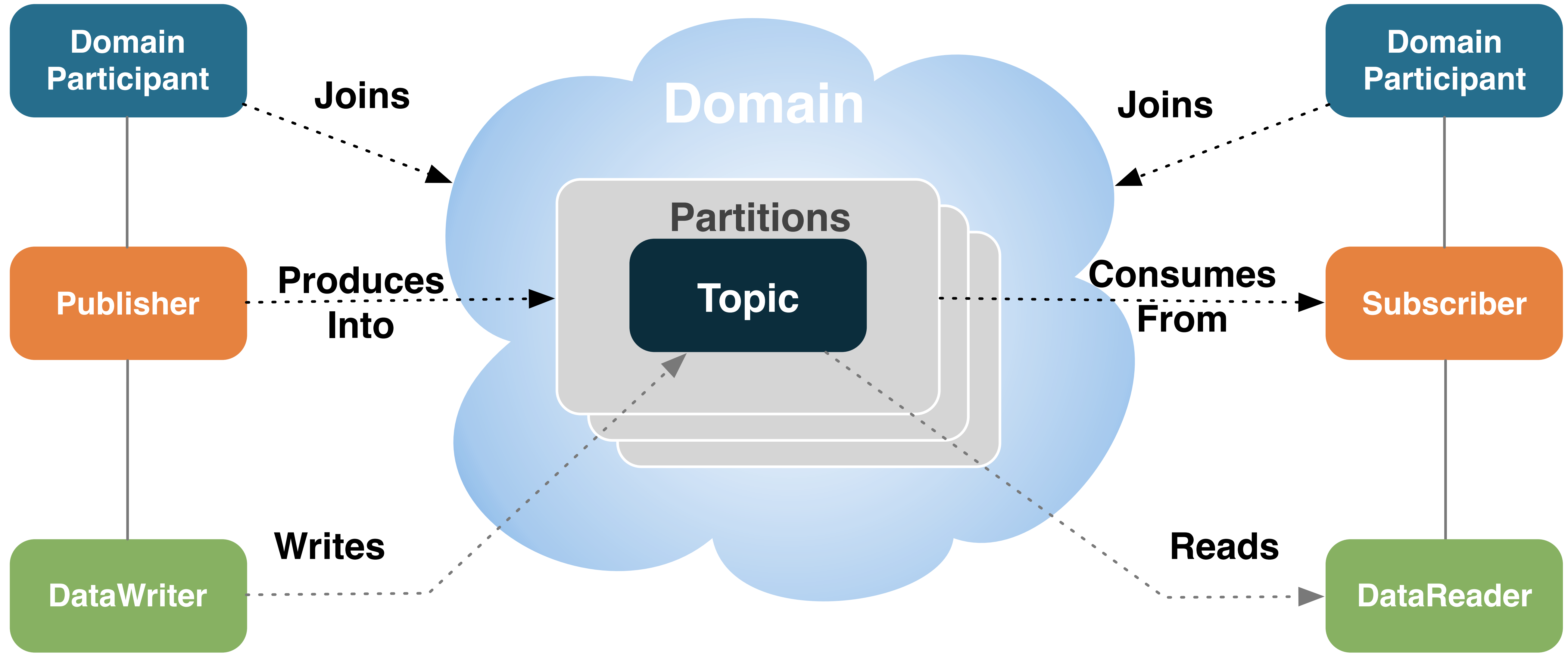
DataReader and **DataWriter** relate to the "**what**"



DDS Entities

DDS **QoS Policies** control, at a large extent, to the **"how" data** is **shared**

QoS Policies also control **resource utilisation**



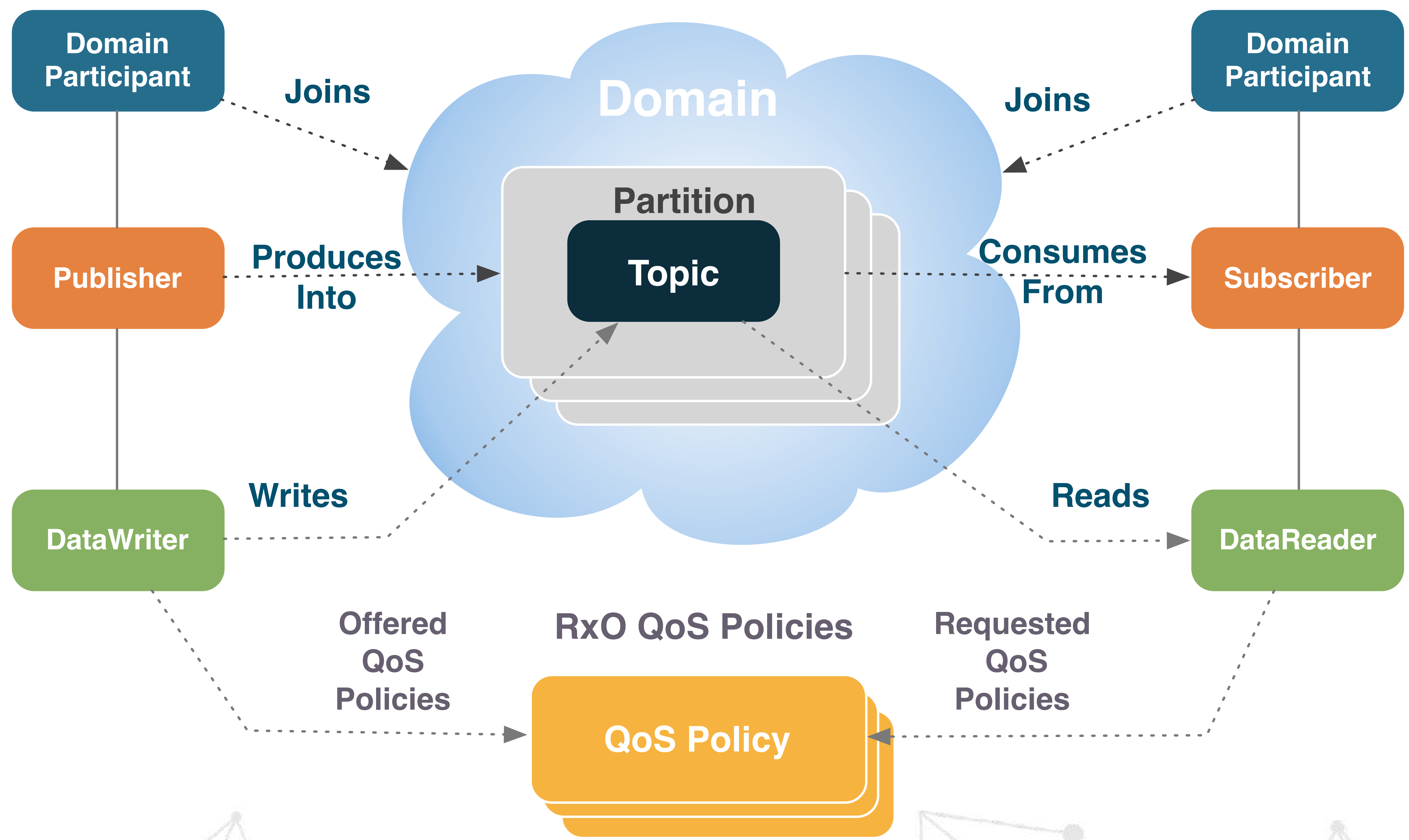
Matching Model

For data to flow from a DataWriter (DW) to one or many DataReader (DR) a few conditions have to apply:

The **DR** and **DW** have to be in the **same domain**

The **partition expression** of the DR's Subscriber and the DW's Publisher should **match** (in terms of regular expression match)

The **QoS Policies offered** by the DW should **exceed or match** those **requested** by the DR

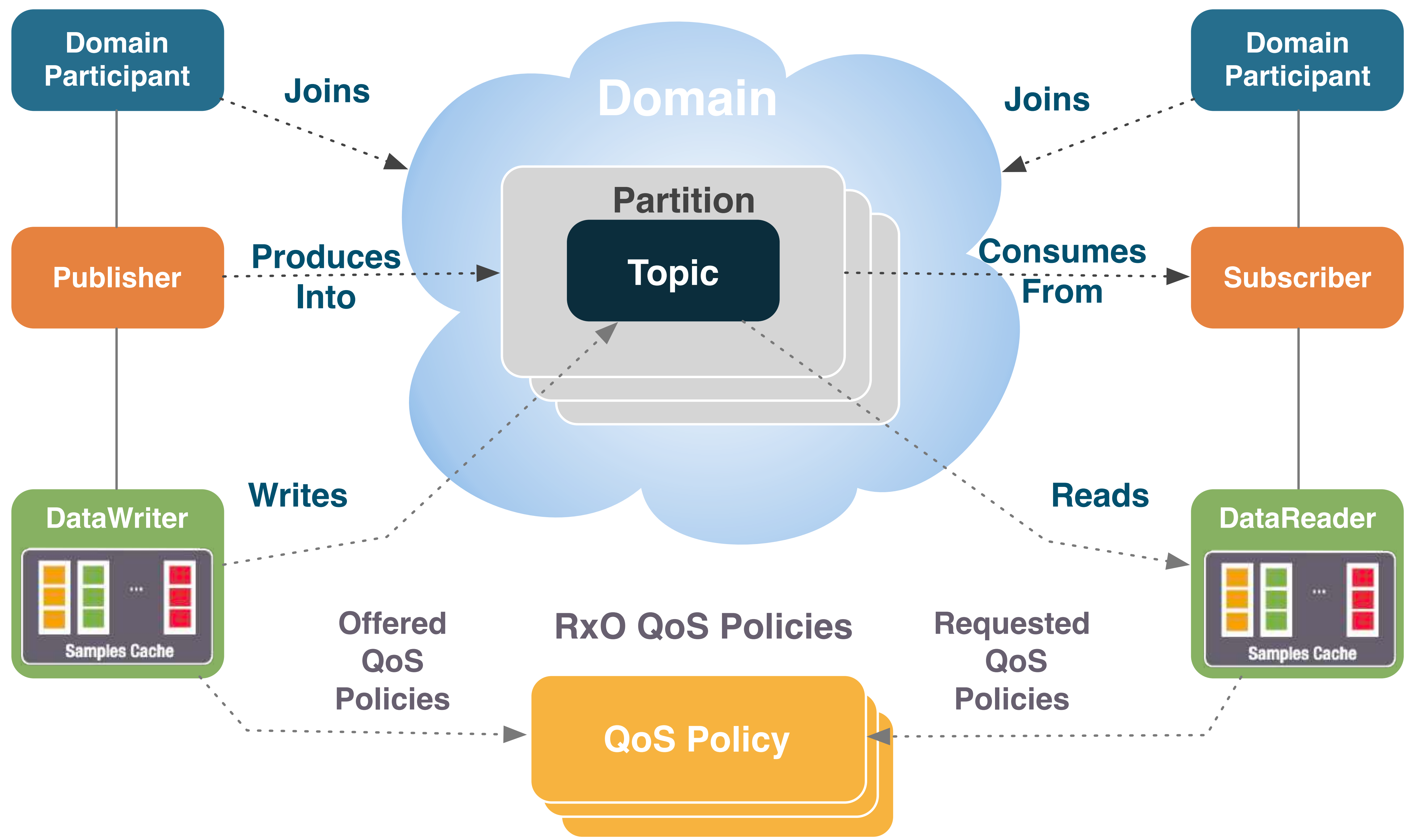


Storage Model

DataWriter and DataReaders have an associated samples cache

In a sense, what DDS does is to project, eventually, the relevant content of the writer cache into matching reader caches

As a consequence of these caches reads and writes are always local and non-blocking*



* **reads** never block and a write will only block, depending on QoS settings if sufficient resources are not available

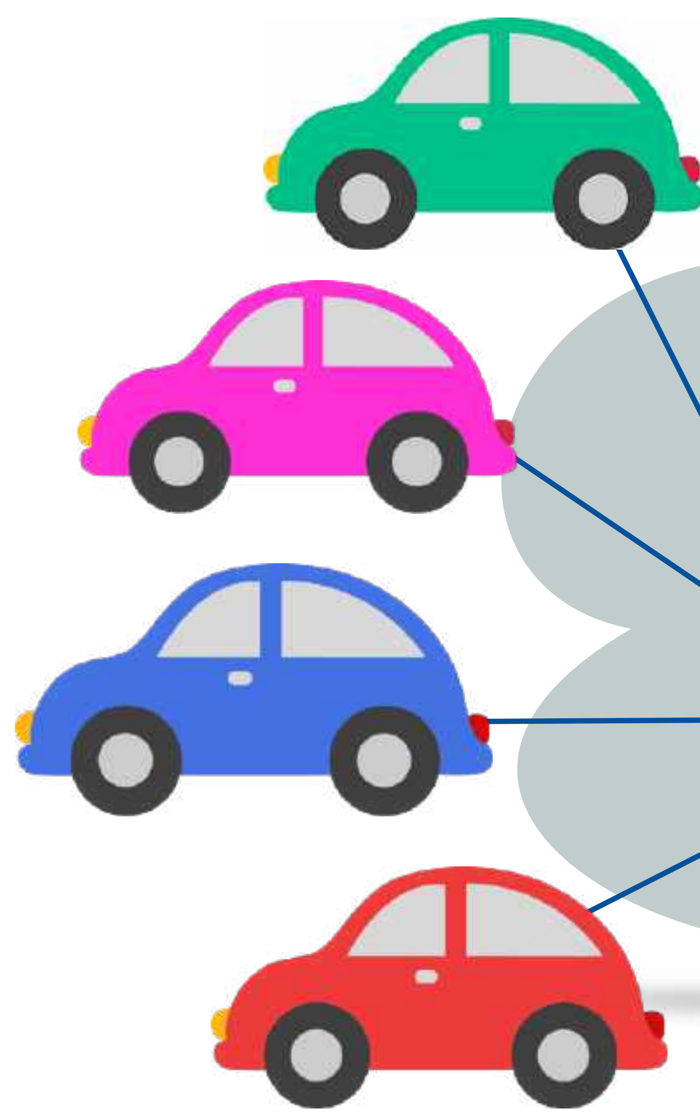
Content Awareness

Content Filtering

Content Filters can be used to **project** on the local cache only the Topic data satisfying a given predicate

Type

```
struct CarDynamics {
   @key
   string  cid;
   long    x;   long  y;
   float   dx;  long  dy;
}
```



CarDynamics

cid	x	y	dx	dy
GR 33N GO	167	240	45	0
LO 00V IN	65	26	65	0
AN 637 OS	32	853	0	50
AB 123 CD	325	235	80	0

`"dx > 50 OR dy > 50"`

cid	x	y	dx	dy
LO 00V IN	65	26	65	0
AB 123 CD	325	235	80	0

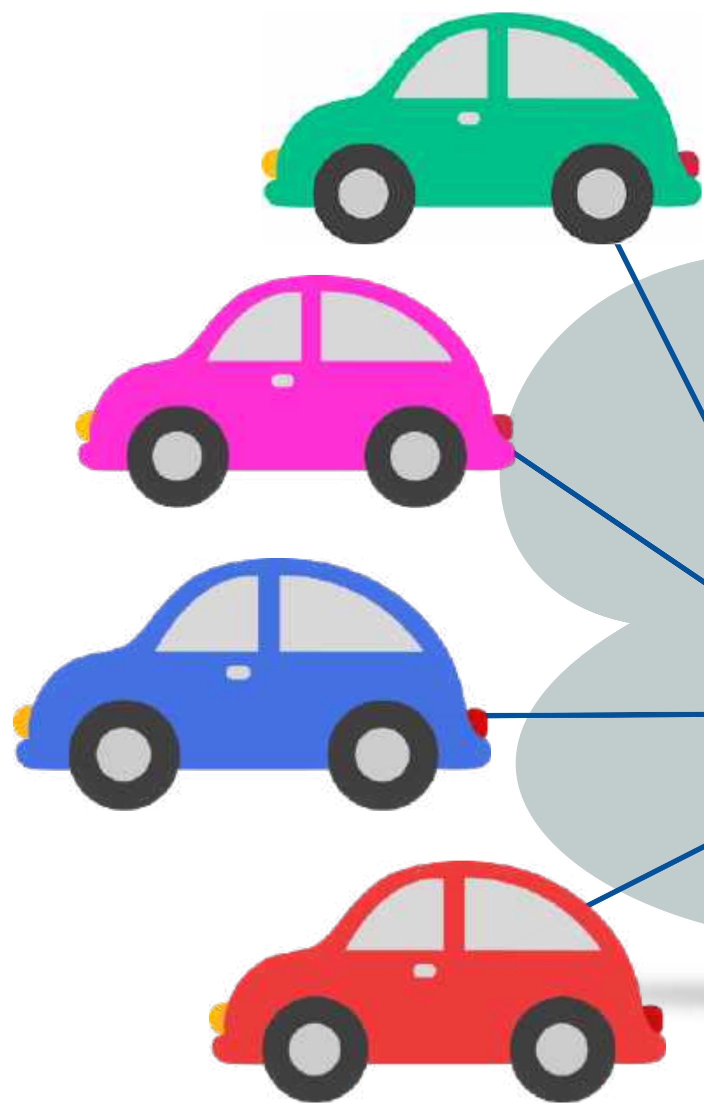
Reader Cache

Queries

Queries can be used to select out of the local cache the data matching a given predicate

Type

```
struct CarDynamics {
   @key
   string  cid;
   long    x;    long  y;
   float   dx;   long  dy;
}
```



CarDynamics

cid	x	y	dx	dy
GR 33N GO	167	240	45	0
LO 00V IN	65	26	65	0
AN 637 OS	32	853	0	50
AB 123 CD	325	235	80	0

cid	x	y	dx	dy
LO 00V IN	65	26	65	0
AB 123 CD	325	235	80	0

"dx > 50 OR dy > 50"
query

cid	x	y	dx	dy
GR 33N GO	167	240	45	0
LO 00V IN	65	26	65	0
AN 637 OS	32	853	0	50
AB 123 CD	325	235	80	0

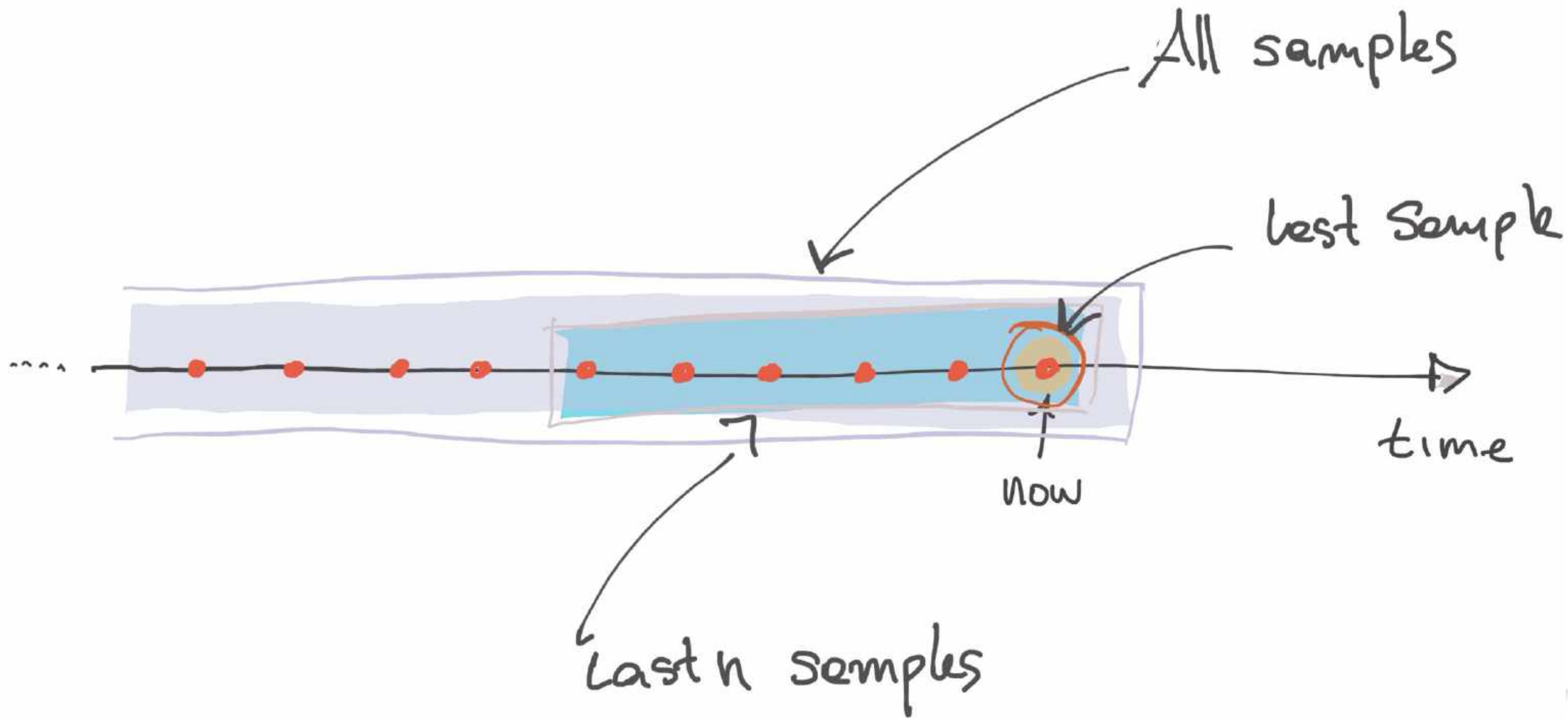
Reader Cache

Stream Durability

Stream Durability

Through QoS settings it is possible to control which subset of the stream data will be **retained** and **made available** *(replayed)* **to late joiners**

DDS can store the last **n** samples (n=1 is a special case) or all the samples written for a topic



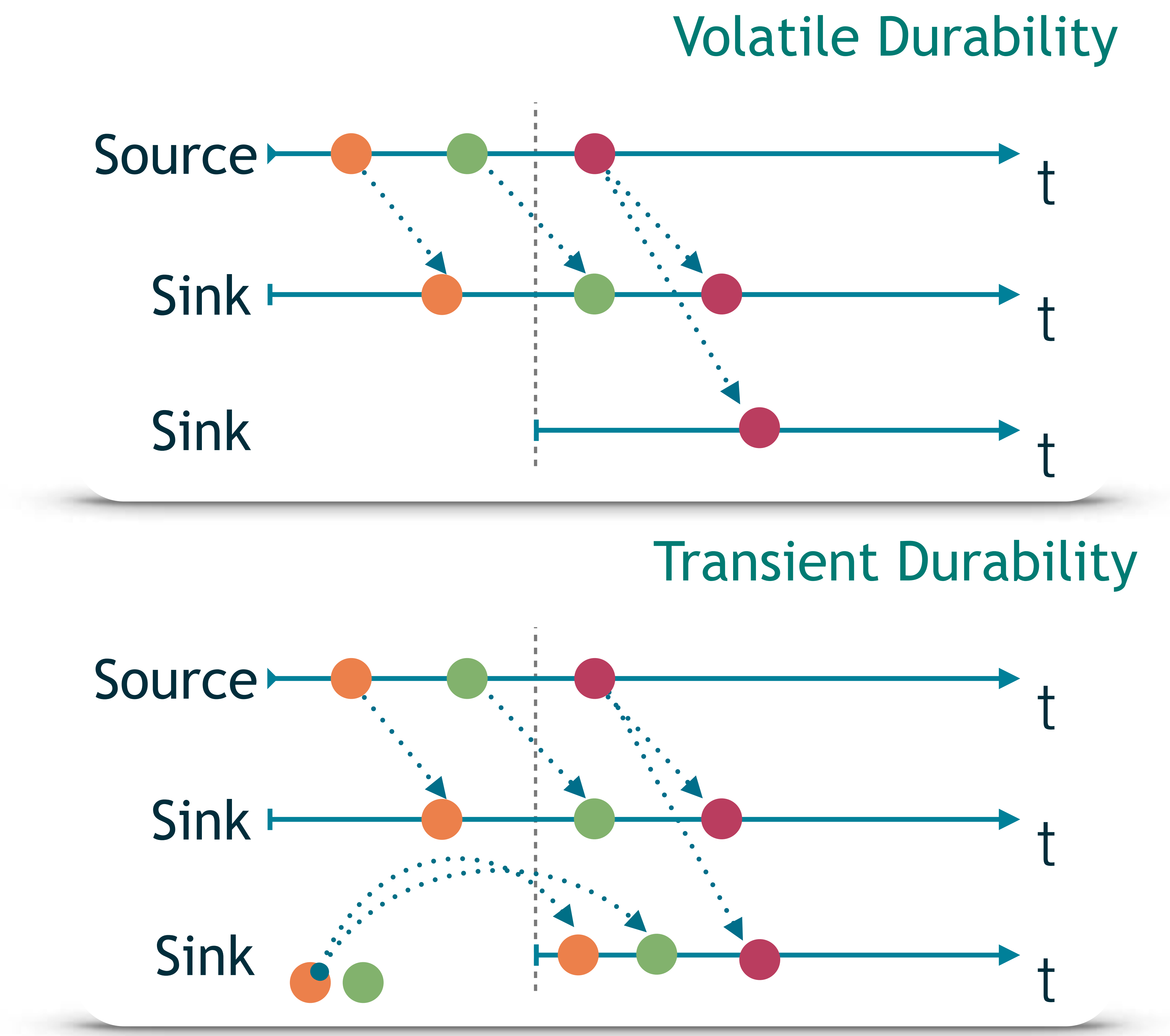
Stream Durability

DDS provides three kinds of durability:

Volatile: i.e. no durability

Transient (Local): data is available for late joiners (re-play) as far as the system (data source) is running

Durable: data is available for late joiners (re-play) as far as the system/source is running

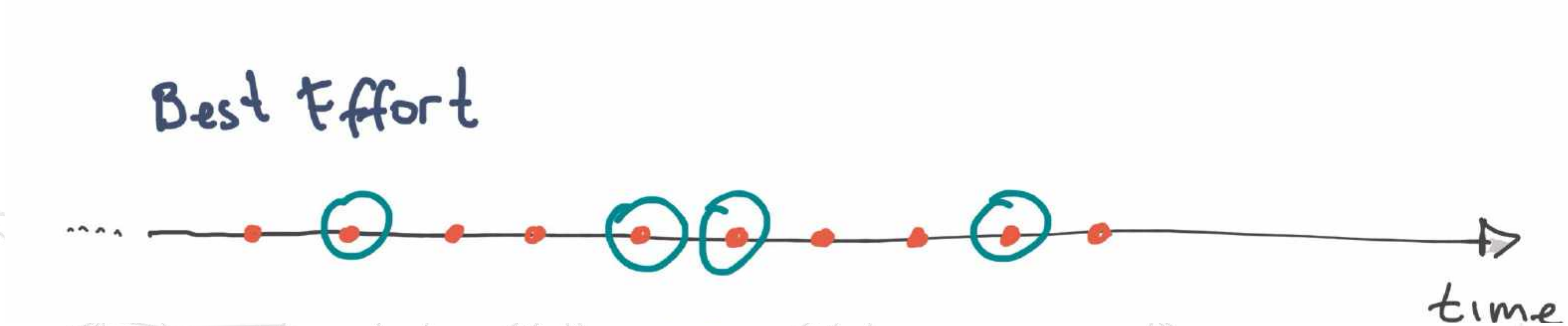


Stream Reliability

Best Effort

DDS will deliver an arbitrary subsequence of the samples written against a Topic Instance

Samples may be dropped because of network loss or because of flow-control

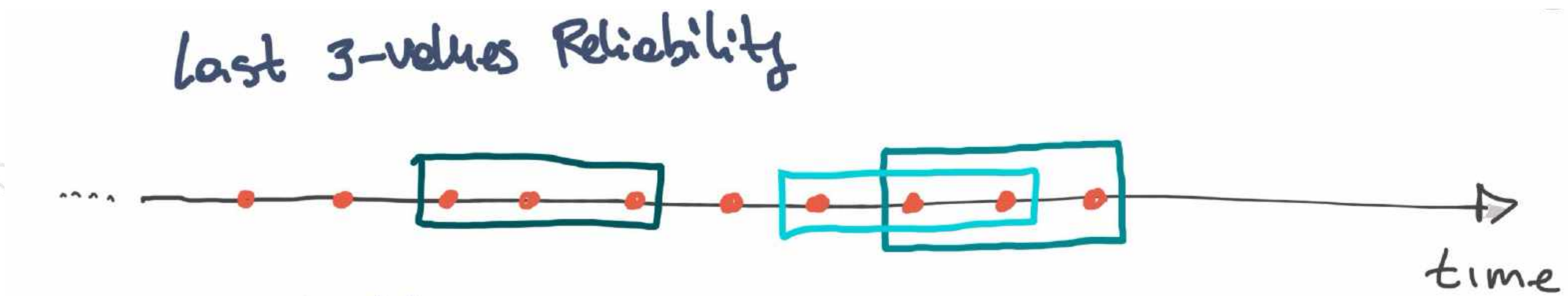


Last n-values Reliability

Under stationary conditions an application is guaranteed to receive the last n-samples written for a Topic Instance

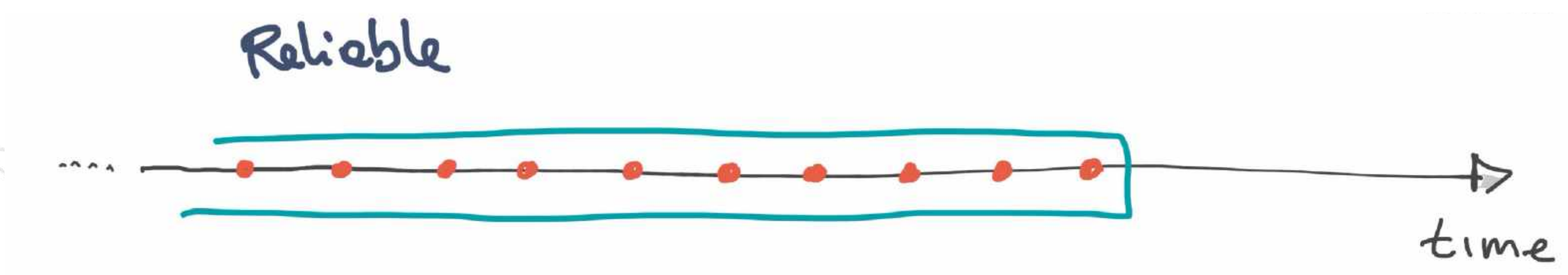
Samples falling outside the history may be dropped at the sending or receiving side for flow/resource control

Notice that this kind of reliability behaves as a circuit breaker for slow consumers



Reliable

All samples written against a Topic Instance are delivered. Since from a theoretical perspective reliability in asynchronous systems either violate progress or requires infinite memory, DDS provides QoS to control both resources as well as blocking time



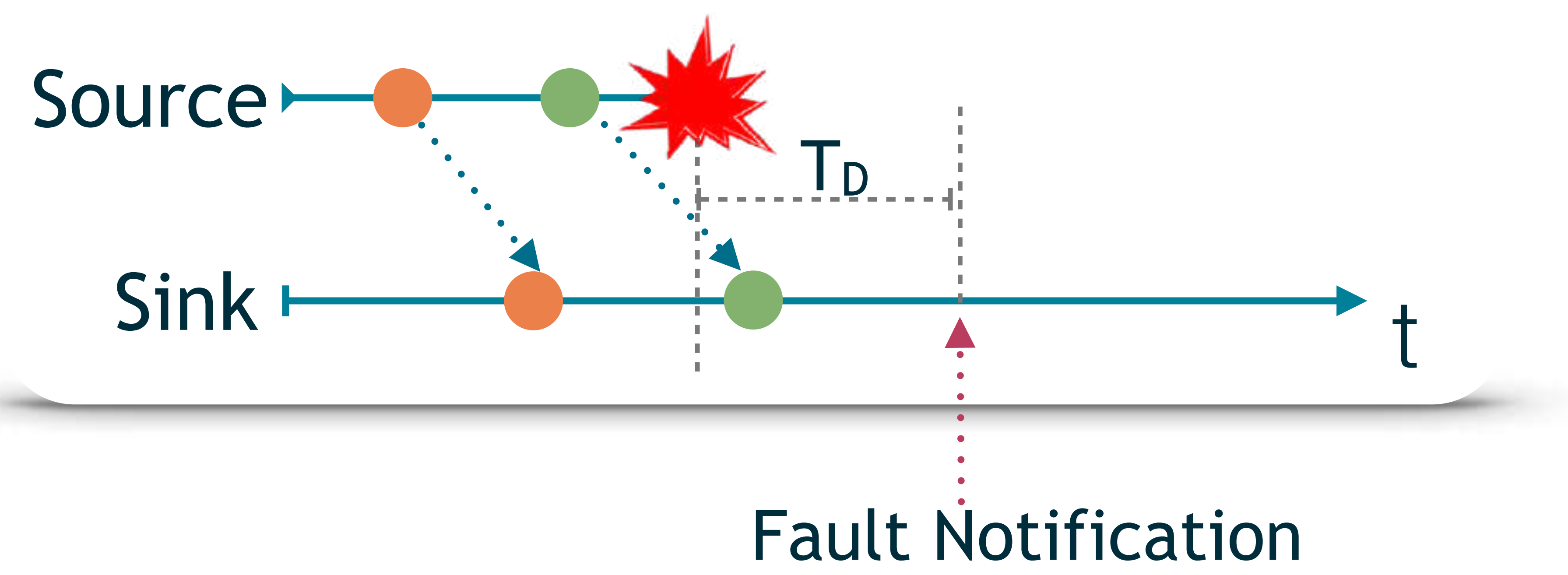
Fault-Tolerance

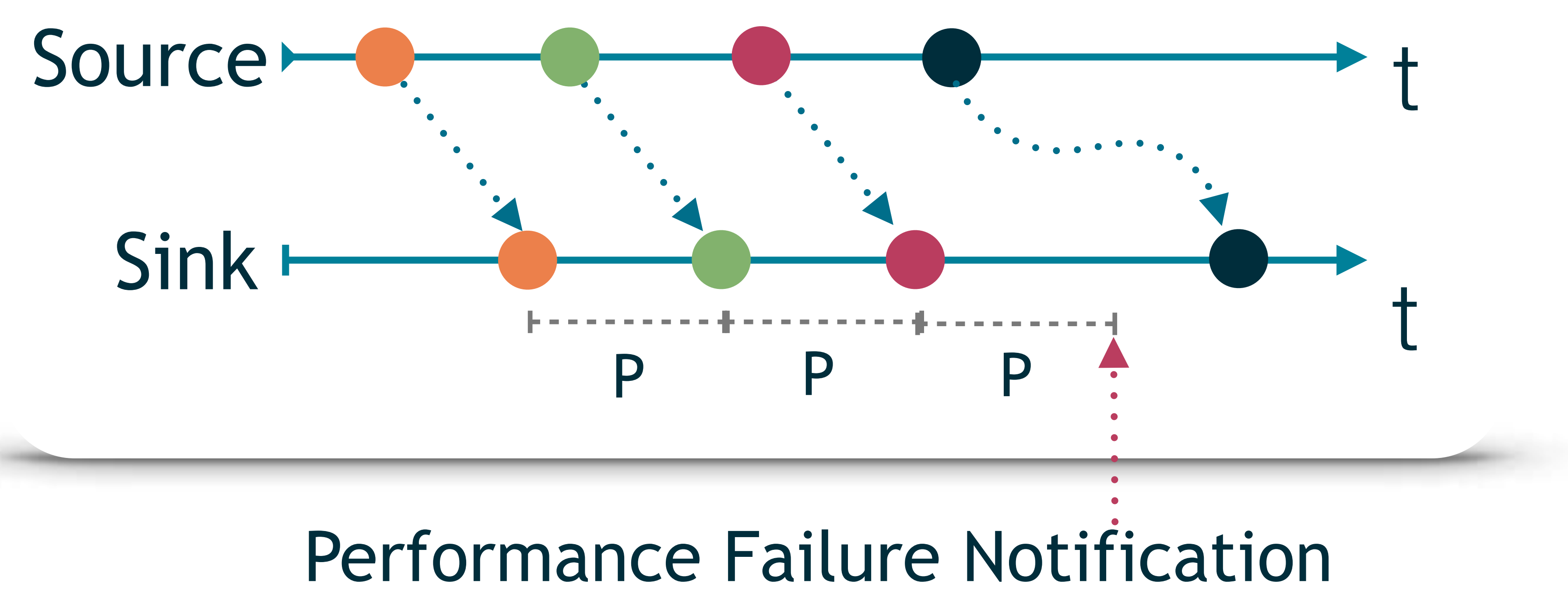
Failure Detection

DDS provides mechanism for detecting traditional faults as well as performance failures

The Fault-Detection mechanism is controlled by means of the DDS Liveliness policy

Performance Failures can be detected using the Deadline Policy which allows to receive notification when data is not received within the expected delays

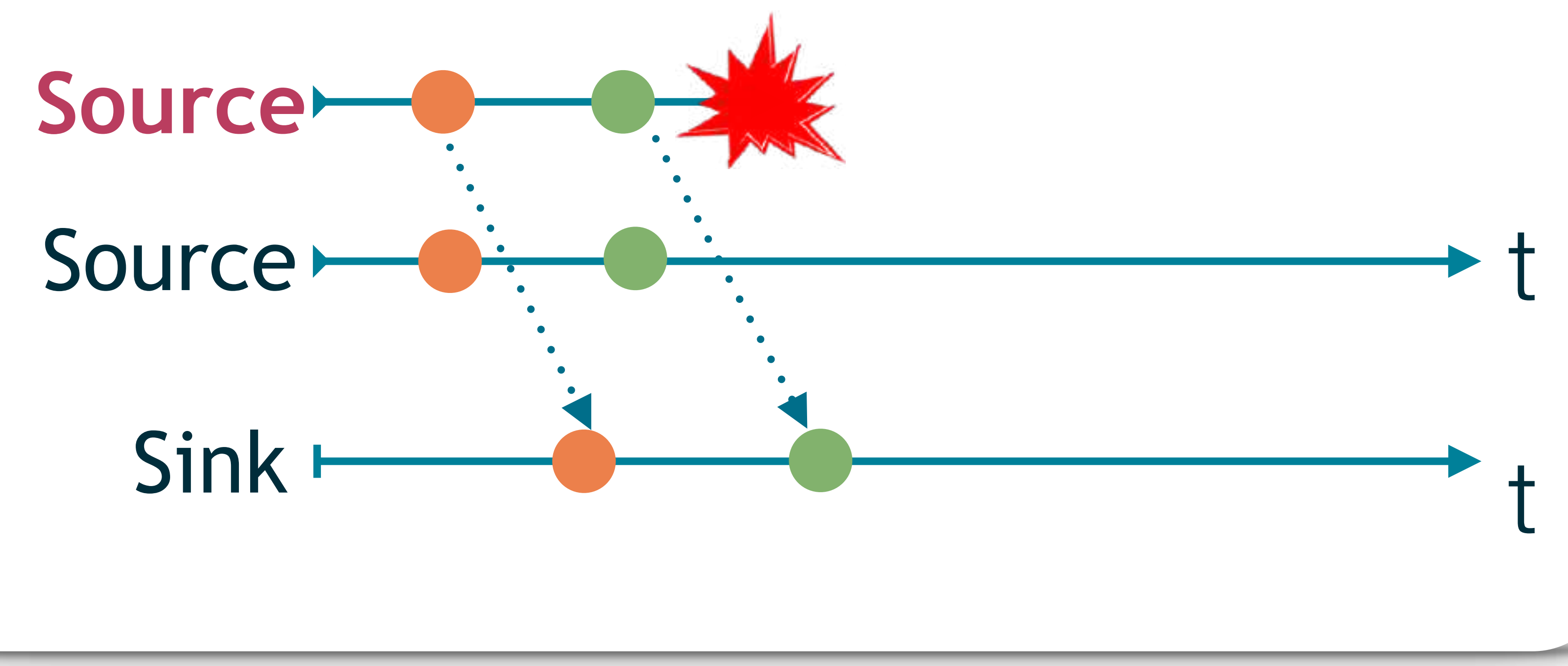




Fault-Masking

DDS provides a built-in fault-masking mechanism that allow to replicate **Sources** and transparently switch over when a failure occurs

At any point in time the “active” source is the one with the highest strength. Where the strength is an integer parameter controller by the user

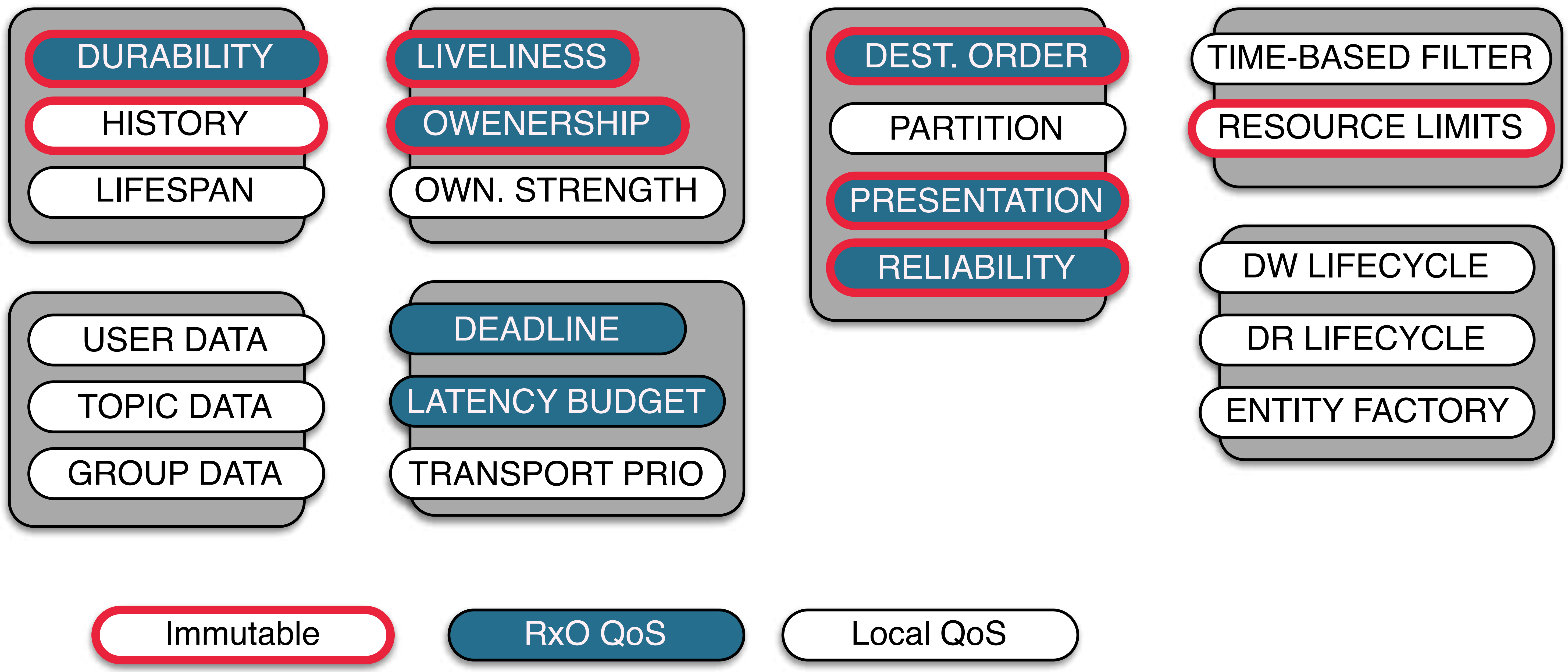


QoS Policies

DDS QoS Policies

DDS provides 20+ standard QoS Policies

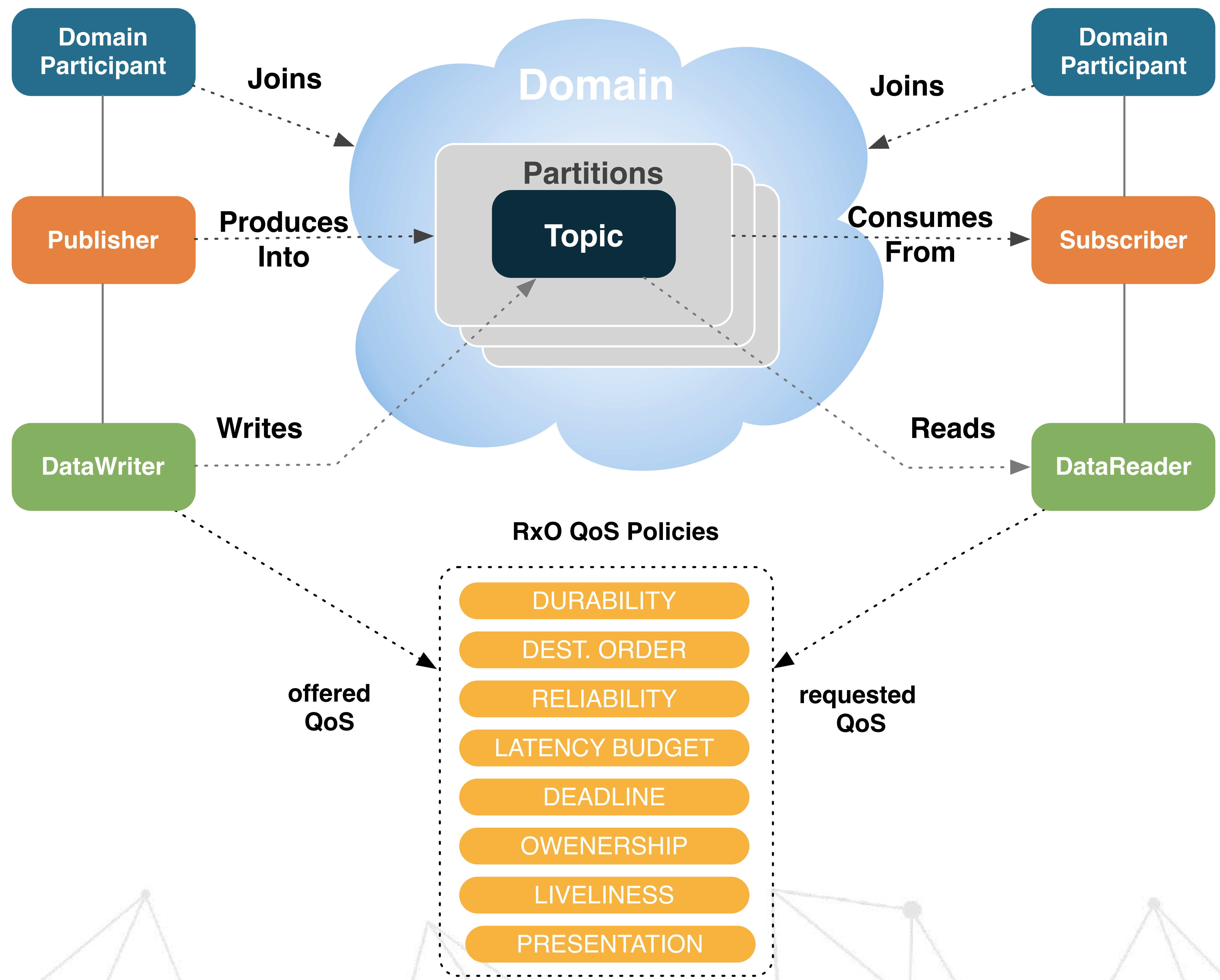
While this may seem a lot of complexity they are often used in combination to achieve certain patterns



RxO Model

A subset of the QoS Policies impact the matching

Most of thee QoS relate to end-to-end properties of data

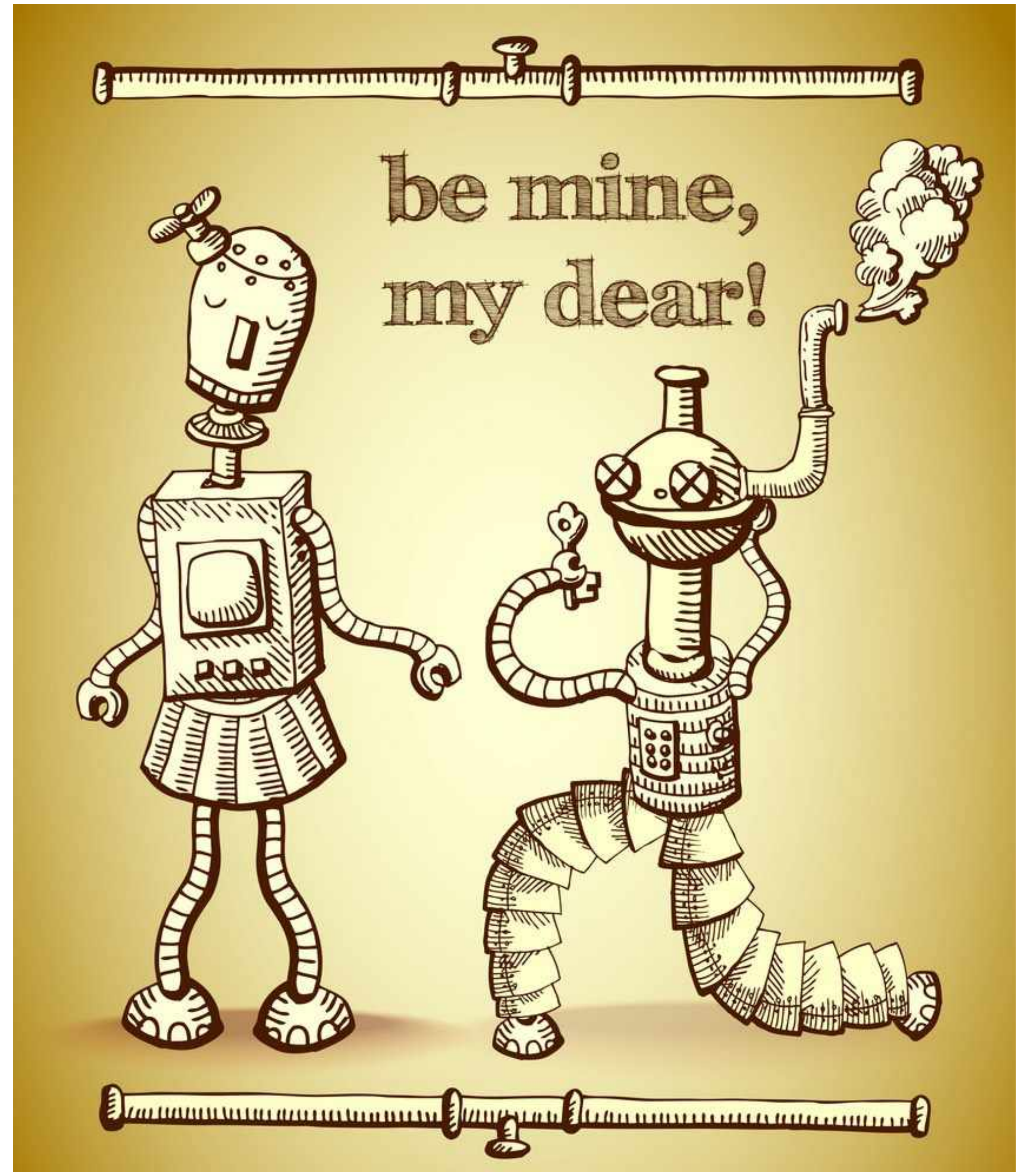


DDS in Robotics

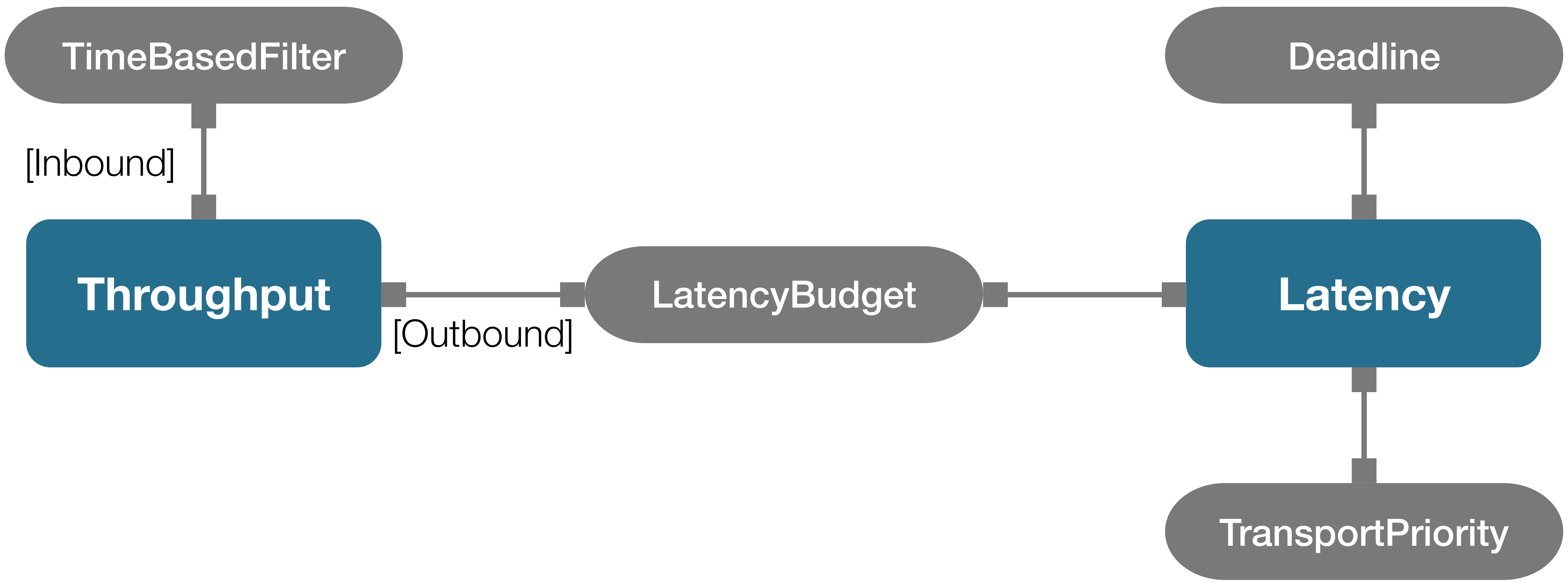
Happy Marriage

DDS is a perfect match for Robotics applications and frameworks, such as ROS2 because of:

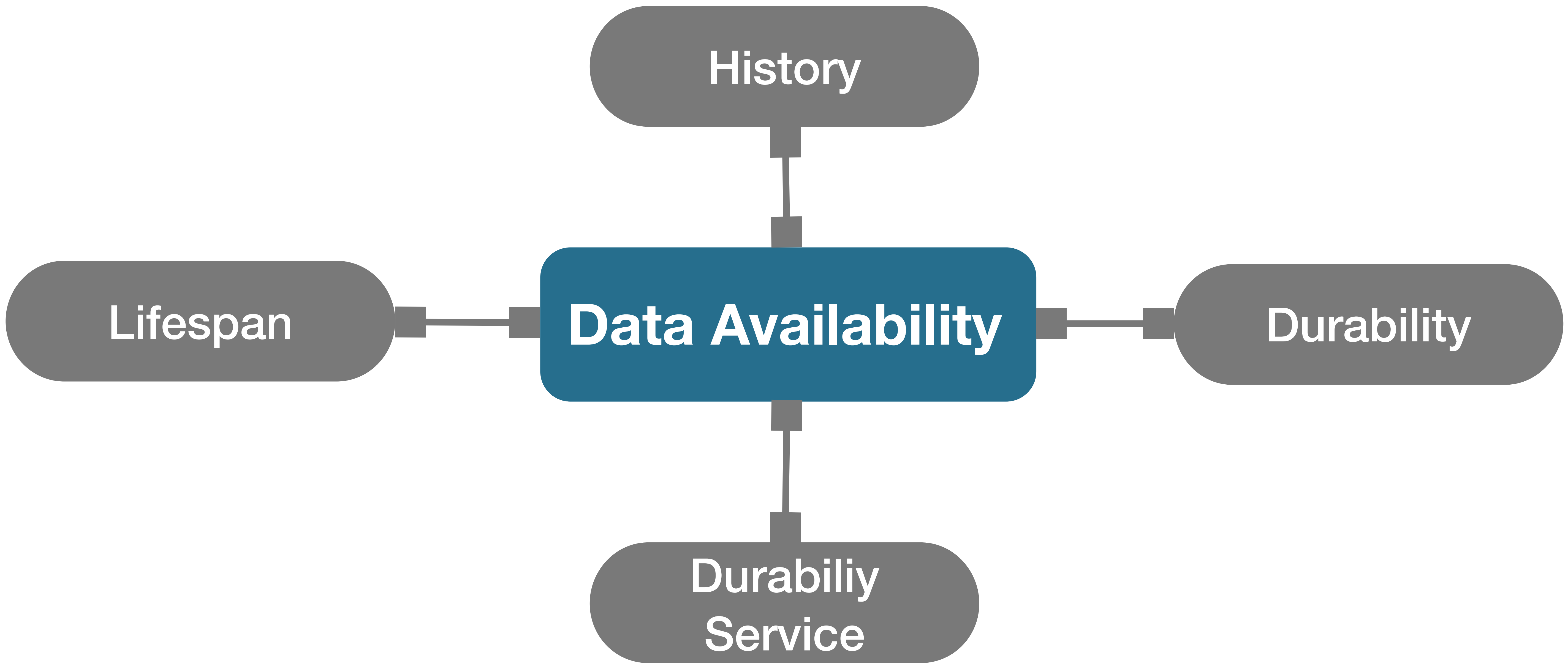
- Data Centric Abstraction
- Decentralised Architecture
- Real-Time Behaviour
- Performance
- QoS-Driven non-functionals



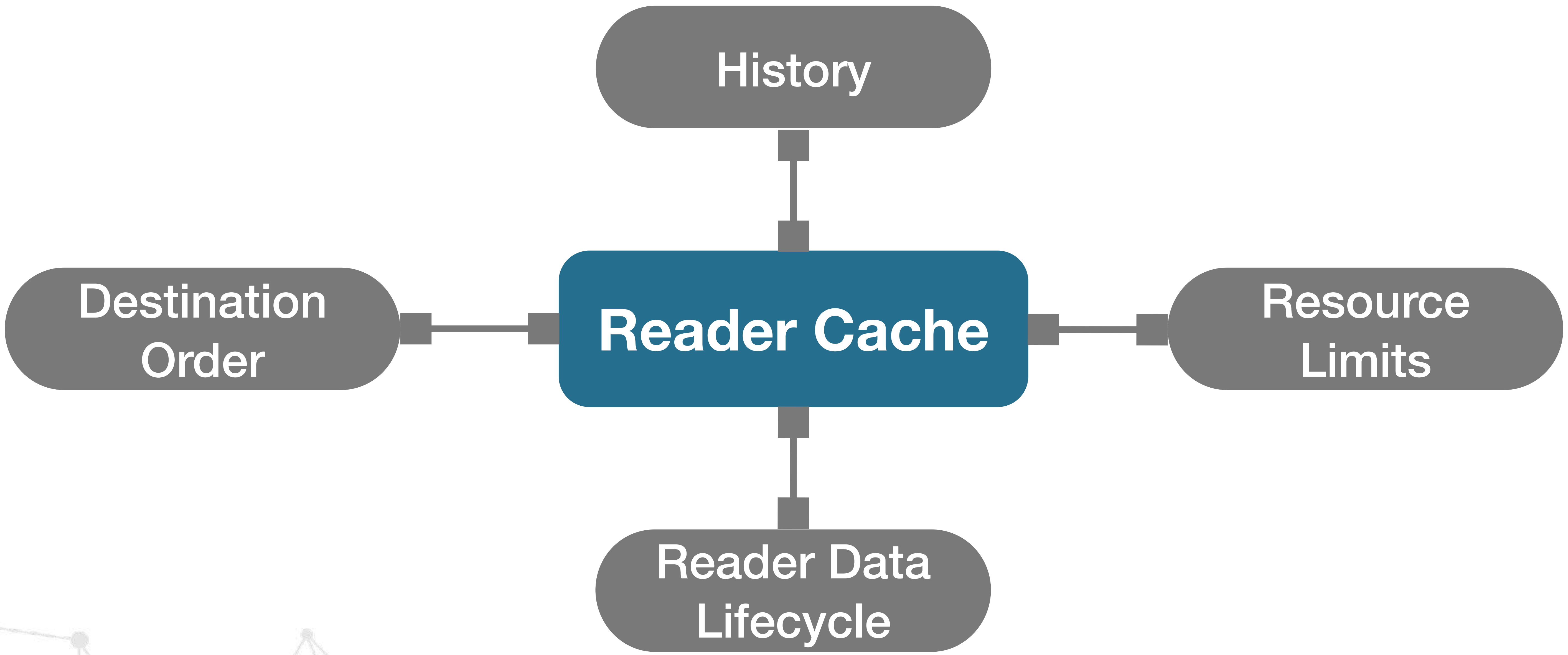
Temporal Properties



Data Availability



Reader Cache



QoS Modeling Idioms

State vs. Events

DDS provides first class support for modelling distributed state and events

Different QoS combination should be used to associate with the topic representing a state or an event the proper semantics

Soft State

In distributed systems you often need to model **soft-state -- a state that is periodically updated**

Examples are the reading of a sensor (e.g. Temperature Sensor), the position of a vehicle, etc.

The QoS combination to model **Soft-State** is the following:

```
Reliability       =>  BestEffort
Durability        =>  Volatile
History           =>  KeepLast(n) [with n = 1 in most of the cases]
Deadline          =>  updatePeriod
LatencyBudget     =>  updatePeriod/3 [rule of thumb]
DestinationOrder  =>  SourceTimestamp [if multiple writers per instance]
```

Hard State

In distributed systems you often need to model **hard-state** -- a **state** that is **sporadically updated** and that often has **temporal persistence requirements**

Examples are system configuration, a price estimate, etc.

The QoS combination to model Hard-State is the following:

```
Reliability          =>  Reliable
Durability           =>  Transient | Persistent
History              =>  KeepLast(n) [with n = 1 in most of the cases]
DestinationOrder     =>  SourceTimestamp [if multiple writers per instance]
WriterDataLifecycle =>  autodispose_unregistered_instances = false
```

Event

In distributed systems you often need to model **events** -- the **occurrence of something noteworthy for our system**

Examples are a collision alert, the temperature beyond a given threshold, etc.

The **QoS** combination to model Events is the following:

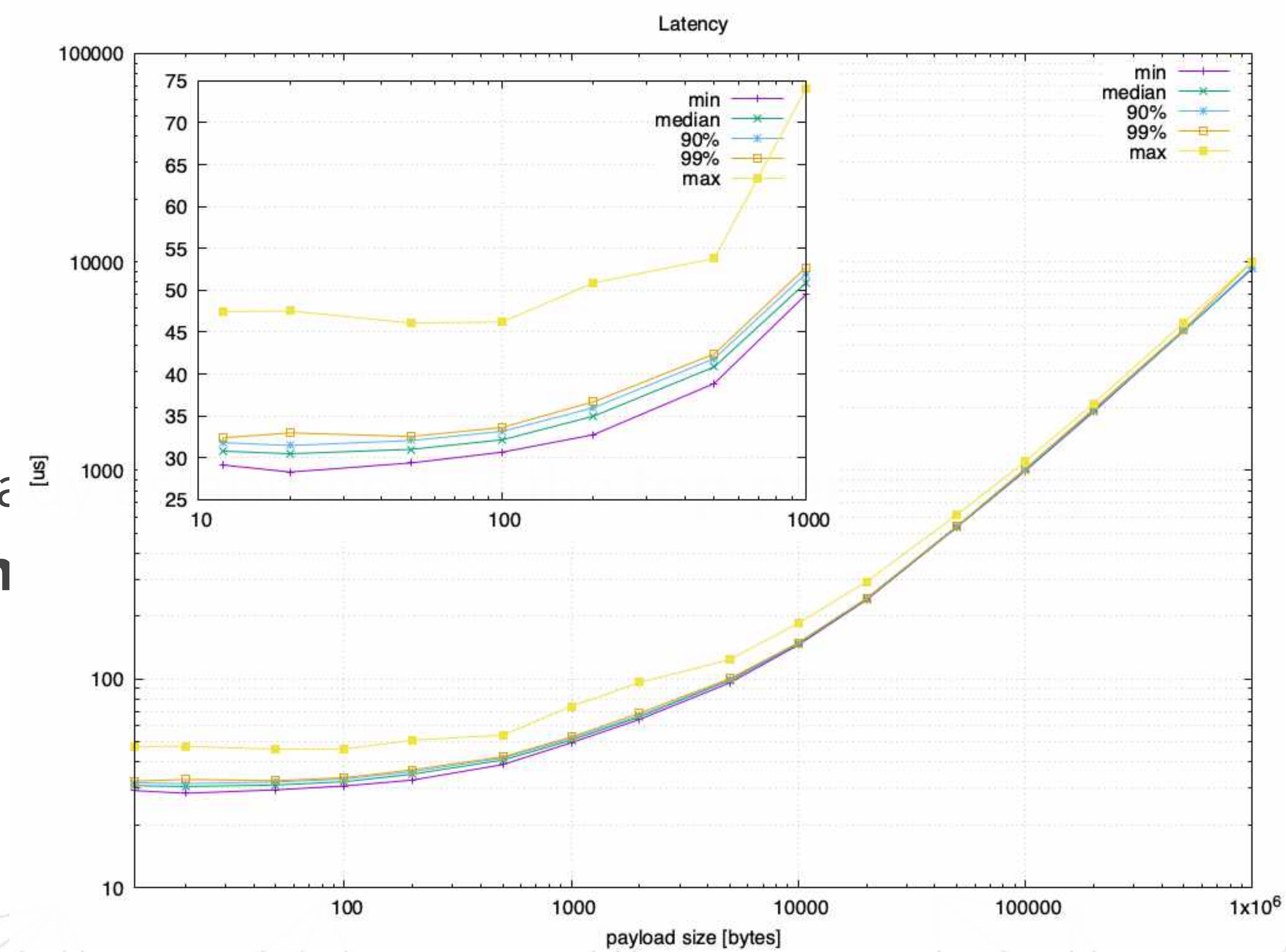
```
Reliability        =>  Reliable
Durability         =>  any         [depends on system requirements]
History            =>  KeepAll [on both DataWriter and DataReader!]
DestinationOrder =>  SourceTimestamp
WriterDataLifecycle => autodispose_unregistered_instances = false
ResourceLimits => [define appropriate bounds]
```

Performance

Cyclone Latency

Cyclone DDS has alrea

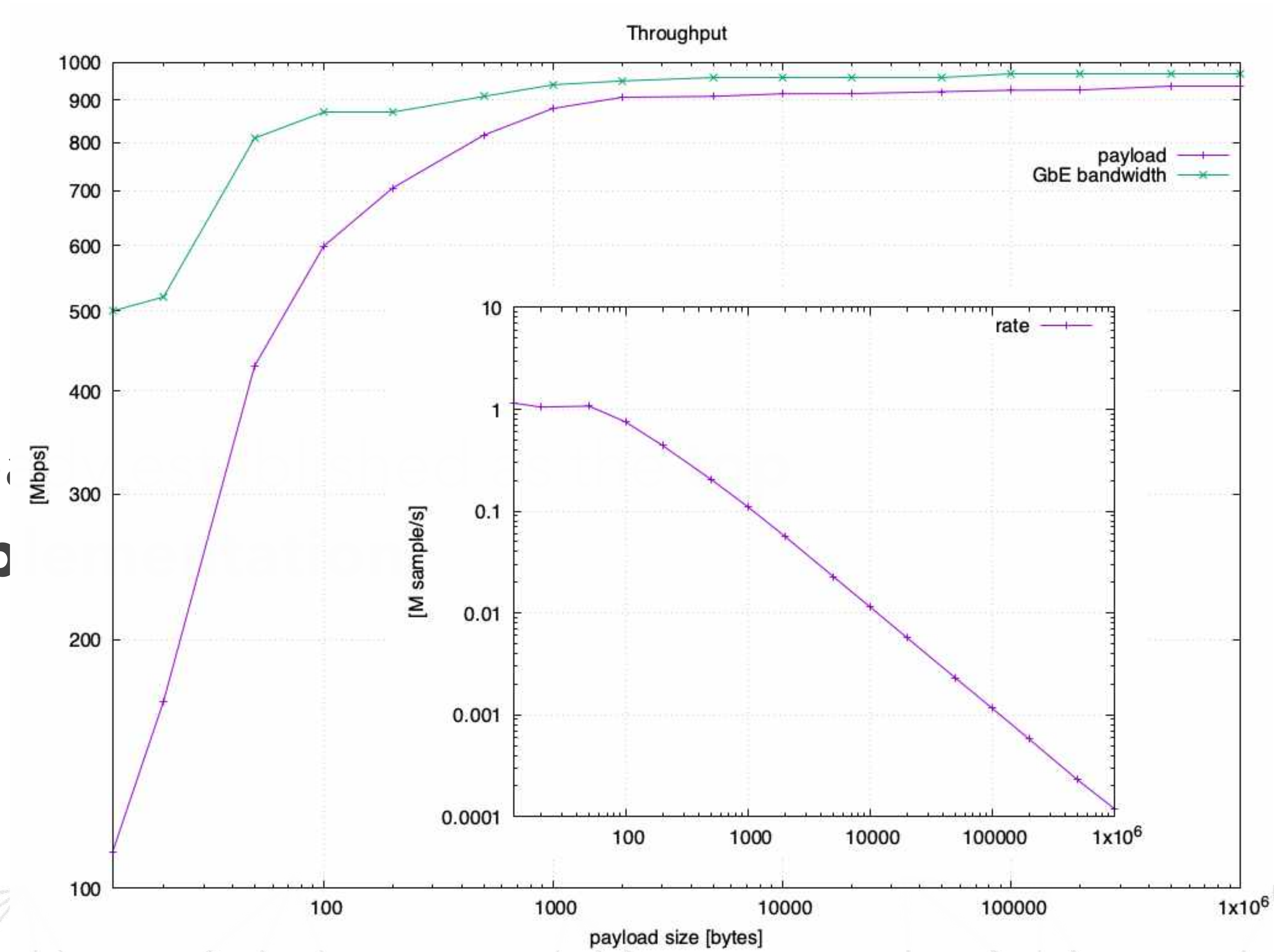
DDS implementation



Cyclone Throughput

Cyclone DDS has alre

performing DDS imp



Lab 3: Performance Evaluation

tspub.cpp

code > cpp > 01 > tspub.cpp > main(int, char * [])

```cpp
using namespace dds::core;
using namespace dds::core::policy;
using namespace dds::topic;
using namespace dds::pub;
using namespace dds::sub;

using namespace org::eclipse;

int
main(int argc, char* argv[])
{

  try {
    // Parse the command line args
    tspub_options opt = parse_tspub_args(argc, argv);

    // Create the domain participant
    dds::domain::DomainParticipant dp(cyclonedds::domain::default_id());

    // Initialize random number generation with a seed
    srandom(clock());

    dds::topic::qos::TopicQos tqos =
      dp.default_topic_qos() << LatencyBudget(Duration(2,0)) << Deadline(Duration(4,0));

    auto topic = Topic<tutorial::TempSensorType>(dp, "TempSensorTopic", tqos);
    auto pub = Publisher(dp);
    auto dw = DataWriter<tutorial::TempSensorType>(pub, topic, tqos);
```

PROBLEMS 8 | OUTPUT | DEBUG CONSOLE | TERMINAL | CMake/Build

```
[variant] Loaded new set of variants
[kit] Successfully loaded 1 kits from /Users/kydos/.local/share/CMakeTools/cmake-tools-kits.json
[main] Configuring folder: dds-tutorial
```

Summing Up

Concluding Remarks

DDS provides an extremely powerful set of abstractions and mechanism for data sharing in large scale distributed systems

DDS appears to be a natural fit for a large class of distributed and real-time applications, including robotics and autonomous driving