Operating Systems for Self-Driving Cars and Robots

- Chi-Sheng SHIH
- National Taiwan University

Agenda

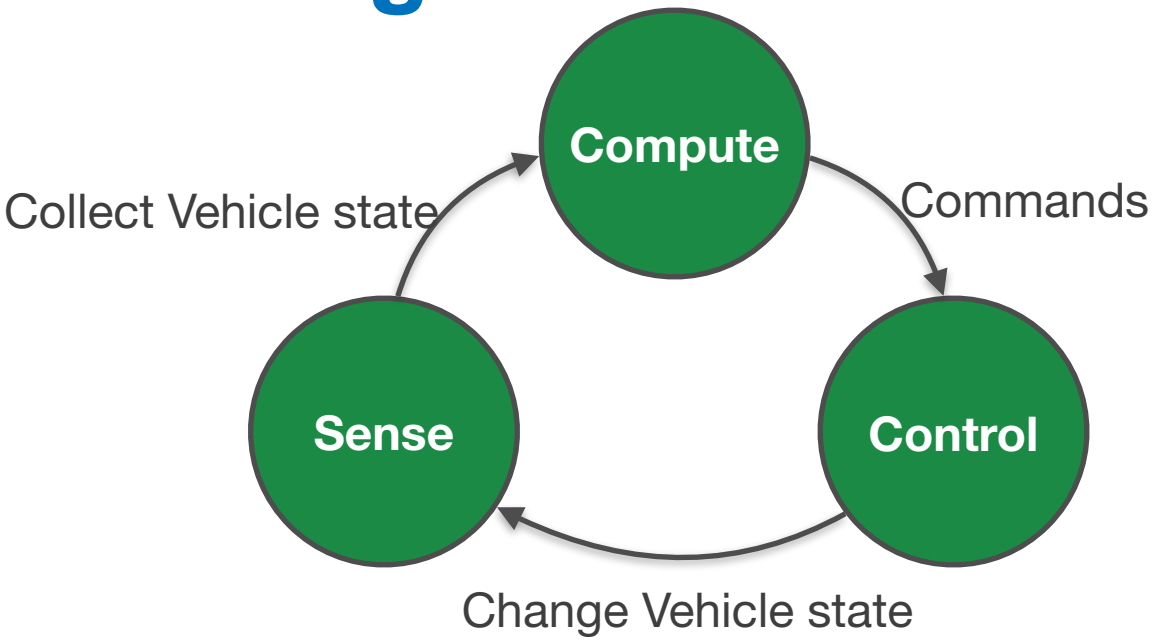
- Requirements for message exchange for self-driving car and robots
- Robot Operating Systems (ROS)
  - Architecture and Philosophy
  - Development Roadmap
- From ROS1 to ROS2
- (Optional) Data Distribution Service (DDS)

Requirements of Message Exchange for Vehicles and Robots

Message Exchange Intra-Vehicle

- One modern vehicle consists of 70 - 100 processing units, which need to exchange messages between them.
- The communications have diverse requirements:
  - Power-assisted mirrors/window/door/seat control: low data rate, can tolerate short loss and delay.
  - Park assistive radar: low data rate, can tolerate short loss
  - Throttle/steering/brake control: low data rate, cannot tolerate loss
  - Lidar/Camera data: high data rate, can tolerate short loss and delay.

Data Exchange in Real-Time Systems



Subsystems in self-driving cars require the ***real-time*** information from other subsystems.

- Late information will lead to wrong/out-of-date decision.
- ***Real-time*** is ***not*** equal to ***fast*** but refers to arrive/ complete the task, which is the transmission here, no later than a ***predictable*** time interval.

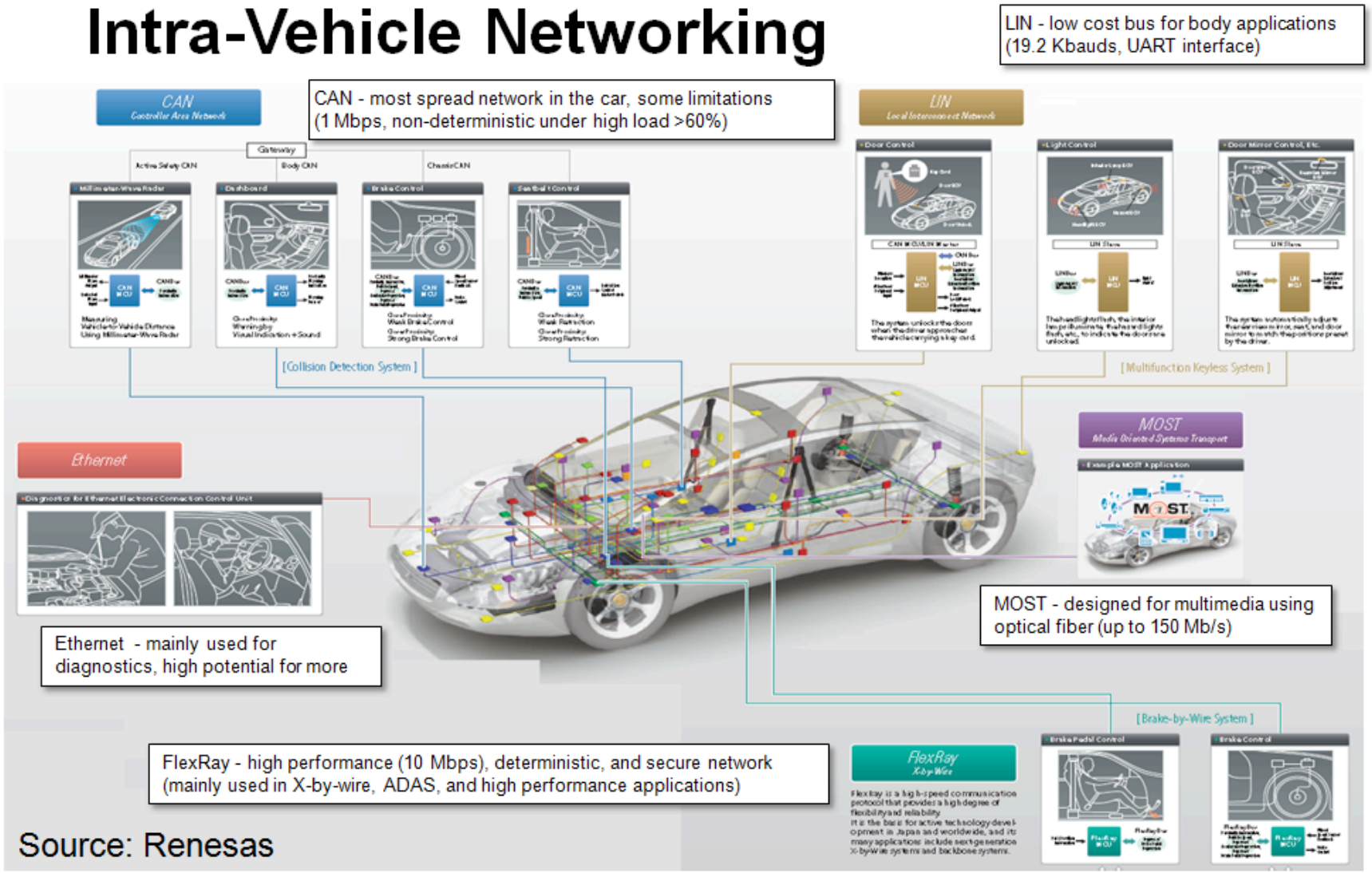
Requirements for Real-time Communication

- Predictable delay: the delay between message sending and deceiving is predictable or bounded by a pre-determined latency.
- Prioritized messaging: the messages are transmitted according to their priority levels and higher priority messages should not be blocked by low priorities messages.
- Scalability: message transmission delay are not affected by the number of computing nodes, including senders, receivers and others, in the systems.

Vehicle Bus

	Messaging	Node Control	Physical	Cost	Applications	Data rate
LIN (Local Interconnect Network)	Deterministic, static message schedule	Master/Slave	1-wire	Low	Displays, lighting, alarm systems, A/C, Seat & Mirror, power windows	10 Kb/s
CAN/TTCAN (Control Area Network)	Event-triggered messages/Time trigger messages	Autonomous	2-wire	Medium/High	Engine, transmission, braking, steering, suspension, assistance, safety, and diagnostics	1 Mbis/s under 40m/10Mb/s
FlexRay	Event/Time triggered messages	Autonomous, Master/Slave	2 Channel, 2-wire	High		10 Mb/s
Ethernet	Non-deterministic, dynamic message	Autonomous	CAT	Low	E-mirror, music, multimedia, camera, lidar data	100 Mb/s, 1Gb/s, 10Gb/s

Vehicle Bus

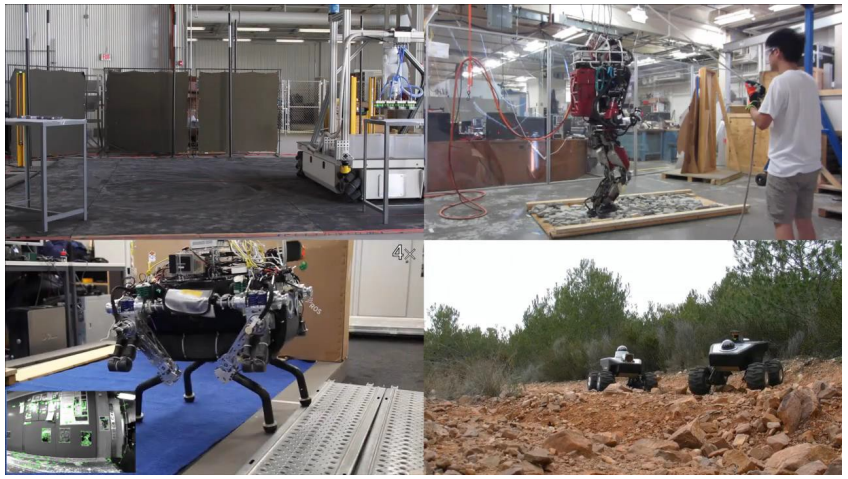

Intra-Vehicle Networking
LIN - low cost bus for body applications (19.2 Kbauds, UART interface)
CAN
Controller Area Network
CAN - most spread network in the car, some limitations (1 Mbps, non-deterministic under high load >60%)
LIN
Local Interconnect Network
Gateway
Active Safety CAN
Body CAN
Chassis CAN
[Collision Detection System]
[Multifunction Keyless System]
MOST
Media Oriented Systems Transport
Ethernet
MOST - designed for multimedia using optical fiber (up to 150 Mb/s)
Ethernet - mainly used for diagnostics, high potential for more
[Brake-by-Wire System]
FlexRay
X-by-Wire
FlexRay - high performance (10 Mbps), deterministic, and secure network (mainly used in X-by-wire, ADAS, and high performance applications)
Source: Renesas

Challenges for Adopting Real-Time Communication protocols

- Resource reservation:
  - Network transmissions are not preemptive.
  - Resources must be reserved in advance to assure that high priority messages have sufficient bandwidth to transmit.
- Offline Analysis:
  - The transmission workloads must be off-line analyzed to prevent undetermined blocking time.
  - Sporadic messages are difficult to guarantee their quality of services.
- Flexibility:
  - Not trivial to balance flexibility and guaranteed quality of services.
  - After-market sub-systems have limited business.

Robot Operating System ROS

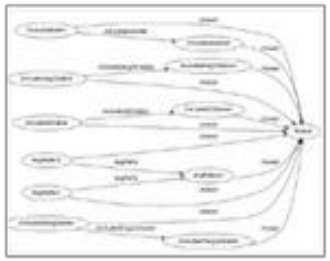
WHAT IS ROS?



- ROS is an open-source robot operating system
- A set of software libraries and tools that help you build robot applications that work across a wide variety of robotic platforms
- Originally developed in 2007 at the Stanford Artificial Intelligence Laboratory and development continued at Willow Garage
- Since 2013 managed by OSRF (Open Source Robotics Foundation)
- Today used by many robots, universities and companies
- De facto standard for robot programming

What is ROS?

ROS = Robot Operating System



Plumbing

- Process management
- Inter-process communication
- Device drivers

Tools

- Simulation
- Visualization
- Graphical user interface
- Data logging

Capabilities

- Control
- Planning
- Perception
- Mapping
- Manipulation

Ecosystem

- Package organization
- Software distribution
- Documentation
- Tutorials

http://www.ros.org

ROS Main Features

ROS has two "sides"

- The ***operating system*** side, which provides standard operating system services such as:
  - hardware abstraction
  - low-level device control
  - implementation of commonly used functionality
  - message-passing between processes
  - package management
- A ***suite of user contributed packages*** that implement common robot functionality such as SLAM, planning, perception, vision, manipulation, etc.

ROS Main Features

ROS

navigation

task executive

visualization

simulation

perception

control

planning

data logging

message passing

device drivers

real-time capabilities

OS

web browser

email client

window manager

memory management

process management

scheduler

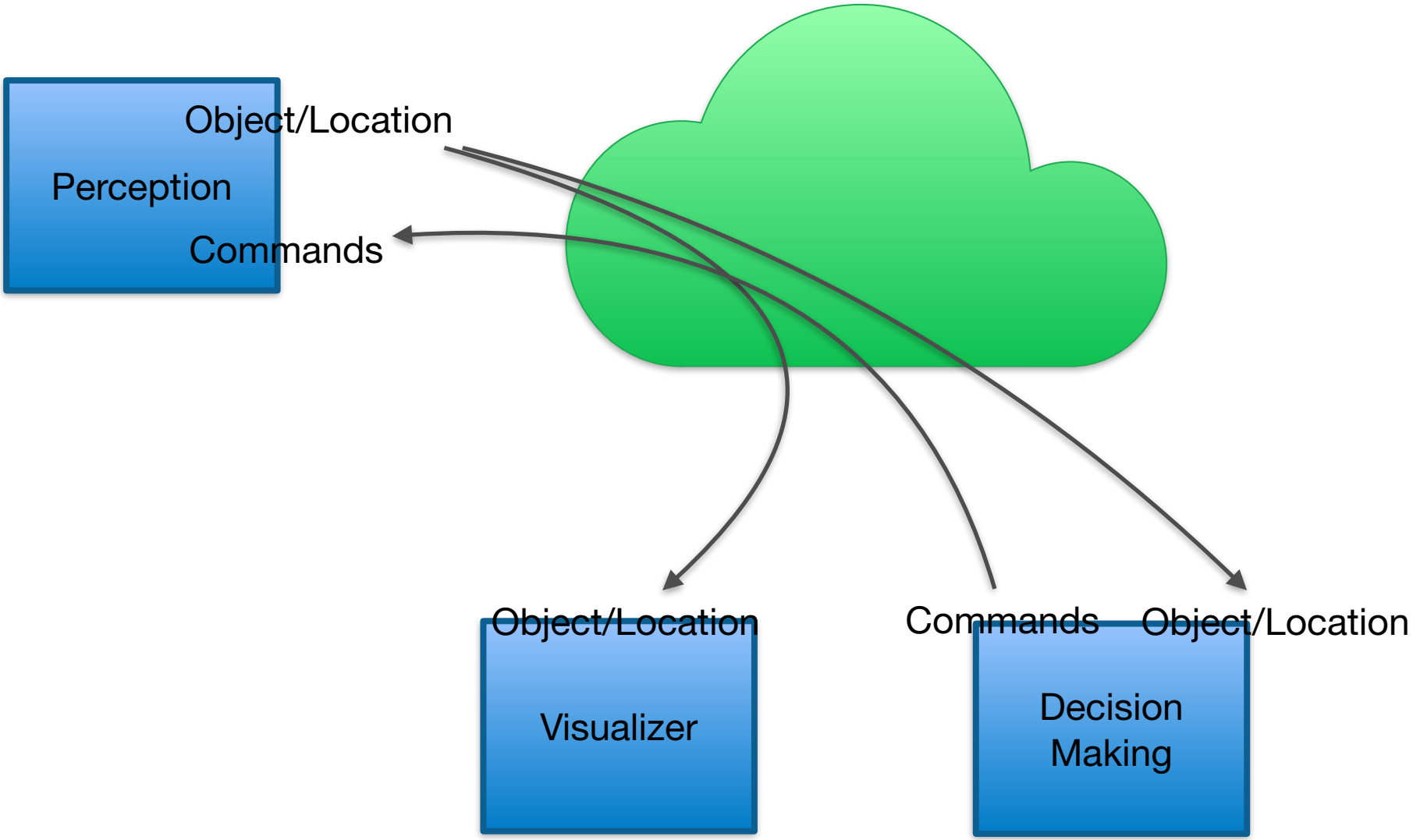
device drivers

file system

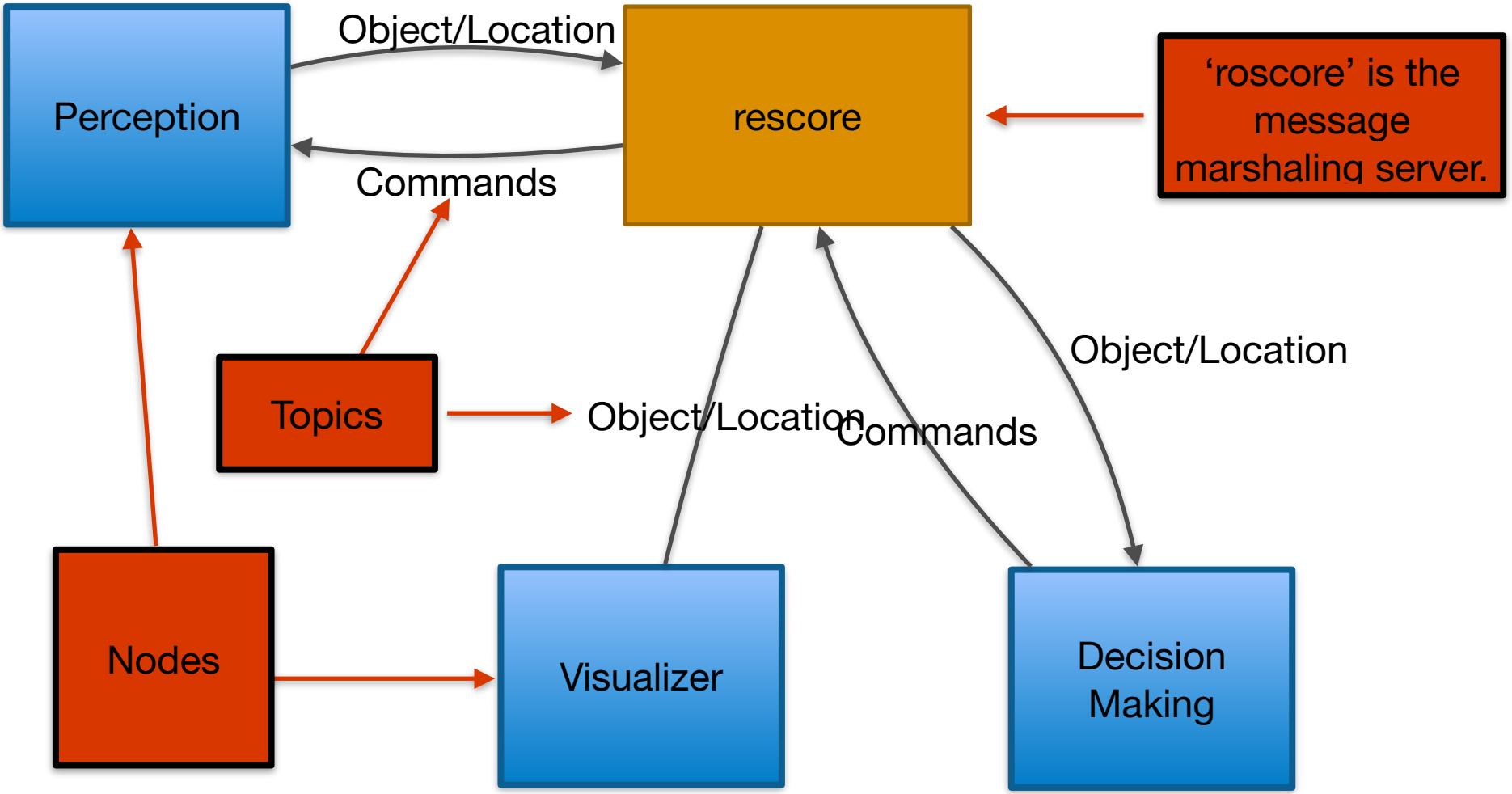
ROS Philosophy

- **Peer to peer**
  Individual programs communicate over defined API (ROS *messages*, *services*, etc.).
- **Distributed**
  Programs can be run on multiple computers and communicate over the network.
- **Multi-lingual**
  ROS modules can be written in any language for which a client library exists (C++, Python, MATLAB, Java, etc.).
- **Light-weight**
  Stand-alone libraries are wrapped around with a thin ROS layer.
- **Free and open-source**
  Most ROS software is open-source and free to use.

ROS Overview (ROS1)

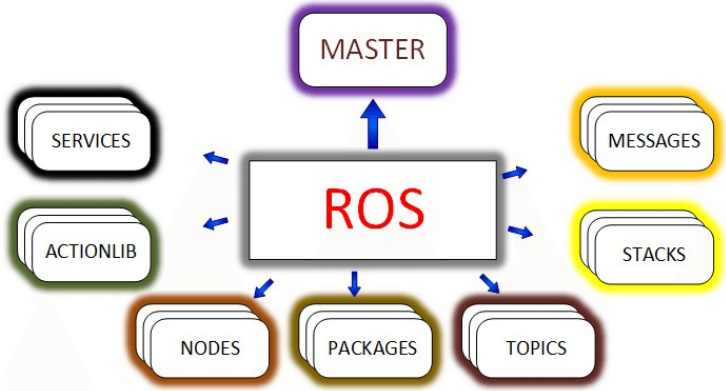


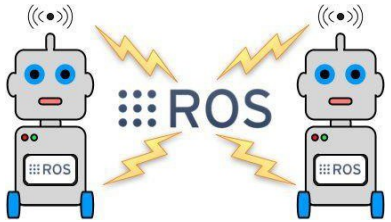
ROS Overview (ROS1)



ROS Core Concepts

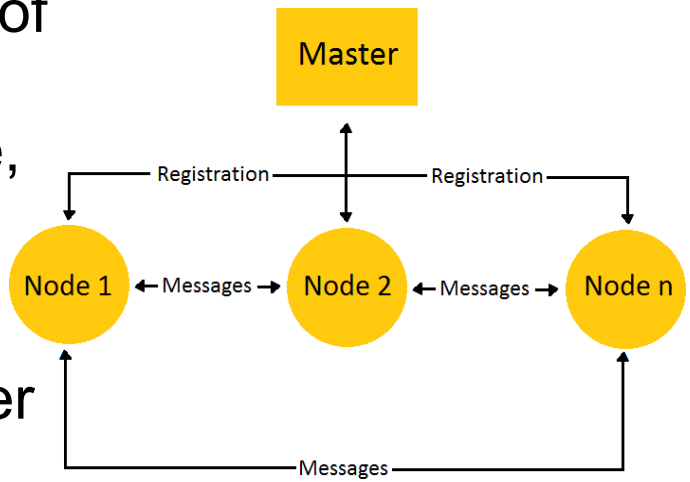
- Nodes
- Messages and Topics
- Services
- Actions
- ROS Master
- Parameters
- Packages and Stacks



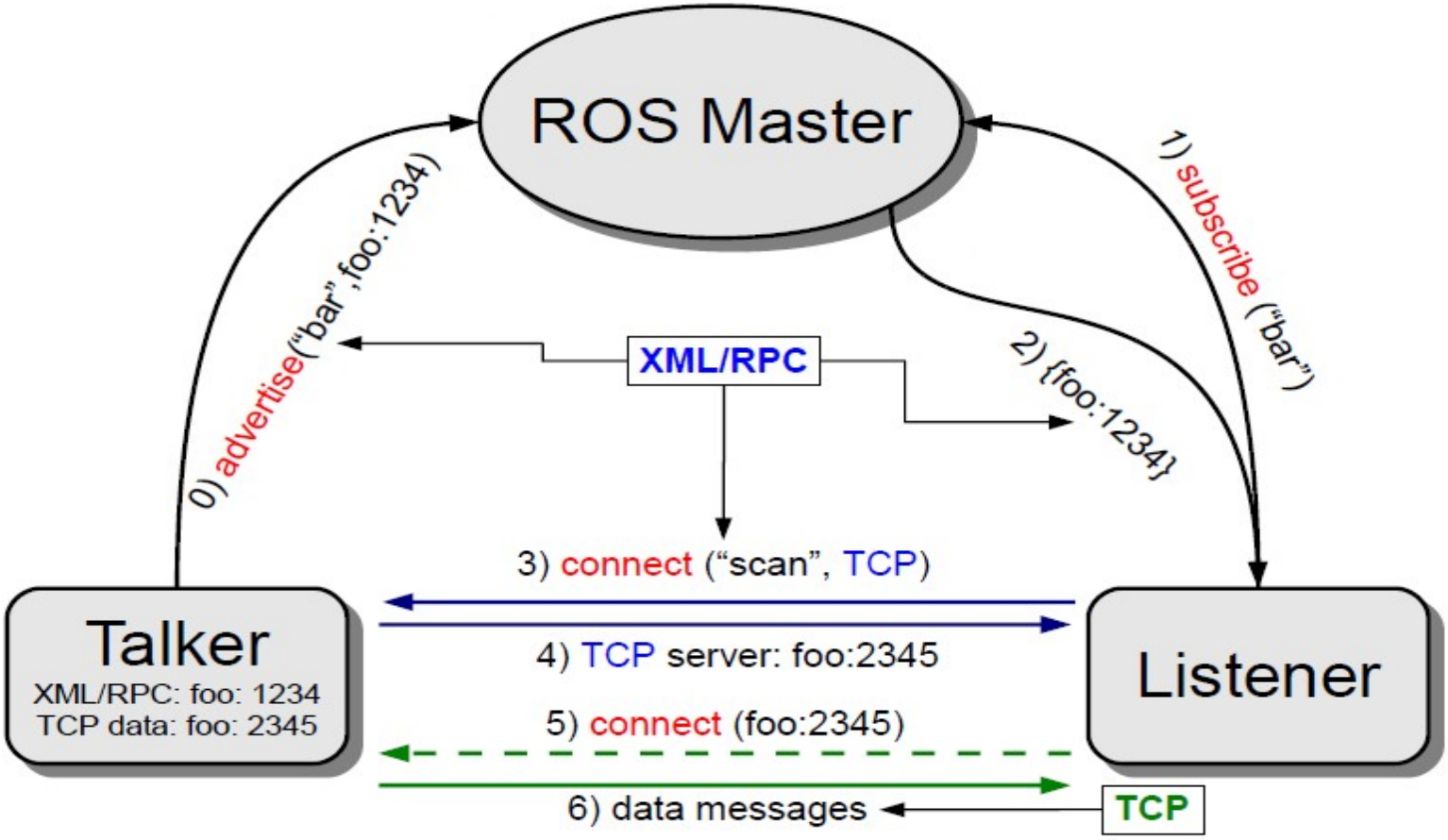


ROS MASTER

- Provides connection information to nodes so that they can transmit messages to each other
  - When activated, every node connects to a specified master to **register** details of the message streams they publish, services and actions that they provide, and streams, services, and action to which that they subscribe.
  - When a new node appears, the master provides it with the information that it needs to form a direct peer-to-peer TCP-based connection with other nodes publishing and subscribing to the same message topics and services.

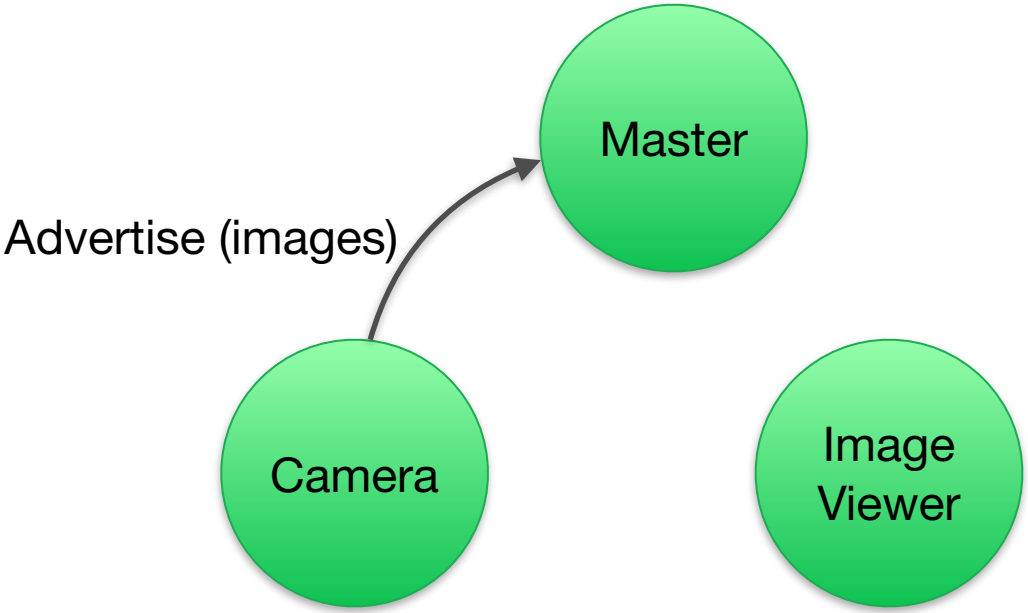


ROS Master


ROS Master
0) advertise("bar",foo:1234)
1) subscribe ("bar")
XML/RPC
2) {foo:1234}
3) connect ("scan", TCP)
4) TCP server: foo:2345
5) connect (foo:2345)
6) data messages
TCP
Talker
XML/RPC: foo: 1234
TCP data: foo: 2345
Listener

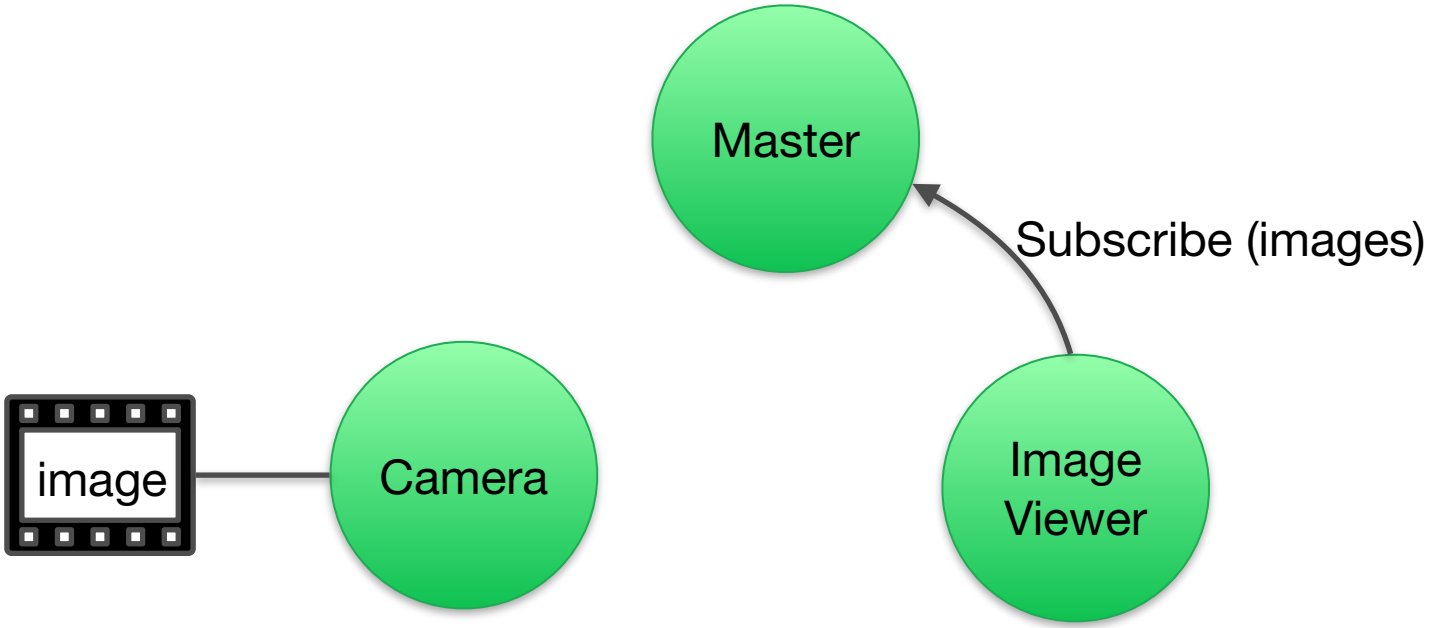
ROS Master

- We have two nodes: a *Camera* node and an *Image Viewer* node
- Typically the camera node would start first notifying the master that it wants to publish images on the topic "*images*":



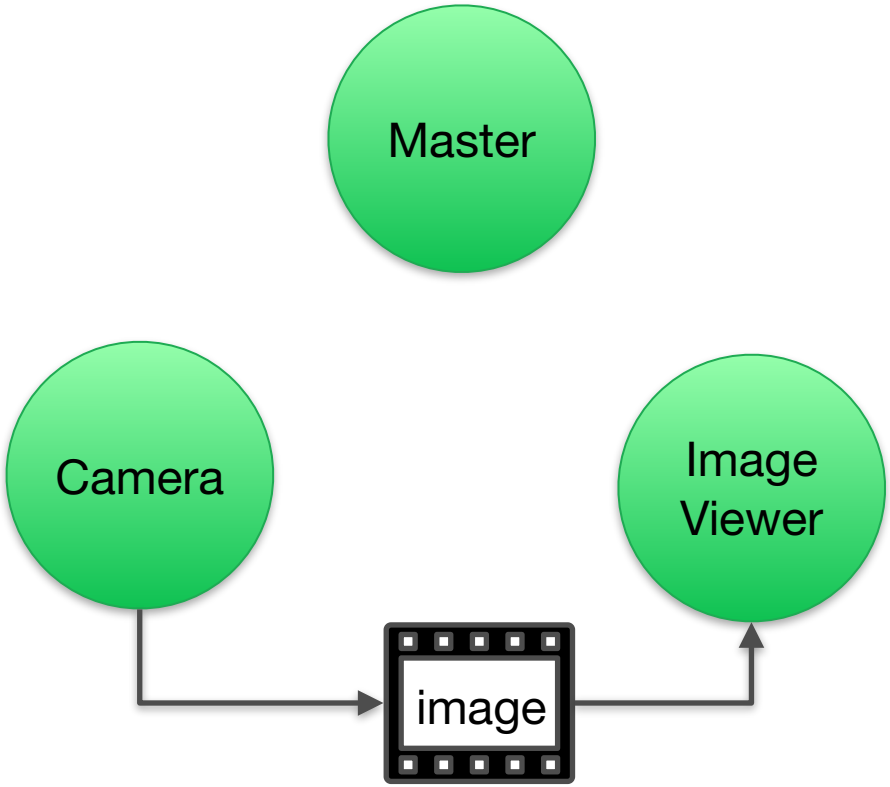
ROS Master

- *Image Viewer* wants to subscribe to the topic "*images*" to get and display images obtained with the camera:



ROS Master

- Now that the topic "*images*" has both a publisher and a subscriber, the master node notifies *Camera* and *Image Viewer* about each others existence, so that they can start transferring images to one another:



ROS Master

- Manages the communication between nodes (processes)
- Every node registers at startup with the master

ROS Master

Start a master

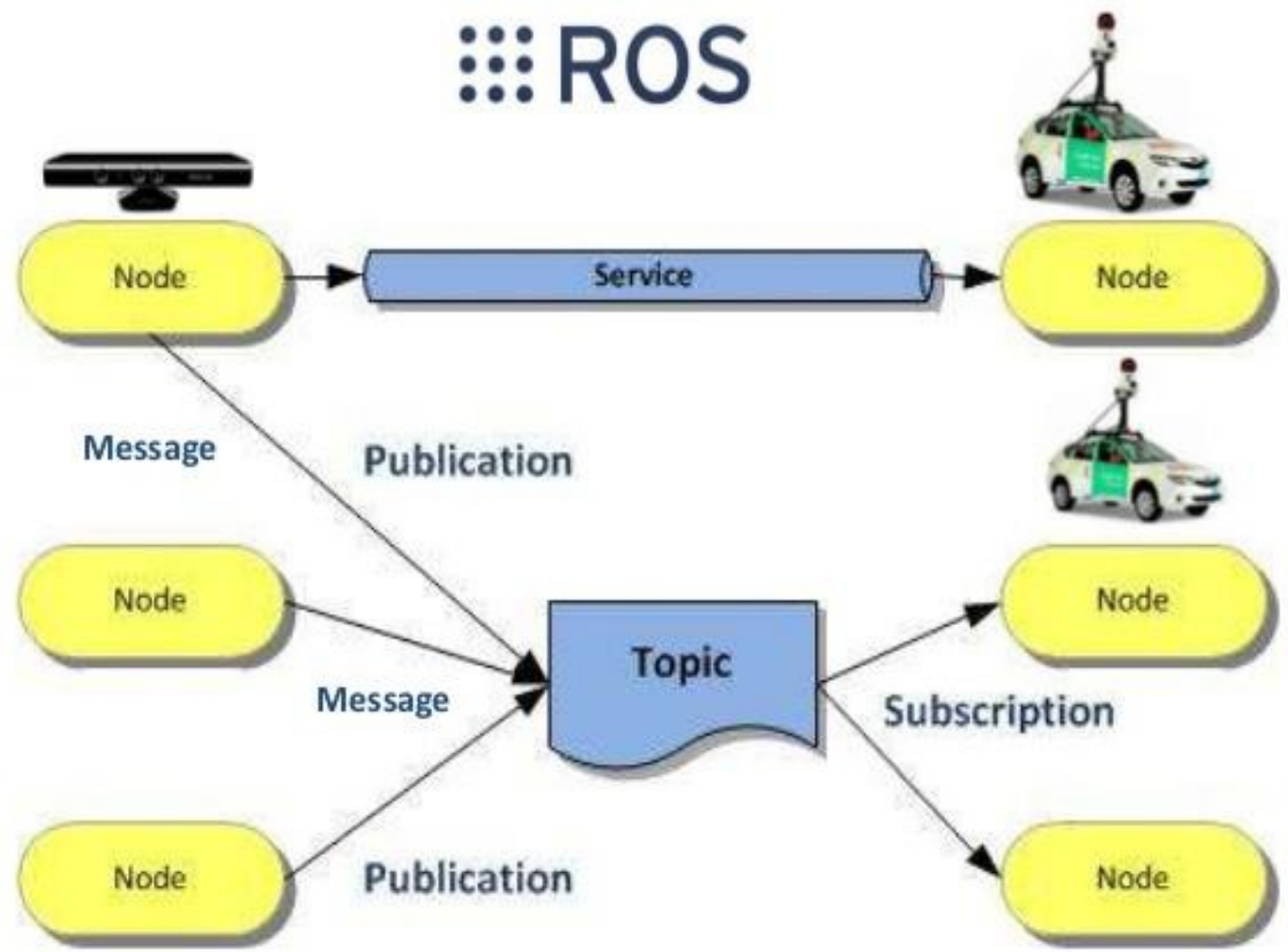
```
> roscore
```

More info
http://wiki.ros.org/Master

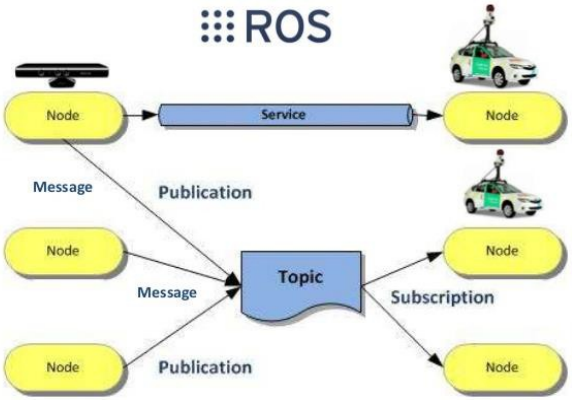
ROS Nodes

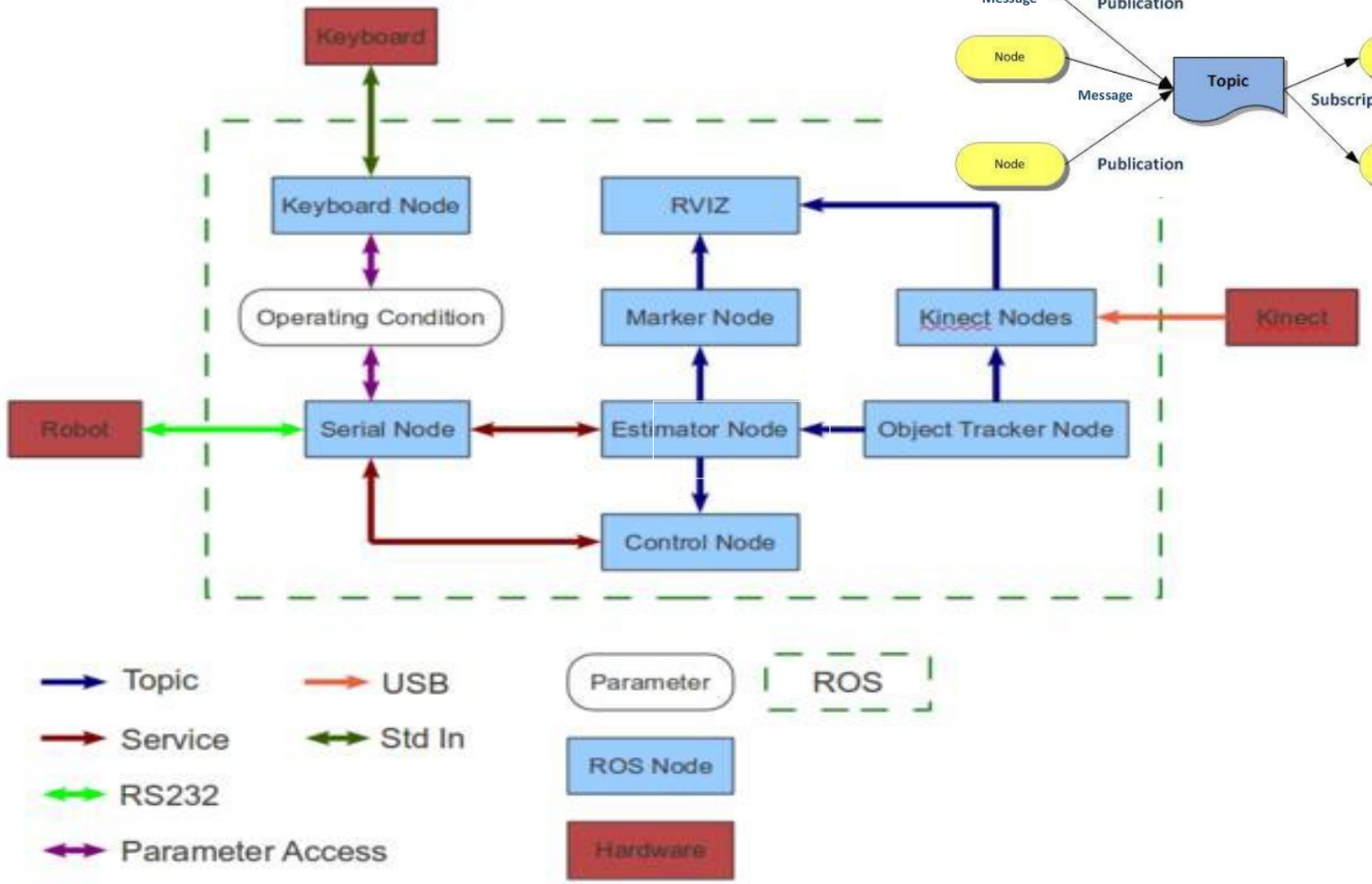
- Single-purposed executable programs
  - e.g. sensor driver(s), actuator driver(s), map building, planner, UI, etc.
- Individually compiled, executed, and managed
- Nodes are written using a ROS client library
  - roscpp – C++ client library
  - rospy – python client library
- Nodes can publish to or subscribe from a Topic
- Nodes can also provide or use a Service or an Action

ROS Nodes


ROS
Node
Service
Node
Message
Publication
Node
Topic
Node
Message
Subscription
Node
Publication
Node

ROS Nodes


ROS
Node
Service
Node
Message
Publication
Node
Topic
Node
Message
Subscription
Node
Publication
Node


Keyboard
Keyboard Node
RVIZ
Operating Condition
Marker Node
Kinect Nodes
Kinect
Robot
Serial Node
Estimator Node
Object Tracker Node
Control Node
Topic
USB
Parameter
ROS
Service
Std In
ROS Node
RS232
Parameter Access
Hardware

ROS Topics and ROS Messages

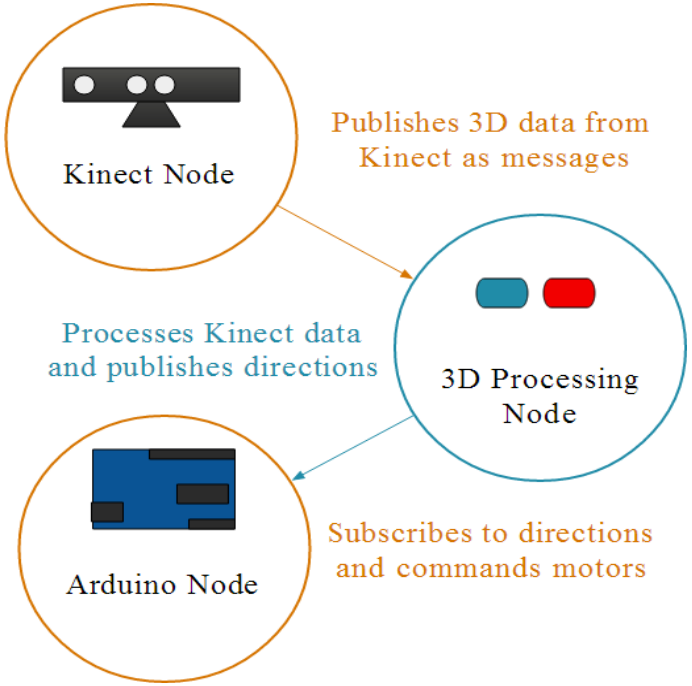
- Topic: named stream of messages with a defined type
  - Data from a range-finder might be sent on a topic called ***scan***, with a message of type *LaserScan*
- Nodes communicate with each other by publishing messages to topics
- Publish/Subscribe model: 1-to-N broadcasting

- Messages: Strictly-typed data structures for inter-node communication
  - *geometry_msgs/Twist* is used to express

velocity commands:

Vector3 linear
Vector3 angular

ROS Topics and ROS Messages

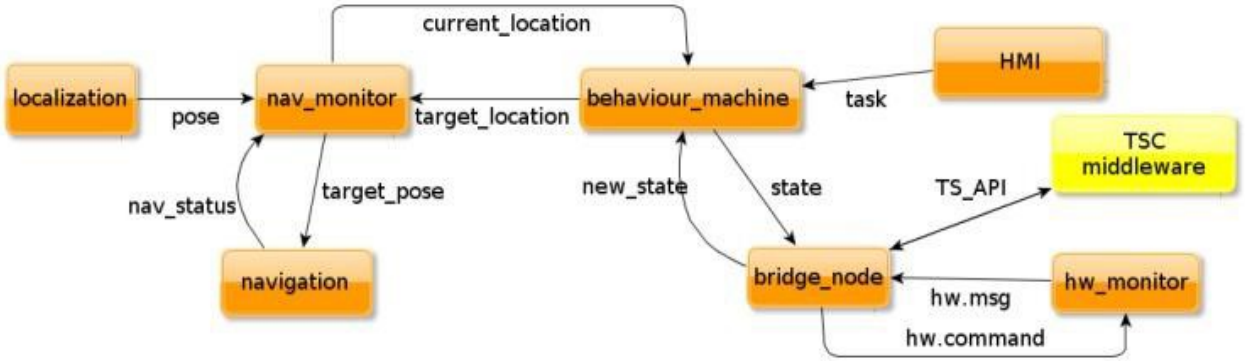


geometry_msgs/Twist

Vector3 linear
Vector3 angular

Vector3

float64 x
float64 y
float64 z



HMI: Human Machine Interface

ROS Master (Recap)

- Manages the communication between nodes (processes)
- Every node registers at startup with the master

ROS Master

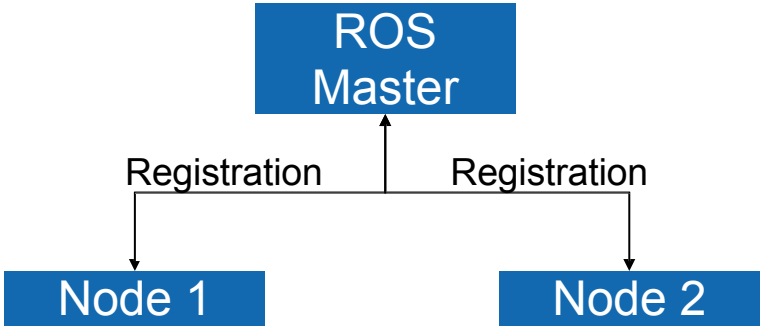
Start a master

```
> roscore
```

More info
http://wiki.ros.org/Master

ROS Nodes

- Single-purpose, executable program
- Individually compiled, executed, and managed
- Organized in *packages*



Run a node with

```
> rosrun package_name node_name
```

See active nodes with

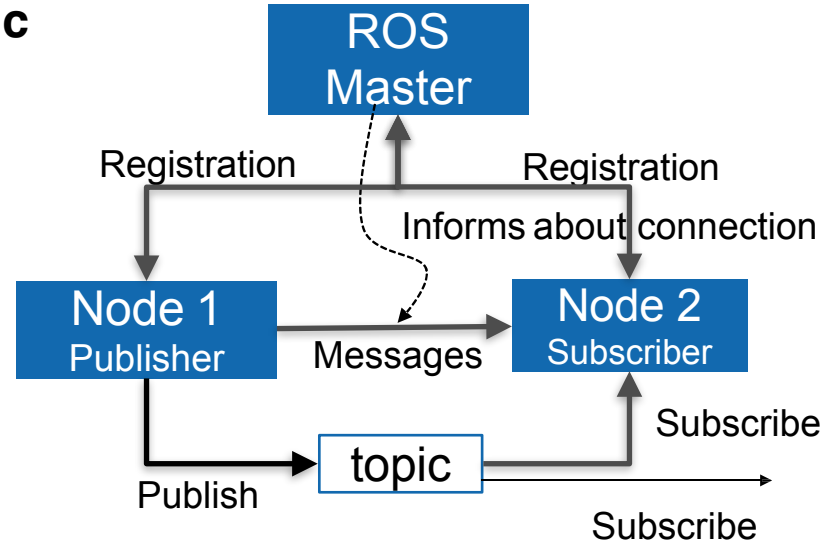
```
> rosnode list
```

Retrieve information about a node with

```
> rosnode info node_name
```

ROS Topics

- **Nodes communicate over *topics***
  - **Nodes can *publish* or *subscribe* to a topic**
  - **Typically, 1 publisher and *n* subscribers**
- **Topic is a name for a stream of *messages***



List active topics with

```
> rostopic list
```

Subscribe and print the contents of a topic with

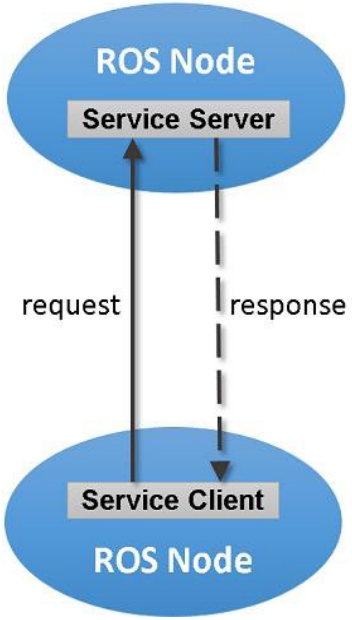
```
> rostopic echo /topic
```

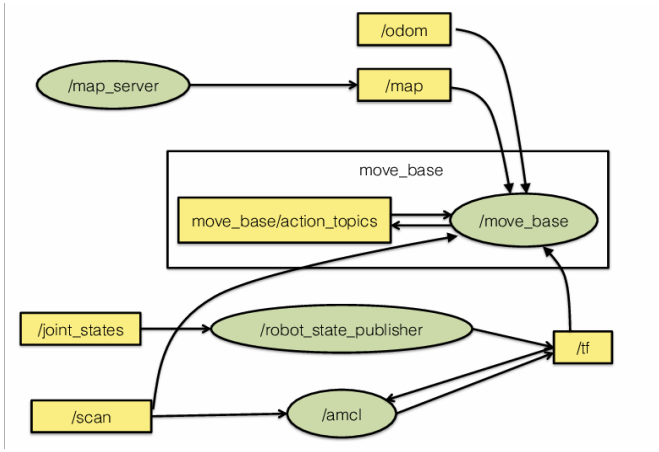
Show information about a topic with

```
> rostopic info /topic
```

ROS Services

- Synchronous inter-node transactions (blocking RPC): ask for something and wait for it
- **Service/Client model**: 1-to-1 request-response
- Service roles:
  - carry out remote computation
  - trigger functionality / behavior
  - *map_server/static_map* – retrieves the current grid map used for navigation



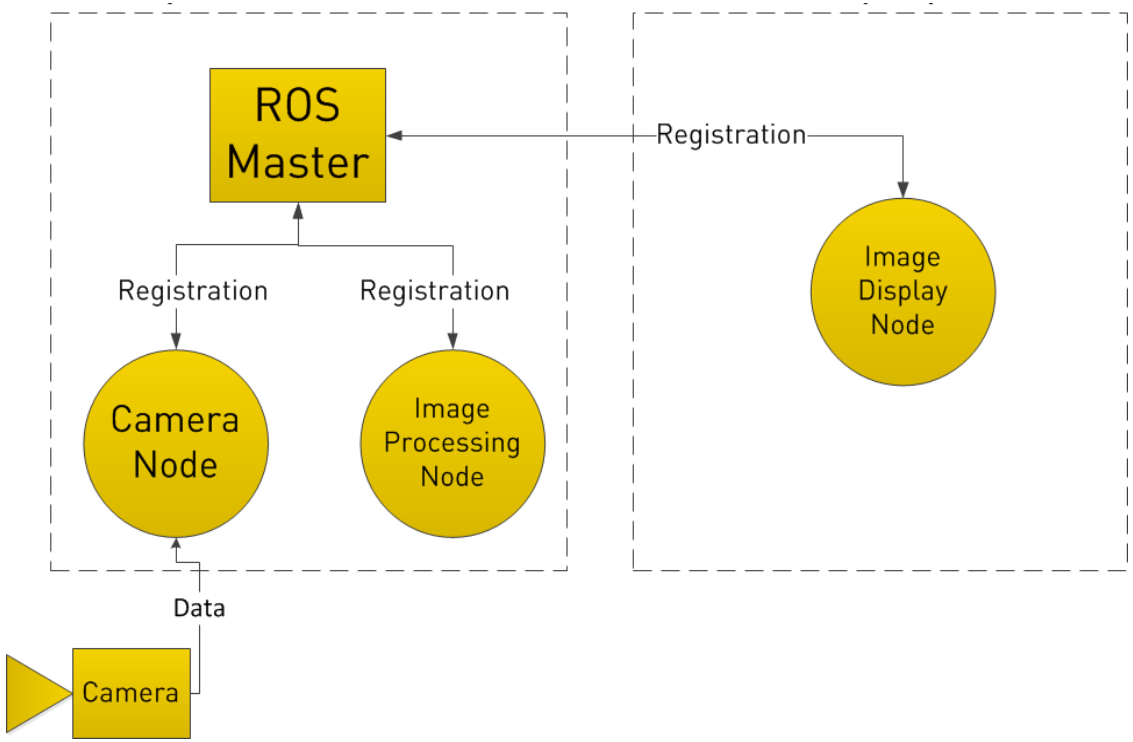


Service Name: **/example_service**
Service Type: **roscpp_tutorials/TwoInts**

Request Type: **roscpp_tutorials/TwoIntsRequest**
Response Type: **roscpp_tutorials/TwoIntsResponse**

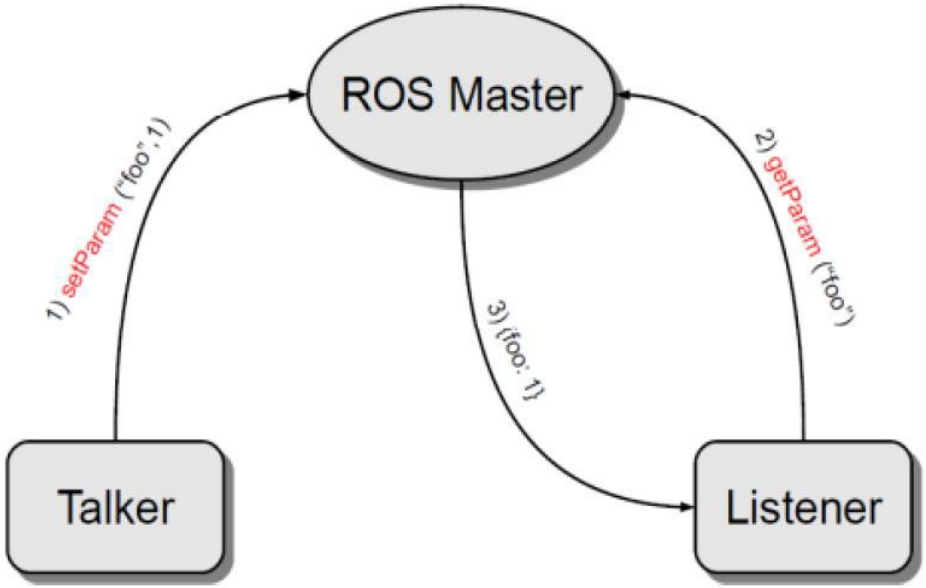
ROS Master

- The scenario can be made even more modular by adding an *Image processing* node, from which the *Image viewer* gets its data.



Parameter Server

- A *shared*, multi-variate dictionary that is accessible via network APIs
- Best used for static, non-binary data such as *configuration parameters*
- Runs inside the ROS master



ROS Bags

- Bags are the primary mechanism in ROS for **data logging**
- Bags subscribe to one or more ROS topics, and *store* the serialized message data in a file as it is received.
- Bag files can also be ***played back*** in ROS to the same topics they were recorded from, or even remapped to new topics.

ROS Supported Platforms

- ROS is currently supported only on **Ubuntu**
  - other variants such as Windows, Mac OS X, and Android are considered experimental
- ROS Kinetic Kame runs on **Ubuntu 16.04** (Xenial) and will support Ubuntu 15.10 (Willy)
- ROS Melodic Morenia runs on **Ubuntu 18.04** (Bionic) and will support Ubuntu 17.10 (Artful)

ROS Kinetic Kame

Released May, 2016

LTS, supported until April, 2021

ROS Melodic Morenia

Released May, 2018

Latest LTS, supported until May, 2023

ROS Environment

- ROS is fully integrated in the Linux environment: the ***rosbash*** package contains useful bash functions and adds tab-completion to a large number of ROS utilities

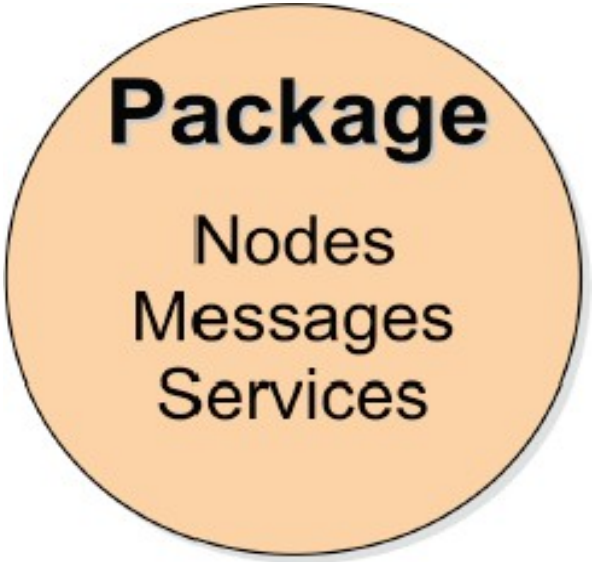
- After installing, ROS, setup.*sh files in '/opt/ros/<distro>/', need to be sourced to start *rosbash*:

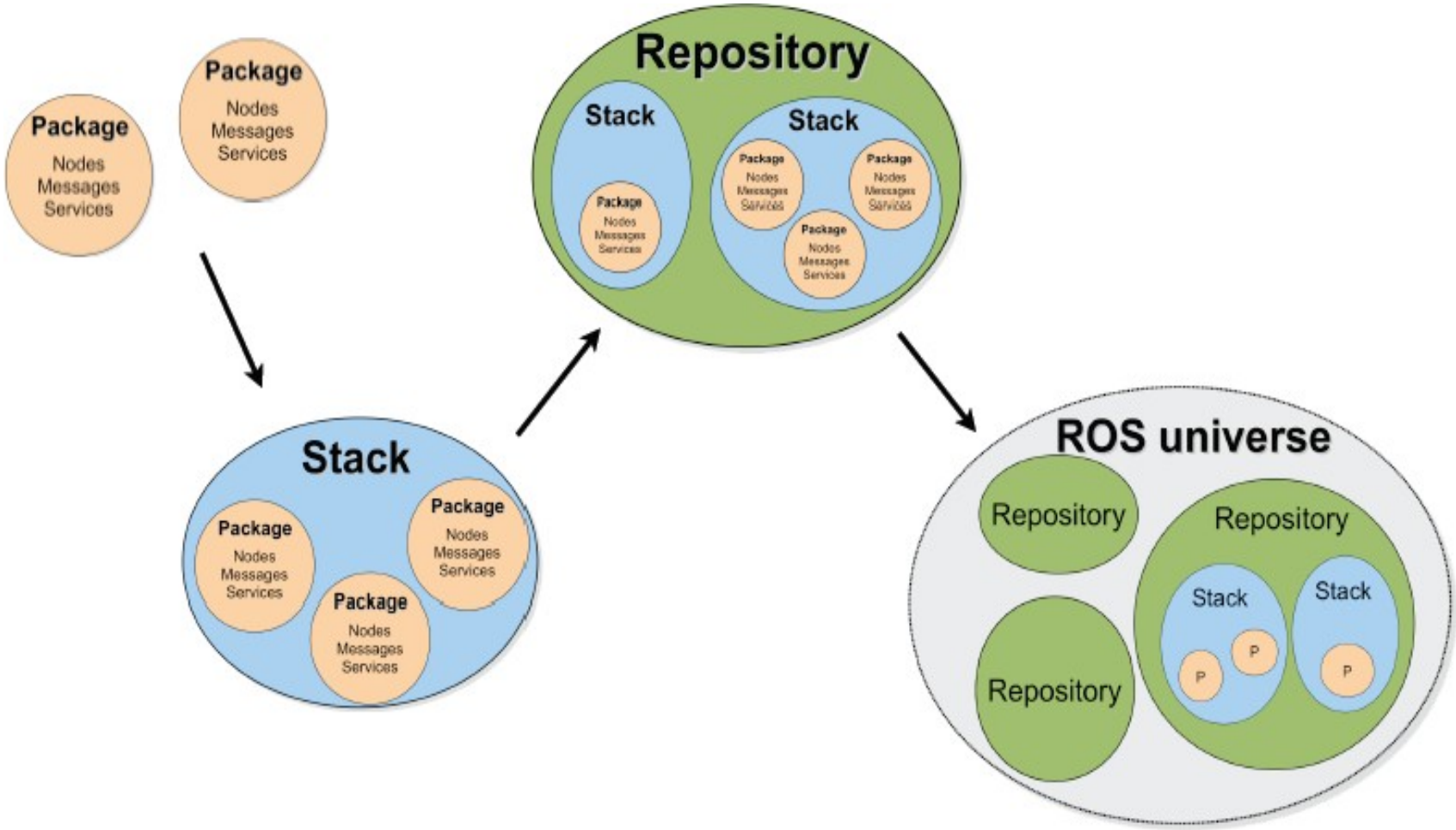
```
$ source /opt/ros/indigo/setup.bash
```

- This command needs to be run on every new shell to have access to the ros commands: an easy way to do it is to add the line to the bash startup file (~/.bashrc)

ROS Packages

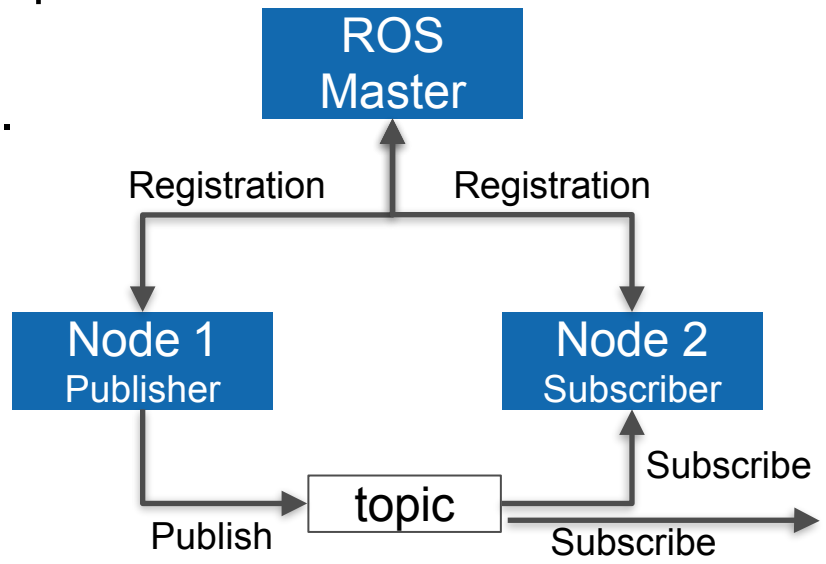
- Software in ROS is organized in ***packages***.
- A package contains one or more nodes, documentation, and provides a ROS interface
- Most of ROS packages are hosted in GitHub




Package
Nodes
Messages
Services
Package
Nodes
Messages
Services
Repository
Stack
Package
Nodes
Messages
Services
Stack
Package
Nodes
Messages
Services
Package
Nodes
Messages
Services
Package
Nodes
Messages
Services
Stack
Package
Nodes
Messages
Services
Package
Nodes
Messages
Services
Package
Nodes
Messages
Services
ROS universe
Repository
Repository
Stack
Stack
P
P
P
Repository

ROS Messages

- Data structure defining the *type* of a topic
- Comprised of a nested structure of integers, floats, booleans, strings etc. and arrays of objects
- Defined in **.msg* files



See the type of a topic

```
> rostopic type /topic
```

Publish a message to a topic

```
> rostopic pub /topic type data
```

**.msg* | Message definition

```
int number
double width string
description etc.
```

ROS Messages
Pose Stamped Example

geometry_msgs/Point.msg

```
float64 x
float64 y
float64 z
```

sensor_msgs/Image.msg

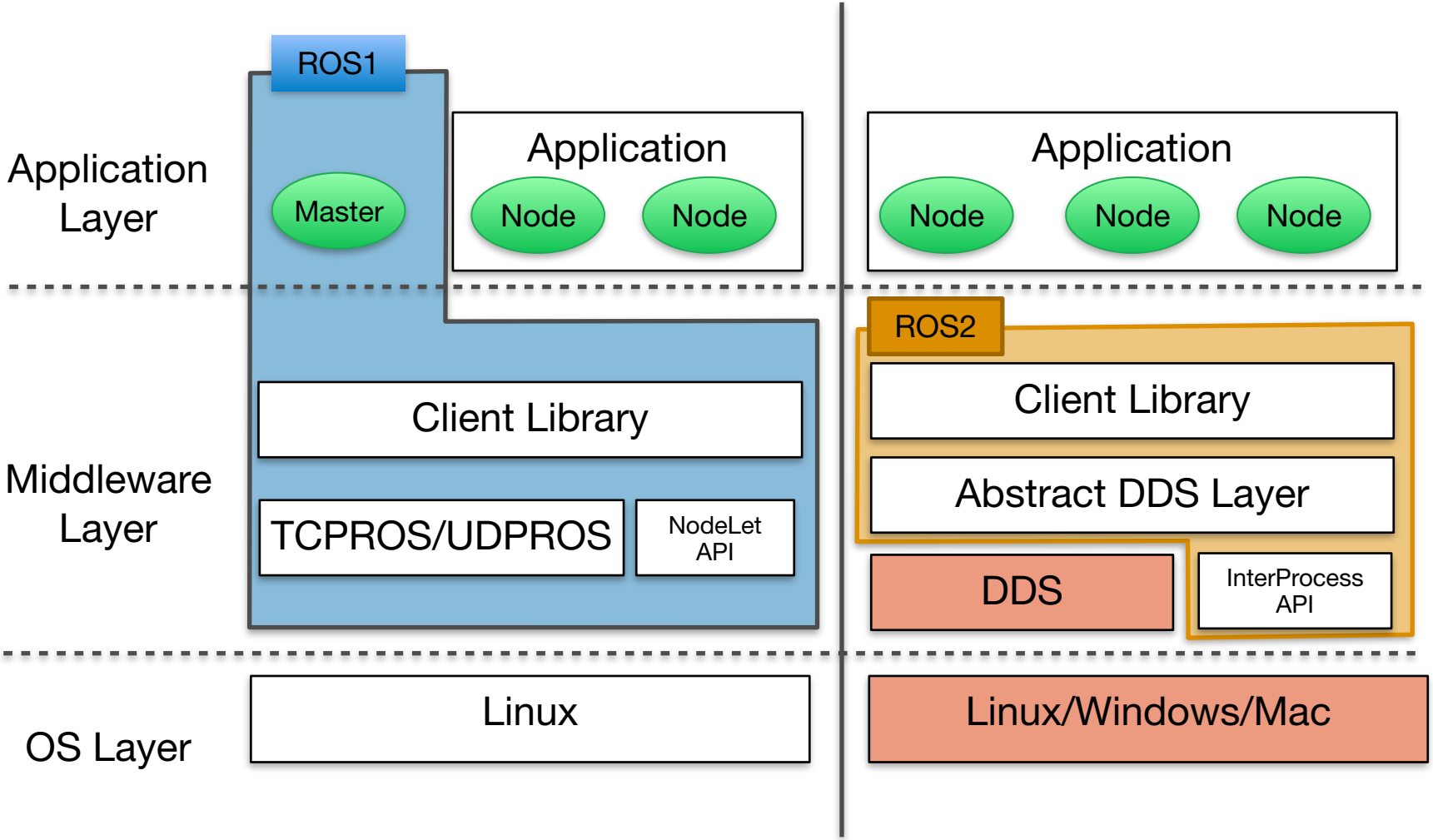
```
std_msgs/Header header
    uint32 seq
time stamp string
frame_id
uint32 height
uint32 width string
encoding uint8
is_bigendian uint32
step uint8[] data
```

geometry_msgs/PoseStamped.msg

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
geometry_msgs/Pose pose
    geometry_msgs/Point position
        float64 x
        float64 y
        float64 z
    geometry_msgs/Quaternion orientation
        float64 x
        float64 y
        float64 z
        float64 w
```

From ROS1 to ROS2

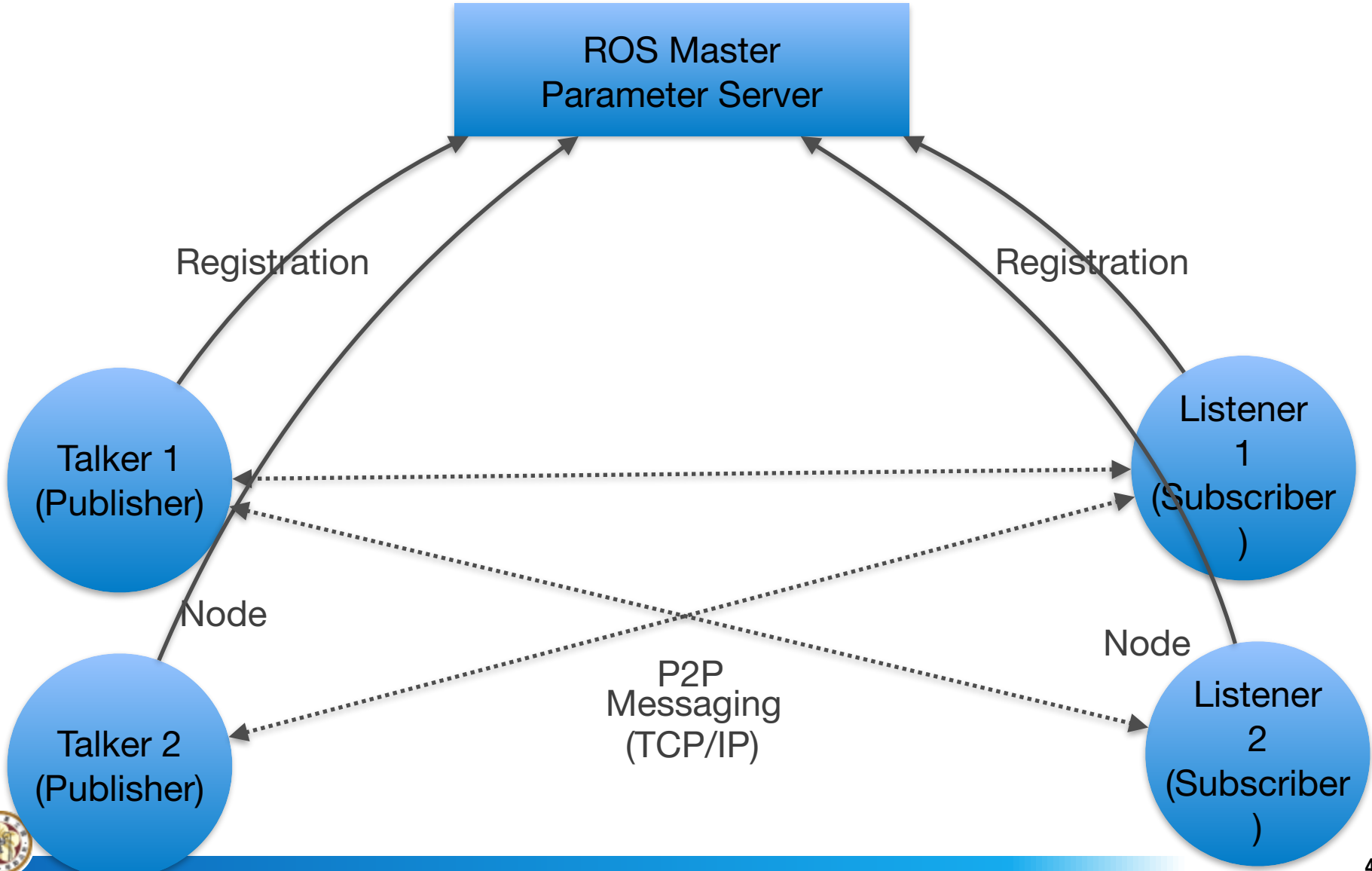
Comparing ROS 1 and ROS 2



Global Data Space

- Global data space is defined as a ***distributed discovery*** and storage services:
  - No single point failure
  - Publishers and subscribers are dynamically discovered.
- One can view it as a distributed storage system, which can be implemented as
  - distributed cache,
  - distributed buffer,
  - distributed file system, or
  - distributed database system

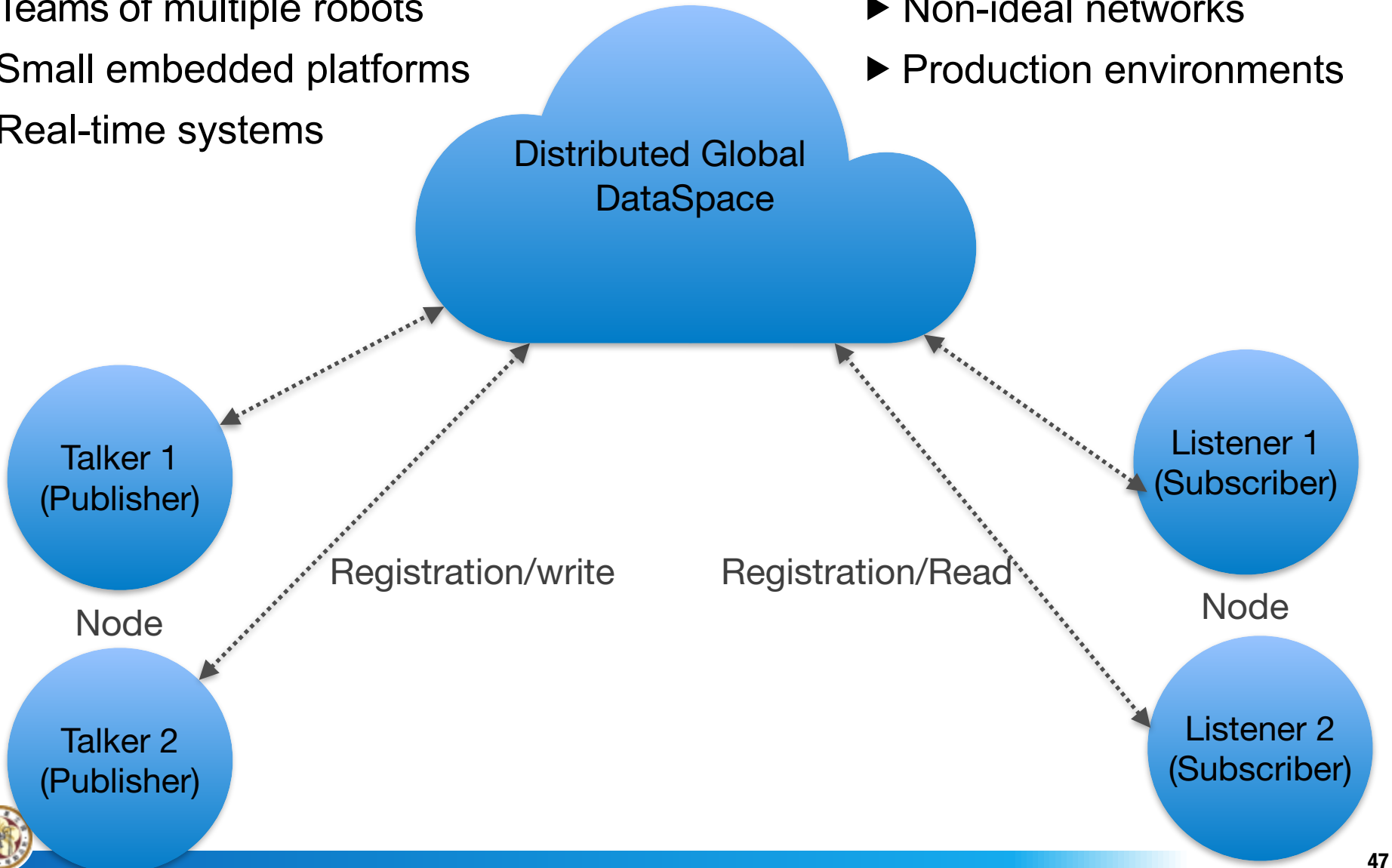
ROS 1 Message using topics



ROS 2 Message using topics

Use-case / basic requirements

- Teams of multiple robots
- Small embedded platforms
- Real-time systems
- Non-ideal networks
- Production environments

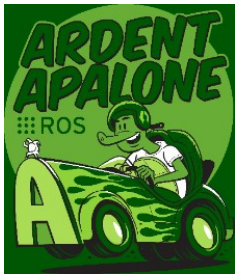


Current Status of ROS 2

History

- Development started in 2014
- First stable release in December 2017
- First LTS release planned in May 2019

LTS: Long Term Support, which is 5 years.



December '17
December '18

July '18

May '19

Nov '19

ROS 2 Supported Platforms

- ROS 2 Eloquent Elusor
  - Eloquent Elusor is primarily supported on the following platforms:
  - Tier 1 platforms:
    - Ubuntu 18.04 (Bionic): amd64 and arm64
    - Mac macOS 10.14 (Mojave)
    - Windows 10 (Visual Studio 2019)
  - Tier 2 platforms:
    - Ubuntu 18.04 (Bionic): arm32
  - Tier 3 platforms:
    - Debian Stretch (9): amd64, arm64 and arm32
    - OpenEmbedded Thud (2.6) / webOS OSE: arm32 and x86

Tier 1 platforms are subjected to our unit test suite and other testing tools on a frequent basis including continuous integration jobs, nightly jobs, packaging jobs, and performance testing. Errors or bugs discovered in these platforms are prioritized for correction by the development team.

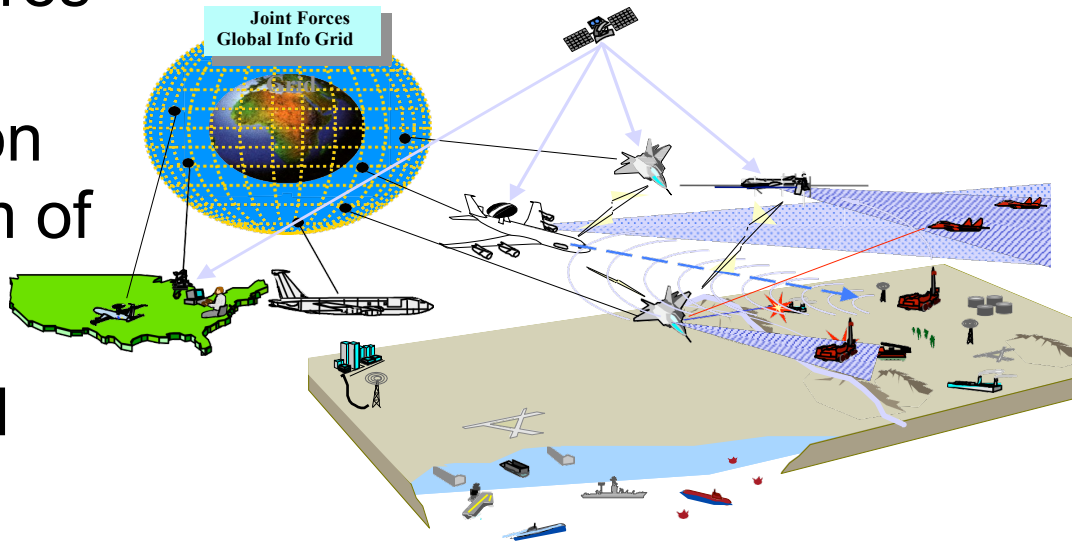
Tier 2 platforms are subject to periodic CI testing which runs both builds and tests with publicly accessible results. The CI is expected to be run at least within a week of relevant changes for the current state of the ROS distribution.

Tier 3 platforms are those for which community reports indicate that the release is functional. The development team does not run the unit test suite or perform any other tests on platforms in Tier 3.

Data Distribution Service (DDS)

The Need for Data Distribution

- Data Centric Architectures are emerging as a key trend for next generation military and civil system of systems
- **Efficient**, **scalable** and **QoS- enabled** data dissemination is an enabling technology for Network Centric Systems

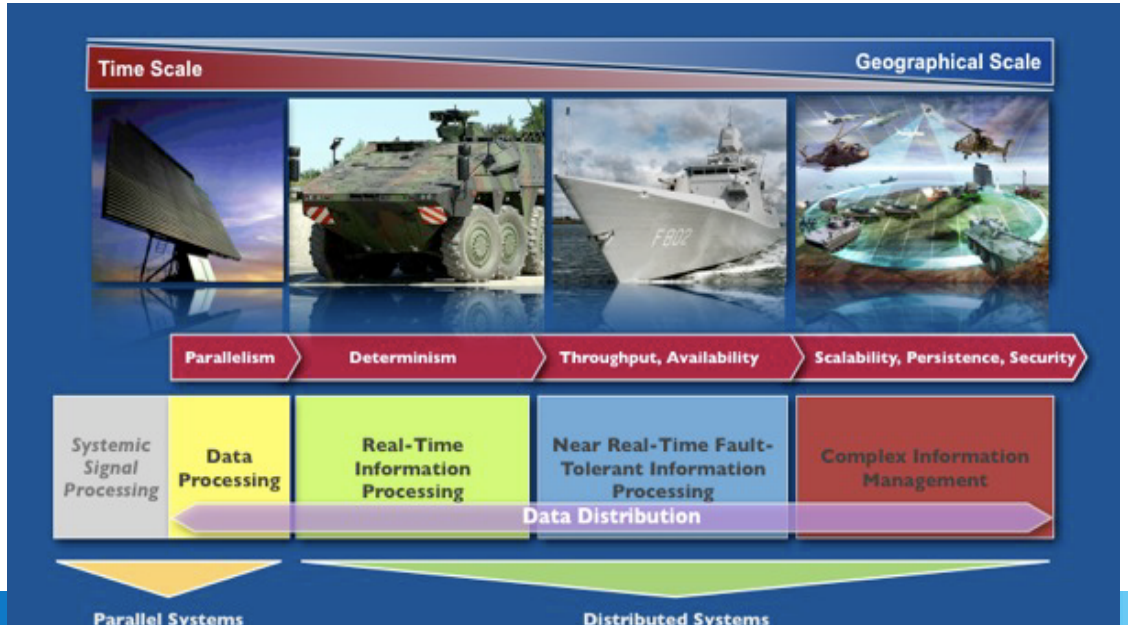


Adapted from "The Future of AWACS", by LtCol Joe Chapa

*The Right **Information** => To the Right **People** => At the Right **Time***

The OMG DDS Standard

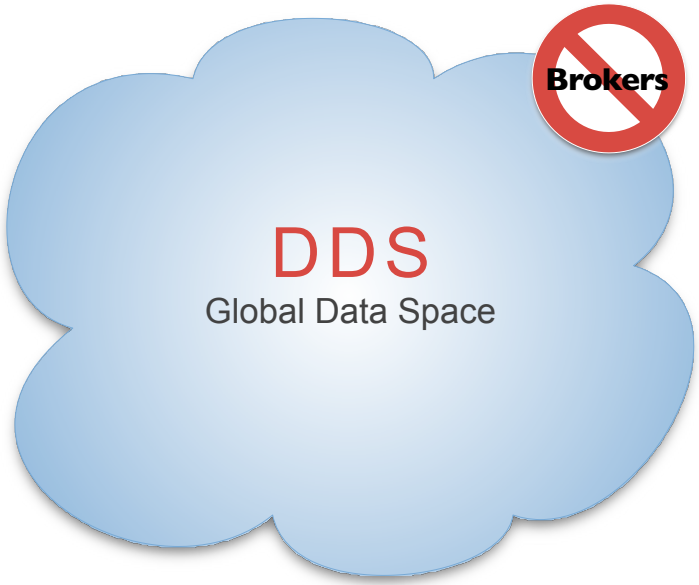
- Introduced in 2004 to address the Data distribution challenges typical of Defense and Aerospace Applications
  - Key requirement for the standard were high performance and scalability from embedded to ultra-large-scale deployments
  - Today recommended by key administration worldwide and widely adopted well beyond Aerospace and Defense in domains, such as, Automated Trading, Simulations, SCADA, Telemetry, etc.



How Does it Work?

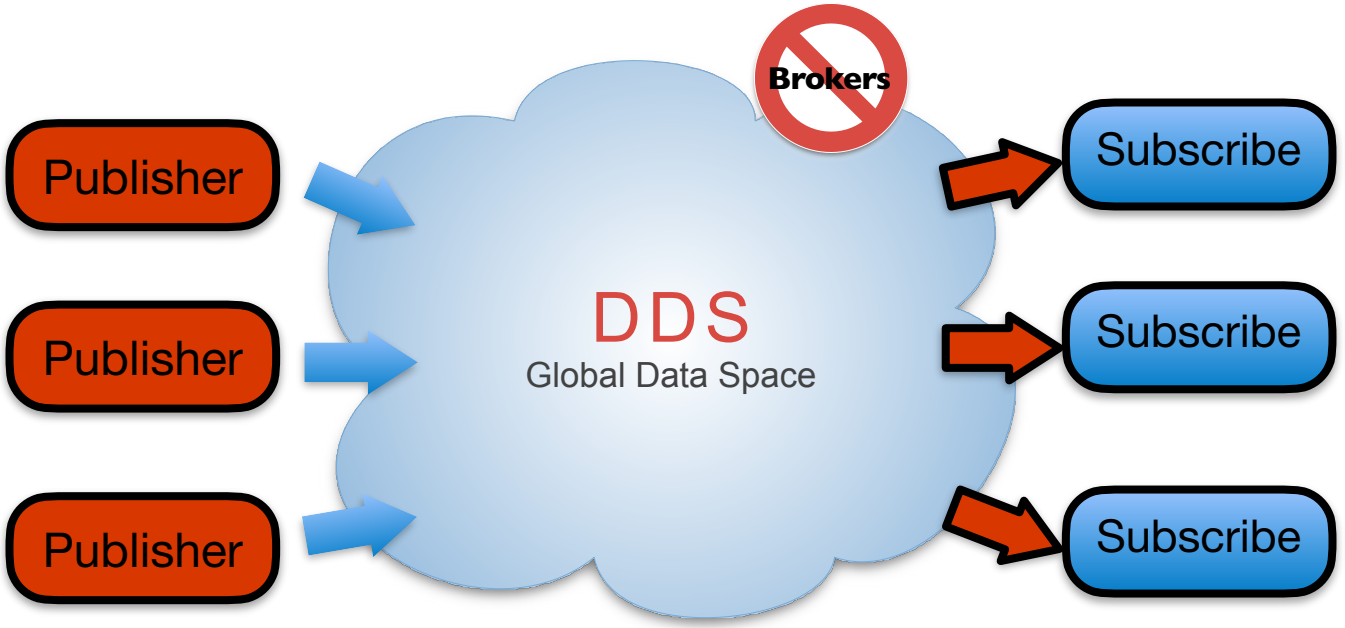
How Does it Work?

- DDS is based around the concept of a fully distributed Global Data Space (GDS)
- Applications can autonomously and asynchronously read/written data in the GDS



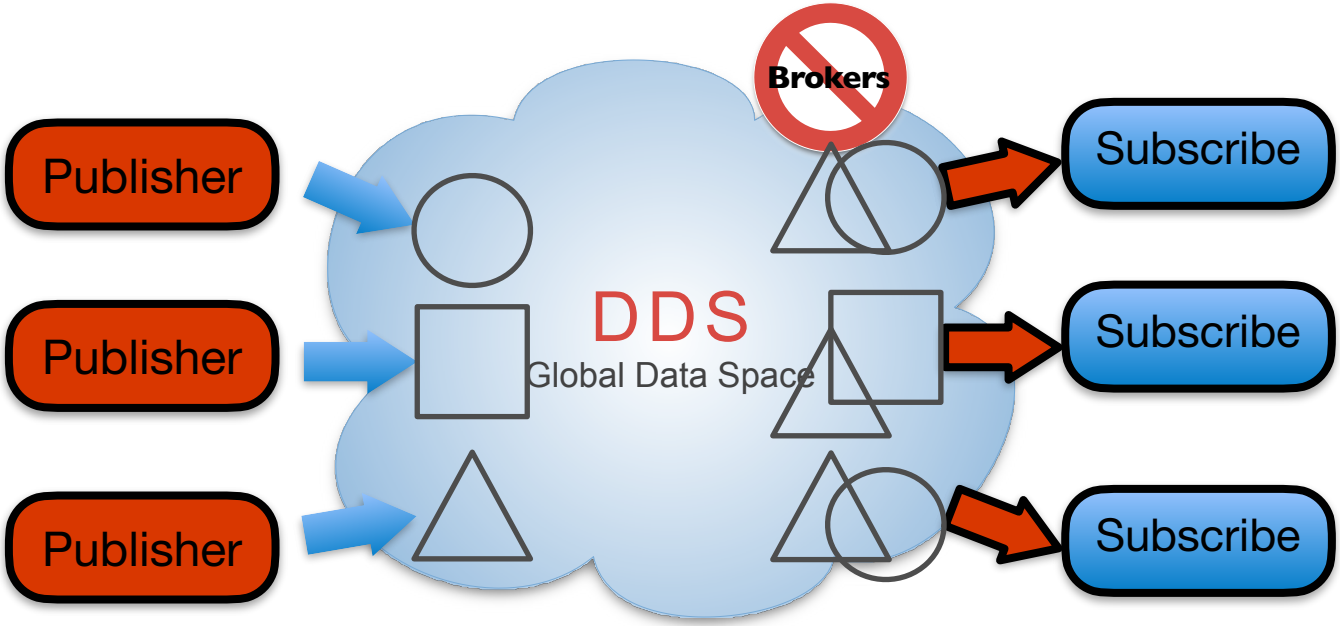
How Does it Work?

- Publishers and Subscribers can join and leave the GDS at any time.



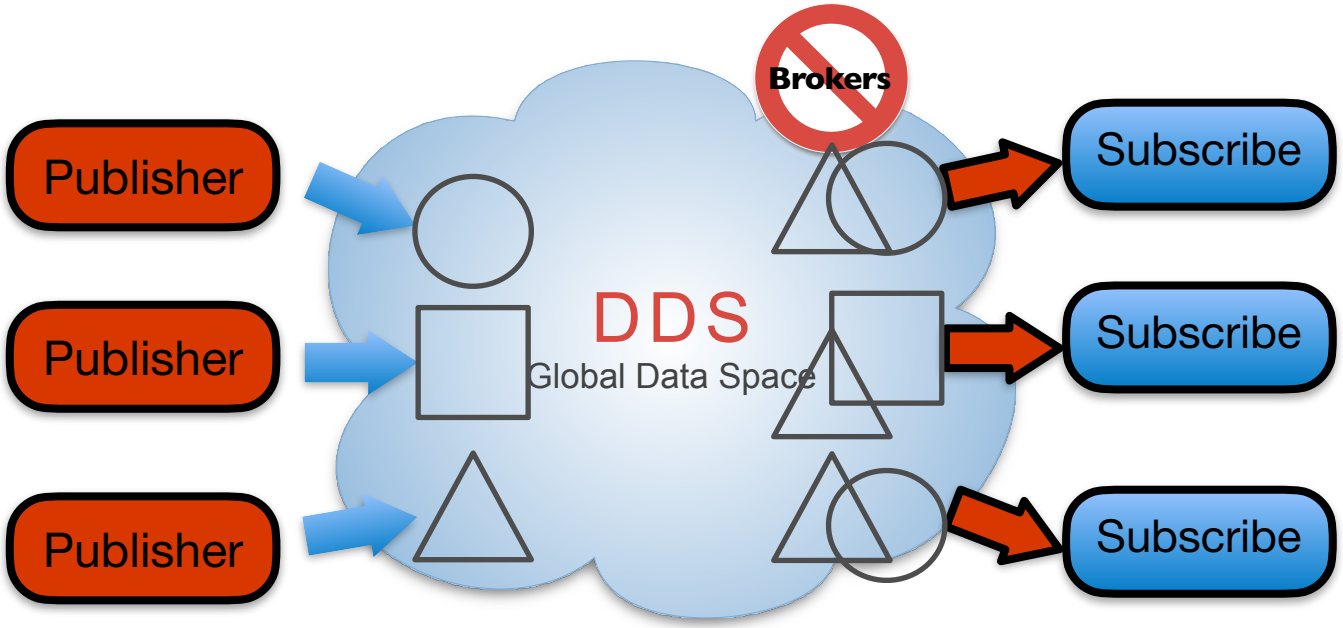
How Does it Work?

- Publishers and Subscribers express their intent to produce/consume specific type of data, e.g., Topics.



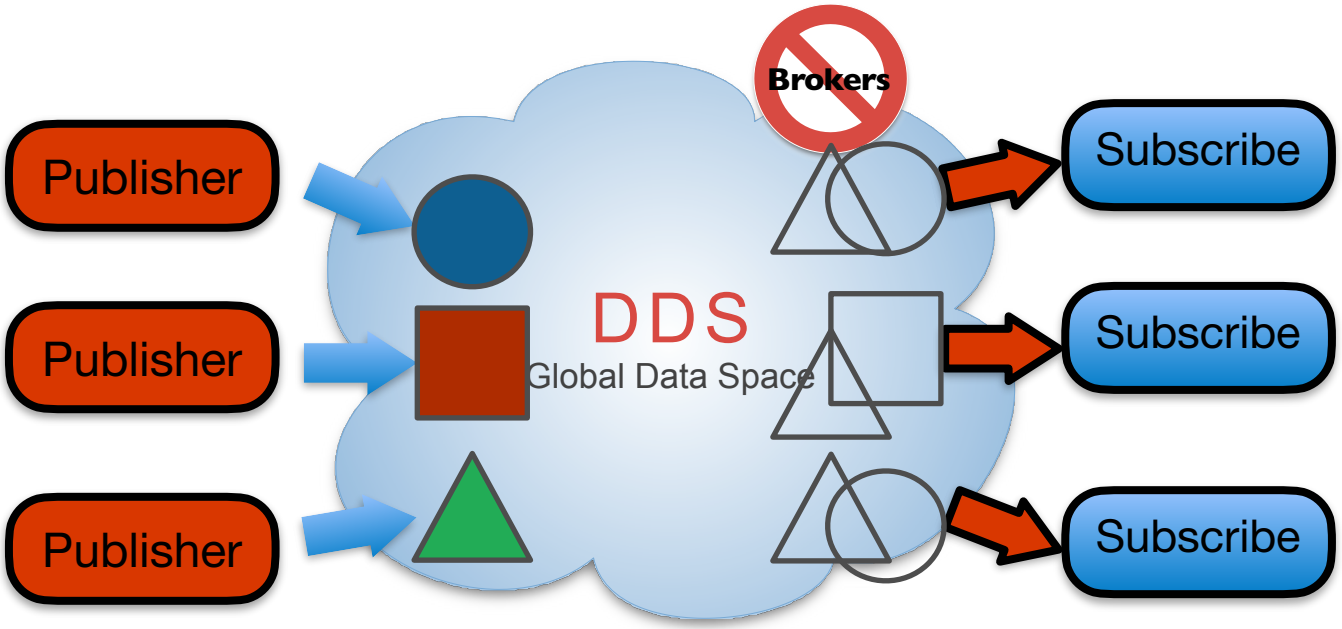
How Does it Work?

- Subscriptions are matched by taking into account topics (name, data type and QoS).



How Does it Work?

- Subscriptions are dynamically matched and Data flows from Publisher to Subscribers.

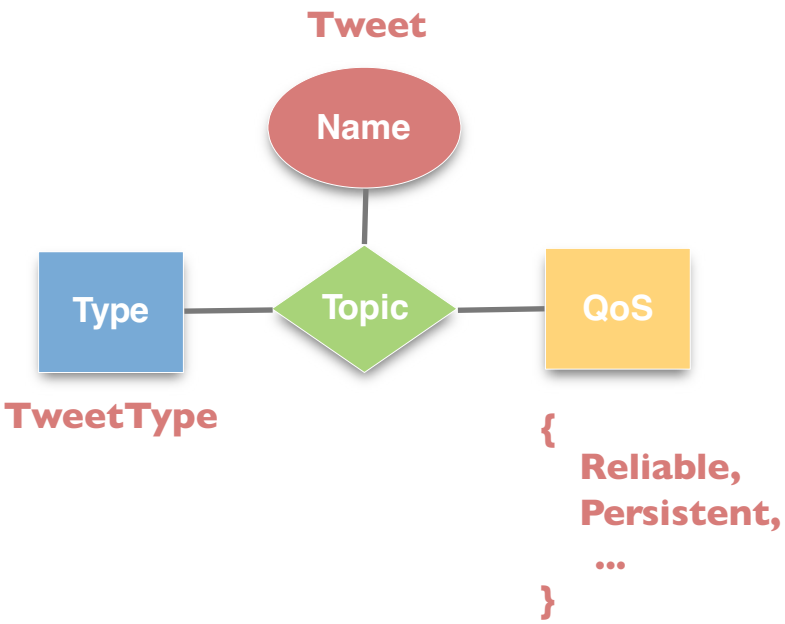


Defining Data

A “Tweet” with DDS

Topic:

- Unit of information exchanged between Publisher and Subscribers.
- An association between a unique name, a type and a QoS setting



A “Tweet” with DDS

Topic Type:

- ***Type*** describing the data associated with one or more Topics.
- A Topic type can have a key represented by an arbitrary number of attributes.
- Expressed in IDL (or XML)

```
struct TweetType {
    string userID;
    string tweet;
};
#pragma keyless Tweet userID
```

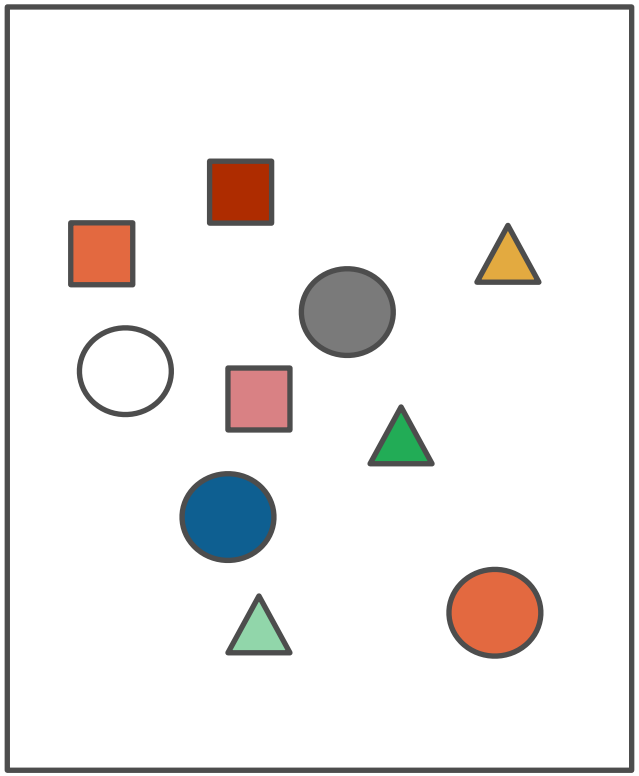
```
struct ShapeType {
    long x;
    long y;
    long shapesize;
    string color;
};
#pragma keyless ShapeType color
```

DDS Topic Instances and Samples

Topic Instances:

- Each key value identifies a unique Topic Instance.
- Topic's instance lifetime can be explicitly managed in DDS.

```
struct ShapeType {
     long x;
     long y;
     long shapesize;
     string color;
};
#pragma keyless ShapeType color
```

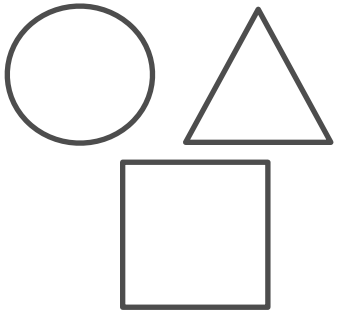


Topic Samples:

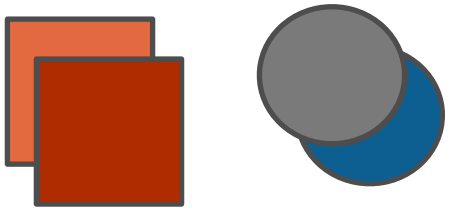
- The values assumed by a Topic Instance over time are referred as Instance Sample.

Topic/Instances/Sample Recap

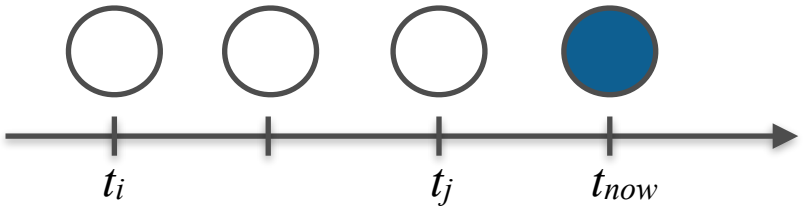
Topics



Instances



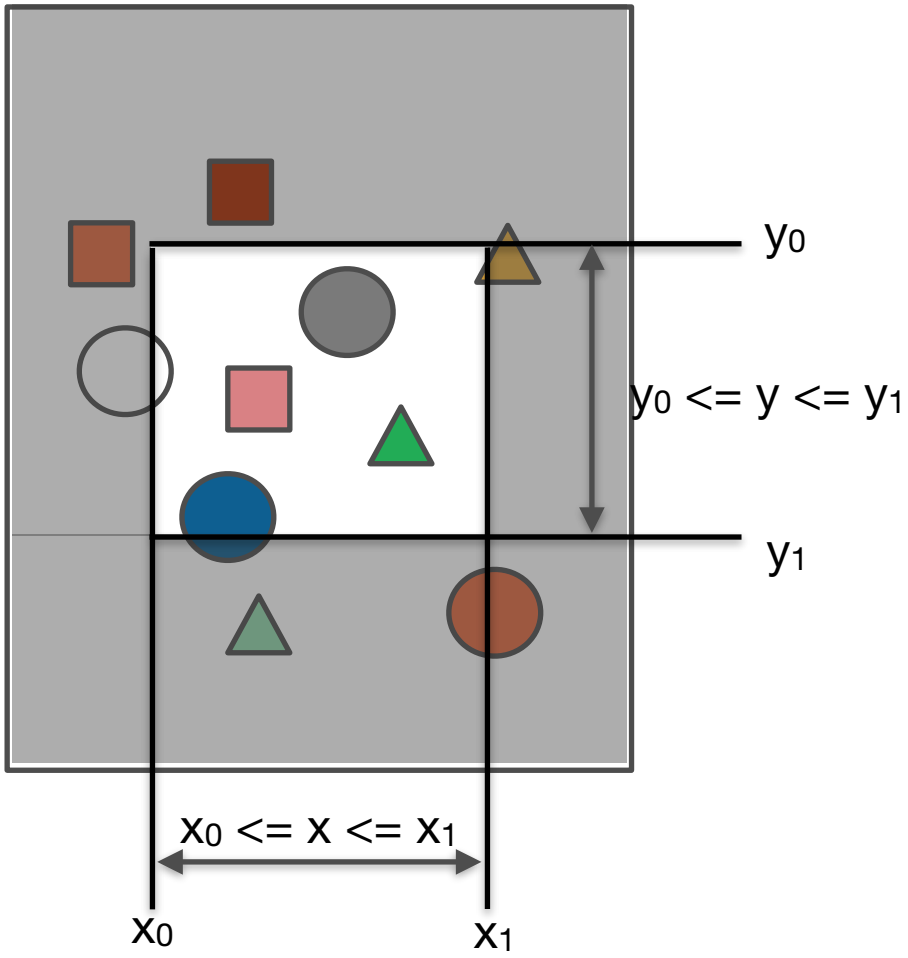
Samples



DDS Topic Instances and Samples

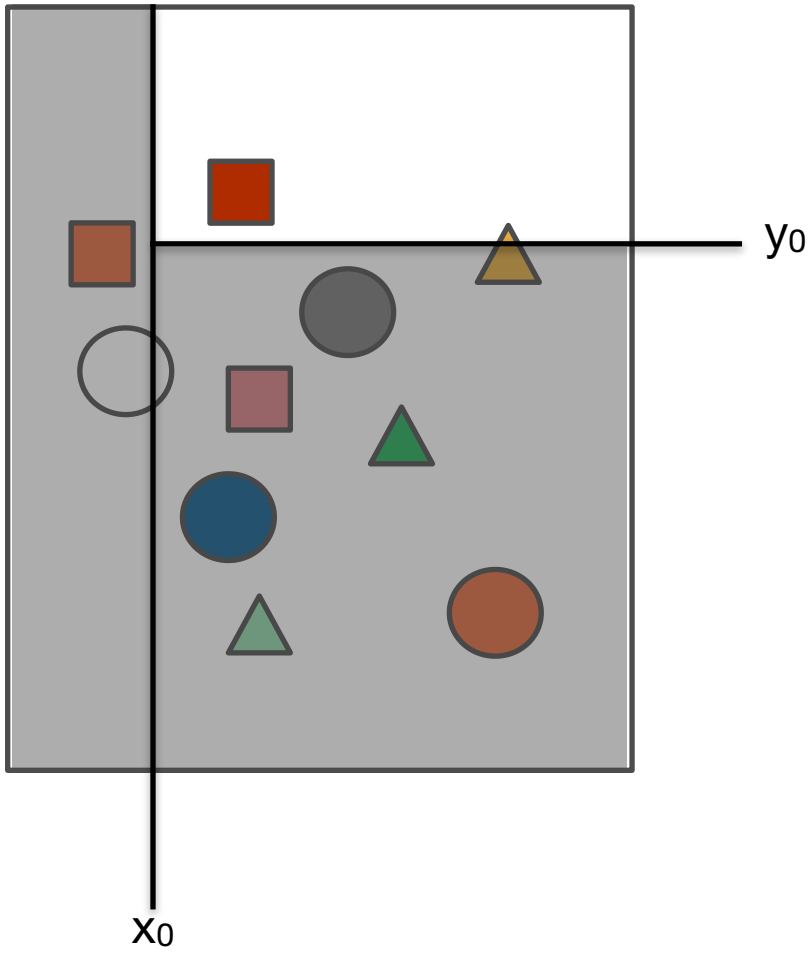
- DDS allows the use of a subset of SQL92 to specify **content-filtered Topics**
- Content filters can be applied on the entire content of the Topic Type
- Content filters are applied by DDS each time a new sample is produced/ delivered.

(x between (RANGE x_0 and x_1))
AND
(y between (RANGE y_0 and y_1))



DDS Topic Instances and Samples

- Subscribed Topics can be seen locally as “Tables”
- A subset of SQL92 can be used for performing queries.
- Queries are performed under user control and provide a result that depends on the current snapshot of the system, e.g., samples currently available



color	x	y	shapesize
red	57	62	50
blue	90	85	50
yellow	30	25	50

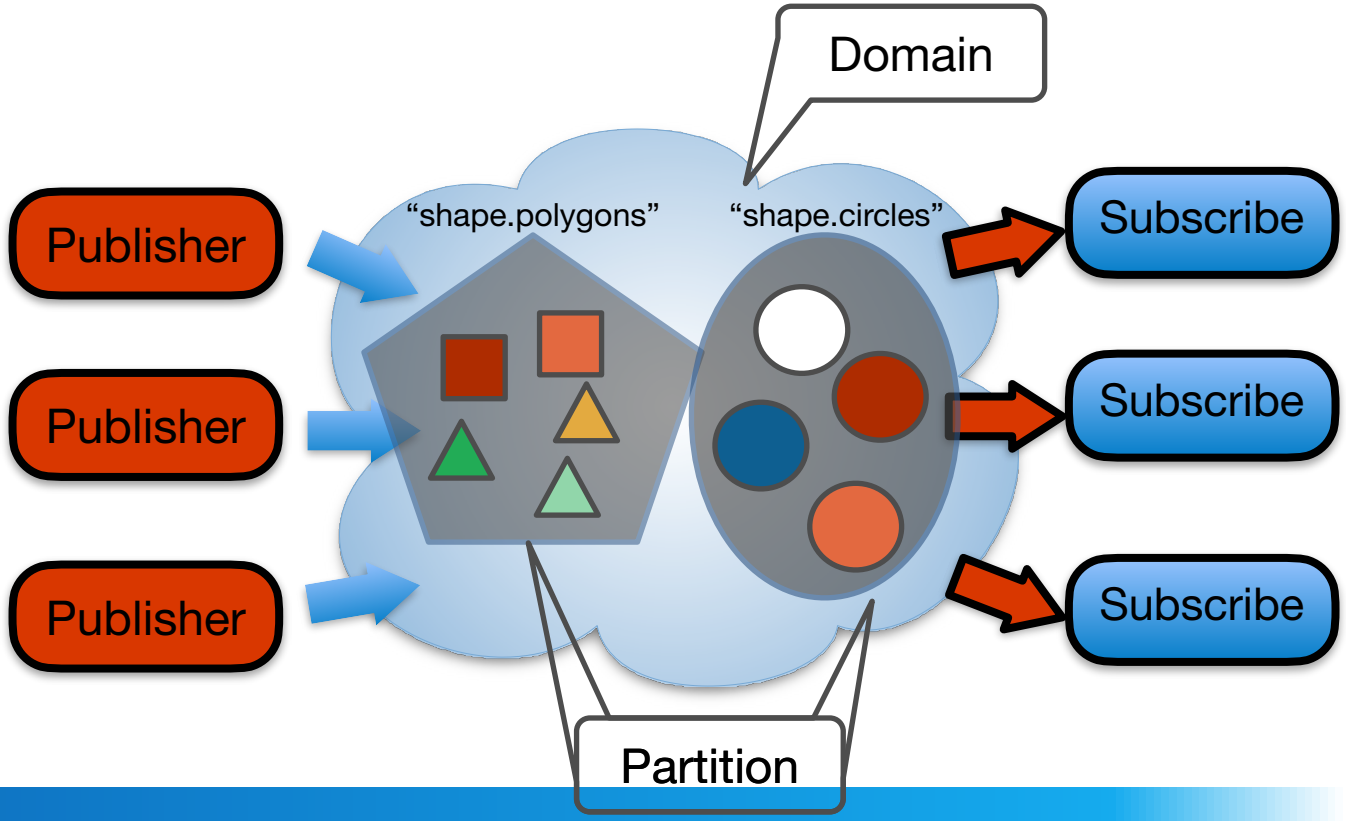
x > 25 AND y < 55

color	x	y	shapesize
yellow	30	25	50

Organizing Data

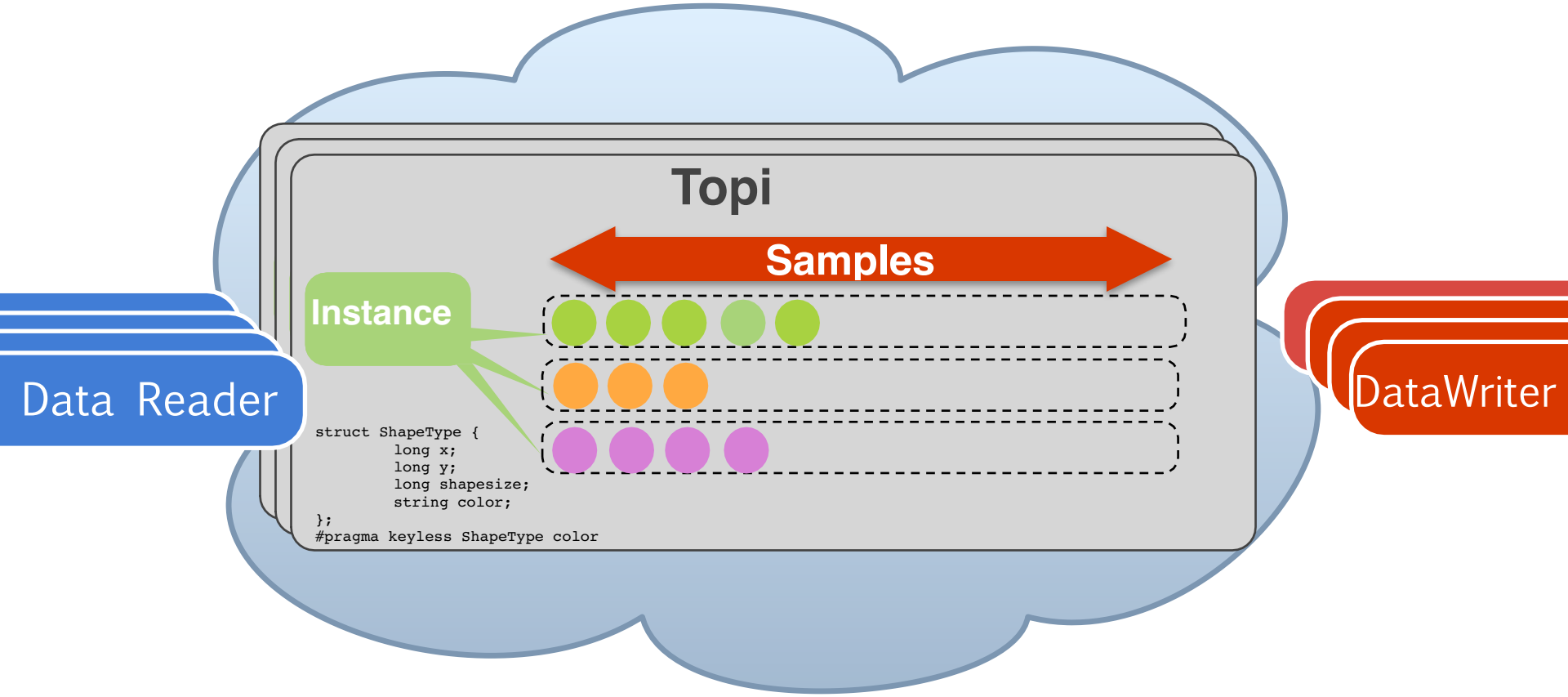
DDS Partitions

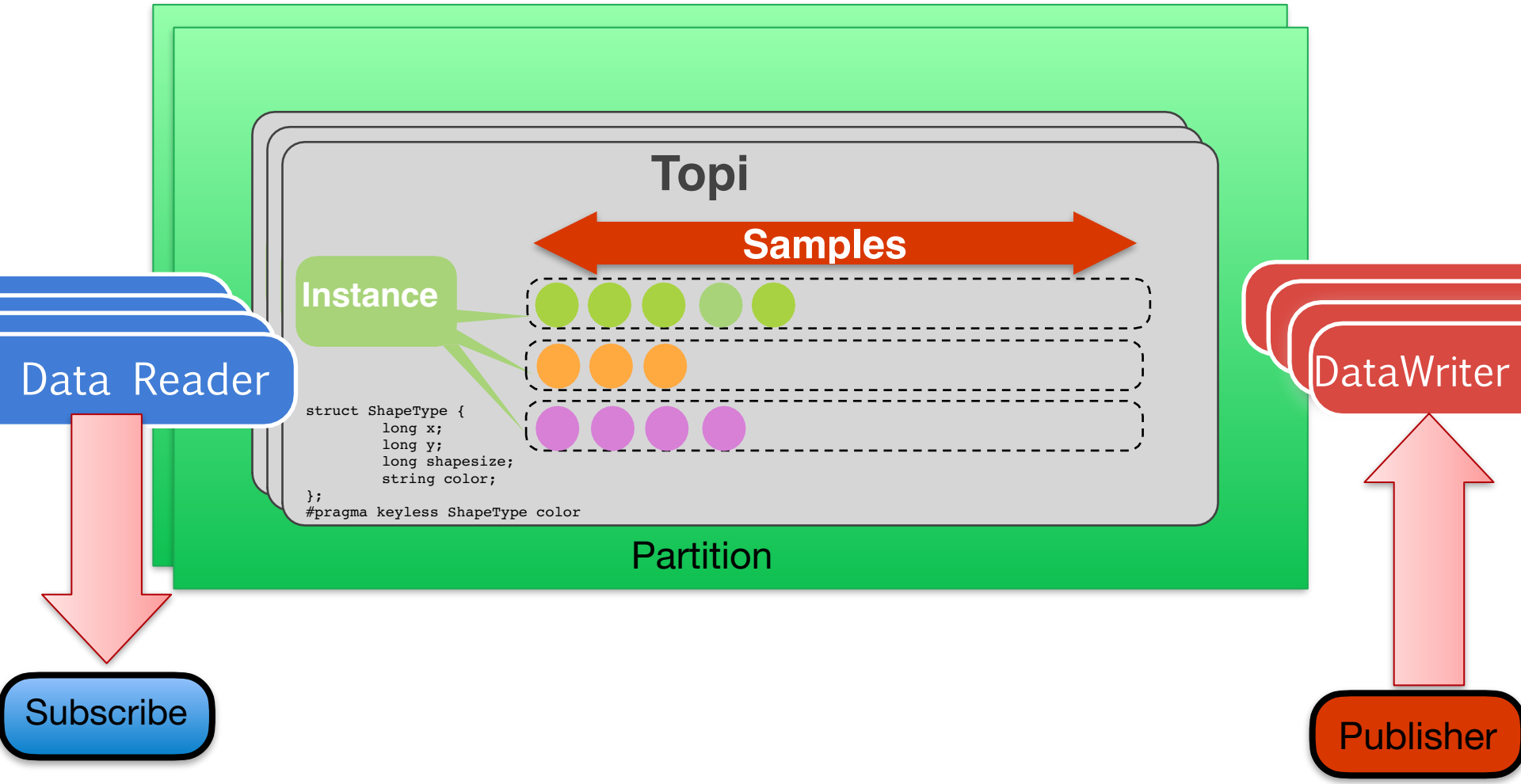
- All DDS communication confined within a **Domain**.
- A domain can organized into **Partitions**.
- Partitions can be used as **subjects** organizing the flow of data.
- Publishers/Subscribers can connect to a Partitions' List which might also contain wildcards, e.g. shape.*
- Topics are published and subscribed across one or more Partitions.

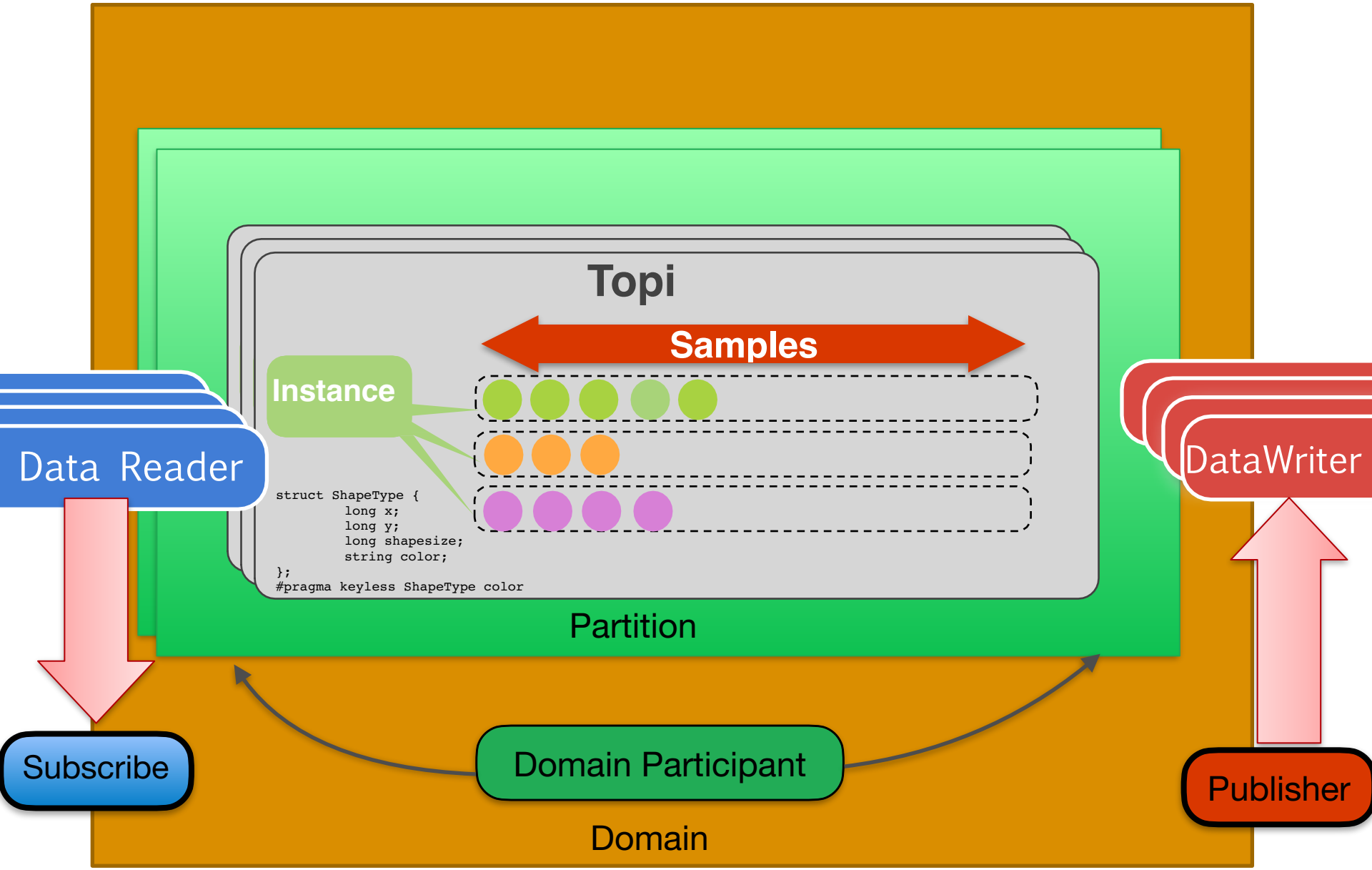


Quality of Services

Anatomy of a DDS Application




Topi
Samples
Instance
Data Reader
DataWriter
struct ShapeType {
long x;
long y;
long shapesize;
string color;
};
#pragma keyless ShapeType color
Partition
Subscribe
Publisher


Topi
Samples
Instance
Data Reader
DataWriter
struct ShapeType {
long x;
long y;
long shapesize;
string color;
};
#pragma keyless ShapeType color
Partition
Subscribe
Domain Participant
Publisher
Domain

QoS Model

- QoS-Polices are sued to control relevant properties of DDS entities, including:
  - Real-Time Delivery
  - Bandwidth
  - Redundancy
  - Persistence

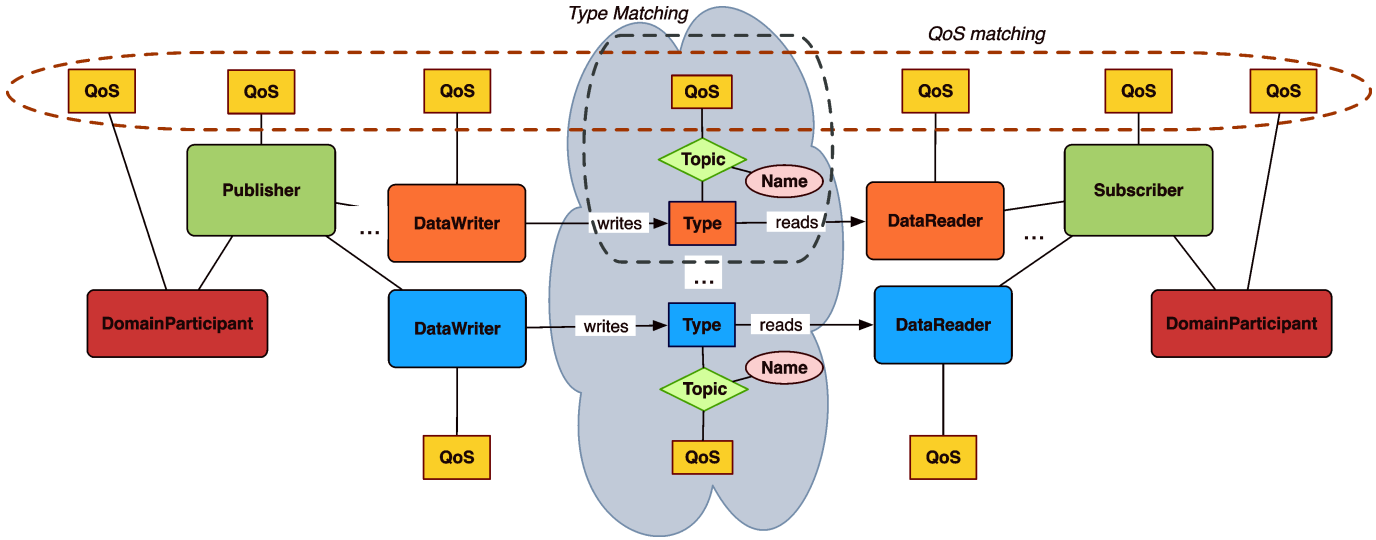
- QoS-Policies are matched based on a Request vs. Offered (RxO) Model thus QoS-enforcement.

Configuration

Real-Time Delivery
Redundancy
Persistence
Bandwidth

DDS Request vs. Offer QoS (RxO) Model

- DDS uses a ‘request vs. offer’ QoS-matching approach:
  - The request is granted if and only if the QoS request does not exceed the QoS produced by the data writer.
- The grant ensures that:
  - types are preserved end-to-end due to the topic type matching, and
  - end-to-end QoS invariants are also preserved.



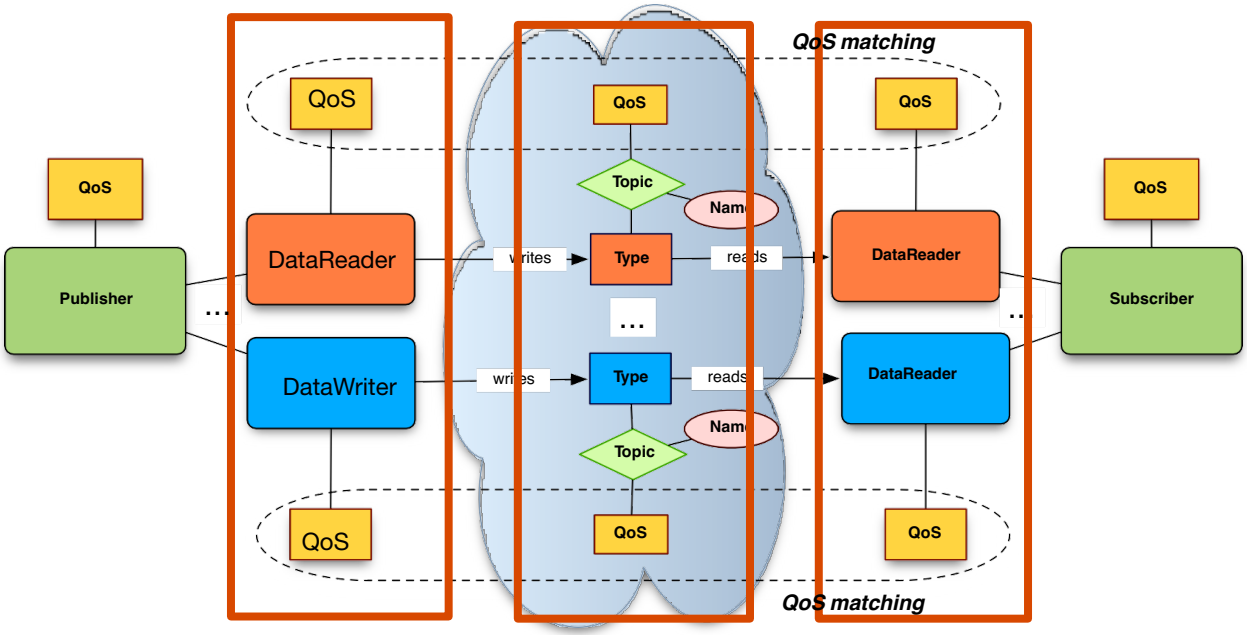
QoS Policy	Applicability	RxO	Modifiable	
DURABILITY	T, DR, DW	Y	N	Data Availability
DURABILITY SERVICE	T, DW	N	N	
LIFESPAN	T, DW	N/A	Y	
HISTORY	T, DR, DW	N	N	
PRESENTATION	P, S	Y	N	Data Delivery
RELIABILITY	T, DR, DW	Y	N	
PARTITION	P, S	N	Y	
DESTINATION ORDER	T, DR, DW	Y	N	
OWNERSHIP	T, DR, DW	Y	N	
OWNERSHIP STRENGTH	DW	N/A	Y	
DEADLINE	T, DR, DW	Y	Y	Data Timeliness
LATENCY BUDGET	T, DR, DW	Y	Y	
TRANSPORT PRIORITY	T, DW	N/A	Y	
TIME BASED FILTER	DR	N/A	Y	Resources
RESOURCE LIMITS	T, DR, DW	N	N	
ENTITY FACTORY				Configuration
USER DATA	DP, DR, DW	N	Y	
TOPIC DATA	T	N	Y	
GROUP DATA	P, S	N	Y	
LIVELINESS	T, DR, DW	Y	N	
WRITER DATA LIFECYCLE	DW	N/A	Y	Lifecycle
READER DATA LIFECYCLE	DR	N/A	Y	

Topic (T), DataWriter (DW), DataReader (DR), DomainParticipant (DP), Publisher (P) or subscriber (S)

Controlling Reliability

Reliability

- The RELIABILITY QoS indicates the level of guarantee offered by the DDS in delivering data to subscribers.

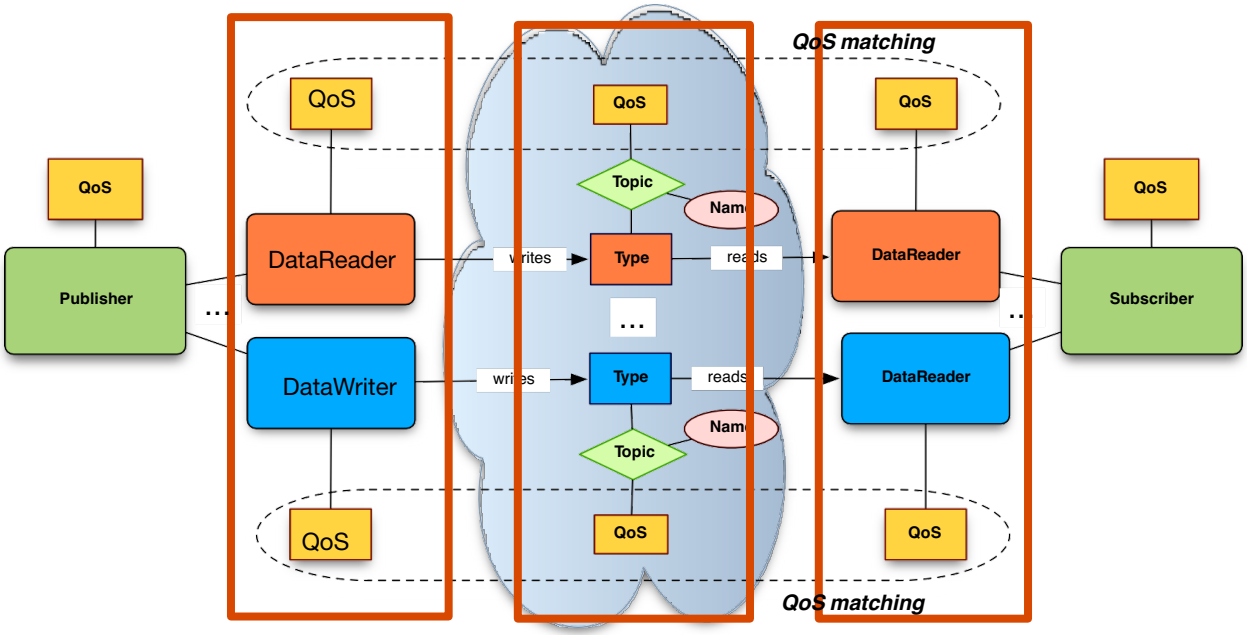


QoS Policy	Applicability	RxO	Modifiable
RELIABILITY	T, DR, DW	Y	N

Reliability

Possible variants are:

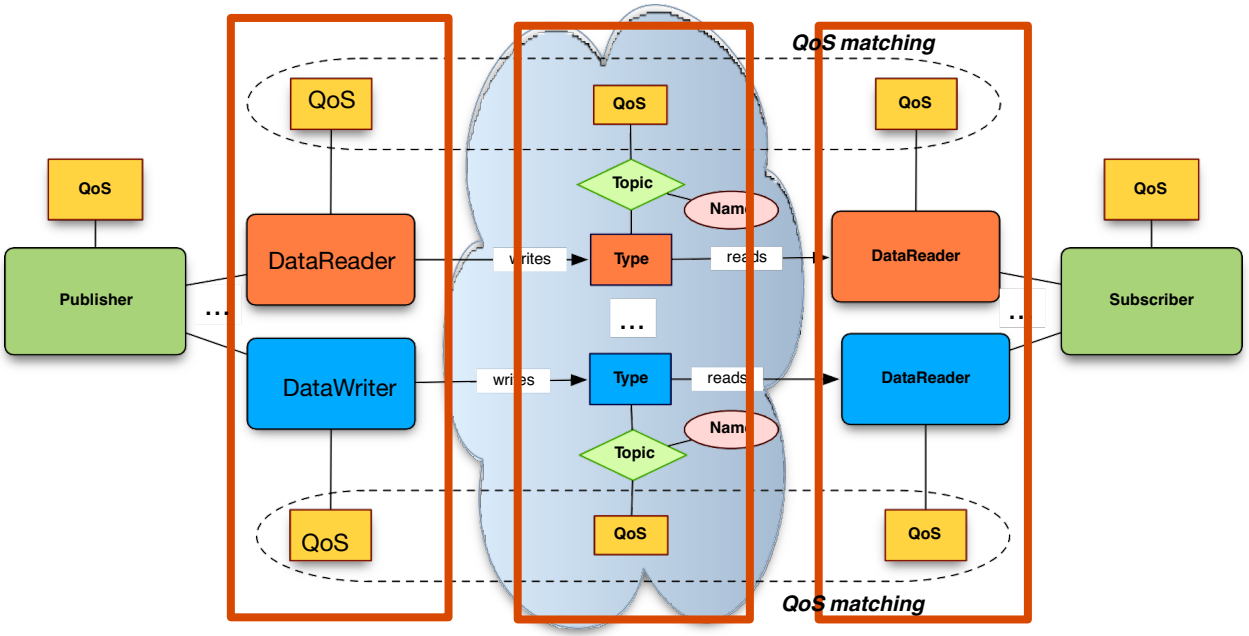
- **Reliable**: In steady-state the middleware guarantees that all samples in the DataWriter history will ***eventually*** be delivered to all the DataReader.
- **Best Effort**: Indicates that it is acceptable to not retry propagation of any samples.



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

History

The **HISTORY** QoS policy controls whether the DDS should deliver only the most recent value, attempt to deliverall intermediate values, or do something in between.

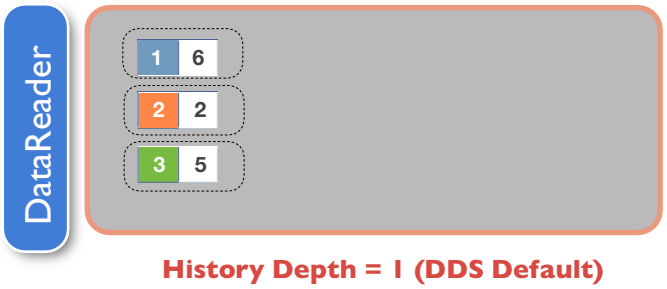


QoS Policy	Applicability	RxO	Modifiable
HISTORY	T, DR, DW	N	N

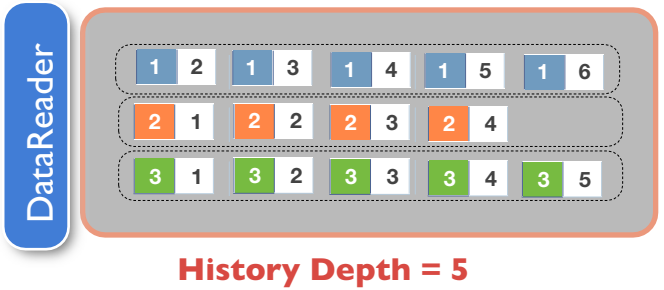
History

The policy can be configured to provide the following semantics:

- **Keep Last**: The DDS will only attempt to keep the most recent "depth" samples of each instance of data identified by its key.
- **Keep All**: The DDS will attempt to keep all the samples of each instance of data identified by its key.



History Depth = 1 (DDS Default)

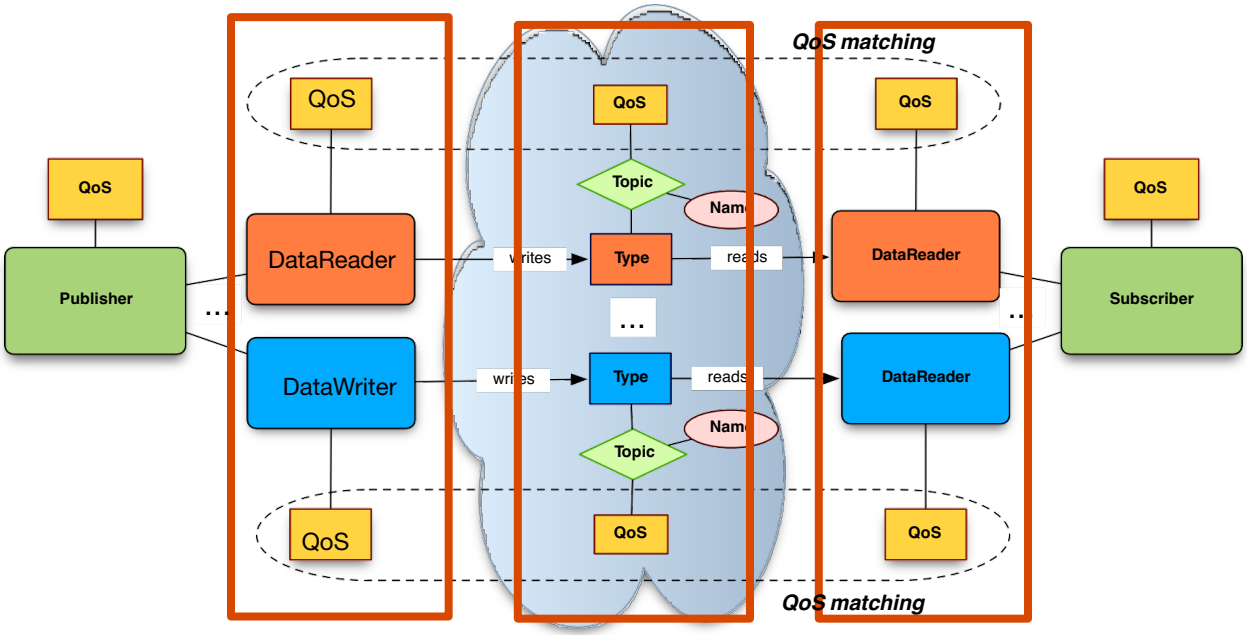


History Depth = 5

Controlling Real-Time Properties

Deadline

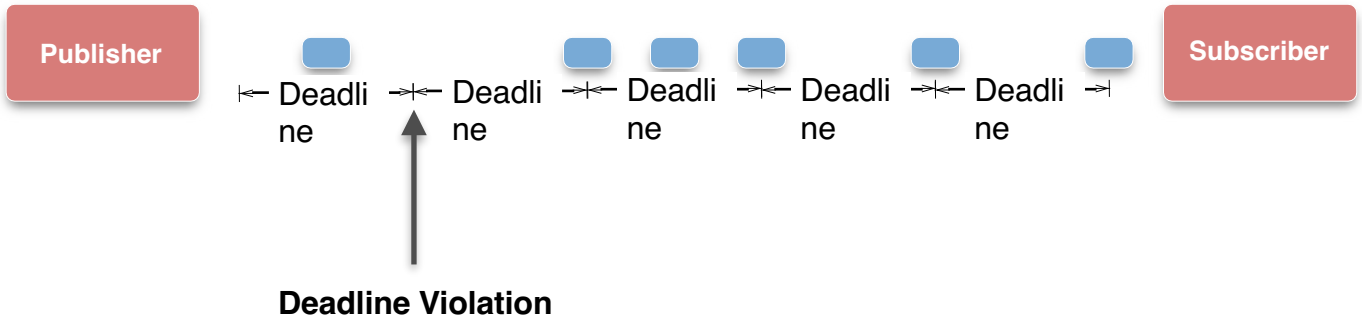
- The **DEADLINE** QoS policy allows to define the **maximum inter-arrival time** between data samples



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

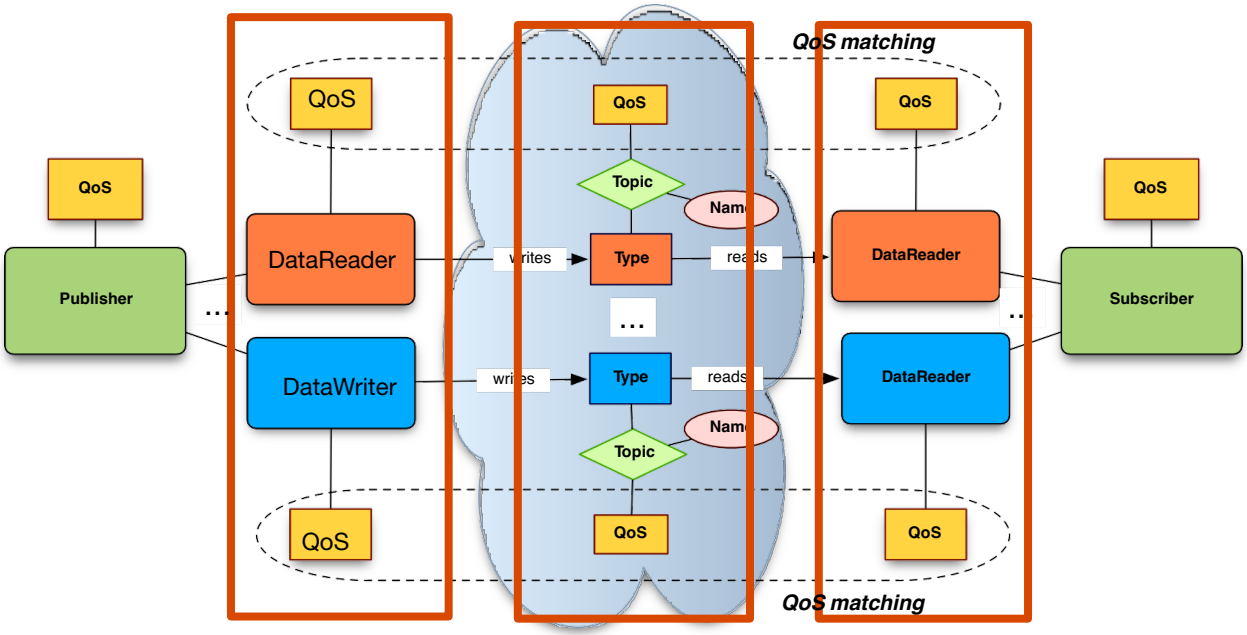
Deadline QoS

- DataWriter indicates that the application commits to write a new value at least once every deadline period.
- DataReaders are notified by the DDS when the DEADLINE QoS contract is violated.



Latency Budget

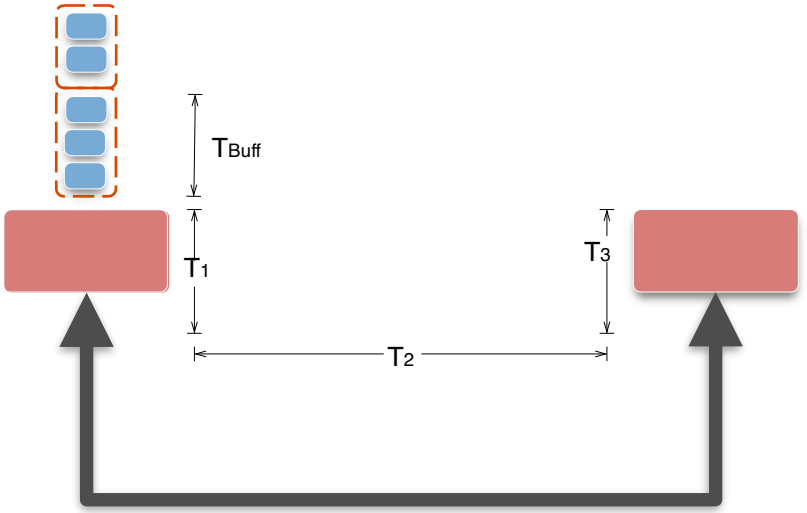
- The LATENCY_BUDGET QoS policy specifies the maximum acceptable delay from the time the data is written until the data is inserted in the receiver's application-cache.



QoS Policy	Applicability	RxO	Modifiable
Latency Budget	T, DR, DW	Y	Y

Latency Budget

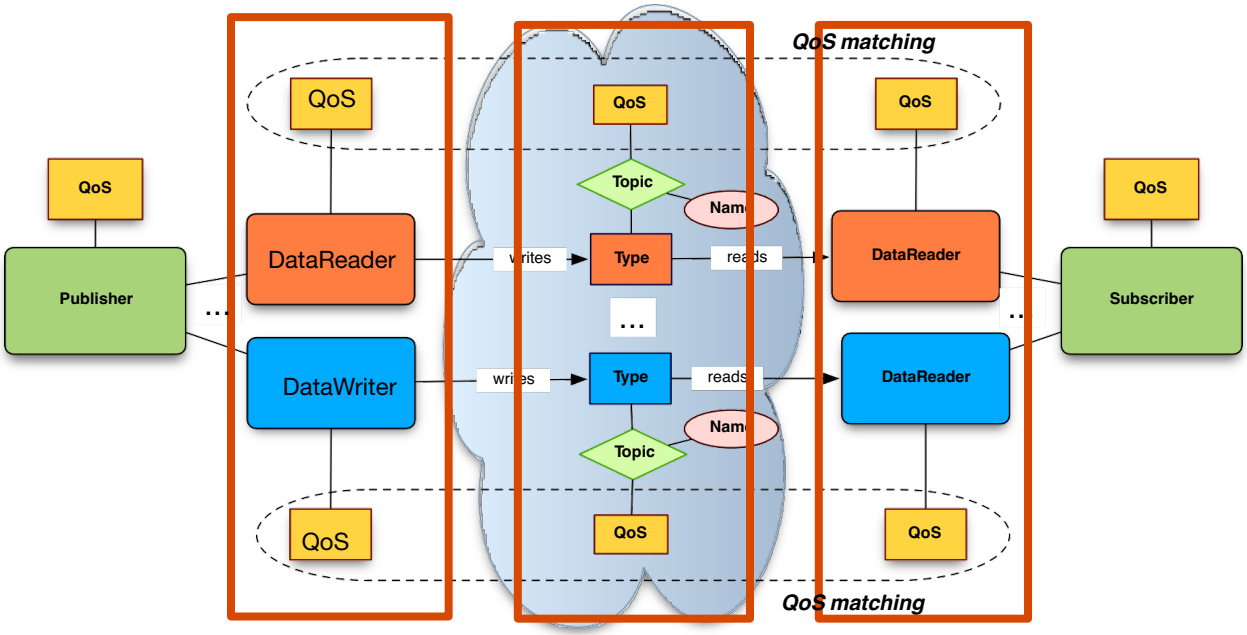
- The default value of the duration is zero indicating that the delay should be minimized.
- This policy is a hint to the DDS, not something that must be monitored or enforced.



Latency Budget = Latency = $T_{Buff} + T_1 + T_2 + T_3$

Transport Priority

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

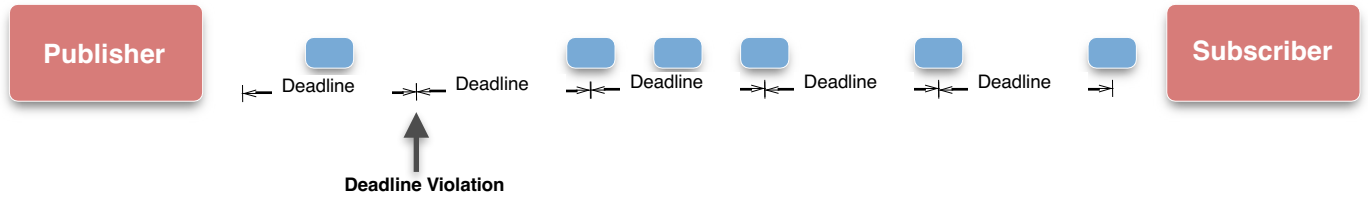


QoS Policy	Applicability	RxO	Modifiable
TRANSPORT_PRIORITY	T, DW	-	Y

Putting it Together

The real-time properties with which data is delivered to applications is impacted in DDS by the following qualities of service:

- **TRANSPORT_PRIORITY**
- **LATENCY_BUDGET**



- In addition, DDS provides means for detecting performance failure, e.g., Deadline miss, by means of the **DEADLINE** QoS.
- Given a periodic task-set {T} with periods D_i (with $D_i < D_{i+1}$) and deadline equal to the period, than QoS should be set as follows:
  - Assign to each task T_i a TRANSPORT_PRIORITY P_i such that P_i
  - Set for each task T_i a DEADLINE QoS of D_i
  - For maximizing throughput and minimizing resource usage set for each T_i a LATENCY_BUDGET QoS between $D_i/2$ and $D_i/3$ (this is a rule of thumb, the upper bound is D_i-(RTT/2)).

Controlling Replication

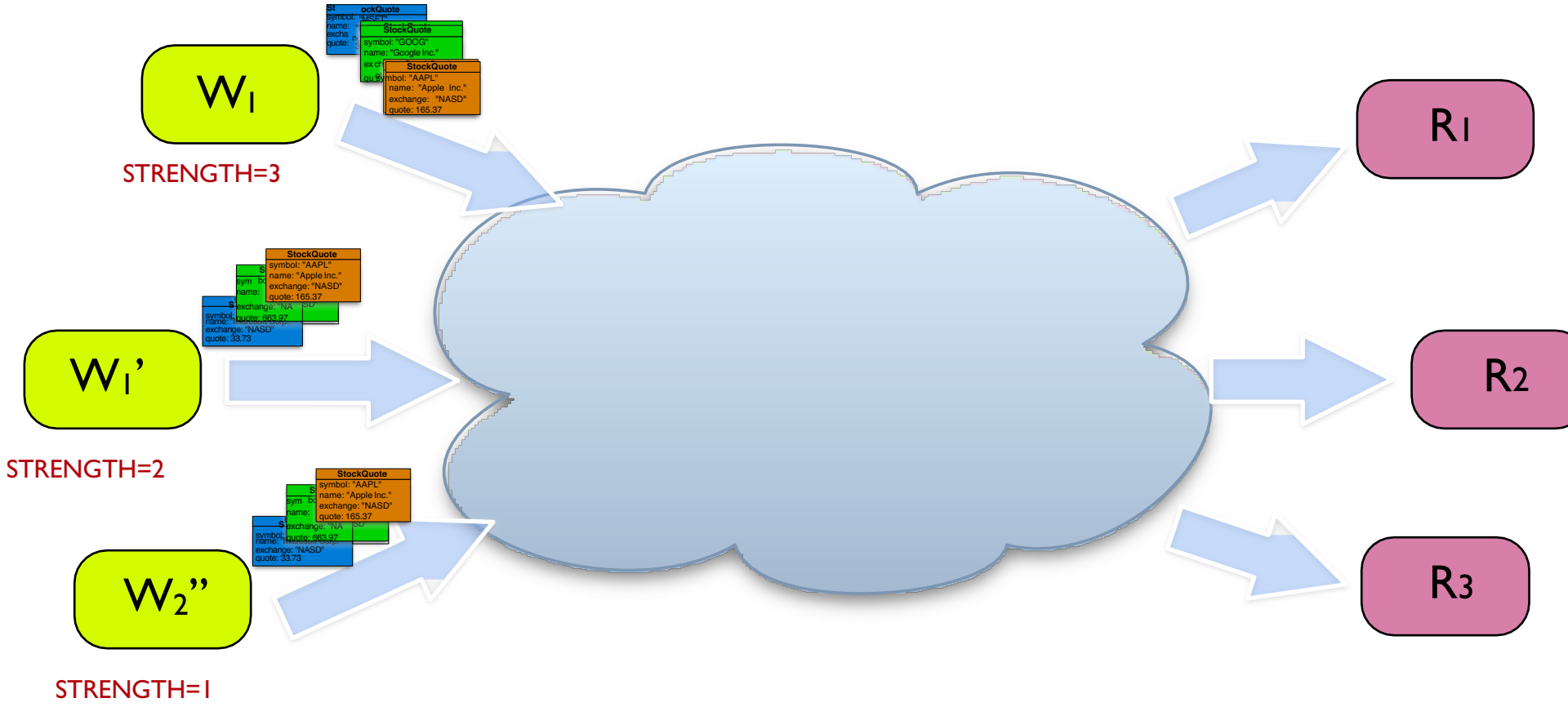
Availability

Availability of data producers can be controlled via two QoS Policies:

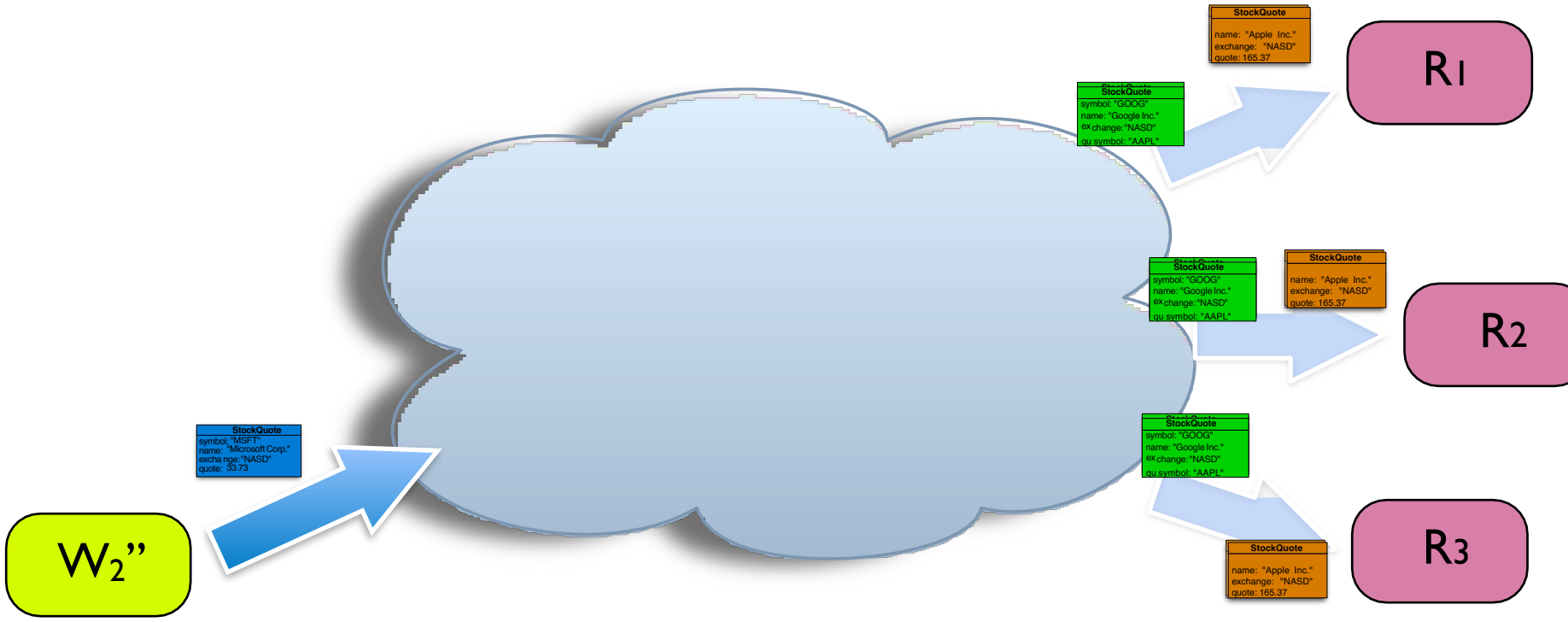
- **OWNERSHIP**
- **OWNERSHIP STRENGTH**

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

Availability



Availability



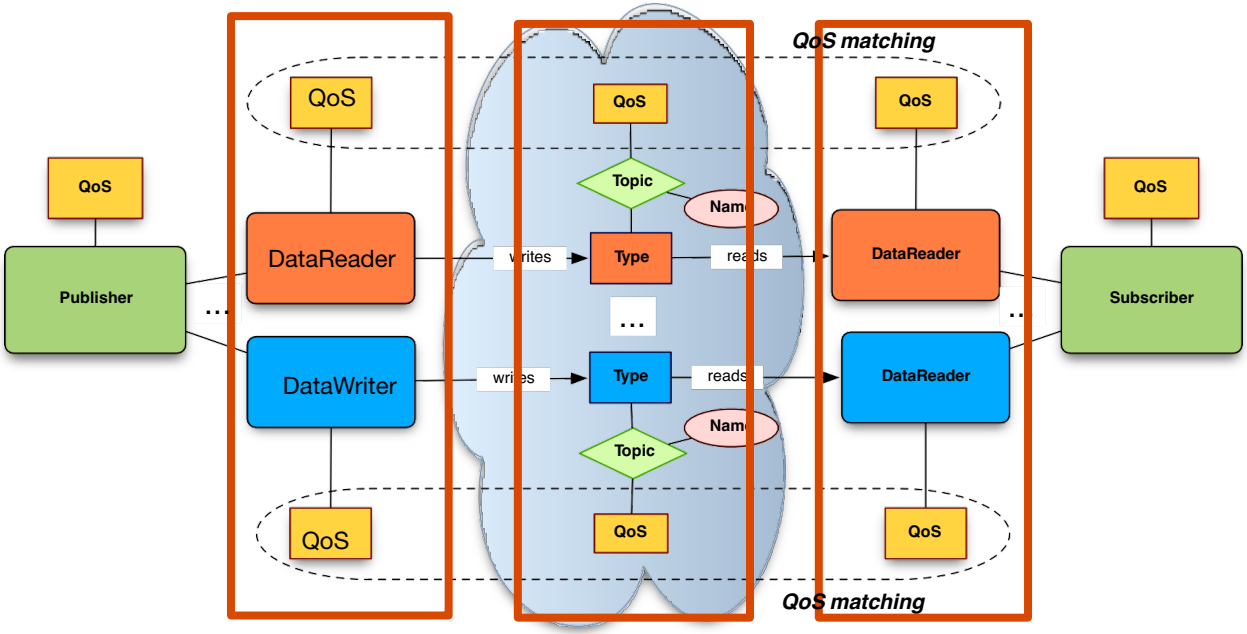
STRENGTH=1

Controlling the Consistency Model

Durability

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

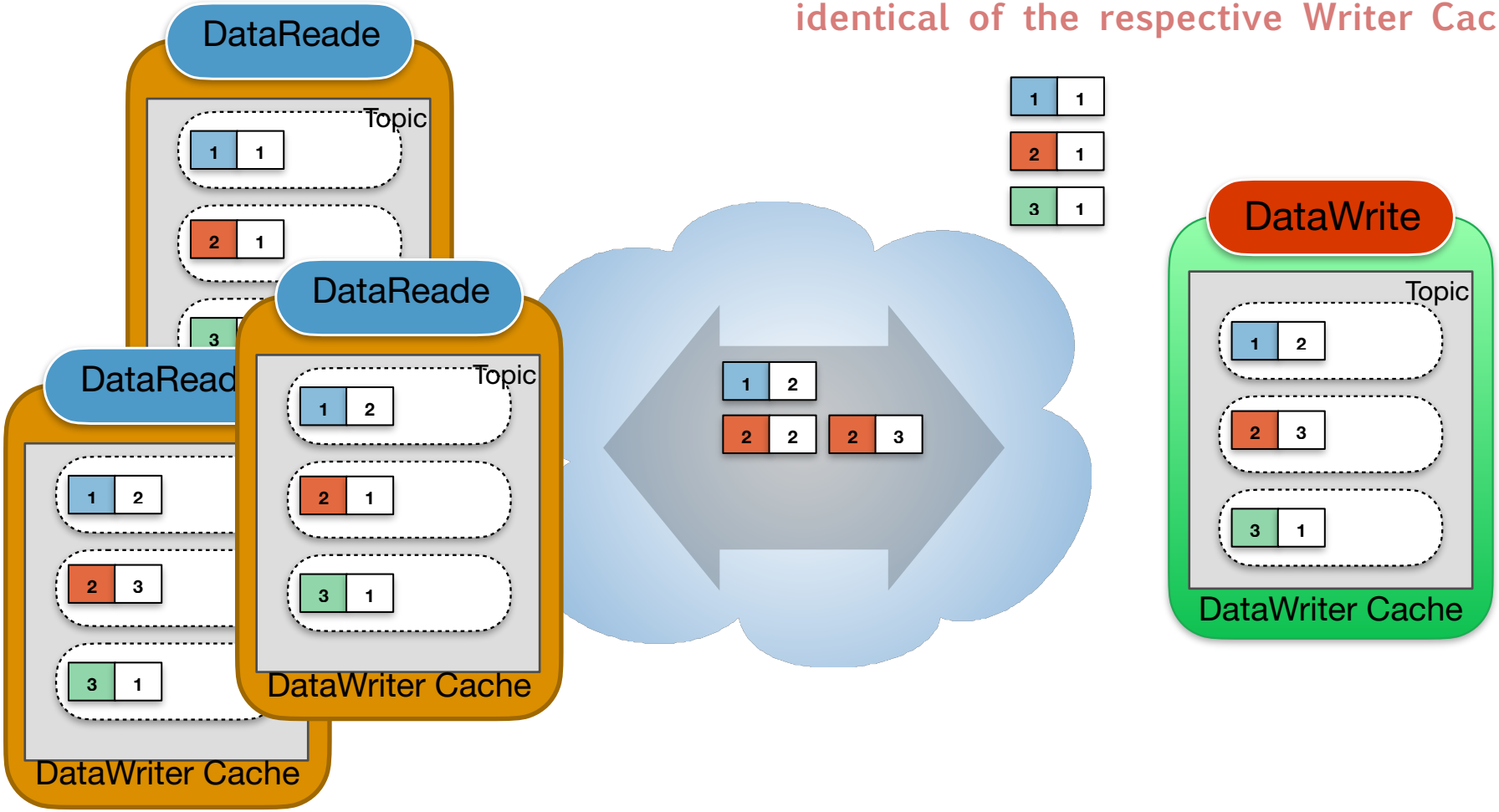
- **Volatile:** No need to keep data instances for late joining data readers
- **Transient Local:** Data instance availability for late joining data reader is tied to the data writer availability
- **Transient:** Data instance availability outlives the data writer
- **Persistent:** Data instance availability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



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

Eventual Consistency & R/WCaches

Under an Eventual Consistency Model, DDS guarantees that all matched Reader Caches will eventually be identical of the respective Writer Cache



Durability

The DDS Consistency Model is a property that can be associated to Topics or further refined by Reader/Writers. The property is controlled by the following QoS Policies:

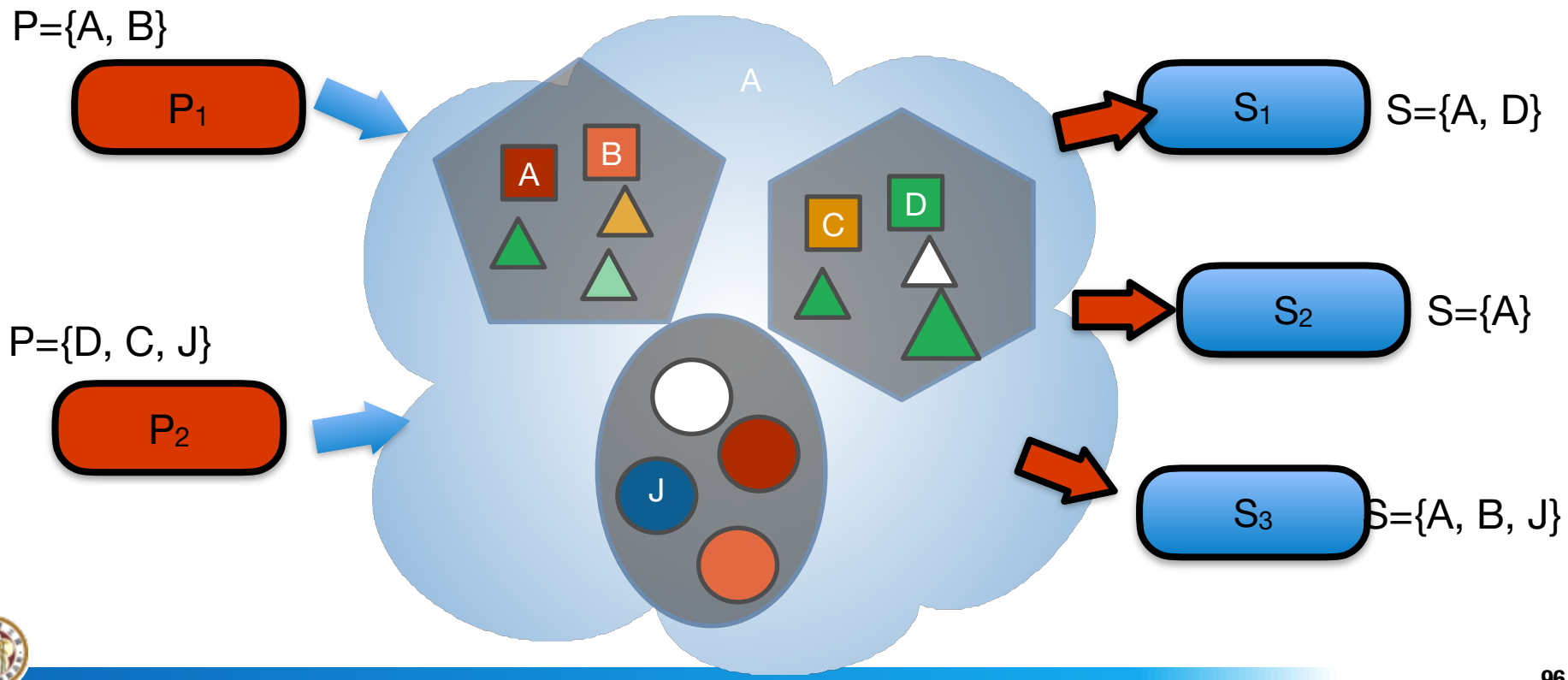
- DURABILITY
  - VOLATILE | TRANSIENT_LOCAL | TRANSIENT | PERSISTENT
- LIFESPAN
- RELIABILITY
  - RELIABLE | BEST_EFFORT
- DESTINATION ORDER
  - SOURCE_TIMESTAMP | DESTINATION_TIMESTAMP

QoS Policy	Applicability	RxO	Modifiable
DURABILITY	T, DR, DW	Y	N
LIFESPAN	T, DW	-	Y
RELIABILITY	T, DR, DW	Y	N
DESTINATION ORDER	T, DR, DW	Y	N

QoS & Consistency Model

	DURABILITY	RELIABILITY	DESTINATION_ORDER	LIFESPAN
Eventual Consistency (No Crash / Recovery)	VOLATILE	RELIABLE	SOURCE_TIMESTAMP	INF.
Eventual Consistency (Reader Crash / Recovery)	TRANSIENT_LOCAL	RELIABLE	SOURCE_TIMESTAMP	INF.
Eventual Consistency (Crash/Recovery)	TRANSIENT	RELIABLE	SOURCE_TIMESTAMP	INF.
Eventual Consistency (Crash/Recovery)	PERSISTENT	RELIABLE	SOURCE_TIMESTAMP	INF.
Weak Consistency	ANY	ANY	DESTINATION_TIMESTAMP	ANY
Weak Consistency	ANY	BEST_EFFORT	ANY	ANY
Weak Consistency	ANY	ANY	ANY	N

	DURABILITY	RELIABILITY	DESTINATION_ORDER	LIFESPAN	
Eventual Consistency (Reader Crash / Recovery)	TRANSIENT_LOCAL	RELIABLE	SOURCE_TIMESTAMP	INF.	**{A}**
Eventual Consistency (Crash/Recovery)	TRANSIENT	RELIABLE	SOURCE_TIMESTAMP	INF.	**{B}**
Weak Consistency	ANY	ANY	ANY	N	**{J}**



Summary

- Message exchange for mission critical systems require real-time guarantee. Communications in (self-driving) cars is one of the latest common use scenario.
- It is ***unrealistic*** and ***impossible*** to guarantee all message exchanges to meet real-time constraints due to limited resources.
  - ROS is one of the real-time communication protocols to be implemented on off-the-shelf networks.
  - Understand the requirements and available resources are mandatory to design the systems.
  - Formal analysis are available and should be taken before dumping data into the network.

Reference

- Forsberg, Andreas. “Comparison of FlexRay and CAN-bus for Real-Time Communication.” (2009).
- The DDS Tutorial, Angelo Corsaro.