

Data Distribution Service

MOHAMAD SHAABAN



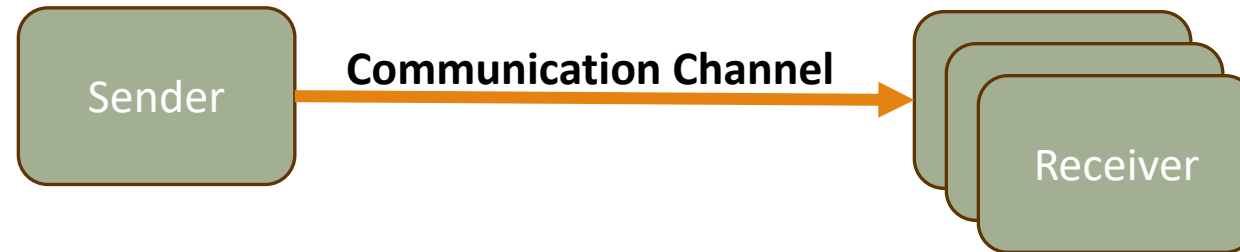
Communication Models

One to One



Sockets

One to Many



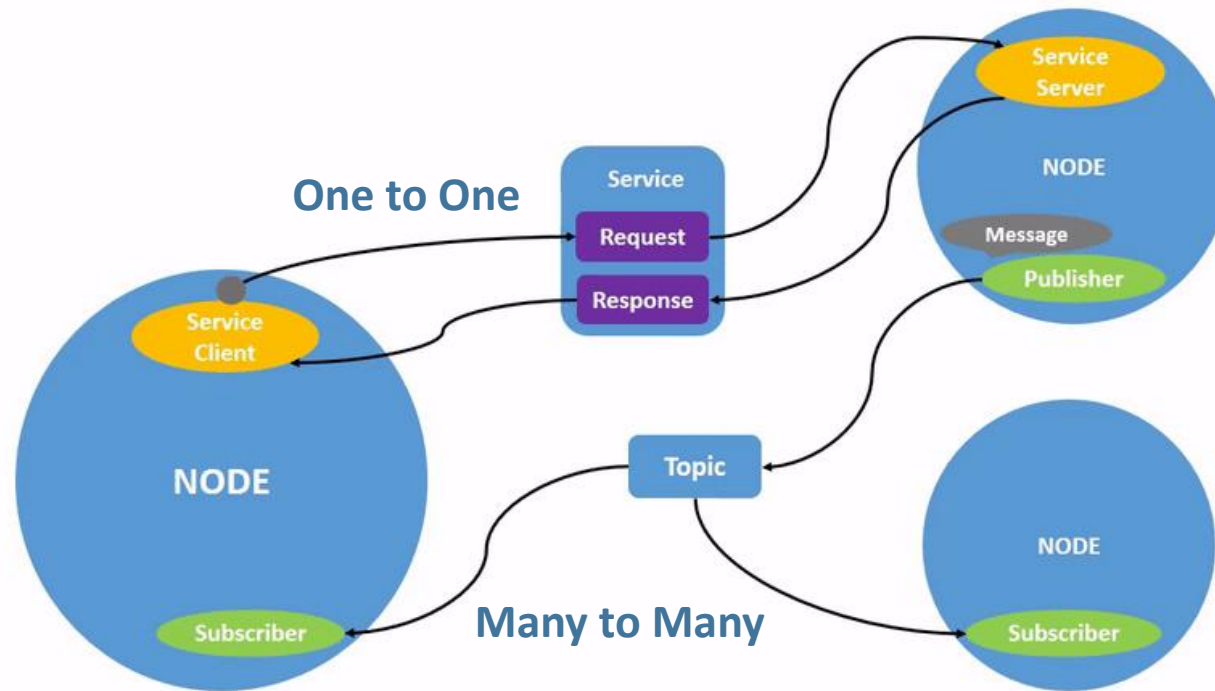
Server/Client

Many to Many



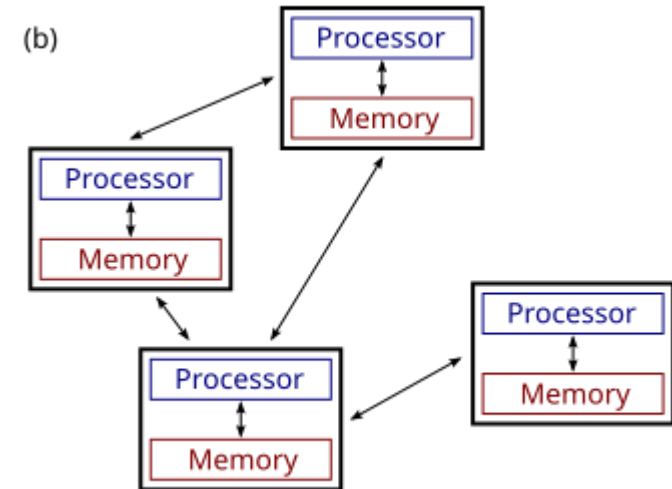
Pub/Sub

Robotics Communication Model (ROS)



What is DDS?

- A data-centric communication protocol for distributed software systems.
- Protocol is designed by **Object Management Group (OMG)**
- Based on the **Data-Centric Publish Subscribe (DCPS)** model.
 - **Publication Entities**: Define information-generating objects.
 - **Subscription Entities**: Define information-consuming objects.
 - **Configuration Entities**
- Defines **APIs** and **Communication Semantics** for data providers and consumers.



Example Project



DDS Middleware Implementations

- **Open-Source Implementations**

- Fast DDS (eProsima)
 - High performance and low latency
 - Extensive QoS support and ROS2 compatibility
 - [Apache-2.0 license](#)
- Cyclone DDS (Eclipse Foundation)
 - Lightweight and optimized for embedded systems.
 - Popular in ROS2 environments.
 - [Eclipse-2.0](#)



DDS Middleware Implementations

- **Commercial Implementations:**

- RTI Connex DDS:
 - Industry-leading performance and reliability.
 - Comprehensive tools for monitoring and debugging.
- OpenSplice DDS:
 - Focused on scalability and fault tolerance.
 - Offers both open-source and commercial versions.





Fast DDS

Key DCPS Elements

- **Publisher:**

- Creates and configures **DataWriters**.
- **DataWriter**: Handles the actual publication of data.
- Publishes messages under assigned **Topics**.

- **Subscriber:**

- Receives data published under subscribed **Topics**.
- Manages **DataReaders** that notify the application of new data.

Key DCPS Elements

- **Topic:**

- Binds publications and subscriptions.
- Ensures uniform data types between publishers and subscribers.
- Unique within a **DDS domain**.

- **Domain:**

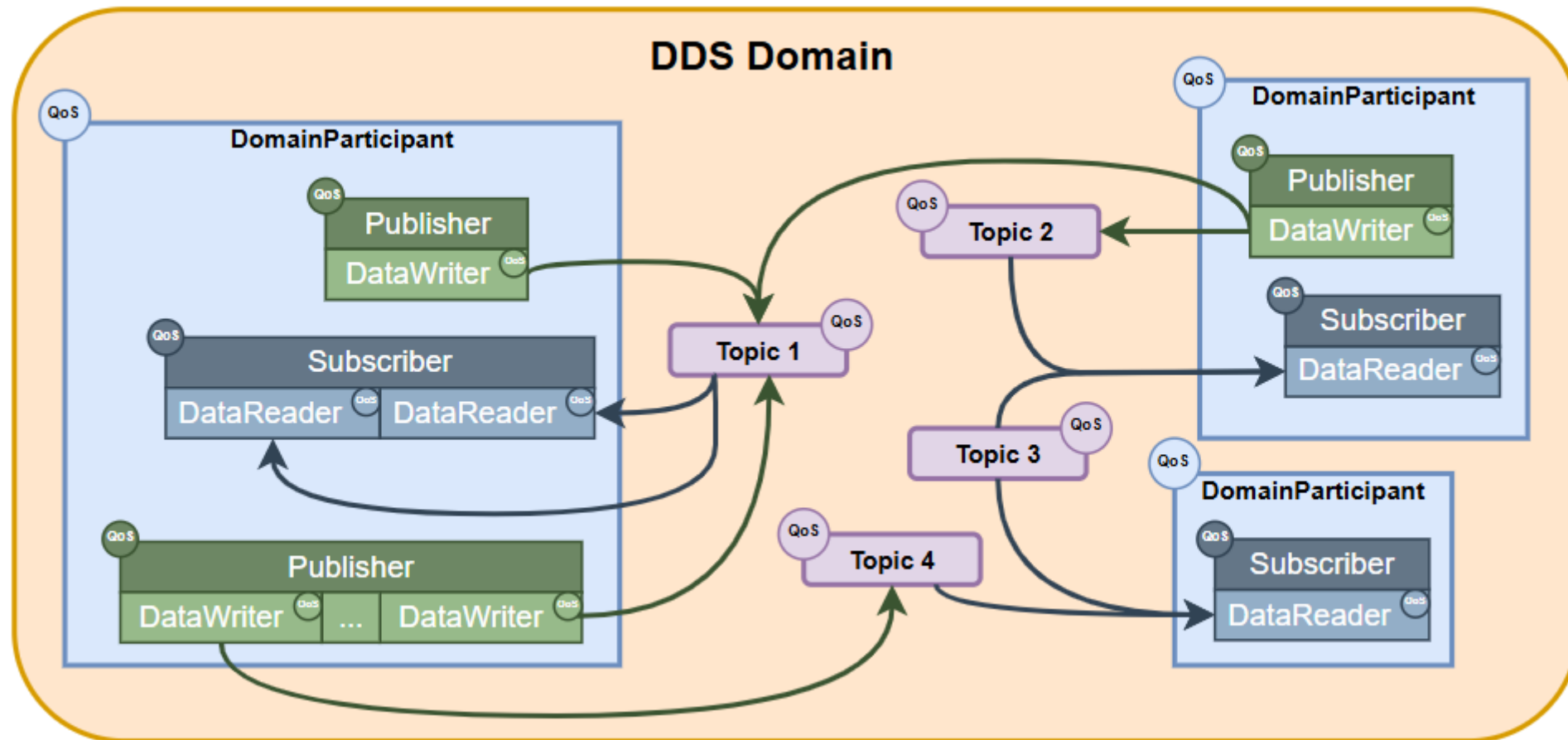
- Links all **Publishers** and **Subscribers** in a DDS domain.
- Identified by a unique **domain ID**:
 - Different IDs create independent communication channels.
 - Prevents interference between multiple applications.

Key DCPS Elements

- **DomainParticipant:**

- Defines the **domain ID** to specify the DDS domain it belongs to.
- Acts as a **container** for DCPS Entities (**Publisher**, **Subscriber**, and **Topic**).
- Serves as a **factory** for creating these entities.

DDS Model (DCPS)





Installing FastDDS (3.1.0)

- Download FastDDS Binaries

```
curl -o fastdds.tgz 'https://www.eprosima.com/component/ars/item/eProsimas_Fast-DDS-v3.1.0-Linux.tgz?format=tgz&category_id=7&release_id=169&Itemid=0'
```

- Unzip fastdds.tgz

```
mkdir fastdds  
tar -xvzf ./fastdds.tgz -C ./fastdds  
cd fastdds  
sudo ./install.sh
```

Linking DDS

Tasks.json → Adding gcc args

"-I/usr/include/fastdds" → FastDDS include directory

"-I/usr/include/fastcdr" → FastCDR include directory

"-std=c++11" → We want to use C++11

"-lstdc++" → link standard C++ library

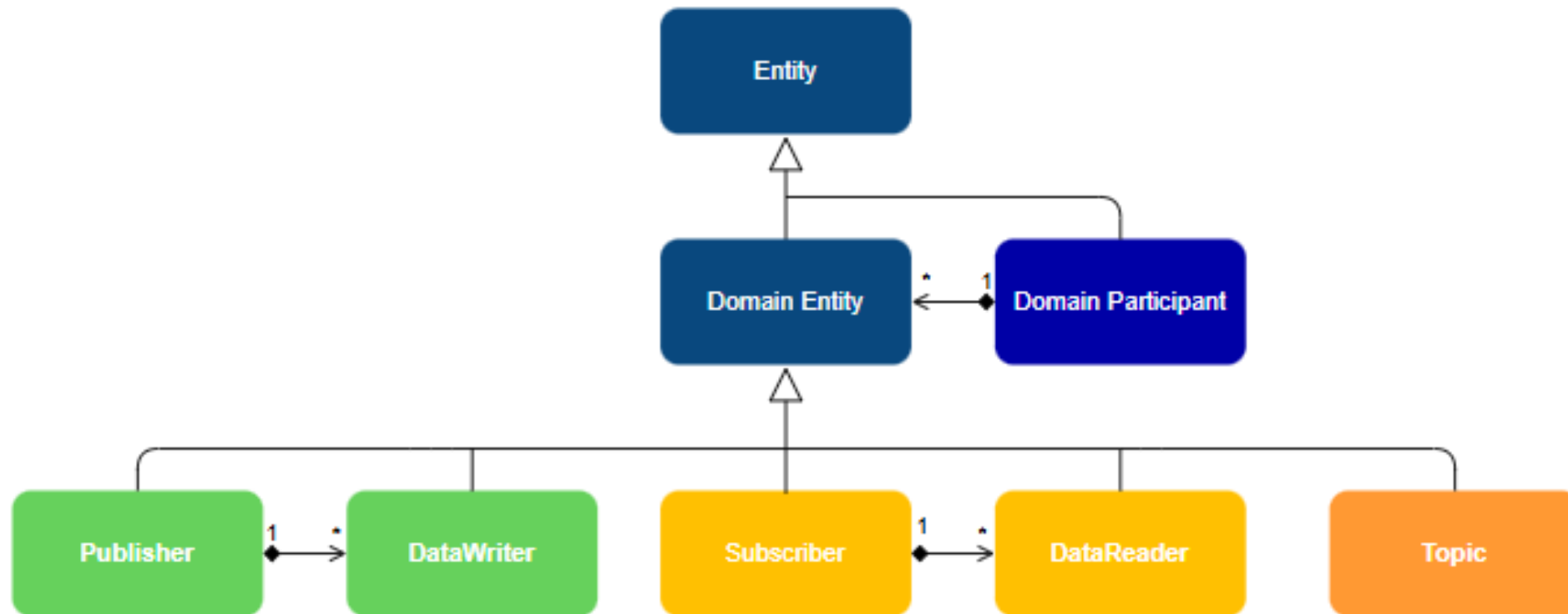
"-lfastcdr" → link libfastcdr (for serialization and deserialization)

"-libfastdds" → link libfastdds

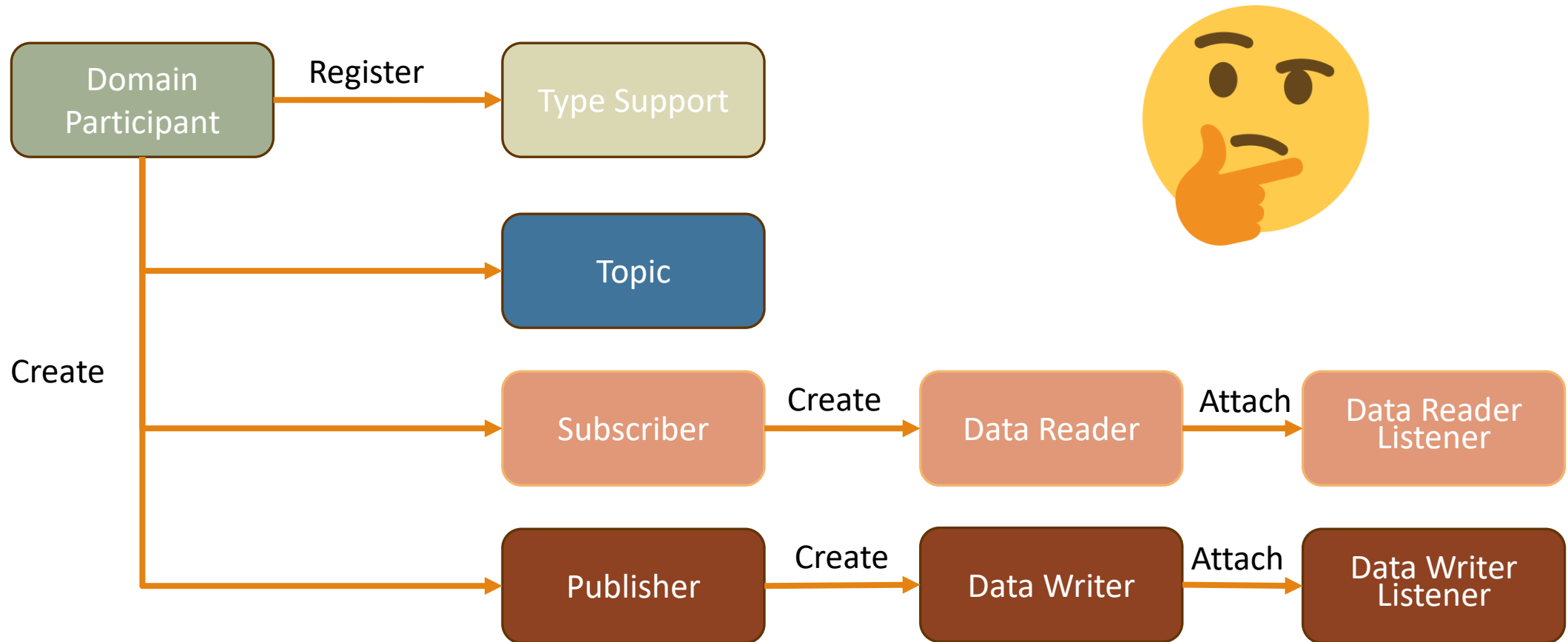


HelloWorld Example

Entities Relations



Entities Relations



Data Listeners

- **Data Reader Listener:** abstract class defining the callbacks that will be triggered in response to state changes on the **DataReader**.
 - **on_subscription_matched:** There is new data available for the application on the **DataReader**.
 - **on_data_available:** The **DataReader** has found a **DataWriter** that matches the **Topic**.
- **Data Write Listener:** abstract class defining the callbacks that will be triggered in response to state changes on the **DataWriter**.
 - **on_publication_matched:** **DataWriter** has found a **DataReader** that matches the **Topic**



Listeners Example

Type Support & IDL

- IDL: Interface Definition Language
- For Publisher and subscriber to be able to communicate they should agree on a message structure
- Message structure is defined using IDL
- To convert from IDL to Cpp we use **Fast DDS-Gen**.
- Fast DDS-Gen is not case sensitive (`string` message = `string` Message)

```
struct HelloWorld
{
    unsigned long index;
    string message;
};
```

IDL	C++11
char	<code>char</code>
octet	<code>uint8_t</code>
short	<code>int16_t</code>
unsigned short	<code>uint16_t</code>
long	<code>int32_t</code>
unsigned long	<code>uint32_t</code>
long long	<code>int64_t</code>
unsigned long long	<code>uint64_t</code>
float	<code>float</code>
double	<code>double</code>
long double	<code>long double</code>
boolean	<code>bool</code>
string	<code>std::string</code>

Primitive Types

Arrays(std::array)

IDL	C++11
char a[5]	<code>std::array<char,5> a</code>
octet a[5]	<code>std::array<uint8_t,5> a</code>
short a[5]	<code>std::array<int16_t,5> a</code>
unsigned short a[5]	<code>std::array<uint16_t,5> a</code>
long a[5]	<code>std::array<int32_t,5> a</code>
unsigned long a[5]	<code>std::array<uint32_t,5> a</code>
long long a[5]	<code>std::array<int64_t,5> a</code>
unsigned long long a[5]	<code>std::array<uint64_t,5> a</code>
float a[5]	<code>std::array<float,5> a</code>
double a[5]	<code>std::array<double,5> a</code>

Sequences (std::vector)

IDL	C++11
sequence<char>	<code>std::vector<char></code>
sequence<octet>	<code>std::vector<uint8_t></code>
sequence<short>	<code>std::vector<int16_t></code>
sequence<unsigned short>	<code>std::vector<uint16_t></code>
sequence<long>	<code>std::vector<int32_t></code>
sequence<unsigned long>	<code>std::vector<uint32_t></code>
sequence<long long>	<code>std::vector<int64_t></code>
sequence<unsigned long long>	<code>std::vector<uint64_t></code>
sequence<float>	<code>std::vector<float></code>
sequence<double>	<code>std::vector<double></code>

IDL to CPP Classes

```
fastddsgen ~/CustomIDL/src/MyMessage.idl -d ~/CustomIDL/src/Generated
```

```
struct MyMessage
{
    unsigned long index;
    double first_number;
    double second_number;
};
```

Generated

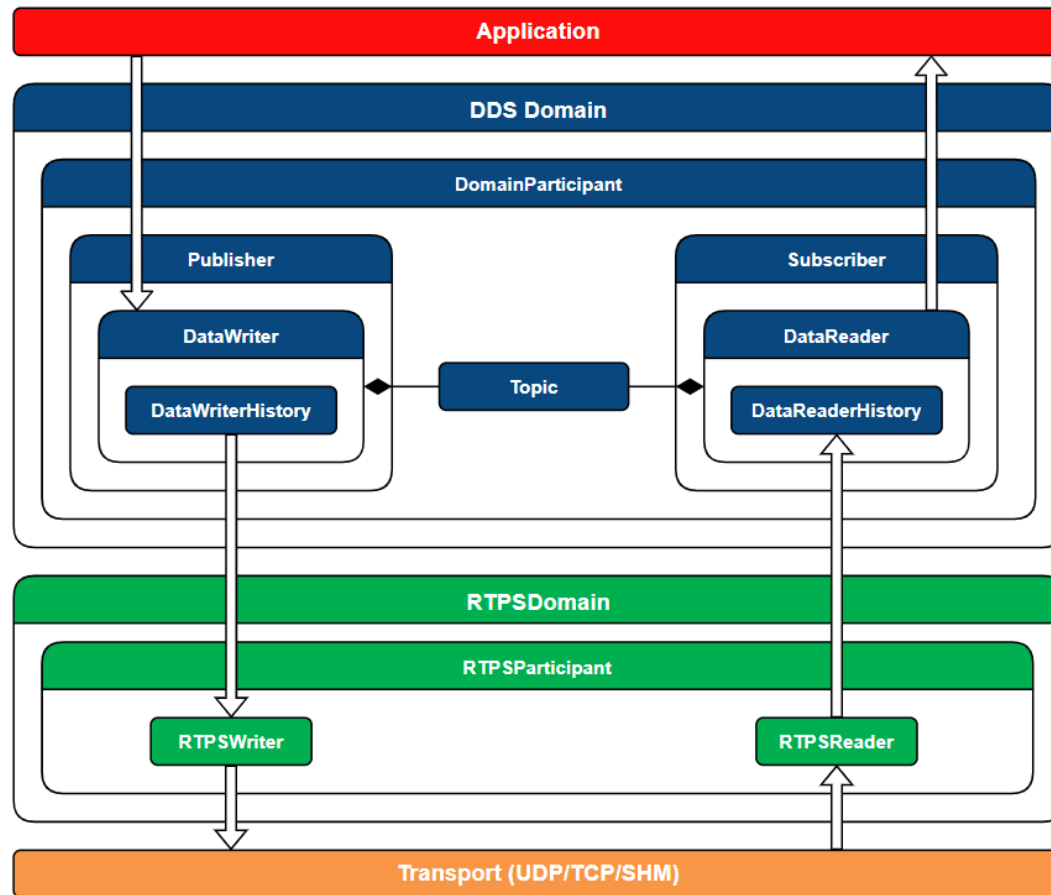
- MyMessage.hpp
- MyMessageCdrAux.hpp
- MyMessageCdrAux.ipp
- MyMessagePubSubTypes.cxx
- MyMessagePubSubTypes.hpp
- MyMessageTypeObjectSupport.cxx
- MyMessageTypeObjectSupport.hpp

Header File to Include



CustomIDL Example

DDS Layer Model



← This is our Application

← DDS Layer that we use to build our App

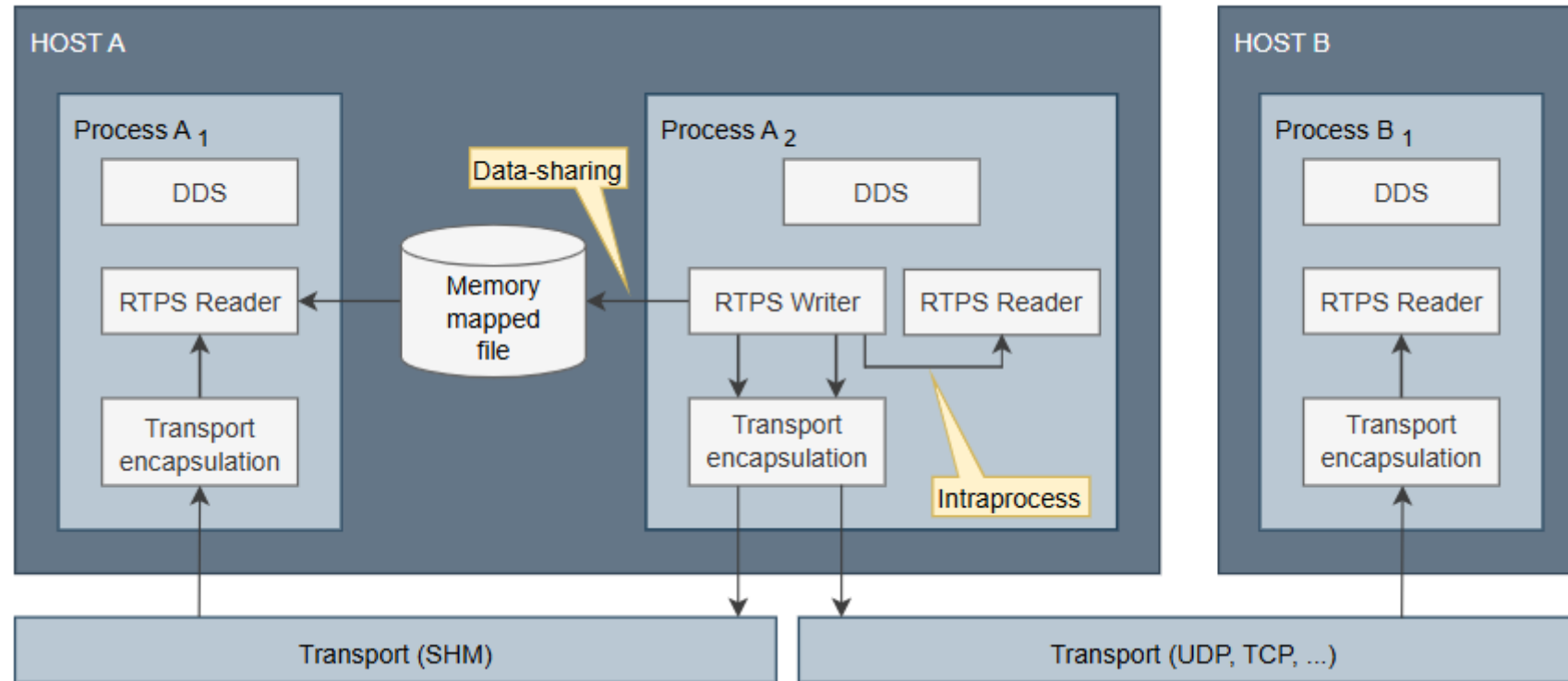
← Abstraction of DDS application entities from the transport layer

← Protocol that is going to be used by the App

Transport Layer

- **UDpv4**: UDP Datagram communication over IPv4. (Default Protocol)
- **UDpv6**: UDP Datagram communication over IPv6.
- **TCPv4**: TCP communication over IPv4.
- **TCPv6**: TCP communication over IPv6.
- **SHM**: Shared memory communication among entities running on the same host.

Transport Layer

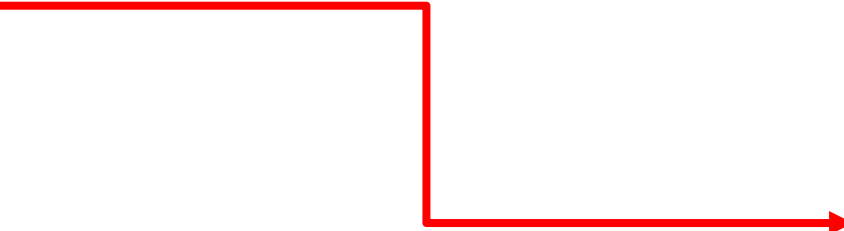




Transports Example


Selecting a Transport

- By default, UDPv4 is used.



```
Starting publisher.  
Publisher matched.  
Using UDPv4
```

```
// Explicit configuration of shm transport  
participantQos.transport().use_builtin_transports = false;  
auto shm_transport = std::make_shared<SharedMemTransportDescriptor>();  
shm_transport->segment_size(10 * 1024 * 1024);  
participantQos.transport().user_transports.push_back(shm_transport);
```



```
Starting subscriber.  
Subscriber matched.  
Using Shared Memory
```

Introduction to Discovery in Fast DDS

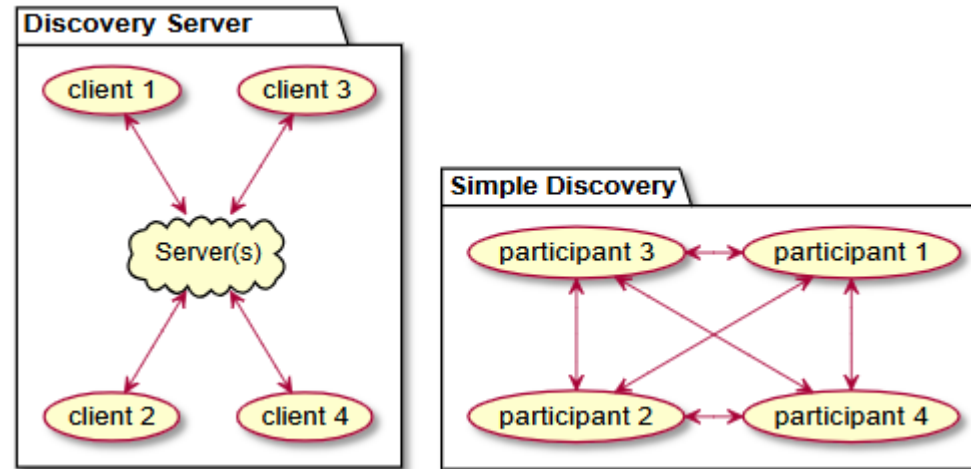
- A mechanism for **automatic identification** of participants (publishers and subscribers) in a distributed system.
- Why is Discovery Important?
 - **Dynamic Networks:** Allows participants to join and leave the system dynamically.
 - **Automatic Matching:** Identifies compatible publishers and subscribers based on Topics, Data Types, Quality of Service (QoS).

Discovery Phases in Fast DDS

- Participant Discovery Phase (PDP):
 - DomainParticipants acknowledge each other's existence.
 - **Matching Criteria:** Must be in the same DDS Domain
- Endpoint Discovery Phase (EDP):
 - DataWriters and DataReaders acknowledge each other.
 - **Matching Criteria:** Topic and data type must match.

Discovery mechanisms

- Simple Discovery: This is the default mechanism. It uses Multicast for discovery.
- Static Discovery: Requires preconfiguring participant information (e.g., IP addresses). Use an XML configuration file to define static participants.
- Discovery Server: A **Discovery Server** acts as a central point where participants registers and participants query the server to find and match endpoints.



Static Discovery

- An xml configuration is required
- Xml contains all the participants and their respective Protocols, Ips, and Ports
- Then this xml file should be loaded at the level of DomainParticipantQos

```
<profiles>
  <participant profile_name="static_discovery">
    <rtps>
      <builtin>
        <initialPeersList>
          <locator>
            <udp4> <!-- Participant protocol -->
              <address>172.17.0.3</address> <!-- Participant IPv4 -->
              <port>7412</port> <!-- Participant port -->
            </udp4>
          </locator>
        </initialPeersList>
      </builtin>
    </rtps>
  </participant>
</profiles>
```

```
DomainParticipantQos pqos;

pqos.wire_protocol().builtin.discovery_config.use_SIMPLE_EndpointDiscoveryProtocol = false;
pqos.wire_protocol().builtin.discovery_config.use_STATIC_EndpointDiscoveryProtocol = true;
pqos.wire_protocol().builtin.discovery_config.static_edp_xml_config("file://static_discovery.xml");
```

Discovery Server

- DomainParticipants may be *clients* or *servers*.
- A **SERVER** is a participant to which the *clients* (and maybe other *servers*) send their discovery information.
- A **CLIENT** is a participant that connects to one or more servers from which it receives only the discovery information they require to establish communication with matching endpoints.
- A **SUPER_CLIENT** is a client that receives the discovery information known by the server, in opposition to clients, which only receive the information they need.

```
DomainParticipantQos pqos;  
  
pqos.wire_protocol().builtin.discovery_config.discoveryProtocol = DiscoveryProtocol::CLIENT;  
pqos.wire_protocol().builtin.discovery_config.discoveryProtocol = DiscoveryProtocol::SUPER_CLIENT;  
pqos.wire_protocol().builtin.discovery_config.discoveryProtocol = DiscoveryProtocol::SERVER;
```

Discovery Server

```
DomainParticipantQos qos;  
qos.wire_protocol().builtin.discovery_config.discoveryProtocol = DiscoveryProtocol_t::CLIENT;  
qos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_back(eprosima::fastrtps::rtps::RemoteServerAttributes("192.168.1.1"));  
DomainParticipant* participant = DomainParticipantFactory::get_instance()->create_participant(0, qos);
```

Client

```
DomainParticipantQos qos;  
qos.builtin.discovery_config.discoveryProtocol = DiscoveryProtocol_t::SERVER;  
qos.builtin.discovery_config.m_DiscoveryServers.push_back(eprosima::fastrtps::rtps::Locator_t());  
DomainParticipant* participant = DomainParticipantFactory::get_instance()->create_participant(0, qos);
```

Server

Advantages of Each Approach

- **Simple Discovery Protocol:**

- Best for small networks with dynamic participants.
- No extra configuration needed.

- **Static Discovery:**

- Useful for stable, preconfigured networks.
- Reduces discovery time.

- **Client-Server Discovery:**

- Scales well in large, distributed systems.
- Minimizes multicast traffic for discovery.