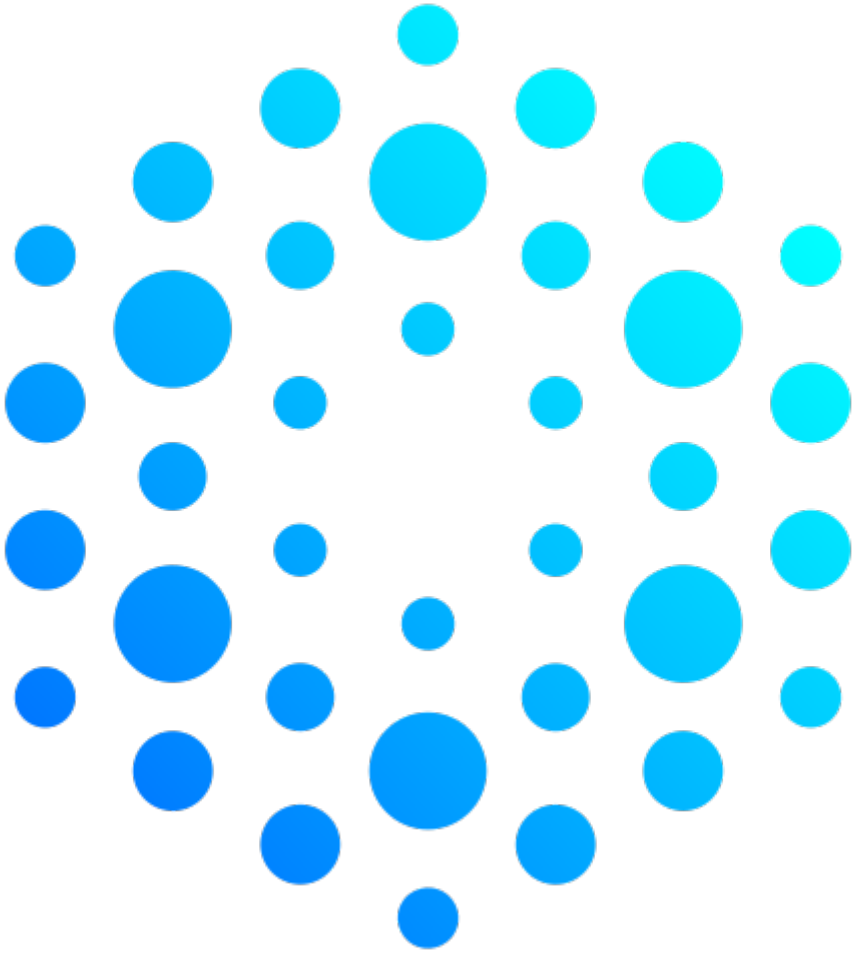


Astute DDS

User Guide

DDS for C++ — DDSI-RTPS 2.5 · DDS-XTypes 1.3 · v1.0.0-rc.1

Table of contents

1. Introduction

AstuteDDS is a commercial-grade C++20 middleware library for real-time, secure data distribution across distributed systems. It implements the full OMG Data Distribution Service (DDS) stack — DCPS API, DDSI-RTPS 2.5 wire protocol, DDS-XTypes 1.3, and optional DDS Security 1.1/1.2 — packaged as a single static library that links into your application with no runtime dependencies.

It is designed for defence, aerospace, robotics, and industrial applications where deterministic low-latency communication, standards compliance, and cross-vendor interoperability are requirements, not options.

1.1 Scope

This guide covers everything needed to integrate AstuteDDS into your application:

- Building and installing the library
- Core DDS concepts and Quality of Service policies
- Step-by-step guides for each major feature
- API overview with links to the generated reference documentation

1.2 What is DDS?

The **Data Distribution Service (DDS)** is an OMG standard middleware protocol and API for data-centric publish-subscribe communication. It enables real-time, scalable data exchange between distributed processes without requiring a central broker.

DDS is used in:

- **Defence and land vehicles** — the UK Def Stan 23-009 Generic Vehicle Architecture (GVA) standard mandates DDS for real-time data distribution inside military platforms.
- **Robotics** — ROS 2 uses DDS as its default middleware layer.
- **Aerospace and avionics** — air traffic control, flight management systems, and unmanned vehicles.
- **Industrial automation** — SCADA, factory automation, and smart grid.
- **Healthcare** — medical device integration and patient monitoring.

Key DDS properties:

- **Broker-less** — participants discover each other automatically via RTPS multicast; no message broker or server is needed.
- **QoS-driven** — 22 standardised policies control reliability, durability, deadlines, liveliness, ownership, and more.
- **Interoperable** — the DDSI-RTPS 2.5 wire protocol allows DDS implementations from different vendors to communicate transparently.

1.3 Capabilities

Category	Details
Wire Protocol	DDSI-RTPS 2.5 · SPDP / SEDP discovery · UDP unicast & multicast · TCP · Shared memory · DATA_FRAG fragmentation
Reliability	HEARTBEAT / ACKNACK · KEEP_LAST / KEEP_ALL history · Resource limits
QoS	All 22 OMG DDS QoS policies · DDS-XML profile loader
Serialisation	XCDR1 · XCDR2 (delimited & mutable)
IDL Compiler	IDL 4.2 → C++ · @key · @optional · @id · @autoid · @data_representation
X-Types	TypeObject · TypeIdentifier · Assignability · DynamicData · TypeLookup Service
DCPS API	All core entities · Content-Filtered Topics · Writer-side filtering · Persistence Service · Recording & Replay · ROS 2 bridge
Security	Authentication · Access Control · AES-GCM Crypto SPIs (DDS Security 1.1/1.2)
Tools	`astutedds-idl` IDL compiler · `astutedds-inspect` Qt6 diagnostics GUI · Qt6 Shapes Demo

1.4 Interoperability

AstuteDDS is tested for wire-level interoperability against:

- **eProsima Fast DDS**
- **Eclipse Cyclone DDS**
- **OpenDDS**
- **RTI Connext DDS**

Interoperability is validated using the OMG RTPS Test Suite and the standard DDS Shapes Demo application.

1.5 Specifications

AstuteDDS implements or references the following OMG specifications:

Specification	Version	Scope
DDS	1.4	DCPS API
DDSI-RTPS	2.5	Wire protocol
DDS-XTypes	1.3	Type system
IDL	4.2	Interface definition language
DDS Security	1.1 / 1.2	Authentication, access control, crypto
Def Stan 23-009	—	UK GVA profile

1.6 Document Structure

Chapter	Content
1. Introduction	This chapter — purpose, scope, feature summary
2. Getting Started	Building, installing, and running the Shapes Demo
3. DDS Concepts	Core DDS principles, QoS policies, topics and types
4. Guides	Step-by-step tutorials (QoS, XML profiles, persistence, recording)
5. API Overview	DCPS, RTPS, and X-Types API reference overview
6. About	Architecture, specifications, licence

2. Getting Started

2.1 Getting Started with AstuteDDS

Follow these steps to go from package installation to a running publisher/subscriber in under five minutes.

2.1.1 Quick Start

1. Install Packages for Your Platform

Start with the package installation guide for your OS:

- Required for code onboarding: Developer Package
- Recommended for discovery/visibility checks: Astute Inspect
- Recommended for visual DDS exploration: Shapes Demo

- Developer Packages

- Astute Inspect Installation
- Shapes Demo Guide

2. Verify Tooling and Environment

Required for the code onboarding path:

```
astutedds-idl --help
```

Optional checks (if installed):

```
astutedds-inspect -h
```

3. Build Your First Application

Follow First Application to generate code from IDL and build publisher/subscriber executables with CMake.

4. Explore Shapes Demo and Inspect

```
astutedds-shapes-demo
astutedds-inspect
```

Use two `astutedds-shapes-demo` instances to verify local pub/sub quickly, then use `astutedds-inspect` to confirm participants and topics are discovered.

2.1.2 Shapes Demo

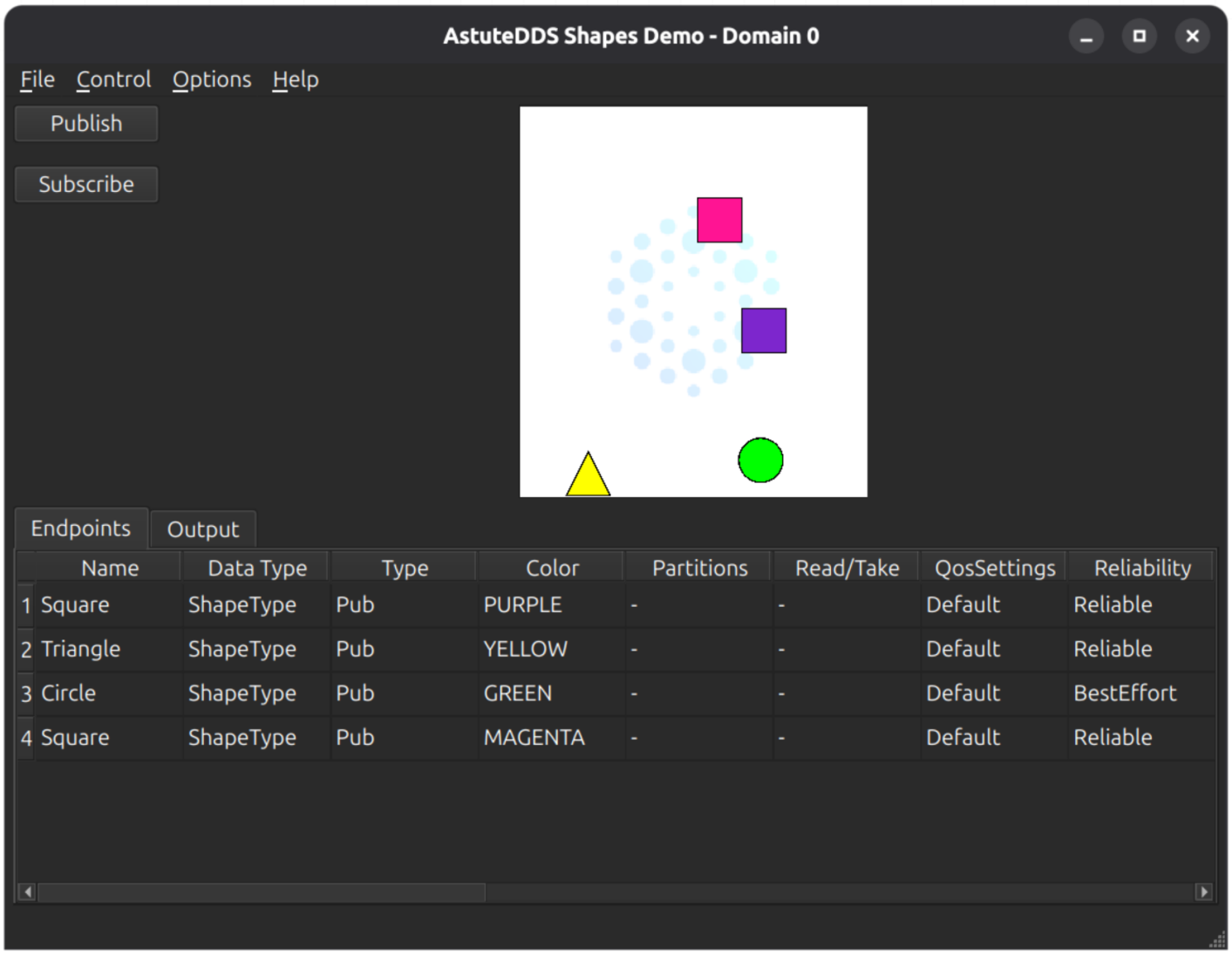
The Shapes Demo is the fastest way to build an intuition for DDS behavior before writing application code. It gives you a live GUI for publishers, subscribers, topics, and QoS settings so you can immediately see the effect of each change.

> **Note**
>
> You can run two Astute Shapes Demo instances back-to-back to test publish and receive behavior immediately. Both instances can run on the same machine, or on different machines on the same DDS domain/network.

What to Try First

- Switch reliability between `BEST_EFFORT` and `RELIABLE` and watch delivery behavior under load.
- Change durability from `VOLATILE` to `TRANSIENT_LOCAL` and then start a late subscriber.
- Adjust history depth and compare how much data is retained and delivered.
- Try different publish rates to observe traffic and responsiveness.

Shapes Demo GUI



For a full walkthrough, see the Shapes Demo Guide.

2.1.3 What's Next?

- Read the DDS Overview
- Try the Shapes Demo
- Learn about QoS Policies

- Secure your DDS system with DDS Security
- Bridge multiple domains with Domain Router
- Monitor your system with AstuteDDS Inspector
- Generate code with the IDL Compiler

2.2 Building from Source

Not required for most users

Customers receive pre-built packages (.deb, .rpm, .msi). See the Installation guide for the recommended setup path.

Building from source is intended for contributors and integrators who need to modify the library itself. Full build instructions, prerequisites, and CMake options are documented in the project README.md.

2.3 Installation

AstuteDDS is distributed as pre-built packages for Windows, Debian/Ubuntu, and RHEL-family Linux. Download the appropriate package for your platform from your Astute Systems licence portal or the GitHub Releases page.

Three package types are available:

Package	Contents
SDK	Static library, public headers, CMake config, `astutedds-idl` compiler, `astutedds-router`, `astutedds-discovery-dump`
Astute Inspect	Qt6 DDS diagnostics and topology GUI
Shapes Demo	Qt6 interoperability test application

2.3.1 Windows

SDK (.msi)

Double-click `AstuteDDS-SDK-<version>-win64.msi` or run silently from an elevated prompt:

```
msiexec /i AstuteDDS-SDK-<version>-win64.msi /quiet
```

The installer registers headers, the static library, CMake config, and command-line tools to standard Program Files locations and adds the tools to `PATH`.

Astute Inspect (.msi)

```
msiexec /i AstuteDDS-Inspect-<version>-win64.msi /quiet
```

Shapes Demo (.msi)

```
msiexec /i AstuteDDS-ShapesDemo-<version>-win64.msi /quiet
```

Verify

Open a new terminal (so `PATH` is refreshed):

```
astutedds-idl --help
astutedds-inspect --help
astute-shapes-demo --help
```

Uninstall

Use **Add or Remove Programs** in Windows Settings, or:

```
msiexec /x AstuteDDS-SDK-<version>-win64.msi /quiet
```

2.3.2 Debian / Ubuntu

SDK (.deb)

```
sudo dpkg -i astutedds-sdk_<version>_amd64.deb
sudo apt-get install -f          # resolve any missing dependencies
```

Astute Inspect (.deb)

```
sudo dpkg -i astutedds-inspect_<version>_amd64.deb
sudo apt-get install -f
```

Shapes Demo (.deb)

```
sudo dpkg -i astutedds-shapes-demo_<version>_amd64.deb
sudo apt-get install -f
```

Verify

```
astutedds-idl --help
astutedds-inspect --help
astute-shapes-demo --help
```

Uninstall

```
sudo apt-get remove astutedds-sdk astutedds-inspect astutedds-shapes-demo
```

2.3.3 RHEL / Rocky Linux / AlmaLinux

SDK (.rpm)

```
sudo dnf install ./astutedds-sdk-<version>-x86_64.rpm
```

Astute Inspect (.rpm)

```
sudo dnf install ./astutedds-inspect-<version>-x86_64.rpm
```

Shapes Demo (.rpm)

```
sudo dnf install ./astutedds-shapes-demo-<version>-x86_64.rpm
```

Verify

```

```

```
astutedds-idl --help
astutedds-inspect --help
astute-shapes-demo --help
```

Uninstall

```
sudo dnf remove astutedds-sdk astutedds-inspect astutedds-shapes-demo
```

2.3.4 What Gets Installed

After installing the SDK package, the following are available:

Path	Contents
`include/astutedds/`	Public C++ headers
`lib/libastutedds.a`	Static library
`lib/cmake/AstuteDDS/`	CMake `find_package` config
`bin/astutedds-idl`	IDL 4.2 → C++ compiler
`bin/astutedds-router`	DDS domain router daemon
`bin/astutedds-discovery-dump`	Discovery diagnostics tool

2.3.5 Using AstuteDDS in a CMake Project

Once the SDK is installed, add to your `CMakeLists.txt`:

```
find_package(AstuteDDS REQUIRED)
target_link_libraries(my_app PRIVATE AstuteDDS::astutedds)
```

See First Application for a complete worked example.

2.3.6 Python Bindings

See Python Bindings → Installation for the `pip install astutedds` workflow and the system `python3-astutedds` `.deb` / `.rpm` packages.

3. DDS Concepts

3.1 DDS Overview

The Data Distribution Service (DDS) is a middleware protocol and API standard for data-centric connectivity from the Object Management Group (OMG). It provides a publish-subscribe pattern for real-time, scalable, and high-performance data exchange.

3.1.1 What is DDS?

DDS enables applications to communicate by publishing and subscribing to data samples identified by topics. It provides:

- **Decoupling**: Publishers and subscribers don't need to know about each other
- **QoS Control**: Fine-grained control over reliability, durability, latency, and resource usage
- **Discovery**: Automatic discovery of participants, topics, and endpoints
- **Type Safety**: Strongly-typed data model with type propagation
- **Scalability**: From small embedded systems to large distributed systems

3.1.2 DDS Architecture

```
graph TB
    subgraph "Domain Participant"
        P1[Publisher]
        S1[Subscriber]
        T1[Topic: Temperature]
        T2[Topic: Pressure]

        P1 --> DW1["DataWriter
        Temperature"]
        P1 --> DW2["DataWriter
        Pressure"]
        S1 --> DR1["DataReader
        Temperature"]
        S1 --> DR2["DataReader
        Pressure"]

        DW1 -.publishes.-> T1
        DR1 -.subscribes.-> T1
        DW2 -.publishes.-> T2
        DR2 -.subscribes.-> T2
    end
```

3.1.3 Key Components

Domain Participant

The entry point for DDS applications. Represents a participant in a DDS domain (isolated communication space).

```
auto participant = astutedds::dcps::DomainParticipantFactory::create_participant(
    domain_id,
    PARTICIPANT_QOS_DEFAULT
);
```

Topic

Named data channel typed by a specific data structure. Topics are the foundation of DDS communication.

```
auto topic = participant->create_topic<SensorData>(
    "SensorTopic",
    TOPIC_QOS_DEFAULT
);
```

Publisher and DataWriter

Publishers create DataWriters that publish samples on topics.

```
auto publisher = participant->create_publisher();
auto writer = publisher->create_datawriter(topic, DATAWRITER_QOS_DEFAULT);

SensorData data;
data.temperature = 25.5;
writer->write(data);
```

Subscriber and DataReader

Subscribers create DataReaders that receive samples from topics.

```
auto subscriber = participant->create_subscriber();
auto reader = subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);

std::vector<SensorData> samples;
reader->take(samples);
```

3.1.4 Communication Flow

Simple Publish-Subscribe

```
sequenceDiagram
    participant Publisher
    participant DDS
    participant Subscriber

    Publisher->>DDS: Create DataWriter
    Subscriber->>DDS: Create DataReader

    Note over DDS: Discovery Protocol
    Note over DDS: (SPDP/SEDP)

    DDS-->>Publisher: Reader discovered
    DDS-->>Subscriber: Writer discovered

    Publisher->>DDS: write(sample)
    DDS->>Subscriber: DATA submessage
    Subscriber-->>DDS: ACKNACK (if RELIABLE)

    Note over Subscriber: Process sample
```

Discovery Process

DDS uses automatic discovery to match publishers and subscribers:

```
sequenceDiagram
    participant App1 as Application 1
    participant App2 as Application 2

    Note over App1,App2: SPDP: Simple Participant Discovery Protocol

    App1->>App2: SPDP: Participant announcement
    App2->>App1: SPDP: Participant announcement

    Note over App1,App2: SEDP: Simple Endpoint Discovery Protocol

    App1->>App2: SEDP: DataWriter metadata
    App2->>App1: SEDP: DataReader metadata

    Note over App1,App2: Endpoints matched!
    Note over App1,App2: Ready for data exchange
```

Reliable Data Exchange

When using RELIABLE reliability QoS:

```
sequenceDiagram
    participant Writer
    participant Reader

    Writer->>Reader: DATA (seq=1)
    Writer->>Reader: DATA (seq=2)
    Note over Reader: seq=2 lost
    Writer->>Reader: HEARTBEAT (seq=1-3)
    Reader->>Writer: ACKNACK (missing: 2)
    Writer->>Reader: DATA (seq=2) [repair]
    Reader->>Writer: ACKNACK (all received)

    Note over Writer,Reader: All data delivered
```

Best-Effort Data Exchange

When using BEST_EFFORT reliability QoS:

```
sequenceDiagram
    participant Writer
    participant Reader

    Writer->>Reader: DATA (seq=1)
    Writer->>Reader: DATA (seq=2)
    Note over Reader: seq=2 lost
    Writer->>Reader: DATA (seq=3)

    Note over Reader: Process seq=1, 3
    Note over Reader: Skip seq=2 (no repair)
```

3.1.5 Quality of Service (QoS)

QoS policies control the behavior of DDS entities. Key policies include:

Reliability

- **BEST_EFFORT**: Fast, no retransmission (good for real-time sensor data)
- **RELIABLE**: Guaranteed delivery with retransmission (good for commands)

Durability

- **VOLATILE**: No historical data (only live data)
- **TRANSIENT_LOCAL**: Late joiners get historical data
- **TRANSIENT**: Historical data survives process restart
- **PERSISTENT**: Historical data persists in database

History

- **KEEP_LAST(n)**: Keep only last n samples
- **KEEP_ALL**: Keep all samples (subject to resource limits)

Example QoS Configuration

```
auto qos = astutedds::dcps::DataWriterQosBuilder()
    .reliability(astutedds::dcps::ReliabilityQosPolicyKind::RELIABLE_RELIABILITY_QOS)
    .durability(astutedds::dcps::DurabilityQosPolicyKind::TRANSIENT_LOCAL_DURABILITY_QOS)
    .history(astutedds::dcps::HistoryQosPolicyKind::KEEP_LAST_HISTORY_QOS, 10)
    .build();

auto writer = publisher->create_datawriter(topic, qos);
```

3.1.6 RTPS Wire Protocol

DDS uses the RTPS (Real-Time Publish-Subscribe) protocol for network communication:

```
graph LR
    A[Application] --> B[DDS API]
    B --> C[RTPS Protocol]
    C --> D[UDP/IP Transport]
    D --> E[Network]
```

RTPS provides:

- **Discovery**: Automatic participant and endpoint discovery
- **Reliability**: HEARTBEAT/ACKNACK protocol for guaranteed delivery
- **Efficiency**: Compact binary encoding (CDR/XCDR)
- **Interoperability**: Standard wire format for cross-vendor communication

3.1.7 Type System (X-Types)

AstuteDDS supports DDS X-Types 1.3 for dynamic type handling:

```
graph TB
    IDL[IDL Definition] --> Compiler[IDL Compiler]
    Compiler --> TypeObject[TypeObject]
    Compiler --> CppCode[C++ Code]

    TypeObject --> TypeRegistry[Type Registry]
    CppCode --> App[Application]
```

```
    TypeRegistry --> DynamicData[Dynamic Data]
    App --> DynamicData

    TypeRegistry -.type discovery.-> RemoteApp[Remote Application]
```

Features:

- **TypeObject**: Runtime type representation
- **Type Discovery**: Automatic type propagation
- **Assignability**: Type compatibility checking
- **Dynamic Data**: Runtime data manipulation without code generation

3.1.8 Data Representation

DDS uses CDR (Common Data Representation) for serialization:

- **XCDR1**: Traditional CDR with alignment rules
- **XCDR2**: Enhanced CDR with delimiters and optional members

```
graph LR
    A[Struct Data] --> B{Encoding}
    B -->|XCDR1| C[Aligned Binary]
    B -->|XCDR2| D[Delimited Binary]

    C --> E[Network Transmission]
    D --> E

    E --> F{Decoding}
    F --> G[Struct Data]
```

3.1.9 Use Cases

DDS is used in:

- **Military & Defense**: Command and control, sensor fusion
- **Aerospace**: Flight control, avionics
- **Industrial IoT**: Factory automation, SCADA
- **Automotive**: ADAS, autonomous vehicles
- **Medical**: Real-time patient monitoring
- **Financial**: High-frequency trading

3.1.10 Next Steps

- Learn about QoS Policies in detail
- Explore Topics and Data Types
- Try the Shapes Demo
- Build your first Publisher/Subscriber

3.2 Quality of Service (QoS)

Quality of Service (QoS) policies control the behavior of DDS entities.

For a complete guide on using QoS policies, see the QoS Usage Guide.

3.2.1 Key QoS Policies

- **Reliability**: BEST_EFFORT vs RELIABLE
- **Durability**: VOLATILE, TRANSIENT_LOCAL, TRANSIENT, PERSISTENT
- **History**: KEEP_LAST vs KEEP_ALL
- **Deadline**: Maximum time between samples
- **Liveliness**: Detecting inactive writers
- **Ownership**: SHARED vs EXCLUSIVE

3.2.2 Example

```
auto qos = astutedds::dcps::DataWriterQosBuilder()
    .reliability(astutedds::dcps::ReliabilityQosPolicyKind::RELIABLE_RELIABILITY_QOS)
    .durability(astutedds::dcps::DurabilityQosPolicyKind::TRANSIENT_LOCAL_DURABILITY_QOS)
    .history(astutedds::dcps::HistoryQosPolicyKind::KEEP_LAST_HISTORY_QOS, 10)
    .build();
```

3.3 Topics and Data Types

3.3.1 Topics

Topics are named data channels in DDS. Publishers write to topics, and subscribers read from them.

```
auto topic = participant->create_topic<MyDataType>(
    "MyTopic",
    TOPIC_QOS_DEFAULT
);
```

3.3.2 Data Types

Data types are defined using IDL and compiled to C++:

```
struct MyDataType {
    @key long id;
    string message;
    float value;
};
```

See the IDL Compiler Guide for details.

4. Guides

4.1 Shapes Demo Guide

The Astute Shapes Demo is a Qt6-based graphical application for testing DDS interoperability. It's the standard tool used by DDS vendors to demonstrate cross-vendor compatibility.

4.1.1 What is the Shapes Demo?

The Shapes Demo publishes and subscribes to geometric shapes (circles, squares, triangles) with different colors on a shared canvas. Each shape's position is updated in real-time over DDS, allowing multiple applications from different vendors to see the same shapes moving.

4.1.2 Install the Astute Shapes Demo

Use the package format for your operating system.

Windows (MSI)

```
msiexec /i astutedds-shapes-demo-<version>-windows-x64.msi
astutedds-shapes-demo.exe
```

RHEL / Rocky / AlmaLinux (RPM)

```
sudo dnf install ./astutedds-shapes-demo-<version>-x86_64.rpm
astute-shapes-demo
```

Debian / Ubuntu (DEB)

```
sudo dpkg -i astutedds-shapes-demo_<version>_amd64.deb
sudo apt-get install -f
astute-shapes-demo
```

4.1.3 Running the Shapes Demo

Starting the Application

```
astute-shapes-demo
```

The application window displays:

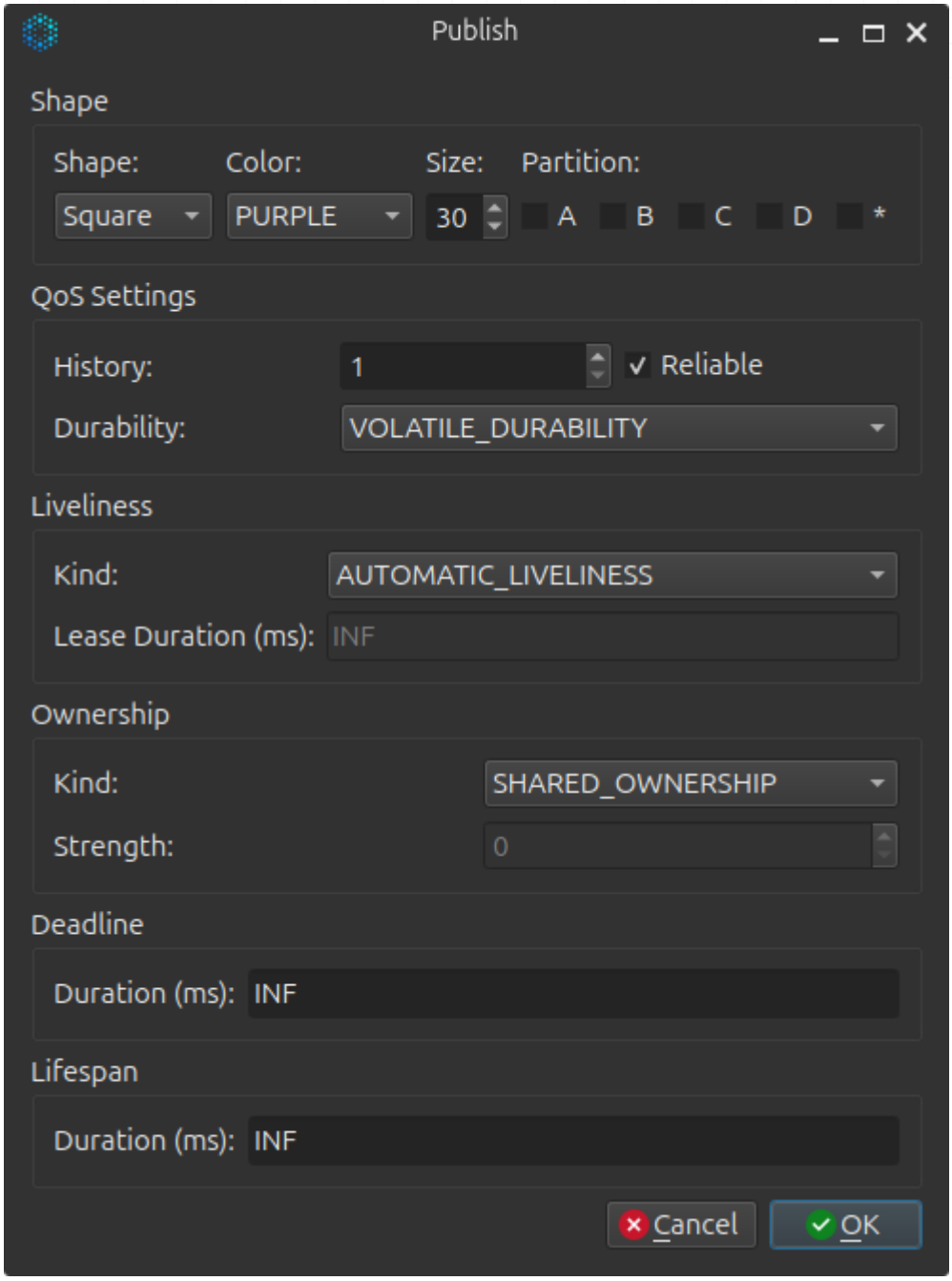
- **Canvas**: 240x270 coordinate space showing shapes
- **Shape Controls**: Select shape type (Circle, Square, Triangle)
- **Color Selector**: Choose from standard colors (RED, GREEN, BLUE, etc.)
- **Publish Button**: Start publishing a shape
- **Subscribe Button**: Subscribe to shapes from other applications
- **QoS Settings**: Configure Quality of Service policies

4.1.4 Publishing Shapes

To publish a shape:

1. **Select Shape Type**: Choose Circle, Square, or Triangle
2. **Choose Color**: Select a color (e.g., BLUE)
3. **Set Size**: Adjust shape size (10-100 pixels)
4. **Enable Auto-move**: Check to animate the shape
5. **Click Publish**: Shape appears and starts moving

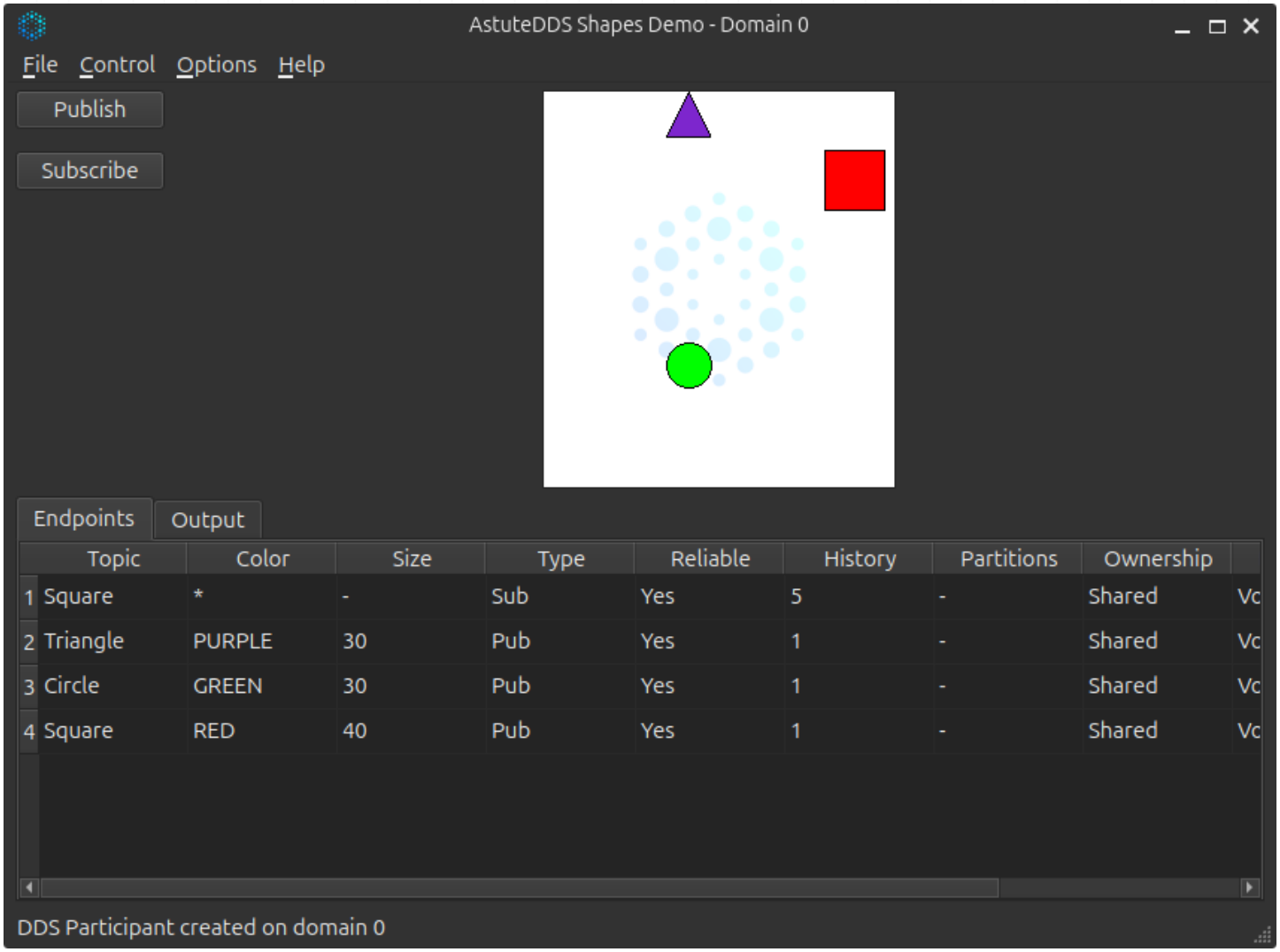
Publish Options Window



The publish options window allows you to configure:

- **Shape type**: Circle, Square, or Triangle
- **Color**: Select from standard DDS colors
- **Size**: Shape size in pixels (10-100)
- **Auto-move**: Enable automatic shape animation
- **QoS Settings**: Reliability, Durability, History depth, Ownership

Publishing Live Window



Once published, you'll see:

- Your shape moving on the canvas
- Real-time position updates
- Active publishers listed in the status area

Publishing Example Workflow

```
sequenceDiagram
    participant User
    participant ShapesDemo
    participant DDS
```

```
User->>ShapesDemo: Select "Circle", "BLUE"
User->>ShapesDemo: Click "Publish"
ShapesDemo->>DDS: Create DataWriter(Circle)
ShapesDemo->>DDS: write(x, y, color, size)

loop Every 33ms
    ShapesDemo->>ShapesDemo: Update position
    ShapesDemo->>DDS: write(new x, new y, ...)
end
```

Standard Colors

The Shapes Demo uses standard DDS colors:

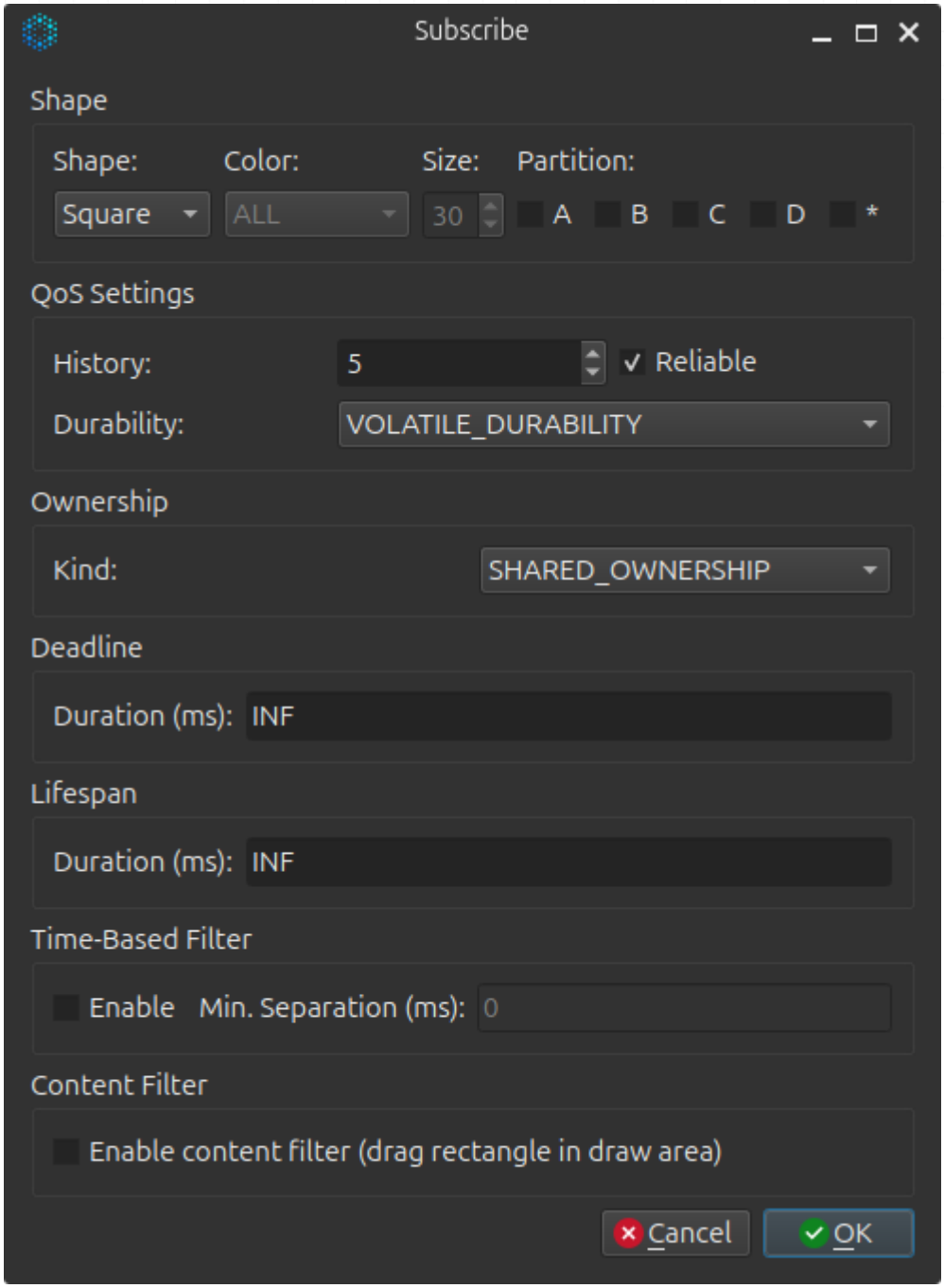
- RED
- GREEN
- BLUE
- YELLOW
- CYAN
- MAGENTA
- ORANGE
- PURPLE

4.1.5 Subscribing to Shapes

To subscribe to shapes:

1. **Select Color**: Choose which color to subscribe to (e.g., BLUE)
2. **Click Subscribe**: Create a DataReader for that color
3. **View Shapes**: Shapes published by any DDS application appear on the canvas

Subscribe Options Window



The subscribe options window allows you to configure:

- **Color**: Select which color to subscribe to
- **QoS Settings**: Match publisher's QoS for successful communication
- **History**: Control how many historical samples to receive

Subscription Example Workflow

```
sequenceDiagram
    participant User
    participant ShapesDemo
    participant DDS
    participant RemoteApp
```

```
    User->>ShapesDemo: Select "BLUE"
    User->>ShapesDemo: Click "Subscribe"
    ShapesDemo->>DDS: Create DataReader(Circle)

    RemoteApp->>DDS: write(Circle, BLUE, x, y)
    DDS->>ShapesDemo: DATA submessage
    ShapesDemo->>ShapesDemo: Draw shape at (x, y)

    loop Continuous updates
        RemoteApp->>DDS: write(new position)
        DDS->>ShapesDemo: DATA submessage
        ShapesDemo->>ShapesDemo: Update shape position
    end
```

4.1.6 Standard Topics

The Shapes Demo uses three standard DDS topics:

Topic Name	Data Type	Description
`Circle`	`ShapeType`	Circular shapes
`Square`	`ShapeType`	Square shapes
`Triangle`	`ShapeType`	Triangular shapes

ShapeType IDL Definition

All topics use the same data type:

```
module ShapesDemo {
    @mutable
    struct ShapeType {
        @key string<128> color;
        long x;
        long y;
        long shapesize;
    };
};
```

Generated C++ structure:

```
namespace ShapesDemo {
    struct ShapeType {
        std::string color;  // Key field
        int32_t x;
        int32_t y;
        int32_t shapesize;
    };
}
```

4.1.7 Coordinate System

The Shapes Demo uses a standard coordinate system:

- **X axis**: 0 to 240 (left to right)
- **Y axis**: 0 to 270 (top to bottom)
- **Origin**: Top-left corner
- **Bounds**: Shapes bounce off edges

```
(0,0) ----------------------- (240,0)
  |                             |
  |        Canvas Area          |
  |                             |
  |                             |
(0,270) --------------------- (240,270)
```

4.1.8 Testing Interoperability

The Astute Shapes Demo is compatible with shapes demos from other DDS vendors:

- eProsima Fast DDS Shapes Demo
- RTI Shapes Demo
- OpenDDS Shapes Demo
- CycloneDDS Shapes Demo
- S2E Systems Shapes Demo

Interoperability Test Steps

1. **Start Astute Shapes Demo**:

```
astute-shapes-demo
```

1. **Start another vendor's Shapes Demo** (e.g., eProsima Fast DDS):

```

```

```
./ShapesDemo  # from Fast DDS installation
```

1. **Publish from Astute Shapes Demo**:
2. Select "Circle" and "BLUE"
3. Click "Publish"
4. Enable "Auto-move"

5. **Subscribe from other vendor**:

6. Select "BLUE"
7. Click "Subscribe"
8. The blue circle should appear and move

9. **Verify bidirectional communication**:

10. Publish from the other vendor's demo
11. Subscribe in AstuteDDS
12. Shapes should appear in both applications

Expected Behavior

```
sequenceDiagram
    participant AstuteDDS
    participant Network
    participant FastDDS

    Note over AstuteDDS,FastDDS: Discovery Phase
    AstuteDDS->>Network: SPDP announcement
    FastDDS->>Network: SPDP announcement
    Network->>AstuteDDS: FastDDS participant discovered
    Network->>FastDDS: AstuteDDS participant discovered

    Note over AstuteDDS,FastDDS: Data Exchange
    AstuteDDS->>Network: DATA(Circle, BLUE, x, y)
    Network->>FastDDS: Receive DATA
    FastDDS->>FastDDS: Draw circle

    FastDDS->>Network: DATA(Square, RED, x, y)
    Network->>AstuteDDS: Receive DATA
    AstuteDDS->>AstuteDDS: Draw square
```

4.1.9 Quality of Service (QoS) Configuration

The Shapes Demo supports various QoS configurations:

Default QoS Settings

```
// Default configuration used by Shapes Demo
auto qos = astutedds::dcps::DataWriterQosBuilder()
    .reliability(astutedds::dcps::ReliabilityQosPolicyKind::RELIABLE_RELIABILITY_QOS)
    .durability(astutedds::dcps::DurabilityQosPolicyKind::VOLATILE_DURABILITY_QOS)
    .history(astutedds::dcps::HistoryQosPolicyKind::KEEP_LAST_HISTORY_QOS, 10)
    .build();
```

Reliability Options

RELIABLE (default):

- Guarantees delivery of all samples
- Uses HEARTBEAT/ACKNACK protocol
- Good for shape position updates

BEST_EFFORT:

- No delivery guarantees
- Lower latency
- Suitable for high-frequency sensor data

Durability Options

VOLATILE (default):

- No historical data
- Late joiners only see new updates

TRANSIENT_LOCAL:

- Late joiners receive last sample per instance
- Useful for state synchronization

History Settings

KEEP_LAST(10) (default):

- Maintains last 10 samples per shape
- Prevents unbounded memory growth

KEEP_ALL:

- Keeps all samples (up to resource limits)
- Higher memory usage

4.1.10 Common Use Cases

Use Case 1: Basic Connectivity Test

Goal: Verify DDS connectivity between two applications.

1. Start Astute Shapes Demo
2. Publish a BLUE Circle
3. Subscribe to BLUE in the same application
4. You should see the circle (loopback test)

Use Case 2: Cross-Vendor Interoperability

Goal: Test interoperability with another DDS implementation.

1. Start Astute Shapes Demo
2. Start Fast DDS Shapes Demo
3. Publish RED Square in AstuteDDS
4. Subscribe to RED in Fast DDS
5. Verify shape appears in both applications

Use Case 3: QoS Compatibility Testing

Goal: Test QoS policy matching.

1. Configure RELIABLE in AstuteDDS
2. Configure BEST_EFFORT in Fast DDS
3. Attempt to communicate
4. Observe that endpoints don't match (incompatible QoS)

Use Case 4: Multiple Instances

Goal: Test multiple instances with different keys.

1. Publish BLUE Circle
2. Publish GREEN Circle
3. Publish RED Circle
4. All circles should appear simultaneously
5. Each is a separate instance (different key = color)

4.1.11 Troubleshooting

Shapes Not Appearing

Problem: Published shapes don't appear in subscribing application.

Solutions:

1. Check network connectivity (ping test)
2. Verify same DDS domain ID (default: 0)
3. Check firewall settings (allow UDP multicast)
4. Ensure QoS policies are compatible
5. Verify topic names match exactly

Discovery Issues

Problem: Applications don't discover each other.

Solutions:

1. Check multicast support on network
2. Verify UDP ports are open (7400, 7401, etc.)
3. Check participant announcements in logs
4. Try running on localhost first

Performance Problems

Problem: Shapes are choppy or slow.

Solutions:

1. Reduce publish rate
2. Use BEST_EFFORT reliability for smoother motion
3. Increase history depth
4. Check CPU and network utilization

4.1.12 Advanced Topics

Custom Shape Behaviors

Modify the shapes demo source to:

- Add custom shape types
- Implement collision detection
- Create pattern generators
- Add physics simulation

Programmatic Control

Create automated test scripts:

```
// Example: Automated shapes test
#include "ShapeType_TypeSupport.hpp"

int main() {
    auto participant = create_participant(0);
    auto topic = participant->create_topic<ShapesDemo::ShapeType>("Circle");
    auto writer = create_datawriter(participant, topic);

    ShapesDemo::ShapeType shape;
    shape.color = "BLUE";
    shape.shapesize = 30;

    // Publish shapes in a pattern
    for (int i = 0; i < 100; ++i) {
        shape.x = 120 + 100 * std::cos(i * 0.1);
        shape.y = 135 + 100 * std::sin(i * 0.1);
        writer->write(shape);
        std::this_thread::sleep_for(std::chrono::milliseconds(33));
    }

    return 0;
}
```

4.1.13 Next Steps

- Learn about QoS Policies in detail
- Explore the IDL Compiler to create custom types
- Build a custom publisher/subscriber
- Review QoS examples

4.1.14 Resources

- Shapes Demo Specification
- Vendor Shapes Demo Links

4.2 Using QoS Policies

Quality of Service (QoS) policies control the behavior of DDS entities. This guide explains how to use and configure QoS policies in AstuteDDS.

4.2.1 What are QoS Policies?

QoS policies define non-functional properties like:

- **Reliability**: Guaranteed vs. best-effort delivery
- **Durability**: Whether late joiners get historical data
- **History**: How many samples to keep
- **Deadline**: Maximum time between samples
- **Liveliness**: How to detect inactive writers
- **Ownership**: Single vs. multiple writers

4.2.2 QoS Policy Categories

Data Delivery Policies

Control how data is delivered from writers to readers.

Reliability

Determines delivery guarantees:

```
#include <astutedds/dcps/qos.hpp>

// RELIABLE: Guaranteed delivery with retransmission
auto reliable_qos = astutedds::dcps::DataWriterQosBuilder()
    .reliability(astutedds::dcps::ReliabilityQosPolicyKind::RELIABLE_RELIABILITY_QOS)
    .build();

// BEST_EFFORT: No delivery guarantees, lower latency
auto besteffort_qos = astutedds::dcps::DataWriterQosBuilder()
    .reliability(astutedds::dcps::ReliabilityQosPolicyKind::BEST_EFFORT_RELIABILITY_QOS)
    .build();
```

When to use RELIABLE: - Commands and control messages - Financial transactions - Configuration updates - Any data that cannot be lost

When to use BEST_EFFORT: - High-frequency sensor data - Video/audio streams - Real-time telemetry - Data where latest value matters most

Durability

Controls late joiner behavior:

```
// VOLATILE: No historical data
auto volatile_qos = astutedds::dcps::DataWriterQosBuilder()
    .durability(astutedds::dcps::DurabilityQosPolicyKind::VOLATILE_DURABILITY_QOS)
    .build();
```

```
// TRANSIENT_LOCAL: Late joiners get last samples
auto transient_qos = astutedds::dcps::DataWriterQosBuilder()
    .durability(astutedds::dcps::DurabilityQosPolicyKind::TRANSIENT_LOCAL_DURABILITY_QOS)
    .build();

// TRANSIENT: Survives process restart
auto transient_persistent_qos = astutedds::dcps::DataWriterQosBuilder()
    .durability(astutedds::dcps::DurabilityQosPolicyKind::TRANSIENT_DURABILITY_QOS)
    .build();

// PERSISTENT: Stored in database
auto persistent_qos = astutedds::dcps::DataWriterQosBuilder()
    .durability(astutedds::dcps::DurabilityQosPolicyKind::PERSISTENT_DURABILITY_QOS)
    .build();
```

Durability levels:

Level	Late Joiners	Process Restart	Database
VOLATILE	❌ No data	❌ Lost	❌ No
TRANSIENT_LOCAL	✅ Last samples	❌ Lost	❌ No
TRANSIENT	✅ Last samples	✅ Survives	⚠️ Filesystem
PERSISTENT	✅ All samples	✅ Survives	✅ Database

History

Controls sample retention:

```
// KEEP_LAST: Keep only last N samples per instance
auto keep_last_qos = astutedds::dcps::DataWriterQosBuilder()
    .history(astutedds::dcps::HistoryQosPolicyKind::KEEP_LAST_HISTORY_QOS, 10)
    .build();

// KEEP_ALL: Keep all samples (up to resource limits)
auto keep_all_qos = astutedds::dcps::DataWriterQosBuilder()
    .history(astutedds::dcps::HistoryQosPolicyKind::KEEP_ALL_HISTORY_QOS)
    .build();
```

Trade-offs: - KEEP_LAST(n): Bounded memory, may lose old samples - KEEP_ALL: Unbounded memory (constrained by resource limits)

Data Timing Policies

Control temporal aspects of data delivery.

Deadline

Maximum time between sample updates:

```
using namespace std::chrono_literals;

// Expect updates at least every 1 second
```

```
auto deadline_qos = astutedds::dcps::DataWriterQosBuilder()
    .deadline(1s)
    .build();

// Reader expects same deadline
auto reader_qos = astutedds::dcps::DataReaderQosBuilder()
    .deadline(1s)
    .build();
```

Use cases: - Periodic sensor data (must arrive every X ms) - Heartbeat monitoring - Time-critical state updates

Callback on missed deadline:

```
reader->set_listener(
    [](const astutedds::dcps::RequestedDeadlineMissedStatus& status) {
        std::cout << "Deadline missed! " << status.total_count << " times\n";
    }
);
```

Liveliness

Detects inactive writers:

```
// AUTOMATIC: DDS manages liveliness
auto auto_liveliness = astutedds::dcps::DataWriterQosBuilder()
    .liveliness(
        astutedds::dcps::LivelinessQosPolicyKind::AUTOMATIC_LIVELINESS_QOS,
        3s  // lease duration
    )
    .build();

// MANUAL_BY_PARTICIPANT: Application asserts participant liveliness
auto manual_participant = astutedds::dcps::DataWriterQosBuilder()
    .liveliness(
        astutedds::dcps::LivelinessQosPolicyKind::MANUAL_BY_PARTICIPANT_LIVELINESS_QOS,
        5s
    )
    .build();

// MANUAL_BY_TOPIC: Application asserts per-writer liveliness
auto manual_topic = astutedds::dcps::DataWriterQosBuilder()
    .liveliness(
        astutedds::dcps::LivelinessQosPolicyKind::MANUAL_BY_TOPIC_LIVELINESS_QOS,
        5s
    )
    .build();
```

Asserting liveliness manually:

```
// For MANUAL_BY_PARTICIPANT
participant->assert_liveliness();

// For MANUAL_BY_TOPIC
writer->assert_liveliness();
```

Data Availability Policies

Ownership

Controls multiple writers to the same instance:

```
// SHARED: All writers can update (default)
auto shared_ownership = astutedds::dcps::DataWriterQosBuilder()
    .ownership(astutedds::dcps::OwnershipQosPolicyKind::SHARED_OWNERSHIP_QOS)
    .build();

// EXCLUSIVE: Highest strength writer wins
auto exclusive_ownership = astutedds::dcps::DataWriterQosBuilder()
    .ownership(astutedds::dcps::OwnershipQosPolicyKind::EXCLUSIVE_OWNERSHIP_QOS)
    .ownership_strength(100)  // Higher = higher priority
    .build();
```

EXCLUSIVE ownership example:

```
// Primary writer (high strength)
auto primary_writer_qos = astutedds::dcps::DataWriterQosBuilder()
    .ownership(astutedds::dcps::OwnershipQosPolicyKind::EXCLUSIVE_OWNERSHIP_QOS)
    .ownership_strength(200)
    .build();

// Backup writer (low strength)
auto backup_writer_qos = astutedds::dcps::DataWriterQosBuilder()
    .ownership(astutedds::dcps::OwnershipQosPolicyKind::EXCLUSIVE_OWNERSHIP_QOS)
    .ownership_strength(100)
    .build();

// Reader only sees samples from primary (strength 200)
// If primary fails, reader switches to backup
```

Resource Management Policies

Resource Limits

Control memory usage:

```
auto resource_limits = astutedds::dcps::DataWriterQosBuilder()
    .resource_limits(
        1000,   // max_samples
        100,    // max_instances
        10      // max_samples_per_instance
    )
    .build();
```

Parameters: - `max_samples`: Total samples across all instances - `max_instances`: Maximum number of different keys - `max_samples_per_instance`: Samples per key

User Data Policies

Attach metadata to entities:

```
```

```
// Attach user data to participant
std::vector<uint8_t> participant_data = {'A', 'p', 'p', '1'};
auto participant_qos = astutedds::dcps::DomainParticipantQosBuilder()
    .user_data(participant_data)
    .build();

// Attach group data to publisher
std::vector<uint8_t> group_data = {'S', 'e', 'n', 's', 'o', 'r', 's'};
auto publisher_qos = astutedds::dcps::PublisherQosBuilder()
    .group_data(group_data)
    .build();

// Attach topic data
std::vector<uint8_t> topic_data = {'v', '1', '.', '0'};
auto topic_qos = astutedds::dcps::TopicQosBuilder()
    .topic_data(topic_data)
    .build();
```

4.2.3 QoS Policy Compatibility

For communication to occur, QoS policies must be compatible:

RxO (Requested vs. Offered) Policies

Some policies follow "Requested vs. Offered" semantics:

Policy	Rule	Example
Reliability	Requested ≤ Offered	Reader RELIABLE ← Writer RELIABLE ✅ Reader RELIABLE ← Writer BEST_EFFORT ❌
Durability	Requested ≤ Offered	Reader TRANSIENT_LOCAL ← Writer TRANSIENT_LOCAL ✅ Reader TRANSIENT_LOCAL ← Writer VOLATILE ❌
Deadline	Offered ≤ Requested	Writer 1s deadline → Reader 2s deadline ✅ Writer 2s deadline → Reader 1s deadline ❌
Liveliness	Offered ≤ Requested	Writer 1s liveliness → Reader 2s lease ✅

Example: Compatible QoS

```
// Writer offers RELIABLE, TRANSIENT_LOCAL
auto writer_qos = astutedds::dcps::DataWriterQosBuilder()
    .reliability(astutedds::dcps::ReliabilityQosPolicyKind::RELIABLE_RELIABILITY_QOS)
    .durability(astutedds::dcps::DurabilityQosPolicyKind::TRANSIENT_LOCAL_DURABILITY_QOS)
    .build();

// Reader requests RELIABLE, TRANSIENT_LOCAL - Compatible! ✅
auto reader_qos = astutedds::dcps::DataReaderQosBuilder()
    .reliability(astutedds::dcps::ReliabilityQosPolicyKind::RELIABLE_RELIABILITY_QOS)
    .durability(astutedds::dcps::DurabilityQosPolicyKind::TRANSIENT_LOCAL_DURABILITY_QOS)
    .build();
```

Example: Incompatible QoS

```
// Writer offers BEST_EFFORT
auto writer_qos = astutedds::dcps::DataWriterQosBuilder()
    .reliability(astutedds::dcps::ReliabilityQosPolicyKind::BEST_EFFORT_RELIABILITY_QOS)
    .build();

// Reader requests RELIABLE - Incompatible! ❌
auto reader_qos = astutedds::dcps::DataReaderQosBuilder()
    .reliability(astutedds::dcps::ReliabilityQosPolicyKind::RELIABLE_RELIABILITY_QOS)
    .build();

// Result: Endpoints won't match, no communication
```

4.2.4 Common QoS Patterns

Pattern 1: Real-Time Sensor Data

High-frequency data where latest value matters:

```
auto sensor_qos = astutedds::dcps::DataWriterQosBuilder()
    .reliability(astutedds::dcps::ReliabilityQosPolicyKind::BEST_EFFORT_RELIABILITY_QOS)
    .durability(astutedds::dcps::DurabilityQosPolicyKind::VOLATILE_DURABILITY_QOS)
    .history(astutedds::dcps::HistoryQosPolicyKind::KEEP_LAST_HISTORY_QOS, 1)
    .build();
```

Pattern 2: Command and Control

Critical messages that must be delivered:

```
auto command_qos = astutedds::dcps::DataWriterQosBuilder()
    .reliability(astutedds::dcps::ReliabilityQosPolicyKind::RELIABLE_RELIABILITY_QOS)
    .durability(astutedds::dcps::DurabilityQosPolicyKind::TRANSIENT_LOCAL_DURABILITY_QOS)
    .history(astutedds::dcps::HistoryQosPolicyKind::KEEP_ALL_HISTORY_QOS)
    .build();
```

Pattern 3: State Synchronization

Current state for late joiners:

```
auto state_qos = astutedds::dcps::DataWriterQosBuilder()
    .reliability(astutedds::dcps::ReliabilityQosPolicyKind::RELIABLE_RELIABILITY_QOS)
    .durability(astutedds::dcps::DurabilityQosPolicyKind::TRANSIENT_LOCAL_DURABILITY_QOS)
    .history(astutedds::dcps::HistoryQosPolicyKind::KEEP_LAST_HISTORY_QOS, 1)
    .build();
```

Pattern 4: Event Stream

Sequence of events that must all be received:

```
auto event_qos = astutedds::dcps::DataWriterQosBuilder()
    .reliability(astutedds::dcps::ReliabilityQosPolicyKind::RELIABLE_RELIABILITY_QOS)
    .durability(astutedds::dcps::DurabilityQosPolicyKind::VOLATILE_DURABILITY_QOS)
    .history(astutedds::dcps::HistoryQosPolicyKind::KEEP_ALL_HISTORY_QOS)
    .build();
```

Pattern 5: Failover System

High-availability with exclusive ownership:

```
// Primary system
auto primary_qos = astutedds::dcps::DataWriterQosBuilder()
    .reliability(astutedds::dcps::ReliabilityQosPolicyKind::RELIABLE_RELIABILITY_QOS)
    .ownership(astutedds::dcps::OwnershipQosPolicyKind::EXCLUSIVE_OWNERSHIP_QOS)
    .ownership_strength(200)
    .liveliness(
        astutedds::dcps::LivelinessQosPolicyKind::MANUAL_BY_TOPIC_LIVELINESS_QOS,
        1s
    )
    .build();

// Backup system
auto backup_qos = astutedds::dcps::DataWriterQosBuilder()
    .reliability(astutedds::dcps::ReliabilityQosPolicyKind::RELIABLE_RELIABILITY_QOS)
    .ownership(astutedds::dcps::OwnershipQosPolicyKind::EXCLUSIVE_OWNERSHIP_QOS)
    .ownership_strength(100)  // Lower strength
    .liveliness(
        astutedds::dcps::LivelinessQosPolicyKind::MANUAL_BY_TOPIC_LIVELINESS_QOS,
        1s
    )
    .build();
```

4.2.5 Complete Example

Here's a complete example showing QoS configuration:

```
#include <astutedds/dcps/domain_participant.hpp>
#include <astutedds/dcps/qos.hpp>
#include "SensorData_TypeSupport.hpp"

int main() {
    using namespace astutedds::dcps;

    // Create participant
    auto participant = DomainParticipantFactory::create_participant(0);

    // Configure topic QoS
    auto topic_qos = TopicQosBuilder()
        .reliability(ReliabilityQosPolicyKind::RELIABLE_RELIABILITY_QOS)
        .durability(DurabilityQosPolicyKind::TRANSIENT_LOCAL_DURABILITY_QOS)
        .build();

    // Create topic
    auto topic = participant->create_topic<SensorData>(
        "SensorTopic",
        topic_qos
    );

    // Configure writer QoS
    auto writer_qos = DataWriterQosBuilder()
        .reliability(ReliabilityQosPolicyKind::RELIABLE_RELIABILITY_QOS)
        .durability(DurabilityQosPolicyKind::TRANSIENT_LOCAL_DURABILITY_QOS)
        .history(HistoryQosPolicyKind::KEEP_LAST_HISTORY_QOS, 10)
        .deadline(std::chrono::seconds(1))
        .resource_limits(1000, 100, 10)
        .build();
```

```
    // Create writer
    auto publisher = participant->create_publisher();
    auto writer = publisher->create_datawriter(topic, writer_qos);

    // Configure reader QoS (must be compatible)
    auto reader_qos = DataReaderQosBuilder()
        .reliability(ReliabilityQosPolicyKind::RELIABLE_RELIABILITY_QOS)
        .durability(DurabilityQosPolicyKind::TRANSIENT_LOCAL_DURABILITY_QOS)
        .history(HistoryQosPolicyKind::KEEP_LAST_HISTORY_QOS, 20)
        .deadline(std::chrono::seconds(2))  // More lenient than writer
        .build();

    // Create reader
    auto subscriber = participant->create_subscriber();
    auto reader = subscriber->create_datareader(topic, reader_qos);

    // Publish data
    SensorData data;
    data.temperature = 25.5;
    data.humidity = 60.0;
    writer->write(data);

    return 0;
}
```

4.2.6 QoS Best Practices

1. **Start with defaults**: Use `DATAWRITER_QOS_DEFAULT` initially
2. **Match carefully**: Ensure writer/reader QoS policies are compatible
3. **Test combinations**: Verify QoS behavior in your specific scenario
4. **Monitor resource usage**: KEEP_ALL can consume unbounded memory
5. **Use RELIABLE sparingly**: Only for data that must not be lost
6. **Set appropriate limits**: Configure resource limits to prevent OOM
7. **Consider late joiners**: Use TRANSIENT_LOCAL for state data
8. **Profile for performance**: BEST_EFFORT has lower latency

4.2.7 Next Steps

- Review QoS Examples for more code samples
- Test QoS with the Shapes Demo
- Learn about IDL Compiler for type generation
- Explore DDS Concepts for detailed theory

4.2.8 References

- DDS Specification - QoS Policies
- OMG DDS QoS Tutorial

4.3 DDS-XML QoS Profiles

AstuteDDS supports loading QoS policies from XML files using the `XmlQosLoader` class (`<astutedds/dcps/xml_config.hpp>`). This lets you configure DDS entities without recompiling, following the OMG DDS XML profile format.

4.3.1 XML Profile Format

Profiles are grouped into libraries. Each profile can contain QoS blocks for DataWriters, DataReaders, Topics, and DomainParticipants.

```
<dds>
  <qos_library name="MyLibrary">

    <qos_profile name="ReliableProfile">
      <datawriter_qos>
        <reliability><kind>RELIABLE</kind></reliability>
        <durability><kind>TRANSIENT_LOCAL</kind></durability>
        <history><kind>KEEP_LAST</kind><depth>10</depth></history>
      </datawriter_qos>
      <datareader_qos>
        <reliability><kind>RELIABLE</kind></reliability>
        <durability><kind>TRANSIENT_LOCAL</kind></durability>
      </datareader_qos>
    </qos_profile>

    <!-- Inherit from ReliableProfile, override history only -->
    <qos_profile name="DeepHistoryProfile" base_name="MyLibrary::ReliableProfile">
      <datawriter_qos>
        <history><kind>KEEP_ALL</kind></history>
      </datawriter_qos>
    </qos_profile>

  </qos_library>
</dds>
```

Profile keys use the format `"LibraryName::ProfileName"`.

4.3.2 Supported QoS Policies

Policy	XML element	Supported values
Reliability	`<reliability><kind>`	`BEST_EFFORT`, `RELIABLE`
Durability	`<durability><kind>`	`VOLATILE`, `TRANSIENT_LOCAL`, `TRANSIENT`, `PERSISTENT`
History	`<history><kind>`, `<depth>`	`KEEP_LAST`, `KEEP_ALL`
Liveliness	`<liveliness><kind>`, `<lease_duration>`	`AUTOMATIC`, `MANUAL_BY_PARTICIPANT`, `MANUAL_BY_TOPIC`
Deadline	`<deadline><period>`	duration in seconds
Ownership	`<ownership><kind>`	`SHARED`, `EXCLUSIVE`
Ownership Strength	`<ownership_strength><value>`	integer
Resource Limits	`<resource_limits>`	`max_samples`, `max_instances`, `max_samples_per_instance`
Destination Order	`<destination_order><kind>`	`BY_RECEPTION_TIMESTAMP`, `BY_SOURCE_TIMESTAMP`
Entity Factory	`<entity_factory><autoenable_created_entities>`	`true`, `false`

4.3.3 Loading Profiles

```
#include <astutedds/dcps/xml_config.hpp>

astutedds::dcps::XmlQosLoader loader;

// Load from a file
if (!loader.load_file("my_profiles.xml"))
{
    std::cerr << "Failed to load QoS profiles\n";
    return -1;
}

// Or load from a string
loader.load_string(xml_string);
```

Multiple files can be loaded; later loads merge into the existing profile map and overwrite duplicate keys.

4.3.4 Applying Profiles to Entities

```
#include <astutedds/dcps/domain_participant.hpp>
#include <astutedds/dcps/xml_config.hpp>

astutedds::dcps::XmlQosLoader loader;
loader.load_file("my_profiles.xml");

// DataWriter QoS
astutedds::dcps::DataWriterQos wqos;
if (loader.get_datawriter_qos("MyLibrary::ReliableProfile", wqos))
{
```

```
    auto writer = publisher->create_datawriter(topic, wqos);
}

// DataReader QoS
astutedds::dcps::DataReaderQos rqos;
if (loader.get_datareader_qos("MyLibrary::ReliableProfile", rqos))
{
    auto reader = subscriber->create_datareader(topic, rqos);
}

// Topic QoS
astutedds::dcps::TopicQos tqos;
loader.get_topic_qos("MyLibrary::ReliableProfile", tqos);

// DomainParticipant QoS
astutedds::dcps::DomainParticipantQos pqos;
loader.get_participant_qos("MyLibrary::ReliableProfile", pqos);
```

4.3.5 Profile Inheritance

A profile may set `base_name` to inherit another profile's QoS blocks. When a QoS block (e.g., `<datawriter_qos>`) is present in the derived profile, it fully overrides the corresponding block from the base. Blocks not present in the derived profile are inherited as-is from the base.

```
<!-- Base: RELIABLE writer -->
<qos_profile name="Base">
  <datawriter_qos>
    <reliability><kind>RELIABLE</kind></reliability>
  </datawriter_qos>
</qos_profile>

<!-- Derived: inherits RELIABLE writer qos block entirely; adds reader block -->
<qos_profile name="Derived" base_name="MyLib::Base">
  <datareader_qos>
    <reliability><kind>RELIABLE</kind></reliability>
  </datareader_qos>
</qos_profile>
```

4.3.6 Listing Available Profiles

```
for (const auto& key : loader.profile_keys())
    std::cout << key << "\n";
// e.g. "MyLibrary::ReliableProfile"
//      "MyLibrary::DeepHistoryProfile"
```

4.3.7 Clearing the Loader

```
loader.clear();   // Remove all loaded profiles
```

4.3.8 Thread Safety

`XmlQosLoader` is thread-safe. Concurrent calls to `load_file()`, `load_string()`, and any `get_*_qos()` method are safe.

4.4 Persistence Service

AstuteDDS provides a built-in Persistence Service (`<astutedds/dcps/persistence_service.hpp>`) that implements TRANSIENT and PERSISTENT durability as specified in OMG DDS 1.4 Section 2.2.3.4 and Section 2.2.3.20.

4.4.1 Overview

The Persistence Service allows late-joining DataReaders to receive data that was written before they were created. Two levels of persistence are supported:

Durability Kind	Storage	Survives writer restart	Survives service restart
`TRANSIENT_LOCAL`	In process memory (writer)	No	No
`TRANSIENT`	In-memory (service)	Yes	No
`PERSISTENT`	File-backed on disk	Yes	Yes

4.4.2 Configuring Durability QoS

Set the appropriate `DurabilityQosPolicyKind` on a DataWriter:

```
#include <astutedds/dcps/qos.hpp>

// TRANSIENT: samples survive DataWriter deletion but not service restart
auto transient_qos = astutedds::dcps::DataWriterQosBuilder()
    .durability(astutedds::dcps::DurabilityQosPolicyKind::TRANSIENT_DURABILITY_QOS)
    .build();

// PERSISTENT: samples survive service and writer restart
auto persistent_qos = astutedds::dcps::DataWriterQosBuilder()
    .durability(astutedds::dcps::DurabilityQosPolicyKind::PERSISTENT_DURABILITY_QOS)
    .build();
```

The matching DataReader should request at least the same (or weaker) durability:

```
auto reader_qos = astutedds::dcps::DataReaderQosBuilder()
    .durability(astutedds::dcps::DurabilityQosPolicyKind::TRANSIENT_DURABILITY_QOS)
    .build();
```

QoS Compatibility

A writer with `PERSISTENT` durability is compatible with readers requesting `VOLATILE`, `TRANSIENT_LOCAL`, `TRANSIENT`, or `PERSISTENT` durability.

4.4.3 Controlling Sample Retention

Use `DurabilityServiceQosPolicy` on the DataWriter to limit how many samples the Persistence Service retains:

```
astutedds::dcps::DurabilityServiceQosPolicy svc_qos;
svc_qos.history_kind = astutedds::dcps::HistoryQosPolicyKind::KEEP_LAST_HISTORY_QOS;
svc_qos.history_depth = 100;
svc_qos.max_samples = 1000;
svc_qos.max_instances = astutedds::dcps::LENGTH_UNLIMITED;
svc_qos.max_samples_per_instance = 100;

auto writer_qos = astutedds::dcps::DataWriterQosBuilder()
    .durability(astutedds::dcps::DurabilityQosPolicyKind::PERSISTENT_DURABILITY_QOS)
    .durability_service(svc_qos)
    .build();
```

4.4.4 Configuring the Storage Directory

By default, persistent stores are written to `.astutedds_persistence/` in the working directory. Override with:

```
astutedds::dcps::PersistenceService::instance().set_persistence_dir("/var/lib/astutedds");
```

This creates one binary file per topic under the configured directory. The files use the `ADPS` magic header and are forward-compatible across AstuteDDS versions.

4.4.5 How It Works

1. When a DataWriter publishes a sample with TRANSIENT or PERSISTENT durability, the sample is forwarded to `PersistenceService::store_sample()`.
2. On first write for a topic with PERSISTENT durability, any previously persisted samples are loaded from disk first, so the in-memory store is always complete.
3. When a late-joining DataReader matches a durable writer, historical samples are delivered via `PersistenceService::get_samples()`.
4. `PersistenceService::notify_writer_deleted()` is called when a writer is destroyed. For TRANSIENT topics the in-memory store is cleared; for PERSISTENT topics the file-backed store is retained.

4.4.6 Purging Persistent Stores

To remove the on-disk store for a topic:

```
astutedds::dcps::PersistenceService::instance().purge_persistent_store("MyTopic");
```

> **Warning**
>
> This permanently deletes the backing file. Use with care in production systems.

4.5 Recording and Replay

AstuteDDS provides `DataLogTool` (`<astutedds/dcps/record_replay.hpp>`) for capturing raw DDS payload streams to disk and replaying them later, preserving the original inter-sample timing.

4.5.1 Use Cases

- **System-level regression testing**: Record production traffic, replay against new software versions.
- **Simulation and training**: Pre-record realistic sensor feeds and replay on demand.
- **Debugging**: Capture anomalous behaviour for offline analysis.
- **Interop logging**: Archive DDS traffic for certification evidence.

4.5.2 Data Structures

```
struct astutedds::dcps::LoggedSample
{
    uint64_t    relative_timestamp_ns{0};  // nanoseconds since first sample
    std::string topic_name;
    std::vector<uint8_t> payload;          // raw XCDR-serialised bytes
};
```

4.5.3 Recording

Collect samples from DataReaders and build the `LoggedSample` list, then write them to a binary `.addslog` file:

```
#include <astutedds/dcps/record_replay.hpp>

std::vector<astutedds::dcps::LoggedSample> samples;
auto t0 = std::chrono::steady_clock::now();

// In your DataReader read/take loop:
for (auto& raw : raw_samples)
{
    auto elapsed = std::chrono::steady_clock::now() - t0;
    samples.push_back({
        static_cast<uint64_t>(
            std::chrono::duration_cast<std::chrono::nanoseconds>(elapsed).count()),
        "MyTopic",
        raw.serialized_payload      // XCDR bytes from the sample info
    });
}

// Write to disk
auto rc = astutedds::dcps::DataLogTool::write_samples("capture.addslog", samples);
if (rc != astutedds::dcps::ReturnCode_t::RETCODE_OK)
    std::cerr << "Failed to write log\n";
```

The file format uses an `ADDSRPL1` magic header and `FORMAT_VERSION = 1`. It is designed for forward compatibility.

4.5.4 Playback

Step 1 — Read the log

```
```

```
std::vector<astutedds::dcps::LoggedSample> samples;
auto rc = astutedds::dcps::DataLogTool::read_samples("capture.addslog", samples);
if (rc != astutedds::dcps::ReturnCode_t::RETCODE_OK)
{
    std::cerr << "Failed to read log\n";
    return 1;
}
```

Step 2 — Replay through a DomainParticipant

```
#include <astutedds/dcps/domain_participant.hpp>

auto participant = astutedds::dcps::DomainParticipantFactory::create_participant(0);

// Samples are replayed with the original inter-sample spacing preserved
auto rc = astutedds::dcps::DataLogTool::replay_samples(*participant, samples);
```

`replay_samples()` drives `DomainParticipant::send_data()` for each sample, sleeping between samples to honour the `relative_timestamp_ns` deltas. Samples **must** be sorted with monotonically non-decreasing timestamps; `RETCODE_BAD_PARAMETER` is returned otherwise.

4.5.5 File Size Limits

The following limits are enforced when reading log files to prevent resource exhaustion:

Limit	Value
Maximum topic name	1 024 bytes
Maximum payload size	32 MiB per sample
Maximum sample count	1 000 000 samples

4.5.6 Combining Recording with XML QoS

Configure writers and readers from an XML profile, then record:

```
#include <astutedds/dcps/xml_config.hpp>
#include <astutedds/dcps/record_replay.hpp>

astutedds::dcps::XmlQosLoader loader;
loader.load_file("profiles.xml");

astutedds::dcps::DataReaderQos rqos;
loader.get_datareader_qos("MyLib::ReliableProfile", rqos);

auto reader = subscriber->create_datareader(topic, rqos);
// … record samples as above …
```

4.6 IDL Compiler Guide

The AstuteDDS IDL compiler (`astutedds-idl`) translates OMG IDL 4.2 definitions into C++20 code with full X-Types support.

4.6.1 What is IDL?

Interface Definition Language (IDL) is a specification language used to define data types in a language-neutral way. The AstuteDDS IDL compiler generates C++ code from IDL definitions for use with DDS.

4.6.2 Installation

For package users, `astutedds-idl` is installed with the AstuteDDS developer package.

```
astutedds-idl --help
```

If you are building from source, build the tool target:

4.7 Compile shapes.idl to current directory

astutedds-idl examples/shapes_demo/shapes.idl mkdir build && cd build cmake .. -DASTUTEDDS_BUILD_TOOLS=ON cmake --build . --target astutedds-idl

```
Source-build binary location:
```

build/tools/idl_compiler/astutedds-idl

```
## Basic Usage

### Command Line Syntax

```bash
astutedds-idl [options] <input.idl> [output_directory]
```

Simple Example

```
# Compile shapes.idl to current directory
astutedds-idl examples/shapes_demo/shapes.idl

# Compile to specific output directory
astutedds-idl -o generated examples/shapes_demo/shapes.idl
```

Command Line Options

```
astutedds-idl --help

Options:
  -o, --output-dir <dir>      Output directory (default: current)
  -l, --language <lang>       Target language (cpp, c, python)
  --xcdr-version <1|2>        XCDR version (default: both)
  --namespace <ns>            Root namespace override
  --type-support              Generate TypeSupport only
```
```

```
  --verbose                  Verbose output
  --version                  Show version
  -h, --help                 Show help
```

4.7.1 Shapes Demo Example

The Shapes Demo uses a simple IDL file that demonstrates common DDS patterns.

shapes.idl

```
// File: examples/shapes.idl
module ShapesDemo {

    enum ShapeKind {
        CIRCLE,
        SQUARE,
        TRIANGLE
    };

    @mutable
    struct ShapeType {
        @key string<128> color;
        long x;
        long y;
        long shapesize;
    };

    @final
    struct Point {
        long x;
        long y;
    };

    @appendable
    struct Circle {
        @key long id;
        Point center;
        long radius;
    };
};
```

Compiling shapes.idl

```
# Compile the shapes IDL
astutedds-idl -o output examples/shapes_demo/shapes.idl

# Generated files:
# output/ShapeKind.hpp          - Enum definition
# output/ShapeType.hpp          - Main struct
# output/ShapeType_TypeSupport.hpp  - DDS TypeSupport
# output/ShapeType_XCDR.hpp     - Serialization code
# output/Point.hpp              - Point struct
# output/Circle.hpp             - Circle struct
```

Minimum Files for First Application

For the first publisher/subscriber workflow, use:

- `generated/<Type>.hpp`
- `generated/<Type>.cpp`

Additional generated files such as `*_TypeSupport.hpp` and `*_XCDR.hpp` are part of broader type support and advanced integration workflows.

Using Generated Code

For an end-to-end generated type workflow (IDL -> C++ -> publisher/subscriber), follow First Application.

The generated headers are then included by your app code:

```
#include "ShapeType.hpp"

ShapesDemo::ShapeType sample;
// Populate fields and publish using the DCPS API used in First Application.
```

4.7.2 IDL Language Features

Troubleshooting

Command Not Found

If `astutedds-idl` is not found, verify the developer package installation and that your shell `PATH` includes the package binary directory.

Parse or Syntax Errors

Use verbose mode to pinpoint the failing construct:

```
astutedds-idl --verbose <input.idl>
```

Then check for:

- Unclosed braces or semicolons
- Invalid annotations or annotation placement
- Unsupported/invalid IDL tokens

Generated Files Not Found in Build

Verify your CMake target includes both generated artifacts:

- `generated/<Type>.hpp`
- `generated/<Type>.cpp`

Also ensure your target include path contains the generated directory.

Modules (Namespaces)

Modules map to C++ namespaces:

```
module Sensors {
    module Temperature {
        struct Reading {
            float celsius;
            long timestamp;
        };
    };
};
```

Generated C++:

```
namespace Sensors {
namespace Temperature {
    struct Reading {
        float celsius;
        int32_t timestamp;
    };
} // namespace Temperature
} // namespace Sensors
```

Primitive Types

IDL primitive types map to C++ types:

IDL Type	C++ Type	Size
`boolean`	`bool`	1 byte
`char`	`char`	1 byte
`octet`	`uint8_t`	1 byte
`short`	`int16_t`	2 bytes
`unsigned short`	`uint16_t`	2 bytes
`long`	`int32_t`	4 bytes
`unsigned long`	`uint32_t`	4 bytes
`long long`	`int64_t`	8 bytes
`unsigned long long`	`uint64_t`	8 bytes
`float`	`float`	4 bytes
`double`	`double`	8 bytes
`long double`	`long double`	16 bytes

Strings

```
struct Message {
    string text;              // Unbounded string
    string<256> limited;      // Bounded string (max 256 chars)
};
```

Generated C++:

```
struct Message {
    std::string text;
    std::string limited;  // Application must enforce length
};
```

Sequences

```
struct DataSet {
    sequence<float> values;          // Unbounded sequence
    sequence<long, 100> readings;    // Bounded sequence (max 100)
};
```

Generated C++:

```
struct DataSet {
    std::vector<float> values;
    std::vector<int32_t> readings;  // Max size checked at runtime
};
```

Arrays

```
struct Matrix {
    float data[4][4];        // 2D array
    long vector[3];          // 1D array
};
```

Generated C++:

```
struct Matrix {
    std::array<std::array<float, 4>, 4> data;
    std::array<int32_t, 3> vector;
};
```

Enumerations

```
enum Status {
    IDLE,
    RUNNING,
    STOPPED,
    ERROR
};

struct SystemState {
```

```
    Status current_status;
};
```

Generated C++:

```
enum class Status : int32_t {
    IDLE = 0,
    RUNNING = 1,
    STOPPED = 2,
    ERROR = 3
};

struct SystemState {
    Status current_status{Status::IDLE};
};
```

Nested Structures

```
struct Point3D {
    double x;
    double y;
    double z;
};

struct Pose {
    Point3D position;
    Point3D orientation;
};
```

Generated C++:

```
struct Point3D {
    double x{0.0};
    double y{0.0};
    double z{0.0};
};

struct Pose {
    Point3D position;
    Point3D orientation;
};
```

4.7.3 IDL Annotations

Annotations control DDS behavior and code generation.

@key - Instance Identity

Marks fields that identify unique instances:

```
struct SensorData {
    @key long sensor_id;     // Key field
    float temperature;
```

```
    float humidity;
};
```

Effect:

- Each unique `sensor_id` is a separate instance
- Readers can track instances independently
- Used for instance lifecycle management

@optional - Optional Fields

Marks fields that may not always be present:

```
struct Config {
    long id;
    @optional string description;
    @optional float calibration_factor;
};
```

Generated C++:

```
struct Config {
    int32_t id{0};
    std::optional<std::string> description;
    std::optional<float> calibration_factor;
};
```

@id - Member ID

Explicitly assigns member IDs for XCDR2 mutable types:

```
@mutable
struct Extensible {
    @id(1) long field_a;
    @id(2) string field_b;
    @id(10) @optional float field_c;
};
```

Benefits:

- Forward/backward compatibility
- Fields can be reordered
- New fields can be added

@autoid - Automatic Member IDs

Generates member IDs automatically:

```
@mutable
@autoid(SEQUENTIAL)  // or HASH
struct AutoIdType {
    long field1;  // ID assigned automatically
    long field2;
```

```
    long field3;
};
```

Extensibility Annotations

Control type evolution compatibility:

```
@final
struct Fixed {
    // Cannot add/remove fields
    long id;
    string name;
};

@appendable
struct Growable {
    // Can add fields at end
    long id;
    string name;
};

@mutable
struct Flexible {
    // Can add/remove/reorder fields
    @id(1) long id;
    @id(2) string name;
};
```

4.7.4 Complete IDL Example

Here's a comprehensive example using various IDL features:

```
// sensor_system.idl
module SensorSystem {

    // Enumeration for sensor types
    enum SensorType {
        TEMPERATURE,
        PRESSURE,
        HUMIDITY,
        LIGHT
    };

    // Simple status enumeration
    enum Status {
        INACTIVE,
        ACTIVE,
        ERROR
    };

    // Point structure
    struct Location {
        double latitude;
        double longitude;
        @optional double altitude;
    };

    // Sensor reading (mutable for evolution)
    @mutable
```

```
    struct SensorReading {
        @key @id(1) long sensor_id;
        @id(2) SensorType type;
        @id(3) float value;
        @id(4) long long timestamp;
        @id(5) @optional Status status;
        @id(6) @optional Location location;
    };

    // Sensor configuration (final - won't change)
    @final
    struct SensorConfig {
        @key long sensor_id;
        string<64> name;
        SensorType type;
        float min_value;
        float max_value;
        long update_rate_ms;
    };

    // Aggregated statistics (appendable)
    @appendable
    struct SensorStatistics {
        @key long sensor_id;
        float mean_value;
        float std_deviation;
        long sample_count;
        sequence<float, 100> recent_values;
    };
};
```

Compiling the Example

```
./build/tools/idl_compiler/astutedds-idl sensor_system.idl generated/
```

Using the Generated Types

```
#include "generated/SensorReading.hpp"
#include "generated/SensorReading_TypeSupport.hpp"
#include "generated/SensorConfig.hpp"
#include "generated/SensorConfig_TypeSupport.hpp"

int main() {
    using namespace SensorSystem;

    // Create sensor reading
    SensorReading reading;
    reading.sensor_id = 42;
    reading.type = SensorType::TEMPERATURE;
    reading.value = 25.5f;
    reading.timestamp = get_current_time();
    reading.status = Status::ACTIVE;

    Location loc;
    loc.latitude = 51.5074;
    loc.longitude = -0.1278;
    loc.altitude = 11.0;
    reading.location = loc;

    // Create sensor configuration
    SensorConfig config;
```

```
    config.sensor_id = 42;
    config.name = "TempSensor-001";
    config.type = SensorType::TEMPERATURE;
    config.min_value = -50.0f;
    config.max_value = 150.0f;
    config.update_rate_ms = 1000;

    // Publish via DDS
    auto participant = create_participant(0);

    auto reading_topic = participant->create_topic<SensorReading>("SensorReadings");
    auto reading_writer = create_datawriter(participant, reading_topic);
    reading_writer->write(reading);

    auto config_topic = participant->create_topic<SensorConfig>("SensorConfigs");
    auto config_writer = create_datawriter(participant, config_topic);
    config_writer->write(config);

    return 0;
}
```

4.7.5 Generated Files

For each IDL type, the compiler generates:

1. Type Header (`TypeName.hpp`)

Contains the C++ struct definition:

```
#ifndef SENSORREADING_HPP
#define SENSORREADING_HPP

#include <cstdint>
#include <string>
#include <optional>

namespace SensorSystem {

struct SensorReading {
    int32_t sensor_id{0};
    SensorType type{SensorType::TEMPERATURE};
    float value{0.0f};
    int64_t timestamp{0};
    std::optional<Status> status;
    std::optional<Location> location;

    bool operator==(const SensorReading&) const = default;
};

} // namespace SensorSystem

#endif
```

2. TypeSupport Header (`TypeName_TypeSupport.hpp`)

Provides DDS integration:

```
#ifndef SENSORREADING_TYPESUPPORT_HPP
#define SENSORREADING_TYPESUPPORT_HPP
```

```
#include "SensorReading.hpp"
#include <astutedds/xtypes/type_object.hpp>

namespace SensorSystem {

class SensorReadingTypeSupport {
public:
    static astutedds::xtypes::CompleteTypeObject get_type_object();
    static astutedds::xtypes::TypeIdentifier get_type_identifier();
    static bool register_type(
        astutedds::dcps::DomainParticipant& participant,
        const std::string& type_name = "SensorSystem::SensorReading"
    );
};

} // namespace SensorSystem

#endif
```

3. XCDR Codec Header (`TypeName_XCDR.hpp`)

Serialization/deserialization code:

```
#ifndef SENSORREADING_XCDR_HPP
#define SENSORREADING_XCDR_HPP

#include "SensorReading.hpp"
#include <astutedds/cdr/cdr_types.hpp>

namespace SensorSystem {

class SensorReadingXCDR1 {
public:
    static bool serialize(
        astutedds::cdr::CDRBuffer& buffer,
        const SensorReading& value,
        astutedds::cdr::EncodingKind encoding
    );

    static bool deserialize(
        astutedds::cdr::CDRBuffer& buffer,
        SensorReading& value,
        astutedds::cdr::EncodingKind encoding
    );
};

class SensorReadingXCDR2 {
public:
    static bool serialize(
        astutedds::cdr::CDRBuffer& buffer,
        const SensorReading& value,
        astutedds::cdr::EncodingKind encoding
    );

    static bool deserialize(
        astutedds::cdr::CDRBuffer& buffer,
        SensorReading& value,
        astutedds::cdr::EncodingKind encoding
    );
};

} // namespace SensorSystem
```



```
#endif
```

4.7.6 Advanced Features

Type Inheritance

```
struct BaseMessage {
    long sequence_number;
    long long timestamp;
};

struct SensorMessage : BaseMessage {
    long sensor_id;
    float value;
};
```

Unions

```
union Data switch (long) {
case 1:
    float temperature;
case 2:
    long humidity;
case 3:
    string message;
default:
    octet raw_data;
};
```

Constants

```
const long MAX_SENSORS = 100;
const float PI = 3.14159;
const string<32> VERSION = "1.0.0";

struct Config {
    long max_count;  // Can use MAX_SENSORS
};
```

4.7.7 Best Practices

1. **Use modules**: Organize types in namespaces
2. **Add @key annotations**: Define instance identity clearly
3. **Use @mutable for evolution**: Allow future changes
4. **Bound collections**: Use `sequence<T, N>` instead of `sequence<T>`
5. **Document types**: Add comments for clarity
6. **Version your IDL**: Track changes over time
7. **Test generated code**: Compile and run tests

4.7.8 Troubleshooting

Common Errors

Error: Unknown type 'Foo'

- Solution: Define types before using them, or use forward declarations

Error: Circular dependency detected

- Solution: Break circular references using forward declarations

Error: Invalid @key annotation

- Solution: Only use @key on struct members, not nested types

Compiler Output

Enable verbose mode for detailed information:

```
./build/tools/idl_compiler/astutedds-idl --verbose examples/shapes.idl
```

4.7.9 Next Steps

- Explore QoS Usage with generated types
- Try the Shapes Demo
- Review IDL Examples
- Read the IDL 4.2 Specification

4.7.10 References

- OMG IDL 4.2 Specification
- DDS-XTypes 1.3
- AstuteDDS IDL Compiler Design

5. API Overview

5.1 DCPS API Overview

The DCPS (Data-Centric Publish-Subscribe) API is the main interface for DDS applications.

> **Full API Reference**
>
> Auto-generated reference documentation (from Doxygen source comments) is available in the API Reference section of the online documentation. The pages below give a guided overview of the main entry points.

5.1.1 Core Entities

DomainParticipant

Entry point for DDS applications.

```
auto& factory = astutedds::dcps::DomainParticipantFactory::instance();
auto* participant = factory.create_participant(0);  // Domain 0
if (participant && participant->enable()) {
    // DDS is now active
}
```

Topic

Named data channel.

```
auto* topic = participant->create_topic(
    "TopicName",
    "MyModule::DataType"
);
```

Publisher / DataWriter

Publish data to topics.

```
auto* publisher = participant->create_publisher();
auto* writer = publisher->create_datawriter(topic);

// Serialize and write data
std::vector<uint8_t> serialized;
// ... populate serialized data ...
writer->write(serialized);
```

Subscriber / DataReader

Subscribe to topic data.

```
auto* subscriber = participant->create_subscriber();
auto* reader = subscriber->create_datareader(topic);

// Read samples
```

```
std::vector<uint8_t> sample;
DDS::SampleInfo info;
if (reader->read_next_sample(sample, info) == DDS::ReturnCode_t::RETCODE_OK) {
    // Process sample
}
```

5.1.2 QoS Policies

See QoS Documentation for details.

5.1.3 XML QoS Profiles

Load QoS policies from OMG DDS XML profile files without recompiling.

```
#include <astutedds/dcps/xml_config.hpp>

astutedds::dcps::XmlQosLoader loader;
loader.load_file("profiles.xml");

astutedds::dcps::DataWriterQos wqos;
if (loader.get_datawriter_qos("MyLibrary::ReliableProfile", wqos))
    auto writer = publisher->create_datawriter(topic, wqos);
```

XmlQosLoader API

Method	Description
`load_file(path)`	Load profiles from an XML file
`load_string(xml)`	Load profiles from an in-memory XML string
`get_datawriter_qos(key, qos)`	Fill a `DataWriterQos` from a named profile
`get_datareader_qos(key, qos)`	Fill a `DataReaderQos` from a named profile
`get_topic_qos(key, qos)`	Fill a `TopicQos` from a named profile
`get_participant_qos(key, qos)`	Fill a `DomainParticipantQos` from a named profile
`profile_keys()`	Return all loaded profile keys
`clear()`	Remove all loaded profiles

Profile keys use the format `"LibraryName::ProfileName"`. Profiles may inherit from a base via the `base_name` attribute. Thread-safe.

See XML QoS Profiles Guide for full details and XML format.

5.1.4 Persistence Service

Delivers historical samples to late-joining DataReaders (OMG DDS 1.4 Section 2.2.3.4).

```
#include <astutedds/dcps/persistence_service.hpp>

// Override the storage directory (default: ".astutedds_persistence/")
astutedds::dcps::PersistenceService::instance().set_persistence_dir("/var/lib/astutedds");

// Configure a durable DataWriter
auto wqos = astutedds::dcps::DataWriterQosBuilder()
    .durability(astutedds::dcps::DurabilityQosPolicyKind::PERSISTENT_DURABILITY_QOS)
    .build();
```

Durability Kind	Storage	Survives restart
`TRANSIENT`	In-memory	No
`PERSISTENT`	File-backed	Yes

See Persistence Service Guide for full details.

5.1.5 Recording and Replay

Capture raw DDS payload streams to disk and replay them with original timing.

```
#include <astutedds/dcps/record_replay.hpp>

// Write
astutedds::dcps::DataLogTool::write_samples("capture.addslog", samples);

// Read back
std::vector<astutedds::dcps::LoggedSample> loaded;
astutedds::dcps::DataLogTool::read_samples("capture.addslog", loaded);

// Replay (preserves inter-sample timing)
astutedds::dcps::DataLogTool::replay_samples(*participant, loaded);
```

LoggedSample

```
struct LoggedSample {
    uint64_t             relative_timestamp_ns;  // ns since first sample
    std::string          topic_name;
    std::vector<uint8_t> payload;                // raw XCDR bytes
};
```

See Recording and Replay Guide for full details.

5.1.6 DDS Security

Secure DDS communication with authentication, access control, and encryption.

```
#include <astutedds/security/security.hpp>

// Create security plugins
auto auth = astutedds::security::create_pki_authentication_plugin();
auto crypto = astutedds::security::create_aes_gcm_crypto_plugin();

// Authenticate
SEC::CertificateCredentials creds = {...};
auto identity = auth->validate_local_identity(creds, 0, {}, ex);

// Encrypt payload before sending
std::vector<uint8_t> encrypted;
crypto->encode_serialized_payload(encrypted, plain, writer_crypto, ex);
```

See DDS Security Guide for complete examples.

5.1.7 Domain Routing

Bridge multiple DDS domains with automatic type checking and QoS mapping.

Use the `astutedds-router` tool to configure and manage domain bridging:

```
./bin/astutedds-router router_config.yaml
```

See DDS Domain Router Guide for configuration details.

5.1.8 AstuteDDS Inspector

Visual topology monitoring and diagnostics tool for DDS systems.

```
# Launch the Qt6-based GUI
./build/tools/inspect/astutedds-inspect

# Or inspect a specific domain
ASTUTEDDS_DOMAIN=1 ./build/tools/inspect/astutedds-inspect
```

Features: - Live participant and endpoint discovery - Real-time QoS compatibility checking - Message statistics and throughput - Topology export and logging

See AstuteDDS Inspector Guide for details.

5.1.9 Header Files

- `<astutedds/dcps/domain_participant.hpp>`
- `<astutedds/dcps/topic.hpp>`

- `<astutedds/dcps/publisher.hpp>`
- `<astutedds/dcps/subscriber.hpp>`
- `<astutedds/dcps/data_writer.hpp>`
- `<astutedds/dcps/data_reader.hpp>`
- `<astutedds/dcps/qos.hpp>`
- `<astutedds/dcps/xml_config.hpp>`
- `<astutedds/dcps/persistence_service.hpp>`
- `<astutedds/dcps/record_replay.hpp>`

5.2 RTPS Core

The RTPS (Real-Time Publish-Subscribe) layer implements the wire protocol.

5.2.1 Components

- **SPDP**: Simple Participant Discovery Protocol
- **SEDP**: Simple Endpoint Discovery Protocol
- **ReliabilityManager**: HEARTBEAT/ACKNACK protocol
- **Transport**: UDP/IPv4 communication

5.2.2 Headers

- `<astutedds/rtps/rtps_types.hpp>`
- `<astutedds/rtps/participant.hpp>`
- `<astutedds/rtps/writer.hpp>`
- `<astutedds/rtps/reader.hpp>`

For most applications, use the DCPS API instead of directly accessing RTPS.

5.3 X-Types API

X-Types provides dynamic type handling and type evolution.

5.3.1 TypeObject

Runtime representation of IDL types.

```
auto type_object = SensorDataTypeSupport::get_type_object();
auto type_id = SensorDataTypeSupport::get_type_identifier();
```

5.3.2 Dynamic Data

Manipulate data without generated code.

```
auto dynamic_data = astutedds::xtypes::DynamicData::create(type_object);
dynamic_data->set_float_value("temperature", 25.5f);
```

5.3.3 Assignability

Check type compatibility.

```
bool compatible = astutedds::xtypes::is_assignable(type_a, type_b);
```

5.3.4 Headers

- `<astutedds/xtypes/type_object.hpp>`
- `<astutedds/xtypes/dynamic_data.hpp>`
- `<astutedds/xtypes/assignability.hpp>`

6. About

6.1 Architecture

AstuteDDS is organized into layered, independently testable components that combine into a single static library:

Layer	Responsibility
RTPS Core	DDSI-RTPS 2.5 wire protocol — discovery, reliability, transport (UDP / TCP / shared memory)
DCPS API	OMG DDS 1.4 application API — DomainParticipant, Topic, Publisher, Subscriber, DataWriter, DataReader
X-Types	DDS-XTypes 1.3 type system — TypeObject, assignability, DynamicData, TypeLookup Service
CDR/XCDR	XCDR1 and XCDR2 serialisation codecs with DHEADER and EMHEADER support
Security SPI	DDS Security 1.1/1.2 plugin interfaces — authentication, access control, AES-GCM crypto
IDL Compiler	IDL 4.2 → C++ code generator, `astutedds-idl` command-line tool

6.1.1 Design Principles

- **Layered** — clear separation between the wire protocol, the DCPS API, and the type system; each layer can be tested independently
- **Static library** — a single `libastutedds.a` with no runtime shared-library dependencies, suitable for embedded and locked-down deployments
- **Modern C++20** — concepts, `std::span`, `std::endian`, and designated initialisers throughout; zero undefined behaviour
- **Thread-safe API** — all public API calls are safe to call from multiple threads concurrently

6.2 Specifications

AstuteDDS implements the following OMG specifications:

6.2.1 Core Specifications

DDSI-RTPS 2.5

Real-Time Publish-Subscribe Protocol

- **Status**: ✅ Implemented
- **Spec**: DDSI-RTPS 2.5
- **Features**: Discovery (SPDP/SEDP), Reliability, UDP transport

DDS-XTypes 1.3

Extensible and Dynamic Topic Types

- **Status**: ✅ Implemented
- **Spec**: DDS-XTypes 1.3
- **Features**: TypeObject, Assignability, Dynamic Data

IDL 4.2

Interface Definition Language

- **Status**: ✅ Implemented
- **Spec**: IDL 4.2
- **Features**: Full grammar, annotations, code generation

DDS Security 1.1/1.2

Security specification

- **Status**: ✅ Implemented
- **Spec**: DDS Security
- **Features**: Authentication, Access Control, Crypto

6.2.2 Additional Standards

Def Stan 23-009

UK Generic Vehicle Architecture

- **Status**: ✅ Complete
- **Spec**: Def Stan 23-009

6.2.3 Interoperability

AstuteDDS is tested for interoperability with:

- OpenDDS
- Fast DDS (eProsima)
- Cyclone DDS
- RTI Connext DDS

6.3 License

AstuteDDS is commercial software developed and maintained by Astute Systems PTY LTD.

6.3.1 License

AstuteDDS is distributed under a commercial license. License terms, permitted uses, and distribution rights are defined in the agreement issued with your purchase or evaluation package. Contact Astute Systems for licensing enquiries.

6.3.2 Third-Party Components

AstuteDDS links against the following open-source components. Each is used in accordance with its respective license:

Component	License
Qt6 (optional, GUI tools only)	LGPL 3.0
OpenSSL (optional, DDS Security)	Apache 2.0

Full license texts for third-party components are included in the `LICENSES/` directory of the distribution package.

6.3.3 Contact

For licensing, support, or commercial enquiries:

- **Website**: astutesys.com
- **Email**: info@astutesys.com