Mini DDS Library Documentation

DDS Project

July 16, 2025

Contents

Chapter 1

Introduction

Mini-DDS is an educational, modern C++ Data Distribution Service library designed to teach advanced concurrency, communication, and system-level programming concepts. This documentation provides an in-depth explanation of the architecture, API, and implementation details, with a focus on modern C++ techniques such as templates, type-erasure, smart pointers, and more.

Chapter 2

Architecture Overview

2.1 Modules and Relationships

2.2 Design Principles

- Separation of concerns - Extensibility - Type safety and runtime polymorphism - Testability and mockability

Chapter 3

Class Diagram

Chapter 4

API Reference

4.1 Publisher (`Publisher.hpp`)

4.1.1 Purpose and Overview

The `Publisher` class template is responsible for publishing messages of a specific type to a given topic within the DDS system. It is designed to be type-safe, extensible, and to delegate the actual message delivery to a transport layer, abstracting away the underlying communication mechanism (e.g., TCP, UDP, or a mock transport for testing).

4.1.2 Class Template Declaration

```
template<typename T>
class Publisher {
public:
    using TopicPtr = std::shared_ptr<Topic<T>>;
    using TransportPtr = std::shared_ptr<Transport>;

    virtual ~Publisher() = default;

    void publish(const Message<T>& msg);

protected:
    Publisher(const TopicPtr& topic, const TransportPtr& transport);
    TopicPtr topic_;
    TransportPtr transport_;
    friend class DomainParticipant;
};
```

4.1.3 Design Rationale

- **Template Parameter `T`:** Ensures compile-time type safety for published messages. Each publisher instance is bound to a specific message type.
- **Smart Pointers:** `std::shared_ptr` is used for both the topic and transport to manage shared ownership and avoid manual memory management. This is crucial in concurrent systems to prevent dangling pointers and memory leaks.
- **Protected Constructor:** The constructor is protected to enforce the use of the factory method in `DomainParticipant`, ensuring consistent initialization and dependency injection.
- **Transport Delegation:** The `publish()` method delegates the actual message delivery to the transport layer, decoupling the publisher from the communication mechanism.

4.1.4 Implementation Details

The `Publisher` class is implemented as a template with a protected constructor, ensuring that only `DomainParticipant` can create instances. The `publish()` method checks that both the transport and topic are valid, then delegates message delivery to the transport layer:

```
void publish(const Message<T>& msg) {
    if (transport_ && topic_) {
        transport_->send(*topic_, msg);
    }
}
```

The use of smart pointers ensures safe resource management. The `friend class DomainParticipant;` declaration allows only the participant to construct publishers, enforcing the factory pattern and dependency injection.

4.1.5 Thread Safety

The `Publisher` class itself is not inherently thread-safe. If multiple threads may call `publish()` concurrently, external synchronization (e.g., mutexes) should be used, or the transport implementation must guarantee thread safety.

4.1.6 Usage Example

```
#include <dds/DomainParticipant.hpp>
#include <dds/Publisher.hpp>
#include <dds/Topic.hpp>
#include <dds/Message.hpp>
#include <dds/Transport.hpp>
```

```
struct MyData { int value; };

auto transport = std::make_shared<dds::MockTransport>();
dds::DomainParticipant participant(transport);
auto topic = std::make_shared<dds::Topic<MyData>>("my_topic");
auto publisher = participant.create_publisher<MyData>(topic);

dds::Message<MyData> msg;
msg.data.value = 42;
msg.topic = "my_topic";
msg.timestamp = std::chrono::system_clock::now();
msg.sequence_number = 1;

publisher->publish(msg);
```

4.1.7 Edge Cases and Best Practices

- **Null Transport or Topic:** If either the transport or topic is null, `publish()` will do nothing. Always ensure valid dependencies.
- **Ownership:** Use `std::shared_ptr` for shared ownership. If unique ownership is required, consider `std::unique_ptr` with appropriate API changes.
- **Extensibility:** The design allows for custom transports (e.g., TCP, UDP, Mock) to be injected without modifying the publisher logic.
- **Performance:** For high-frequency publishing, consider batching or lock-free queues in the transport layer.

4.1.8 Advanced C++ Concepts

Templates Templates provide compile-time type safety and eliminate the need for dynamic casting. Each publisher is bound to a specific message type, preventing accidental type mismatches.

Smart Pointers `std::shared_ptr` is used to manage the lifetime of shared resources. This is especially important in concurrent systems where multiple entities may reference the same topic or transport.

Type-Erasure and Polymorphism While `Publisher` is templated, it can be used alongside type-erased constructs (e.g., `AnyMessage`) for generic APIs or logging. This hybrid approach combines the safety of templates with the flexibility of runtime polymorphism.

Friendship and Encapsulation The use of `friend class DomainParticipant;` restricts direct instantiation, enforcing the factory pattern and ensuring proper dependency injection.

4.2 Subscriber (`Subscriber.hpp`)

4.2.1 Purpose and Overview

The `Subscriber` class template is responsible for receiving messages of a specific type from a given topic within the DDS system. It is designed to be type-safe, extensible, and to delegate the actual message reception and delivery to a transport layer. The subscriber registers a callback function to be invoked when a new message arrives, supporting both synchronous and asynchronous/event-driven designs.

4.2.2 Class Template Declaration

```
template<typename T>
class Subscriber {
public:
    using TopicPtr = std::shared_ptr<Topic<T>>;
    using TransportPtr = std::shared_ptr<Transport>;
    using Callback = std::function<void(const Message<T>&)>;

    virtual ~Subscriber() = default;

    void set_callback(Callback cb);

protected:
    Subscriber(const TopicPtr& topic, const TransportPtr& transport);
    TopicPtr topic_;
    TransportPtr transport_;
    Callback callback_;
    friend class DomainParticipant;
};
```

4.2.3 Design Rationale

- **Template Parameter `T`:** Ensures compile-time type safety for received messages. Each subscriber instance is bound to a specific message type.
- **Smart Pointers:** `std::shared_ptr` is used for both the topic and transport to manage shared ownership and avoid manual memory management.
- **Callback Mechanism:** The subscriber uses a `std::function` callback to allow flexible message handling, including lambdas, function pointers, or functors.

- **Transport Delegation:** The `set_callback()` method registers the callback with the transport layer, decoupling the subscriber from the communication mechanism.
- **Protected Constructor:** The constructor is protected to enforce the use of the factory method in `DomainParticipant`, ensuring consistent initialization and dependency injection.

4.2.4 Implementation Details

The `Subscriber` class is implemented as a template with a protected constructor, ensuring that only `DomainParticipant` can create instances. The `set_callback()` method stores the callback and registers it with the transport layer:

```
void set_callback(Callback cb) {
    callback_ = cb;
    if (transport_ && topic_) {
        transport_->set_receive_callback(*topic_, cb);
    }
}
```

Smart pointers are used for safe resource management. The `friend class DomainParticipant;` declaration enforces the factory pattern and proper dependency injection.

4.2.5 Thread Safety and Asynchronous Operation

The `Subscriber` class itself is not inherently thread-safe. If multiple threads may call `set_callback()` concurrently, external synchronization should be used. The actual message delivery is typically handled asynchronously by the transport layer, which may use its own threads or event loops to invoke the callback. This design enables event-driven and reactive programming models.

4.2.6 Usage Example

```
#include <dds/DomainParticipant.hpp>
#include <dds/Subscriber.hpp>
#include <dds/Topic.hpp>
#include <dds/Message.hpp>
#include <dds/Transport.hpp>

struct MyData { int value; };

auto transport = std::make_shared<dds::MockTransport>();
dds::DomainParticipant participant(transport);
auto topic = std::make_shared<dds::Topic<MyData>>("my_topic");
auto subscriber = participant.create_subscriber<MyData>(topic);
```

```
subscriber->set_callback([](const dds::Message<MyData>& msg) {
    std::cout << "Received value:" << msg.data.value << std::endl;
});
```

4.2.7 Edge Cases and Best Practices

- **Null Transport or Topic:** If either the transport or topic is null, `set_callback()` will not register the callback. Always ensure valid dependencies.
- **Callback Lifetime:** Ensure that any resources captured by the callback (e.g., via lambda capture) outlive the subscriber or are managed safely to avoid dangling references.
- **Reentrancy:** If the callback modifies shared state, use appropriate synchronization (e.g., mutexes) to avoid data races.
- **Multiple Callbacks:** The current design supports a single callback per subscriber. For multiple listeners, consider a callback registry or observer pattern.
- **Performance:** For high-frequency message delivery, consider lock-free queues or thread pools in the transport layer.

4.2.8 Advanced C++ Concepts

Templates Templates provide compile-time type safety and eliminate the need for dynamic casting. Each subscriber is bound to a specific message type, preventing accidental type mismatches.

Smart Pointers `std::shared_ptr` is used to manage the lifetime of shared resources. This is especially important in concurrent systems where multiple entities may reference the same topic or transport.

Callback Registration and std::function The use of `std::function` allows for flexible callback registration, supporting lambdas, function pointers, and functors. This enables a wide range of message handling strategies, including capturing local state or integrating with event loops.

Asynchronous/Event-Driven Design The subscriber is designed to work with asynchronous transport layers. The callback may be invoked from a different thread or event loop, enabling reactive and event-driven programming models.

Friendship and Encapsulation The use of `friend class DomainParticipant;` restricts direct instantiation, enforcing the factory pattern and ensuring proper dependency injection.

4.3 Topic (`Topic.hpp`)

4.3.1 Purpose and Overview

The `Topic` class template represents a named channel for message exchange in the DDS system. Each topic is associated with a specific message type, ensuring type safety and clear separation of communication channels. Topics are used to bind publishers and subscribers together, allowing only compatible message types to be exchanged.

4.3.2 Class Template Declaration

```
template<typename T>
class Topic {
public:
    Topic(const std::string& name);
    virtual ~Topic() = default;

    const std::string& name() const;
    const std::type_info& type() const;

private:
    std::string name_;
};
```

4.3.3 Design Rationale

- **Template Parameter `T`:** Binds the topic to a specific message type, ensuring compile-time type safety for all publishers and subscribers associated with the topic.
- **Name and Type Information:** Each topic has a unique name and exposes its message type via `typeid(T)`. This allows for runtime type inspection and dynamic discovery mechanisms.
- **Smart Pointer Usage:** Topics are typically managed using `std::shared_ptr` to allow shared ownership between publishers, subscribers, and registries.

4.3.4 Member Functions

`Topic(const std::string& name)` Constructs a topic with the given name. The name should be unique within the DDS domain.

`const std::string& name() const` Returns the name of the topic. This is used to identify the topic in registries, transport layers, and for logging or debugging.

`const std::type_info& type() const` Returns the type information for the topic's message type. This is useful for runtime type inspection, type-erasure mechanisms, and generic registries.

4.3.5 Usage Example

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

struct MyData { int value; };

auto topic = std::make_shared<dds::Topic<MyData>>("my_topic");
std::cout << "Topic name: " << topic->name() << std::endl;
std::cout << "Topic type: " << topic->type().name() << std::endl;
```

4.3.6 Edge Cases and Best Practices

- **Unique Names:** Ensure that topic names are unique within the DDS domain to avoid message routing conflicts.
- **Type Mismatches:** Attempting to use a topic with a mismatched message type (e.g., subscribing with a different type than the publisher) will result in compile-time errors due to template enforcement.
- **Type Information:** The `type()` method returns a reference to a `std::type_info` object, which can be compared using `==` or `!=` for type-safe runtime checks.
- **Ownership:** Use `std::shared_ptr` for topics shared between multiple publishers and subscribers. For unique ownership, consider `std::unique_ptr`.

4.3.7 Advanced C++ Concepts

Templates and Type Safety Templates ensure that only compatible message types are published and subscribed on a given topic, preventing accidental type mismatches at compile time.

Runtime Type Information (RTTI) The `type()` method exposes the message type using `typeid(T)`, which returns a reference to a `std::type_info` object. This enables runtime type inspection, dynamic discovery, and integration with type-erasure mechanisms (e.g., `AnyTopic`).

Type-Erasure Integration Topics can be wrapped in type-erased containers (e.g., `AnyTopic`) for use in generic registries, logging, or dynamic discovery systems. The combination of compile-time and runtime type information provides both safety and flexibility.

4.4 Message (`Message.hpp`)

4.4.1 Purpose and Overview

The `Message` struct template encapsulates the data payload and associated metadata for messages exchanged in the DDS system. It is designed to be type-safe, extensible, and to carry essential information such as the topic name, Quality of Service (QoS) policies, timestamp, and a sequence number. This design enables advanced features like message ordering, delivery guarantees, and time-based filtering.

4.4.2 Struct Template Declaration

```
template<typename T>
struct Message {
    T data;
    std::string topic;
    QoS qos;
    std::chrono::system_clock::time_point timestamp;
    uint64_t sequence_number = 0;
};
```

4.4.3 Design Rationale

- **Template Parameter `T`:** Binds the message to a specific data type, ensuring compile-time type safety for all message exchanges.
- **Data Payload:** The `data` member holds the actual user-defined content of the message.
- **Topic Name:** The `topic` member records the name of the topic to which the message belongs, enabling routing and filtering.
- **QoS Policies:** The `qos` member allows for the specification of delivery guarantees, reliability, and other quality-of-service parameters.
- **Timestamp:** The `timestamp` member records the time at which the message was created or sent, enabling time-based filtering and ordering.
- **Sequence Number:** The `sequence_number` member provides a unique identifier for the message within a topic, supporting ordering and deduplication.

4.4.4 Member Fields

`T data` The user-defined payload of the message. This can be any type, including structs, classes, or primitive types.

`std::string topic` The name of the topic to which the message belongs. This is used for routing and filtering in the transport and middleware layers.

`QoS qos` The Quality of Service policies associated with the message. This struct can be extended to include reliability, durability, deadline, and other DDS QoS parameters.

`std::chrono::system_clock::time_point timestamp` The timestamp indicating when the message was created or sent. This uses the C++ standard library's `chrono` facilities for high-resolution, type-safe time representation.

`uint64_t sequence_number` A monotonically increasing sequence number for the message within its topic. This supports message ordering, deduplication, and replay.

4.4.5 Usage Example

```
#include <dds/Message.hpp>
#include <chrono>

struct MyData { int value; };

dds::Message<MyData> msg;
msg.data.value = 42;
msg.topic = "my_topic";
msg.qos = dds::QoS{/* ... */};
msg.timestamp = std::chrono::system_clock::now();
msg.sequence_number = 1;
```

4.4.6 Edge Cases and Best Practices

- **Default Initialization:** Always initialize all fields, especially `timestamp` and `sequence_number`, to ensure correct message ordering and time-based operations.
- **QoS Extension:** The `QoS` struct is designed to be extensible. Add fields as needed for your application (e.g., reliability, deadline, priority).
- **Topic Consistency:** Ensure that the `topic` field matches the topic used by the publisher and subscriber to avoid routing errors.
- **Copy and Move Semantics:** The struct supports default copy and move operations. For large payloads, consider using smart pointers or move semantics for efficiency.
- **Chrono Usage:** Use `std::chrono` for all time-related fields to ensure type safety and avoid unit mismatches.

4.4.7 Advanced C++ Concepts

Templates and Type Safety Templates ensure that only compatible message types are exchanged, preventing accidental type mismatches at compile time.

Chrono and Time Representation The use of `std::chrono::system_clock::time_point` provides a high-resolution, type-safe way to represent timestamps, supporting advanced time-based filtering and ordering.

Struct Layout and Performance The `Message` struct is a Plain Old Data (POD) type, enabling efficient copying, moving, and serialization. For large or complex payloads, consider using smart pointers or custom allocators.

Type-Erasure Integration Messages can be wrapped in type-erased containers (e.g., `AnyMessage`) for use in generic queues, logging, or dynamic dispatch systems. The combination of compile-time and runtime type information provides both safety and flexibility.

4.5 Types (`Types.hpp`)

4.5.1 Purpose and Overview

The `Types.hpp` header provides common type definitions and policy structures used throughout the DDS system. It centralizes shared types such as topic names and Quality of Service (QoS) policies, promoting consistency, extensibility, and maintainability across the codebase.

4.5.2 Type Definitions and Structures

```
using TopicName = std::string;

struct QoS {
    // Add QoS policy fields here (e.g., reliability, durability)
};
```

4.5.3 Design Rationale

- **Type Aliases:** The `TopicName` alias provides semantic clarity, making it explicit when a string is used as a topic name rather than arbitrary text.
- **QoS Policy Struct:** The `QoS` struct is designed to be extensible, allowing the addition of fields for reliability, durability, deadline, priority, and other DDS-specific policies as needed.
- **Centralization:** By defining common types in a single header, the codebase remains consistent and easier to refactor or extend.

4.5.4 Member Types

`using TopicName = std::string;` A type alias for topic names. This improves code readability and allows for future changes (e.g., switching to a different string type or adding validation) without widespread refactoring.

`struct QoS` A placeholder for Quality of Service policies. Extend this struct to include fields such as:

- `bool reliable;` – Whether reliable delivery is required
- `int priority;` – Message priority for scheduling
- `std::chrono::milliseconds deadline;` – Maximum acceptable delivery delay
- `bool durable;` – Whether messages should be persisted

4.5.5 Usage Example

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

dds::TopicName name = "my_topic";
dds::QoS qos;
qos.reliable = true;
qos.priority = 10;
qos.deadline = std::chrono::milliseconds(100);
qos.durable = false;
```

4.5.6 Edge Cases and Best Practices

- **Extensibility:** Always design policy structs like `QoS` to be forward-compatible. Use default values and document each field's semantics.
- **Type Aliases:** Prefer type aliases for semantically meaningful types. This improves code clarity and future-proofs the codebase.
- **Validation:** If topic names or QoS fields require validation, encapsulate logic in helper functions or class methods.
- **Policy-Based Design:** Consider using policy-based design patterns, where behaviors (e.g., reliability, durability) are implemented as separate policy classes or traits and composed via templates.

4.5.7 Advanced C++ Concepts

Struct Design and Extensibility Design structs like `QoS` to be open for extension but closed for modification (Open/Closed Principle). Use default member initializers and document intended usage.

Policy-Based Design For advanced users, consider policy-based design patterns, where behaviors (e.g., reliability, durability) are implemented as separate policy classes or traits and composed via templates.

Type Aliases and Semantic Clarity Type aliases (`using`) improve code readability and maintainability, making it clear when a value represents a specific concept (e.g., a topic name) rather than a generic type.

4.6 DomainParticipant (`DomainParticipant.hpp`)

4.6.1 Purpose and Overview

The `DomainParticipant` class acts as the central entry point for interacting with the DDS system. It is responsible for managing the domain context, providing factory methods to create publishers and subscribers, and injecting shared dependencies such as the transport layer. This design enforces consistent initialization, encapsulation, and dependency management across the system.

4.6.2 Class Declaration

```
class DomainParticipant {
public:
    using TransportPtr = std::shared_ptr<Transport>;

    DomainParticipant(const TransportPtr& transport);
    virtual ~DomainParticipant() = default;

    template<typename T>
    std::shared_ptr<Publisher<T>> create_publisher(const std::shared_ptr<Topi

    template<typename T>
    std::shared_ptr<Subscriber<T>> create_subscriber(const std::shared_ptr<To

private:
    TransportPtr transport_;
};
```

4.6.3 Design Rationale

- **Factory Pattern:** By providing factory methods for publishers and subscribers, the `DomainParticipant` ensures that all entities are created with the correct dependencies and configuration.

- **Dependency Injection:** The transport layer is injected into the participant and propagated to all created publishers and subscribers, promoting loose coupling and testability.
- **Smart Pointers:** `std::shared_ptr` is used for transport and created entities to manage shared ownership and avoid manual memory management.
- **Encapsulation:** The participant encapsulates domain-level configuration and resources, preventing accidental misuse or inconsistent initialization.

4.6.4 Member Functions

`DomainParticipant(const TransportPtr& transport)` Constructs a participant with the given transport. The transport is shared by all publishers and subscribers created by this participant.

`template<typename T> std::shared_ptr<Publisher<T>> create_publisher(const std::shared_ptr topic)` Creates a publisher for the specified topic and message type, injecting the shared transport.

`template<typename T> std::shared_ptr<Subscriber<T>> create_subscriber(const std::shared_p topic)` Creates a subscriber for the specified topic and message type, injecting the shared transport.

4.6.5 Implementation Details and Advanced C++ Notes

Protected Constructors and Smart Pointers:

In C++, if a class has a protected constructor (as is the case for `Publisher` and `Subscriber`), `std::make_shared` cannot be used to construct instances, because `make_shared` is not a friend and cannot access protected/private constructors. Instead, you must use direct `new`:

```
// Correct way to construct with protected constructor:
return std::shared_ptr<Publisher<T>>(new Publisher<T>(topic, transport_));
return std::shared_ptr<Subscriber<T>>(new Subscriber<T>(topic, transport_));
```

This ensures that only `DomainParticipant` (as a friend) can construct these objects, enforcing the factory pattern and encapsulation.

Template Method Implementations in Header:

C++ requires that template method implementations be available in every translation unit that uses them. Therefore, all template methods (such as `create_publisher` and `create_subscriber`) must be implemented in the header file, not in a `.cpp` file. Otherwise, you will get linker errors for missing symbols.

Summary:

- Use direct `new` with `std::shared_ptr` for protected constructors.
- Keep all template method implementations in the header.

- This approach is standard and necessary for modern C++ template-based factory patterns.

4.6.6 Usage Example

```
#include <dds/DomainParticipant.hpp>
#include <dds/Publisher.hpp>
#include <dds/Subscriber.hpp>
#include <dds/Topic.hpp>
#include <dds/Transport.hpp>

struct MyData { int value; };

auto transport = std::make_shared<dds::MockTransport>();
dds::DomainParticipant participant(transport);
auto topic = std::make_shared<dds::Topic<MyData>>("my_topic");
auto publisher = participant.create_publisher<MyData>(topic);
auto subscriber = participant.create_subscriber<MyData>(topic);
```

4.6.7 Edge Cases and Best Practices

- **Transport Lifetime:** Ensure that the transport outlives all publishers and subscribers created by the participant to avoid dangling references.
- **Multiple Participants:** Multiple participants can coexist, each with its own transport and configuration. This enables isolated domains or test environments.
- **Thread Safety:** The participant itself is not inherently thread-safe. If multiple threads may create publishers or subscribers concurrently, external synchronization is required.
- **Extensibility:** The participant can be extended to manage additional domain-level resources, such as topic registries, discovery services, or configuration policies.

4.6.8 Advanced C++ Concepts

Factory Pattern and Encapsulation The use of factory methods enforces encapsulation and consistent initialization, preventing direct instantiation of publishers and subscribers with incorrect or missing dependencies.

Dependency Injection Injecting the transport layer at the participant level promotes loose coupling, testability, and the ability to swap out transport implementations (e.g., for testing or different network protocols).

Smart Pointer Ownership `std::shared_ptr` is used to manage the lifetime of shared resources, ensuring that the transport and created entities remain valid as long as needed.

Extensibility and Domain Management The participant can be extended to manage additional domain-level features, such as topic discovery, QoS negotiation, or security policies, making it a flexible foundation for advanced DDS systems.

4.7 Transport and MockTransport (`Transport.hpp`, `MockTranspor`

4.7.1 Purpose and Overview

The `Transport` class provides an abstract interface for message delivery in the DDS system. It decouples publishers and subscribers from the underlying communication mechanism, enabling support for various transports such as TCP, UDP, shared memory, or mock/test transports. The `MockTransport` class is a concrete implementation used for testing and in-process message delivery, simulating network behavior without actual I/O.

4.7.2 Class Declarations

```
class Transport {
public:
    virtual ~Transport() = default;

    template<typename T>
    using ReceiveCallback = std::function<void(const Message<T>&)>;

    template<typename T>
    virtual void send(const Topic<T>& topic, const Message<T>& msg) = 0;

    template<typename T>
    virtual void set_receive_callback(const Topic<T>& topic, ReceiveCallback<
};

class MockTransport : public Transport {
public:
    MockTransport() = default;
    ~MockTransport() override = default;

    template<typename T>
    void send(const Topic<T>& topic, const Message<T>& msg) override;

    template<typename T>
    void set_receive_callback(const Topic<T>& topic, ReceiveCallback<T> cb) o
};
```

4.7.3 Design Rationale

- **Abstract Interface:** The `Transport` class is a pure virtual interface, enabling multiple implementations (e.g., TCP, UDP, Mock) to be swapped in without changing publisher or subscriber logic.
- **Template Methods:** Template methods allow the transport to handle messages of any type, supporting type-safe delivery and callback registration.
- **Callback Registration:** Subscribers register callbacks with the transport, which invokes them when messages are received for a given topic.
- **Type-Erasure:** Internally, transports may use type-erasure (e.g., `std::any`) to store callbacks for different message types in a generic registry.
- **MockTransport:** The mock implementation enables in-process testing, simulating message delivery and callback invocation without real network I/O.

4.7.4 Type-Erased Transport Interface

The `Transport` interface now uses type-erasure for all message delivery and callback registration. Instead of template virtual methods, it provides:

```
virtual void send(const AnyTopic\& topic, const AnyMessage\& msg) = 0;
virtual void set_receive_callback(const AnyTopic\& topic, std::function<void(
```

This enables runtime polymorphism and allows the transport to handle messages and topics of any type.

4.7.5 MockTransport Implementation

`MockTransport` overrides the type-erased interface and provides template convenience methods for type safety:

```
void send(const AnyTopic\& topic, const AnyMessage\& msg) override;
void set_receive_callback(const AnyTopic\& topic, std::function<void(const An

template<typename T>
void send(const Topic<T>& topic, const Message<T>& msg) {
    send(AnyTopic(topic), AnyMessage(msg));
}

template<typename T>
void set_receive_callback(const Topic<T>& topic, std::function<void(const Mes
    set_receive_callback(
        AnyTopic(topic),
        [cb, type = typeid(Message<T>)](const AnyMessage& any_msg) {
            if (any_msg.type() == type) {
```

```
                cb(any_msg.get<T>());
            }
        }
    );
}
```

All callbacks are stored as type-erased lambdas, and message delivery is performed by looking up the callback by topic name and invoking it with the type-erased message.

4.7.6 Publisher and Subscriber Changes

`Publisher` and `Subscriber` now use the type-erased interface:

```
// Publisher
void publish(const Message<T>& msg) {
    if (transport_ && topic_) {
        transport_->send(AnyTopic(*topic_), AnyMessage(msg));
    }
}
// Subscriber
void set_callback(Callback cb) {
    callback_ = cb;
    if (transport_ && topic_) {
        transport_->set_receive_callback(
            AnyTopic(*topic_),
            [cb, type = typeid(Message<T>)](const AnyMessage& any_msg) {
                if (any_msg.type() == type) {
                    cb(any_msg.get<T>());
                }
            }
        );
    }
}
```

This ensures that all message delivery and callback registration is type-safe and runtime-polymorphic, while still allowing template convenience for user code.

4.7.7 Usage Example

```
#include <dds/Transport.hpp>
#include <dds/Topic.hpp>
#include <dds/Message.hpp>

struct MyData { int value; };

dds::MockTransport transport;
```

```
dds::Topic<MyData> topic("my_topic");

dds::Message<MyData> msg;
msg.data.value = 42;
msg.topic = "my_topic";

transport.set_receive_callback(topic, [](const dds::Message<MyData>& m) {
    std::cout << "Received: " << m.data.value << std::endl;
});

transport.send(topic, msg); // Immediately invokes the callback
```

4.7.8 Edge Cases and Best Practices

- **Type Safety:** Template methods ensure that only messages of the correct type are sent and received for each topic. Type mismatches are caught at compile time.
- **Callback Registry:** Internally, transports may use a map from topic name to a type-erased callback (e.g., `std::unordered_map<std::string, std::any>`). Use `std::any_cast` to safely retrieve the correct callback type.
- **Thread Safety:** For real transports, ensure that callback registration and message delivery are thread-safe. Use mutexes or lock-free data structures as needed.
- **Event Loops:** Advanced transports may use event loops, thread pools, or asynchronous I/O to deliver messages.
- **Testability:** The mock transport enables unit testing of publishers and subscribers without real network dependencies.

4.7.9 Advanced C++ Concepts

Type-Erasure and std::any To support callbacks for arbitrary message types, transports may use `std::any` to store callbacks in a generic registry. Use `std::any_cast` to retrieve the correct callback type at runtime. Example:

```
std::unordered_map<std::string, std::any> callbacks_;
// Register:
callbacks_[topic.name()] = callback;
// Retrieve:
auto cb = std::any_cast<ReceiveCallback<T>>(callbacks_[topic.name()]);
```

CRTP and Static Polymorphism For advanced users, the Curiously Recurring Template Pattern (CRTP) can be used to implement static polymorphism for transports, enabling compile-time dispatch and optimization.

Event Loops and Asynchronous Delivery Real transports may use event loops, thread pools, or asynchronous I/O to deliver messages. The mock transport delivers messages synchronously for simplicity.

Testability and Mocking The mock transport enables comprehensive unit testing by simulating message delivery and callback invocation without external dependencies.

4.8 AnyMessage (`AnyMessage.hpp`)

4.8.1 Purpose and Overview

The `AnyMessage` class provides a type-erased container for messages of arbitrary types in the DDS system. It enables runtime polymorphism, allowing messages of different types to be stored, passed, and processed generically. This is particularly useful for logging, generic queues, dynamic dispatch, and systems where the message type is not known at compile time.

4.8.2 Class Declaration

```
class AnyMessage {
public:
    AnyMessage() = default;
    AnyMessage(const AnyMessage&) = default;
    AnyMessage(AnyMessage&&) = default;
    AnyMessage& operator=(const AnyMessage&) = default;
    AnyMessage& operator=(AnyMessage&&) = default;
    ~AnyMessage() = default;

    template<typename T>
    AnyMessage(const Message<T>& msg);

    const std::type_info& type() const;

    template<typename T>
    const Message<T>& get() const;

    bool has_value() const;
private:
    std::any msg_;
    const std::type_info* type_ = nullptr;
};
```

4.8.3 Design Rationale

- **Type-Erasure:** By storing messages as `std::any`, `AnyMessage` can hold any `Message<T>` instance, decoupling code from specific message types.
- **Runtime Type Information:** The `type()` method exposes the type of the contained message, enabling safe runtime inspection and dispatch.
- **Safe Access:** The `get<T>()` method uses `std::any_cast` to safely retrieve the contained message, throwing an exception if the type does not match.
- **Copy and Move Semantics:** All standard copy and move operations are supported, making `AnyMessage` suitable for use in containers and generic APIs.

4.8.4 Member Functions

`template<typename T> AnyMessage(const Message<T>& msg)` Constructs an `AnyMessage` from a `Message<T>`, storing it in a type-erased form and recording its type information.

`const std::type_info& type() const` Returns the type information of the contained message. This can be compared to `typeid(Message<T>)` for safe type checks.

`template<typename T> const Message<T>& get() const` Retrieves the contained message as a `Message<T>`. Throws `std::bad_any_cast` if the type does not match. Always check `type()` before calling `get<T>()` to avoid exceptions.

`bool has_value() const` Returns true if a message is stored, false otherwise.

4.8.5 Usage Example

```
#include <dds/AnyMessage.hpp>
#include <dds/Message.hpp>

struct MyData { int value; };

dds::Message<MyData> msg;
msg.data.value = 42;

dds::AnyMessage any_msg(msg);

if (any_msg.type() == typeid(dds::Message<MyData>)) {
    const auto& recovered = any_msg.get<MyData>();
    std::cout << "Recovered value: " << recovered.data.value << std::endl;
}
```

4.8.6 Edge Cases and Best Practices

- **Type Safety:** Always check `type()` before calling `get<T>()` to avoid exceptions from `std::any_cast`.
- **Exception Handling:** `get<T>()` throws `std::bad_any_cast` if the type does not match. Use try-catch blocks or type checks as needed.
- **Performance:** Type-erasure introduces a small runtime overhead compared to direct template usage. Use only where flexibility is required.
- **Copy and Move:** `AnyMessage` supports copy and move semantics, making it suitable for use in containers and generic APIs.
- **Integration:** Use `AnyMessage` for logging, generic queues, or dynamic dispatch where message types are not known at compile time.

4.8.7 Advanced C++ Concepts

Type-Erasure and std::any Type-erasure allows code to operate on objects without knowing their concrete type at compile time. `std::any` is a standard C++17 facility for type-erasure, storing any copyable type and retrieving it with `std::any_cast`.

Safe Casting and Runtime Type Information `std::any_cast` checks the stored type at runtime and throws an exception if the types do not match. Always use `type()` to check the type before casting.

Integration with Generic APIs `AnyMessage` enables the construction of generic APIs, such as message queues, loggers, or dispatchers, that can handle messages of arbitrary types at runtime.

4.9 AnyTopic (`AnyTopic.hpp`)

4.9.1 Purpose and Overview

The `AnyTopic` class provides a type-erased container for topics of arbitrary message types in the DDS system. It enables runtime polymorphism, allowing topics of different types to be stored, passed, and processed generically. This is particularly useful for topic registries, dynamic discovery, logging, and systems where the topic type is not known at compile time.

4.9.2 Class Declaration

```
class AnyTopic {
public:
    AnyTopic() = default;
    AnyTopic(const AnyTopic&) = default;
```

```
    AnyTopic(AnyTopic&&) = default;
    AnyTopic& operator=(const AnyTopic&) = default;
    AnyTopic& operator=(AnyTopic&&) = default;
    ~AnyTopic() = default;

    template<typename T>
    AnyTopic(const Topic<T>& topic);

    const std::string& name() const;
    const std::type_info& type() const;
    bool valid() const;
private:
    std::string name_;
    const std::type_info* type_ = nullptr;
};
```

4.9.3 Design Rationale

- **Type-Erasure:** By storing only the topic name and type information, `AnyTopic` can represent any `Topic<T>` instance, decoupling code from specific message types.
- **Runtime Type Information:** The `type()` method exposes the type of the contained topic, enabling safe runtime inspection and dispatch.
- **Name Access:** The `name()` method provides access to the topic's name, supporting registries, logging, and discovery.
- **Validity Check:** The `valid()` method indicates whether the topic is properly initialized.
- **Copy and Move Semantics:** All standard copy and move operations are supported, making `AnyTopic` suitable for use in containers and generic APIs.

4.9.4 Member Functions

`template<typename T> AnyTopic(const Topic<T>& topic)` Constructs an `AnyTopic` from a `Topic<T>`, storing its name and type information in a type-erased form.

`const std::string& name() const` Returns the name of the contained topic. This is used for identification in registries, logging, and discovery.

`const std::type_info& type() const` Returns the type information of the contained topic. This can be compared to `typeid(T)` for safe type checks.

`bool valid() const` Returns true if the topic is properly initialized (i.e., has a non-empty name and valid type information).

4.9.5 Usage Example

```
#include <dds/AnyTopic.hpp>
#include <dds/Topic.hpp>

struct MyData { int value; };

dds::Topic<MyData> topic("my_topic");
dds::AnyTopic any_topic(topic);

if (any_topic.valid() && any_topic.type() == typeid(MyData)) {
    std::cout << "Topic name: " << any_topic.name() << std::endl;
    std::cout << "Topic type: " << any_topic.type().name() << std::endl;
}
```

4.9.6 Edge Cases and Best Practices

- **Type Safety:** Always check `type()` before using the topic in a type-specific context to avoid logic errors.
- **Validity:** Use `valid()` to ensure the topic is properly initialized before accessing its properties.
- **Performance:** Type-erasure introduces minimal overhead, as only the name and type information are stored.
- **Integration:** Use `AnyTopic` for registries, discovery services, or logging where topic types are not known at compile time.
- **Type Indexing:** For advanced use cases, consider using `std::type_index` to store and compare type information efficiently in registries.

4.9.7 Advanced C++ Concepts

Type-Erasure and Runtime Type Information Type-erasure allows code to operate on topics without knowing their concrete type at compile time. `AnyTopic` stores the topic's name and a pointer to its `std::type_info`, enabling safe runtime inspection and dispatch.

Type Indexing and Registries For efficient lookup and management of topics in registries or discovery services, use `std::type_index` (a wrapper around `std::type_info`) as a key in associative containers.

Integration with Generic APIs `AnyTopic` enables the construction of generic APIs, such as topic registries, loggers, or discovery mechanisms, that can handle topics of arbitrary types at runtime.

Chapter 5

Advanced C++ Techniques

5.1 Templates and Type-Erasure

Templates and type-erasure are two powerful techniques in modern C++ for achieving type safety and runtime flexibility.

5.1.1 Templates

Templates enable compile-time polymorphism, allowing code to be written generically for any type while maintaining type safety and performance. In DDS, templates are used for publishers, subscribers, topics, and messages, ensuring that only compatible types interact.

```
template<typename T>
class Publisher { /* ... */ };
```

Pros:

- Zero runtime overhead (all checks at compile time)
- Type safety: errors are caught early
- No need for dynamic casting

Cons:

- Code bloat for many types (each instantiation generates new code)
- Cannot store different types in the same container without type-erasure
- API can become complex for users unfamiliar with templates

5.1.2 Type-Erasure

Type-erasure enables runtime polymorphism by hiding the concrete type behind a uniform interface. In C++17, `std::any` is a standard facility for type-erasure. DDS uses type-erasure for `AnyMessage` and `AnyTopic` to allow generic storage and dispatch.

```
std::any value = Message<MyData>{};
if (value.type() == typeid(Message<MyData>)) {
    auto& msg = std::any_cast<Message<MyData>&>(value);
}
```

Pros:

- Store and process heterogeneous types in generic containers
- Enables dynamic dispatch and plugin architectures
- Useful for logging, registries, and dynamic discovery

Cons:

- Small runtime overhead (type checks, heap allocation)
- Loss of compile-time type safety (errors at runtime)
- Requires careful use of `any_cast` and type checks

5.1.3 Hybrid Patterns

Many real-world systems use a hybrid approach: templates for the fast path, type-erasure for generic APIs, logging, or dynamic dispatch. This combines the safety and performance of templates with the flexibility of type-erasure.

5.2 Smart Pointers and Ownership

Smart pointers manage resource ownership and lifetime, preventing memory leaks and dangling pointers. The main types are:

- `std::unique_ptr<T>`: Exclusive ownership, non-copyable, movable. Use for strict RAII.
- `std::shared_ptr<T>`: Shared ownership, reference-counted. Use when multiple entities need to share a resource.
- `std::weak_ptr<T>`: Non-owning reference to a `shared_ptr`. Use to break cycles and avoid leaks.

```
auto ptr = std::make_shared<MyType>();
std::weak_ptr<MyType> weak = ptr;
if (auto locked = weak.lock()) {
    // Safe to use *locked
}
```

Best Practices:

- Prefer `unique_ptr` for exclusive ownership
- Use `shared_ptr` only when necessary (shared ownership is rare in well-designed systems)
- Use `weak_ptr` to break cycles (e.g., observer patterns)
- Avoid raw pointers for ownership; use them only for non-owning references
- Be aware of thread safety: `shared_ptr` is thread-safe for reference counting, but not for the pointed-to object

5.3 std::any, any_cast, and Runtime Type Information

`std::any` is a C++17 type-erasure facility that can store any copyable type. `any_cast` retrieves the stored value, throwing `std::bad_any_cast` if the type does not match.

```
std::any value = 42;
try {
    int n = std::any_cast<int>(value);
} catch (const std::bad_any_cast&) {
    // Handle type mismatch
}
```

Runtime Type Information (RTTI):

- `typeid(T)` returns a `std::type_info` object for type T
- `typeid(expr)` returns the dynamic type of expr (if polymorphic)
- `std::type_index` is a wrapper for `type_info` for use as a map key

5.4 Thread Safety and Concurrency

Modern C++ provides a rich set of concurrency primitives:

- `std::mutex`, `std::lock_guard`, `std::unique_lock`: Mutual exclusion
- `std::atomic<T>`: Lock-free atomic operations
- `std::thread`: Thread creation and management
- `std::condition_variable`: Thread synchronization
- `std::future`, `std::promise`: Asynchronous result passing

```cpp
#include <thread>
#include <mutex>

std::mutex mtx;
int shared = 0;

void increment() {
    std::lock_guard<std::mutex> lock(mtx);
    ++shared;
}

std::thread t1(increment), t2(increment);
t1.join(); t2.join();
```

Best Practices for DDS:

- Minimize lock contention by reducing critical sections
- Prefer lock-free data structures for high-frequency message passing
- Use thread pools for scalable concurrency
- Always document thread safety guarantees in APIs
- Use `std::atomic` for simple counters, flags, and reference counts

Chapter 6

Usage Examples

6.1 Basic Publish/Subscribe Example

The following example demonstrates how to set up a simple publish/subscribe flow using the mini-DDS library. It shows how to create a domain participant, set up a mock transport, define a topic and message type, and connect a publisher and subscriber.

6.1.1 Example Code: pubsub_basic.cpp

```
#include <dds/DomainParticipant.hpp>
#include <dds/MockTransport.hpp>
#include <dds/Publisher.hpp>
#include <dds/Subscriber.hpp>
#include <dds/Topic.hpp>
#include <dds/Message.hpp>
#include <iostream>
#include <chrono>
#include <thread>

struct MyData {
    int value;
};

int main() {
    using namespace dds;

    // Create a shared MockTransport
    auto transport = std::make_shared<MockTransport>();

    // Create a DomainParticipant with the transport
    DomainParticipant participant(transport);
```

```
    // Create a Topic for MyData
    auto topic = std::make_shared<Topic<MyData>>("my_topic");

    // Create a Publisher and Subscriber for the topic
    auto publisher = participant.create_publisher<MyData>(topic);
    auto subscriber = participant.create_subscriber<MyData>(topic);

    // Register a callback for the subscriber
    subscriber->set_callback([](const Message<MyData>& msg) {
        std::cout << "[Subscriber] Received value: " << msg.data.value << std
        std::cout << "[Subscriber] Topic: " << msg.topic << std::endl;
        std::cout << "[Subscriber] Sequence: " << msg.sequence_number << std:
    });

    // Prepare a message
    Message<MyData> msg;
    msg.data.value = 42;
    msg.topic = "my_topic";
    msg.timestamp = std::chrono::system_clock::now();
    msg.sequence_number = 1;

    // Publish the message
    std::cout << "[Publisher] Publishing value: " << msg.data.value << std::e
    publisher->publish(msg);

    // Give time for the callback to execute (not needed for MockTransport, b
    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    return 0;
}
```

6.1.2 Explanation

- **DomainParticipant:** Manages the DDS domain and acts as a factory for publishers and subscribers.
- **MockTransport:** Provides in-process message delivery for testing and development.
- **Topic:** Defines a named channel for messages of type `MyData`.
- **Publisher:** Publishes messages to the topic via the transport.
- **Subscriber:** Registers a callback to receive messages from the topic.
- **Message:** Encapsulates the data payload and metadata (topic, timestamp, sequence number).

- **Flow:** The publisher sends a message, which is immediately delivered to the subscriber's callback by the mock transport.

6.1.3 Expected Output

```
[Publisher] Publishing value: 42
[Subscriber] Received value: 42
[Subscriber] Topic: my_topic
[Subscriber] Sequence: 1
```

Chapter 7

Development Roadmap

Appendix A

References and Further Reading