The Simple DDS API

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

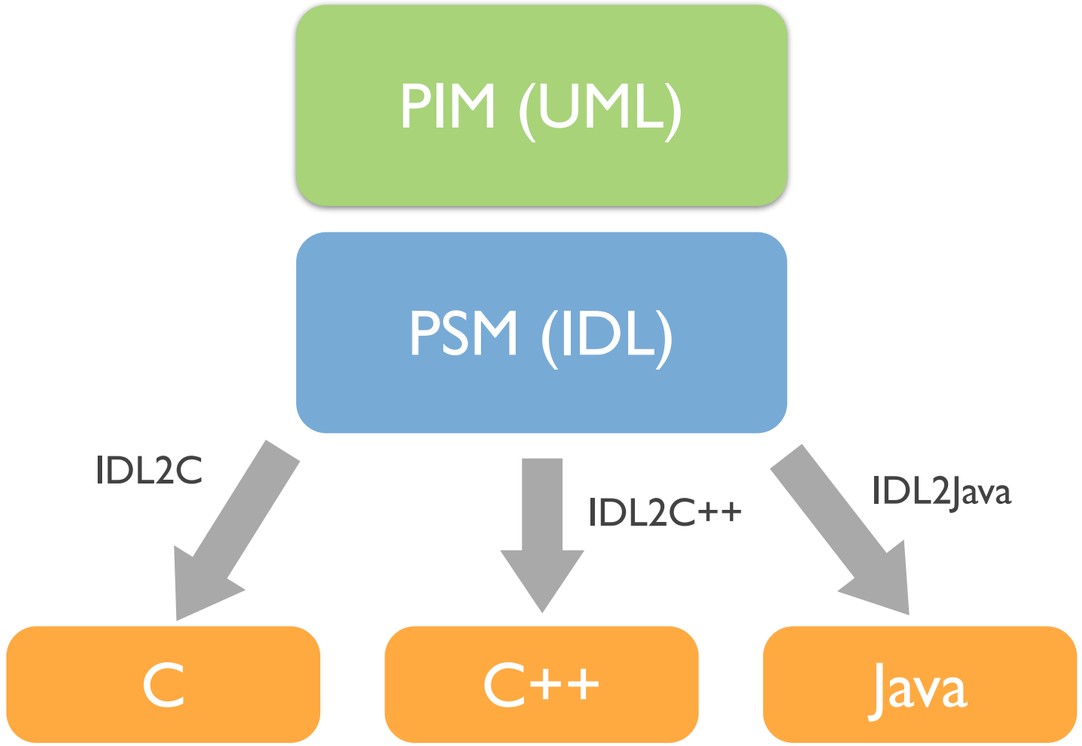
Angelo Corsaro, Ph.D.
Chief Technology Officer
PrismTech
OMG DDS SIG Co-Chair
angelo.corsaro@prismtech.com

The Simple DDS API

BACKGROUND

DDS PIM/PSM Approach

- The DDS standard is composed by a PIM specified in UML and A PSM specified in IDL
- Language specific APIs are derived via IDL-based mappings (e.g. IDL2C++, etc.)



What is the problem?

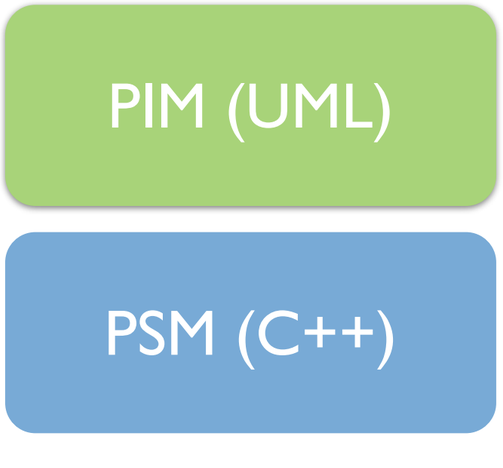
- IDL is very good for describing DDS Topic Types, yet...
- IDL's biggest strength, namely (programming) language independence, becomes its biggest weakness when trying to define APIs that are well integrated with a programming language
- For some programming languages (e.g. C++) the IDL2C++ mapping is "seasoned"

Why is this a problem?

- Resulting DDS APIs are more complicated than they should
- API don't feel natural to programmers
- API (esp. for C++) don't integrate well with standard libraries nor implements common idioms/patterns.

SimD: a Native C++ PSMs

- SimD addresses the problem by deriving a new C++ PSM from DDS starting directly from the PIM
- SimD takes advantage of all the features available in C++ in order to provide a natural, productive and safe API



SimD simplifies DDS Programming w/o limiting the expensiveness or introducing limitations in what possible

SimD in a Nutshell

- Provides transparent, safe, precise and real-time automatic memory management
- Takes advantage of C++ Template Meta-programming to automate tasks such as type registration and provide a strongly typed API (no downcast ever!)
- Provides a DDS API that exploits Iterators and containers as well as other C++ Standard Types
- DDS Events are types and are using Signal/Slots

SimD

The Simple DDS API

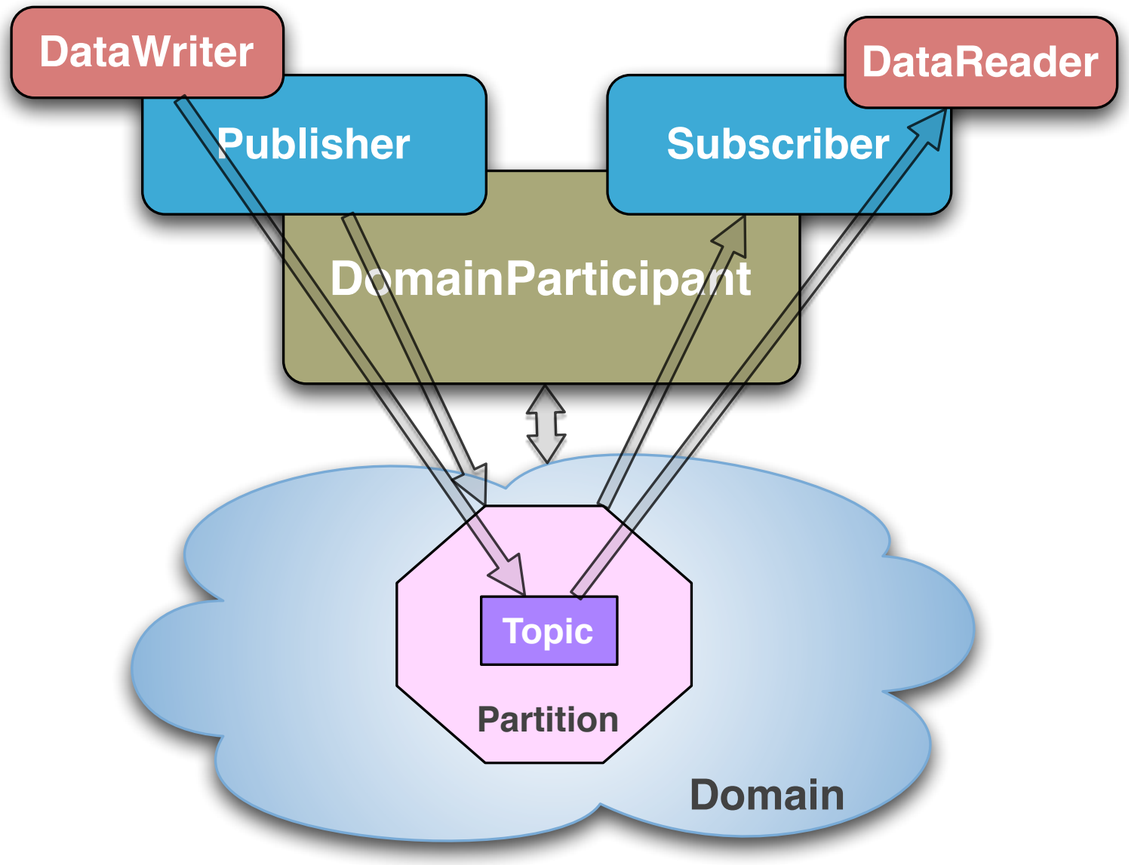
:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

LOOKING AT A DDS APP...

Anatomy of a DDS Application

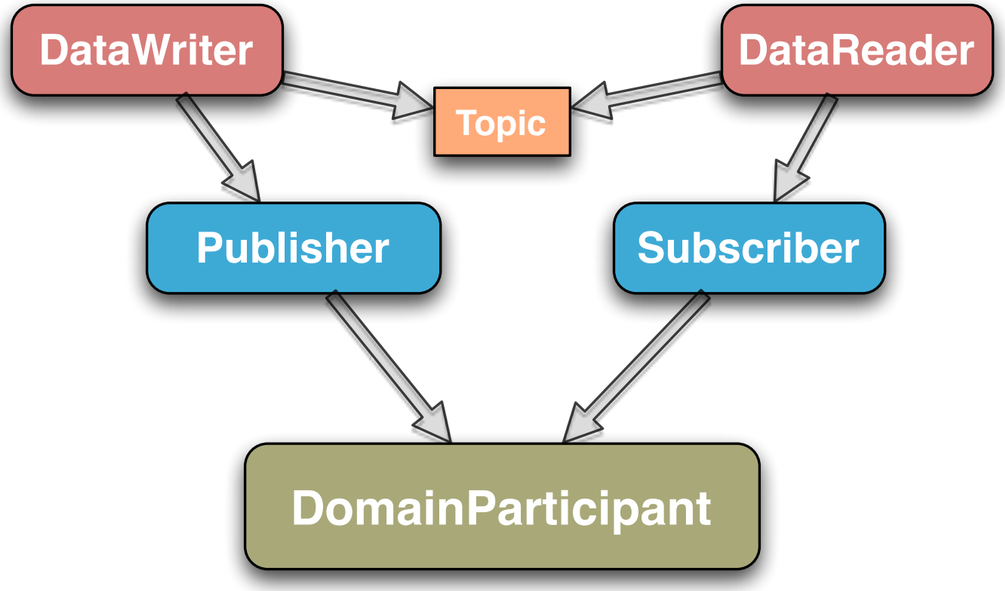
- Tree-like dependency between DP->(P,S)->(DR, DW) in creation and life-cycle management
- Example: **DW** alive requires the parent **P** to be alive
- This tree-like dependency allows precise GC of DDS Entities via reference counting

DP: DomainParticipant **DR**: DataReader
P: Publisher **DW**: DataWriter
S: Subscribe



SimD Memory Management

- SimD distinguishes between "Values" and "Reference" types
- Reference types are subject to automatic memory management
- DDS Entities (DP, P, S, DR, DW) and Conditions are references
- Policies and Samples are values



SiMD: STEP BY STEP

DDS App Steps

1. Define the Information Model

DDS App Steps

1. Define the Information Model
2. Connect to proper Domain/Partition

DDS App Steps

1. Define the Information Model
2. Connect to proper Domain/Partition
3. Create Topic

DDS App Steps

1. Define the Information Model
2. Connect to proper Domain/Partition
3. Create Topic
4. Create DataWriter

DDS App Steps

1. Define the Information Model
2. Connect to proper Domain/Partition
3. Create Topic
4. Create DataWriter
4. Create DataReader

Temperature Control App.

- Let's assume that we need to write an application for controlling the temperature of a building
- The first step is to be able to read/write the Temperature from sensors

```
enum TemperatureScale {
  CELSIUS,
  KELVIN,
  FAHRENHEIT
};

struct TempSensorType {
  short id;
  float temp;
  float hum;
   TemperatureScale scale;

};
#pragma keylist TempSensor id
```

SimD Runtime

Exploiting the structure of DDS applications, SimD provides a Runtime class that automates some of the common boilerplate

- SimD Runtime joins the default-domain and creates a Pub/Sub couple connected to the default partition
- The default behaviour can be overridden by passing proper parameters when creating the Runtime object

```
Runtime();
Runtime(const std::string& partition);
Runtime(const std::string& partition, const std::string& domain);
```

SimD Topic

- SimD Topics are template classes parametrized in the Topic-Type
- The Topic takes care of automatically registering the type
- If no TopicQoS is explicitly provided to the ctor, the topic is created with default Qos

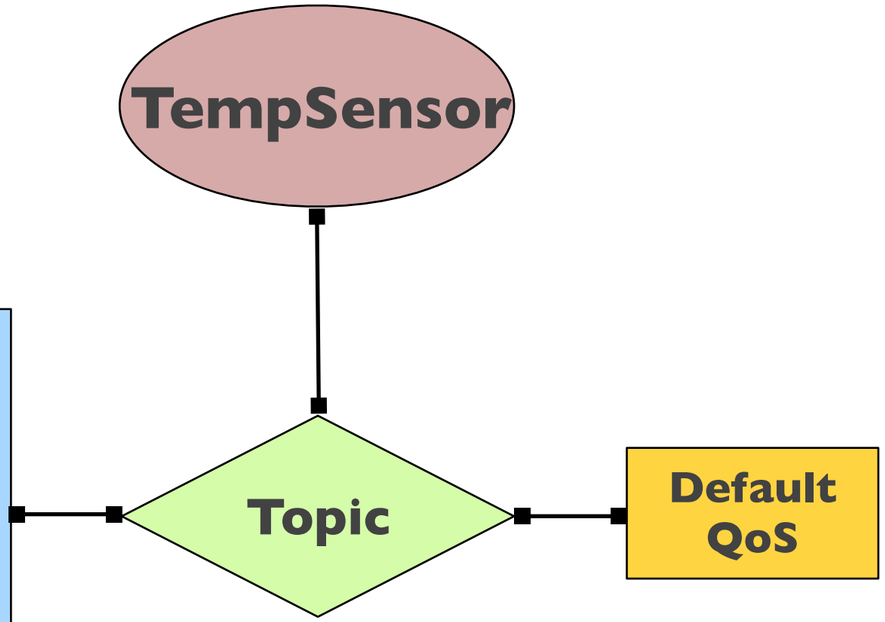
```
Topic(const std::string& name);

Topic(const std::string& name, const TopicQos& qos);

Topic(const std::string& name, const std::string& type_name);

Topic(const std::string& name, const std::string& type_name, const TopicQos& qos);
```

Registering the TempSensor Topic




```
struct TempSensorType {
  short id;
  float temp;
  float hum;
   TemperatureScale scale;

};
```

```
dds::Topic<TempSensorType> topic("TempSensor");
```

SimD DataWriter

- SIMD provides different DataWriter constructors allowing to control the level of customization required for the specific writer

```
template <typename T>
class dds::DataWriter : public dds::Entity {
public:
   DataWriter();

   DataWriter(Topic<T> topic)

   DataWriter(Topic<T> topic, const DataWriterQos& qos)

   DataWriter(Topic<T> topic, const DataWriterQos& qos, Publisher pub);
// ...
};
```

Writing Data with SimD

- SIMD provides two generic writes as well as a method for creating a writer dedicated to a specific instance
- The DataInstanceWriter provides constant time writes as it does not need to look-up the key-fields

```
dds::ReturnCode_t write(const T& sample);

dds::ReturnCode_t write(const T& sample, const dds::Time_t& timestamp);

dds::DataInstanceWriter<T> register_instance(const T& key);
```

Writing TempSensor Samples


```
// Join the default-domain and the “SensorData” partition
dds::Runtime runtime("SensorData");

// Create the “TempSensor” Topic
dds::Topic<TempSensorType> topic("TempSensor");

// Create a DataWriter
dds::DataWriter<TempSensorType> dw(Topic);

TempSensorType ts = {1, 26.0F, 70.0F, CELSIUS};

// Write Data
dw.write(ts);
```

Writing Data with QoS


```
dds::Duration latency_budget = {2, 0};
dds::Duration deadline = {4, 0};

// Create the TopicQos
dds::TopicQos tqos;

// Set Latency Budget, Transport Priority & Deadline
tqos.set_latency_budget(latency_budget);
tqos.set_deadline(deadline);
tqos.set_priority(tsPrio);

// Create the “TempSensor” Topic
dds::Topic<TempSensorType> topic("TempSensor", tqos);
dds::DataWriterQos dwqos(tqos);
// Create a DataWriter
dds::DataWriter<TempSensorType> dw(topic, dwqos);
```

SimD DataReader

- SIMD provides different DataReader constructors allowing to control the level of customization required for the specific reader

```
template <typename T>
class dds::DataReader : public dds::Entity {
public:
   DataReader();

   DataReader(Topic<T> topic)

   DataReader(Topic<T> topic, const DataReader Qos& qos)

   DataReader(Topic<T> topic, const DataReader Qos& qos, Subscriber pub);
// ...
};
```

Reading with SimD

- SimD supports legacy reads using Sequences along with std::vector and iterator-based(Forward + Output) read/take API

```
void read(TSeq& samples, dds::SampleInfoSeq& infos);

void read(std::vector<T>& data,  std::vector<dds::SampleInfo> info);

template <typename DataForwardIterator, typename InfoForwardIterator>
uint32_t
read(DataForwardIterator data_begin, InfoForwardIterator info_begin, uint32_t max_samples);

template <typename DataOutputIterator, typename InfoOutputIterator>
uint32_t
read(DataOutputIterator data_begin, InfoOutputIterator info_begin);
```

Swiss Army Knife Read

- The most generic read supported by DDS is provided through the TSeq, std::vector and iterator variations

```
void read(std::vector<T>& data,
          std::vector<dds::SampleInfo> info,
          dds::SampleStateMask samples_state,
          dds::ViewStateMask views_state,
          dds::InstanceStateMask instances_state)
```

Reading TempSensor Samples


```
// Join the default-domain and the “SensorData” partition
dds::Runtime runtime("SensorData");

// Create the “TempSensor” Topic
dds::Topic<TempSensorType> topic("TempSensor");

// Create a DataReader
dds::DataReader<TempSensorType> dr(topic);

std::vector<TemSensorType> data;
std::vector<SampleInfo> info;

while (true) {
   dr.read(data, info);
   for (int i =0; i < data.length(); ++i)
      std::cout << data[i] << std::endl;
   sleep(1);
}
```

Reading with QoS


```
dds::Duration latency_budget = {2, 0};
dds::Duration deadline = {4, 0};

// Create the TopicQos
dds::TopicQos tqos;

// Set Latency Budget, Transport Priority & Deadline
tqos.set_latency_budget(latency_budget);
tqos.set_deadline(deadline);
tqos.set_priority(tsPrio);

// Create the Topic
dds::Topic<TempSensorType> tsTopic("TempSensor", tqos);

// Create the DataReaderQos from TopicQos
dds::DataReaderQos drqos(tqos);

// Create the DataReader
dds::DataReader<TempSensorType> dr(tsTopic, drqos);
```

SimD WaitSets

- SimD WaitSets provides an improved usability along with new behaviour
- SimD WaitSets and Conditions are strongly typed
- SimD WaitSet rely on C++ functors to handle condition triggering (handling is done in the context of a user call)
- Let's see them in action...

Condition Handler


```
class TempSensorDataHandler {
public:
   void operator() (dds::DataReader<TempSensorType>& reader) {
      std::vector<TempSensorType> data;
      std::vector<dds::SampleInfo>     info;
      reader.read(data, info);
      //[NOTE #1] Do whatever is needed with the data
   }
};
```

SimD WaitSets


```
dds::DataReader<TempSensorType> dr(topic);

//[NOTE #1]: Create an instance of
//the handler
TempSensorDataHandler handler;

//[NOTE #2]: Create a read condition for
// the given reader
auto rcond =
   dr.create_readcondition(handler);

//[NOTE #3]: Create a Waitset and attach
// the condition
dds::WaitSet ws;
ws.attach(rcond);

dds::Duration timeout = {1, 0};
```

```
//[NOTE #4] Wait for some condition to
// become active, and retrieve all active
// conditions
auto conds = ws.wait(timeout);

//[NOTE #5] Execute the condition handlers
for (auto i = conds.begin();
     i < conds.end(); ++i)
   i->execute();


//[NOTE #5] Automatically dispatch the
// condition handlers when conditions
//are notified
ws.dispatch(timeout);
```

SimD “Listeners”

- SimD takes a different approach to DDS event notifications
- SimD associate a type to each event that DDS entities might ever raise
- Handlers can be connected only for supported events
- Handlers can be attached to individual events
- Handler can be any C++ callable entity that matches the expected signature, e.g., function, functor, methods

Event Handler


```
class TempSensorDataHandler {
public:
   void handle_data(dds::DataReader<TempSensorType>& reader) {
      std::cout << "Reading..." << std::endl;
      std::vector<TempSensorType> data;
      std::vector<dds::SampleInfo>     info;
      reader.read(data, info);
      //[NOTE #1] Do whatever is needed with the data
   }
};
```

Registering Handler


```
dds::DataReader<TempSensorType> dr(topic);

//[NOTE #1]: Create an instance of the handler
TempSensorDataHandler handler;


auto func =
   boost::bind(TempSensorDataHandler::handle_data,
               &handler, _1);

//[NOTE #2]: Register the handler for the relevant event
auto connection =
   dr.connect<on_data_available>(func);
```

SimD

The Simple DDS API

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

DEMO

SimD

The Simple DDS API

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

STATUS

SimD Status

- SimD has been an incubator for the new ISO C++ DDS PSM
- The code is already used by several OpenSplice DDS users in both R&D as well as commercial applications
- SimD API will transition to v1.x only after the adoption of the new C++ PSM so to ensure stable API with v1.x

SimD Availability

- SimD is an OpenSource project licensed under LGPL
- SimD is available for Linux/ Unix/Windows at:

http://code.google.com/p/simd-cxx/

SimD

The Simple DDS API

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

SUMMING UP +/-

Summing Up

- SimD provides a safe, efficient and productive C++ API for DDS
- SimD has been instrumental in experimenting ideas for evolving the ISO C++ DDS PSM
- SimD code is already pretty robust and used by a large community