



OpenSplice|DDS

Delivering Performance, Openness, and Freedom

Angelo Corsaro, Ph.D.

Chief Technology Officer

OMG DDS SIG Co-Chair

angelo.corsaro@prismtech.com



The DDS Tutorial

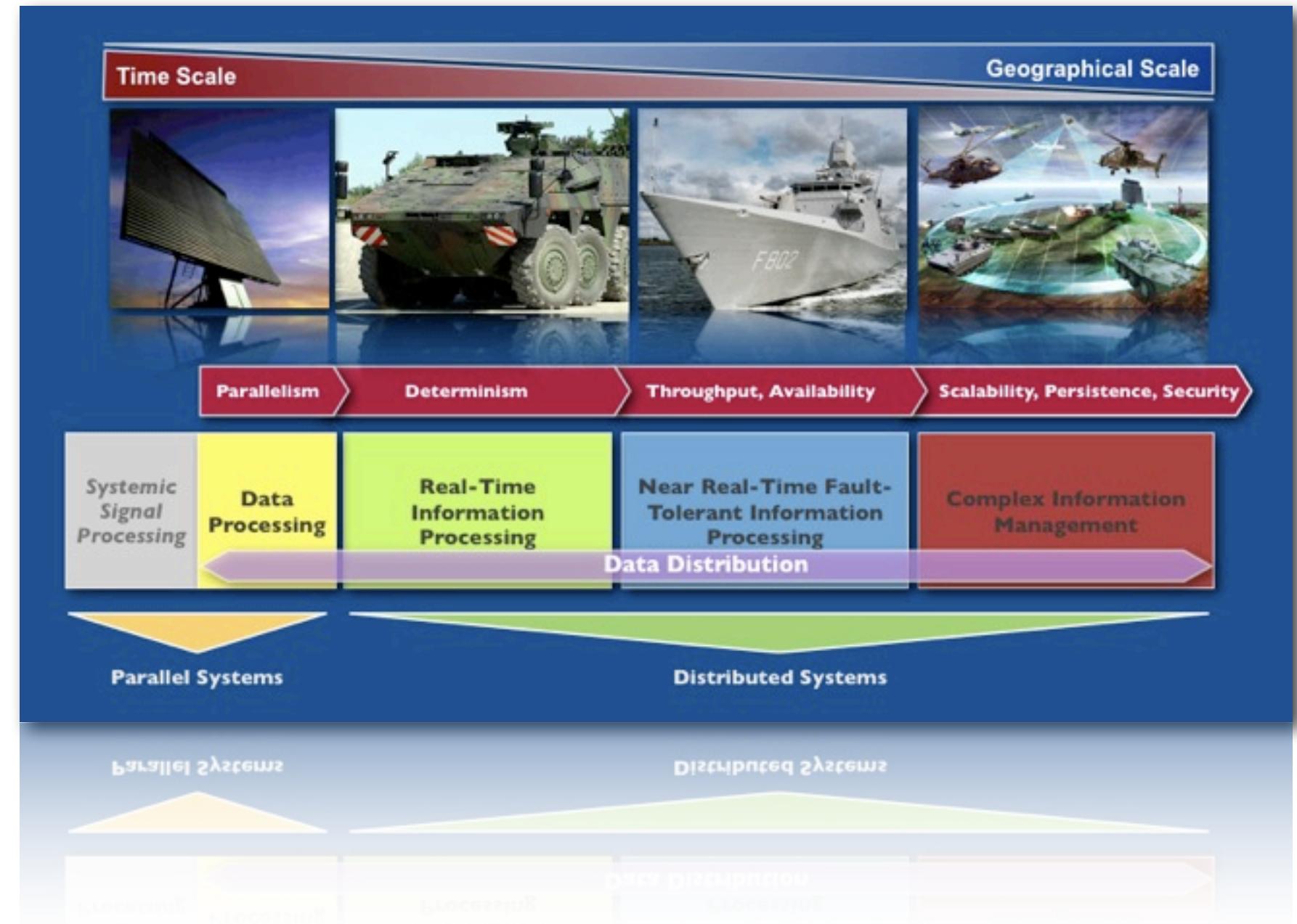
::Part I

Addressing Data Distribution Challenges

DDS is standard designed to address the data-distribution challenges across a wide class of Defense and Aerospace Applications

The OMG DDS Standard

- ▶ Introduced in 2004, DDS is a standard for **Real-Time, Dependable and High-Performance** Publish/Subscribe
- ▶ DDS behaviour and semantics can be controlled via a rich set of QoS Policies
- ▶ DDS is today **recommended by key administration worldwide** and **widely adopted** across several different application domains, such as, Automated Trading, Simulations, SCADA, Telemetry, etc.



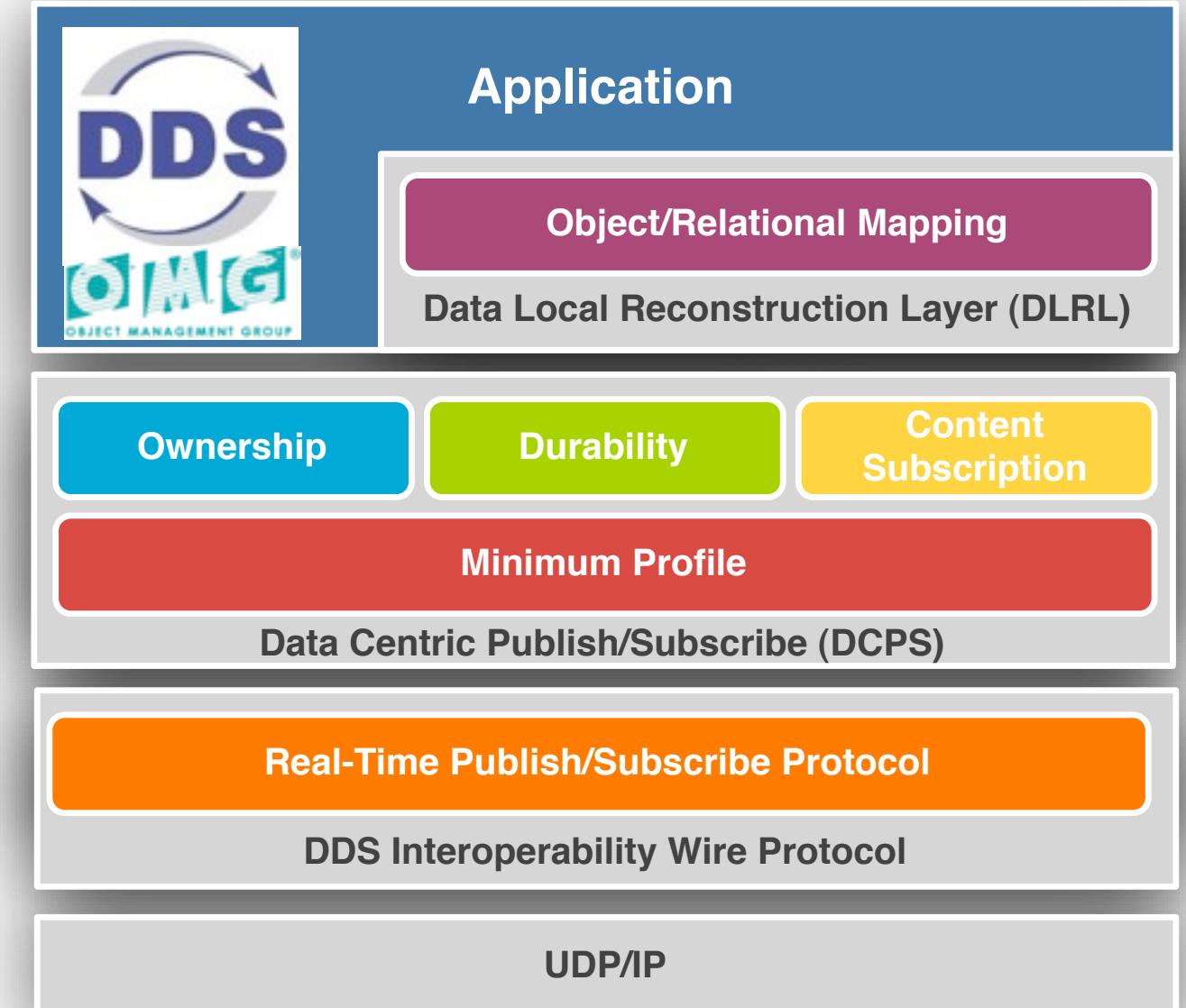
The OMG Data Distribution Service (DDS)

DDS v1.2 API Standard

- ▶ Language Independent, OS and HW architecture independent
- ▶ **DCPS.** Standard API for Data-Centric, Topic-Based, Real-Time Publish/Subscribe
- ▶ **DLRL.** Standard API for creating Object Views out of collection of Topics

DDSI/RTPS v2.1 Wire Protocol Standard

- ▶ Standard wire protocol allowing interoperability between different implementations of the DDS standard
- ▶ Interoperability demonstrated among key DDS vendors in March 2009



Tutorial Scope

Scope & Goals

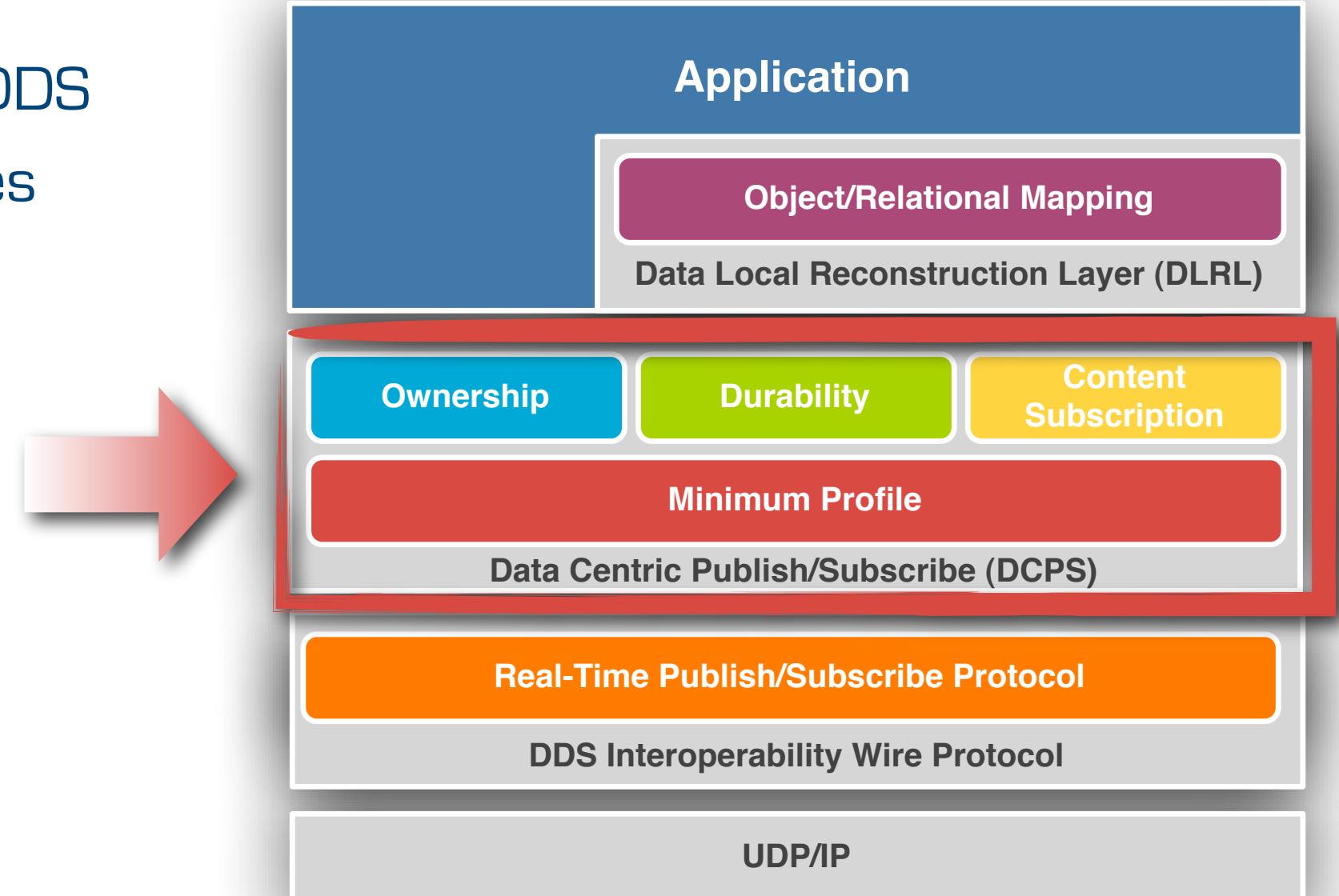
- ▶ The Tutorial will cover the DCPS layer of DDS
- ▶ It will give you enough details and examples to make sure that you can get started writing DDS applications

Software

- ▶ OpenSplice DDS
 - ▶ <http://www.opensplice.org>
- ▶ SIMple Dds (SIMD)
 - ▶ <http://code.google.com/p/simd-cxx>

Prerequisite

- ▶ Basic C++ understanding



openSplice|DDS

Delivering Performance, Openness, and Freedom

Your will learn:

- What is a Topic
- How to define Topic Types
- How to register a Topic

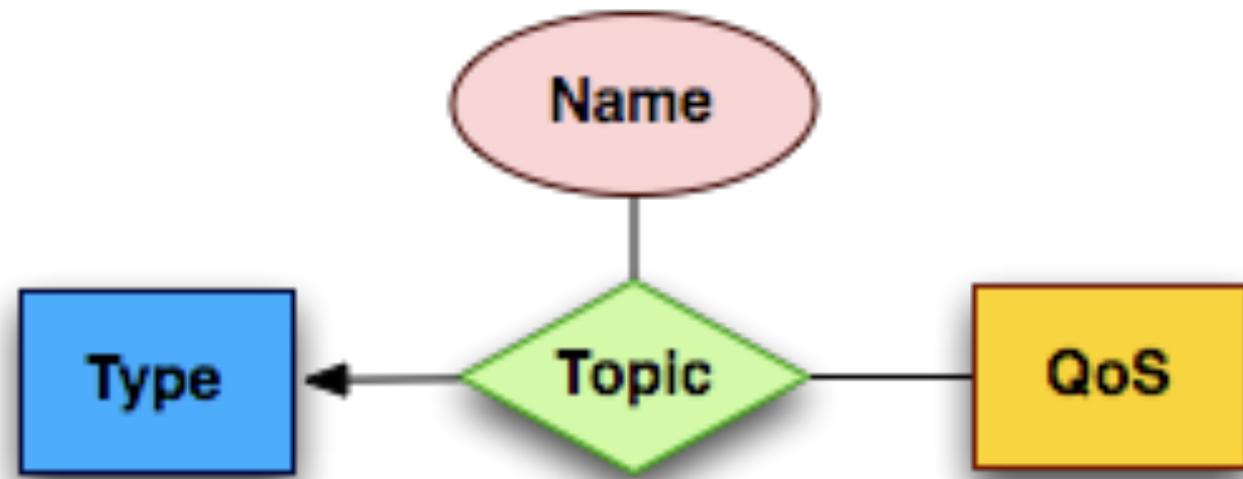
Step I

Defining the Data

Topics

Topic

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



Topic Types

A DDS **Topic Type** is described by an **IDL Structure** containing an **arbitrary number for fields** whose **types might be**:

- ▶ IDL primitive types, e.g., octet, short, long, float, string (bound/unbound), etc.
- ▶ Enumeration
- ▶ Union
- ▶ Sequence (bounded or unbounded)
- ▶ Array
- ▶ Structure (nested)

Examples

```
struct HelloTopicType {  
    string message;  
};
```

```
struct PingType  
{  
    long counter;  
    string<32> vendor;  
};
```

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

```
struct Counter {  
    long cID;  
    long count;  
};
```

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

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

Topic Types & Keys

- ▶ Each Topic Type **has to define its key-set** (which might be the empty set)
- ▶ There are no limitations on the number of attributes used to represent a key
- ▶ Keys can be top-level attributes as well as nested-attributes (i.e. attributes in nested structures)

Key Examples -- Empty Key-Set

```
struct HelloTopicType {  
    string message;  
};  
#pragma keylist HelloTopicType
```

```
struct PingType  
{  
    long counter;  
    string<32> vendor;  
};  
#pragma keylist PingType
```

Key Examples -- User-Defined Keys

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

```
struct Counter {  
    long cID;  
    long count;  
};  
#pragma keylist Counter cID
```

```
enum TemperatureScale {  
    CELSIUS,  
    FAHRENHEIT,  
    KELVIN  
};  
  
struct TempSensorType {  
    short id;  
    short roomid;  
    float temp;  
    float hum;  
    TemperatureScale scale;  
};  
#pragma keylist TempSensorType id roomid
```

Topic Keys Gotchas

- ▶ Keys are used to identify specific data “instances”
- ▶ If we want to make a parallel with OO then we could say that:
 - ▶ Keyless Topic as singletons, e.g. there is only one instance!
 - ▶ Keyed Topics identify a class of instances. Each instance is identified by a key value
 - ▶ Think at each different key value as really instantiating a new “object” in your system. That will avoid making mistakes in your keys assignment
- ▶ Never do something like this:

```
struct Counter {  
    long cID;  
    long count;  
};  
#pragma keylist Counter count
```

... As it will create a new topic instance for each ping you send thus consuming an unbounded amount of resources!

Compiling Topic Types

- ▶ Topic types have to be compiled with the DDS-provided IDL compiler
 - ▶ The compilation process will take care of generating code for
 - ▶ Strongly typed Reader and Writers
 - ▶ Type Serialization
 - ▶ When compiling a target language should be chosen, such as C/C++/Java/C#
-
- ▶ Example:

```
$ idlpp -S -l cpp -d gencxx ShapeType.idl
```

```
$ idlpp -S -l java -d genjava ShapeType.idl
```

Standalone mode

Target Language

Target Directory

Target File

IDL Compilation in SIMD

- ▶ SIMD provides a template makefile that you can use to compile your IDL files.
- ▶ The default language is C++ (as SIMD currently supports only C++)

Makefile.idl

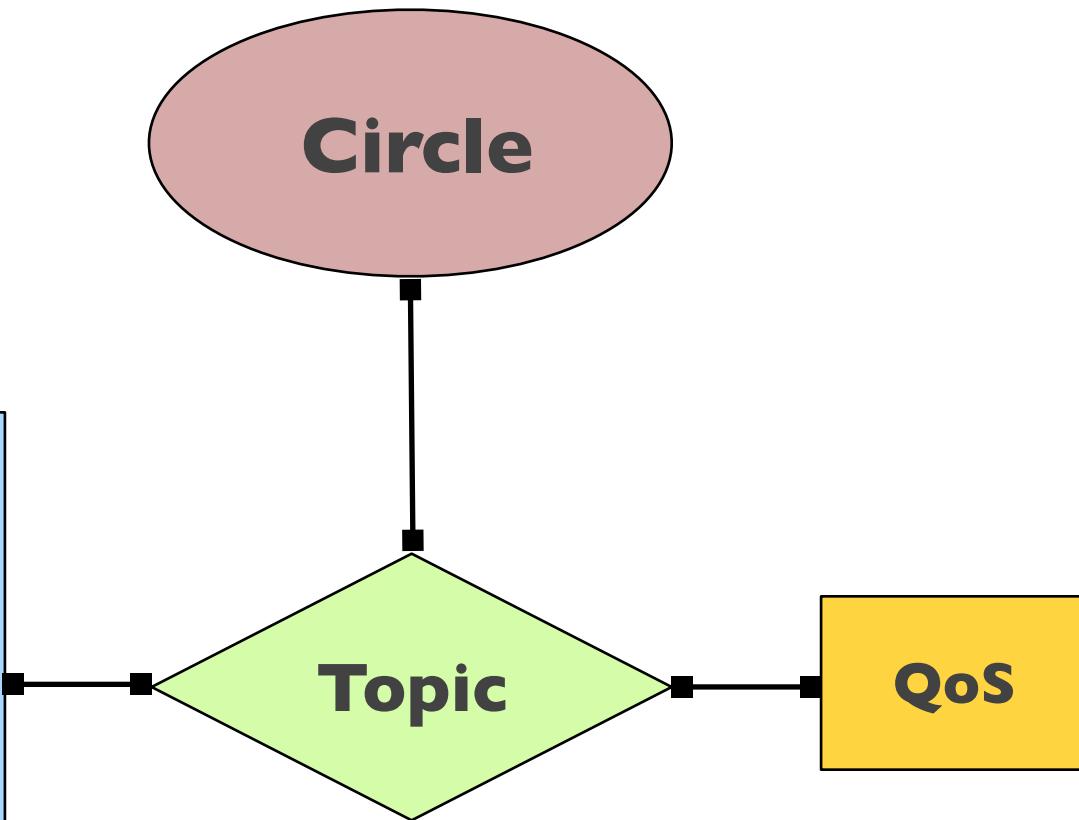
```
#-*=Makefile=-
include $(SIMD_HOME)/config/apps/Macros-idl.GNU

TARGET_IDL=ShapeType.idl

include $(SIMD_HOME)/config/apps/Rules-idl.GNU
```

Putting it all Together

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



Registering Topics with SIMD

- ▶ SIMD provides several constructors that allow to register a topic:

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

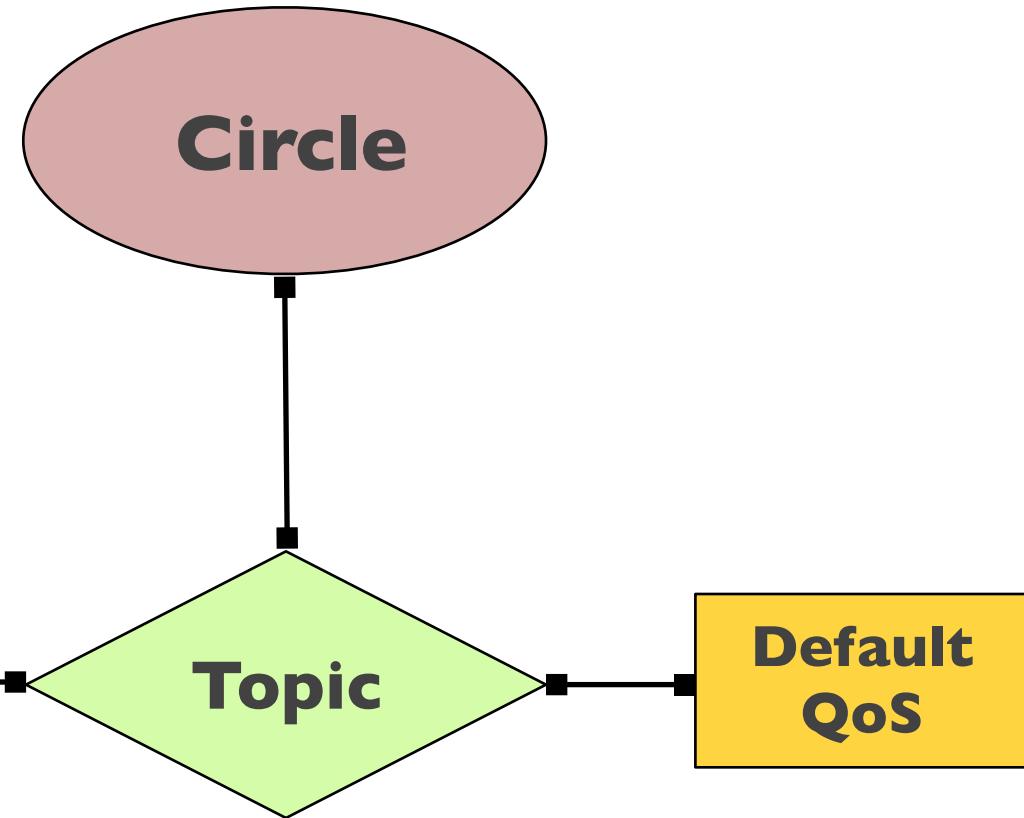
```
Topic(const std::string& name, const TopicQos& qos);
```

```
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 Circle Topic

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



```
dds::Topic<ShapeType> shape("Circle");
```

Topic Registration Gotchas

- ▶ Topics registration is idempotent as far as you register the topic in the same way from various applications.
- ▶ It is an error to try to register a topic with the same name but a different type.
- ▶ **Example:**

Application 1

```
dds::Topic<ShapeType> shape("Circle");
```

Application 2

```
dds::Topic<ShapeType> shape("Circle");
```

OK

```
dds::Topic<ShapeType> shape("Circle");
```

```
dds::Topic<AnotherType> shape("Circle");
```

Error

openSplice|DDS

Delivering Performance, Openness, and Freedom

Your will learn:

- What are DDS Partitions
- How to partitions work

Step II

Defining the Scope

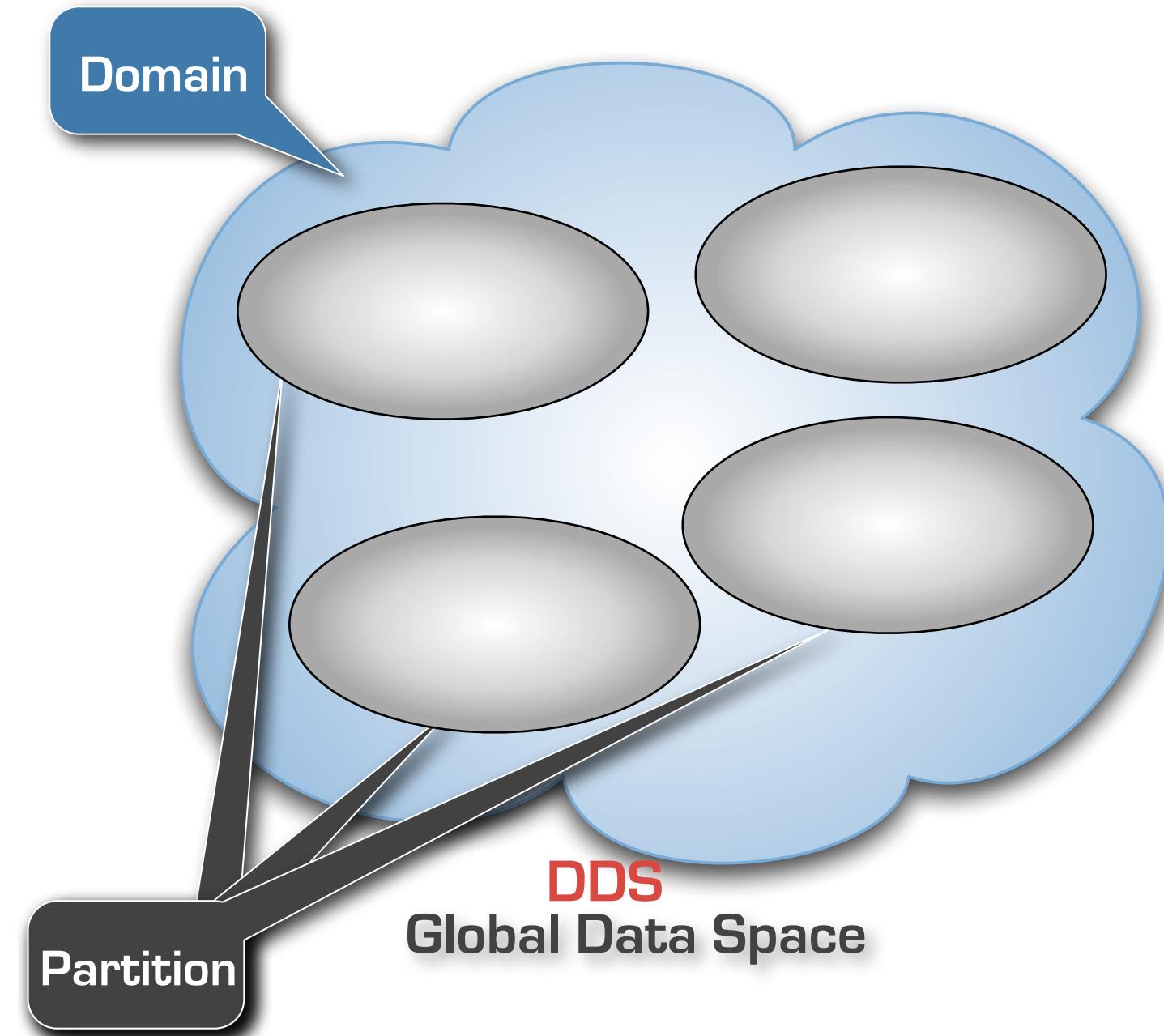
Domains and Partitions

Domain

- ▶ A Domain is one instance of the DDS Global Data Space
- ▶ DDS entities always belong to a specific domain

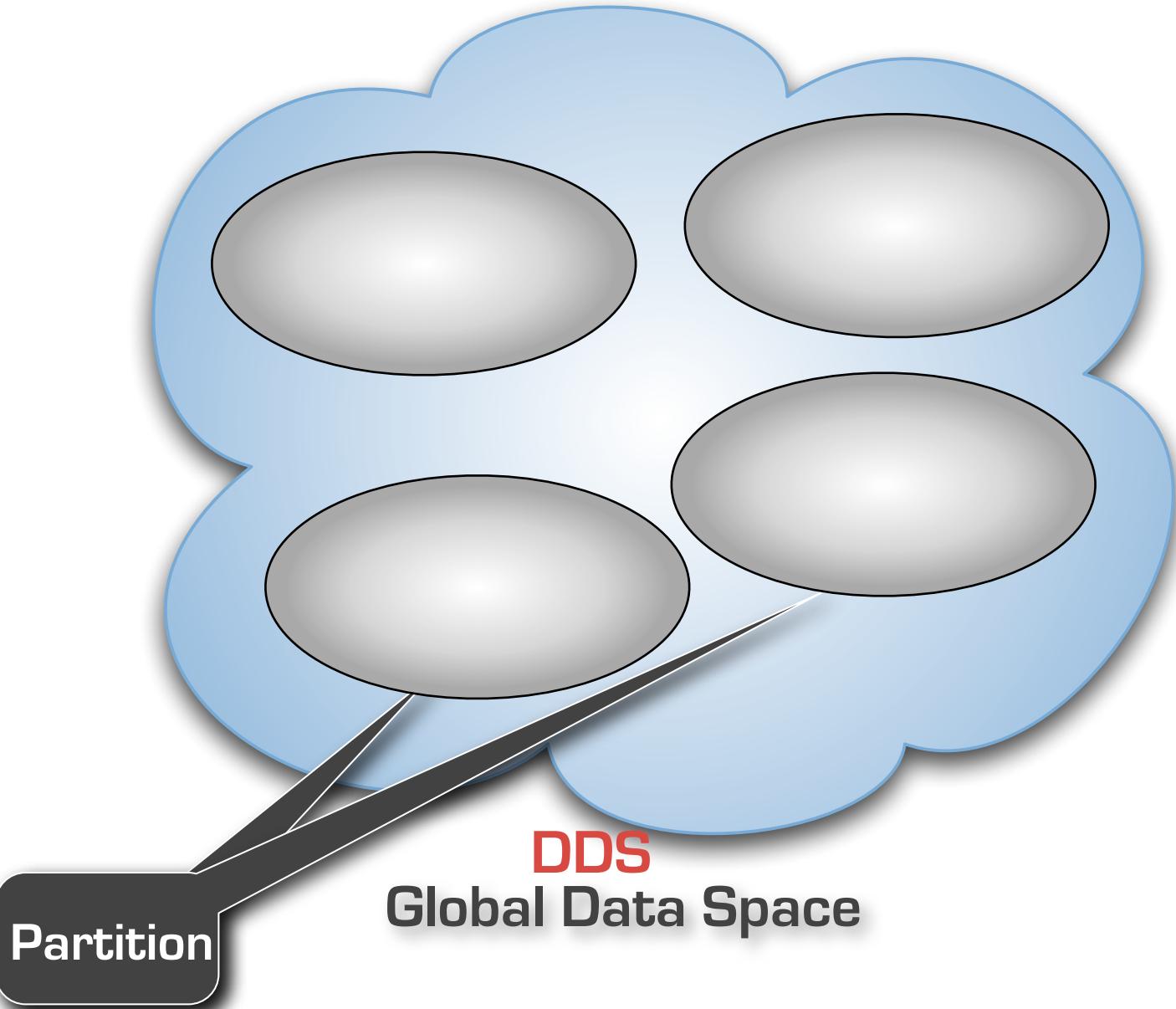
Partition

- ▶ A partition is a scoping mechanism provided by DDS to organize a partition



More about Partitions

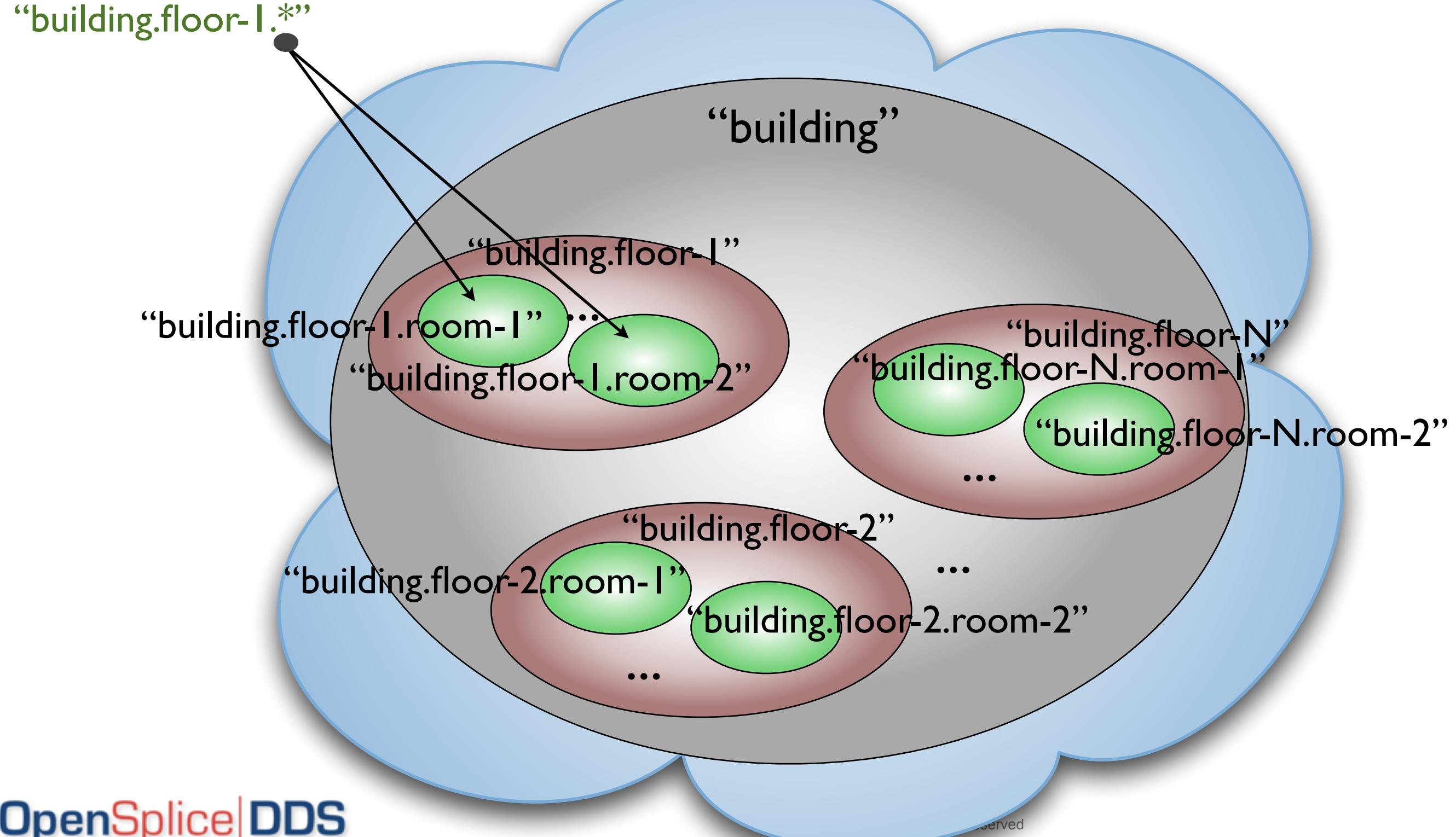
- ▶ Each partition is identified by a string, such as “sensor-data”, “log-data” etc.
- ▶ Read/Write access to a partition is gained by means of DDS Publisher/Subscribers
- ▶ Each Publisher/Subscriber can be provided with a list of Partitions name, which might as well include wildcards ,or generic regular expression, such as “*-data”



Partition as Namespaces

- ▶ Although DDS does not support explicit nesting of partitions, a powerful way of organizing your data is by always using a hierarchical “dotted” notation to describe them.
- ▶ For instance, for a building in which you are deploying the new temperature control system you might use a scheme such as “building.floor-level.room-number” for scoping the data that flows in each room.
 - ▶ **building.floor-2.room-10**
 - ▶ **building.floor-3.room-15**
- ▶ In this way, accessing the data for a specific floor can be done by using the partition expression “**building.floor-2.***”
- ▶ While the data for all the building is available via “**building.***”

Emulating Partition Nesting



Connecting to Partitions in SIMD

- ▶ SIMD provides two ways of connecting to partitions.
- ▶ A simple one is to bound the full runtime to a partition expression by passing a string to the Runtime class at construction time
- ▶ The other is to configure a specific Publisher/Subscriber with the relevant list of partitions

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

```
Publisher(const std::string& partition);  
Publisher(const std::string& partition, ::dds::DomainParticipant dp);  
Publisher(const ::dds::PublisherQos& qos, ::dds::DomainParticipant dp);
```

```
Subscriber(const std::string& partition);  
Subscriber(const std::string& partition, ::dds::DomainParticipant dp);  
Subscriber(const ::dds::SubscriberQos& qos, ::dds::DomainParticipant dp);
```

openSplice|DDS

Delivering Performance, Openness, and Freedom

Your will learn:

- What is a Data Writer
- How to Create a Data Writer
- How to write Data

Step III

Producing the Data

Writing Data in SIMD

- ▶ Writing data with SIMD takes two steps.
- ▶ First you have to create the DataWriter by using the proper constructor (this depends on the level of customization you require)
- ▶ Then, you'll have to decide how you want to write the data

Creating a DataWriter

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

```
template <typename T>
class dds::pub::DataWriter : public dds::core::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);
```

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

Writing “Circling” Circle Samples

```
1  dds::Topic<ShapeType> shape(opt.topic);
2  dds::DataWriter<ShapeType> dw(shape);
3
4  float delta = 0.1F;
5  int x;
6  int y;
7  ShapeType st;
8  st.color = DDS::string_dup(opt.color.c_str());
9  st.shapesize = opt.size;
10
11
12 long long sec = 0;
13 long long nsec = opt.period;
14
15 // nsec has to be <= 999999999
16 if (nsec >= ONE_SECOND) {
17     sec = nsec / ONE_SECOND;
18     nsec = opt.period - (sec * ONE_SECOND);
19 }
20
21 timespec period = {
22     sec,
23     nsec
24 };
```

```
26     float theta = 0;
27     timespec leftover;
28     for (int i = 0; i < opt.samples; ++i) {
29         x = opt.X0 + static_cast<int>(opt.radius * cos(theta));
30         y = opt.Y0 + static_cast<int>(opt.radius * sin(theta));
31         theta += delta;
32         st.x = x;
33         st.y = y;
34         dw.write(st);
35         nanosleep(&period, &leftover);
36     }
37 }
```

openSplice|DDS

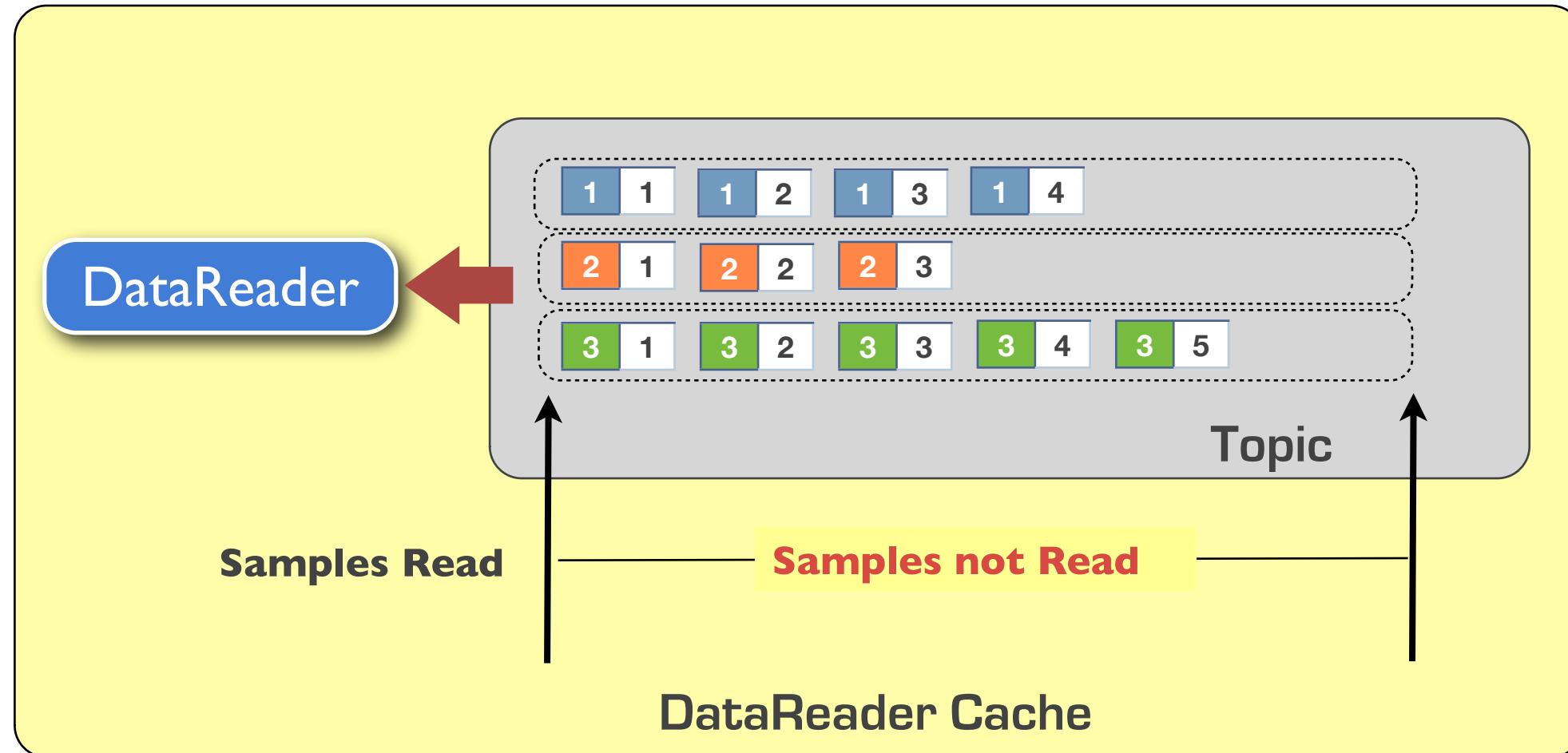
Delivering Performance, Openness, and Freedom

Your will learn:

- Reading vs Taking data
- Sample State
- How to Create a Data Reader
- How to read/take data

Step IV
Consuming Data

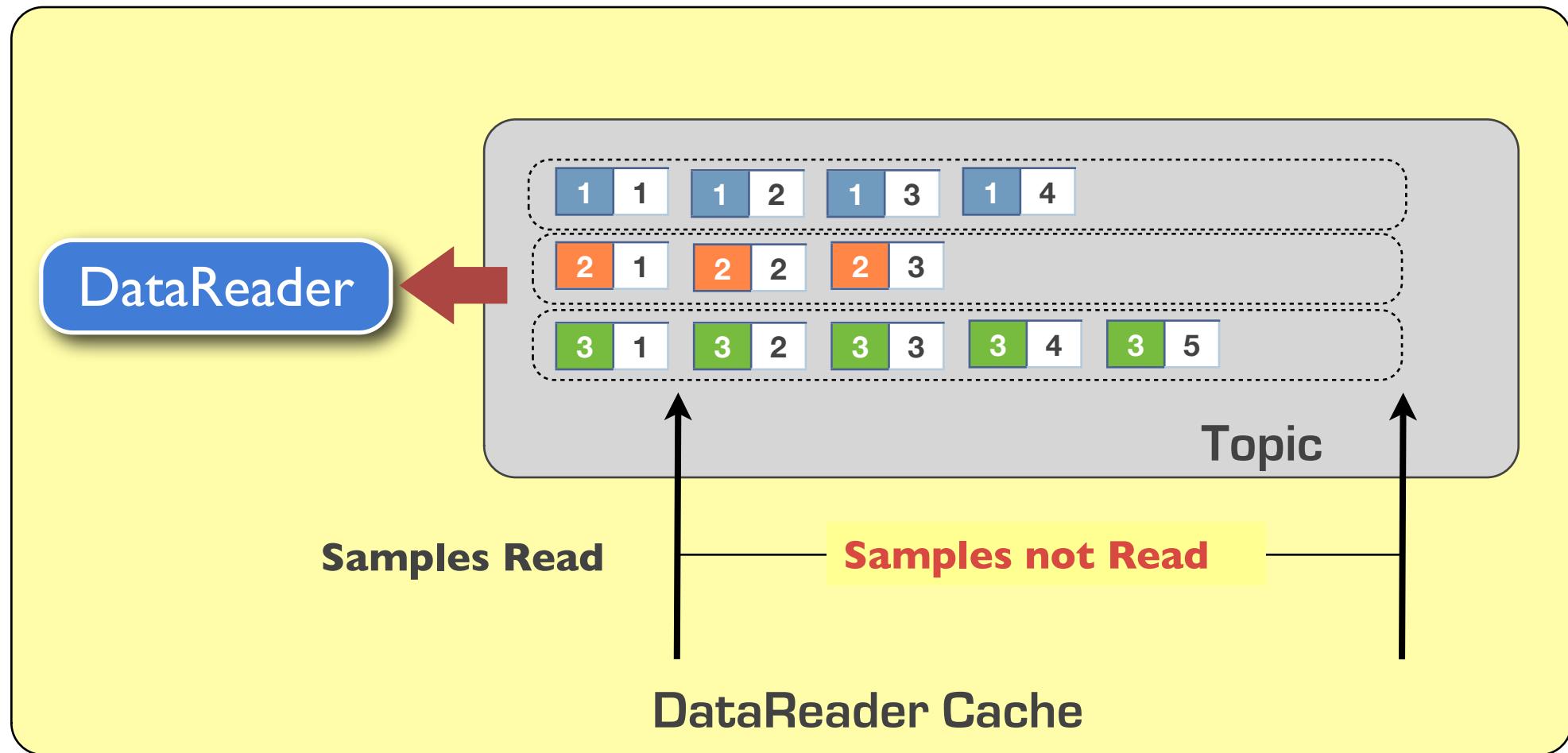
Reading Samples



- ▶ Read iterates over the available sample instances
- ▶ **Samples are not removed from the local cache as result of a read**
- ▶ Read samples can be read again, by accessing the cache with the proper options (more later)

```
struct Counter {  
    int cID;  
    int count;  
};  
#pragma keylist Counter cID
```

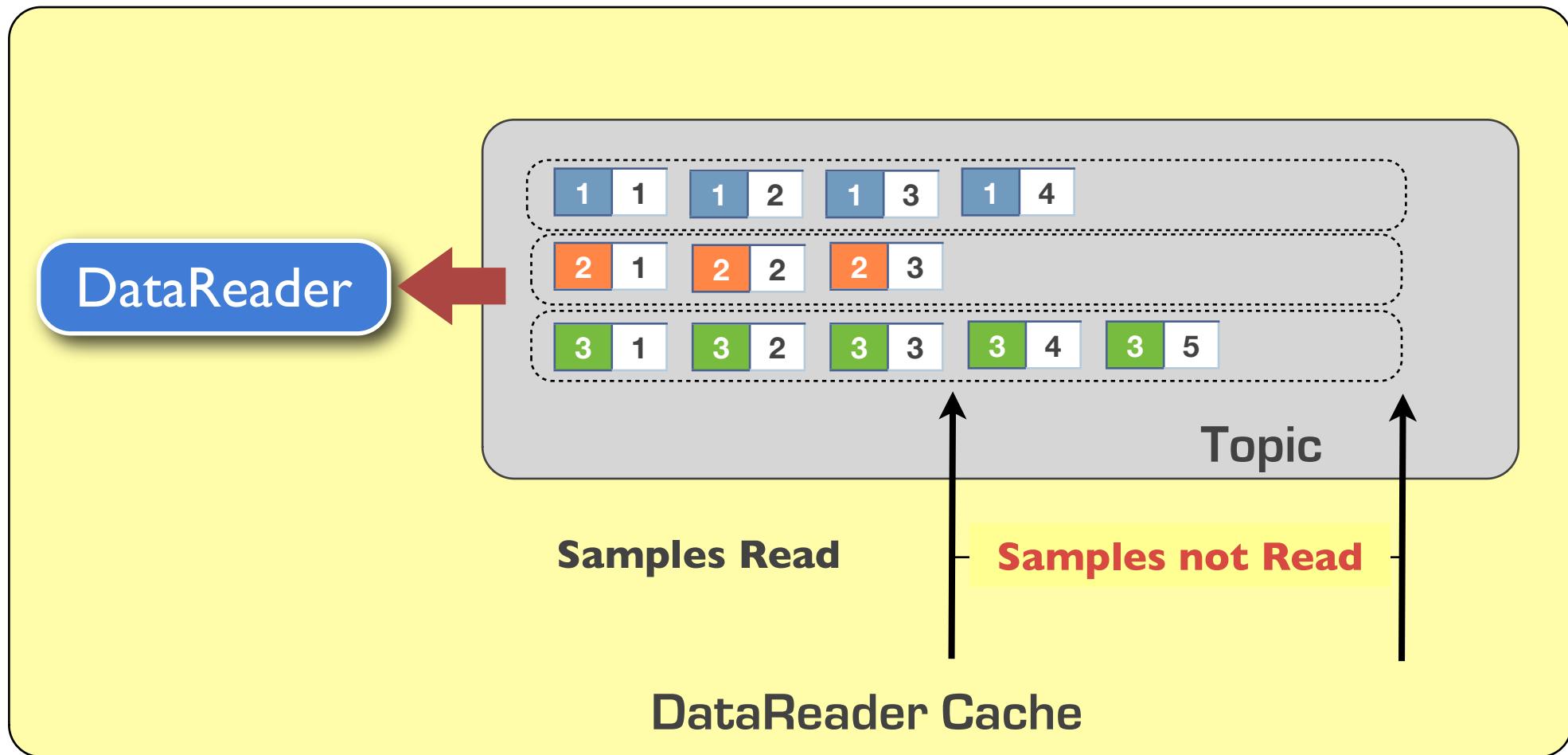
Reading Samples



- ▶ Read iterates over the available sample instances
- ▶ **Samples are not removed from the local cache as result of a read**
- ▶ Read samples can be read again, by accessing the cache with the proper options (more later)

```
struct Counter {  
    int cID;  
    int count;  
};  
#pragma keylist Counter cID
```

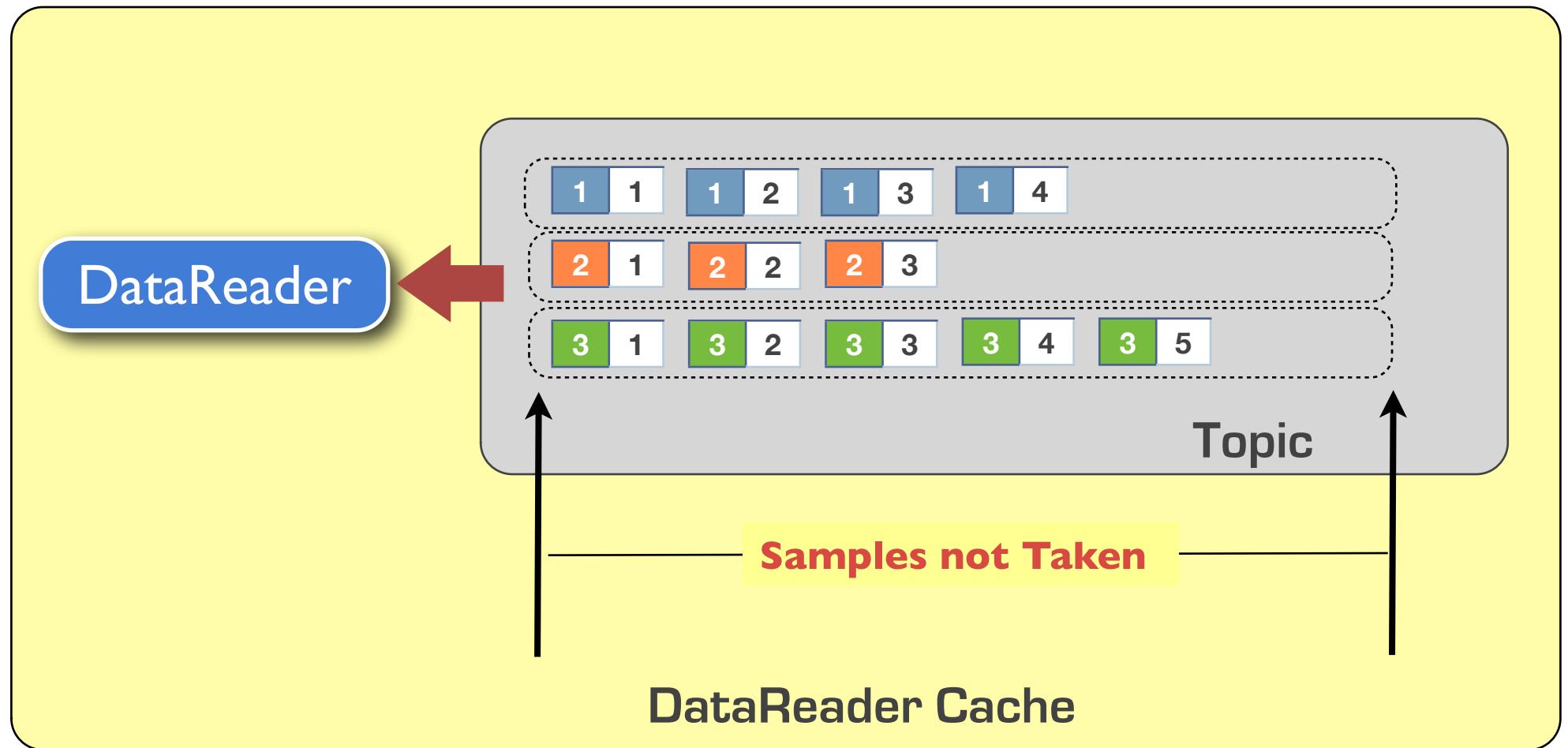
Reading Samples



- ▶ Read iterates over the available sample instances
- ▶ **Samples are not removed from the local cache as result of a read**
- ▶ Read samples can be read again, by accessing the cache with the proper options (more later)

```
struct Counter {  
    int cID;  
    int count;  
};  
#pragma keylist Counter cID
```

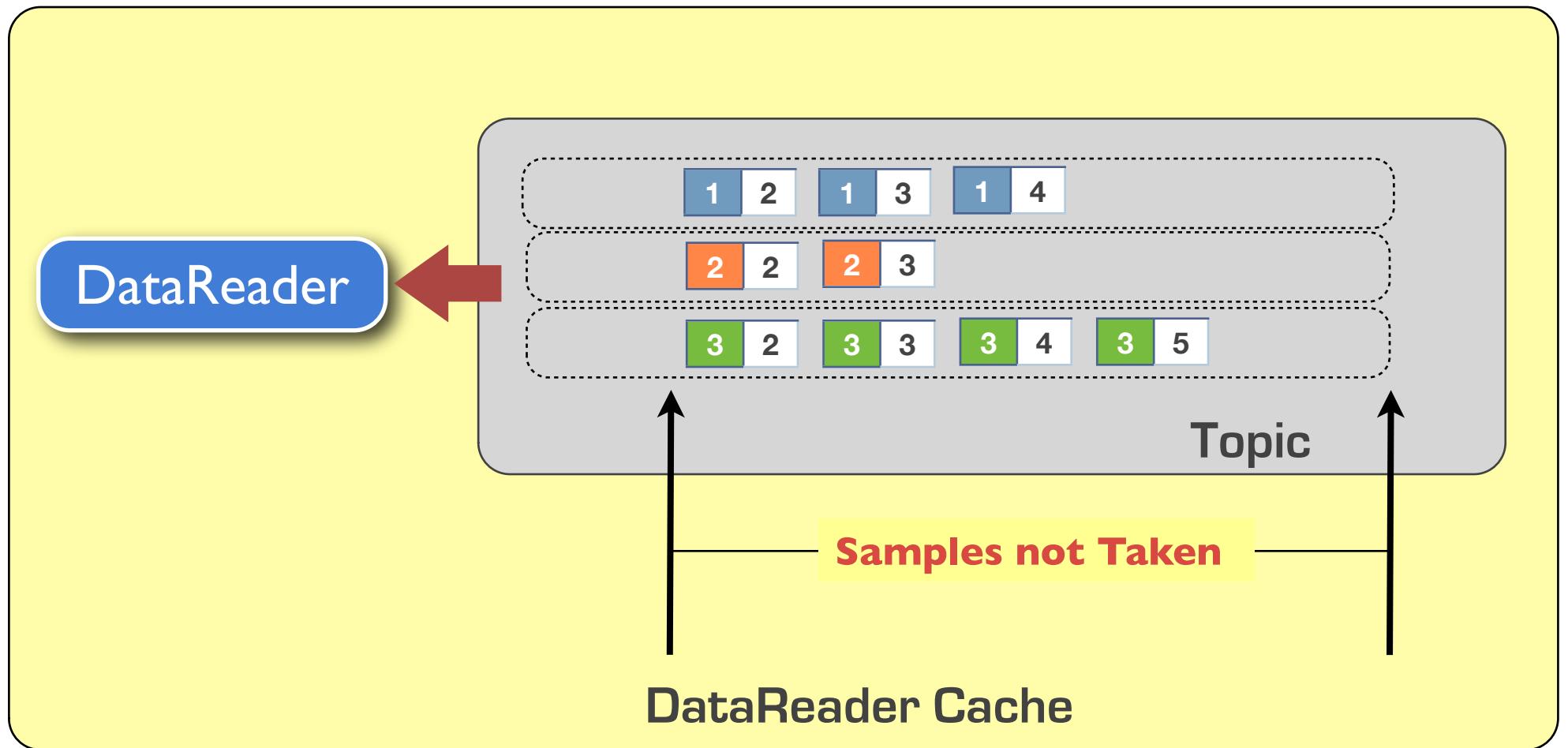
Taking Samples



- ▶ Take iterates over the available sample instances
- ▶ Taken Samples are **removed from the local cache** as result of a take

```
struct Counter {  
    int cID;  
    int count;  
};  
#pragma keylist Counter cID
```

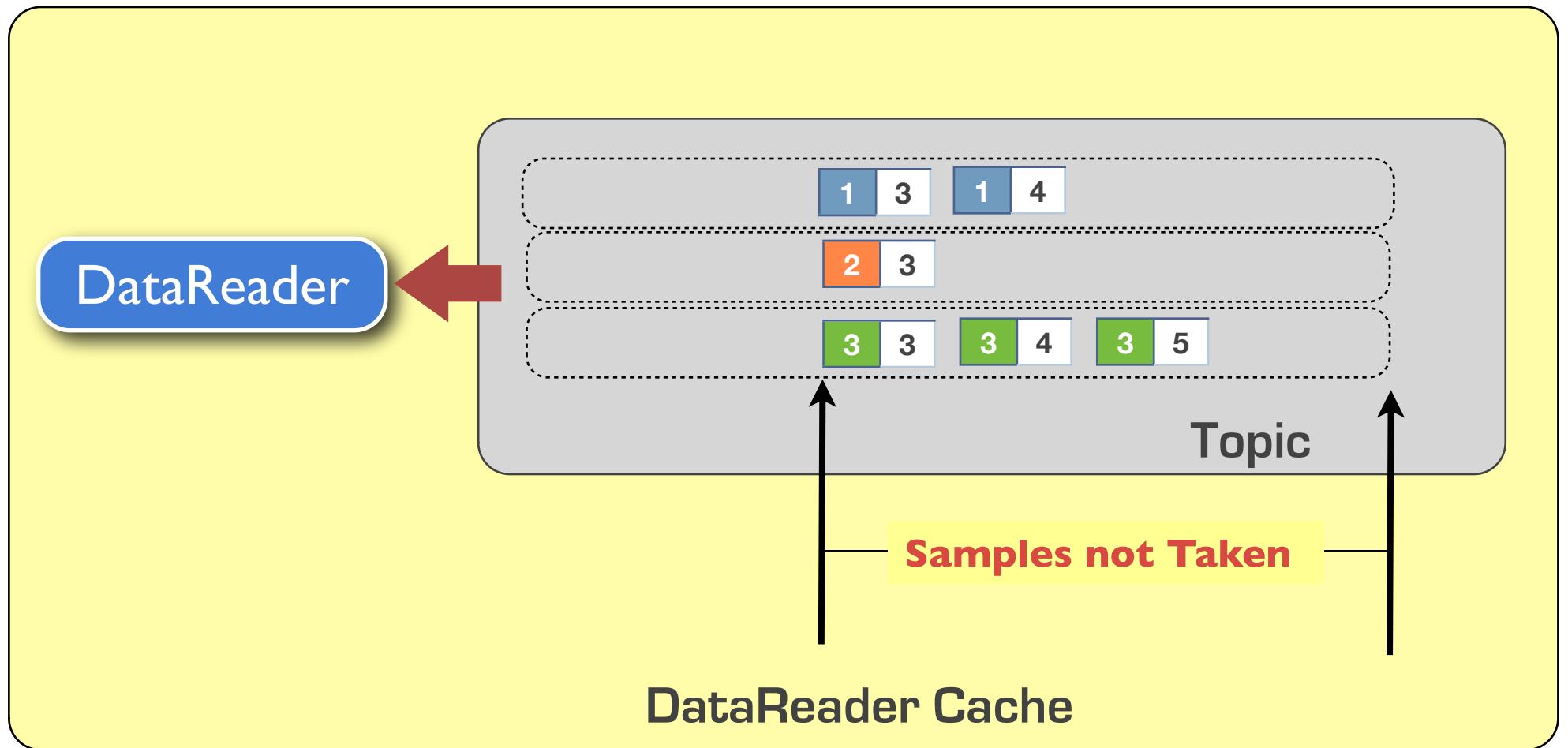
Taking Samples



- ▶ Take iterates over the available sample instances
- ▶ Taken Samples are **removed from the local cache** as result of a take

```
struct Counter {  
    int cID;  
    int count;  
};  
#pragma keylist Counter cID
```

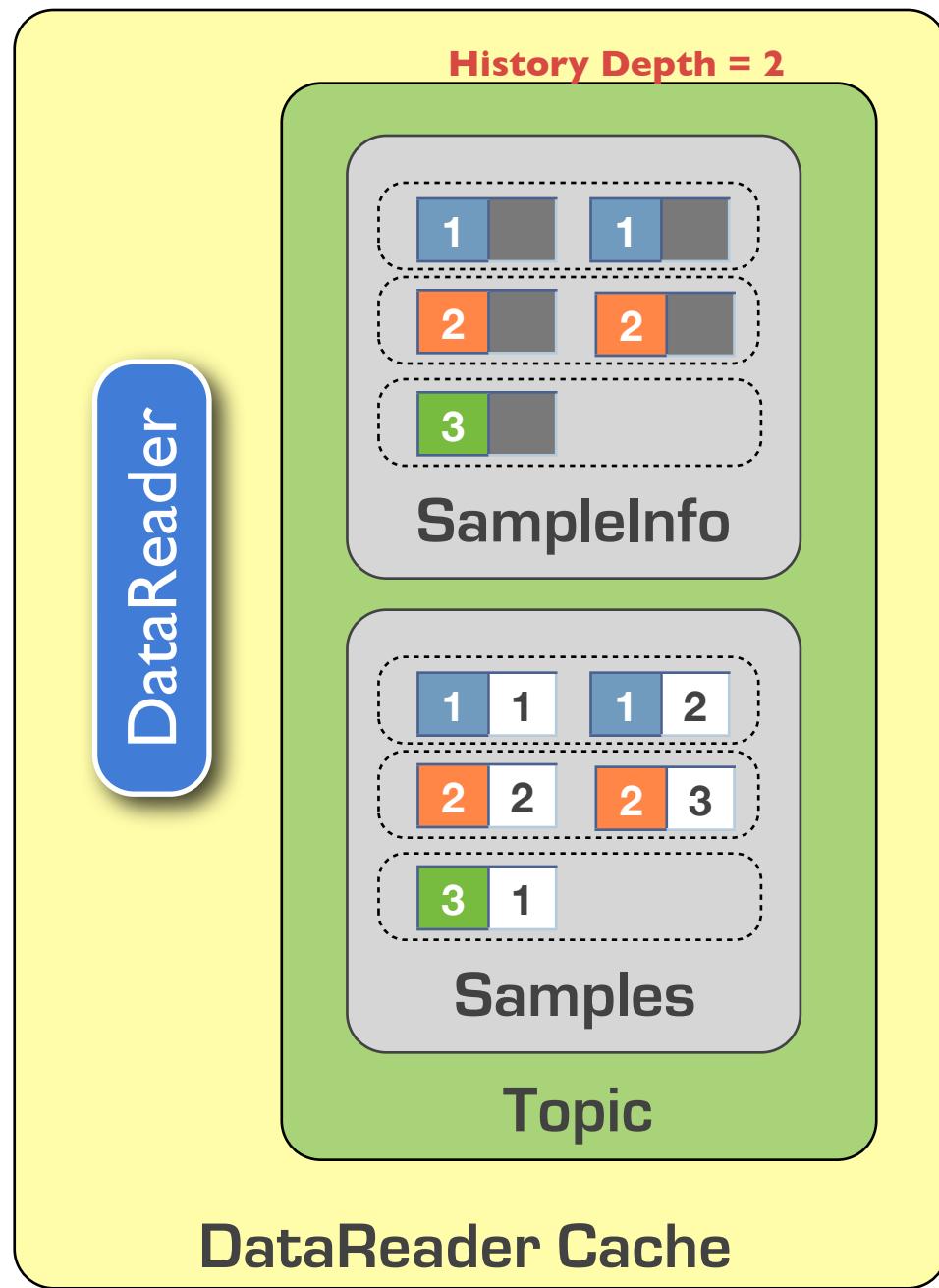
Taking Samples



- ▶ Take iterates over the available sample instances
- ▶ Taken Samples are **removed from the local cache** as result of a take

```
struct Counter {  
    int cID;  
    int count;  
};  
#pragma keylist Counter cID
```

Sample, Instance and View States



- ▶ Along with data samples, DataReaders are provided with state information allowing to detect relevant transitions in the life-cycle of data as well as data writers
- ▶ **Sample State (READ | NOT_READ)**: Determines whether a sample has already been read by this DataReader or not.
- ▶ **Instance State (ALIVE, NOT_ALIVE, DISPOSED)**. Determines whether (1) writer exist for the specific instance, or (2) no matched writers are currently available, or (3) the instance has been disposed
- ▶ **View State (NEW, NOT_NEW)**. Determines whether this is the first sample of a new (or re-born) instance

Application / DDS Coordination

DDS provides three main mechanism for exchanging information with the application

- ▶ **Polling.** The application polls from time to time for new data or status changes. The interval might depend on the kind of applications as well as data
- ▶ **WaitSets.** The application registers a WaitSet with DDS and waits (i.e. is suspended) until one of the specified events has happened.
- ▶ **Listeners.** The application registers a listener with a specific DDS entity to be notified when relevant events occur, such as state changes or

Reading Data with SIMD

```
/**  
 * Reads all new samples from any view state and alive instances. Notice  
 * that this call is intended to loan the <code>samples</code> as  
 * well as the <code>infos</code> containers, thus will require a  
 * return_loan.  
 */  
DDS::ReturnCode_t read(TSeq& samples, DDS::SampleInfoSeq& infos)  
  
/**  
 * Reads at most <code>max_samples</code> samples that have not been  
 * read yet from all vies and alive instances.  
 */  
DDS::ReturnCode_t read(TSeq& samples, long max_samples)  
  
/**  
 * Most generic <code>read</code> exposing all the knobs provided by  
 * the OMG DDS API.  
 */  
DDS::ReturnCode_t  
read(TSeq& samples, DDS::SampleInfoSeq& infos, long max_samples,  
     DDS::SampleStateMask samples_state, DDS::ViewStateMask views_state,  
     DDS::InstanceStateMask instances_state)  
  
DDS::ReturnCode_t  
return_loan(TSeq& samples, DDS::SampleInfoSeq& infos);
```

Taking Data with SIMD

```
/**  
 * Reads all new samples from any view state and alive instances. Notice  
 * that this call is intended to loan the <code>samples</code> as  
 * well as the <code>infos</code> containers, thus will require a  
 * return_loan.  
 */  
DDS::ReturnCode_t take(TSeq& samples, DDS::SampleInfoSeq& infos)  
  
/**  
 * Reads at most <code>max_samples</code> samples that have not been  
 * read yet from all vies and alive instances.  
 */  
DDS::ReturnCode_t take(TSeq& samples, long max_samples)  
  
/**  
 * Most generic <code>read</code> exposing all the knobs provided by  
 * the OMG DDS API.  
 */  
DDS::ReturnCode_t  
take(TSeq& samples, DDS::SampleInfoSeq& infos, long max_samples,  
     DDS::SampleStateMask samples_state, DDS::ViewStateMask views_state,  
     DDS::InstanceStateMask instances_state)
```

WaitSets in SIMD

- ▶ SIMD provides a strongly typed WaitSet that supports automatic dispatching to functors
- ▶ The best way of understanding SIMD waitsets is to look at an example:

```
1 class ShapeUpdateHandler {  
2     public:  
3         ShapeUpdateHandler() { }  
4         ~ShapeUpdateHandler() { }  
5     public:  
6         void operator()(dds::DataReader<ShapeType>& reader) {  
7             ShapeTypeSeq data;  
8             DDS::SampleInfoSeq status;  
9             reader.read(data, status);  
10            for (int i = 0; i < data.length(); ++i)  
11                std::cout << std::dec << "">>>[DR]: " << data[i] << std::endl;  
12  
13            reader.return_loan(data, status);  
14        }  
15    };
```

```
1     ShapeUpdateHandler handler;  
2     dds::ActiveReadCondition arc =  
3     dr.create_readcondition(handler);  
4     ::dds::ActiveWaitSet ws;  
5     DDS::ReturnCode_t retc = ws.attach(arc);  
6  
7     while (read_samples_ < opt.samples) {  
8         ws.dispatch();  
9     }
```

Listeners in SIMD

- ▶ SIMD provides a strongly typed Listeners based on the Signals/Slots patterns
- ▶ The best way of understanding SIMD Listeners is to look at an example...

```
1 class ShapeUpdateHandler {
2     public:
3         ShapeUpdateHandler(int samples, boost::barrier& barrier)
4             : samples_(samples),
5              barrier_(barrier)
6         { }
7
8         ~ShapeUpdateHandler() { }
9
10    public:
11        void handle_data(dds::DataReader<ShapeType>& reader)
12        {
13            ShapeTypeSeq data;
14            DDS::SampleInfoSeq status;
15            reader.read(data, status);
16            samples_ -= data.length();
17            for (int i = 0; i < data.length(); ++i)
18                std::cout << std::dec << "DR >> " << data[i] << std::endl;
19
20            // Notice it is OK to call the barrier_.wait() here since it is
21            // not really going to wait but simply reach the barrier count and
22            // exit the program.
23            if (samples_ <= 0)
24                barrier_.wait();
25        }
26
27        void handle_liveliness_change(dds::DataReader<ShapeType>& reader,
28                                     const DDS::LivelinessChangedStatus& status)
29        {
30            std::cout << status << std::endl;
31        }
32
33    private:
34        int samples_;
35        boost::barrier& barrier_;
36    };
```

```
1 | boost::barrier completion_barrier(2);
2 | ShapeUpdateHandler handler(opt.samples, completion_barrier);
3 |
4 | dds::sigcon_t con_data =
5 | dr.on_data_available_signal_connect(boost::bind(&ShapeUpdateHandler::handle_data,
6 |                                                 &handler,
7 |                                                 _1));
8 |
9 |
10 | dds::sigcon_t con_liv =
11 | dr.on_liveliness_changed_signal_connect(boost::bind(&ShapeUpdateHandler::handle_liveliness_change,
12 |                                                 &handler,
13 |                                                 _1,
14 |                                                 _2));
15 |
16 |
17 | completion_barrier.wait();
18 | con_data.disconnect();
19 | con_liv.disconnect();
```

openSplice|DDS

Delivering Performance, Openness, and Freedom

Step V

Compile and Run...

What You've Learned today

- ▶ Defining Topics and Topic Types
- ▶ Scoping Information with Partitions
- ▶ Writing Data
- ▶ Reading (Taking) data with Waitsets and Listeners
- ▶ Writing an example that demonstrate all of the above

What I'll Cover Next Time

- ▶ Content Filtered Topics and Queries
- ▶ QoS and the Request vs. Offered Model
- ▶ Setting QoS on DDS Entities
- ▶ Tuning OpenSplice DDS Configuration

Online Resources



Delivering Performance, Openness, and Freedom

- * <http://www.opensplice.com/>
- * <mailto:openspicedds@prismtech.com>



- * <http://bit.ly/1Sreg>



- * <http://www.youtube.com/OpenSpliceTube>



- * <http://www.slideshare.net/angelo.corsaro>



- * <http://twitter.com/acorsaro/>



- * <http://opensplice.blogspot.com>