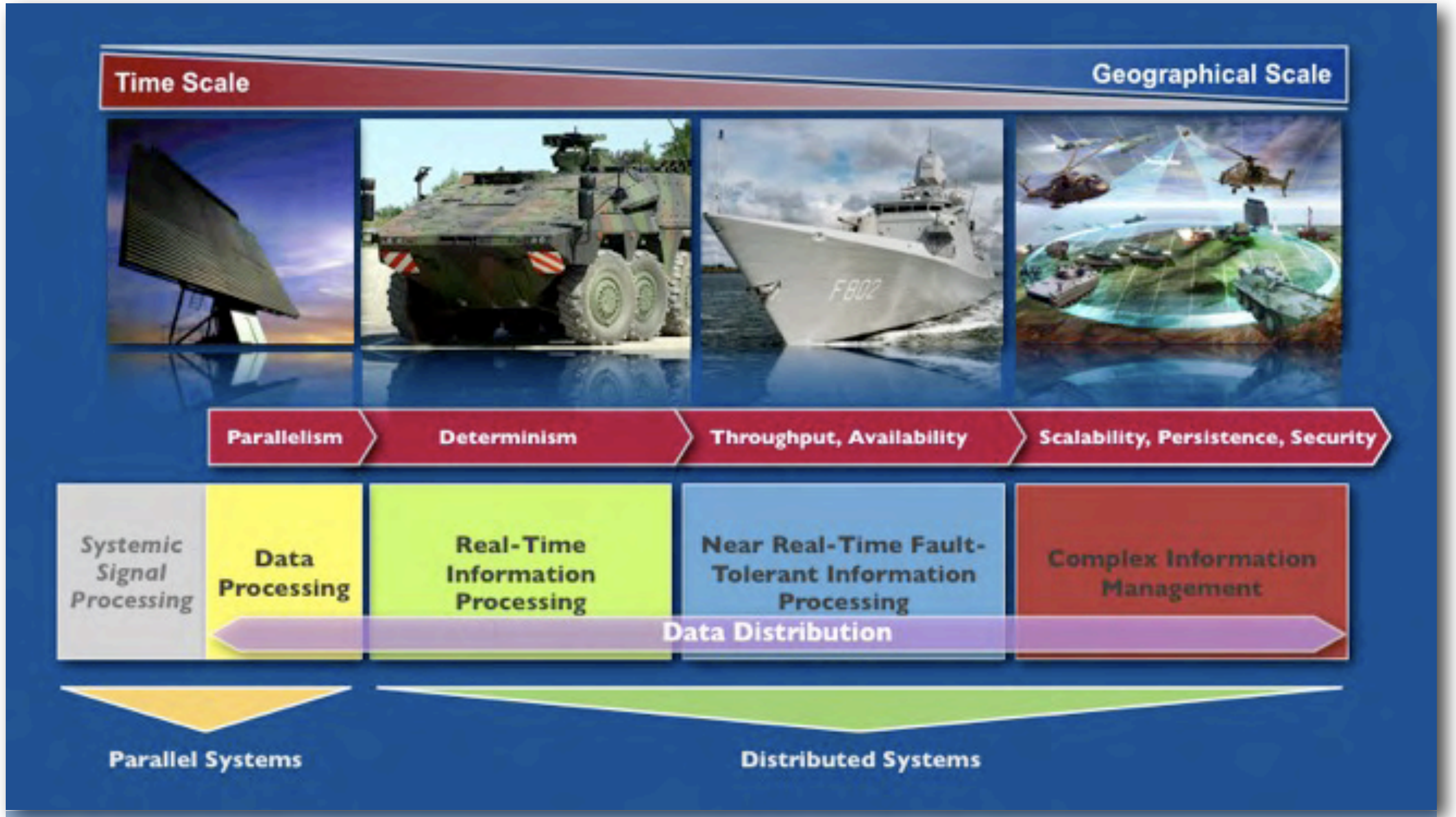
Angelo Corsaro, Ph.D.
Chief Technology Officer
OMG DDS SIG Co-Chair
angelo.corsaro@prismtech.com

PrismTech
Powering Netcentricity

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.

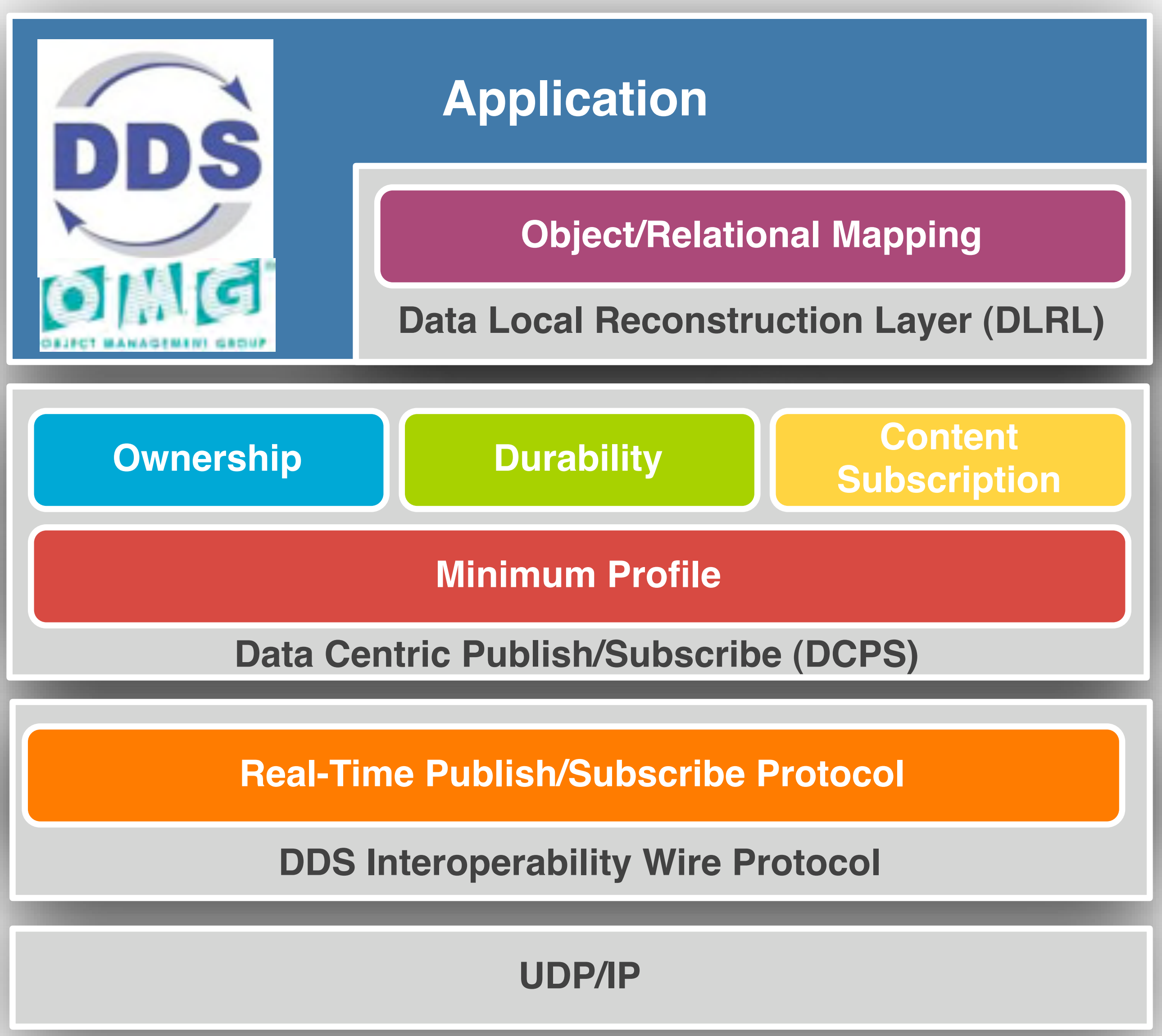
The OMG Data Distribution Service (DDS)

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

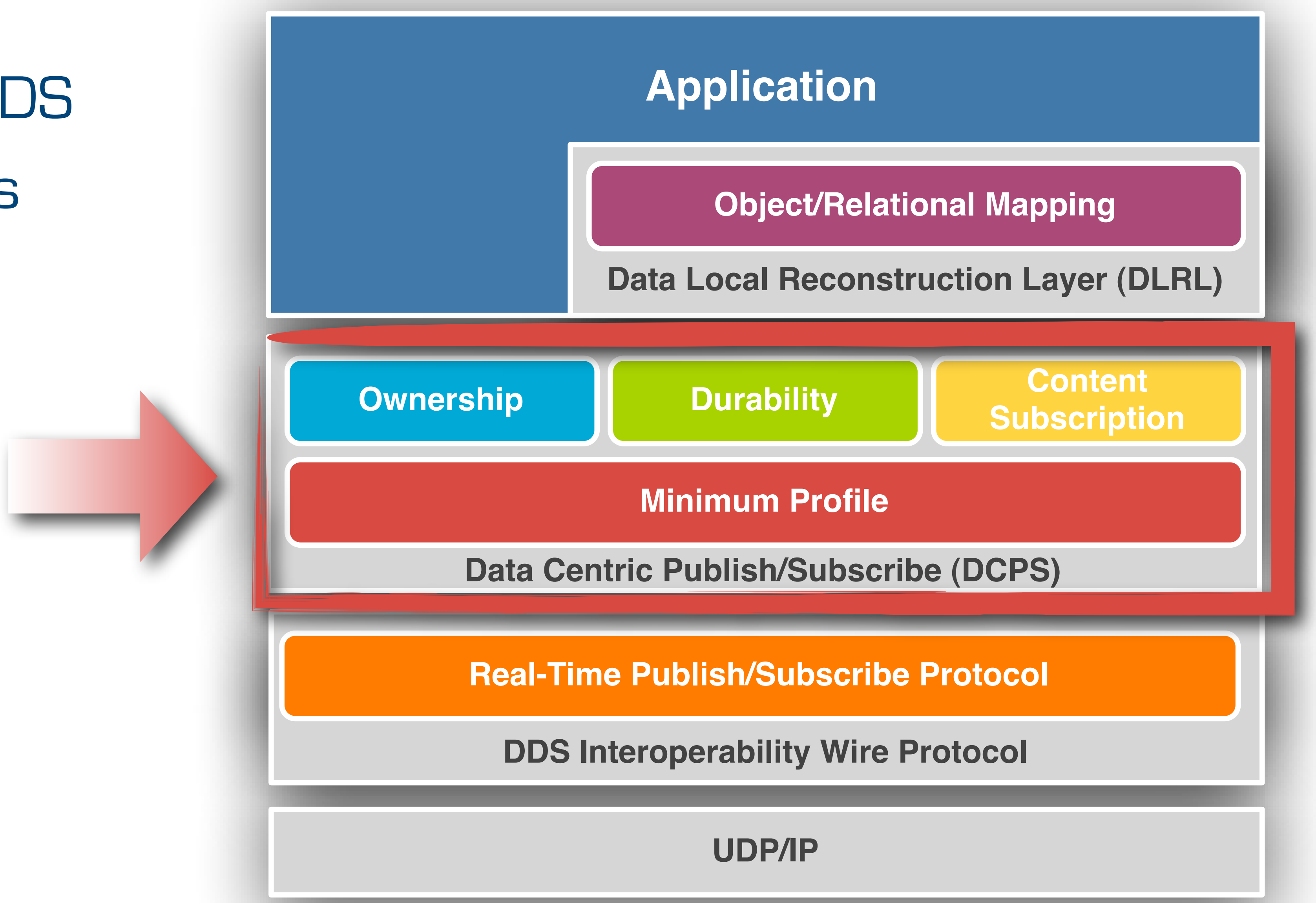
- The Tutorial will cover the DCPS layer of DDS
- It will give you enough details and examples to make sure that you can get started writing DDS applications

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

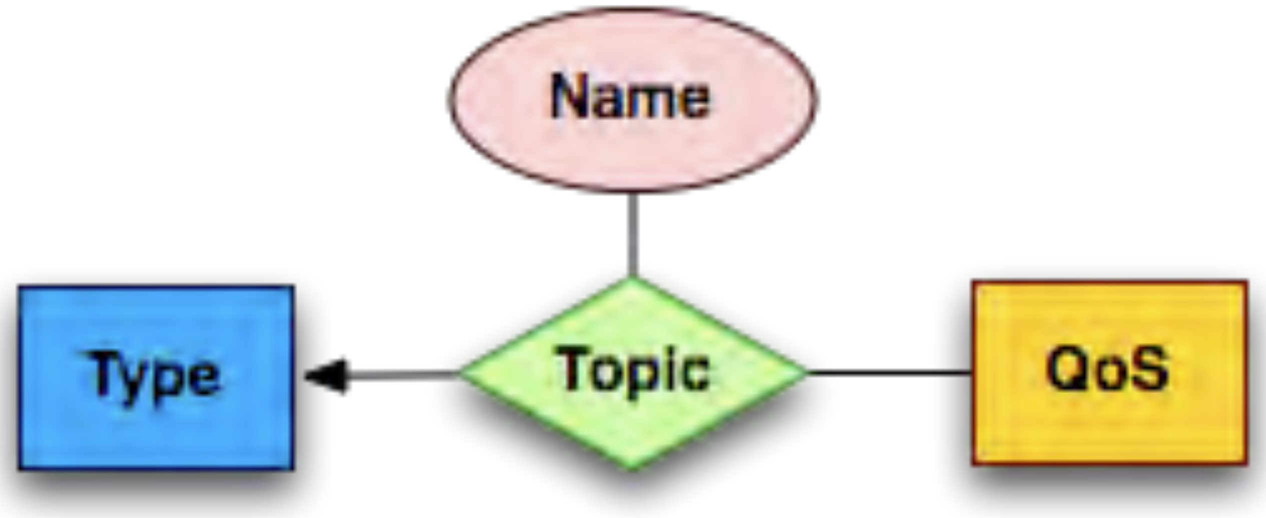
Step I

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”
- It 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
```

-S: Standalone mode
-l java: Target Language
-d genjava: Target Directory
ShapeType.idl: 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

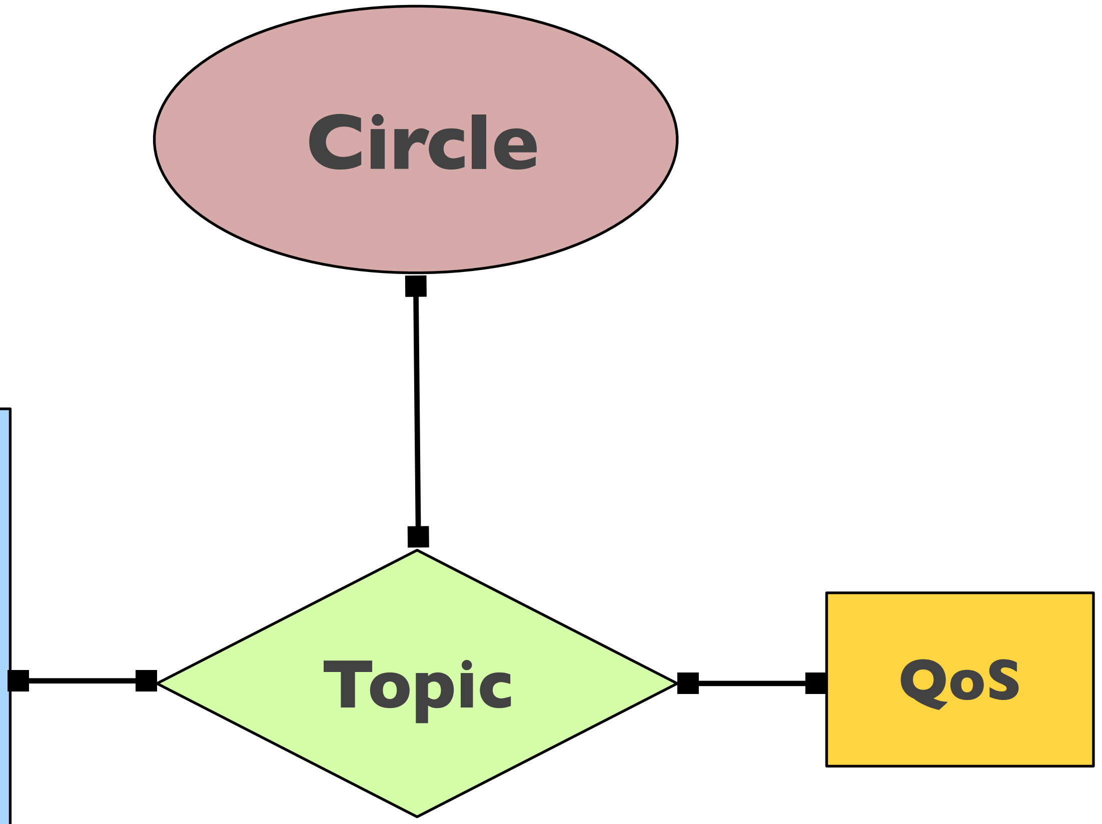
```
#-*-Makefile-*-
include $(SIMD_HOME)/config/apps/Macros-idl.GNU

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:

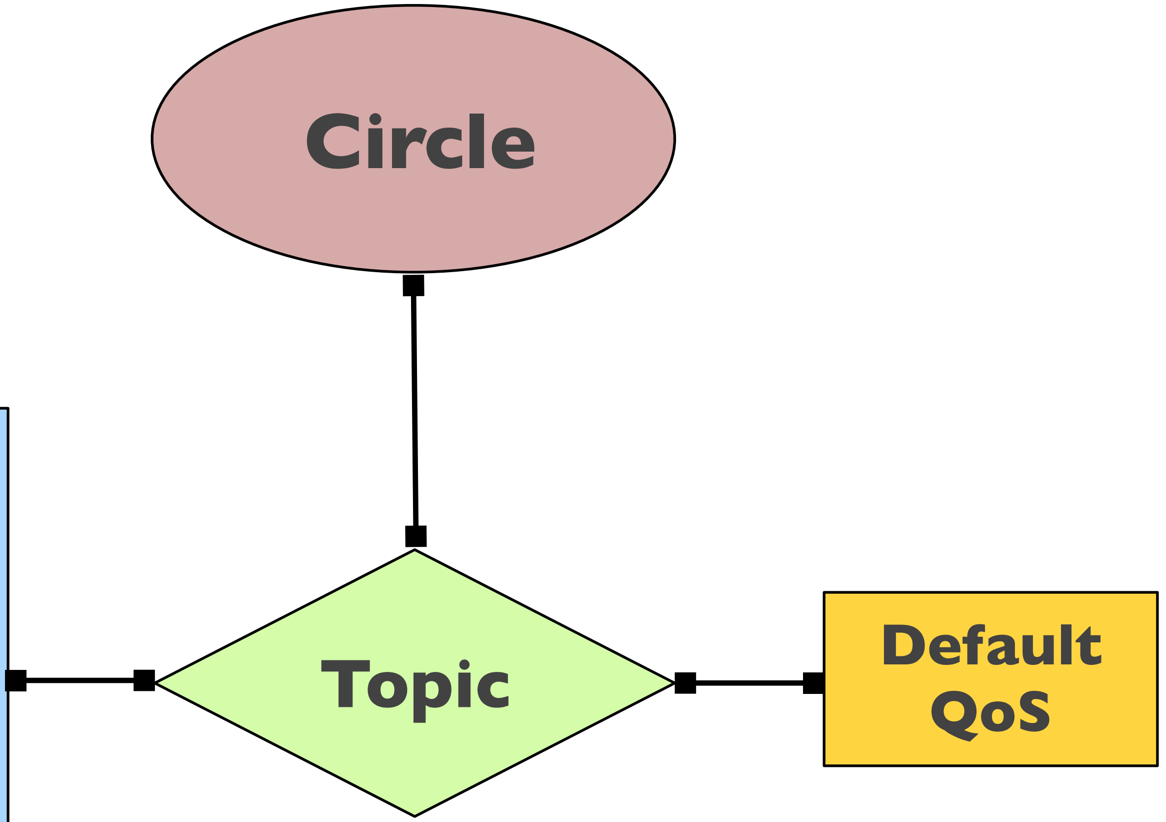
```
Topic(const std::string& name);

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");
```

OpenSplice | DDS

PRISMTECH

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	Application 2	
`dds::Topic<ShapeType> shape("Circle");`	`dds::Topic<ShapeType> shape("Circle");`	**OK**
`dds::Topic<ShapeType> shape("Circle");`	`dds::Topic<AnotherType> shape("Circle");`	**Errror**

OpenSplice|DDS

PRISMTECH

OpenSplice | DDS

Delivering Performance, Openness, and Freedom

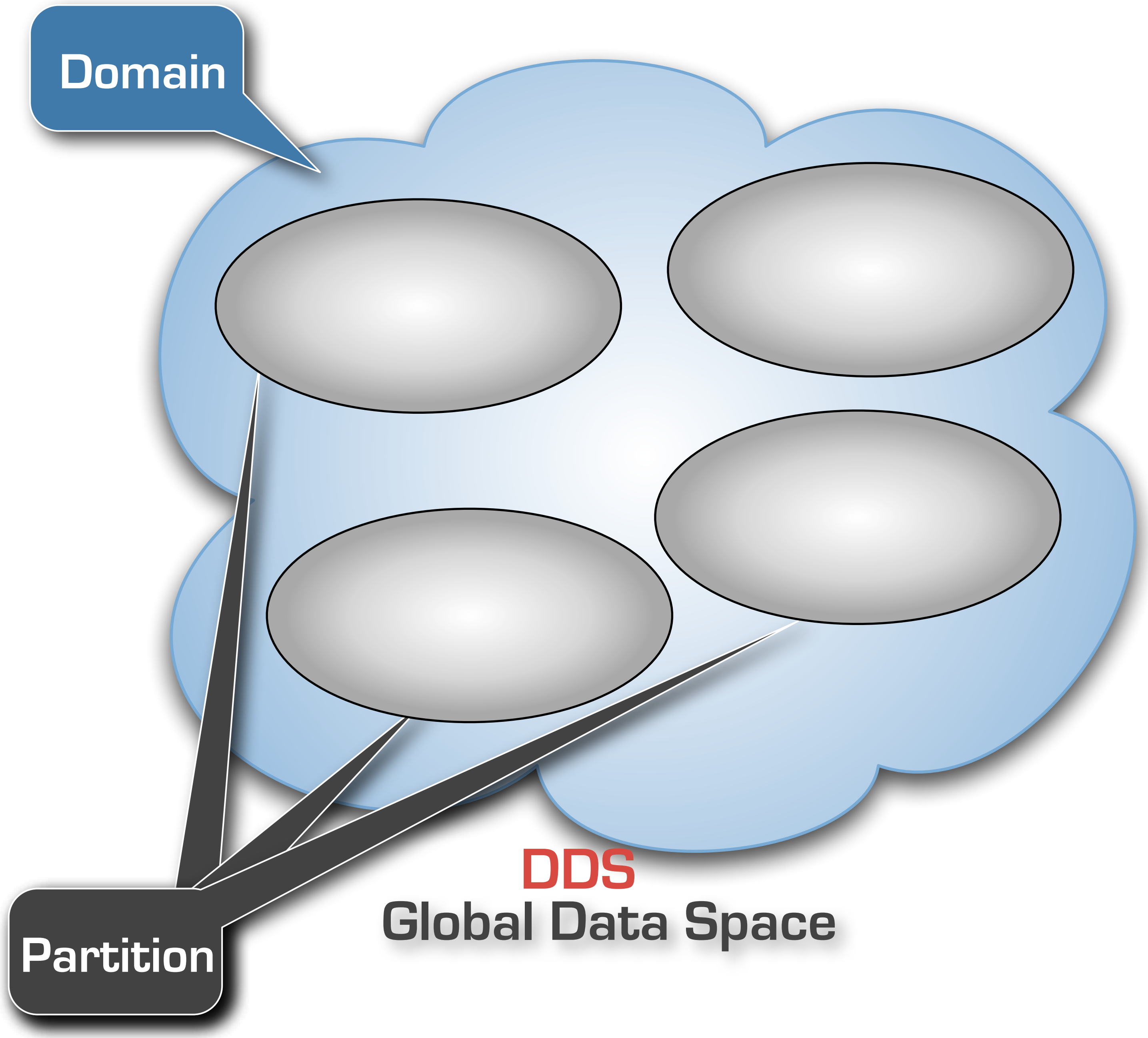
Your will learn:

- What are DDS Partitions
- How to partitions work

Step II

Defining the Scope

Domains and Partitions



Domain

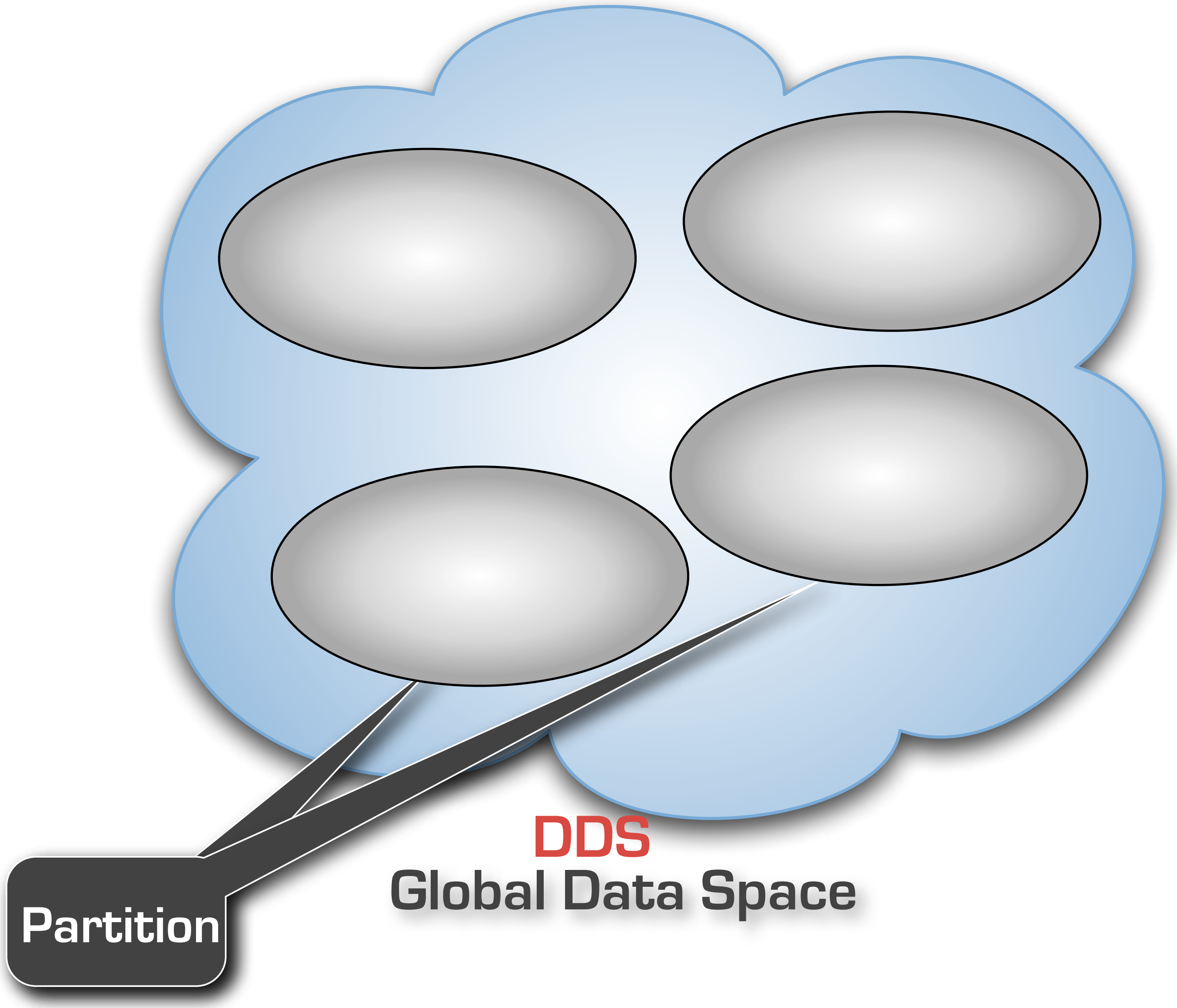
- A Domain is one instance of the DDS Global Data Space
- DDS entities always belong to a specific domain

Partition

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

More about Partitions

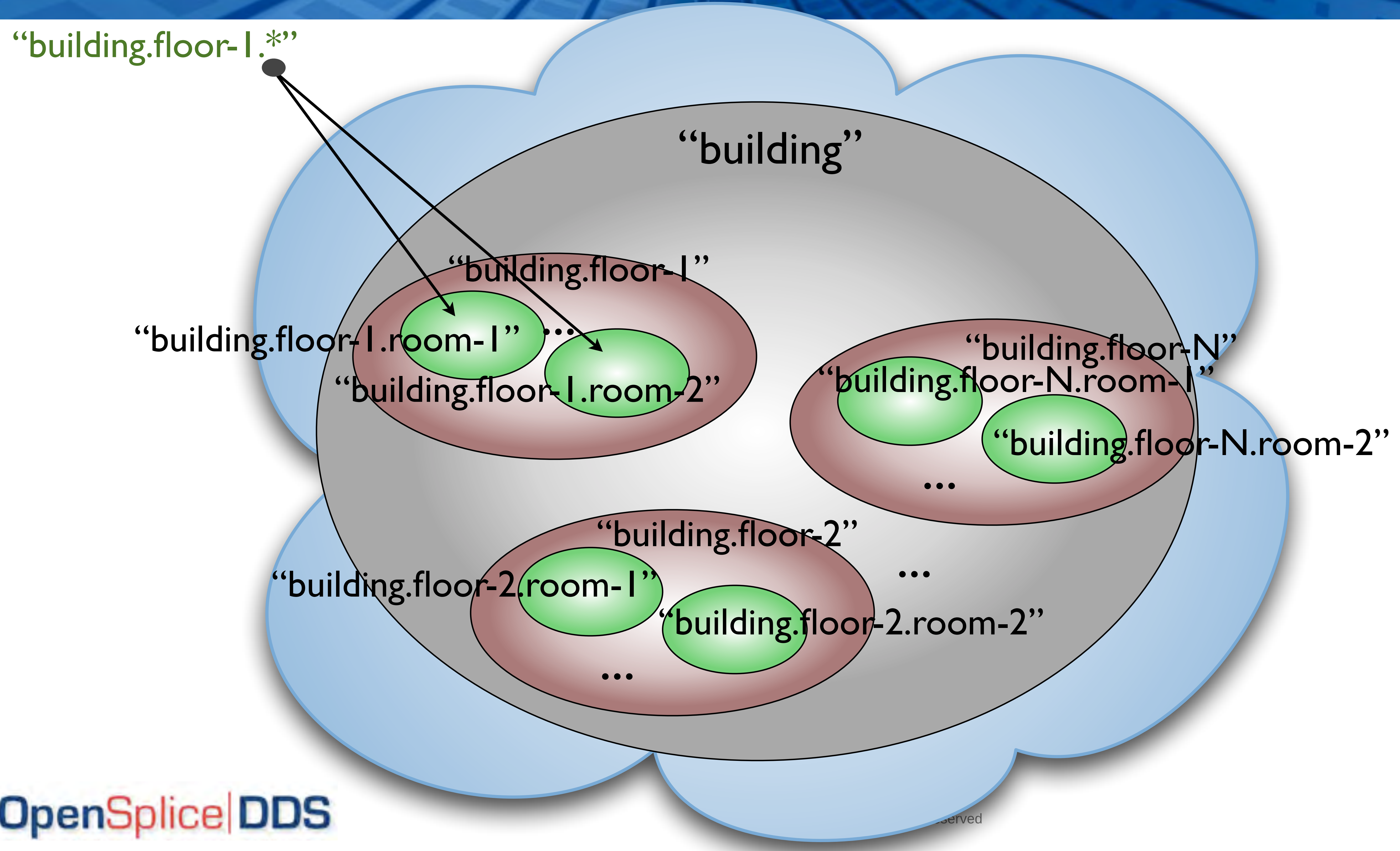
- Each partition is identified by a string, such as “sensor-data”, “log-data” etc.
- Read/Write access to a partition is gained by means of DDS Publisher/Subscribers
- Each Publisher/Subscriber can be provided with a list of Partitions name, which might as well include wildcards ,or generic regular expression,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

```
dds::Topic<ShapeType> shape(opt.topic);
dds::DataWriter<ShapeType> dw(shape);

float delta = 0.1F;
int x;
int y;
ShapeType st;
st.color = DDS::string_dup(opt.color.c_str());
st.shapesize = opt.size;


long long sec = 0;
long long nsec = opt.period;

    // nsec has to be <= 999999999
if (nsec >= ONE_SECOND) {
    sec = nsec / ONE_SECOND;
    nsec = opt.period - (sec * ONE_SECOND);
}

timespec period = {
    sec,
    nsec
};
```

```
float theta = 0;
timespec leftover;
for (int i = 0; i < opt.samples; ++i) {
    x = opt.X0 + static_cast<int>(opt.radius * cos(theta));
    y = opt.Y0 + static_cast<int>(opt.radius * sin(theta));
    theta += delta;
    st.x = x;
    st.y = y;
    dw.write(st);
    nanosleep(&period, &leftover);
    std::cout << "DW << " << st << std::endl;
}
```

OpenSplice|DDS

Delivering Performance, Openness, and Freedom

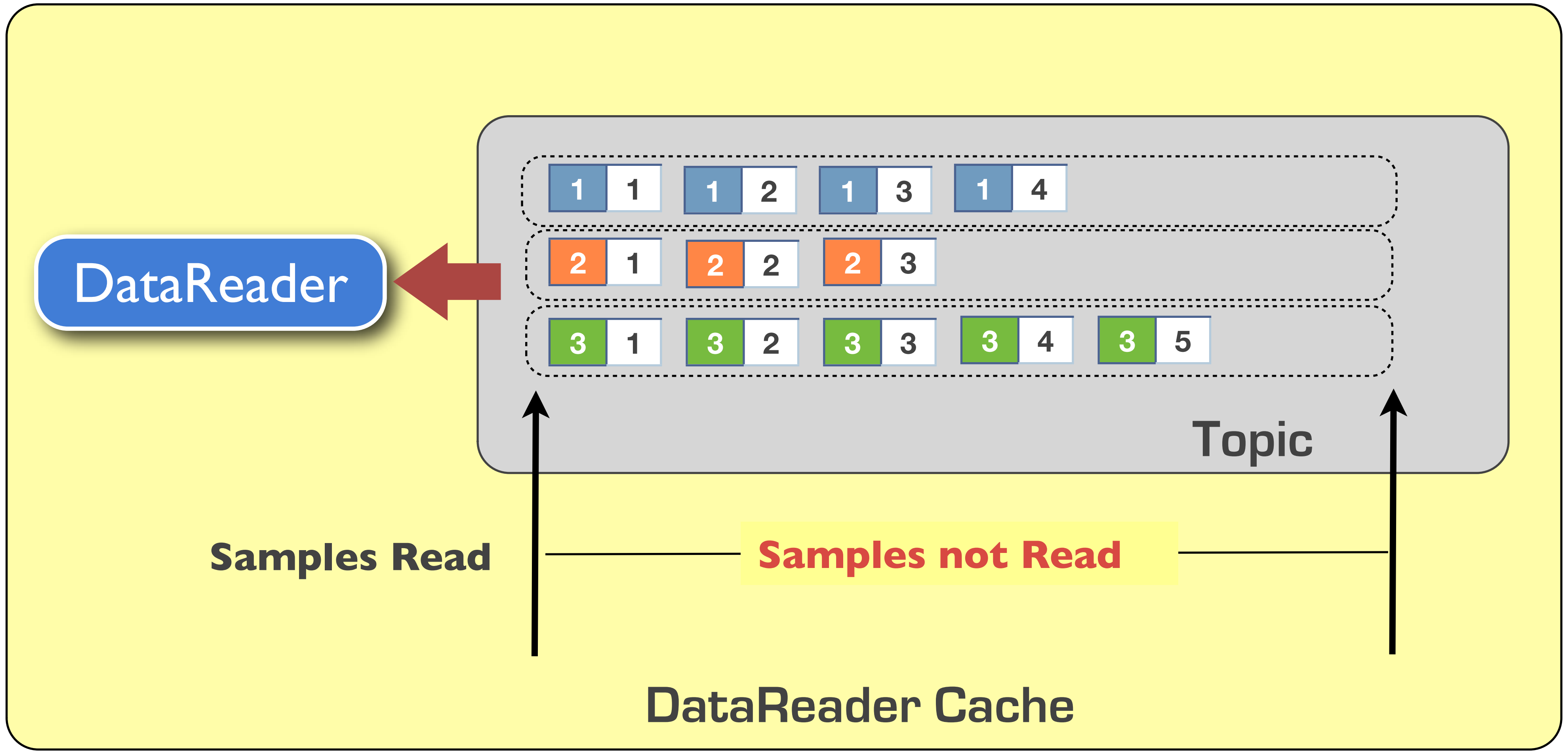
Your will learn:

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

Step IV

Consuming Data

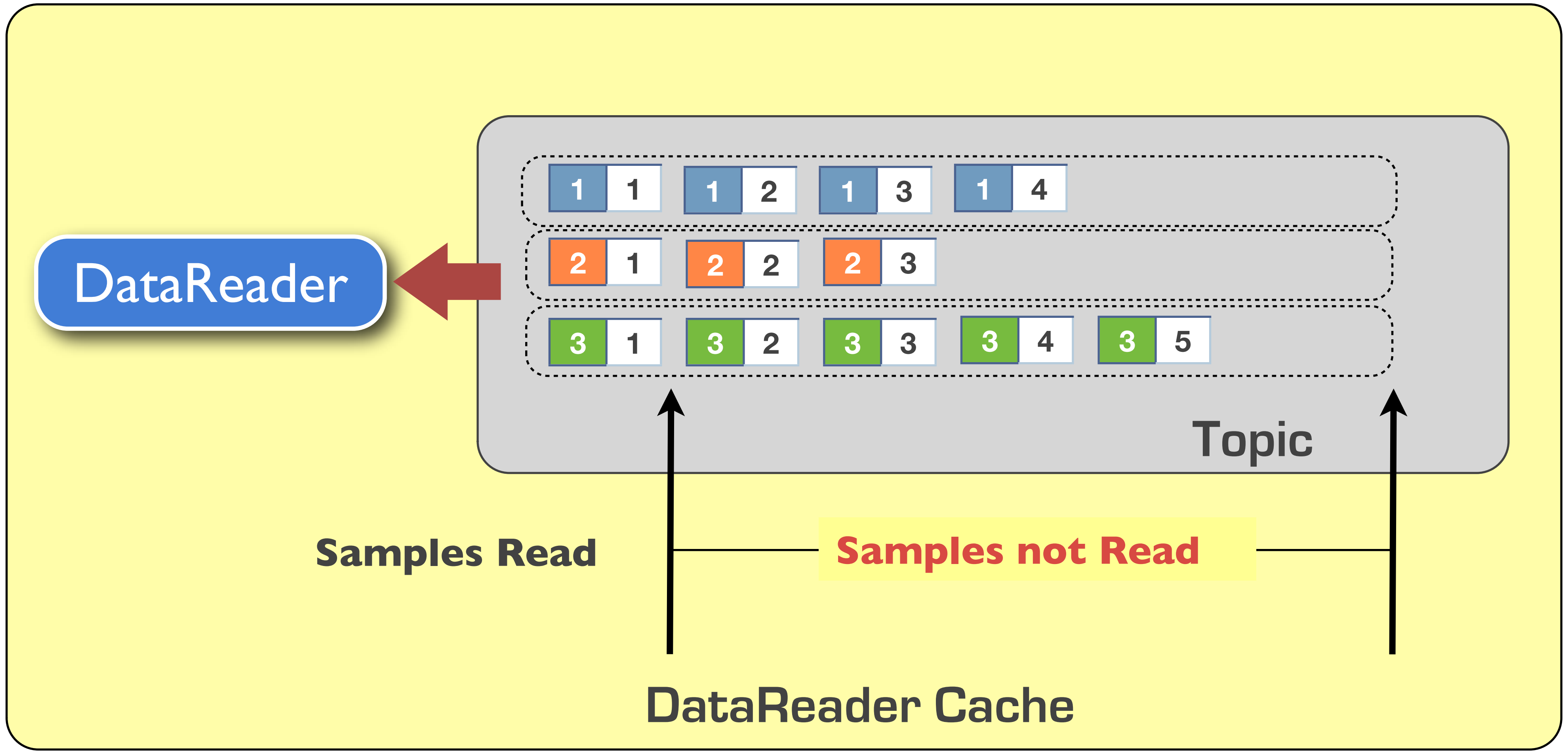
Reading Samples



- 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
```

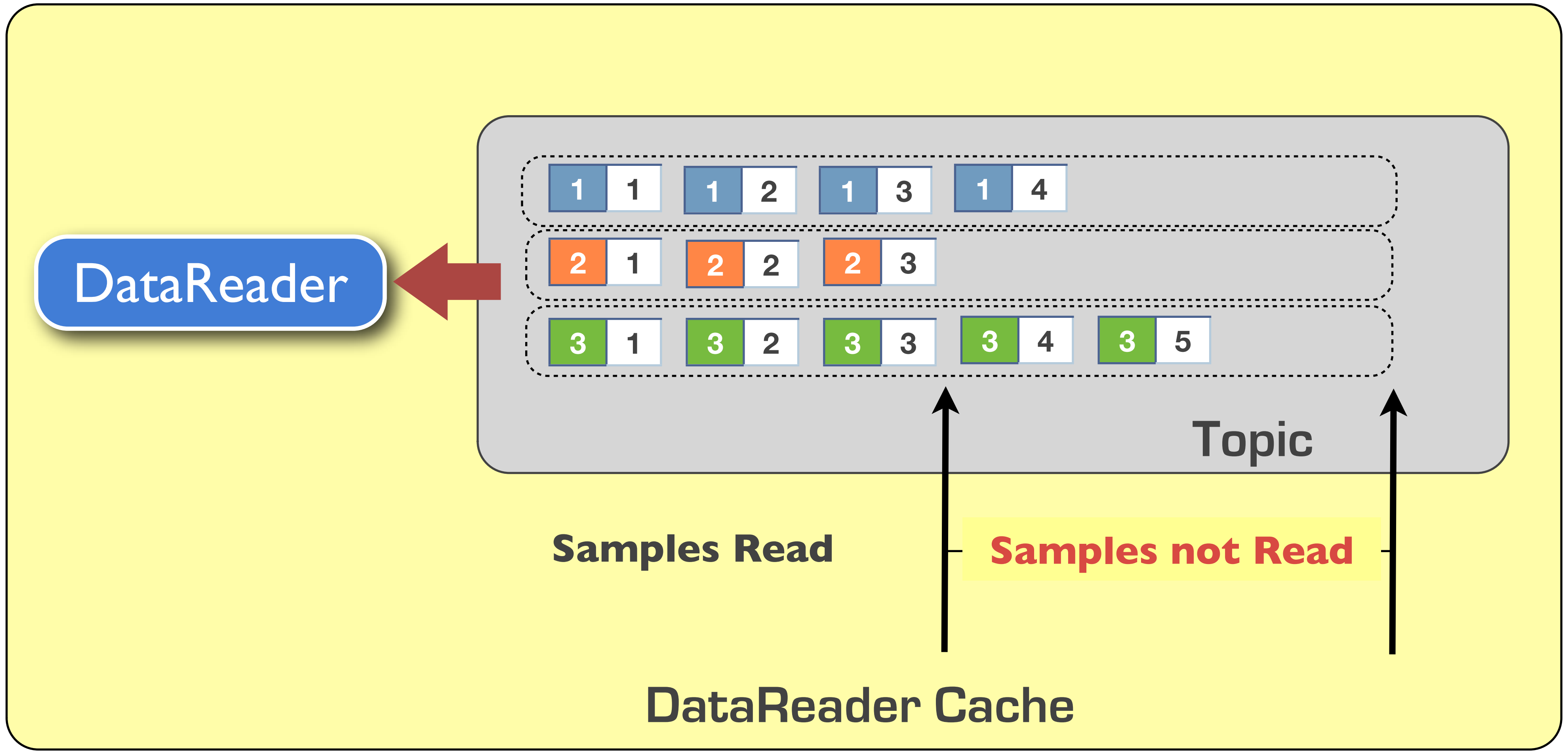
Reading Samples



- 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
```

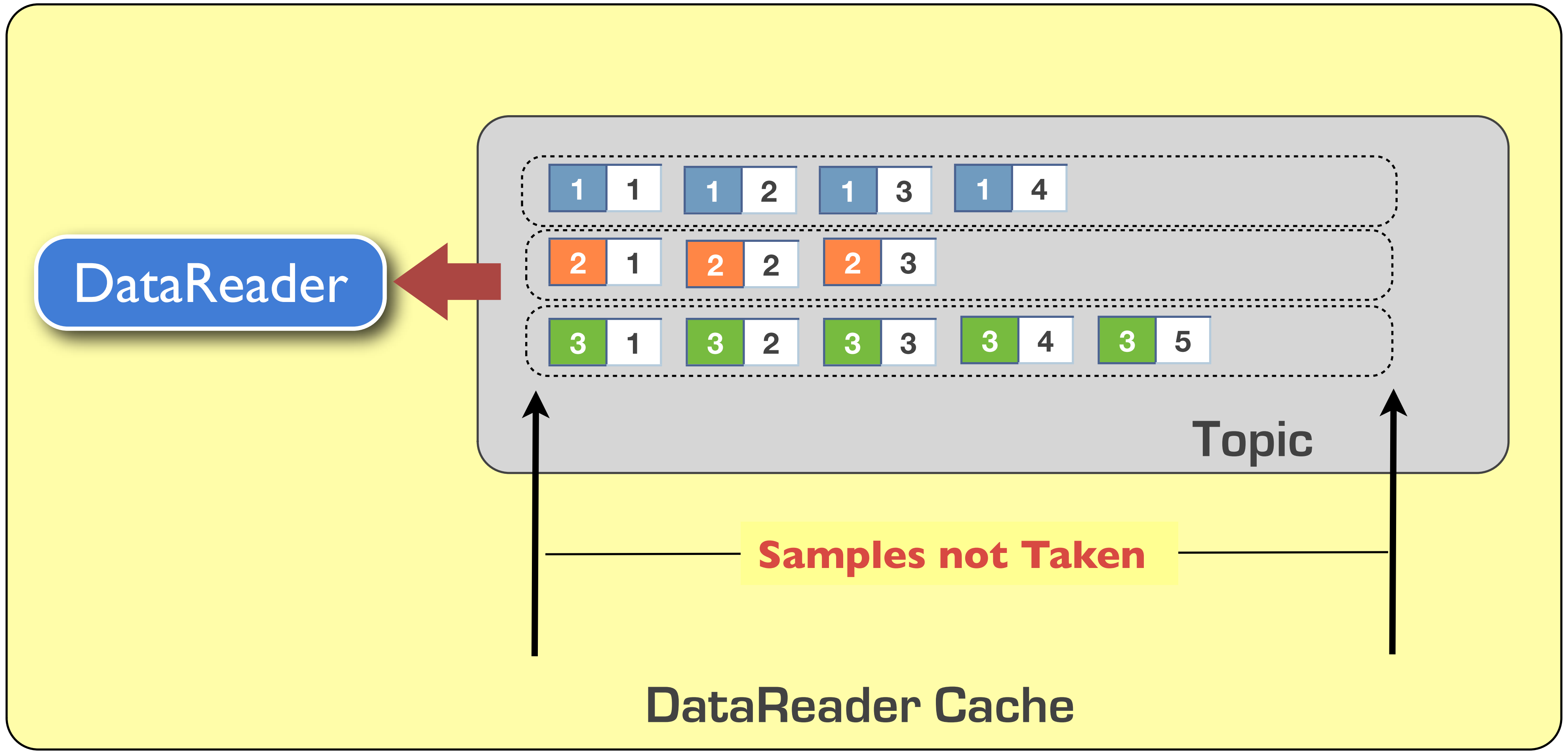
Reading Samples



- 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
```

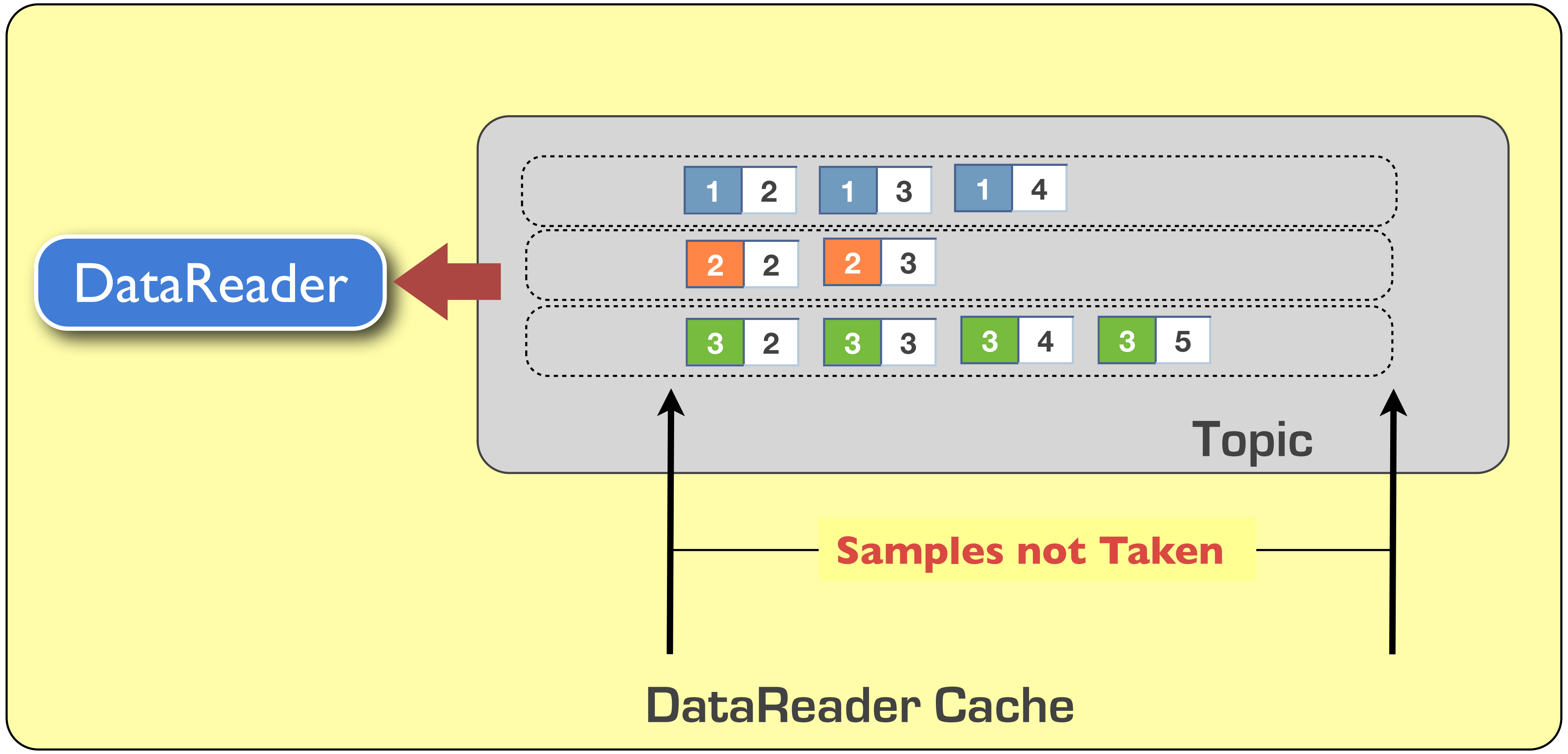
Taking Samples



- 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
```

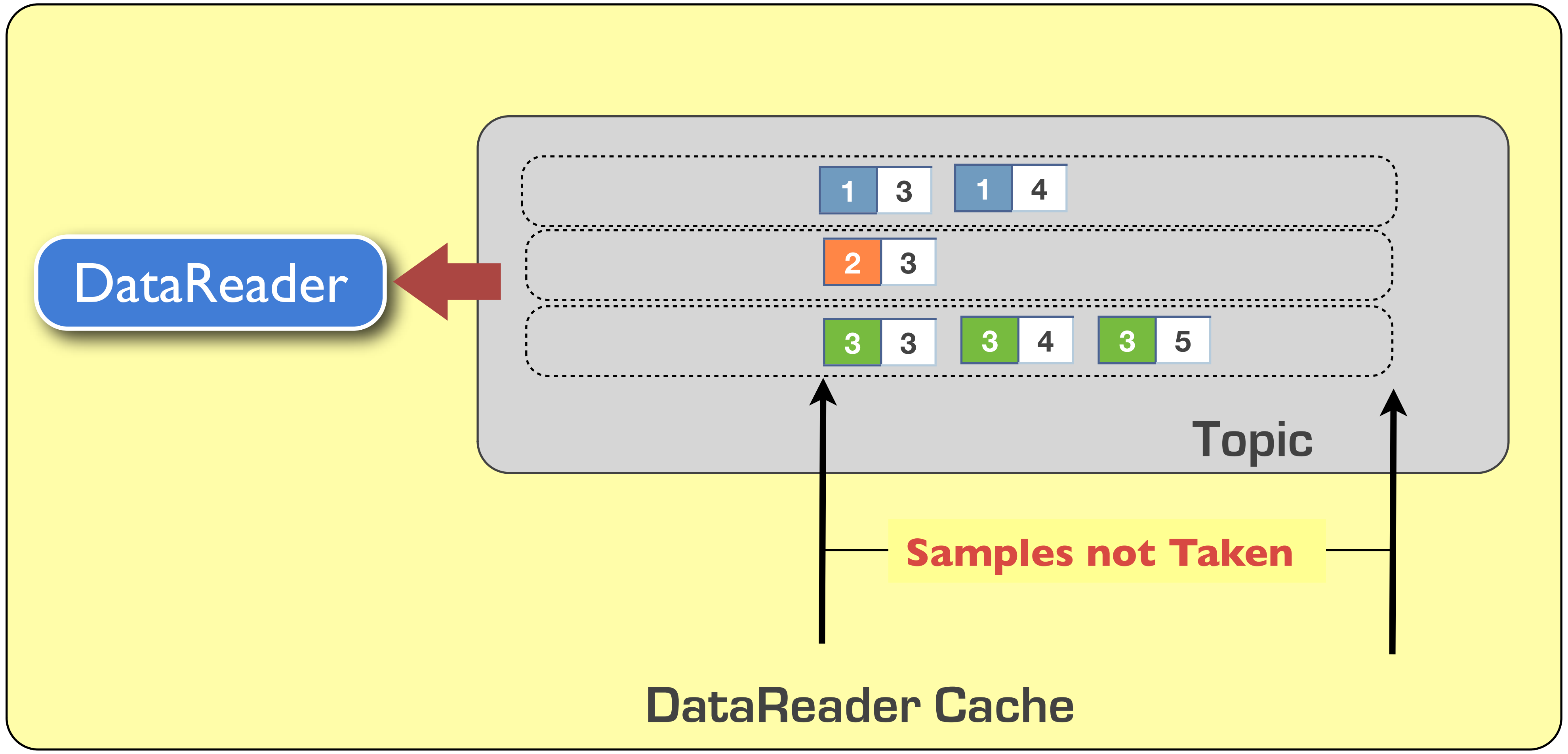
Taking Samples



- 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
```

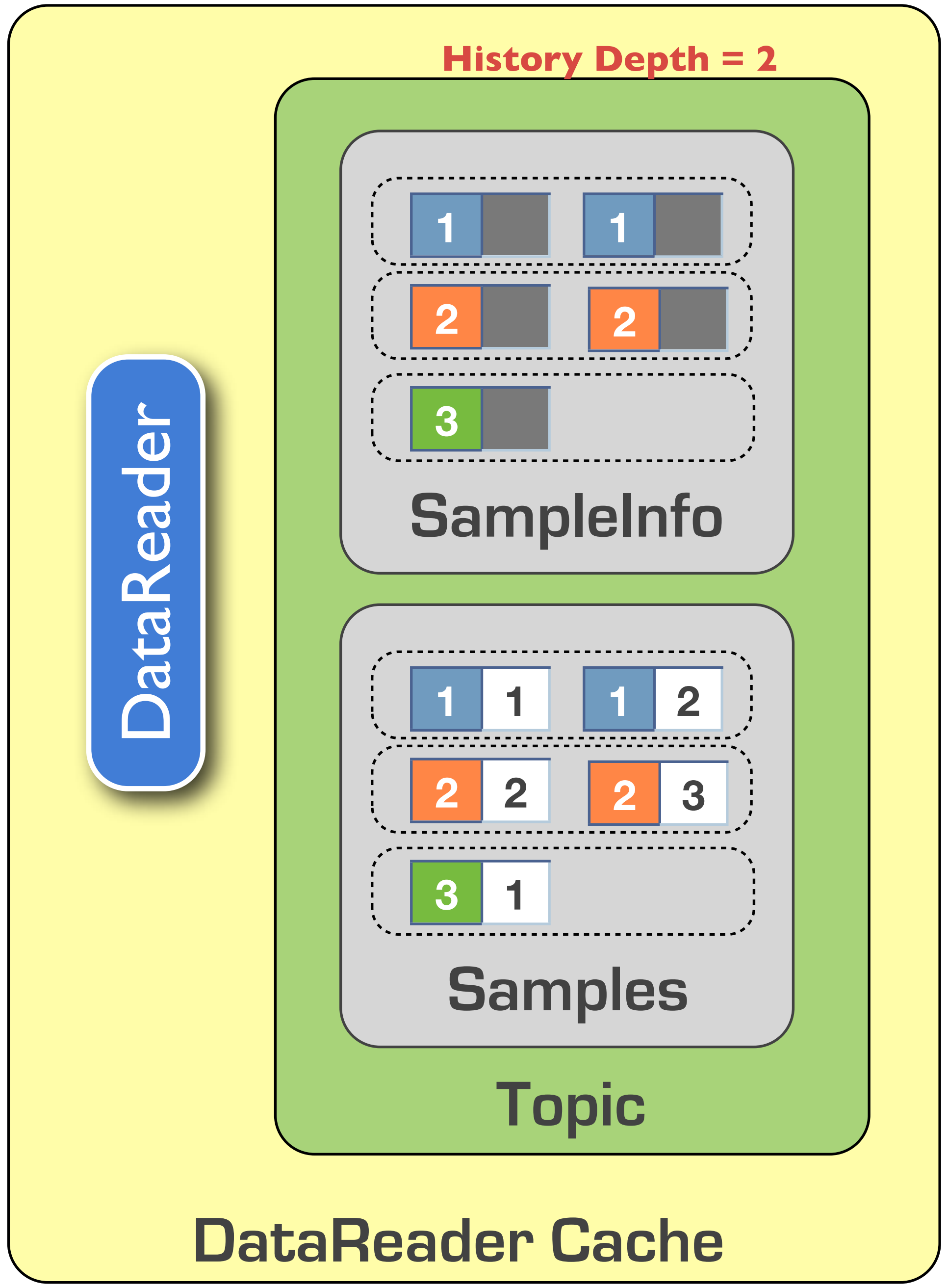
Taking Samples



- 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 wether a sample has already been read by this DataReader or not.
- **Instance State (ALIVE, NOT_ALIVE, DISPOSED).** Determines wether (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 wether 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 <conde>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 <conde>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:

```
class ShapeUpdateHandler {
public:
   ShapeUpdateHandler()  {  }
   ~ShapeUpdateHandler() { }
public:
   void operator()(dds::DataReader<ShapeType>& reader) {
      ShapeTypeSeq data;
      DDS::SampleInfoSeq status;
      reader.read(data, status);
      for (int i = 0; i < data.length(); ++i)
         std::cout << std::dec << ">>[DR]: " << data[i] << std::endl;

      reader.return_loan(data, status);
   }
};
```

```
ShapeUpdateHandler handler;
dds::ActiveReadCondition arc =
dr.create_readcondition(handler);
::dds::ActiveWaitSet ws;
DDS::ReturnCode_t retc = ws.attach(arc);

while (read_samples_ < opt.samples) {
   ws.dispatch();
}
```

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...

```
class ShapeUpdateHandler {
public:
    ShapeUpdateHandler(int samples, boost::barrier& barrier)
    : samples_(samples),
    barrier_(barrier)
    {  }

    ~ShapeUpdateHandler() { }

public:
    void handle_data(dds::DataReader<ShapeType>& reader)
    {
        ShapeTypeSeq data;
        DDS::SampleInfoSeq status;
        reader.read(data, status);
        samples_ -= data.length();
        for (int i = 0; i < data.length(); ++i)
            std::cout << std::dec << "DR >> " << data[i] << std::endl;

            // Notice it is OK to call the barrier_.wait() here since it is
            // not really going to wait but simply reach the barrier count and
            // exit the program.
        if (samples_ <= 0)
            barrier_.wait();
    }

    void handle_liveliness_change(dds::DataReader<ShapeType>& reader,
                                  const DDS::LivelinessChangedStatus& status)
    {
        std::cout << status << std::endl;
    }

private:
    int samples_;
    boost::barrier& barrier_;
};
```

```
boost::barrier completion_barrier(2);
ShapeUpdateHandler handler(opt.samples, completion_barrier);

dds::sigcon_t con_data =
dr.on_data_available_signal_connect(boost::bind(&ShapeUpdateHandler::handle_data,
                                                &handler,
                                                _1));


dds::sigcon_t con_liv =
dr.on_liveliness_changed_signal_connect(boost::bind(&ShapeUpdateHandler::handle_liveliness_change,
                                                    &handler,
                                                    _1,
                                                    _2));


completion_barrier.wait();
con_data.disconnect();
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

OpenSplice DDS
Delivering Performance, Openness, and Freedom

- http://www.opensplice.com/
- emailto:opensplicedds@prismtech.com

slideshare
Present Yourself

- http://www.slideshare.net/angelo.corsaro

webex

- http://bit.ly/1Sreg

twitter

- http://twitter.com/acorsaro/

YouTube

- http://www.youtube.com/OpenSpliceTube

Blogger

- http://opensplice.blogspot.com