



VANDERBILT  
UNIVERSITY



Your systems.  
Working as one.

# Reactive Stream Processing for Data-Centric Publish-Subscribe

## **Vanderbilt University**


Shweta Khare, Kyoungho An, Aniruddha Gokhale  
{shweta.p.khare, kyoungho.an, a.gokhale}@vanderbilt.edu  
Nashville, Tennessee, USA

## **Real Time Innovations Inc.**


Sumant Tambe, Ashish Meena  
{sumant,ashish}@rti.com  
Sunnyvale, California, USA




# Industrial Internet of Things (IIoT)




- **IoT:** Expansion of internet to include physical objects.
- **IIoT:** Industry oriented and mission critical systems.




Asset Tracking




Healthcare




Industrial Automation



Distributed Power Generation



Intelligent Transportation



2

The Internet of Things (IoT) - expansion of the Internet to include physical devices; thereby bridging the divide between the physical world and cyberspace. These devices or "things" are uniquely identifiable, fitted with sensors and actuators, which enable them to gather information about their environment and respond intelligently.

The Industrial IoT (IIoT) (distinct from consumer IoT) will help realize critical infrastructures, such as smart-grids, intelligent transportation systems, advanced manufacturing, health-care tele-monitoring, etc.

**Healthcare systems:** Hospital errors is the 6<sup>th</sup> Leading cause of preventable deaths. These errors are caused by false alarms, slow responses, and inaccurate treatment delivery. By integrating multiple monitoring devices, alarms can be triggered only when multiple devices indicate critical physiological parameters, thereby reducing the risk of false alarms. Smart drug delivery systems can react to patient conditions much faster and more reliably.

**Distributed Power Generation:** A wind farm comprises of around 100 turbines and each turbine has upto a 1000 sensors. Integrating these turbines together and building a distributed control system, is very challenging. Each turbine costs more than a million dollars. The distributed control system should be able to manage load and prevent damage to these expensive assets.

**Asset tracking:** Monitoring high-value mobile assets like locomotives, marine vessels and industrial equipment. Condition based Maintenance.

**Industrial Automation:** Integration of various PLC( Programmable logic controllers) on factory assembly lines.

**IIoT systems are characterized by:**

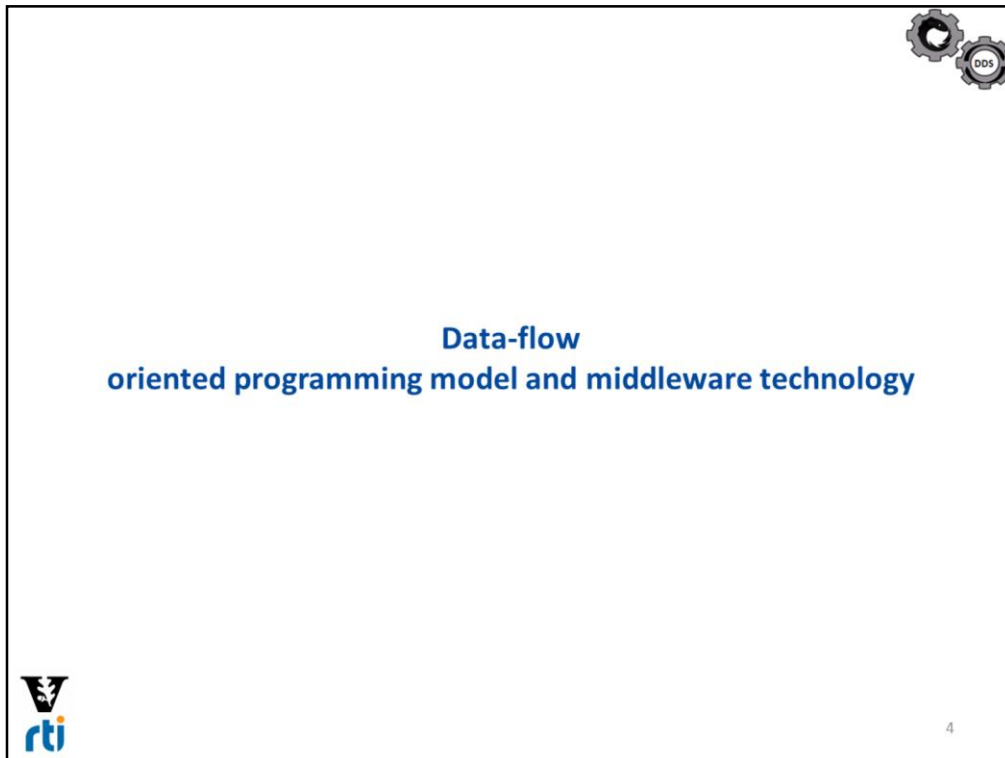
- 1) large-scale, distributed systems with many data publishers, publishing data at high volume and velocity.
- 2) Resulting unbounded asynchronous streams of data must be processed on the fly in a distributed and parallel manner to provide low-latency results.
- 3) Analyzed information must be transmitted downstream to a heterogeneous set of subscribers.

**Emerging IIoT systems can be understood as a distributed asynchronous data-flow.**



**IIoT systems can be visualized as a distributed data-flow**





#### KEY CHALLENGE:

Lies in developing a dataflow oriented programming model and middleware technology that can address both data distribution and processing requirements adequately.

Distribution aspects of the data-flow are sufficiently handled by pub-sub technologies like OMG DDS. However, the data-processing aspects – local to the individual stages of a distributed data-flow- are often not implemented as a dataflow due to the lack of sufficient composability and generality in API of pub-sub middleware.

For Example:

DDS offers various ways to receive data such as:

**listener callbacks** for push-based notification,

**read/take functions for polling,**

**waitset and read-condition to receive data from several entities at a time,**

**and query-conditions to enable application specific filtering and de-multiplexing.**

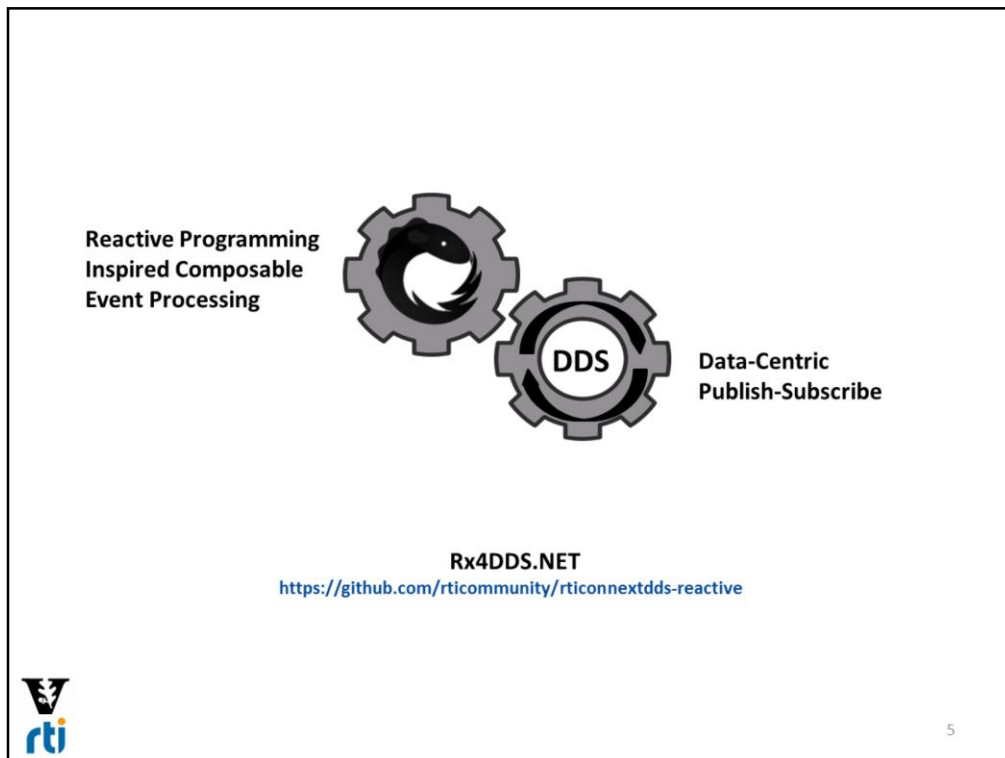
BUT:

**These primitives, are designed for data and meta-data delivery as opposed to processing.**  
Further, the lack of proper abstractions

forces programmers to develop event-driven applications using the observer pattern.

**A desirable programming model is one that provides a first-class abstraction for streams; and one that is composable and offers a reusable set of primitives for multiplexing, demultiplexing, merging, splitting and filtering over streams.**

**MOREOVER, implementing these stream processing primitives in a functional style as opposed to imperative style, yields a significant improvement in the expressiveness, composability and scalability of the programs.**



To develop an **end-to-end dataflow model** that unifies **both** local data-processing and distribution: We have investigated in the integration of composable event processing inspired by Reactive programming and have blended it with data-centric publish-subscribe.

We have combined **concrete instances of pub-sub technology and reactive programming, to evaluate and demonstrate our research ideas.**

The data-centric pub/sub instance we have used is OMG's DDS, more specifically the DDS implementation provided by Real Time Innovations

Inc;

while the reactive programming instance we have used is Microsoft's .NET Reactive Extensions (Rx.NET). We have made our solution available as a reusable open-source library called Rx4DDS.NET.

**Reactive programming languages** provide a dedicated abstraction for time-changing values called signals or behaviors. The language runtime tracks changes to the values of signals/behaviors and propagates the change through the application by re-evaluating dependent variables automatically. Originally reactive programming was developed in the context of pure functional languages like Haskell and was termed as Functional Reactive Programming(FRP). Since then RP has been implemented in many other languages like scala (Scala.React), javascript (FlapJax) and java (Frappe).

**Composable Event Processing:** Is a modern variant of FRP and is an emerging new way of developing scalable reactive applications. For example it is being used at Netflix to efficiently scale and handle user requests.



# Outline



- **Overview of Technologies**
  - ❖ [OMG DDS](#)
  - ❖ [Microsoft Rx](#)
- **Rx4DDS.NET**
- **Evaluation Case-Study**
  - ❖ [DEBS 2013 Grand Challenge](#)
- **Benefits of Rx4DDS.NET**
  - ❖ [Declarative Style](#)
  - ❖ [Composability and Program structure](#)
  - ❖ [Flexible Component Boundaries](#)
  - ❖ [Declarative Concurrency Management](#)
  - ❖ [Backpressure](#)
- **Experimental Results**
- **Conclusion**



6

- 1) First we provide a brief overview of the two technologies: DDS and Rx
- 2) Then we show the strong correspondence between the distributed asynchronous data-flow model of DDS and the local asynchronous processing model of Rx. The remarkable overlap and synergy between them forms the basis of Rx4DDS.NET
- 3) To evaluate our solution we implemented DEBS 2013 grand challenge queries using both the imperative style of programming and reactive style by using Rx4DDS.NET library.
- 4) We present the benefits our reactive solution over the imperative solution to the case-study problem, i.e., DEBS 2013 grand challenge queries.
- 5) Finally we present some experimental results and conclude.

# Outline



## ▪ Overview of Technologies

- ❖ **OMG DDS**

- ❖ **Microsoft Rx**

## ▪ Rx4DDS.NET

## ▪ Evaluation Case-Study

- ❖ DEBS 2013 Grand Challenge

## ▪ Benefits of Rx4DDS.NET

- ❖ Declarative Style

- ❖ Composability and Program structure

- ❖ Flexible Component Boundaries

- ❖ Declarative Concurrency Management

- ❖ Backpressure

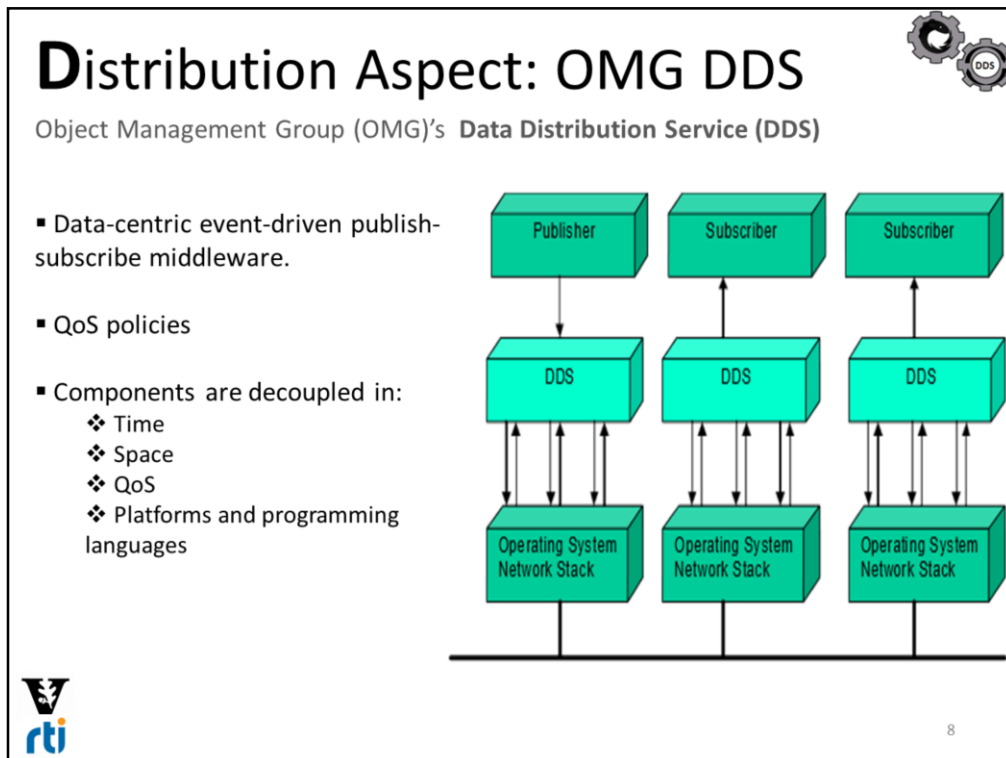
## ▪ Experimental Results

## ▪ Conclusion



7

- 1) First we provide a brief overview of the two technologies: DDS and Rx
- 2) Then we show the strong correspondence between the distributed asynchronous data-flow model of DDS and the local asynchronous processing model of Rx. The remarkable overlap and synergy between them forms the basis of Rx4DDS.NET
- 3) To evaluate our solution we implemented DEBS 2013 grand challenge queries using both the imperative style of programming and reactive style by using Rx4DDS.NET library.
- 4) We present the benefits our reactive solution over the imperative solution to the case-study problem, i.e., DEBS 2013 grand challenge queries.
- 5) Finally we present some experimental results and conclude.



OMG's Data Distribution Service or DDS – is a Data-Centric and **Event-Driven** publish-subscribe middleware.

DDS supports a variety of QoS policies and thereby supports **highly flexible and complex data flow requirements posed by IIoT systems**.

DDS is event-driven and supports loose-coupling between - the Publishers (data generators) and Subscribers (data receivers)

Which are decoupled wrt to:

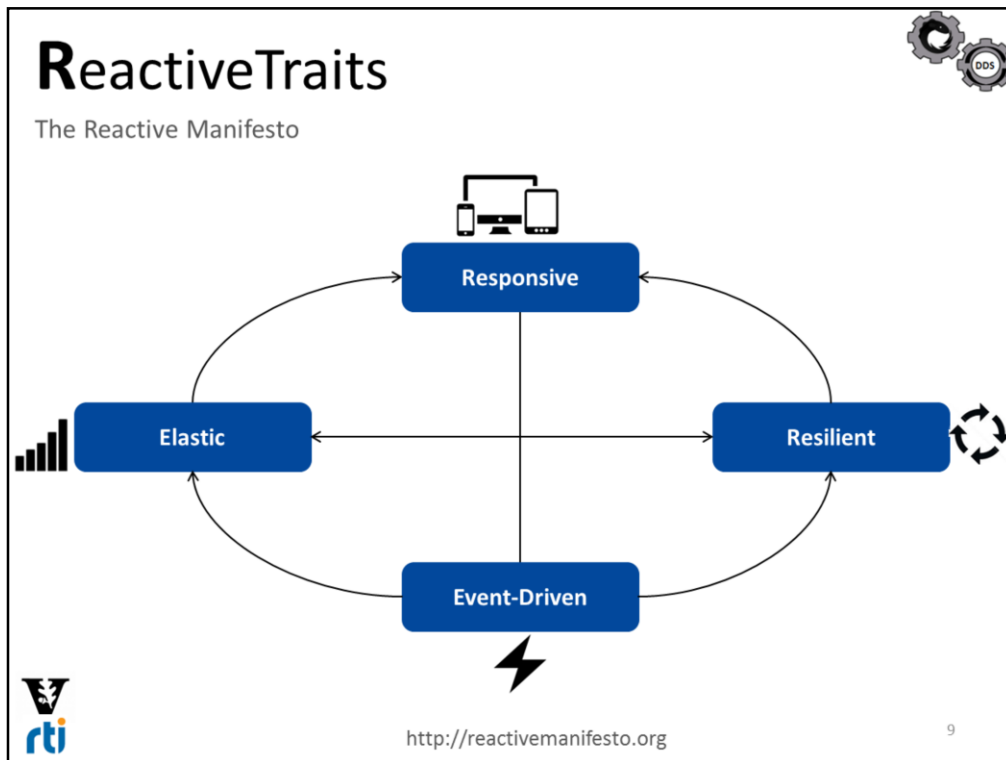
Time – They need not be present at the same time

Space – They may be located any where

QoS – Publishers must offer better or equivalent QoS than required by data subscribers.

Platforms and programming languages.

Event-Driven design is a pre-requisite for building systems that are REACTIVE.



Reactive Systems, are quite simply – Systems that react to all external Stimuli.

**It should react to →**

- **Failure, Isolate it and Recover from it.**
- **Load Variations**
- **User interaction events and incoming data.**

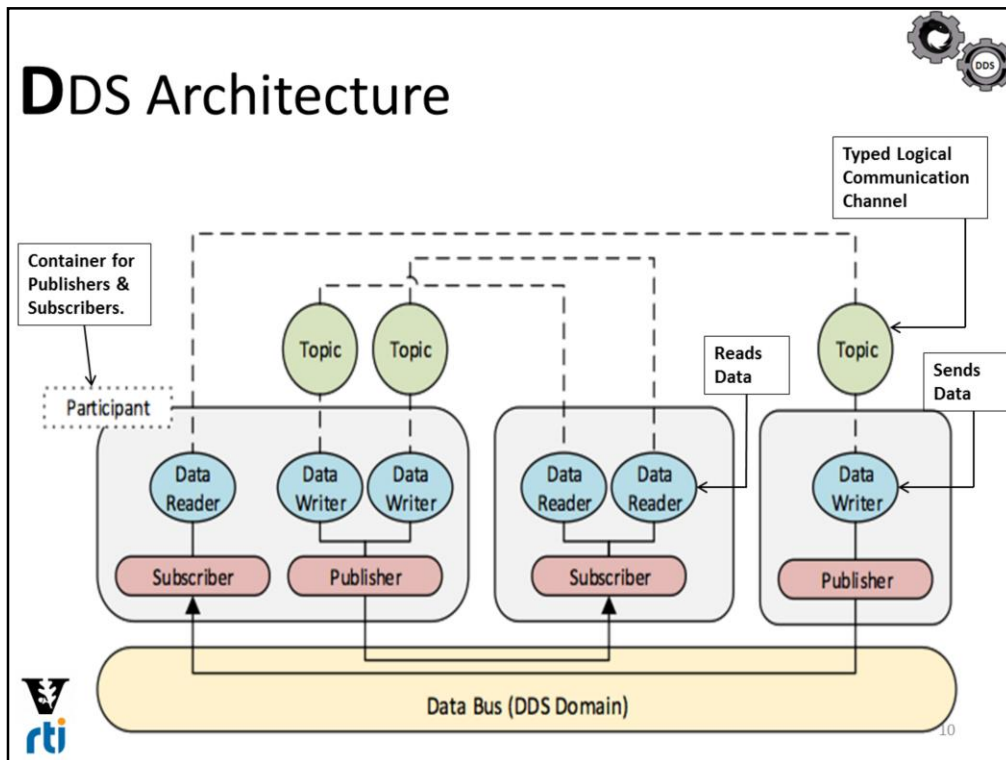
**Event driven design promotes loose-coupling and asynchrony between system components which in turn helps in achieving the remaining three traits of:**

**Resilience:** Ability to isolate and contain faults

**Elasticity:** Adapt to changing workload characteristics while maximizing resource usage.

**Responsiveness:** Rapid and CONSISTENT/ PREDICTABLE response times

Moreover, asynchronous event-based architectures **unify scaling up and scaling out** while deferring the choice of the scalability mechanism at deployment-time without hiding the network from the programming model.



DDS provides a global data-space that is accessible to all interested communicating entities. Data objects are identified by their topic name, Data-type and Key. A Topic is a logical data-stream in DDS.

**DDS DataWriters** (belonging to the publisher) and **DataReaders** (belonging to the subscriber) are endpoints used in DDS applications to write and read typed data messages (DDS samples) from the global data space. DDS ensures that the endpoints are compatible with respect to the topic name, data type, and the QoS policies.

DDS supports keyed data types much like a primary key in a database. Keyed data types partition the global data-space into logical streams called **instances** which have an observable lifecycle.

# Microsoft Reactive Extensions (Rx)



- Rx is a library for composing Asynchronous and Event-Based programs using Observable Sequences And Linq-Style query operators.

**Rx = Observables**

+

**LINQ**

+

**Schedulers**

- RxJava, RxJs, RxPython, RxCpp, RxScala etc.



11

- Rx is:
  - ❖ set of types representing asynchronous data streams
  - ❖ set of operators for querying asynchronous data streams.
  - ❖ set of types to parameterize concurrency.

Streams are first-class.

Built-in operators for multiplexing, de-multiplexing, filtering, merging and performing time-based operations on streams.

Rx has been classified as a “**cousin of reactive programming**” since Rx does not provide a dedicated abstraction for time-changing values which can be used in ordinary language expressions (i.e. automatic lifting of operators to work on behaviors/signals); rather it provides a container (observable) and the programmer needs to manually extract the values from this container and encode dependencies between container values explicitly (i.e. manual lifting of operators).

An object is first-class when it:

- can be **stored** in variables and data structures
- can be **passed** as a parameter to a subroutine
- can be **returned** as the result of a subroutine
- can be **constructed** at runtime
- has **intrinsic identity**

# Interfaces in Rx



## **IObservable<T> & IObserved<T>**

Duals of IEnumerable<T> and IEnumerator<T>

```
class IObservable<in T>
{
    IDisposable Subscribe(IObserved<T> observer);
}
```

```
class IObserved<in T>
{
    void OnNext(T value);
    void OnError(Exception error);
    void OnCompleted();
}
```




12

An **IObservable<T>** produces values of type T/represents a stream of type T. This interface has a single function: Subscribe using which **Observers** subscribe to data streams much like the Subject-Observer pattern.



IObservable<T> **supports chaining of functional operators** to create pipelines of data processing stages.

An **IObserved<T>** has three methods: onNext, onError and onCompleted. An observer's onNext method is called back when a stream has a new data element; If the stream completes or has an error, the OnCompleted, and OnError operations are called, respectively.



**IObservable<IObservable<T>>**

**IObservable<IGroupedObservable<Tkey,T>>**



13

Rx supports streams of streams where every object produced by an Observable is another Observable (e.g., `IObservable<IObservable<T>>`).

Some Rx operators, such as `GroupBy`, demultiplex a single stream of `T` into a stream of keyed streams producing `IObservable<IGroupedObservable<Key,T>>`



# Interfaces in Rx



**ObserveOn (IScheduler sched)**  
**SubscribeOn (IScheduler sched)**

```
interface IScheduler
{
    DateTimeOffset Now { get; }
    IDisposable Schedule(Action work);
    IDisposable Schedule(TimeSpan dueTime, Action work);
    IDisposable Schedule(DateTimeOffset dueTime, Action work);
}
```



14

No Concurrent Work Is Introduced In Rx Without Going Through **IScheduler** Interface.

# Outline



- **Overview of Technologies**

- ❖ OMG DDS
- ❖ Microsoft Rx

- **Rx4DDS.NET**

- **Evaluation Case-Study**

- ❖ DEBS 2013 Grand Challenge

- **Benefits of Rx4DDS.NET**

- ❖ Declarative Style
- ❖ Composability and Program structure
- ❖ Flexible Component Boundaries
- ❖ Declarative Concurrency Management
- ❖ Backpressure

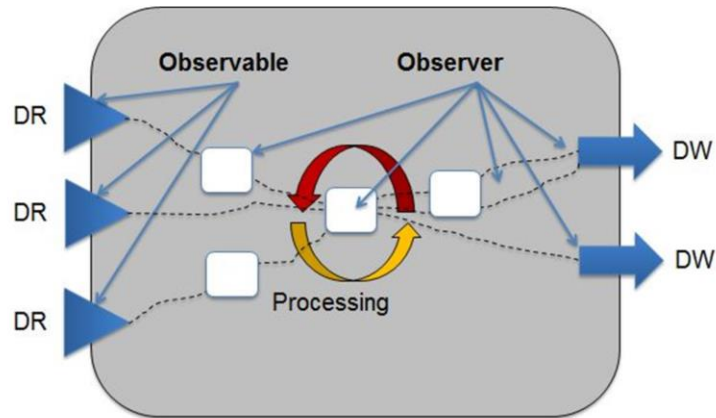
- **Experimental Results**

- **Conclusion**



# Rx4DDS.NET

DDS for distribution, Rx for processing



16

Diagram shows the conceptual mapping between DDS and Rx: the data received by a DataReader is converted into an Rx Observable which is later consumed by downstream query operators



DDS Concept	Rx Concept/Type/Operation
Topic of type T	An object that implements IObservable<T>, which internally creates a DataReader<T>
Communication status, Discovery event streams	IObservable<SampleLostStatus> IObservable<SubscriptionBuiltinTopicData>
Topic of type T with key type=Key	IObservable<IGroupedObservable<Key, T>>
Detect a new instance	Notify Observers about a new IGroupedObservable<Key, T> with key==instance. Invoke IObservable<IGroupedObservable<Key, T>>.OnNext()
Dispose an instance	Notify Observers through IObservable<IGroupedObservable<Key, T>>.OnCompleted()
Take an instance update of type T	Notify Observers about a new value of T using IObservable<T>.OnNext()



For keyed data-types, DDS distinguishes each key as a separate instance. An instance can be thought of as a continuously changing row in a database table. Since, each instance is a logical stream in within a topic, a keyed topic can be viewed as a stream of keyed streams. Or IObservable<IGroupedObservable<Key,T>>. Rx4DDS.NET library detects a new key, it reacts by producing a new IGroupedObservable<Key,T> with a new key.

DDS provides various events to keep track of communication status, such as **deadlines missed and samples lost** between DataReaders and DataWriters. For discovery of DDS entities, the DDS uses predened built-in topics: These can also be mapped to Rx observables.

The contract between any two consecutive stages composed with Rx Observables is based on only two notions:

- (1) the static type of the data owing across and
- (2) and the pair of IObservable and IObservable interfaces that represents the lifecycle of a data stream.

These notions can be mapped directly to DDS in the form of strongly typed topics and the notion of instance lifecycle.



DDS Concept	Rx Concept/Type/Operation
Read with history=N	IObservable<T>.Replay(N)
SELECT in CFT expression	IObservable<T>.Select(...)
FROM in CFT expression	DDSObservable.FromTopic("Topic1") DDSObservable.FromKeyedTopic("Topic2")
WHERE in CFT expression	IObservable<T>.Where(...)
ORDER BY in CFT expression	IObservable<T>.OrderBy(...)
MultiTopic (INNER JOIN)	IObservable<T>.SelectMany(nestedSelector), where nestedSelector maps T to IObservable<U>
Time-based filter	IObservable<T>.Sample(...)



Some DDS QoS policies like history QoS and Time-based filters can be mapped to Rx Replay and Sample operators.

# Outline



- **Overview of Technologies**

- ❖ OMG DDS
- ❖ Microsoft Rx

- **Rx4DDS.NET**

- **Evaluation Case-Study**

- ❖ **DEBS 2013 Grand Challenge**

- **Benefits of Rx4DDS.NET**

- ❖ Declarative Style
- ❖ Composability and Program structure
- ❖ Flexible Component Boundaries
- ❖ Declarative Concurrency Management
- ❖ Backpressure

- **Experimental Results**

- **Conclusion**



# DEBS Grand Challenge 2013

## Real-time complex analytics over high-velocity sensor data

- Ball sensor emit samples @2000 Hz. Player sensors @200 Hz.  
Approx. 15,000 data points/sec.

**sid, ts, x, y, z, |v|, |a|, vx, vy, vz, ax, ay, az**

- Query 1: Current Running Analysis

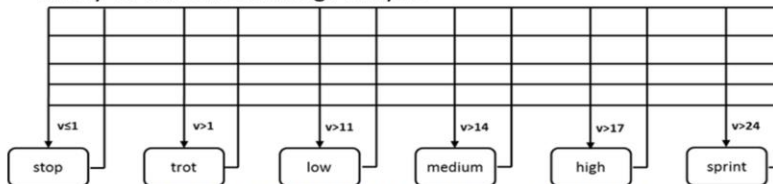


Figure 2 - Transitions between running intensities for Query 1

- Query 3: Player Position Heatmap



20

2013 Grand Challenge problem comprises real-life data from a soccer game and challenge problem consists of four distinct queries that must be executed on the incoming streams of data on the fly.

The sensors are located near each player's cleats, in the ball, and attached to each goal keeper's hands. The sensors attached to the players generate data at 200Hz while the ball sensor outputs data at 2,000Hz. Each data sample contains the sensor ID, a timestamp in picoseconds, and three-dimensional coordinates of location, velocity, and acceleration.

For brevity we only describe queries 1 and 3 and present results for the same:

### Query-1:

Two sets of results current running statistics and aggregate running statistics must be returned.

**Current running statistics** should return the distance, speed and running intensity of a player, where running intensity is classified into six states (stop, trot, low, medium, high and sprint) based on the current speed.

**Aggregate running statistics** for each player are calculated from the current running statistics and must be reported for four

different time windows: 1 minute, 5 minutes, 20 minutes and entire game duration.

### Query-3:

Heat map statistics capturing how long each player stays in various pre-defined regions of the field.

The soccer field is divided into four grids with x rows and y columns (8x13, 16x25, 32x50, 64x100) and results should be generated for each grid type. Moreover, calculations must also be performed for 4 different time windows. As a result, query 3 must output 16 result streams.

We implemented the DEBS 2013 grand challenge both in an imperative style in C# and in a reactive style by making use of the Rx4DDS.NET library.

# Outline



- **Overview of Technologies**
  - ❖ OMG DDS
  - ❖ Microsoft Rx
- **Rx4DDS.NET**
- **Evaluation Case-Study**
  - ❖ DEBS 2013 Grand Challenge
- **Benefits of Rx4DDS.NET**
  - ❖ Declarative Style
  - ❖ Composability and Program structure
  - ❖ Flexible Component Boundaries
  - ❖ Declarative Concurrency Management
  - ❖ Backpressure
- **Experimental Results**
- **Conclusion**



21



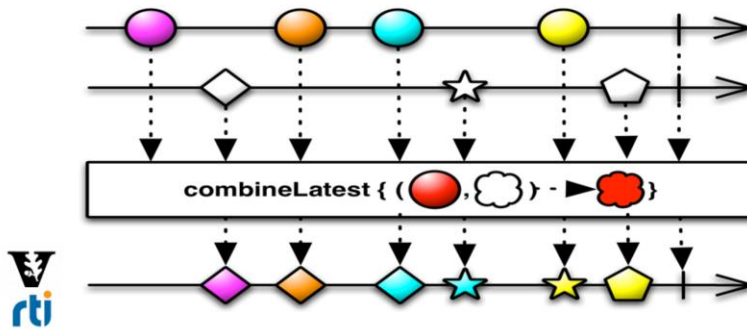
# Benefits of Rx4DDS.NET



over imperative style

- ❖ **Declarative Style**
- ❖ **Dedicated Stream abstraction and built-in Stream processing constructs.**

```
List<IObservable<SensorData>> sensorStreamList =
    new List<IObservable<SensorData>>();
Observable
    .CombineLatest(sensorStreamList)
    .Select(lst => returnPlayerData(lst));
```



Select  
SelectMany  
Where  
Join  
Buffer  
First  
Last  
Do  
Scan  
Merge  
.....

22

**Imperative Approach:** manually code the logic and maintain relevant state information for merging, joining, multiplexing, de-multiplexing and capturing data dependencies between multiple streams of data.

Example:

To calculate average sensor data for a player from the sensor readings, we had to cache the sensor data for each sensor id as it arrives in a map of sensor id to sensor data. If the current data is for sensor id 13, then the corresponding player name is extracted and a list of other sensors also attached to this player is retrieved. Subsequently using the retrieved sensor ids as keys, the sensor data is retrieved from the map and used to compute the average player data.

In the reactive style:

We can obtain the latest sample for each sensor attached to the player with the CombineLatest function and then calculate the average sensor values.

**CombineLatest stream operator can be used to synchronize multiple streams into one by combining a new value observed on a stream with the latest values on other streams.**

sensorStreamList is a list that contains references to each sensor stream associated with sensors attached to a player. For example, for player Nick Gertje with attached sensor ids (13, 14, 97, and 98), sensorStreamList for Nick Gertje holds references to sensor streams for sensors (13, 14, 97 and 98). Doing a CombineLatest on sensorStreamList returns a list (lst in Listing 1) of latest sensor

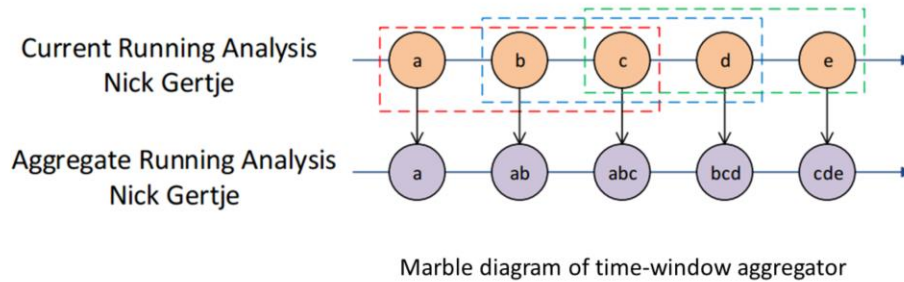
data for each sensor attached to this player. returnPlayerData function is then used to obtain the average sensor values.

# Benefits of Rx4DDS.NET



over imperative style

❖ Time Window Abstraction



23

One of the recurrent patterns in stream processing is to calculate statistics over a moving time window. All four queries in the case study require this support for publishing aggregate statistics collected over different time windows.

Rx provides the “window abstraction” which is most commonly needed by stream processing applications, and it supports both discrete (i.e., based on number of samples) and time-based windows.

# Benefits of Rx4DDS.NET



over imperative style

## ❖ Declarative Management of Concurrency

```
player_streams
    .selectMany( player_stream =>
    {
        return player_stream
            .ObserveOn(Scheduler.Default)
            .CurrentRunningAnalysis();
    }).Subscribe();
```



24

Rx provides abstractions that make concurrency management declarative. Thereby removing the need to make explicit calls to create threads or thread pools.

Rx has a **free threading model** such that developers can choose to subscribe to a stream, receive notifications and process data on

different threads of control with a simple call to **subscribeOn** or **ObserveOn**, respectively.

In Query 1, the current running statistics and aggregate running statistics get computed for each player independently of the other players. Thus, we can use a pool of threads to perform the necessary computation on a per-player stream basis.

Player\_streams represents a stream of all player streams i.e., an `IObservable<IGroupedObservable<String,PlayerData>>`.

Each player stream, which is an `IGroupedObservable<String,PlayerData>` keyed on player's name, is then processed further on a separate thread by using `ObserveOn`.

# Benefits of Rx4DDS.NET



over imperative style

## ❖ Composability and program structure.

```
player_streams.Subscribe (player_stream =>
{
    //a player's current running data stream
    var curr_running=player_stream
        .ObserveOn(ThreadPoolScheduler.Instance)
        .CurrentRunningAnalysis()
        .Publish();

    // compute aggregate running data for 1 min
    curr_running.AggregateRunningTimeSpan(1);
    // compute aggregate running data for 5 mins
    curr_running.AggregateRunningTimeSpan(5);
    // compute aggregate running data for 20 min
    curr_running.AggregateRunningTimeSpan(20);
    // compute aggregate running data for full game
    curr_running.AggregateRunningFullGame();
    curr_running.Connect();
}
```



25

**The composability of operators in Rx allows us to write programs that preserve the conceptual high-level view of**

**the application logic and data-flow.**

For example:

Query 1 computes the AggregateRunningData for each player for 1 minute, 5 minutes, 20 minutes and full game duration.

Player\_streams is a stream of streams (e.g. IObservable<IGroupedObservable<String,PlayerData>>) comprises a stream for each player). Each player stream, represented by the variable player\_stream is processed on a separate pooled thread by means of a single code statement, ObserveOn(ThreadPoolScheduler.Instance).

The CurrentRunningData for each player (curr\_running) is computed by the function CurrentRunningAnalysis() and is subsequently used by AggregateRunning\*() to compute the AggregateRunningData for each player for 1 minute, 5 minutes, 20 minutes and full game durations, respectively.

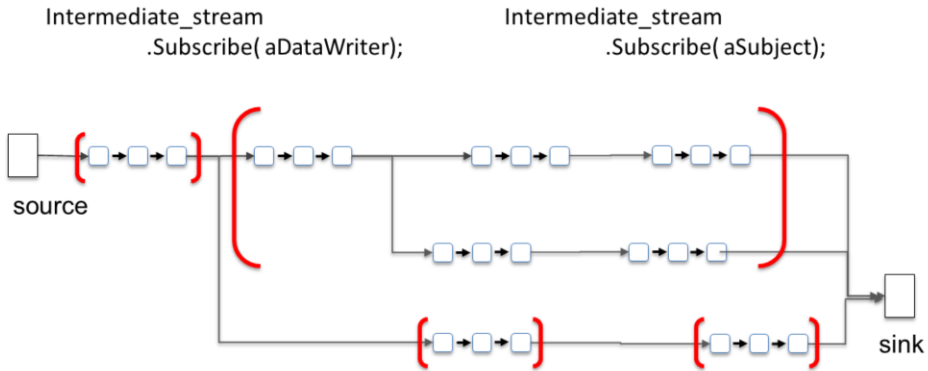
The use of Publish() and Connect() pair ensures that a single underlying subscription to curr\_running stream is shared by all subsequent AggregateRunning\*() computations otherwise the same CurrentRunningData will get re-computed for each downstream AggregateRunning\*() processing pipeline.

# Benefits of Rx4DDS.NET



over imperative style

❖ **Flexible Component Boundaries**



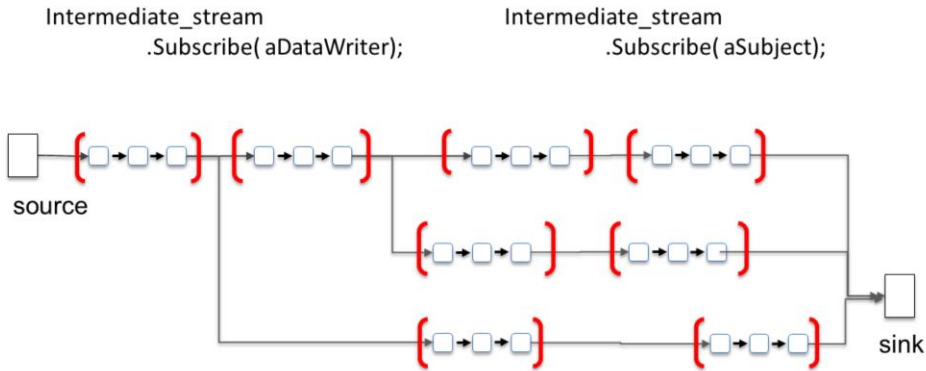
26

# Benefits of Rx4DDS.NET



over imperative style

❖ **Flexible Component Boundaries**



27

# Outline



- **Overview of Technologies**
  - ❖ OMG DDS
  - ❖ Microsoft Rx
- **Rx4DDS.NET**
- **Evaluation Case-Study**
  - ❖ DEBS 2013 Grand Challenge
- **Benefits of Rx4DDS.NET**
  - ❖ Declarative Style
  - ❖ Composability and Program structure
  - ❖ Flexible Component Boundaries
  - ❖ Declarative Concurrency Management
  - ❖ Backpressure
- **Experimental Results**
- **Conclusion**



# Quantitative Results



- Both **Imperative** and **Reactive** solutions were tested under single-threaded and multi-threaded query execution configurations

- **Test Setup:**

- ❖ Host with 2 6-core AMD Opteron 4170HE, 2.1 GHz processors, 32 GB RAM
- ❖ Sensor data was published by a separate process.
- ❖ Queries were executed in another process.
- ❖ Shared Memory Inter-process communication.
- ❖ All tests were performed 10 times; error bars denote one standard deviation.





# Quantitative Results

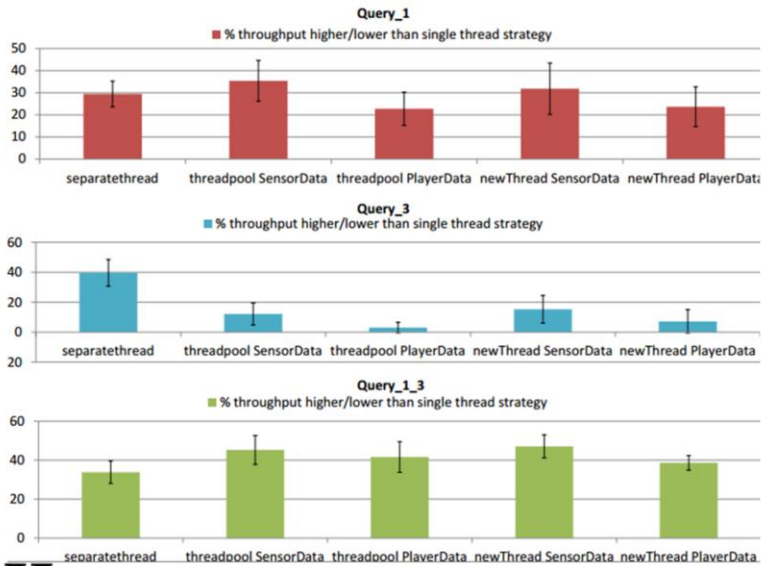


## Imperative Solution

Imperative Strategy	Description
<b>SeparateThread</b>	SensorData is received on one thread while the entire query execution is offloaded to a separate thread.
<b>ThreadPool-SensorData</b>	offloads the received SensorData on a threadpool. Each player's PlayerData calculation and subsequent player-specific processing happens on the threadpool.
<b>ThreadPool-PlayerData</b>	PlayerData is calculated from SensorData on the thread that receives SensorData from DDS; thereafter the calculated PlayerData is offloaded to a threadpool for player-specific processing.
<b>NewThread-SensorData</b>	Each player has its own designated processing thread. SensorData is dispatched to this player-specific thread, which computes that player's PlayerData and processes it further.
<b>NewThread-PlayerData</b>	Each player has its own designated processing thread. PlayerData is calculated from SensorData on the thread that receives data from DDS. PlayerData is then dispatched to player-specific thread for further processing.



# Imperative Strategies:



## Query-1

ThreadPool-SensorData shows **35%** higher throughput than single-threaded

## Query-3:

SeparateThread shows **40%** higher throughput than single-threaded

## Query-1\_3:

NewThread-SensorData shows **47%** higher throughput than single-threaded



# Quantitative Results



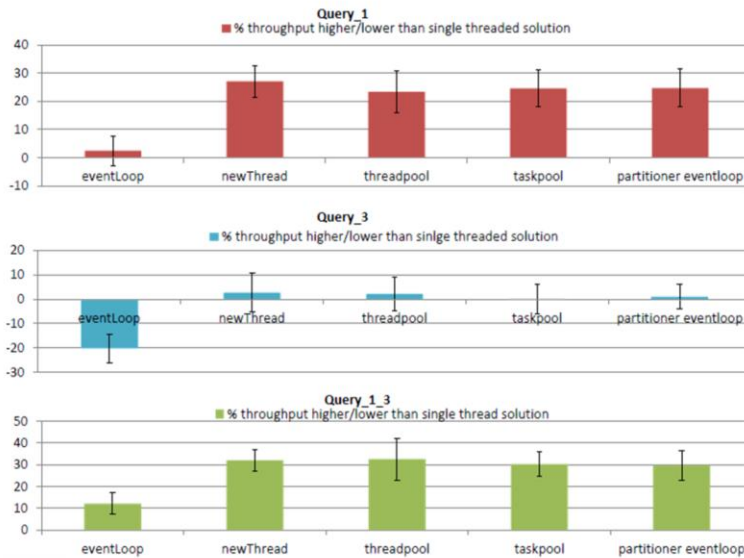
## Reactive Strategies

- ObserveOn is used with different in-built Rx schedulers to offload player-specific downstream processing on the specified scheduler.

Reactive Strategy	Description
<b>EventLoop Scheduler</b>	EventLoopScheduler provides a dedicated thread which processes scheduled tasks in FIFO order.
<b>NewThread Scheduler</b>	NewThreadScheduler processes each scheduled task on a new thread
<b>ThreadPool Scheduler</b>	ThreadPoolScheduler processes the scheduled tasks on the default threadpool
<b>TaskPool Scheduler</b>	TaskPoolScheduler processes scheduled tasks on the default taskpool.
<b>Partitioner Eventloop</b>	Incoming SensorData is de-multiplexed and offloaded onto a player-specific EventLoop (each player has its own EventLoop). PlayerData is also calculated on player-specific Eventloop unlike above strategies. Similar to Imperative NewThread-SensorData strategy.



# Reactive Strategies:



## Query-1

NewThread  
Scheduler shows  
**27%** higher  
throughput than  
single-threaded

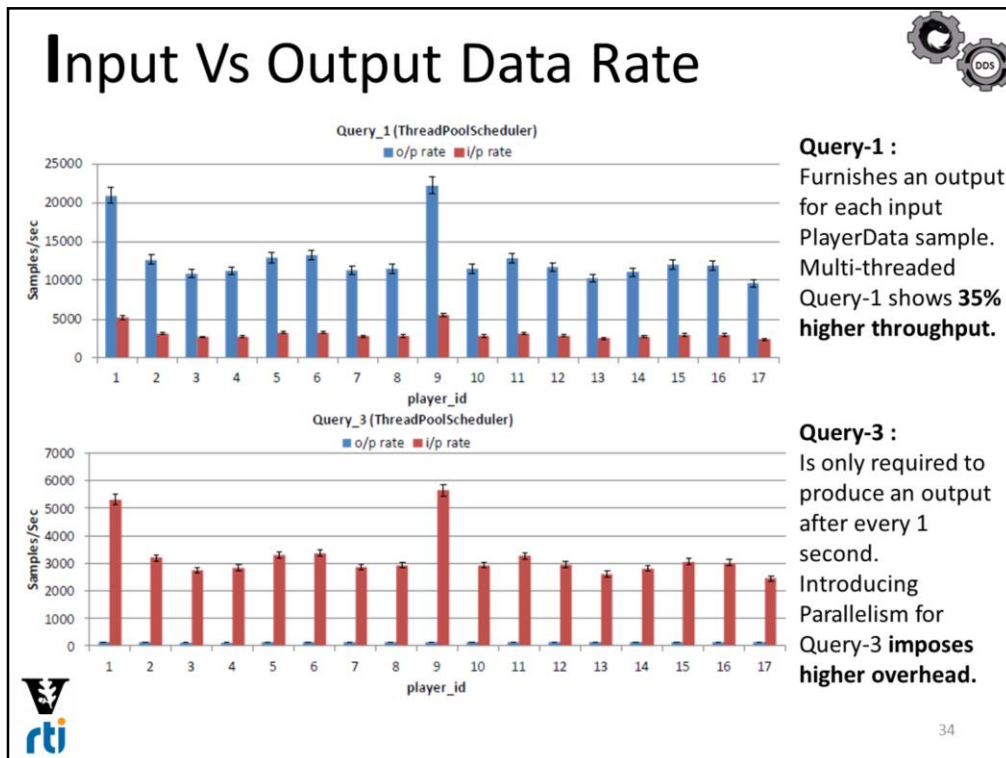
## Query-3:

NewThread  
Scheduler shows  
**3%** higher  
throughput than  
single-threaded

## Query-1\_3:

ThreadPool  
Scheduler shows  
**33%** higher  
throughput than  
single-threaded

33



Since Rx provides abstractions which make concurrency management declarative, **testing different concurrency options for an application requires negligible effort**. By changing a few lines of code we can test whether introducing parallelism provides increased performance gain (e.g., query 1) which is worth the added overhead or degrades it due to greater overhead (e.g., query 3).

In contrast, gaining such insights by testing different implementation alternatives in the imperative approach was more complex, requiring a fair amount of changes in the code.

# Outline



- **Overview of Technologies**
  - ❖ OMG DDS
  - ❖ Microsoft Rx
- **Rx4DDS.NET**
- **Evaluation Case-Study**
  - ❖ DEBS 2013 Grand Challenge
- **Benefits of Rx4DDS.NET**
  - ❖ Declarative Style
  - ❖ Composability and Program structure
  - ❖ Flexible Component Boundaries
  - ❖ Declarative Concurrency Management
  - ❖ Backpressure
- **Experimental Results**
- **Conclusion**



# Conclusion



- Rx4DDS.NET unifies local and distributed stream processing aspects under a common dataflow programming model.
- It enables the development of highly expressive and composable programs that achieve data distribution via DDS and data processing using Rx.
- It provides the following advantages:
  - ❖ Declarative Style of programming
  - ❖ Composability
  - ❖ Flexible Component Boundaries
  - ❖ Declarative management of concurrency





**T**hank You.







Questions?

