Industry Paper: Reactive Stream Processing for Data-centric Publish/Subscribe

Shweta Khare, Kyoungho An, Aniruddha Gokhale
Dept of EECS, Vanderbilt University
Nashville, TN 37212, USA
{shweta.p.khare,kyoungho.an,a.gokhale}@vanderbilt.edu

Sumant Tambe, Ashish Meena
Real-Time Innovations
Sunnyvale, CA 94089, USA
{sumant,ashish}@rti.com

ABSTRACT

The Internet of Things (IoT) paradigm has given rise to a new class of applications wherein complex data analytics must be performed in real-time on large volumes of fast-moving and heterogeneous sensor-generated data. Such data streams are often unbounded and must be processed in a distributed and parallel manner to ensure timely processing and delivery to interested subscribers. Dataflow architectures based on event-based design have served well in such applications because events support asynchrony, loose coupling, and helps build resilient, responsive and scalable applications. However, a unified programming model for event processing and distribution that can naturally compose the processing stages in a dataflow while exploiting the inherent parallelism available in the environment and computation is still lacking. To that end, we investigate the benefits of blending Reactive Programming with data distribution frameworks for building distributed, reactive, and high-performance stream-processing applications. Specifically, we present insights from our study integrating and evaluating Microsoft .NET Reactive Extensions (Rx) with OMG Data Distribution Service (DDS), which is a standards-based publish/subscribe middleware suitable for demanding industrial IoT applications. Several key insights from both qualitative and quantitative evaluation of our approach are presented.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; C.2.4 [**Programming Languages**]: Language constructs and features

Keywords

Reactive Programming, Reactive Extensions (Rx), Stream Processing, Data Distribution Service (DDS), Publish/Subscribe

DEBS '15, June 29 - July 03, 2015, Oslo, Norway
ACM 978-1-4503-3286-6/15/06...$15.00.
http://dx.doi.org/10.1145/2675743.2771880.

1. INTRODUCTION

The *Internet of Things* (IoT) is a significant 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 [8]. 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. Industrial IoT are also called Cyber-Physical Systems (CPSs). They share several key cross-cutting aspects. First, they are often large-scale, distributed systems comprising several, potentially mobile, publishers of information that produce large volumes of asynchronous events. Second, the resulting unbounded asynchronous streams of data must be combined with one-another and with historical data and analyzed in a responsive manner. While doing so, the distributed set of resources and inherent parallelism in the system must be effectively utilized. Third, the analyzed information must be transmitted downstream to a heterogeneous set of subscribers. In essence, the emerging IIoT systems can be understood as a *distributed asynchronous dataflow*. The key challenge lies in developing a dataflow-oriented programming model and a middleware technology that can address both *distribution* and *asynchronous processing* requirements adequately.

The distribution aspects of dataflow-oriented systems can be handled sufficiently by data-centric publish/subscribe (pub-/sub) technologies [13], such as Object Management Group (OMG)'s Data Distribution Service (DDS) [19]. DDS is an event-driven publish-subscribe middleware that promotes asynchrony and loose-coupling between data publishers and subscribers which are decoupled with respect to (1) *time* (*i.e.*, they need not be present at the same time), (2) *space* (*i.e.*,they may be located anywhere), (3) *flow* (*i.e.*, data publishers must offer equivalent or better quality-of-service (QoS) than required by data subscribers), (4) *behavior* (*i.e.*, business logic independent), (5) platforms, and (6) programming languages. In fact, as specified by the Reactive Manifesto [5], event-driven design is a pre-requisite for building systems that are *reactive*,*i.e.* readily responsive to incoming data, user interaction events, failures and load variations-traits which are desirable of critical IIoT systems. Moreover, asynchronous event-based architectures unify scaling up (e.g., via multiple cores) and scaling out (e.g., via distributed compute nodes) while deferring the choice of the scalability mechanism at deployment-time without hiding

the network from the programming model. Hence, the asynchronous and event-driven programming model offered by DDS makes it particularly well-suited for demanding IIoT systems.

However, the data processing aspects, which are local to the individual stages of a distributed dataflow, are often not implemented as a dataflow due to lack of sufficient composability and generality in the application programming interface (API) of the pub/sub middleware. 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 demultiplexing. These primitives, however, are designed for data and meta-data *delivery* [1] as opposed to *processing*. Further, the lack of proper abstractions forces programmers to develop event-driven applications using the observer pattern– disadvantages of which are well documented [16].

A desirable programming model is one that provides a first-class abstraction for streams; and one that is composable. Additionally, it should provide an exhaustive set of reusable coordination primitives for reception, demultiplexing, multiplexing, merging, splitting, joining two or more data streams. We go on to argue in this paper that a dataflow programming model that provides the coordination primitives (combinators) implemented in functional programming style as opposed to an imperative programming style yields significantly improved expressiveness, composability, reusability, and scalability. [2] A desirable solution should enable an end-to-end dataflow model that unifies the local as well as the distribution aspects.

To that end we have focused on composable event processing inspired by Reactive Programming [7] and blended it with data-centric pub/sub. 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. Hence, the application can be visualized as a *data-flow*, wherein data and respectively changes thereof implicitly flow through the application [21, 9]. Functional Reactive Programming (FRP) [12] was originally developed in the context of pure functional language, Haskell. and has since been implemented in other languages, for example, Scala.React (Scala) [16], FlapJax (Javascript) [18], Frappe (Java) [11].

Composable event processing–a modern variant[3] of FRP– is an emerging new way to create scalable reactive applications [22], which are applicable in a number of domains including HD video streaming [4] and UIs. It offers a declarative approach to event processing wherein program specification amounts to "what" (i.e., declaration of intent) as opposed to "how" (looping, explicit state management, etc.). State and control flow are hidden from the programmers, which enables programs to be visualized as a data-flow. Furthermore, functional style of programming elegantly supports composability of asynchronous event streams. It tends to avoid shared mutable state at the application-level, which is instrumental for multicore scalability. Therefore, there is a compelling case to systematically blend reactive programming paradigm with data-centric pub/sub mechanisms for realizing emerging IIoT applications.

In this paper we have combined concrete instances of pub-/sub technology and reactive programming, to evaluate and demostrate 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) [3]. This paper makes the following contributions:

1. We show the strong correspondence between the distributed asynchronous dataflow model of DDS and the local asynchronous dataflow model of Rx. We integrated the two technologies in the Rx4DDS.NET open-source library. The remarkable overlap between the two technologies allows us to substitute one for the other and overcome the missing capabilities in both, such as the lack of a composable data processing API in DDS and the lack of interprocess communication and back-pressure support in .NET Rx; [4]

2. We present the advantages of adopting functional style of programming for real-time stream processing. Functional *stream* abstractions enable seamless composability of operations and preserve the conceptual "shape" of the application in the actual code. Furthermore, state management for sliding time-window, event synchronization and concurrency management can be delegated to the run-time which is made possible by the functional tenets, such as the immutable state.

3. We evaluate the Rx4DDS.NET library using a publicly available high-speed sensor data processing challenge [14]. We present the ease and the effect of introducing concurrency in our functional implementation of "queries" running over high-speed streaming data. Our dataflow programming admits concurrency very easily and improves performance (up to 35%).

4. Finally, we compare our functional implementation with our imperative implementation of the same queries in C#. We highlight the architectural differences and the lessons learned with respect to "fitness for a purpose" of stream processing, state management, and configurability of concurrency.

The rest of the paper is organized as follows: Section 2 compares our proposed solution with prior efforts; Section 3 describes our reactive solution that integrates Rx and DDS; Section 4 reports on both our qualitative and quantitative experience building a reactive solution to solve a specific case study problem; and finally Section 5 provides concluding remarks and lessons learned.

[1] Strictly, DDS API is designed for retrieving the state of an object rather than individual updates about an object

[2] Microsoft Robotics Coordination and Concurrency Runtime (CCR) and Robot Operating System (ROS) http://wiki.ros.org/

[3] without continuous time abstraction and denotation semantics

[4] Reactive Streams project [1], RxJava [2] support backpressure

2. RELATED WORK

A research roadmap towards applying reactive programming in distributed event-based systems has been presented in [20]. In this work the authors highlight the key research challenges in designing distributed reactive programming systems to deal with "data-in-motion". Our work on Rx4DDS-.NET addresses the key open questions raised in this prior work. In our case we are integrating Reactive Programming with DDS that enables us to build a loosely coupled, highly scalable and distributed pub/sub system, for reactive stream processing.

Nettle is a domain-specific language developed in Haskell, a purely-functional programming language, to solve the low-level, complex and error-prone problems of network control [23]. Nettle uses Functional Reactive Programming (FRP) including both the discrete and continuous abstractions and has been applied in the context of OpenFlow software defined networking switches. Although the use case of Nettle is quite different from our work in Rx4DDS.NET, both approaches aim to demonstrate the integration of reactive programming with an existing technology: we use DDS where as Nettle uses OpenFlow.

The ASEBA project demonstrates the use of reactive programming in the event-based control of complex robots [15]. The key reason for using reactive programming was the need for fast reactivity to events that arise at the level of physical devices. Authors of the ASEBA work argue that a centralized controller for robots adds substantial delay and presents a scalability issue. Consequently, they used reactive programming at the level of sensors and actuators to process events as close to the source as possible

Our work on Rx4DDS.NET is orthogonal to the issues of where to place the reactive programming logic. In our case such a logic is placed with every processing element, such as the subscriber that receives the topic data.

Prior work on Eventlets [6] comes close to our work on Rx4DDS.NET. Eventlets provides a container abstraction to encapsulate the complex event processing logic inside a component so that a component-based service oriented architecture can be realized. The key difference between Eventlets and Rx4DDS.NET is that the former applies to service oriented architectures and component-based systems, while our work is used in the context of publish/subscribe systems. Although this distinction is evident, there are ongoing efforts to merge component abstractions with pub/sub systems such that we may be able to leverage component abstractions in our future work.

Functional programming style (akin to Rx) has been used effectively in Spark Streaming [24] in the context of Lambda Architecture (LA) [17] to write business logic just once using functional combinator libraries and reuse that implementation for both real-time and batch processing of data. In a typical LA, the batch layer maintains the master data whereas the "speed layer" compensates for the high latency of the batch layer and also trades accuracy for speed. Business queries represented using the functional style abstract away the source of data (batch/streaming) and improve code reuse.

An ongoing project called Escalier [10] has very similar goals as our work. Escalier provides a Scala language binding for DDS. The future goals of the Escalier project are to provide a complete distributed reactive programming framework, however, we have not yet found sufficient related publications nor are we able to determine from their github site whether this project is actively maintained or not. Similarly, OpenDDS [27] and OpenSplice [25] describe integration of DDS with Rx and other functional-style stream processing technologies. However, to the best of our knowledge, our work includes the most comprehensive comparison and evaluation of the two technologies together.

3. DESIGN OF THE RX4DDS.NET LIBRARY

We now describe our approach to realizing Rx4DDS.NET. To better understand our solution, we first provide a brief overview of DDS and Rx. We then illustrate some drawbacks of our imperative solution implemented only using DDS, which motivates the need for Rx4DDS.NET.

3.1 Overview of OMG DDS Data-Centric Pub-/Sub Middleware

OMG DDS is a *data-centric* middleware that understands the schema/structure of "data-in-motion". The schemas are explicit and support keyed data types much like a primary key in a database. Keyed data types partition the global data-space into logical streams (*i.e.*, instances) of data that have an observable lifecycle.

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, respectively. DDS ensures that the endpoints are compatible with respect to the topic name, data type, and the QoS policies.

3.2 Microsoft Reactive Extensions (Rx)

Microsoft Reactive Extensions (Rx) [3] is a library for composing asynchronous and event-based programs. Using Rx, programmers represent asynchronous data streams with *Observables*, query asynchronous data streams using a library of composable functional *Operators*, and parameterize the concurrency in the asynchronous data streams using *Schedulers*. Rx offers many built-in primitives for filtering, projecting, aggregating and composing multiple sources of events. Rx has been classified as a "cousin of reactive programming" [7] 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).

3.3 Challenges Manifested In Our Imperative Solution

We implemented the DEBS 2013 grand-challenge queries [14] in an imperative style using DDS and C#. This experience highlighted a number of challenges with *our* imperative solution which motivates our work on Rx4DDS.NET. We describe these challenges below:

- **Lack of built-in streaming constructs** – We had to manually code the logic and maintain relevant state information for merging, joining, multiplexing, de-multiplexing and capturing data dependencies between multiple streams of data.

- **Lack of a concurrency model to scale up event processing by employing multiple cores** – Since DDS utilizes a single dedicated thread for a DataReader to receive an input event, there was a need to manually create threads or a thread pool to exploit available cores for concurrent data processing.

- **Lack of a reusable library for sliding time windows** – A system for complex event processing typically requires handling events based on different sliding time-windows (*e.g.*, last one hour or one week). A reusable library for sliding time-windows which also operates with other streaming constructs is required. In our imperative approach, we had to reinvent the solution every time it was needed.

- **Lack of flexibility in component boundaries** – In DDS, data-writers/readers are used for publishing/-subscribing intermediate results between processing stages. However, this approach incurs overhead due to serialization and de-serialization of DDS samples across the data writer-reader boundary, even if event processing blocks are deployed on the same machine. The use of data-writers/readers imposed a hard component boundary and there was no way to overcome that transparently.

3.4 Rx4DDS.NET: Integrating Rx and DDS

To address the challenges with our imperative approach, we designed our reactive programming solution that integrates .NET Reactive Extensions (Rx) framework with DDS. This solution is made available as a reusable library called Rx4DDS.NET. We describe our design by illustrating the points of synergy between the two.

In Rx, asynchronous data streams are represented using Observables. For example, an `IObservable<T>` produces values of type T. Observers subscribe to data streams much like the *Subject-Observer* pattern. Each Observer is notified whenever a stream has a new data using the observer's `OnNext` method. If the stream completes or has an error, the `OnCompleted,` and `OnError` operations are called, respectively. `IObservable<T>` supports chaining of functional operators to create pipelines of data processing operators (a.k.a. combinators).

Some common examples of operators in Rx are `Select, Where, SelectMany, Aggregate, Zip,` etc. Since Rx has first-class support for streams, Observables can be passed and returned to/from functions. Additionally, 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>>.` The keyed streams (`IGroupedObservable<Key,T>`) correspond directly with DDS instances as described next.

In DDS, a topic is a logical data stream in the global dataspace. DataReaders receive notifications when an update is available on a topic. Therefore, a topic of type T maps to Rx's `IObservable<T>.` This conceptual mapping is shown in Figure 1, where the data received by a DataReader is converted into an Rx Observable which is later consumed by downstream query operators (represented by white squares in Figure 1).

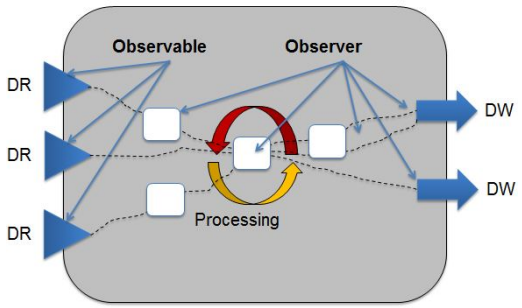

Figure 1: Conceptual Illustration of the Rx4DDS.NET Integration (DR = DataReader, DW = DataWriter)

DDS supports a key field in a data type that represents a unique identifier for data streams defined in a topic. A data stream identified by a key is called instance. If a DataReader uses a keyed data type, DDS distinguishes each key in the data as a separate instance. An instance can be thought of as a continuously changing row in a database table. DDS provides APIs to detect instance lifecycle events including Create, Read, Update, and Delete (CRUD). Since each instance is a logical stream by itself, a keyed topic can be viewed as a stream of keyed streams thereby mapping to Rx's `IObservable<IGroupedObservable<Key,T>>.`

Thus, when our Rx4DDS.NET library detects a new key, it reacts by producing a new `IGroupedObservable<Key,T>` with a new key. Subsequently, Rx operations can be composed on the newly created `IGroupedObservable<Key, T>` for instance-specific processing. As a result, pipelining and data partitioning can be implemented very elegantly using our integrated solution.

Table 1 summarizes how various DDS concepts map naturally to a small number of Rx concepts. 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 middleware uses special types of DDS entities to exchange discovery events with remote peers using predefined *built-in topics*. As introduced in the table, discovery events using built-in topics and communication status events can be received and processed by Rx4DDS.NET API, but they are currently not implemented in our library and forms part of our ongoing improvements to the library.

Due to the similarity in the dataflow models, Rx and DDS are quite interchangeable. Table 1 forms the basis of our integration and the Rx4DDS.NET library. The contract between any two consecutive stages composed with Rx Observables is based on only two notions: (1) the static type of the data flowing across and (2) and the pair of `IObservable` and `IObserver` 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*. No more (or less) information is required for a successful mapping as long as default QoS are used in DDS. The converse is also true, however, only a subset of QoS attributes can be mapped to Rx operators as of this writing. For example, DDS time-based filters can be mapped to Rx's `Sample` operator; Durability QoS with history maps to the `Replay` operator.

Table 1: Mapping of DDS concepts to Rx concepts

DDS Concept	Corresponding Rx Concept and the Rx4DDS.NET API
Topic of type T	An IObservable<T> created using DDSObservable.fromTopic<T>(...). Produces a hot observable. Internally creates a DataReader<T>.
Topic of type T with key-type=Key	An IObservable<IGroupedObservable<Key,T≫ created using DDSObservable.fromKeyedTopic<Key, T>(keySelector) where keySelector maps T to Key. Internally uses a DataReader<T>. Produces a hot observable.
A new instance in a topic of type T	An IGroupedObservable<Key,T> with Key==instance's key. Notified using IObserver<IGroupedObservable<Key,T≫.OnNext(IGroupedObservable<Key,T≫)
Disposal an instance (graceful)	Notified using IObserver<IGroupedObservable<Key,T≫.OnCompleted()
Dispose an instance (not alive, no writers)	Notified using IObserver<IGroupedObservable<Key,T≫.OnError(err)
DataReader<T>.take()	Push new values of T using IObserver<T>.OnNext(T). The fromTopic<T>() and fromKeyedTopic<Key,T>() factories produce hot observables.
DataReader<T>.read()	Push potentially repeated values of T using IObserver<T>.OnNext(T). The readFromDataReader<T>() and readFromDataReader<Key,T>() factories produce cold observables.
A transient local DataReader<T> with history = N	IObservable<T>.Replay(N) which caches the last N samples.
Hard error on a DataReader	Notified using Observer.OnError(err)
Entity status conditions (e.g., deadline missed, sample lost etc.)	Separate IObservable<T> streams per entity where T is communication status types. For example, IObservable<DDS::SampleLostStatus>.
Built-in discovery topics	Keyed observables for each built-in topic. For example, IObservable<IGroupedObservable<Key, T≫ where T=Subscription/Publication/Participant BuiltInTopicData and Key=BuiltinTopicKey.
Read Conditions (parameterizes sample state, view state, and instance state)	IObservable<T>.Where() for filtering on sample state; New IGroupedObservable<Key,T> instance for new view state; and IObserver<IGroupedObservable<Key,T≫.OnCompleted() for disposed instance state.
Query Conditions	IObservable<T>.Where() for content-based filtering.
SELECT * in content-based filter topic (CFT) expression	IObservable<T>.Select(elementSelector) where elementSelector maps T to *
FROM "Topic" in CFT expression	DDSObservable.FromTopic<T>("Topic") or DDSObservable.FromKeyedTopic<Key, T>("Topic") if keyed
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 and IObservable<U>. Other alternatives are Join, CombineLatest, and Zip
Time-based filter	IObservable<T>.Sample(...)

4. EVALUATING RX4DDS.NET BASED SOLUTION

This section reports on our qualitative and quantitative experience in evaluating our Rx4DDS.NET based solution. For the evaluations we have used a case study, which we also describe briefly.

4.1 Case Study: DEBS 2013 Grand Challenge Problem

The ACM International Conference on Distributed Event-based Systems (DEBS) 2013 Grand Challenge problem comprises real-life data from a soccer game and queries in event-based systems [14]. Although the data is recorded in a file for processing, this scenario reflects IoT use cases where streamed data must be processed at runtime and not as a batch job.

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. The challenge problem consists of four distinct queries that must be executed on the incoming streams of data. Figure 2 shows the high-level view of the four query components and the flow of data between them. For brevity we only describe queries 1 and 3 for which we also present experimental results later.

Query 1: The goal of query 1 is to calculate the running statistics for each player. 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: Query 3 requires 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, distinct calculations are required for different time windows. As a result, query 3 must output 16 result streams (a combination of 4 different grid sizes and 4 time windows).

4.2 Qualitative Evaluation of the Rx4DDS.NET Solution

We now evaluate our Rx4DDS.NET based solution along the dimensions of challenges expounded in Section 3.3 and compare it qualitatively with our imperative solution for the case study.

4.2.1 Automatic State Management

Recall that the imperative approach requires additional logic to maintain state and dependencies. For example, in

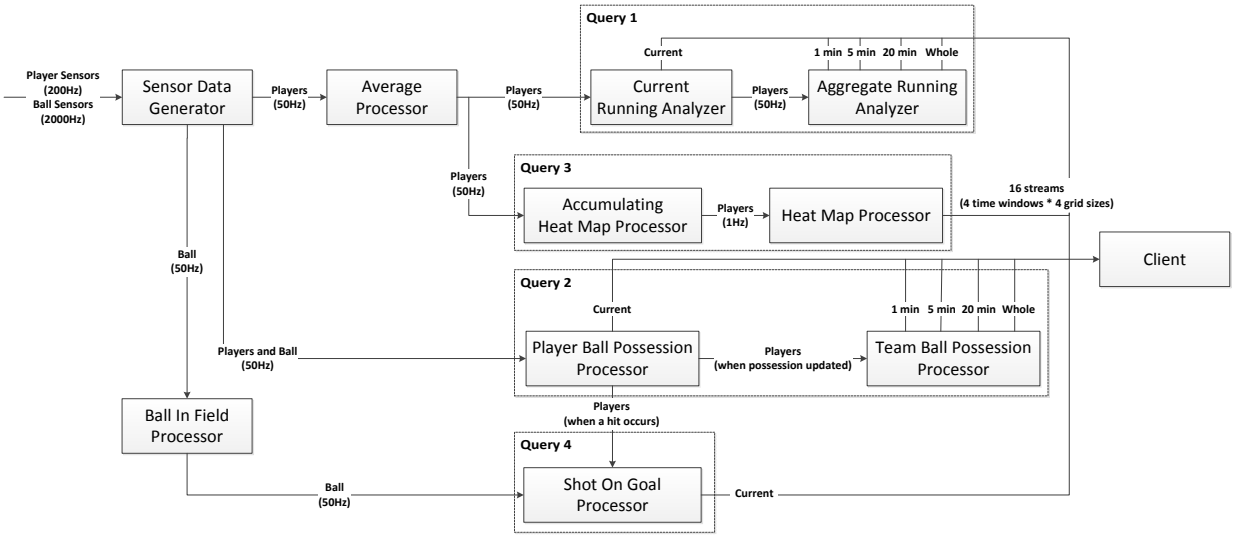

Figure 2: High Level Data Flow Architecture of DEBS 2013 Grand Challenge

the case study, 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 functional style, there is no need to store the sensor values. 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.

In Listing 1, *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. The Marble diagram[5] for CombineLatest is shown in Figure 3.

Listing 1: CombineLatest Operator Example Code

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

As another example of automatic state management, in query 1 the current running statistics need to be computed from average sensor data for each player (*PlayerData*). The distance traveled and average speed of a player (observed in

[5]Marble diagrams are a way to express and visualize how the operators in Rx work. For details see `http://rxwiki.wikidot.com/marble-diagrams`.

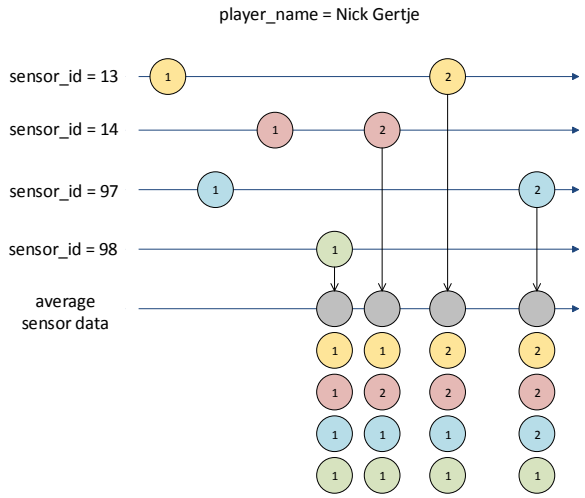

Figure 3: Marble Diagram of CombineLatest Operator

the interval between the arrivals of two consecutive *PlayerData* samples) is calculated. Since our computation depends on the previous and current data samples, we can make use of the built-in `Scan` function and avoid maintaining previous state information manually. `Scan` is a runtime accumulator that will return the result of the accumulator function (optionally taking in a seed value) for each new value of source sequence. Figure 4 shows the marble diagram of the `Scan` operator. In the imperative approach, we employed the middleware cache to maintain previous state.

4.2.2 *Concurrency Model to Scale-up Multi-core Event Processing*

Rx provides abstractions that make concurrency manage-

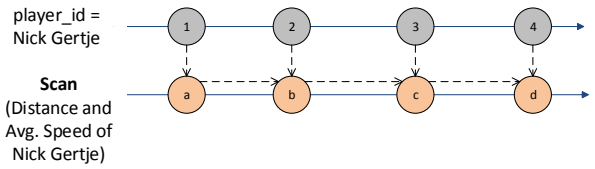

Figure 4: Marble Diagram of Scan Operator

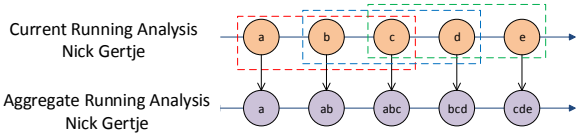

Figure 5: Marble Diagram of Time-window Aggregator

ment 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. Delegating the management of shared state to stream operators also makes the code more easily parallelizable. Implementing the same logic in the imperative approach incurred greater complexity and the code was more verbose with explicit calls for creating and managing the thread pools.

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. In Listing 2, *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`.

Listing 2: Concurrent Event Processing with Multithreading

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

4.2.3 *Library for Computations based on Different Time-windows*

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. In the imperative approach we had to reimplement the necessary functionality and manually maintain pertinent previous state information for the same because DDS does not support a time-based cache which can cache samples observed over a time-window.

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. Figure 5 depicts aggregation performed over a moving time window.

4.2.4 *Flexible Component Boundaries*

Interchangeability of Rx and DDS provides incredible flexibility to the developer in demarcating their component boundaries or points of data distribution. In fact, the points of distribution can be chosen at deployment-time. The imperative solution often does not possess a composable dataflow-oriented structure. Hence, more often than not, developers tend to over-commit to various interprocess communication mechanisms by hard-coding the dependency and eliminating the choice of an alternative mechanism. If scale-out or placement of these components on different machines is required, then this design is desirable, otherwise overcommitment to a specific distribution mechanism isolates the components and imposes "hard" component boundaries. The resulting structure is very rigid and hard to co-locate efficiently. For example, each query processor in our imperative solution is a component. Moving the functionality of one into another is intrusive and cannot be easily accomplished.

In Rx4DDS.NET, a stream of intermediate results can either be distributed over a DDS topic for remote processing or can be used for local processing by chaining stream operators. The details of whether the "downstream" processing happens locally or remotely can be abstracted away using the Dependency Injection pattern [26]. As a consequence, component boundaries become more agile and the decision of data distribution need not be taken at design time but can be deferred until deployment.

In our implementation, developers may choose to distribute data over DDS by simply passing a DDS DataWriter to the `Subscribe` method. Alternatively, for local processing, a `Subject<T>` could be used in place of DDS DataWriter. The choice of a Subject versus a DataWriter is configurable at deployment-time.

Table 2 summarizes the key distinctions between our imperative and Rx4DDS.NET based solution for the case-study along each dimension of the challenges.

4.2.5 *Program Structure*

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, as shown in Listing 3.

In Listing 3, *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* stream in Listing 3) is computed by the function *CurrentRunningAnalysis()* and is subsequently used by *AggregateRunning*()* to compute

Table 2: Comparison of Our Imperative and Reactive Solutions

	Imperative Solution	Reactive Solution
State Management	Manual state management	State-management can be delegated to stream operators
Concurrency Management	Explicit management of low level concurrency	Declarative management of concurrency
Sliding Time-window Computation	Manual implementation of time window abstraction	Built-in support for both discrete and time-based window
Component Boundaries	Inflexible and hard component boundaries	Flexible and more agile component boundaries

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.

Listing 3: Program Structure of Query 1

```
player_streams.Subscribe(player_stream =>
{
    var curr_running=
         player_stream
        .ObserveOn(ThreadPoolScheduler.Instance)
        .CurrentRunningAnalysis()
        .Publish();
   curr_running.AggregateRunningTimeSpan(1);
   curr_running.AggregateRunningTimeSpan(5);
   curr_running.AggregateRunningTimeSpan(20);
   curr_running.AggregateRunningFullGame();

   curr_running.Connect();
}
```

4.2.6 Backpressure

Integration of Rx with DDS allows us to leverage DDS QoS configurations and Real Time Publish Subscribe (RTPS) protocol to implement backpressure across the data reader-writer boundary. DDS offers a variety of QoS policies like *Reliability*, *History* and *Resource Limits* QoS policies that can be tuned to implement the desired backpressure strategy. *Reliability* QoS governs the reliability of data delivery between DataWriters and DataReaders. It can be set to either `BEST_EFFORT` or `RELIABLE` reliability. `BEST_EFFORT` configuration does not use any cpu/memory resources to ensure guaranteeded delivery of data samples. `RELIABLE` configuration, on the other hand, uses ack/nack based protocol to provide a spectrum of reliability guarantees from strict to best-effort. Reliability can be configured with *History* QoS, which specifies how many data samples must be stored by the DDS middleware cache for the DataReader/DataWriter subject to *Resource Limits* QoS settings. It controls whether DDS should deliver only the most recent value (*i.e.*, history depth=1), attempt to deliver all intermediate values (*i.e.*, history depth=unlimited), or anything in between. *Resource Limits* QoS controls the amount of physical memory allocated for middleware entities. If `BEST_EFFORT` QoS setting is used, the DataWriter will drop the samples when the writer side queue (queue size determined by *History* and *Resource Limits* QoS) becomes full. We can use this strategy to cope with a slow subscriber or bursty input data rates if the application semantics support transient loss of data. On the other hand, if we use `RELIABLE` configuration, backpressure can be supported across the data reader-writer boundary. If the DataReader is not fast enough, it will start buffering the incoming samples upto a pre-configured limit (including unlimited, as configured using *History* and *Resource Limits* QoS) before throttling down the DataWriter in accordance with the reliability protocol semantics. However, this backpressure is only limited to work across two DDS entities. Local processing stages implemented in Rx .NET do not support backpressure. Hence, if operators with unbounded buffer sizes (e.g., *ObserveOn*, *Zip*) are used then we may observe an unbounded increase in queue lengths, arbitrarily large response times or out-of-memory exceptions.

Unlike Rx NET., the Reactive-Streams specification [1] implements a *dynamic push-pull model* for implementing backpressure between local processing stages. Their model can shift dynamically from being push-based (when the consumer can keep up with incoming data rate) to a pull-based model if the consumer is getting overwhelmed. The consumer specifies its "demand" using a backpressure channel to throttle the source. The producer can also use the "demand" specifications of downstream operators to perform intelligent load-distribution. In the future, we plan to integrate the Reactive-Streams specification with DDS for end-to-end backpressure semantics.

4.3 Quantitative Evaluation of Rx4DDS.NET

To assess and compare the performance of Rx4DDS.NET library with that of the imperative approach, we implemented the DEBS 2013 Grand Challenge queries in an imperative style using C# so that both implementations used C#. Specifically, we compare the performance of our imperative and Rx4DDS.NET solutions under single threaded and multi-threaded query implementations. All the tests have been performed on a host with two 6-core AMD Opteron 4170 HE, 2.1 GHz processors and 32 GB RAM. The raw sensor stream was published by a DDS publisher by reading out the sensor data file in a separate process, while the queries were executed in another process by subscribing to the raw sensor stream published over DDS. Interprocess communication happens over shared-memory.

We implemented the following strategies in the imperative solution for parallelizing query execution along the lines of available Rx schedulers: `SeparateThread`, `ThreadPool-SensorData`, `ThreadPool-PlayerData`, `NewThread-SensorData` and `NewThread-PlayerData`. In the `SeparateThread` strategy, `SensorData` is received on one thread while the entire query execution is offloaded to a separate thread (all player streams are processed on this separate thread). The `ThreadPool-SensorData` strategy offloads the received `SensorData` on a threadpool such that each player's `PlayerData` calculation and subsequent player-specific processing happens on the threadpool. In the `ThreadPool-PlayerData`

Table 3: Performance Comparison of Rx4DDS.NET over Imperative Solution

Rx Scheduler over Imperative Strategy	query_1 %throughput difference	query_1 Std. Dev.	query_3 %throughput difference	query_3 Std. Dev.	query_1_3 %throughput difference	query_1_3 Std. Dev
Rx single-thread over Imperative single-thread	-9.26	6.29	-4.3	7.3	1.19	5.67
Rx NewThread scheduler over Imperative NewThread-PlayerData Strategy	-6.7	4.44	-8.61	2.9	-3.75	3.28
Rx ThreadPool scheduler over Imperative ThreadPool-SensorData Strategy	-8.73	6.55	-5.47	4.56	-5.3	6.05
Rx Partitioner Eventloop over Imperative NewThread-SensorData Strategy	-13.87	7.04	-15.93	6.43	-10.87	3.67

strategy, the `PlayerData` is calculated from `SensorData` on the thread that receives `SensorData` from DDS; thereafter the calculated `PlayerData` is offloaded to a threadpool for further player specific processing. The `NewThread-SensorData` strategy creates a designated thread for processing each player's data. The received `SensorData` is dispatched to its specific player thread, which computes that player's `PlayerData` and processes it further. The `NewThread-PlayerData` strategy also creates a separate thread for processing each player's data. However, the `PlayerData` is calculated from `SensorData` on the thread that receives data from DDS, which is then dispatched to the player-specific thread for further processing.

Figure 6 presents the performance of different imperative strategies over single-threaded implementations of `query_1`, `query_3` and `query_1_3` (runs both queries 1 and 3 together). Each query was run ten times and the error bars in the graphs denote one standard-deviation of values. For `query_1`, the average throughput gains per strategy over the single threaded implementation of `query_1` are, respectively, 29% for the `SeparateThread` strategy, 35% for the `ThreadPoolSensorData` strategy, 23% for the `ThreadPool-PlayerData` strategy, 32% for the `NewThread-SensorData` strategy and 24% for the `NewThread-PlayerData` strategy. For `query_3`, the `SeparateThread` strategy shows an average of 40%, the `ThreadPool-SensorData` strategy shows an average of 12%, the `ThreadPool-PlayerData` strategy shows an average of 3%, the `NewThread-SensorData` shows an average of 15% and the `NewThread-PlayerData` strategy shows an average of 7% higher throughput than single-threaded implementation of `query_3`. For `query_1_3`, the `SeparateThread` strategy shows an average of 34%, the `ThreadPool-SensorData` strategy shows an average of 45%, the `ThreadPool-PlayerData` strategy shows an average of 42%, the `NewThread-SensorData` strategy shows an average of 47% and the `NewThread-PlayerData` strategy shows an average of 39% higher throughput than single-threaded implementation of `query_1_3`.

To evaluate multi-threaded query implementations in our Rx4DDS.NET solution, we made use of the built-in Rx schedulers as shown in Listing 3. `ObserveOn` in Listing 3 causes each player stream(`player_stream`)'s data to get offloaded on the specified scheduler and all downstream processing of that player's `PlayerData` takes place on the specified scheduler passed to `ObserveOn`. Hence, in this case `PlayerData` gets calculated on the thread which receives data from DDS, but subsequent processing is offloaded to the specified `ObserveOn` scheduler. Rx offers many built-in schedulers such as the `EventLoopScheduler`, `NewThreadScheduler`, `ThreadPoolScheduler`, `TaskPoolScheduler`, etc. for parameterizing the concurrency of the application. `EventLoopScheduler` provides a dedicated thread which processes scheduled tasks

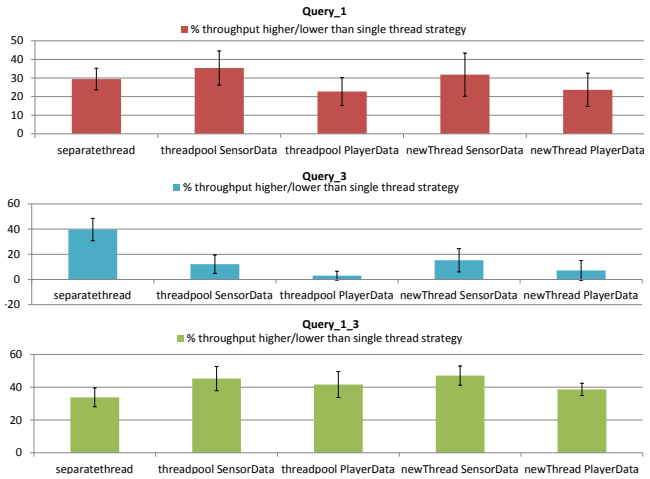

Figure 6: Performance of Imperative Strategies over Single Threaded implementation

in a FIFO fashion; `NewThreadScheduler` processes each scheduled task on a new thread; `ThreadPoolScheduler` processes the scheduled tasks on the default threadpool while `TaskPoolScheduler` processes scheduled tasks on the default taskpool. Apart from using `ObserveOn` to process each player's `PlayerData` using a different scheduler, we also tested a variant test-case named `Partitioner EventLoop`, wherein the incoming `SensorData` is de-multiplexed and offloaded onto a player-specific `EventLoop` (each player has its own `EventLoop`) which will first calculate `PlayerData` and then perform further processing. This is similar to imperative `NewThread-SensorData` strategy, wherein each player thread is also responsible for calculating `PlayerData` from received `SensorData`.

Figure 7 presents the performance of different Rx schedulers over single-threaded implementations of `query_1`, `query_3` and `query_1_3`. For `query_1`, `EventLoopScheduler` shows an average of 3%, `NewThreadScheduler` shows an average of 27%, `ThreadPoolScheduler` shows an average of 23%, `TaskPoolScheduler` shows an average of 25% and `Partitioner EventLoop` shows an average of 25% increase in throughput over single threaded implementation. For `query_3`, `EventLoopScheduler` shows an average of 20% lower performance, while `NewThreadScheduler` shows an average of 3%, `ThreadPoolScheduler` shows an average of 2%, `TaskPoolScheduler` shows an average of .04% and `Partitioner EventLoop` shows an average of 1% increase in performance over single threaded implementation. For `query_1_3`, `EventLoopScheduler` shows an average of 12%, `NewThre-`

adScheduler shows an average of 32%, ThreadPoolScheduler shows an average of 33%, TaskPoolScheduler shows an average of 30% and Partitioner EventLoop shows an average of 30% increase in performance over single threaded solution.

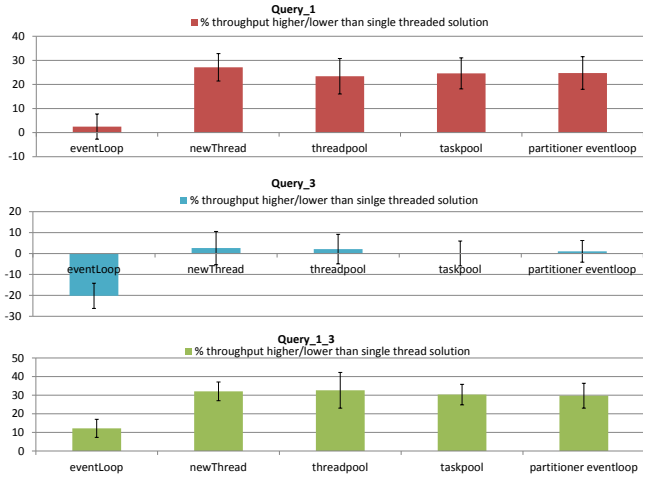

Figure 7: Performance of Different Rx schedulers over Single Threaded implementation

Query_1 processes each player's aggregate running data for all four time-windows, i.e. 1 min, 5 mins, 20 mins and full-game duration, which is updated for each input PlayerData sample. Query_1 shows inherent parallelism wherein each player's data can be processed independently of each other in parallel. The multi-threaded implementation of query_1 shows an increased performance with a maximum performance gain of 35% over single threaded implementation. Query_3, wherein each player's heatmap is calculated for all four time-windows(1min, 5mins, 10mins and full-game duration), also shows inherent parallelism in that each player's heatmap information can be calculated independently. However, query_3 is only required to furnish an update after every 1 second (based on sensor timestamps) unlike query_1 which furnishes an update for each input sample. Hence in case of query_3 we find that introducing parallelism imposes a greater overhead without significant performance gain. Figure 8 presents the difference in input and output data-rates for query_1 and query_3 in our Rx4DDS.NET based implementation parameterized with ThreadPoolScheduler.

Table 3 compares the difference in the performance of Rx schedulers over its corresponding imperative solution strategy. While it is expected that the Rx library will impose some overhead, it offers several advantages due to its declarative approach towards system development, improved expressiveness and composability. 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

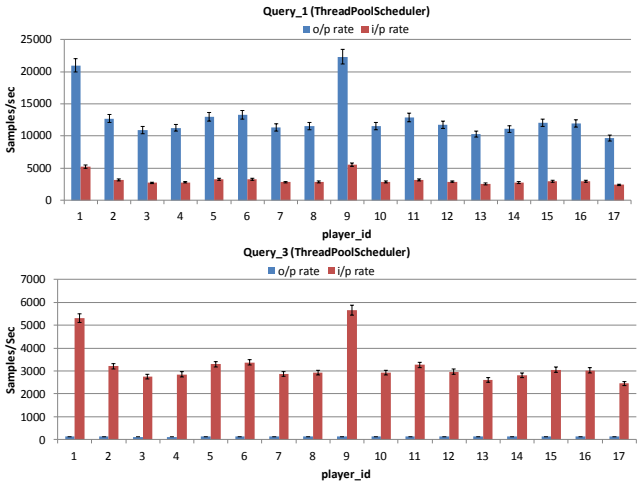

Figure 8: Input Vs Output data rate for Rx4DDS.NET implementation of Query_1 and Query_3 with ThreadPoolScheduler

complex, requiring a fair amount of changes in the code.

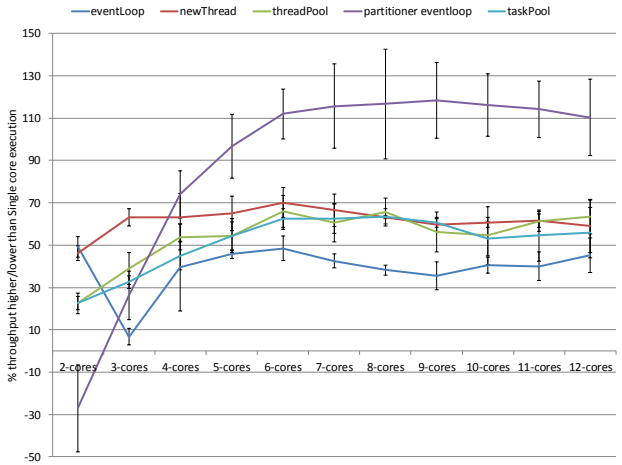

Figure 9: Throughput of Rx4DDS.NET implementation of Query_1 with different schedulers under increasing number of cores.

To verify whether the performance of multi-threaded query implementations scale with the availability of additional hardware resources, we restricted the query-processor process to run using a specified number of available cores. Figure 9 shows the throughput of query_1 configured with different Rx schedulers under increasing number of cores assigned to the query-processor process. In accordance with Amdhal's law, we observe that the throughput of query_1 increases initially as the number of assigned cores increases, but beyond that we see no difference in the performance even if we keep increasing the number of assigned cores. However, we note some erratic behavior in the case of EventLoopScheduler for 3-cores and Partitioner EventLoop for 2-cores, where the performance drops sharply and we are investigating the

reason behind this.

5. CONCLUSIONS

Reactive programming is increasingly becoming important in the context of real-time stream processing for big data analytics. While reactive programming supports event-driven design, most of the generated data must be disseminated from a large variety of sources (*i.e.*, publishers) to numerous interested entities, called subscribers while maintaining anonymity between them. These properties are provided by pub/sub solutions, such as the OMG DDS, which is particularly suited towards performance-sensitive applications. Bringing these two technologies together helps solve both the scale-out problem (*i.e.*, by using DDS) and scale-up using available multiple cores on a single machine (*i.e.*, using reactive programming).

This paper describes a concrete realization of blending the Rx .NET reactive programming framework with OMG DDS, which resulted in the Rx4DDS.NET library. Our solution was evaluated and compared against an imperative solution we developed using DDS and C# in the context of the DEBS 2013 grand challenge problem. The following lessons were learned from our team effort and alludes to future work we plan to pursue in this space.

- The integration of Rx with DDS as done in the Rx4DDS.NET library unifies the local and distributed stream processing aspects under a common dataflow programming model. It allows highly composable and expressive programs that achieve data distribution using DDS and data processing using Rx with a seamless end-to-end dataflow architecture that is closely reflected in the code.

- Our quantitative results indicate that Rx parameterizes concurrency and avoids application-level shared mutable state that makes multi-core scalability substantially easier. We showed increase (up to 35%) in performance of `Query_1` by simply configuring the schedulers in Rx.

- Our future work includes enhancing Rx4DDS.NET library to map all available DDS features with Rx, to identify most commonly used stream processing constructs which can be distilled to be a part of this reusable library. Our goal is to utilize it in the context of large-scale industrial IoT applications, such as transportation, smart grid, medical, and industrial internet.

The Rx4DDS.Net framework and the implementation of the case study queries are available for download from `https://github.com/rticommunity/rticonnextdds-reactive`.

6. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation CAREER CNS 0845789 and AFOSR DDDAS FA9550-13-1-0227. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF and AFOSR.

7. REFERENCES

[1] Reactive-Streams. `http://www.reactive-streams.org/`.

[2] RxJava. `https://github.com/ReactiveX/RxJava`.

[3] The Reactive Extensions (Rx). `http://msdn.microsoft.com/en-us/data/gg577609.aspx`.

[4] Reactive Programming at Netflix. `http://techblog.netflix.com/2013/01/reactive-programming-at-netflix.html`, 2013.

[5] The Reactive Manifesto. `http://www.reactivemanifesto.org`, 2013.

[6] S. Appel, S. Frischbier, T. Freudenreich, and A. Buchmann. Eventlets: Components for the Integration of Event Streams with SOA. In *Service-Oriented Computing and Applications (SOCA), 2012 5th IEEE International Conference on*, pages 1–9, Dec 2012.

[7] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, Aug. 2013.

[8] L. Coetzee and J. Eksteen. The internet of things - promise for the future? an introduction. In *IST-Africa Conference Proceedings, 2011*.

[9] G. H. Cooper and S. Krishnamurthi. Embedding Dynamic Dataflow in a Call-by-value Language. In *Programming Languages and Systems*, pages 294–308. Springer, 2006.

[10] A. Corsaro. Escalier: The Scala API for DDS. `https://github.com/kydos/escalier`.

[11] A. Courtney. Frappé: Functional reactive programming in java. In *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, PADL '01, pages 29–44, London, UK, UK, 2001. Springer-Verlag.

[12] C. Elliott and P. Hudak. Functional Reactive Animation. In *ACM SIGPLAN Notices*, volume 32, pages 263–273. ACM, 1997.

[13] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*

[14] Z. Jerzak and H. Ziekow. The ACM DEBS 2013 Grand Challenge. `http://www.orgs.ttu.edu/debs2013/index.php?goto=cfchallengedetails`, 2013.

[15] S. Magnenat, P. ReÌĄtornaz, M. Bonani, V. Longchamp, and F. Mondada. ASEBA: A Modular Architecture for Event-Based Control of Complex Robots. *Mechatronics, IEEE/ASME Transactions on*, 16(2):321–329, April 2011.

[16] I. Maier and M. Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, 2012.

[17] N. Marz and J. Warren. *Big Data: Principles and Best Practices of Scalable Real-time Data Systems*. O'Reilly Media, 2013.

[18] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A Programming Language for Ajax Applications. In *ACM SIGPLAN Notices*, volume 44, pages 1–20. ACM, 2009.

[19] OMG. The Data Distribution Service specification, v1.2. `http://www.omg.org/spec/DDS/1.2`, 2007.

[20] G. Salvaneschi, J. Drechsler, and M. Mezini. Towards Distributed Reactive Programming. In R. Nicola and C. Julien, editors, *Coordination Models and Languages*, volume 7890 of *Lecture Notes in Computer Science*, pages 226–235. Springer Berlin Heidelberg, 2013.

[21] G. Salvaneschi, P. Eugster, and M. Mezini. Programming with implicit flows. *IEEE Software*, 31(5):52–59, 2014.

[22] D. Synodinos. Reactive Programming as an Emerging Trend. `http://www.infoq.com/news/2013/08/reactive-programming-emerging`, 2013.

[23] A. Voellmy and P. Hudak. Nettle: Taking the Sting Out of Programming Network Routers. In R. Rocha and J. Launchbury, editors, *Practical Aspects of Declarative Languages*, volume 6539 of *Lecture Notes in Computer Science*, pages 235–249. Springer Berlin Heidelberg, 2011.

[24] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized Streams: An Efficient and Fault-tolerant Model for Stream Processing on Large Clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[25] Building Reactive Data-centric Applications with Vortex, Apache Spark and ReactiveX. `http://www.prismtech.com/products/vortex/resources/youtube-videos-slideshare/building-reactive-data-centric-applications-vort`.

[26] Dependency Injection Pattern. `https://msdn.microsoft.com/en-us/magazine/cc163739.aspx`.

[27] Reactive Open DDS. `http://www.ociweb.com/resources/news/2014/11/05/reactive-opendds-part-i`.