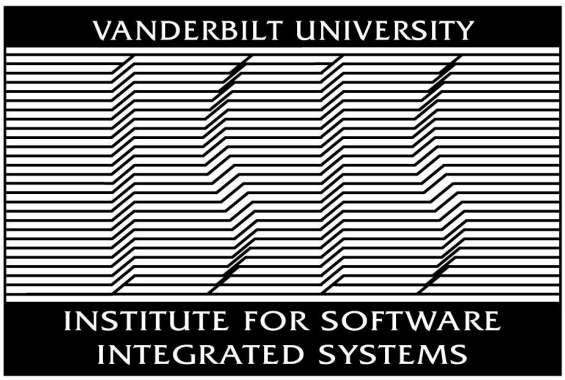

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, 37212

Scalable Reactive Stream Processing Using DDS and Rx

Shweta Khare, Sumant Tambe, Kyoungho An,
Aniruddha Gokhale, Paul Pazandak

TECHNICAL REPORT

ISIS-14-103

April, 2014

Scalable Reactive Stream Processing Using DDS and Rx

Shweta Khare[1], Sumant Tambe[2], Kyoungho An[1], Aniruddha Gokhale[1], and Paul Pazandak[2]

[1] Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN 37212, USA
[2] Real-Time Innovations
Sunnyvale, CA 94089, USA

Abstract. Event-driven design is fundamental to developing resilient, responsive, and scalable reactive systems as it supports asynchrony and loose coupling. The OMG Data Distribution Service (DDS) is a proven event-driven technology for building data-centric reactive systems because it provides the primitives for decoupling system components with respect to time, space, quality-of-service, and behavior. DDS, by design, supports distribution scalability. However, with increasing core count in CPUs, building multicore-scalable reactive systems remains a challenge. Towards that end, we investigate the use of Functional Reactive Programming (FRP) for DDS applications. Specifically, this paper presents our experience in integrating and evaluating Microsoft .NET Reactive Extensions (Rx) as a programming model for DDS-based reactive stream processing applications. We used a publicly available challenge problem involving real-time complex analytics over high-speed sensor data captured during a soccer game. We compare the FRP solution with an imperative solution we implemented in C++11 along several dimensions including code size, state management, concurrency model, event synchronization, and fitness for the purpose of "stream processing." Our experience suggests that DDS and Rx together provide a powerful infrastructure for reactive stream processing, which allows declarative specification of concurrency and therefore dramatically simplifies multicore scalability.

1 Introduction

Reactive Systems, often found in domains, such as robotics, industrial control, smart grid, animation, and video games, must meet strict timing requirements. Traditionally, reactive systems are long running systems that must respond to the external stimuli at speeds determined by its environment [1]. The Reactive Manifesto [2] describes four essential traits of reactive systems: *event-driven* (*i.e.*, push messages to consumers), *scalability* (*i.e.*, accommodate growing load while maximizing resource usage), *resilience* (*i.e.*, isolate faults), and *responsiveness* (*i.e.*, high degree of predictability). Event-driven design is a prerequisite for the other three traits because it enables loose coupling and asynchrony, which in turn helps to isolate faults when they occur. Asynchronous events unify scaling up (e.g., via multiple cores) and scaling out (e.g., via multiple machines) as a deployment-time configuration decision without hiding the network from the programming model. Finally, asynchrony implies non-blocking behavior, which is critical for responsiveness.

Towards that end the Data Distribution Service (DDS) standard [3] provides a powerful data-centric publish-subscribe technology to implement reactive systems because it promotes loose coupling between distributed system components. DDS is event-driven and asynchronous – capabilities that are desired of reactive systems. The data publishers and subscribers 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.

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. Data/state is maintained by the middleware in a global data-space and is distributed to the interested consumers (*i.e.*, subscribers) when the values change. Keyed data types partition the global data-space into logical streams (*i.e.*, instances) of data that have an observable lifecycle. For example, if all the flights in the US commercial airspace are modeled using DDS, a flight number "DL-1234" may serve as a unique identifier (key) for updates regarding a specific flight.

DDS inherently supports distribution scalability (*i.e.*, scale-out). However, with the changing landscape of modern hardware, reactive systems built using DDS must also support multicore scalability (*i.e.*, scale-up). The DDS specification does not provide any guidance on how to achieve multicore scalability. As a result, multicore scalability depends on the application's concurrency architecture (*e.g.*, explicit multi-threading, task-based, actors). Some architectures, such as Actors [4], that support both aspects of scalability may not be always suitable due to the lack of configurable QoS support, content-aware distribution, standards for portability and interoperability, and tool support all of which are provided by DDS. Despite these advantages, developing highly multicore-scalable applications using DDS remains a challenge due to the lack of a concurrency-friendly programming model. Overcoming this challenge is the goal of this industrial research presented in this paper.

To that end we have investigated the use of Functional Reactive Programming (FRP) for DDS applications. FRP has emerged [5] as a promising new way to create scalable reactive applications and has already shown its potential in a number of domains including robotics [6,7], animation [8], HD video streaming [9], and responsive user interfaces [10]. These domains are reactive because they require interaction with a wide range of inputs ranging from keyboards to machinery. FRP is a declarative approach to system development wherein program specification amounts to "what" as opposed "how". Such a program description can be viewed as a data-flow [11] system where the state and control flow are hidden from the programmers. FRP offers high-level abstractions that avoid the verbosity that is commonly observed in callback-based techniques. Furthermore, FRP avoids shared mutable state at the application-level, which is instrumental for multicore scalability.

Accordingly, this paper presents our experience in integrating and evaluating Microsoft .NET Reactive Extensions (Rx) [12] with DDS. Rx is perhaps the most widely used FRP library available in most mainstream languages including C#, Java, JavaScript, C++, Ruby, Python, and others. 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 and DDS are quite complementary because Rx is based on the *Subject-Observer* pattern, which is analogous to the publish/subscribe pattern of DDS. Furthermore, the core tenet of Rx composition of operations over *values that change over time* complements DDS *instances*, which are data objects that change over time. Consequently, combining Rx with DDS enables a holistic end-to-end dataflow architecture for both data distribution (which is performed by DDS) and processing (which is done by Rx). The resulting applications admit concurrency declaratively and DDS (*i.e.*, distribution) can be introduced elegantly where pure data projections are found in the data-flow.

This paper makes the following contributions.

1. To test our hypothesis regarding multicore scalability and ease of programming using FRP (*i.e.*, Rx with DDS), we describe in Section 3 the challenges we faced in our earlier effort at using a plain, DDS-based imperative programming solution using C++11 for the 2013 Distributed Event-Based Systems (DEBS) Grand Challenge [13]. The challenge problem involves real-time complex analytics over high-velocity sensor data captured during a soccer game.
2. To overcome these limitations using FRP, in Section 4.1 we present the design of a new library (RxDDS.NET) built in C# that adapts DDS data streams to Rx Observables. The library enables composition of functional operators on data instances received over DDS, which lends itself easily to reactive stream processing.
3. In Section 4.2, we present our solution to the DEBS 2013 Grand Challenge Problem using our RxDDS.NET FRP solution. Our implementation adheres to the core tenets of functional programming and avoids application-level shared mutable state in most cases.
4. In Section 4.3, we compare our FRP-based solution with the imperative (C++11) solution to the same Grand Challenge problem and describe our experiences. We present the insights we obtained on the architectural differences and the lessons learned with respect to "fitness for a purpose", code size, state management, concurrency model, and finally, abstractions for stream processing.

The rest of the paper is organized as follows: Section 2 provides background information and compares our proposed solution to prior efforts; Section 3 highlights the limitations in our imperative programming solution to achieve scalability for multicores using the DEBS 2013 challenge problem as a real-world use case; Section 4 describes our FRP solution that integrates Rx and DDS, and reports on our experience building

a FRP-based solution to solve the DEBS 2013 challenge problem; and finally Section 5 provides concluding remarks and lessons learned.

2 Background and Related Work

This section first provides a brief overview of the DDS standard alluding to the details relevant to this paper. Next, we discuss related research comparing these solutions to our approach.

2.1 OMG DDS Data-Centric Pub/Sub Middleware

The OMG Data Distribution Service (DDS) describes a data-centric publish/subscribe standard with support for a number of QoS properties. Using DDS applications can share a *global data space* (or *domain*) governed by data schema specified using the XTypes [14] standard. Each topic data schema is described as a structured data type in Interface Definition Language (IDL). The data type can be keyed on one or more fields. Each key identifies an instance (similar to a primary key in a database table) and DDS provides mechanisms to control the lifecycle of instances. Instance lifecycle supports CRUD (create, read, update, delete) operations. Complex delivery models can be associated with data-flows by simply configuring the topic QoS.

DataWriter and DataReader *DataWriters*(data producers) and *DataReaders*(data consumers) 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. Creating a DataReader with a known topic and data type implicitly crates a *subscription*, which may or may not match with a DataWriter depending upon the QoS.

Data Caching A data sample received by a DataReader is stored in a local cache managed by the middleware. The application accesses this data using one of two functions: `read()` or `take()`; `read()` leaves the data in middleware cache until it is removed by either calling `take()` or it is overwritten by the subsequent data samples. The `Resource Limits QoS` prevents the middleware caches from growing out of bounds. The feature called *query conditions* provides a powerful mechanism to write SQL-like expressions on the data type members and retrieve data samples that satisfy the defined predicates.

DDS Quality-of-Service DDS supports multiple QoS policies. Most QoS policies have request/offered semantics (request by a DataReader and offered by a DataWriter) and can configure the data-flow between each pair of DataReader and DataWriter. We used the following QoS policies in our solution.

- **Reliability QoS**: controls the reliability of the data-flow between DataWriters and DataReaders. `BEST_EFFORT` and `RELIABLE` are two possible alternatives. `BEST_EFFORT` configuration does not use any cpu/memory resources to ensure delivery of samples. `RELIABLE`, on the other hand, uses an ack/nack based protocol to provide a spectrum of reliability guarantees from strict (*i.e.,* fully reliable) to best effort. The reliability can be tuned using the `History QoS` policy.
- **History QoS**: This QoS policy specifies how much data must be stored in DDS middleware caches by a DataWriter or DataReader. 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.
- **Time-based Filter QoS**: High rate data-flows can be throttled using the `Time-based Filter QoS` policy. A `minimum separation` configuration parameter used by this QoS specifies the minimum period between two successive arrivals of data samples. When configured, the DataReader receives only a subset of data.
- **Resource Limits QoS**: This QoS policy controls the memory used to cache samples. The maximum number of samples, maximum number of samples-per-instance and maximum number of instances can be specified.

2.2 Related Work

Reactive Programming and in particular the functional reactive programming (FRP) have been used to solve a number of challenges in distributed and networked systems. Here we provide a sampling of related efforts focussing on event-based systems.

A research roadmap towards applying reactive programming in distributed event-based systems has been presented in [15]. 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 RxDDS.NET addresses the key open questions raised in this prior work. In our case we are integrating FRP with DDS that enables us to build a loosely coupled, highly scalable and distributed publish/subscribe system, which can process complex events stemming from the high velocity data streams at every processing element in the distributed system using FRP.

Nettle is a domain-specific language developed in Haskell to solve the low-level, complex and error-prone problems of network control [16]. Nettle uses FRP including both the discrete and continuous abstractions and has been applied in the context of OpenFlow software defined networking switches. OpenFlow is a centralized but programmable switch which enables the separation of the network control plane from the data plane. Thus, the centralized switch must react to events and take control actions. Although the use case of Nettle is quite different from our work in RxDDS.NET, both approaches aim to demonstrate the integration of FRP with an existing technology: in our case it is FRP with DDS while with Nettle it is FRP with OpenFlow.

The ASEBA project demonstrates the use of reactive programming in the event-based control of complex robots [17]. The key reason for using reactive programming was the need for fast reactivity to events that rise at the level of physical devices. The problem addressed by ASEBA is diametrically opposite to that addressed by Nettle [16]. In Nettle, the FRP logic was maintained in the centralized controller provided by the OpenFlow switch. In contrast, the authors of the ASEBA work argue that a centralized controller for robots adds substantial delay and presents a scalability issue. Therefore, there was a need to handle these events as close as possible to the physical devices themselves. Consequently, reactive programming was used at the level of sensors and actuators, which are as close to the physical objects as possible.

Our work on RxDDS.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 (*i.e.*, DataReader) that receives the topic data. Thus, if such a subscriber was placed at a robot actuator, then RxDDS.NET can provide similar capabilities as ASEBA. On the other hand, if our subscriber was an OpenFlow switch, then we can provide similar capabilities as Nettle.

Prior work on Eventlets [18] comes close to our work on RxDDS.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. Application lifecycle can be managed in much the same way as in any component-based abstraction. The key difference between Eventlets and RxDDS.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. Thus, we believe that in future we may be able to leverage the desired properties of component abstractions in our work.

An ongoing project called Escalier [19] has very similar goals as our work. The key difference is the language binding: Escalier provides a Scala language binding for DDS while we are leveraging the Rx extensions provided in .NET framework and hence our work can potentially use all the languages supported by the .NET Rx platform. The future goals of the Escalier project are to provide a complete FRP framework, however, we have not yet found sufficient published literature on the subject nor are we able to determine from the github site whether this project is actively maintained or not.

3 Experiences Developing an Imperative Solution

To test our hypothesis regarding the limitations of imperative programming-based DDS solutions to support multicore scalability, this section describes a DDS-based imperative programming solution that uses C++11 to solve the DEBS 2013 grand challenge problem. To that end, we first briefly describe the DEBS 2013 Grand Challenge Problem. Subsequently we describe the imperative solution and highlight the challenges we faced.

3.1 DEBS 2013 Grand Challenge Problem

The ACM International Conference on Distributed Event-based Systems (DEBS) 2013 Grand Challenge problem comprises real-life data and queries in event-based systems. The goal of the DEBS 2013 Grand Challenge is to implement an event-based system for real-time, complex event processing of high velocity sensor data collected from a soccer game [13]. The real-time analytics for the DEBS Grand Challenge comprises continuous statistics gained by processing raw sensor data such as running analysis, ball possession, heat map, and shots on the goal. Contemporary approaches to gathering data for soccer matches utilizes a complex system of multiple cameras to record every part of a soccer field and computers to process the feed from the cameras. The DEBS Grand Challenge adopts a different approach by using inertial sensors to collect all data.

The sensor data is collected by a real-time localization system from an actual soccer game. The data is recorded in a file and provided to the DEBS Grand Challenge teams. The sensors are located near each player's shoe, in the ball, and attached to each goal keeper's hands. The sensors attached to 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 for which innovative solutions are sought.

The goal of query 1 is to calculate the running statistics of 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 over the observed interval, 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.

The aim of query 2 is to calculate the ball possession time for each player and for each team. We calculate the total time of the ball possession and the number of hits for each players. For team ball possession, the total time of the ball possession and percentage of ball possession for the specified team are generated. Like query 1, the team ball possession statistics must be provided for four different time windows. Finally, the statistics for these queries need to be updated at 50Hz.

Query 3 provides heat map statistics capturing how long each player stays in various regions of the field. The soccer field is divided into defined grids with x rows and y columns (8x13, 16x25, 32x50, 64x100) and results should be generated for each grid size. Moreover, distinct calculations need to be provided for different time windows like query 1 and query 2. As a result, query 3 must output 16 result streams, a combination of 4 different grid sizes and 4 time windows, in total. Each stream includes the percentage of time that a player spends in each user-defined cell boundary.

Query 4 provides statistics for shots on goal, which is defined as a player attempting to score a goal. It needs to keep track of shots scored or saved by the goalkeeper or other players. The result stream of the query includes the player ID and the ball sensor data.

3.2 An Imperative Solution in C++11

Our implementation of the four queries uses a DDS-based solution in C++11. The key architectural pieces of our solution, illustrated in Figure 1, comprise processing blocks and a high-level data-flow. Each processing block consists of one or more DDS DataWriters and DataReaders to publish or subscribe to data, respectively, where the data-flow in the network is logically partitioned by topics. Using DDS, application developers do not need to consider the network connections of each data-flow between DataWriters and DataReaders because it is handled by the underlying middleware. The architectural setup is geared towards solving all four queries of the DEBS 2013 challenge problem.

We defined data structures in IDL for each data-flow as well as its point-to-point segments. Key attributes, such as *sensor_id* and *player_id*, define unique DDS instances to group the updates for the same physical entity (*e.g.,* a sensor). Through the grouping mechanisms enabled by defining a key in DDS, a processor can easily distinguish between multiple players. The keys also help in partitioning the data-flow(s) across multiple processors to exploit inherent parallelism in the per-player data streams. Each DataReader in the processing blocks has a (in-memory) cache managed by the middleware to store and access the samples when needed. For instance, we utilized the cache data to implement the historical statistics required in some queries, such

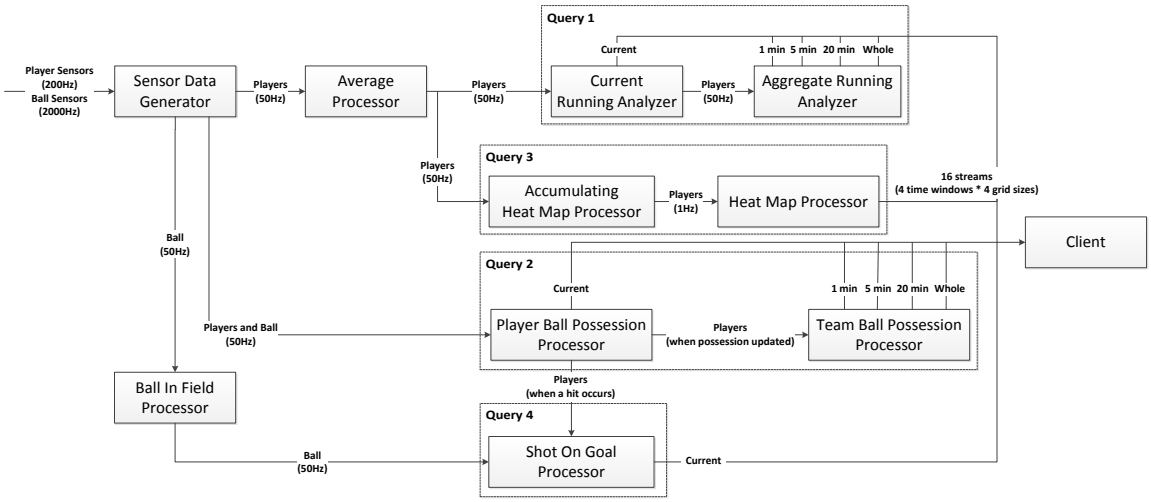

Fig. 1. High Level Data Flow Architecture

as aggregated data for the specified number of minutes. Cached data was also used for calculating distances between the ball and players to find the closest player to the ball for query 2 at a certain time.

The *SensorDataGenerator* processing block publishes sensor data in a real-time manner. The published data is sent to multiple processing blocks to be processed in parallel. In the case of query 1, a processor named *AverageProcessor* merges the data of two sensors attached to a player by averaging the sensor values. In the processing block, the ball data is filtered out because only the player's information is needed for query 1. We utilized `Time-based Filter QoS` to efficiently manage constrained resources and satisfy the required update interval of 50Hz. The next processing block for query 1 is *CurrentRunningAnalyzer* in which a player's running state, speed and distance are calculated, and the results are published to clients and the *AggregateRunningAnalyzer*. Finally, in the *AggregateRunningAnalyzer*, the accumulated values of distance and speed categorized by a player's running states are streamed as a result, which is divided into multiple streams for the specified time windows.

When a new sample arrives at the DataReader of a processor, a callback function, `on_data_available()` provided by the DataReader's listener interface is invoked. Since the processor of the query computes the required statistics when a new sample arrives, we implement our business logic in the callback function.

The statistics needed for query 2 are ball possession of players and teams. To acquire these statistics, we created two processing blocks: *PlayerBallPossessionProcessor* and *TeamBallPossessionProcessor*. In the *PlayerBallPossessionProcessor*, the sensor data of players and balls are subscribed from *SensorDataGenerator* to find ball possession by computing the distance between players and the ball, and the ball's acceleration.

Query 3 is perhaps the most straightforward query in its logic. However, the amount of data required to be kept in the system to accurately calculate the results grows quickly and hence needs to be handled properly. To do this, we have a receiving node, called *AccumulatingHeatMapProcesser* (Accumulator), that processes player positions from the *AverageProcessor* at 50Hz. This Accumulator reads in the location data for each player and builds a logical data structure of 6,401 integers: the total number of samples that were recorded for that second, and then values for each of the 64x100 row/column cell value.

Query 4's goal is to detect events of a player shooting in the direction of the goal. Shots on goal occur when a player hits the ball (*i.e.*, his foot is close to the ball and the ball's acceleration is greater than 55 m/s^2) and the projection of the ball's movement would lead it within one of the two goal areas. Since detection of hit events is already carried out to compute query 2, *ShotsOnGoalProcessor* subscribes to a data-stream published by by *PlayerBallPossessionProcessor*. Similarly, the filtered data-stream reporting data about the ball currently used in the game is used instead of raw sensor data.

3.3 Challenges Faced

An advantage DDS provided us in building our imperative solution was the way in which we could focus on developing the algorithms to address each query while relying on the network middleware to provide us out-of-box solutions needed for reliable and real-time data distribution to scale out by employing multiple machines. However, we faced some challenges while solving this problem.

- **Lack of a capability to automatically manage states of events** – When an event processing block deals with input events to generate output events, some input events might be relevant and dependent on each other. Hence, some input events should wait until relevant events arrive at the processing block to produce output events. In the imperative approach, we manually implemented the logic to synchronize relevant events (*e.g.*, events for sensor data attached to a player). Additionally, we needed to manually manage a previous state of an event for each player to calculate distance moved between previous and current timestamps.
- **Lack of a scalable concurrency model to scale up event processing employing multicore** – Since DDS usually utilizes a dedicated single thread for a DataReader to receive an input event, there is a need to manually create threads or a thread pool to exploit multicores when processing streams concurrently. In the imperative solution, to utilize multicores, multiple processes are spawned and data streams are partitioned to multiple streams by keys, where each stream is injected and processed by each process. This approach is acceptable when scaling out to multiple machines over a network, but can be inefficient if multicore machines need to be fully utilized for an event processing block.
- **Lack of a reusable library to compute events based on different sliding time-windows** – A system for complex event processing typically requires handling events based on different sliding time-windows (*i.e.*, last 1 hour or 1 week). If a reusable library that takes care of this pattern is provided, it helps to reduce development time to build such a system. Currently, we had to reinvent the solution every time it was needed. In the imperative approach, we made use of data cache allocated by the underlying middleware to compute average values of events changing by sliding time window. This data cache maintains a fixed number of history of events (controlled by `Resource Limits QoS`) and replaces an old event with a new event when it arrives. This replacement event was used by a DDS application to calculate average values by a changing range of a time window. This approach assumes that events arrive at the expected frequency (controlled by `Time-based Filter QoS`), and depending on the frequency, the size of data cache should be determined manually.
- **Lack of flexibility in component boundaries** – In complex event systems, a mechanism to publish or subscribe events is required to partition, merge, and reuse events by event processing blocks. As DDS supports this mechanism through DataWriters and DataReaders, we naturally used these DDS entities for incoming and outgoing events between event processing blocks. However, this approach incurs overhead by sending (serializing) and receiving (de-serializing) DDS samples even if event processing blocks are deployed in the same machine to process intermediate results.

We believe that the challenges we have outlined above are likely to be faced in similar situations that require real-time stream processing. We believe this is where we may derive substantial benefit from using FRP in conjunction with technologies, such as DDS pub/sub. Section 4 describes how functional reactive programming can help address these challenges. Specifically, Section 4.3 provides a qualitative comparison between the FRP approach and imperative approach alluding to these challenges. In the interest of brevity we focus on query 1 and 2 in this paper.

4 Experiences Developing a FRP-based Solution

In this section, we will first describe our FRP solution that integrates Rx with DDS and provides it as a reusable library called RxDDS.NET. Next we describe how we used the RxDDS.NET FRP library to implement DEBS 2013 Grand Challenge Problem and describe our experience through qualitative and quantitative comparisons with our imperative solution.

4.1 Design of the RxDDS.NET Library

This section describes the design of our FRP solution that integrates .NET Reactive Extensions (Rx) framework with DDS. This solution is made available as a reusable library called RxDDS.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 processing stages. 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 `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 data-space. DataReaders receive notifications when an update is available on a topic. Therefore, a topic of type T maps naturally to Rx's `IObservable<T>.` 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 itself, a keyed topic can also be viewed as a stream of keyed streams and therefore maps naturally to Rx's `IObservable< IGroupedObservable<Key, T> >.`

Thus, when our RxDDS.NET library detects a new key, it reacts by producing a new `IGroupedObservable<Key, T>` with the same 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. Mapping DDS concepts to Rx concepts

DDS Concept	**Corresponding Rx Concept and the RxDDS.NET API**
Topic of type T	Create a new IObservable¡T¿ using DDSObservable.fromTopic¡T¿(..)
Topic of type T with key-type=Key	Create a new IObservable¡ IGroupedObservable¡Key,T¿ ¿ using DDSObservable.fromKeyedTopic¡Key, T¿(..)
Detect a new instance	Notify Observers about a new IGroupedObservable¡Key, T¿ with key==instance. Invoke Observer.OnNext()
Dispose an instance	Notify Observers through Observer.OnCompleted()
Read an instance update	Notify Observers about a new value of T using Observer.OnNext()
Hard error on a DataReader	Notify Observers through Observer.OnError()
Communication Statuses (e.g., deadline missed, sample lost etc.)	Separate Observables for each communication status. For example, IObservable¡DDS::SampleLostStatus¿. Notify Observers through Observer.OnNext(). Currently not implemented.
Discovery Events (i.e., built-in topics)	Separate keyed Observables for each built-in topic. For example, IObservable¡DDS::SubscriptionBuiltinTopicData¿. Notify Observers through Observer.OnNext(). Currently not implemented

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 RxDDS.NET API, but they are currently not implemented in our library and forms part of our ongoing improvements to the library.

4.2 DEBS 2013 Grand Challenge Solution using RxDDS.NET

Integration of Rx with DDS enables us to operate on "streams of data" that are made available to the subscriber by DDS. The transformation of a data stream from DDS into an Rx Observable provides a seamless continuation of the "data-in-motion" view. We are now able to compose and perform operations on streams of data as opposed to operating on each data sample individually as it is made available in the `on_data_available` callback. In DDS, when a new data sample arrives, DDS informs the subscriber by calling its `on_data_available` callback method.

This view of performing operations on a stream of data offers a very powerful abstraction where we can leverage existing stream operators to perform complex stream processing without the overhead of manually maintaining state. Rx allows us to manage concurrency in application logic in a declarative manner without explicitly creating threads, using monitor locks or wait handles. The resulting code is much cleaner and centered on "what" as opposed to "how".

We re-implemented the 2013 DEBS grand challenge queries by using the functional reactive style of programming provided by our RxDDS.NET library. In the following section we show snippets of our code as part of evaluating the qualities of the FRP solution.

4.3 Evaluating the FRP Solution

We now evaluate our FRP solution along the dimensions of challenges expounded in Section 3.3 and compare it qualitatively with our imperative solution. Our ongoing work is focusing on quantitative evaluations. Table 2 summarizes the key distinctions between the imperative and FRP solutions along each dimension of the challenges.

Table 2. Comparison of Imperative and Functional Reactive Solutions

	Imperative	**Functional Reactive**
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

Automatic State Management We can take advantage of built-in stream operators to delegate the required implementation and state-management to Rx. Since the common state information no longer needs to be maintained by the application logic, the code is succinct and can be easily parallelized.

For example, to calculate average sensor data for a player from the attached sensor readings, we had to cache sensor data for each *sensor_id* as it arrived 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. Now 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 reactive 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. The Marble diagram for CombineLatest is shown in Figure 2. Marble diagrams are a way to express and visualize how the operators in Rx work. For details on Marble diagrams, the reader is encouraged to reference `http://rxwiki.wikidot.com/marble-diagrams`.

Listing 1.1. CombineLatest operator example code

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

In listing 1.1, *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* 13,14,97,98; *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 1.1) of latest sensor data for each sensor attached to this player. *returnPlayerData* function is then used to obtain the average sensor values.

Note that `CombineLatest` will not produce any output if one of the streams does not produce any data; so if one of the sensor's data does not appear for a long time, this operator will not produce any output for that player's average sensor data stream. To avoid that, we pass a dummy value on each sensor stream with `Once` operator. `Once` operator was implemented such that it will send a dummy variable as the very first data sample on a stream (when the stream is subscribed to) in order to unblock `CombineLatest's` operation. `CombineLatest` will use the dummy variable as the latest data on this stream until actual stream data appears.

Listing 1.2. Once operator example code

```
foreach (var val in sensorList)
{
        sensorStreamList
        .Add(src
              .Where(ss => ss.sensor_id == val)
              .Once(new SensorData
                    {
                          // dummy fields
                    }));
}
```

The use of `Once` is illustrated in listing 1.2. Here for each *sensor_id* attached to a player stored in *sensorList*, we add a reference to the stream associated with this *sensor_id* to *sensorStreamList* (as used in listing 1.1). For each sensor stream the once operator emits a dummy value, so that averaged sensor data for this player can be obtained even if one of his sensor's data doesn't show up until later in the game.

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 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 3 shows the marble diagram of the `Scan` operator. In the imperative approach, we employed the middleware cache to maintain previous state.

While conventional stream processing applications can delegate state maintenance to stream operators, as the requirement becomes more involved there will be a need to maintain state information explicitly, *e.g.*, in query 2 where ball possession data for a player is published when ball possession state changes.

Since ball possession state can change on a new event information, *e.g.*, ball hit by the same player, ball hit by a different player, ball out of field or game ended; we cannot look at a single stream of player data in an independent and isolated manner. This case degenerates into the developer having to write the logic to react when new event information is produced, much like the imperative approach where we process data when a new sample arrives. In this case we have to manually maintain state information for each player, *e.g.*, timestamp of last hit, previous possession time and previous number of hits in order to publish updated player ball possession data on change in ball possession state.

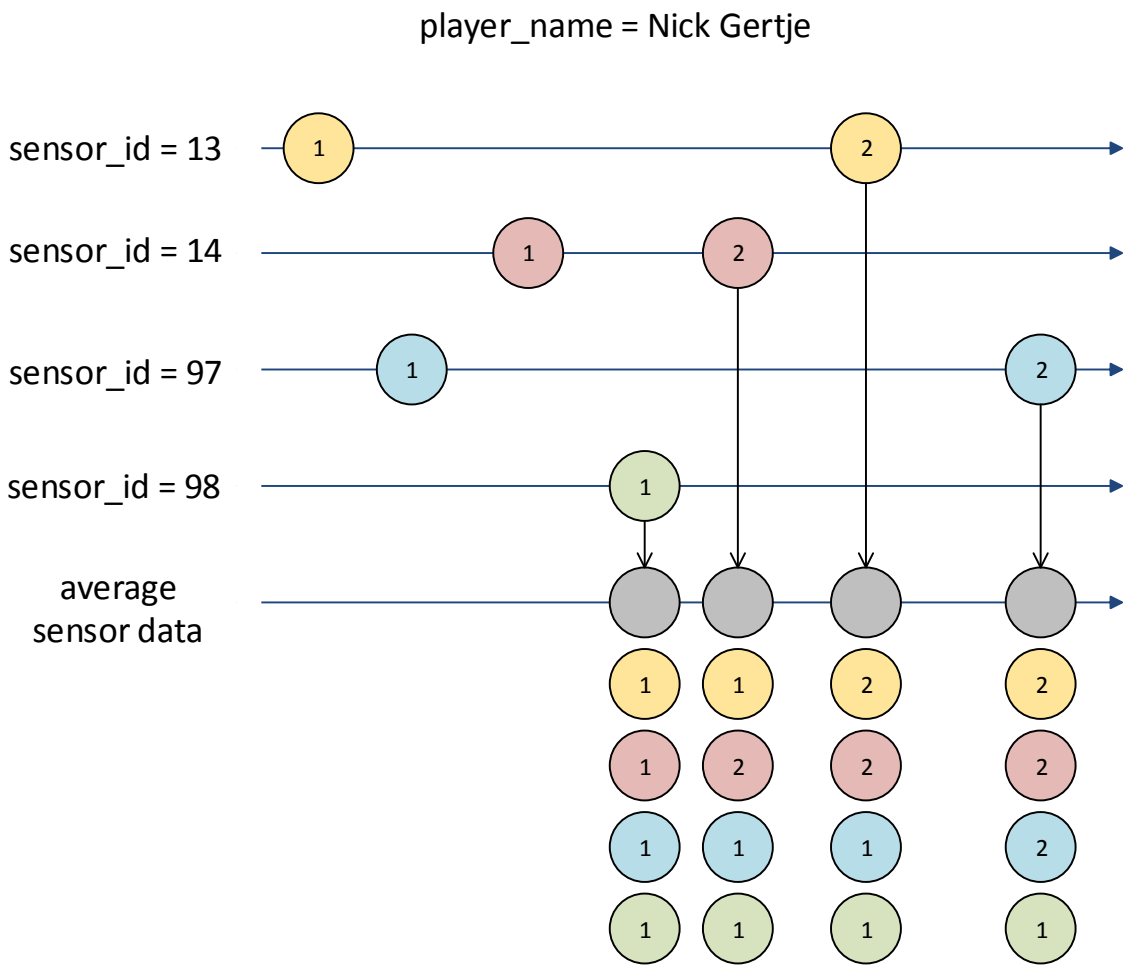

Fig. 2. Marble Diagram of CombineLatest Operator

Finally, in event processing we always have some or the other dependency between multiple events that we must address during computation. In the imperative approach we need to design and update supporting data-structures for capturing those dependencies – may it be waiting until a new data sample arrives on each stream before doing the required computation, using the latest data sample as it arrives, replaying previous values, etc. Maintaining concurrent access to these supporting data-structures becomes even more complicated if multiple threads can access the shared state. Rx captures these event dependencies in its built-in operators that we can readily use for our requirement. The resulting code is cleaner and is focused on implementation's functional aspect.

Concurrency model to scale up multi-core event processing Rx provides abstractions that make concurrency management declarative, thereby removing the need to make explicit calls to `Thread`, `Thread Pool` or `Tasks`. 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.
Rx concurrency "contract" is that all stages of a pipeline are replicated each time `Subscribe` is called. This effectively precludes management of shared state information. To prevent synchronization overhead

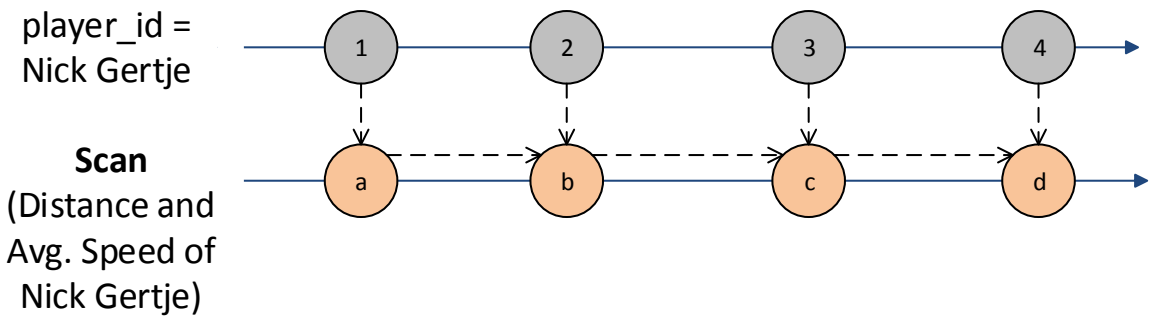

Fig. 3. Marble Diagram of Scan Operator

at each stage, `onNext`,`onCompleted` and `onError` are never called concurrently for a pipeline. However for different pipelines `onNext`,`onCompleted` and `onError` can be called concurrently. Some inbuilt Rx operators like `Zip` and `CombineLatest` that operate on multiple streams, are inherently concurrent and must uphold the "contract" for the downstream pipeline.

`RxDDS.Net` functions: `fromTopic` and `fromKeyedTopic` create an `Observable` and `GroupedObservable` respectively from a DDS topic. The functions, `fromTopic/fromKeyedTopic` are single threaded and use DDS "receive" thread. Hence `ObserveOn` must be called right after creating an `Observable/GroupedObservable` with `fromTopic/fromKeyedTopic`, to prevent holding-up the receive thread and to separate application level threads from middleware threads. When correlating two different topics, `SubscribeOn(ThreadPool)` should be called right after `fromTopic/fromKeyedTopic`. Unless `subscribeOn(ThreadPool)` is used, initialization of the subscription takes place in the receive thread and a delayed subscription, may cause the recieve thread to die.

In query 1, the current running statistics and aggregate running statistics are being 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.

As shown in listing 1.3, the stream of average player data (*playerDataStream*) can be de-multiplexed into multiple streams for each player with the `GroupBy` operator. Now the per-player stream computation can be off-loaded on a specified scheduler with `ObserveOn`. 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-pool.

Listing 1.3. Concurrent event processing with multi-threading

```
playerDataStream
        .GroupBy(data => data.player_name)
        .ObserveOn(Scheduler.Default)
```

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 2013 DEBS grand challenge 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 to this end.

DDS does not support a time-based cache which can cache samples observed over a time-window. The "window" abstraction which is most commonly needed by stream processing applications is provided by Rx which supports both discrete (*i.e.*, based on number of samples) and time-based (*i.e.*, wall-clock) windows. `Aggregate` and `Scan` functions perform stream accumulation wherein `Aggregate` produces a single result

upon stream completion and scan produces runtime accumulation results on applying the accumulator function to each new data sample over the specified time-window.Figure 4 depicts aggregation performed over a moving time window.

We implemented a time-window aggregator operator that will aggregate values observed within a specified time-interval based on time-stamped data. The time-window aggregator uses the time-stamp value in each data sample to keep track of elapsed time rather than relying on wall-clock. This gives a more accurate representation of elapsed time with respect to played-back sensor data.

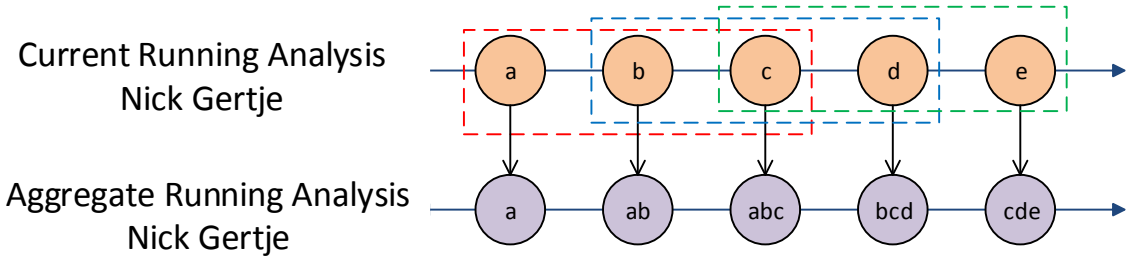

Fig. 4. Marble Diagram of Time-window Aggregator

Points of distribution (component boundaries) The integration of Rx with DDS allows the continuation of the concept of "stream of data." This concept provides incredible flexibility to the developer in demarcating their component boundaries or points of data distribution.

The imperative solution does not possess a higher-level structure beyond what DDS offers. Hence more often than not, developers tend to publish intermediate results over a DDS topic to be consumed by another subscriber. If scale-out or placement of these components on different machines is required, then this design is desirable, otherwise publishing intermediate results over a DDS topic isolates the components and imposes a "hard" component boundary. The resulting structure is very rigid and hard to co-locate. 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 RxDDS, 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 Factory pattern. 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 as shown in Table 3. Alternatively, for local processing, a `Subject<T>` could be used in the place of DDS DataWriter. The choice of a Subject vs. a DataWriter is configurable at deployment-time.

5 Conclusions

Reactive programming is increasingly becoming important in the context of real-time stream processing for big data analytics. Reactive programming supports four key traits: event-driven, scalable, resilient and responsive. While reactive programming is able to support these properties, 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,

Table 3. Flexible Remote(DataWriter/DataReader) or Local(Subject/Observer) Processing

	Publish	**Subscribe (on a remote machine)**
Remote (DDS)	intermediateStream .Subscribe(aDataWriter);	DDSObservable .fromTopic(...) *.SomeQueryOperator(...)*;
	Subject	**Observer (same process)**
Local (`Subject<T>`)	intermediateStream .Subscribe(aSubject);	aSubject *.SomeQueryOperator(...)*;

such as the OMG DDS, which is particularly suited towards real-time 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).

To that end, this paper presented our experiences from an industry-academia collaboration integrating the Rx .NET reactive programming framework with OMG DDS, which resulted in the RxDDS.NET library. To understand the advantages gained by this effort, we have used the DEBS 2013 grand challenge problem to compare a solution that uses RxDDS with a plain, imperative solution we developed using DDS and C++11, and made qualitative comparisons between these two efforts.

The following lessons were learned from our team effort and alludes to future work we plan to pursue in this space.

- Integration of Rx with DDS allows seamless continuation of the notion of "streams of data" and provides a holistic end-to-end architecture for both – data distribution by DDS and data processing by Rx.
- Rx offers first-class support for streams, which can be passed to/returned from functions and operations can be composed on them to form pipe-lines of data processing. Initially thinking in terms of operations on "streams" of data may not be intuitive, but soon the thought-process adapts to that view.
- FRP approach abstracts away state and control flow information from programmers such that the code is focused more on the functional aspect. The resulting code is very concise, readable and easily maintainable. With FRP approach, query 1 was coded in 364 lines of codes (loc) and query 2 in 349 loc, which is 27% and 57% less respectively from the imperative approach. In the imperative approach, query 1 took 500 loc and query 2 took 825 loc.
- Rx enables parameterizing concurrency in a declarative manner and avoids application level shared mutable state which makes multi-core scalability much easier.
- Our future work includes building RxDDS.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 and to construct better support for concurrency with use of DDS waitsets. We are also performing quantitative comparison of our imperative solution with FRP solution wrt to throughput and latency.
- We believe Rx along with DDS provides a powerful infrastructure for resilient and responsive reactive systems which can be easily scaled-up or scaled-out.

References

1. Halbwachs, N.: Synchronous Programming of Reactive Systems. Number 215. Springer (1992)
2. : The Reactive Manifesto. http://www.reactivemanifesto.org (2013)
3. OMG: The Data Distribution Service specification, v1.2. http://www.omg.org/spec/DDS/1.2 (2007)
4. Agha, G.: A Model of Concurrent Computation in Distributed Systems. MIT Press (1986)
5. Synodinos, D.: Reactive Programming as an Emerging Trend. http://www.infoq.com/news/2013/08/reactive-programming-emerging (2013)
6. Pembeci, I., Nilsson, H., Hager, G.: Functional Reactive Robotics: An exercise in Principled Integration of Domain-specific Languages. In: Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming, ACM (2002) 168–179
7. Wan, Z., Taha, W., Hudak, P.: Event-driven FRP. In: Practical Aspects of Declarative Languages. Springer (2002) 155–172
8. Elliott, C., Hudak, P.: Functional Reactive Animation. In: ACM SIGPLAN Notices. Volume 32., ACM (1997) 263–273

9. : Reactive Programming at Netflix. http://techblog.netflix.com/2013/01/reactive-programming-at-netflix.html (2013)
10. Meyerovich, L.A., Guha, A., Baskin, J., Cooper, G.H., Greenberg, M., Bromfield, A., Krishnamurthi, S.: Flapjax: A Programming Language for Ajax Applications. In: ACM SIGPLAN Notices. Volume 44., ACM (2009) 1–20
11. Cooper, G.H., Krishnamurthi, S.: Embedding Dynamic Dataflow in a Call-by-value Language. In: Programming Languages and Systems. Springer (2006) 294–308
12. : The Reactive Extensions (Rx). http://msdn.microsoft.com/en-us/data/gg577609.aspx
13. Jerzak, Z., Ziekow, H.: The ACM DEBS 2013 Grand Challenge. `http://www.orgs.ttu.edu/debs2013/index.php?goto=cfchallengedetails` (2013)
14. OMG: Extensible and Dynamic Topic Types For DDS (DDS-XTypes). http://www.omg.org/spec/DDS-XTypes (2012)
15. Salvaneschi, G., Drechsler, J., Mezini, M.: Towards Distributed Reactive Programming. In Nicola, R., Julien, C., eds.: Coordination Models and Languages. Volume 7890 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2013) 226–235
16. Voellmy, A., Hudak, P.: Nettle: Taking the Sting Out of Programming Network Routers. In Rocha, R., Launchbury, J., eds.: Practical Aspects of Declarative Languages. Volume 6539 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2011) 235–249
17. Magnenat, S., Retornaz, P., Bonani, M., Longchamp, V., Mondada, F.: ASEBA: A Modular Architecture for Event-Based Control of Complex Robots. Mechatronics, IEEE/ASME Transactions on **16**(2) (April 2011) 321–329
18. Appel, S., Frischbier, S., Freudenreich, T., Buchmann, A.: Eventlets: Components for the Integration of Event Streams with SOA. In: Service-Oriented Computing and Applications (SOCA), 2012 5th IEEE International Conference on. (Dec 2012) 1–9
19. Corsaro, A.: Escalier: The Scala API for DDS. https://github.com/kydos/escalier