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, 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 Functional Reactive Programming (FRP) 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.

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

I. INTRODUCTION

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 data produced by external sources. Common examples of such systems are intelligent transportation systems (ITS) and industrial condition-based maintenance (CBM) systems wherein a variety of sensors and other special equipment continuously generate data of the sensed environment. The acquired data is often complex in structure, large in volume, rich in variety, and often arrives at different velocities. The data is then analyzed in real-time to infer conditions of the assets and the prognostics are shared with the stake-holders depending on their urgencies.

Such systems are commonly known as Cyber-Physical Systems (CPS) or Industrial IoT (IIoT) systems. 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 sent downstream in different formats and at different levels of urgencies and accuracy to a heterogeneous set of subscribers. In essence, the emerging IoT systems can be understood as a distributed dataflow. The key challenge therefore lies in developing a dataflow-oriented programming model and a middleware technology that can address both distribution and processing requirements adequately.

Dataflows \cite{1} are closely related to asynchronous and event-driven reactive systems \cite{2}, which are often found in domains, such as robotics, industrial control, smart grid, animation, and video games. Reactive systems are long running systems that must respond to external stimuli at speeds determined by its environment \cite{3}. The Reactive Manifesto \cite{2} describes four essential traits of reactive systems: event-driven (i.e., push messages to consumers), elasticity (i.e., accommodate upwards and downwards load patterns while maximizing resource usage), resilience (i.e., isolate faults), and responsiveness (i.e., high degree of predictability).

In fact, event-driven design is a prerequisite for the other three traits (elasticity, resilience and responsiveness) because it enables loose coupling and asynchrony, which in turn helps to isolate faults when they occur. Asynchronous event-based architectures unify scaling up (e.g., via multiple cores) and scaling out (e.g., via distributed compute nodes) and defer the choice of scalability mechanism at deployment-time without hiding the network from the programming model. This approach shows promise as it can effectively exploit the distributed and inherently concurrent nature of the environment and the problem at hand.

The distribution aspects of the dataflow-oriented reactive systems are handled sufficiently by data-centric publish/subscribe (pub/sub) technologies \cite{4}, such as Object Management Group (OMG)’s Data Distribution Service (DDS) \cite{5}. The data processing aspects which are local to the individual stages of a dataflow, however, are often not implemented as a dataflow due to lack of sufficient generality in the application programming interface (API) of the pub/sub
middleware. A desirable programming model would provide an exhaustive set of reusable coordination primitives for reception, demultiplexing, multiplexing, merging, splitting, join, and more. 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. 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 Functional Reactive Programming (FRP) [6] and blended it with data-centric pub/sub. FRP is a declarative approach for system development wherein program specification amounts to “what” (i.e., declaration of intent) as opposed to “how” (looping, explicit state management, etc.). Declarative programs written using the FRP style often use the dataflow abstraction because 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. Therefore, there is a compelling case to systematically blend reactive systems with data-centric pub/sub mechanisms to solve key challenges in realizing emerging IoT applications.

Since FRP and pub/sub technologies evolved independently from each other, this research has been carried out in the context of two concrete technologies to demonstrate and evaluate the research ideas. The FRP instance we used is the Microsoft .NET Reactive Extensions (Rx) [7] while the data-centric pub/sub instance is the OMG’s DDS. Specifically, we use the DDS implementation by Real-Time Innovations (RTI).

This paper makes the following contributions.

1) We show the strong correspondence between the distributed dataflow model of DDS and the local 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 composable data processing API in DDS and the lack of interprocess communication and back-pressure support in Rx;

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 actual code. Furthermore, state management for sliding time-window, event synchronization, concurrency management can be delegated to the run-time thanks to 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 [8]. 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 significantly (up to 75%) improves performance. Our empirical evaluations indicate the need for “instance-level parallelism” within pub/sub middleware to further improve performance on multicore processors and use Rx more effectively.

4) Finally, we compare our functional implementation with our imperative implementation of the same queries in C++11 [8]. 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 II compares our proposed solution with prior efforts; Section III describes our FRP solution that integrates Rx and DDS; Section IV reports on both our qualitative and quantitative experience building a FRP-based solution to solve a specific case study problem; and finally, Section V provides concluding remarks and lessons learned.

II. RELATED WORK

A research roadmap towards applying reactive programming in distributed event-based systems has been presented in [9]. 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 work, we are integrating FRP 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 [10]. Nettle uses 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 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 [11]. 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 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 [12] 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 [13] in the context of Lambda Architecture (LA) [14] 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 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 [15] 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 related publications nor are we able to determine from the github site whether this project is actively maintained.

III. 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 imperative solutions using DDS for event-driven systems, which motivates the need for Rx4DDS.NET.

A. Overview of OMG DDS Data-Centric Pub/Sub Middleware

The 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.

B. Challenges Manifested by Imperative Approaches

Our prior experience [8] developing imperative solutions for an event-driven system case study used a DDS-only solution using the C++11 language. This experience highlighted a number of challenges with imperative solutions as described below:

- **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 dependent on each other (e.g., calculating average values of input stream events). Hence, some input events should wait until relevant events arrive at the processing block to produce output events. In our imperative approach, we had manually implemented the logic to synchronize relevant events which included logic we had to code to also maintain history of events.

- **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 was a need to manually create threads or a thread pool to exploit multicores when processing incoming streams concurrently. In the imperative solution, to utilize multicores, multiple processes had to be spawned and data streams had to be manually partitioned into multiple streams by keys, where each stream was injected and processed by each process. Such an 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 (e.g., last one hour or one week). If a reusable library that takes care of this pattern is provided, it helps to reduce development time to build such a system. In our imperative approach, we had to reinvent the solution every time it was needed. We also had to make use of a data cache (allocated memory by the underlying

---

2We have used the same case study to evaluate the FRP approach.
Observables. For example, an `IObservable<T>`
the points of synergy between the two.

e.g. This solution is made available as a reusable library
integrates .NET Reactive Extensions (Rx) framework with
C. Rx4DDS.NET: Integrating Rx and DDS

We believe that substantial benefits can be derived from
using FRP in conjunction with technologies, such as DDS
pub/sub to overcome these challenges. Section [IV] validates
these claims.

C. Rx4DDS.NET: Integrating Rx and DDS

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 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 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 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 data-
space. 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]

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 an instance. If a

[1] 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 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 [I] 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 similarity in the dataflow models, Rx and DDS are
quite interchangeable. Table [II] forms the basis of our argument
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) the `IObservable` interface that
represents the lifecycle of a data stream. These notions can
be directly mapped 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. Converse is also true,
however, only a subset of QoS attributes can be mapped
to Rx operators as of this writing. For example, DDS’s time-based filters can be mapped to Rx’s Sample operator; History QoS maps to the Replay operator.

IV. EVALUATING THE FRP-BASED RX4DDS.NET SOLUTION

This section reports on our experience evaluating the Rx4DDS.NET solution. For the evaluations we have used a case study and report on both the qualitative and quantitative findings.

A. 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 [16]. 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 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. 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 defined 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 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).

B. Qualitative Evaluation of the Rx4DDS.NET Solution

We now evaluate our FRP Rx4DDS.NET solution along the dimensions of challenges expounded in Section III-B and compare it qualitatively with our imperative solution [8] for the case study.

1) Automatic State Management: Recall that the imperative approach requires additional logic to maintain state and dependencies. For example, in the case study to calculate average sensor data for a player from the 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 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 for CombineLatest is shown in Figure 2.

Listing 1. CombineLatest Operator Example Code

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

Figure 2. Marble Diagram of CombineLatest Operator

2) 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 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.

Using Rx for 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. Listing 2 shows how the stream of average player data (playerDataStream in Listing 2) can be demultiplexed into multiple streams for each player with the GroupBy operator. Now the per-player stream computation can be offloaded on a specified scheduler with ObserveOn.

Listing 2. Concurrent Event Processing with Multi-threading

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

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. We implemented a time-window Rx aggregator operator that will aggregates 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. Figure 3 depicts aggregation performed over a moving time window.

Figure 3. Marble Diagram of Time-window Aggregator

C. 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 do not posses 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 distribution mechanism isolates the components and imposes a “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. 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.

D. Program Structure

The composability of operators in Rx, allows us to write programs that preserve the conceptual high-level view of application logic and data-flow. For example in Query 1, it computes the AggregateRunningData for each player for 1 minute, 5 minutes, 20 minutes and full game duration, and the query code is as shown in [5]

Listing 3. Program Structure of Query 1

```
player_streams.Subscribe(player_stream =>
{
    var curr_running =
        .Publish();
    curr_running.AggregateRunningTimeSpan(1);
    curr_running.AggregateRunningTimeSpan(5);
    curr_running.AggregateRunningTimeSpan(20);
    curr_running.AggregateRunningFullGame();
    curr_running.Connect();
}
```

In Listing [5] player_streams is a stream of streams (e.g. IObservable<IObservable<PlayerData>> comprises of 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 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.

Table II summarizes the key distinctions between the imperative and FRP solutions along each dimension of the challenges.

### Table II

<table>
<thead>
<tr>
<th>Component Boundaries</th>
<th>Imperative</th>
<th>Functional Reactive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Manual state</td>
<td>Declarative state-managed</td>
</tr>
<tr>
<td></td>
<td>management</td>
<td>state management</td>
</tr>
<tr>
<td>Concurrency Management</td>
<td>Explicit</td>
<td>Declarative</td>
</tr>
<tr>
<td></td>
<td>management</td>
<td>management</td>
</tr>
<tr>
<td>Sliding Time-window</td>
<td>Manual implementation of time window abstraction</td>
<td>Built-in support for both discrete and time-based windows</td>
</tr>
<tr>
<td>Computation</td>
<td>Inflexible and hard component boundaries</td>
<td>Flexible and more agile component boundaries</td>
</tr>
</tbody>
</table>

E. Quantitative Evaluation of Rx4DDS.NET FRP Solution

We measured input rate, i.e., the rate at which input data is received by a query and output rate or throughput, i.e., the rate at which output samples are produced after processing. The measurements are done in two different configurations: single-process and two-processes. In the single-process configuration, the raw sensor stream was generated by reading out the raw data file asynchronously in the same process in which the queries were run. On the other hand, in two-processes configuration, the raw sensor stream was published by a DDS publisher in a separate process and the query process computes on the subscribed sensor data over DDS in another process. Interprocess communication happens over shared-memory.

We evaluated single threaded and multi-threaded implementations of Query_1, (Configured as Listing [3], Query_3, and Query_1_3 in which both Query_1 and Query_3 were run together. All the tests were performed on a host with Intel Core i5, 1.8 GHz dual-core processor (4 hyper-threads) and 4GB RAM.

**Query 1 Experimental Results**: Figure 4 shows the input and output data rates for single-threaded and multi-threaded implementation of Query_1 configured with ThreadPoolScheduler under both single-process and two-processes configurations. Query_1 produces aggregate running data for each player for four time windows and hence the output rate seen is nearly four times the input data rate. The single-process, multi-threaded configuration of Query_1 shows 14% improvement in input and output data...
rates compared to the single-threaded configuration. On the contrary, in the two-processes configuration, processing different player streams on a ThreadPoolScheduler results in 12% decrease in both input and output data rates.

Figure 5 shows the throughput for all the scheduling strategies namely, single-threaded, EventLoopScheduler, ThreadPoolScheduler and TaskPoolScheduler. In the single-process configuration, the EventLoopScheduler shows 0.3% lower throughput whereas the ThreadPoolScheduler and TaskPoolScheduler show 14% and 17% higher throughput, respectively. In the two-process configuration, the EventLoopScheduler and ThreadPoolScheduler show 27% and 11% lower throughput respectively, whereas the TaskPoolScheduler shows 24% higher throughput than the single-threaded configuration.

Query 1 Result Analysis: As expected, in multi-threaded configurations, the performance improves as the inherent parallelism in the query computation is exploited using parallel execution. We also note that in the two-processes case, the use of schedulers results in decreased performance compared to the single-threaded configuration. We believe the loss in performance is due to two reasons: (1) Query_1 is computationally trivial, hence the gain in performance by introducing parallelism is less as compared to its added overhead; (2) the overhead of interprocess communication increases the ratio of the serial component of the Query_1 which adversely affects the little gain in performance that was afforded by the use of schedulers in case of this simple query.

Since Rx provides abstractions which make concurrency management declarative (hence trivially configurable), evaluating different concurrency options for our application becomes incredibly easy. By injecting the right instance of the scheduler, we evaluated whether introducing parallelism provides increased performance gain which is worth the added overhead or just presents higher overhead without much performance gain for a given problem. In contrast, evaluating the imperative implementation with different concurrency alternatives would have been much more complex, requiring a fair amount of changes in the code, careful use of mutexes and would be more verbose due to explicit calls to threads, thread-pools and/or tasks.

Query 3 Experimental Results: Figure 6 presents the difference in the input and output data rates for single-threaded and ThreadPool based configurations of Query_3. The output data rate for Query_3 is substantially less than its input data rate since Query_3 only updates HeatMapData every second (according to sensor timestamps) as opposed to Query_1 which produces an update for each input sample for Query_3. Processing each player stream on a ThreadPoolScheduler results in 75% increase in input and output data rates over single-threaded Query_3 in the single-process configuration. In two-processes case, the ThreadPool implementation shows nearly 60% increase.

Figure 7 shows the throughput observed using different Rx schedulers to parallelize Query_3 in both configurations. In single process, EventLoopScheduler, ThreadPoolScheduler, and TaskPoolScheduler show 46%, 75%, and 69% increase in throughput, respectively. In the two-processes configuration, EventLoopScheduler, ThreadPoolScheduler, and TaskPoolScheduler results in 45%, 60%, and 47% increase in throughput, respectively.

For Query_1_3, where we run both Query_1 and Query_3 are run together in the same process, Figure 8 shows the input and output data rates for
single-threaded and ThreadPoolScheduler configurations. For the single-process configuration, ThreadPoolScheduler based solution shows a 37% increase in output and input data rates, whereas for two-process configuration, the use of ThreadPoolScheduler results in a 34% increase.

Finally, Figure 9 presents the throughput observed with different Rx schedulers to parallelize Query_1_3 in both configurations. EventLoopScheduler, ThreadPoolScheduler and TaskPoolScheduler give 35%, 37% and 41% higher throughput over single threaded implementation in single-process configuration. In the two-process configuration, we see 29%, 34%, and 52% higher throughput over single-threaded configuration with EventLoopScheduler, ThreadPoolScheduler and TaskPoolScheduler respectively.

**Query 3 Result Analysis:** The results clearly indicate that the query implementations were able to achieve higher throughput by simply configuring the right Rx scheduler. The functional programming style lends itself naturally to dataflow processing, which can be easily parallelized if the program exhibits such a structure (also known as embarrassingly parallelizable structure). Rx allows us to capture the dataflow-oriented shape of the application naturally and further helps in improving performance using simple configuration of schedulers.

**V. CONCLUSIONS**

This paper describes a concrete realization of blending the Rx .NET reactive programming framework with OMG DDS, which resulted in the Rx4DDS .NET library to support both scale-out and scale-up. Our solution was evaluated and compared against an imperative solution we developed using DDS and C++11 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 seamless end-to-end dataflow architecture that is closely reflected in code.

- Our quantitative results indicate that Rx parameterizes concurrency and avoids application-level shared mutable state that makes multi-core scalability much easier. We showed increase (up to 75%) in performance of Query-1 and Query-3 by simply configuring the schedulers in Rx. Additionally, our experiments indicate the need for instance-level parallelism in DDS implementations for sustaining high performance on multi-core processors.

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

ACKNOWLEDGMENTS

This work was supported in part by the NSF 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.

REFERENCES


