StreamCoCo: A DSL for Processing Data-Centric Streams for Industrial IoT Edge Applications

[Industry Experience Report]

Sumant Tambe
Real-Time Innovations, Inc.

ABSTRACT

We report our experience of developing and using a simple yet an effective flow-based programming language and its distributed execution engine for detecting behavioral anomalies in physical assets in industrial IoT systems. Our stream processing systems is built using the Reactive Extensions (Rx) library for composing asynchronous data streams and the OMG Data Distribution Service (DDS) for publish-subscribe communication over the network. Our little language is called Stream Concatenation and Coordination (StreamCoCo) due to its similarity to the UNIX shell pipes-and-filter syntax. The novelty lies in the simple declarative programming model baked into the language that upon detection of anomalies in a stream, takes snapshots of other streams which may be distributed. Further, dynamic parallel pipelines of stateful stream processing operators are trivial to implement using StreamCoCo. We leverage the core capabilities of the language for infrastructure health monitoring and data analytics at the edge to assist remote human operators in problem diagnosis.

Keywords

Stream processing, Dataflow, Publish-Subscribe, Reactive Extensions (Rx), Data Distribution Service (DDS)

1. Introduction

The Industrial Internet of Things (IIoT) [1] is exponentially expanding the reach of digital computing and IP networks to the "edges" of large, geographically distributed systems. Industrial IoT—separate from just IoT—promises to deliver 80% of the economic value of the IoT hype [2] by enabling smart transportation, smart power grids [3], smart hospitals, smart cities, smart manufacturing, software-defined machines, and cloud analytics. IIoT combines physical machinery, networked sensors and software using the Internet.

As these IIoT systems are part of critical infrastructure and often deployed in remote, harsh environments, they are heavily sensorized to enable telemetry, remote health and condition monitoring for timely maintenance. The sensors produce raw data that is extremely large in volume, velocity, and variety. Not all the raw data can be sent to the centralized, cloud-based big-data analytics primarily due to bandwidth limitations and the cost of transmission. Instead, a more hierarchical and distributed approach is preferred where multiple streams of raw data are aggregated, reduced, filtered, and analyzed locally to identify behavioral anomalies in the fielded assets and escalate issues needing human attention to the centralized monitoring stations as needed.

Often the data streams representing physical values, such as temperature, pressure, vibrations, power output, etc. tend to be correlated. Known correlations have been exploited [4] in developing Bayesian Belief Networks in complex systems for health assessment of components and sensors. However, automatic health monitoring has its limitations and human intervention is necessary in many cases. Instantaneous data values are insufficient for humans to perform remote diagnosis because the overall *context* in which the anomaly arises may be unavailable. I.e., upon detection of divergent behavior, simply alerting a human is not sufficient. What is really needed is an accurate description of the detected situation along with the *state* of the overall system in close *temporal proximity* of the anomaly. It is important to understand the overall system state right *before* and *after* the anomalous situation (which may be transient or persistent).

The state of the system before the anomalous condition often contains clues to diagnose the potential *causes* and the state of the system after the anomalous condition contains *symptoms*. The state, however, is not necessarily the "program state", which is often inaccessible in service-oriented, loosely coupled systems. The state we refer to here is simply the most recently observed data samples across a number of distributed streams (dataflows) before and after the anomalous condition. The clues often lie in data streams separate (identity and space) from the one that raised the alert. As a consequence, whenever an alert is raised, *snapshots* from a number of parallel data streams must be captured and sent to centralized location for forensic analysis.

The snapshots also assist in understanding *trends* in closely related data streams. For instance, temperatures reported by a group of temperature sensors attached to a boiler must be highly correlated. Each temperature sensor is independent and forms a data stream in its own right. In our notion of the snapshot, it is crucial to capture parallel *instances* of same logical data stream (e.g., temperatures). A set of consecutive data samples from a set of closely related instances help identify trends.

As the rules of what constitutes an alert are highly domain-specific, we needed a specification mechanism to codify the rules that could be used by the subject matter experts without any programming background. In our experience, a dataflow abstraction works well in such situations due to its inherently declarative nature and expressive power. However, analytics at the "edge" must be light-weight and must support platforms that are resource-constrained (memory, CPU, and networking). COTS Complex Event Processing (CEP) engines that offer declarative SQL-like event processing languages are often too heavyweight.

We needed a lightweight, easy-to-use, easy-to-learn distributed dataflow processing system that could be scripted remotely. Scripting allows remote subject matter experts to augment existing analytics and insert additional flow processors without affecting existing dataflows.

To meet the above requirements, we developed a simple flow-based programming language named Stream Concatenation and Coordination (StreamCoCo), which has the following novel capabilities/features.

- StreamCoCo provides plain-English syntax for operators and a simple pipes-and-filter style for concatenating processing operators, which is motivated by the I/O redirection support in UNIX shells. It builds on the Data Distribution Service (DDS) [6] standard for publish subscribe communication and Reactive Extensions (Rx) [5] for processing asynchronous data streams. Background in both them is not necessary to be productive in StreamCoCo, which we believe is a major win for this language. As a consequence, non-programmers can leverage many powerful capabilities of DDS and Rx with little to no learning curve.
- StreamCoCo intelligently marries the concept of DDS *instances* with Rx's `groupby` operator and thereby supports dynamic parallel pipelines of stateful stream processing operators. Stateful processing is crucial for instance-based data analysis and trending.
- The flow-based programming model of StreamCoCo enables both local and remote propagation of alerts by unifying the (inter-process) publish/subscribe model of DDS and the (intra-process) subject-observer model of Rx. As a consequence, in a StreamCoCo program, distribution is a strictly deployment time decision and does not affect program source code in any way.

In the following sections we discuss our experience of developing StreamCoCo and its application to a on-demand programmable, remote problem diagnosis application. Secondary goal is to highlight suitability of Rx for DDS data processing. Therefore, our findings are more generally applicable regardless of StreamCoCo.

2. Overview of DDS and Rx

In this section we briefly describe DDS and Rx.

2.1 Data Distribution Service (DDS)

The OMG Data Distribution Service (DDS) [6] is a data-centric publish/subscribe standard with support for a number of QoS properties. DDS allows applications to share data by publishing and subscribing typed data samples to a *topic*. The name of the topic must be agreed upon between application priori. Topics belong to a global data space (a domain) governed by types specified using the Interface Definition Language (IDL). The data type may be keyed on one or more fields. Each key identifies an instance (similar to a primary key in a database table) and DDS provides APIs in C/C++/Java/C# to control the lifecycle of instances. Instance lifecycle supports CRUD (create, read, update, delete) operations, which are conceptually similar to database operations. Complex delivery models can be associated with data-flows by simply configuring the topic QoS.

2.2 Reactive Extensions (Rx)

Rx [5] is a library for composing asynchronous data streams based on the principles of Functional Reactive Programming (FRP). FRP is a declarative approach for program design 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 use the *dataflow* abstraction because the state and control flow are hidden from the programmers. FRP offers high-level abstractions that avoid verbosity.

Rx represents asynchronous data streams 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 to DDS instances.

3. Syntax and Semantics of SteamCoCo

StreamCoCo is a domain-specific scripting language for stream processing and coordination. It embodies *Save Query; Run Data* paradigm, which is inverse of the

traditional database approach. The current implementation of the language is based on Node.js, it compiles just-in-time to native code on Windows and Linux platforms.

StreamCoCo has a declarative English-like syntax and as a consequence little/no programming experience necessary to use it. StreamCoCo uses JavaScript Object Notation (JSON) syntax. The smallest block of specification in StreamCoCo is called a *probe*, which encapsulates a name, a filter, and a set of human readable tags. Filters inspect streams and raise alerts. Listing 1 shows an example of a probe named `TempAvgProbe`.

```
{
 "name"     : "TempAvgProbe",
 "filter"   : "<see listing 2>",
 "trigger"  : true,
 "tags"     : [ "overheat", "heat_warning" ]
}
```

Listing 1: A Probe in StreamCoCo

The `filter` section includes the core analytical logic of a probe, which uses simple pipes-and-filter style syntax to chain multiple processing stages—*operators*. StreamCoCo allows composition of operators using pipes similar to the UNIX I/O redirection facility. It supports a library of operators including data sourcing, predicates, arithmetic, time windows, conditionals, I/O, filtering, data partitioning, throttling, join, staleness detection, etc.

Listing 2 shows an example of a filter in `TempAvgProbe` (Listing 1).

```
source Temperature                          |
hastype temperature_readings                |
has sensor_id                               |
has fahrenheit                              |
match { "host" : "climate_monitor" }        |
insert eval((fahrenheit-32)*5/9) as celsius
delete fahrenheit                           |
groupby sensor_id                           |
insert avg(celsius) over 30 sec as
extra.degree_avg                            |
snapshot timerange(-60 sec,+60 sec)         |
greater_than_equal extra.degree_avg 65      |
interval 120 sec
```

Listing 2: A filter for detecting out of range temperature sensors (instances)

We use Listing 2 as an example to describe the semantics of various operators and their relationship. Whenever a data sample propagates through all the operators in a filter, it is transformed in to an a*lert*. An alert contains the data that caused it, the name of the probe, and the tags. The alert is simply forwarded to a well-known "Alerts" DDS topic that all the DSP agents are subscribed to. As a consequence, any alert produced anywhere in the system is propagated to all DSP agent automatically. The `snapshot` operator describes later makes use of alerts.

3.1 I/O Operators

The `source` operator subscribes a *stream* identified by a name. It may be a previously defined probe or a DDS topic. The source operator hides the true source of the data. The `keyed_dds_source` operator is designed to work with DDS source only and is described in subsection 3.4 due to its built-in data partitioning behavior.

Data sourcing operators, such as `mergesources`, `combinesources`, and `zipsources` use more than one asynchronous data source (i.e., DDS topics or other probes) and join them in some fashion. These operators map to `merge`, `combineLatest`, and `zip` combinators in Rx. All the multi-source operators produce a composite structure from individual samples from the constituent streams.

We use JSON for construction of composite structures from the data samples received over individual streams. Thanks to JSON's dynamic, self-describing data representation.

Finally, `output` is an operator for publishing data samples to a named DDS topic (which may be subscribed elsewhere).

3.2 Predicate Operators

Operators, `hastype`, and `has` define predicates that ensure that the incoming data samples have the necessary structure. This step is often unnecessary when the true source of data is a DDS topic as the schema of DDS topics is always well-defined. It is useful when the true source of data is a local stream or dynamic.

The `match` operator is also a predicate and uses a partial JSON object for pattern matching. It filters data samples that do not match the partial object. The `greater_than_equal` is an operator in a family of logical operators that filters samples that do not satisfy the condition. Finally, the `contains` operators does a substring search in a given property.

All the predicate operators are mapped to the `Where` combinator in Rx.

3.3 Data Manipulation Operators

The `insert` operator is a data projection operator, which allows computation of new data values and fields from existing one. The new data values produce new key-value pairs in the current data sample being processed. The `insert` operator is quite flexible and supports counter and time-based windows, arithmetic functions such as `avg`, `min`, `max`, `count` (akin to SQL), and general-purpose tree query language (e.g., `JSONPath`) for traversing and extracting deeply nested elements from complex hierarchically structured data samples. The `insert` operator maps to `Map` or `Select` combinator in Rx.

StreamCoCo has a `delete` operator, which deletes a key-value pair.

3.4 Data Partitioning Operator

The `groupby` operator in StreamCoCo partitions data according to a *key* (e.g., `sensor_id`) in the data sample. Keys are fundamental to data-centric communication supported by DDS. Keys give rise to instances in DDS, which are basically partitioned data streams. While DDS handles *distribution* of instances, StreamCoCo allows us to *process* instances by leveraging the Rx programming model.

The `groupby` operator not only partitions data but also lazily evaluates the subsequent stages for each new key. For every new key, it instantiates a fresh copy of all the subsequent stages allowing parallel stateful pipelines. As a result, a single filter description in StreamCoCo manages multiple DDS instances with ease. When the data stream representing the instances completes (i.e., when DDS instances are *disposed*), it reclaims the resources.

The `keyed_dds_source` is a data sourcing operator that uses the grouping semantics directly supported by DDS in a given *keyed* topic. As DDS supports CRUD operations on instances, `keyed_dds_source` reacts to them by lazily instantiating (or disposing) downstream operators for each new (disposed) instance. In that regard the behavior of `keyed_dds_source` is identical to the `groupby` operator that partitions stream independent of DDS.

Both DDS and Rx have a steep learning curve, and despite their suitability we do not expect operators (non-programmers) to learn such rigorous distributed programming methodology. StreamCoCo nearly eliminates the learning curve of both DDS and Rx and enables non-programmers to use intuitive chaining of operators for analyzing DDS instances. Implementing equivalent capability in vanilla DDS and Rx APIs requires proficiency at least one major programming language supporting lambdas (i.e., anonymous functions) and related programming patterns. StreamCoCo, however, enables non-programmers to use this complex feature with ease.

3.5 Rate Control Operators

The `interval` operator throttles the rate at which alerts are produced. In Listing 2, two successive alerts are separated by at least 2 minutes. This allows the It maps to the `Throttle` combinator in Rx.

3.6 Distributed Snapshot Operator

The snapshot operator is a novel operator (not built-in in Rx or DDS) and is key to the selection and propagation of temporally correlated data samples in distributed streams. Nominally, the snapshot operator subscribes to the predefined "alerts" stream and in absence of any alerts has no side effect on the behavior of the system.

Whenever an alert is produced, it causes all other probes (that have a `snapshot` stage) to *snapshot* the streams they are inspecting. A snapshot is a set of data samples in a stream that fall within a window of time, such as $-n$ to $+p$ minutes, from the time of reception of an alert. Such a set of data samples forms the *observable state* of the system, which perhaps contains clues to investigate the cause of the deviation in the behavior. The key idea behind the probes is to capture the observable state of the system *around* the time of the alert event. Obviously, the state could be distributed because the DSP agents are distributed. A snapshot, therefore, can be thought of as a thin *slice* through time and space that captures the system state in terms of the monitored streams. Figure 1 is a schematic of data, alert, and snapshot streams.

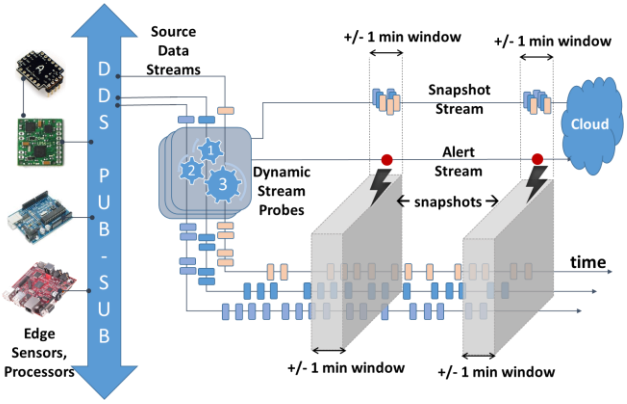

Figure 1: DSPs analyze data, produce alerts and transmit snapshots (best viewed in color)

The snapshot stage in Listing 2 extracts 2 minutes worth of data flowing through the filter when an alert is detected. Note that the time-range indicates 1 minute into future so data yet to be received is also "captured" by the snapshots. If/when a probe (including itself) produces an alert, the snapshot operators of all the probes forward the buffered data samples to a *snapshot* stream, which is the only non-alert data sent to human operators for detailed forensic analysis. The data samples in every snapshot are tagged with the same tags reported in an alert. This allows operators to classify data samples based on what caused it to be captured. A data sample may be tagged with multiple alerts coinciding a time window.

3.7 Operator for Joining Multiple Streams

StreamCoCo provides the `until_next` operator that joins data samples from two asynchronous data streams. For example, consider the following listing.

```
until_next join p from Pressure select
p.pascal as Pa, celsius
```

The keyword `join` is only to improve readability. The `until_next` operator has an implicit stream that the operator is part of. The second stream is specified as "`p from Pressure`" where "Pressure" is a DDS topic name or another probe. Subsequently, the `select` part of the operator selects the properties from either of the streams. Optionally, the name of the new projected property can be

specified using `as`. As a result, the example produces a stream of structures containing `Pa` and `celsius` properties, which are copies of `pascal` and `celsius` from the secondary and primary streams respectively.

Clearly, the `until_next` operator is motivated by the SQL JOIN statement. However, there are significant differences. A SQL JOIN would nominally join each row from one table with every row from the other table. It is a cartesian product. SQL allows optional WHERE in a JOIN which may prune the overall result set. For infinite data streams, such semantics of join would require unbounded amount of resources as both streams could be infinite.

The `until_next` operator suggests a notion of time, which executes join with the second stream only until the next data sample in the primary stream is received. For example, if the data rate of the "Pressure" topic is two times higher than the "Temperature" topic, the output of `until_next` is as follows.

Input Streams T=Temperature, P=Pressure	`until_next` output[1]

`T 0-1-2-3-4-5...`
`P ABCDEFGHIJK...`
	`0,A`
	`0,B`
	`1,C`
	`1,D`
	`2,E`
	`2,F ...`

As a result, `until_next` operator always uses a bounded amount of storage while executing a join. The behavior of `until_next` operator is quite suitable for edge applications in IIoT as nearly all streams are updated continuously and periodically, at fixed rates ranging from KHz frequencies to multi-second periods. Suitable delivery quality-of-service are best-effort reliability with low latencies and jitter. In our target systems, stale data is often unimportant.

4. The StreamCoCo Virtual Machine

The virtual machine that *executes* StreamCoCo programs—probes—is called a Dynamic Stream Probing (DSP) agent. DSP agents are dynamic because they can receive new probes dynamically and launch them without disrupting existing dataflows. Moreover, DSP agents can be easily distributed and new instances join existing agents as long as they all use a preconfigured DDS domain. Each DSP agent automatically subscribes to the "Alerts" DDS topic and makes it available to the probes. Probes either subscribe to any topic visible in the DDS domain or an existing probe that is running in a given DSP instance.

The existing DSP agent implementation is in JavaScript using the Node.js service-side JavaScript VM. DDS integration is supported using the Node.js connector [8].

Despite its reliance on the JSON syntax and internal JSON data representation, StreamCoCo is not inherently tied to JavaScript or Node.js in any way. These choices were a consequence of the available technology at the time we started with the project.

In the current implementation, StreamCoCo is just-in-time-compiled to native code because the JavaScript engine underlying Node.js—the Google V8 engine—just-in-time-compiles JavaScript to native code. DSP agents translate probe filters to JavaScript code before launching them. The process of translation amounts to chaining Rx combinators to produce a dataflow. Each probe translates to a single dataflow of chained Rx combinators. Some operators, such as `groupby` are dynamic and instantiate downstream operators lazily for every unique key. Rx combinators corresponding to StreamCoCo operators are configured by passing closures (i.e., instances of JavaScript anonymous functions). The translator produces the *right* closure instances at run-time. Inherent composibility of Rx combinators allows chains of arbitrary length with ease.

StreamCoCo can be implemented efficiently in other languages that implement the Rx library, which include C#, JavaScript, C++, Java, Scala, and more. Even though StreamCoCo probes are akin to scripting languages, they are *not* interpreted at run-time except the expression in `eval`.

The probes are descriptions of dataflows that are translated in to chains of Rx combinators at run-time. An Rx dataflow is just a data structure (like a tree) that is executable using a scheduler, such as a single-threaded scheduler that we use in our implementation. A StreamCoCo VM implemented in a statically safe compiled language would simply construct an equivalent dataflow by composing multiple Rx operators. Execution of it would be delegated to the language's native scheduler, if any. As StreamCoCo does not allow adding your own combinators, it is a *closed* language. I.e., each operator in StreamCoCo is akin to a *keyword*. The closed nature of the language is instrumental to its efficiency.

5. Related Work

StreamCoCo has similarities with a number of Complex Event Processing (CEP) and Continuous Query Language (CQL) products and research languages. We're not aware of any comparable stream processing language that is suitable for edge-level applications in IIoT where resources are constrained and server-based solutions are often not deployable. StreamCoCo VM is light-weight and can be embedded in traditional C/C++ programs. We believe our

[1] The `combineLatest` combinator in Rx is similar but has more non-deterministic behavior. I.e, `1,B` is a possible output. However, `until_next` joins a data sample from the primary stream with only *subsequent* data samples from the secondary stream—unlike `combineLatest`.

use of the `groupby` operator is novel as it easily supports parallel pipelines of stateful operators.

Our objective in StreamCoCo is to allow remote human operators to add probes on-demand without any programming background. SQL, despite its declarative nature, tend to get complex due to its tendency to nest subqueries and projections. In StreamCoCo, subqueries are just separate probes that do not generate alerts (only intermediate results). As a result, it is more gradually composable (like a pipeline) than SQL and alleviates learning curve among non-programmers.

DDS specification [10] describes syntax and semantics of content-based query and filters that are akin to SQL. Specifically, DDS allows *MultiTopic* expressions, which joining more than one topics with SQL-style projections and predicates. At the time of this writing we are unaware of any DDS implementation that supports MultiTopic. Our research in StreamCoCo is a firm step towards supporting MultiTopic for RTI Connext DDS. Our research suggests that operators such as `until_next` may be more suitable in the domain of IIoT than true relational join semantics.

Our prior work in the Rx4DDS.NET [9] project first explored the integration of Rx with DDS. The objective in Rx4DDS.NET is to expose the suitability of Rx for DDS data processing in general-purpose languages. The project focuses on scalable concurrency for DDS applications.

For more related work, readers are directed to [9].

6. Conclusion

Edge applications in the Industrial Internet of Things (IIoT) offers new opportunities to improve the reliability of industrial assets, enabling stake-holders to progress toward higher overall uptime. Remote problem diagnosis is an important subset of IIoT edge applications where human operators must monitor asset conditions in real-time and assess likely root causes when divergent behavior is observed.

To rapidly build edge-level analytics, we developed a DSL called StreamCoCo and a virtual machine to execute StreamCoCo programs, which we call *probes*. StreamCoCo is a declarative dataflow language for analyzing and snapshotting periodic distributed streams. It is designed to be used by remote human operators (non-programmers) and therefore has fluid, English-like syntax and simple pipes-and-filter style composition.

The StreamCoCo VM uses DDS to subscribe to raw data streams, and publishes alerts and snapshots of streams when one or more probes *fire*. StreamCoCo achieves composability by using the Rx library, which supports chaining of combinators for asynchronous data processing.

Our experience strongly suggests that Rx is a highly suitable model DDS data processing. It directly supports dataflow-oriented design, which is inherently compatible with DDS's data-centric publish-subscribe communication model. Adoption of Rx in IIoT is likely to face friction due to its use of the functional programming paradigm, which requires significant learning and a mental shift compared to traditional procedural thinking.

StreamCoCo, on the other hand, does not require a background in functional programming and instead uses a much more familiar pipes-and-filter-style composition for building complex dataflow programs from basic building blocks. It allows users to leverage non-trivial DDS capabilities, such as keys, instances, filters, and multi-topic with ease.

Our planned future work includes C++ implementation of StreamCoCo for authoring data model transformations for integrating disparate IIoT subsystems. We are exploring visual programming for authoring probes. Remote lifecycle management of DSP instances and controlling probe execution (i.e., alerts, snapshots, queues) is also of interest.

7. References

[1] The Industrial Internet Consortium (IIC), http://iiconsortium.org/

[2] Gartner Hype Cycle for Emerging Technologies, http://www.gartner.com/newsroom/id/2819918 published Aug 2014.

[3] Stuart Laval, Bill Godwin, *"Duke Energy Distributed Intelligence Platform Reference Architecture"*, Volume 1, 2015

[4] Sumant Tambe, Fernando Garcia Aranda, Joe Schlesselman, *"An Extensible Architecture for Avionics Sensor Health Assessment Using Data Distribution Service"*, AIAA Infotech@Aerospace, Boston 2013

[5] Rx: An API for Asynchronous Programming with Observable Streams, http://reactivex.io/

[6] OMG Data Distribution Service, http://portals.omg.org/dds/

[7] Extensible and Dynamic Topic Types for DDS (DDS-XTypes). http://www.omg.org/spec/DDS-XTypes/

[8] Connecting DDS Apps to Web Services using Javascript and Node.js, http://blogs.rti.com/2015/01/14/connecting-your-dds-apps-to-web-services-using-javascript-and-node-js/

[9] Shweta Khare, Sumant Tambe, Kyoungho An, Aniruddha Gokhale, "Functional Reactive Stream Processing for Data-centric Publish/Subscribe Systems" In the 9th Distributed Event Based Systems Conference (DEBS 2015), Oslo, Norway

[10] DDS 1.2 Specification, http://www.omg.org/spec/DDS/1.2/