A Java-Based Real-Time Reactive Stream Framework

HaiTao Mei, Ian Gray and Andy Wellings
University of York, UK, Email: (hm857, Ian.Gray, Andy.Wellings)@york.ac.uk

***Abstract*—This paper presents a framework for real-time reactive stream processing. The approach is to extend the proposed Java 9 Reactive Streams model and integrate it with the Real-Time Specification for Java. The approach leverages a real-time version of the Java 8 Stream processing framework. Our approach addresses the major issue when using Reactive Streams in real-time: there is no way to set the timeout. Our evaluation shows there is significant improvement in the predictability of stream processing with our framework over that of one implemented using regular Java.**

I. Introduction

The problem of handling streaming data has been the focus of much attention in recent years. Much of this has centred on the *Reactive Stream* initiative (see www.reactive-streams.org), which aims to provide a "standard for asynchronous stream processing with non-blocking back pressure". In its most general sense, a stream processing system consists of a collection of modules that compute in parallel and communicate via channels [16]. Modules can be either *source capturing* (that pass data from a source into the system), *filters* (that perform atomic operations on the data) or *sinks* (that either consume the data or pass it out of the system). Stream data sources can be classified into two types [18]: batched and streaming. A *batched* data source is where the data is already present in memory, and its content and size does not change during processing. A *streaming* data source represents data that arrives dynamically, its content and size will change with time, although there is no modification of the data source by the stream itself. Streaming data source arrival can be periodic, aperiodic or sporadic. The focus of the Reactive Stream initiative is on streaming rather than batched data.

Real-time stream processing systems are stream processing systems that have time constraints associated with the processing of the data as it flows through the system from its source to its sink. Typically, the sources of streaming data may originate from an embedded system (for example, the Large Hadron Collider can output a raw data stream of approximately 1PB/s [19]) or from a variety of internet locations (e.g. Twitter's global stream of Tweet data).

The most recent version of Java (Java 8) has introduced Streams and lambda expressions to support the efficient processing of in-memory stream sources (e.g. a Java Collection) in parallel, with functional-style code. One of the primary goals is "to accelerate operations upon large amounts of data by dividing the task between multiple threads (processors)" [5]. The implementation builds upon the `java.util.concurrent` ForkJoin framework introduced in Java 7. The Java 8 Stream processing infrastructure assumes that its data source has been stored in main memory before processing, that the size of data will not change, and that the goal is to process the data as fast as possible using all of the available processors. Hence it is targeted at batched streams.

As a supplement to Java 8 Streams, Reactive Streams are being built for the forthcoming Java SE 9 as part of Java Enhancement Proposal 266 [6]. The goal is to define a minimal set of interfaces that can provide data flow management in a wide range of reactive processing systems. A `Flow` class has been defined to encapsulate the interfaces and includes a `Publisher` for *source capturing* and a `Subscriber` that acts as a *filter* or a *sink*. The interaction between the two is managed by a `Subscription`.

The goal of the work presented in this paper is to develop a Reactive Stream framework for real-time Java-based applications. By doing so we have evaluated the efficacy of the Java 9 API for use within an RTSJ (Real-time Specification for Java) context. The API is generic and can be used with our framework. However, we shall see that our framework supports a particular "pattern" of real-time Reactive Streams which supports timeouts for multiple subscribers when communication between publisher and subscriber is based of *collections* of data, rather than individual data items. Hence, for this purpose we defined an extended API. Our approach can be used with a single producer to multiple subscribers, to create networks of stream processors.

The overall approach to developing our framework is to batch incoming data into collections and then to use a real-time version of the Java 8 stream processing facilities [14] to process each batch; thus exploiting the preexisting mechanisms for efficient parallel processing. Our approach is independent of the underlying scheduling supported by the RTSJ platform. Our framework can be configured for global, partitioned, or clustered priority-based systems.

The paper is structured as follows. First, in Section II, we summarise the proposed Java SE 9 Reactive Stream support. Then in Section III our real-time Reactive Stream approach is discussed. This is followed by a description of the implementation in Section IV. Section V evaluates our approach. Related work is given in Section VI. Finally we present our conclusions.

II. THE JAVA 9 REACTIVE STREAMS MODEL

Reactive Streams defines three core concepts: the `Publisher` that produces items consumed by one or more `Subscribers`, and `Subscriptions` that are used to manage their interactions. They are defined as generic interfaces along with a set of requirements that must be met by an implementation.

- `Publisher`
  A `Publisher` represents a provider that generates a potentially infinite sequence of data elements, pushing them to its `Subscribers` according to their indicated demand. A `Subscriber` connects to a `Publisher` by invoking `Publisher`'s `subscribe(Subscriber<? super T> subscriber)` method, where a `Subscription` will be created to manage this relation.
- `Subscriber`
  A `Subscriber` is a consumer which receives each data element published by connected `Publisher`(s). Data elements are received via the `onNext(T item)` method. The `onError(Throwable throwable)` method should be invoked if an error is encountered, and the `onComplete()` method should be invoked when the `Publisher` runs out of data. When subscribing to a `Publisher`, the `onSubscribe(Subscription subscription)` method should be invoked by the `Publisher`. Typically the implementation of this method requests data element through the subscription, which is passed in its `onSubscribe` method.
- `Subscription`
  A `Subscription` manages the interaction between a `Publisher` and its `Subscribers`. A `Subscriber` receives data elements only when requested, by invoking the `Subscription`'s `request(long n)` method. The `onNext(T item)` method in the `Subscriber` will be invoked up to `n` times. A `Subscription` can be cancelled via its `cancel()` method.

Note that, the interface `Processor<T, R> extends Subscriber<T>, Publisher<R>` can be employed to provide multiple stages of data flow management.

A key motivation for the Reactive Stream enhancement to Java 9 is the claim that it hugely reduces the *back pressure* problem. The model requires the subscribers to request how many data items they are prepared to receive (via the `onNext` method), thereby allowing a subscriber to control the size of the buffer it needs to provide, and informing the publisher of how much data it can expect to send. This reduces and removes intermediate buffering inside the system.

III. THE REAL-TIME REACTIVE STREAMS

The overall goal of this work is to provide a real-time Reactive Stream processing framework. The Reactive Stream model provided by Java 9 is necessarily very generic and can be instantiated for many different applications. For real-time, the model needs to be instantiated in a constrained environment so that predictability of the application can be ensured. This section introduces a pattern for real-time Reactive Stream processing and provides a framework that supports that pattern.

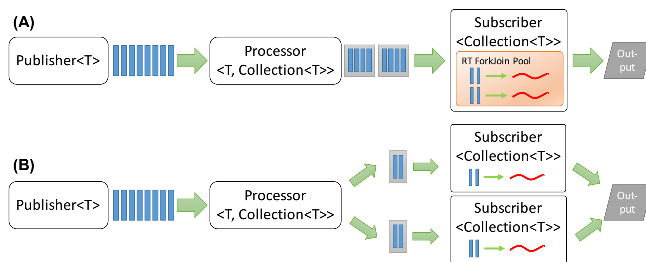


Fig. 1: The Real-time Pattern for Reactive Streams. Threads of execution are represented by red curved lines.

A. A Real-Time Pattern for Reactive Streams

Optimised push models of streaming data collect the data into micro batches in order to improve efficiency; for example the Spark streaming framework [8]. This approach has also been suggested when using Storm in a real-time environment [11]. The approach adopted by this work is to use the strictly streaming data model of Reactive Streams with batches of data (stored in Java collections) and to process these collections in parallel using real-time threads. There are several ways in which this pattern can be realised in Java; two approach are illustrated in Figure 1.

In the approach illustrated in Figure 1.A, a `Publisher<T>` attaches to a data source, takes each single data element, and connects to a `Processor<T, Collection<T>>` which groups data elements into collections. Then the `Processor<T, Collection<T>>` is connected by a `Subscriber<Collection<T>>`, which processes each collection using real-time Java 8 Streams using a real-time ForkJoin thread pool to perform the required work at the desired priority.

In the approach illustrated in Figure 1.B, instead of using Java 8 streams to process the data multiple subscribers are created and the batched data is partitioned between them. In approach A, parallel processing is achieved by using the real-time thread pool. In approach B, it comes from the use of multiple real-time subscribers.

In this pattern, the `Subscriber<Collection>` requests a collection each time with a collection size (n) and a timeout (t) from the `Processor<T, Collection<T>>`. The `Processor<T, Collection<T>>` requests n data elements from `Publisher<T>`, and stores them in a collection, whilst the `Publisher<T>` publishes them as soon as possible. The `Processor<T, Collection<T>>` pushes its collection to the `Subscriber<Collection<T>>` either when there are n data elements in the collection, or the timeout t has expired. The timeout is important because it is used to ensure that aperiodic data streams are handled within their timing constraints. The main difference between the two versions of the pattern is the size of the requested collection.

B. Supporting the Real-time Pattern

This paper provides a summary of the implementation. Full details of the interfaces and APIs presented in this work are also available [7].

The `Subscription` gathers the request from a `Subscriber` via its `request(long n)` method. With the current proposed Reactive Streams API there there is no way to set a timeout for the request. In addition, the `Subscription` only allows the maximum number of data requested on the declared type to be configured. In our real-time pattern, n in the `Subscription.request(long n)` method represents how many collections are requested from a `Subscriber<Collection<T>>`, rather than the number of elements in the collection. For clarity, therefore, we define new interfaces that support the real-time pattern.

The `RealtimeSubscription` interface is created to support requests for a collection of a required size, and a timeout, as shown below.

```
public interface RealtimeSubscription extends
    Subscription{
  public void requestCollection(int size,
      RelativeTime timeout);
  public void cancel();
}
```

In order to use a `RealtimeSubscription`, a `RealtimeSubscriber` is created, which is defined as follows:

```
public interface RealtimeSubscriber<T>
    extends Subscriber<Collection<T>>{
  public void onSubscribe(
   RealtimeSubscription subscription);
}
```

The `RealtimeSubscriber` that is declared on generic type `T` extends the `Subscriber` declared on `Collection<T>`. This guarantees any instance of `RealtimeSubscriber` processes collections, i.e. the `onNext()` method takes a `Collection<T>` as its argument.

Similarly, the `RealtimePublisher` and `RealtimeProcessor` are defined that can publish collections. This is shown below along with the `RealtimeProcessor` interface.

```
public interface RealtimePublisher<T> extends
    Publisher<Collection<T>>{
  public void subscribe(
   RealtimeSubscriber<? super T> RTsubscriber);
}

public interface RealtimeProcessor<T,R>
    extends
  RealtimeSubscriber<T>, RealtimePublisher<R> {
}
```

C. The RealtimeReactiveStream Framework

A new framework called the `RealtimeReactive-Stream` framework has been developed to support processing streams in real-time, and which uses the interfaces introduced in the previous subsection.

It provides the `RealtimeReceiverPublisher` as the publisher, which attaches itself to a data source (e.g. an infinite sequence of data elements), groups the data items into collections, and publishes these collections. A `RealtimeStreamSubscriber` consumes the collection using a real-time Java 8 Streams API to enable pipeline-style data processing.

1) RealtimeReceiverPublisher: The `Realtime-ReceiverPublisher` is both a `RealtimeReceiver` and a `RealtimePublisher`. Its role is to receive data from outside of the system and to publish it as part of the real-time framework. When a request arrives, the `RealtimeReceiverPublisher` publishes the collection either when it receives enough data or the timeout expires.

The built-in real-time subscription maintains a real-time thread to handle each request from `RealtimeStreamSubscribers`. This thread maintains a count of the data elements in the buffer. When a subscriber tries to request a collection (size=n, timeout=t) if there are not enough data elements the thread goes to the blocking queue and will wake up on the timeout. Once there are enough data elements the `RealtimeReceiverPublisher` wakes up that thread, which tries to fetch up to n data elements from the buffer and store them in a collection. In the case where there are enough data elements when a request arrives, the request handling thread immediately moves n data elements into a collection. Once the collection is ready, the `onNext` method of the `RealtimeStreamSubscriber` is invoked.

2) RealtimeStreamSubscriber: Approach A: The `RealtimeStreamSubscriber` implements the `RealtimeSubscriber` interface, and provides a real-time Java Streams API to process each collection. The `RealtimeStreamSubscriber` asynchronously processes each passed collection using Java Streams and issues the next request immediately within the `onNext` method.

The `RealtimeStreamSubscriber` maintains a real-time ForkJoin pool [14], which is a pool of aperiodic real-time threads. The priority of each worker thread is assigned when the pool is created. The real-time constraint is placed on the `RealtimeStreamSubscriber` by submitting the processing of each passed collection within the `onNext` method to a real-time ForkJoin thread pool.

A requirement of our pattern is that the collection within the `onNext` method will only be processed using Java Streams. We make use of our `ReusableStreams` (which implement the standard Java Streams API) to create a pipeline that can, unlike Java Streams, be reused on a different data source even if its terminal operation has been invoked. Once the `onNext` method is invoked, the `RealtimeStreamSubscriber` processes the collection using `ReusableStreams`, and optionally, employing the `SubscriberCallback` to further process (e.g. accumulating) the result. The `SubscriberCallback` is a functional interface, the method of which will be invoked by the

ReusableStream once its terminal operation returns, and acquires the returned result. The SubscriberCallback interface is shown below:

```
@FunctionalInterface
public interface SubscriberCallback<R> {
  void update(R result);
}
```

The pipeline of the ReusableStream is required to be initialised before processing any passed in collection. The constructor of the RealtimeStreamSubscriber uses a functional interface named ReusablePipelineInitialiser to initialise the reusable pipeline. The ReusablePipelineInitialiser is described as follows:

```
@FunctionalInterface
public interface
    ReusablePipelineInitialiser<T> {
  public void initialise(
    ReusableReferencePipeline<T> p);
}
```

The functional interface is used to enable the RealtimeStreamSubscriber to take advantage of lambda expressions which make the code more concise. An example which calculates how many words have been received by a publisher is shown below

```
long count = 0;
RealtimeReceiverPublisher<String> publisher;
RealtimeStreamSubscriber<String> subscriber =
    new
  RealtimeStreamSubscriber<>(
   1024, /* request collection size */
   new RelativeTime(5000, 0), /* timeout */
   p -> p.flatMap(line ->
    Stream.of(line.split("\\W+"))).count());
subscriber.setCallback(
  r -> count += (long) r);
publisher.subscribe(subscriber);
```

The reusable pipeline in this example counts how many words are in a collection in the onNext method, and is the same as using standard Java Streams. The callback accumulates all the local results.

3) RealtimeStreamSubscriber: Approach B: The RealtimeStreamSubscriber in this approach is very similar with the one that was described in approach A. The differences are how to process the data, the time when to issue the next request within the onNext method, and the execution-time server registration. An additional requirement is that the publisher should maintain a FIFO queue that stores all the requests from multiple subscribers.

When processing each passed collection, each subscriber uses a sequential Java Stream, which is evaluated by its real-time ForkJoin thread pool that contains only one worker thread. The next request will only be issued after the current collection has been processed. This means the request from an idle subscriber goes into the publisher's request queue ahead of a request from a busy subscriber. In the implementation, the next timeout is configured to the absolute time, which is the time of the previous timeout plus the timeout value. In order to bound the impact this has on other threads in the system (see next section), each real-time ForkJoin worker thread is required to register to the execution-time server in this approach.

4) Bounding the Impact of Data Flow Processing: Typically stream data processing is computationally-intensive, and the unpredictability of data flows makes the corresponding CPU demand unpredictable. Moreover, stream processing is typically latency-sensitive.

In an RTSJ runtime environment, stream processing is most likely to occur within a soft real-time task. With all such soft real-time activities, there is tension between achieving a short response time without jeopardising any hard real-time activities. Running stream data processing at the lowest priority in the system will not give good response times, but running it at too high a priority might cause critical activities to miss their deadlines. Hence, an appropriate priority level must be found, and any spare CPU capacity that becomes available must be made available as soon as practical.

The impact of data flow processing is bounded by associating servers that are described in [14] with real-time thread pools. Performing the previous example with real-time constraints requires the server and the priority to be configured. A real-time ForkJoin thread pool with the desired priority associated is created to process each collection using the given pipeline.

```
long count = 0;
RealtimeReceiverPublisher<String> publisher;
RealtimeStreamSubscriber<String> subscriber =
    new
  RealtimeStreamSubscriber<>(
   1024, /* request collection size */
   new RelativeTime(5000, 0), /* timeout */
   new PriorityParameters(26), /* priority */
   new DeferrableServer(...), /* server */
   p -> p.flatMap(line ->
    Stream.of(line.split("\\W+"))).count());
subscriber.setCallback(
  r -> count += (long) r);
publisher.subscribe(subscriber);
```

IV. Implementation

Real-time Reactive Streams are implemented using the JamaicaVM [4] RTSJ, which provides multiprocessor support including affinity sets. There are three components: the RealtimeReceiverPublisher, the RealtimeStreamSubscriber and the ReusableStream.

A. The RealtimeReceiverPublisher

The buffer in the RealtimeReceiverPublisher is implemented as a linked list rather than multiple fixed arrays. This is because when handling requests from mul-

tiple subscribers each subscriber may request collections of different sizes. The data elements are stripped out the buffer by unlinking them from the linked list and added into a collection that supports $O(1)$ random access (such as an `ArrayList`). When handling multiple subscribers, all threads blocking on the buffer are notified once there are enough data elements for the smallest request. The RTSJ `HighResolutionTime.waitForObject` method is employed to block the calling thread, and wakes up the thread with a timeout.

B. The `RealtimeStreamSubscriber`

The collection within the `onNext` method will only be processed using Java Streams. This is guaranteed by using `ReusableStreams` (see III-C2). The `RealtimeStreamSubscriber`'s constructor accepts a functional interface to define the processing pipeline so that lambda expressions can be used.

Note that when making requests within the `onNext` method, the handling in the `Publisher` must avoid recursion. This results in a stack overflow when dealing with infinite data sequences. Recursive invocation can be avoided by using dedicated threads to handle the `request` and `onNext` events.

C. The `ReusableStream` Pipeline

Recall that the purpose of `ReusableStreams` is to create a pipeline of operations which may be repeatedly applied to different collections. In addition, `ReusableStream` must remain compatible with the existing Stream API.

`ReusableStreams` are defined as an interface that extends the Java `Stream` interface. `ReusableStream` define a method named `processData` which takes a reference to a data source (collection) to be processed, and optionally a callback which is called to present the result.

In a `ReusableStream`, operation pipelining uses a linked list. Each node maintains one intermediate operation and its arguments, and each intermediate operation returns a new node that will be appended to the tail of the linked list. When the terminal operation is invoked, the execution thread travels through the pipeline, and performs each operation on each data element. In order to make a pipeline reusable, the terminal operation is added to the linked list as well, rather than forcing evaluation. This is the only difference between the use of standard Java Streams and `ReusableStreams`.

D. Real-Time Stream Processing

The data flow is processed at different priority levels using `RealtimeStreamSubscribers`. This is achieved by processing the collection within the `onNext` method using `ReusableStreams`, which will be submitted to the real-time ForkJoin pools at the configured priority for its execution.

In a globally scheduled system, each worker thread within the real-time ForkJoin thread pool can execute on (or migrate to) any available processor. No CPU affinity is applied. In a fully-partitioned system, each worker thread within the real-time ForkJoin thread pool is constrained to execute on one processor, and task migration is forbidden using CPU affinity. The implementation uses `javax.realtime.AffinitySet` to pin each worker thread within a real-time ForkJoin pool to different processors. A semi-partitioned system extends the fully partitioned system, so that a certain number of tasks can migrate to a set of allowed processors. In a semi-partitioned system, different worker threads are allocated with different affinity sets, which determine the set of processors the task can migrate to.

V. Evaluation

This section evaluates the latency of stream processing using our real-time Reactive Stream framework. We first demonstrate that the framework provides guaranteed latency when using real-time Reactive Streams compared with a regular (non-real-time) Java framework. We then explore the latency distribution and the efficiency of using the different approaches to support the real-time stream processing pattern that was mentioned in section III.

The experiments were performed on a platform with a 3.7 GHz Intel Core i7 processor (with 4 physical cores), running Debian 7 Linux (3.2.0-4-rt-amd64 real-time kernel), and with hyperthreading off. The Linux "taskset" shell command was employed to select three physical cores for the experiment. aicas JamaicaVM version 6.5 is the RTSJ-compliant JVM used.

All times are in milliseconds. Minimum and maximum inter-arrival times are represented as 'MIT' and 'MAT' respectively. Worst-Case Execution Times are synthetic.

A. Latency Guarantees

This experiment considers using a real-time Reactive Stream to process a data flow on a single processor (Processor 2). Processor 2 also executes three periodic real-time threads at the same time. The experiment demonstrates that the latency of processed data flows can be guaranteed by using real-time Reactive Streams.

The real-time receiver thread maintained by the publisher and the underlying real-time stream processing framework have medium priority. The data flow simulated in this experiment is described in Table I. The timeout of the real-time Reactive Stream is configured to be 10, and the request size is set to 1024, which ensures that the data flow in this experiment will not be processed except when timeouts expire. The real-time characteristics of other real-time threads are shown in Table II. Note that, the response times of these lower priority activities are not of interest in this experiment.

TABLE I: Data Flow Characteristics.

MIT	MAT	WCET	Deadline	Generated From
200	400	33	60	Processor 1

The publisher is started at time 0, and all real-time threads are released according to their release times. Processor 1 starts

TABLE II: Periodic Low Priority Real-time Activitiy Characteristics

Name	Priority	WCET	First Release	Period	Deadline	Processor ID
T1	Low	28	0	100	100	2
T2	Low	28	130	200	200	2
T3	Low	28	50	400	400	2

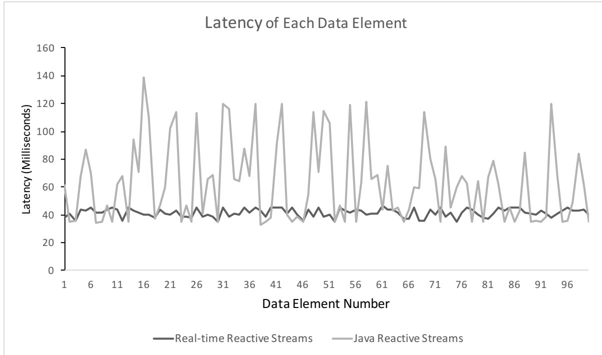


Fig. 2: The Latency of Each Data Element In A Data Flow.

to generate the data flow that contains 100 strings after 400 milliseconds. The latency of each data element in the data flow is measured, and illustrated in Figure 2.

The latency of each data element varies significantly when using the standard Java Reactive Streams. As a consequence, several data elements miss their deadlines. This is because the processing suffers from priority inversion on Processor 2, where all the worker threads in the standard Java ForkJoin pool are pre-empted by the periodic real-time threads.

The latency is bound to a range when using real-time Reactive Streams, as illustrated by the grey line in Figure 2. Priority inversion is avoided and each of the data elements in the flow meets its deadline. Note that the variance of the latency when using real-time Reactive Streams is introduced by the waiting time before each timeout occurs, because data can arrive at the publisher at any time within that period. The experiment was repeated 30 times, the results do not vary significantly.

B. Latency Distribution

This experiment considers the latency distribution when using the stream subscriber (approaches A and B, see Section III). In this experiment, the data flow is similar to the one used in Section V-A, but with an MIT of 5 and MAT of 65. The speed of the data flow is exactly 80 messages per second (in the 2-core experiments) and 160 message per second (in the 4-core experiments). The timeout for both approaches is one second.

In approach A, the stream subscriber receives all arrived data from the publisher when the timeout occurs. Each subscriber in approach B receives an equal share of the total data elements when its timeout occurs. In addition, we also optimised the Java 8 Stream framework so that the data in each collection will be processed by the ForkJoin pool in the order of arrival, rather than Java 8 Stream's normal processing order. When using standard parallel streams, each worker thread of the ForkJoin thread pool recursively splits the current collection into two parts until the size is less or equal to $Max(1, SizeOfCollection/Parallelism)$. During splitting, alternately the left then right splits are pushed into the worker thread's task queue. The worker thread accesses its task queue in a LIFO order, and steals tasks from other workers using FIFO ordering when idle. This all means that data elements are processed in an order very different from the order in which they arrived.

In this experiment, each data element is given the same WCET for its processing. This is configured in different experiments (shown in Table III) to be 4 (Low), 5 (Mid), 20 (High).

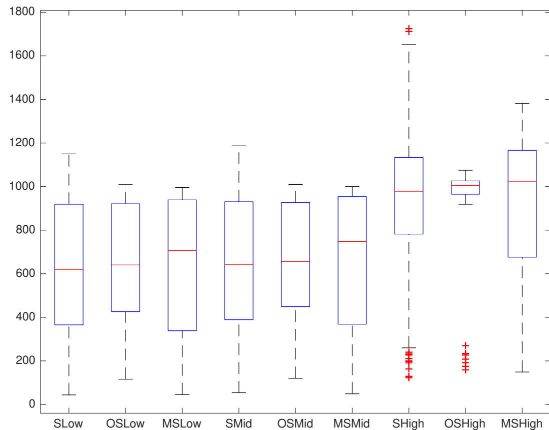
TABLE III: Latency Distribution Experiment Configuration

Name	Approach	Processing Framework
S	A	RT Parallel Stream
OS	A	RT Optimised Parallel Stream
MS	B	RT Sequential Stream

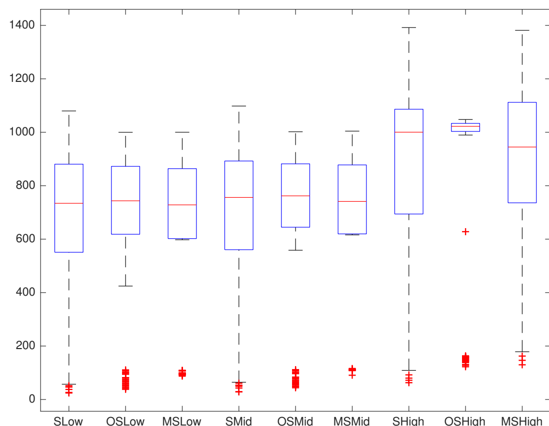
In approach A the real-time (optimised) parallel stream is configured to use 2 and 4 processors, and there are equivalent amounts of real-time subscribers for approach B. The publisher runs on another processor. The experiments were performed 30 times, the latency of data flows is measured and shown in Figure 3.

The latency distribution of the standard parallel streams is higher compared to the other approaches because the processing order that is used by may result in LIFO processing ordering, thus the maximum latency is increased.

There is no significant difference in the maximum latency of the optimised parallel and approach B when the load is less than or equal to MIT. However, when the processing time is bigger than MIT the processing order will have significant impact on the maximum latency, which is shown in the last three plots in both Figure 3a and Figure 3b. The maximum latency is reduced significantly when the optimised real-time stream subscriber compared to approach B. The reason is that when the WCET is less or equal to MIT, the latency of the first data element can represent the maximum latency. For example, consider the case where there are $N(1, 2, 3...N)$ data elements, the arriving interval is always MIT (i.e. the worst case), and the parallelism is P. The optimised parallel streams use a FIFO order, the latency of ith data is represented by: $Latency(i) = \lceil i/P \rceil WCET + (N - i + 1)MIT$. The function $Latency(i)$ will not increase when WCET is less or equal to MIT. Similarly, the latency of ith data when using approach B is: $Latency(i) = (i \, mod \, (N/P))WCET + (N - i + 1)MIT$. In each partition of the input, the latency will also not increase when WCET is less or equal to MIT.



(a) The Latency of Data Elements When Using 2 Processors.



(b) The Latency of Data Elements When Using 4 Processors.

Fig. 3: Latency Distribution Experiment Results.

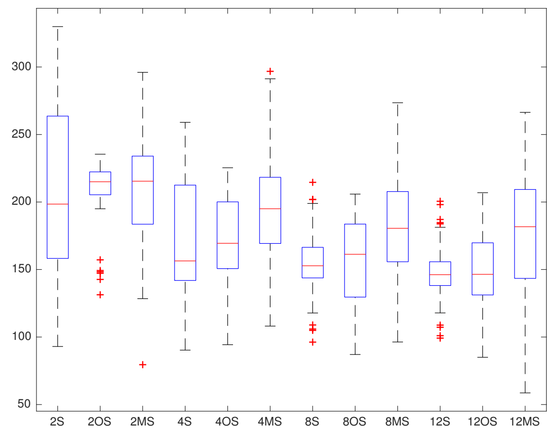


Fig. 4: Mean latency for different numbers of cores.

C. Efficiency Evaluation

This experiment evaluates the overall efficiency of the two approaches A and B described in Section V-B. Approach A includes using both standard and optimised Java 8 parallel streams, and B uses sequential Java 8 streams with multiple subscribers. In this experiment MIT=2, MAT=60, and the execution time for processing each data element is 30 milliseconds. All approaches are configured with a timeout of 200 milliseconds.

The experiment was performed on a 16 core AMD Opteron 8350, 1GHz processor platform. The latency of 100 data elements is measured using 2, 4, 8, and 12 cores for stream processing. Performing the experiment 30 times, the latency is measured and shown in Figure 4.

The mean latency of approach A that uses standard Java 8 parallel streams is consistently smallest, followed closely by approach A with our optimised parallel stream. The mean latency when using optimised parallel streams is higher because it introduces more splitting and creates more stream instances than standard streams. If the number of data elements in a collection is N, and the parallelism of processing is P, the splitting in the optimised parallel stream is $O(N-1)$, whilst in the standard parallel stream it is $O(\sum_{i=0}^{log_2 N - log_2 Max(N/4P,1)} 2^i)$. However, the difference in the two approaches is very small.

The mean latency of using multiple subscribers (approach B) with sequential streams is the highest, even though it only creates $O(P)$ stream instances. This is due to the fact that the data flow rate is not constant; the arrival interval of each data element is not always MIT. Because each subscriber has to request $Timeout/MIT/NumberOfSubscribers$ data elements each time in order to avoid any data elements missing their deadlines, poorer load balancing is possible with approach A which increases mean latency. This issue becomes more significant when the system is fully partitioned and there are interferences from higher priority activities, because the data can not be processed by another subscriber once it has been allocated to another. The work stealing algorithm that is used by Java 8 parallel streams can balance the load amongst different threads.

VI. Related Work

Whilst Java 8 Streams target static data sources, Java 9's Reactive Streams take the first step towards true streaming data processing. This section summarises related stream processing frameworks, languages, and several real-time streaming frameworks.

StreamIt [9] is a programming language that was specifically designed for stream processing. StreamIt targets a range of platforms including embedded systems, high performance systems, and large scale systems etc. StreamIt defines several concepts for stream data processing, for example, a `filter` is used to operate on data. A Java-like API is also provided. Borealis [10] targets stream processing in distributed systems, and defines stream operation abstractions, such as `join`, `map`

etc., which are written in Java. However, neither of these approaches provide real-time support.

Storm [3], Samza [2], and Heron [12] target distributed stream processing. Stream processing jobs are defined to be directed acyclic graphs (DAGs), where vertices represent the operations on data and the edges represent the data flow. Spark Streaming [8] extends Spark [1] to support distributed stream data processing by periodically grouping the data in a flow into micro batches, and then processing these micro batches using the Spark engine. None of these systems support real-time constraints.

StreamFlex [15] proposes a latency-guaranteed stream processing approach, which is inspired by the RTSJ and StreamIt. Similar to StreamIt, StreamFlex also defines several stream operation abstractions, such as `filters`, which have similar functionality to those in StreamIt. StreamFlex provides latency guarantees by patching the runtime virtual machine to support real-time periodic threads, a memory model that avoids interference introduced by garbage collectors, and isolation of computational activities. Bounding the impact of a stream processing job to other real-time activities in StreamFlex is not provided, and priority assignment is not supported.

Mattheis [13] investigates work stealing algorithms in parallel stream processing in soft real-time systems. The impact of employing different work stealing strategies and queueing approaches on stream data processing are investigated. The proposed strategy uses FIFO ordering when taking tasks from the global queue, and using LIFO ordering for stealing from other workers' local queues. Note that this is the Java 8 Stream evaluation model, which we used unchanged.

A real-time Storm is proposed by [11]. It extends Storm and defines a real-time processing stack consisting of a real-time OS, a real-time Java runtime environment, and real-time Storm. The `Spout` (source of a stream) and the `Bolt` (consumer) are extended to be sporadic real-time threads with configurable priorities, computations times, and minimum interval times. A fixed-priority scheduler is provided. Storm uses an eager computation model, which does not provide all of the optimisation opportunities of the lazy model that is used by Java 8 Streams [17].

VII. Conclusions

This paper has proposed a pattern for real-time Reactive Stream processing based on micro-batching input data items into Java collections. This pattern has been instantiated with a real-time Java 8-based stream processing infrastructure and an infrastructure based on multiple real-time subscribers. Where possible we have used the proposed Java 9 Reactive Stream APIs and have found then, in general, to be sufficiently flexible for our requirements. However, our pattern does make use of timeouts which are not present in the current API.

Our evaluation shows that both instantiations of our real-time stream processing pattern are more predictable than a standard non real-time version. The instantiation that is integrated with an optimized Java 8 real-time stream processing framework showed better scalability than that with multiple real-time subscribers. This is because more efficient load balancing can be obtained.

References

[1] Apache Spark - Lightning-Fast Cluster Computing. http://spark.apache.org/. Accessed December 5, 2015.

[2] Apache Samza. http://samza.apache.org. Accessed December 5, 2015.

[3] Apache Storm. http://storm.apache.org/. Accessed December 5, 2015.

[4] Jamaicavm — aicas.com. https://www.aicas.com/cms/en/JamaicaVM. Accessed December 1, 2015.

[5] JEP 107: Bulk Data Operations for Collections. http://openjdk.java.net/jeps/107. Accessed December 5, 2015.

[6] JEP 266: More Concurrency Updates. http://openjdk.java.net/jeps/266. Accessed March 22, 2016.

[7] Real-Time Reactive Streams. https://github.com/RTSYork/Realtime-Reactive-Streams. Accessed March 22, 2016.

[8] Spark Streaming — Apache Spark. http://spark.apache.org/streaming/. Accessed December 5, 2015.

[9] StreamIt-Research. http://groups.csail.mit.edu/cag/streamit/shtml/research.shtml. Accessed December 5, 2015.

[10] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[11] P. Basanta-Val, N. Fernández-García, A. Wellings, and N. Audsley. Improving the predictability of distributed stream processors. *Future Gener. Comput. Syst.*, 52(C):22–36, Nov. 2015.

[12] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[13] S. Mattheis, T. Schuele, A. Raabe, T. Henties, and U. Gleim. Work stealing strategies for parallel stream processing in soft real-time systems. In *Architecture of Computing Systems–ARCS 2012*, pages 172–183. Springer, 2012.

[14] H. T. Mei, I. Gray, and A. Wellings. Integrating java 8 streams with the real-time specification for java. In *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems*, page 10. ACM, 2015.

[15] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. Streamflex: high-throughput stream programming in Java. *ACM SIGPLAN Notices*, 42(10):211–228, 2007.

[16] R. Stephens. A survey of stream processing. *Acta Informatica*, 34:491–541, 1997.

[17] X. Su, G. Swart, B. Goetz, B. Oliver, and P. Sandoz. Changing engines in midstream: A Java stream computational model for big data processing. *Proc. VLDB Endow.*, 7(13):1343–1354, Aug. 2014.

[18] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer, 2002.

[19] X. Vidal and R. Manzano. Taking a closer look at lhc. http://www.lhc-closer.es/1/3/12/.