Master thesis

WORKING WITH UNRELIABLE OBSERVERS USING REACTIVE EXTENSIONS

November 10, 2016

Dorus Peelen (s4167821)
Computing science
Radboud University
dpeelen@gmail.com

—

Supervisors:
Radboud University: Rinus Plasmeijer
Aia Software: Jeroen ter Hofstede

Contents

1 Introduction **5**
1.1 Overview of thesis . . . 7

2 Background **8**
2.1 Aia Software . . . 8
2.2 Current situation . . . 8
2.3 The use case . . . 9
2.3.1 Technical details . . . 10
2.3.2 Potential problems . . . 11
2.4 Problem description . . . 11
2.5 Desired Properties of the new system . . . 11

3 Overview of Rx **13**
3.1 Basic Ideas of Rx . . . 13
3.2 Observer and Observable . . . 15
3.3 Hot vs Cold observables . . . 15
3.4 Marble diagrams . . . 16
3.5 Transformations on Event Streams . . . 16
3.6 Schedulers . . . 16
3.7 Control process in Rx . . . 17
3.8 Interactive vs reactive . . . 18
3.9 Push-back and backpressure . . . 18
3.10 Operators with unbounded Queue . . . 19
3.11 Explanation and example of different operators in Rx . . . 19
3.11.1 `Select` . . . 19
3.11.2 `SelectMany` . . . 20
3.11.3 `Where` . . . 20
3.11.4 `Delay` . . . 20
3.11.5 `Merge` and `Concat` . . . 21
3.11.6 `Buffer` and `Window` . . . 21
3.11.7 `GroupBy` . . . 21
3.11.8 `Amb` . . . 21
3.11.9 `Debounce`, `Sample` and `Throttle` . . . 22
3.11.10 `ObserveOn` . . . 22
3.12 Related work . . . 22
3.12.1 TPL . . . 23
3.12.2 iTasks . . . 23

4 The experiment **25**
4.1 Important properties . . . 25
4.2 Test architecture . . . 26
4.2.1 Design . . . 26

4.2.2 Concepts . . . 26
4.3 Measurements . . . 27
4.4 Scenario setup . . . 28
4.5 Test data . . . 28
4.5.1 Data size . . . 28

5 Possible solutions **29**
5.1 The problem . . . 29
5.2 Solutions in RxJava . . . 29
5.3 Solutions requirements . . . 30
5.4 Solutions implemented in experiment . . . 30
5.5 Usefulness of solutions . . . 31

6 Results **33**
6.1 Solution: Nothing . . . 33
6.2 Solution: WithObserveOn . . . 35
6.3 Solution: LockQueue . . . 36
6.4 Solution: WithSample . . . 36
6.5 Solution: DropQueue . . . 37

7 Conclusions **40**

Abstract

As a project for Aia Software[1], we investigated if Reactive Extensions can be used to monitor a message stream. These additional monitors cause extra stress on the system as they will also function as receivers. This may increase the delay time of the original message stream or the loss of messages in the event that the system is overloaded. The monitors might even prove to be unreliable. Different reactive systems have been studied to investigate how they deal with unreliable receivers. A number of possible solutions were selected that are available in Reactive Extensions and that allowed for implementation. The level of performance of the solutions was tested in a simulation of the monitored system. This has resulted in six solutions, each with its own advantages and drawbacks. Two of these solutions proved to be suited and one performed especially well.

[1]Currently Aia Software, A Kofax Company

1 Introduction

Data streams are everywhere on the internet. For example to request a web-page, view a video, use a web-cam, receive stock market data or get live soccer results. Even email can be regarded as a data stream.

Synchronous data streams provide information at a constant rate until consuming stops (e.g. a web-cam). They can also provide a short stream with a clearly defined end such as a web-page or an email message. Synchronous streams function in email by requesting the latest status at certain intervals – a process which is known as polling. Each time a request is made, an amount of new information is delivered. The overhead increases as the number of requests goes up; with less frequent requests, delay times increase until new data becomes available. All these streams are pull based, in other words they are served upon request.

An asynchronous stream or event stream is a different type of data stream which is push based. Asynchronous streams usually do not transfer data constantly and stay open over time. They notify when new data is available. This makes them more useful for live data like email or soccer results because there is no need for polling. Instead, a notification will be issued in case of a new event such as a new email message. The advantage of pushing data in this manner is the fact that it requires minimal overhead and realises fast response times in case of new information.

The reactive programming paradigm is a key element in this thesis. Before presenting it in more detail, other - more commonly used - programming paradigms will be discussed first, i.e. imperative programming and two types of streaming models (streams/pipelines and publisher/subscriber). This approach was chosen to create a better understanding of the main reasons for the implementation of reactive programming.

Traditionally, imperative programming focuses on how a program works. It describes every step of the process including all possible exceptions and deviations. IO calls are hidden as synchronous activities and they will block the system until they are completed. All data is handled monolithically. Once the data has been processed, blocking calls return the results and the data is forwarded. This will then start off the next step in the process. This circular process will continue in the same manner until output is realised.

Streaming model

A more flexible way to handle data is to evaluate it in a stream or a pipeline. In a pipeline, the data is split up into separate messages and operators on the pipeline modify one single message at a time. Programs are no longer written from the perspective of the program flow but 'data' is the dominant principle. Often only the first and last messages require a special case. In a synchronous stream, the size of the dataset is known as well as the arrival time of the next message. Certain data parts may be skipped and earlier parts can be retraced. In an asynchronous stream, the only available direction is forwards as the source pushes data and the size of the dataset will only be known after the end of the data stream.

Pipelines decouple data and asynchronous code decouples the time aspect (i.e.

decouples synchronization). The publisher / subscriber pattern also decouples space (i.e. decouples sender from receiver) and in this way fully decouples communication between participants. The publisher does not need to have any information about subscribers, it simply publishes data into a pipeline. Subscribers receive this data when they subscribe to the pipeline. Multiple subscribers can be subscribed to a single pipeline or none at all. This allows for greatly improved scalability.

Reactive programming

Working with asynchronous data streams is quite different from using synchronous data streams. With an asynchronous data stream, it is not known whether the next piece of data will arrive the next millisecond, the next second or the next day. Another difference is that reactive code not only deals with timing but also produces timing-based events. When code reacts to arriving data, this is called reactive programming. This, in contrast with imperative programming where it is defined when something happens. Reactive code, on the other hand, defines what to do when something happens.

The advantage of an asynchronous streaming model is huge. Because you only process one element at a time, there is no history, but the space requirements are decreased to a single element; time is constant for every element. Data can be processed as the first element arrives and there is no need to wait for the complete result set. This enables working with endless collections and the results can be shown as soon as the data is available.

However, asynchronous programming poses difficulties because of the limited support in a large number of programming languages. Often callbacks are used and a continuation function is passed as a parameter. The flow of the program will continue in the continuation function once the asynchronous call is completed. This disrupts the normal control such as loops and if/then statements and causes the program to be spread out over many functions. Loops can be generated but only by recursively calling the continuation function. Programmers often refer to this situation as 'callback hell'.

For a programmer that is used to imperative programming, it is often tempting to switch back. Many libraries even require this because they do not support asynchronous calls. Programmers who are used to the imperative process may find it difficult to come up with the solution in a reactive style.

In addition to callbacks, a number of more modern techniques are slowly being introduced in modern languages. Lambda expressions help with defining callback functions; async/await structure allows imperative style code on asynchronous calls. Task objects[2] which represent the result of an asynchronous computation, mix callbacks in a more Object Orientated style. They come with a library to run tasks in parallel, chain them or run them recursively.

Despite all these new techniques, many problems remain unresolved. Debugging tools need to be adapted because call stacks no longer point to the calling code but to the scheduler that coincidentally scheduled the call. Tasks that are on hold, are no longer

[2]Task is used in C#, but Java has a similar structure with Future and Javascript with Promis etc.

found in the active thread list but are hidden in a task queue on various threads or on the schedulers. Testcases are hard to write because asynchronous calls might involve waiting.

A library that aids writing reactive programs is Reactive Extensions[1] or Rx. This is a library for writing queries on event streams. Rx uses Observables that represent multiple results or a stream of results of an asynchronous call. Similar to Task object that only holds a single result, Rx has many operators that chain, transform or otherwise mix these Observables - allowing for easy composition of asynchronous calls. Schedulers are introduced to abstract timing. With the test scheduler for example, test cases can be written that span days but actually run in milliseconds.

It is worth mentioning that modern libraries are adding asynchronous calls and are moving towards asynchronous programming. Synchronous is becoming less popular.

1.1 Overview of thesis

This thesis is part of a project for a company, i.e. Aia Software. They want to investigate if reactive programming can be used in a reliable manner for a specific use case (chapter 2.3) using Reactive Extensions (Rx) (chapter 3). The important properties are explained in section 4.1 and a simulation of the use case is introduced in section 4.2. Various solutions (chapter 5) inspired by similar systems and other Rx implementations are tested in the simulation. The results are stated in chapter 6 and the conclusions are described in chapter 7.

2 Background

This chapter introduces the use case from Aia Software that contains the problem that is investigated. It shows the current situation and what problems are expected from the system extension.

2.1 Aia Software

Aia Software provides an infrastructure for customer communication for companies that want to automate customer communication. This enables businesses to send welcome letters, insurance policies etc. Figure 1 shows the outline of this process to configure the infrastructure. The company can generate text blocks, include specific paragraphs for different situations or may create different versions of the same text. The system puts together these text blocks in a single text and tailors the text to each individual customer. The system also decides which message to send to whom and at what time. For example, customers should not receive advertisements for products they already use. After completing these steps, the actual email or letter is created and delivered.

While most work takes place in the first half of the process (creating the text and setting up the integration rules and tailoring to individual customers) in the back end most computer time is spent on executing these rules. The back end servers are rented from cloud providers, companies that operate many servers and rent them out on demand. In the cloud, the entire process of combining text blocks and tailoring them for specific customers is a job that is executed on request.

Figure 1: The process to automate communication to the customer.
Source: Kofax.com

2.2 Current situation

Jobs are constantly sent to the cloud, where they have to be scheduled on the servers. The Cloud Manager (CM) functions as a gatekeeper to the cloud; it receives and assigns the jobs to the various servers. To allow monitoring, the Cloud Manager produces events when work is processed and these events are shared to the Failover that monitors the health of the cloud. If the current cloud fails, the Failover transfers all work to a different

cloud. This is a very invasive (and expensive) operation that should not be started erroneously. This means that the message stream to the Failover must be reliable so no wrong conclusions are reached. See figure 2.

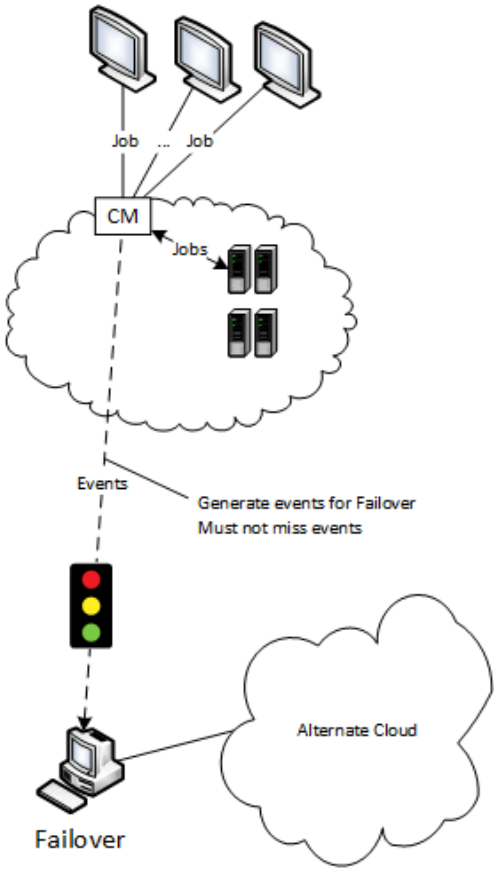

Figure 2: Current cloud configuration

2.3 The use case

Currently, the Cloud Manager and the Failover form a closed system that cannot be observed. The messages from the cloud stream directly to the Failover. However, employees or clients may want to know how much work is flowing through the cloud and how busy or how responsive the systems are. This information can also be extracted from the event stream from the Cloud Manager to the Failover. Therefore the Cloud Manager would need to share these events to monitor applications. A monitor can show the progress of the work to employees or clients. See figure 3 for the new situation.

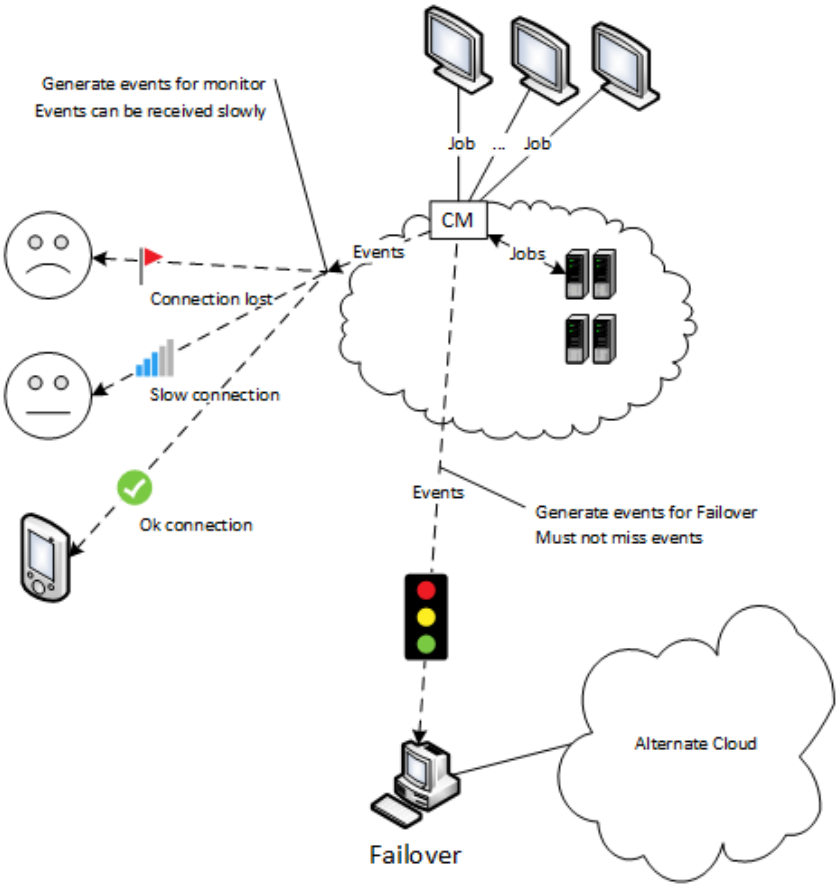

Figure 3: Cloud configuration for the use case

2.3.1 Technical details

Various users send jobs to the cloud, the Cloud Manager receives these jobs and distributes them over the cloud based on pre-set rules, load numbers, etc. When jobs start or stop, an event message is generated and is sent to the Failover. The Failover uses these messages to track the health of the cloud.

In the use case, these events are shared with various monitor applications that display the progress of various jobs to employees or clients. These monitor applications might run on slow and unreliable hardware such as smart phones with poor connections. Moreover, these platforms may cancel their connection at any time or disappear without notice.

It is also important for the Cloud Manager to communicate with the different listeners without causing any delay itself. If the Cloud Manager becomes delayed, all jobs will be delayed as well. Only the Failover is important enough to delay the work. However, in general we assume that the Failover is fast enough to process all events from the Cloud Manager at maximum speed.

In order to create this extension, the event stream to the Failover must be duplicated.

This duplication may be necessary a number of times as there might be multiple monitors at different locations. Currently the event stream is very simple; events are generated by the Cloud Manager and are directly delivered to the Failover. No special messaging service is required because this is a one-on-one relation. Furthermore, the Failover is completely under control of Aia Software so its speed and reliability can be guaranteed.

In the new situation, the event stream becomes a one-on-many relation so a messaging service is required. When duplicating a stream, the performance of the original stream is also affected. Therefore, the messaging service must be able to duplicate the stream in such a way that it does not affect the performance of the original stream too much. After all, a disturbed stream can cause undesired results when the Failover reaches incorrect conclusions as it will unnecessarily move the system to a different cloud.

Currently the Failover and Cloud Manager are written in C# so the extension should also be written in this language.

2.3.2 Potential problems

A complicating factor when the Cloud Manager splits the datastream to the Failover and shares it with the monitors, may be the fact that the monitors will run on potentially slow systems that may have unreliable connections. They might also be (intentionally or unintentionally) malicious and behave in unexpected ways. Because Aia Software controls the Failover, they guarantee that it processes all messages faster than they are generated. The newly added monitors do not give any such guarantees at all. This creates additional difficulties for the messaging service as some monitors might not process the stream as fast as it arrives. But it would still be desirable for the other monitors to receive the data-stream as reliably and completely as possible.

2.4 Problem description

Aia Software wants to investigate if it is possible to monitor the Cloud Manager's event stream with multiple monitors without affecting Failover reliability as these monitors are potentially unreliable. They may process events slower than they are generated and they may process events with a delay. They may also be engaged in other work and not process anything for some time. In that case, they have to catch up or they could randomly get disconnected and completely disappear. Monitors can also be malicious and deliberately try to disturb the system.

By default Rx does not provide answers on how to handle unreliable receivers. Therefore a solution strategy needs to be found to avoid such issues.

2.5 Desired Properties of the new system

The goal is to find a way in Rx to connect to different observers. The essentional observers (such as the Failover) should not get disturbed by the nonessentional observers (the monitors) that are potentially unreliable, unresponsive or malicious. Nor should the Cloud Manager run out of resources. Thirdly, well-behaving nonessential observers (i.e.

those that perform well enough to keep up) should not be disturbed by poorly-performing siblings.

3 Overview of Rx

The Rx library, called Reactive Extensions or ReactiveX [2] supports programming on event streams in a reactive way. It makes it possible to write (LINQ[3] style) queries on event streams to easily transform and manipulate streams (map, filter, reduce, every, etc). Rx has been ported to many languages: C#, Java, php, JavaScript and many others. This thesis will focus on Rx.Net written in C# because this is most relevant for the simulation.

In this chapter, some of the basic concepts behind Rx will be discussed. A way to visualize streams and some query operations will be shown and the manner in which Rx works internally with schedulers and concurrency will be explained.

Schedulers are especially important in understanding how Rx divides the work load. They generate internal queues and manage the manner in which different operators can support working with long queues.

3.1 Basic Ideas of Rx

Rx's Observer pattern is dual to Iterator [4]. The observer pattern from Rx is very close to the one known from the "gang of four"[5, p.326-338], but with the addition of a completion message (either completed or error).

Reactive Extensions uses continuation-passing style [6] [7] where the system is instructed how to act when a new event arrives on the stream. Before the arrival of a new message, the system is engaged with other tasks. However, Rx leverages the input (Producer/Publisher/Observable) and output (Consumers/Subscriber/Observer) to first-class citizen, to avoid the unusual control structures introduced by the continuation-passing style. In essence, this allows for simple reactive programming. The next paragraphs show the differences between CPS, async/await and Rx code.

Callback (CPS):

```
private void onSave(err) {
  if (err)  {
    //error handling
    return;
  }
  console.log('success');
}

public void save() {
  Something.save(onSave);
}
```

First, code with a callback. The code starts with *Something.save*(*onSave*) but then continues with *onSave* declared elsewhere. This breaks the flow of the program because the program continues elsewhere.

Lambda expression:

```
public void save() {
  Something.save((err) => {
    if (err)  {
      //error handling
      return;
    }
    console.log('success');
  });
}
```

When lambda expressions are used, the code is already easier to follow. The code is all declared together but the code inside the Lambda will only run after the save function completes. Any code placed below the *save* function (outside the lambda expression) would run in parallel with the save call.

Task async/await:

```
public Task save() {
  Try {
    var save = await Something.save();
    console.log('success');
  } catch (err) {
     //error handling
  }
}
```

With async/await the code looks much more like usual synchronous code, if *Something.save*() had been a synchronous call, the code would have been the same except the *await* keyword would have been missing and the return type of the function would have been *void*. This function will return a *Task* object when it reaches the *await* keyword and the code after the *await* keyword will only run once the *save* function returnes.

Rx:

```
public IDisposable save() {
  return Something.save()
    .subscribe((err) => {
        //error handling
    }, () => console.log('success'));
}
```

Finally, the same code in Rx. Rx seperates the callback for error and success, also (not used in this example) as Rx works with streams, there is a third callback that will be called for individual items on the stream. Also additonal operators can be added before *subscribe* to modify the sequence. More on that later.

3.2 Observer and Observable

An Observable is the source of the events, the publisher. The observer is the receiver, also called the subscriber. The observable yields zero or more `onNext` events, and optionally closes the stream with an `onCompleted` event or `onError` event. A stream of events is also called a sequence. An observer can connect to an observable by subscribing to it. The result is known as a subscription. The subscription manages the lifetime of this connection.

3.3 Hot vs Cold observables

Cold observables function like a DVD. In this metaphor, subscribing can be compared to placing a DVD in the player, which will then start playing after clicking on 'play'. Once the DVD is running, the DVD player "pushes" the video and it cannot be watched at a faster or slower speed. Of course, the player may be equipped with a remote control offering fast forward or rewind options but it is the operating system of the DVD player that decides if this option is available.

Hot observables can be compared to a radio broadcast. If you turn it on today, it will play a song but if you tune in tomorrow, it will play something else. Many people can listen to the broadcast simultaneously without any additional side-effects on the source.

In computer terms, a cold observable is like reading a file from the disk. A hot observable can be a mouse stream or a security camera feed.

Most observables are cold; they only become active when they are subscribed to. Only during a subscription they generate a sequence. They generate a new sequence for every new subscription and every sequence has its own execution context (but that sequence typically contains exactly the same events). After the subscription ends, they dispose their internal resources. A cold observable can be seen as a delayed execution: the programmer defines what events will be created, with what values and at what time, but these values will only actually be generated when an observer subscribes. There is a subtle difference with a normal closure that is run later: a Cold observable keeps running in the background once activated and can return values multiple times at any time (or none at all). Common examples are observables that start a timer, query an external resource or return a fixed list of items etcetera.

A hot observable shares side-effect between subscribers, there is only one execution context and the results are shared (multi cast) between subscribers. They typically describe live data. The hot observable multicasts the same source to every subscriber and produces values regardless of subscription. For example, mouse movements are a hot observable; there is only one mouse no matter how many times you listen to it. With the help of certain techniques a hot observable can be made cold. Previous events can be buffered for example and replayed to a later subscriber. Certain specialized multicast operators can make a cold observable hot to allow multiple subscriptions to the same source.

3.4 Marble diagrams

To visualize Rx streams, marble diagrams are often used. In a marble diagram, time flows from left to right and a horizontal line represents a stream, complemented with values for every element and two special characters for a complete or error message (usually a vertical line or a cross). Vertical or diagonal lines between streams can be used to indicate how events are transformed between streams. Colour markings are also used at times.

```
-o- Next
--| Completed
--X Error
```

As time flows from left to right in the marble diagram, vertically-aligned events happen simultaneously. A simple example with the `Select` function, also known as `Map`:

```
---1---3---4---2---8-- o1
   ↓   ↓   ↓   ↓   ↓
---2---6---8---4---16- .select(e => e * 2)
```

In this example, two streams can be distinguished. The top stream `o1` contains the elements 1,3,4,2 and 8; it is transformed into the bottom stream by having every element multiplied by 2, resulting in 2,6,8,4 and 16. Also see the marble diagram map (figure 4) on page 20.

3.5 Transformations on Event Streams

There are a number of different operators on Rx. Most commonly used are the transforming operators that transform one sequence to another. Next to transforming operators, creating operators are also required; they can independently create sequences. In addition to some utility methods, subscribe can also be used to activate a sequence and match an observer with an observable.

Transforming operators transform a sequence, resulting into a new sequence. There are methods that manipulate time (delay all events for 1 second, or collect all data for 1 second and group them), manipulate the stream (add one to every element, filter events larger than 10) or combine streams (take all elements from both streams).

A typical way to write programs in Rx is to chain multiple operators. The stream starts with a creating operator that creates an observer and every next operator transforms the source stream into another stream that will act as source for the next operator. Operators further on in the pipeline are sometimes referred to as the downstream operators. Eventually all events flow through every operator and into the attached observer.

3.6 Schedulers

All work in Rx is done on a scheduler, and the scheduler controls the concurrency behaviour of the system. Scheduler is an abstract interface and by selecting a particular

scheduler implementation, the concurrency behaviour of Rx can be changed. All operators that schedule work[3], take (optional) a scheduler as an argument, so the programmer has full control over how the work load will be scheduled.

The scheduler interface allows work to be scheduled in the future. Operators that need to work with timing inside the sequence, will use the available scheduler to schedule the work load. If an operator needs to know the current time, it must request the time from the scheduler interface. It is the responsibility of the scheduler to make sure that the requested work is executed in a timely manner. With these properties, the scheduler has full control over how and when work will be executed but the operator decides which work is executed. If there is too much work, the scheduler will queue up the work and schedule it when it has a thread available.

There are many possible schedulers and multiple options can be implemented. The five most commonly used schedulers that are readily available are:

a) The thread pool scheduler operates a thread pool and schedules all requested work on a pool of different threads;

b) The trampoline scheduler schedules all requested work on the current thread;

c) The new thread scheduler that uses a new thread each time instead;

d) The immediate scheduler that schedules all requested work directly on the threads that tries to schedule it;

e) A test scheduler that does not schedule work based on time like the others but leaves it to the creator of the test to advance time in the test scheduler, allowing writing tests that span days and that are executed in milliseconds.

In addition to these five default schedulers, it is also possible to add other custom schedulers. For example, an user interface might use a scheduler that schedules on the user interface threads.

If no scheduler is selected, Rx follows the strategy of minimal concurrency. Operators that do not need to introduce concurrency will not do so, unless explicitly requested by the programmer. Operators that work with timing, like `Delay`, `Sample` and `Throttle` [4] must however schedule their work on a scheduler.

3.7 Control process in Rx

Rx is designed to run thread-free, which means code runs independently of which thread it is called on. This is important for the schedulers (3.6) that can schedule work on any thread. The Rx interface contract dictates that all events should be delivered serialized (serialized means the next event will not be processed until the first one returns). This way, Rx does not need to do any housekeeping on potential locking and synchronization problems as usual with multithreaded code. Rx operators that

[3] As per Rx design guidelines [8, paragraph 6.9].

[4] These operators are explained in 3.11.

introduce concurrency, should handle this locking internally so downstream operators can assume that the sequence is run serialized. This allows Rx to process a message on one thread and hand it over to another thread at some point for the next step in the pipeline. This is allowed since the second thread processes a different set of operators in the pipeline.

Sometimes this is even required: when a message needs to be delayed for 10 seconds, it is not viable to let a thread wait 10 seconds before it continues because in that case the next message will also be delayed 10 seconds and Rx can only handle one message every 10 seconds. Instead, we want to shift the stream 10 seconds into the future but we cannot guarantee that the current thread will be free 10 seconds from now. Because of this, we schedule the work in the future. The Rx scheduler is then allowed to pick another available thread to continue. In order to process work parallel within the Rx contract, the work can be split to multiple observables that are processed in parallel. At the end the results are merged again.

As a result of the previous propeties, two things can happen when one observer is slow at processing messages. If concurrency was introduced and more work will continue to be scheduled quicker than it is processed, it will result in an endless queue. If no concurrency was introduced, Rx will delay new events until the last one was processed.

3.8 Interactive vs reactive

Imperative programs run at their own speed, they request (pull) the next bit of work when they are ready to process it. Reactive programs work at the speed at which they receive pushed data. Imperative programs can have more work available than they are processing. They will only pull the work when they are ready to execute it. Thereby they create so-called backpressure automatically. An imperative program can also have no new work available, and block. The process cannot execute any other work for the time that it is blocked. Reactive programs work in the opposite way: they get work pushed onto them and start running and processing this work. They work exactly as fast as work gets pushed. If no work gets pushed, they are available to perform other work.

However, they run into problems when more data is pushed than they can process as they cannot implicitly create backpressure. In this case, they get non-blocking backpressure where work keeps coming in uncontrolled.

3.9 Push-back and backpressure

As long as no concurrency is introduced into the pipeline, Rx will use the push-back (blocking) model. This means that the next element will be delayed until the current one is completely handled. When an operator introduces concurrency, Rx will use an unbounded queue to store all resulting elements until the receiving observer is ready. This is because of another important property of the Rx contract: all messages must be serialized [5], which means messages cannot be forwarded concurrently. So to avoid forwarding messages concurrently, they are stored. Rx assumes observers will be fast

[5] Rx design guideline [8, paragraph 4.2]

enough to keep up with observables. If, however, an observer misbehaves and is unable to process all values in time, the unbounded queue may slowly take up all memory. Another disadvantage is that as messages get delayed longer, the observer receives older messages. Therefore, a so-called backpressure strategy is required. This will be discussed in more detail in Chapter 5.

There are various ways to release non-blocking backpressure. Some Rx operators such as `Throttle` or `Sample` use lossy backpressure; they ignore values that are sent too quickly after each other. That does help with a structurally slow client, however, it does not necessarily guard against a stuck or otherwise misbehaving client.

In some cases it might not even be desirable to lose values; in that case, other solutions like push-back should be introduced again.

Another model (used in the Java variant of Rx (RxJava)) is to switch to a pull model when an observer runs behind. This has the advantage over the Sample operator that an Observer can dynamically change its own load[6].

3.10 Operators with unbounded Queue

Because of unbounded queues, all excessive values in the sequence will be stored for later use. They are processed once resources are available. Different operators use an unbound queue for different reasons. The most common one is the introduction of concurrency: if a value is handed off to the scheduler, it is pointless to keep the current thread busy until the next thread is done. Some operators use more complicated patterns. `Buffer` for example stores all values in a sequence for the pre-set buffer time, `Delay` stores all values received during the given delay. Some other operators also introduce unbounded queues even without concurrency. Zip takes two sequences and combines the values from both into one, if one of the two sequences produces significantly more values; these values will all be stored until the first sequence has caught up. Even operators like `GroupBy` and `Merge` use an unbounded queue under the hood when they split/merge multiple sequences.

3.11 Explanation and example of different operators in Rx

To understand the uses of Rx, some Rx operators and examples will be shown. The included marble diagrams should help to understand the operators. The operator naming from C# is used but a few aliases from other implementations of Rx are also added such as RxJs and RxJava.

3.11.1 `Select`

`Select`, better known as a map function, modifies an `Observable<A>` into `Observable<B>` given a function with the type `A->B`. For example, if we take the function $x => 10 * x$ and a list of 1,2,3. The result is 10,20,30, see figure 4. Note that this function did not change the type of the Observable but did change the values.

[6] See also [9]

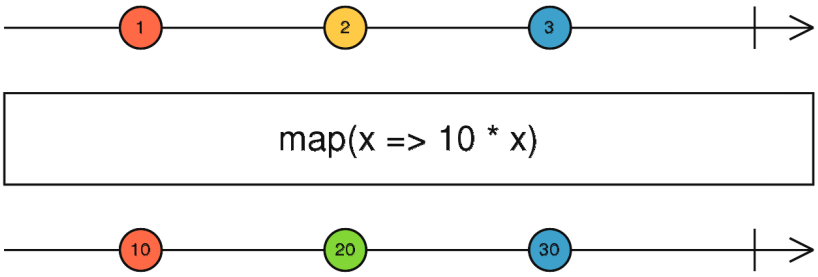

Figure 4: `Map` marble diagram, alias for `Select`
Source: reactivex.io

3.11.2 `SelectMany`

One of the most important operators in Rx is `SelectMany`, alias `flatMap` or `mergeMap`. `SelectMany` allows asynchronous queries, resulting in an observable of observables and it flattens the results, see figure 5. There may be multiple inner observables that run simultaneously, so the results from these inner observables may be intertwined.

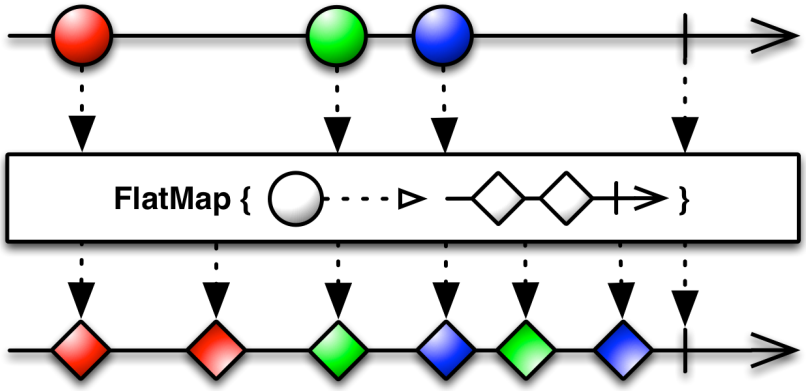

Figure 5: `FlatMap` marble diagram
Source: reactivex.io

3.11.3 `Where`

The `Where` operators, alias `filter`, filters elements based on a condition. See figure 6.

3.11.4 `Delay`

`Delay` will project the sequence unmodified, but shifted into the future with a specified delay.

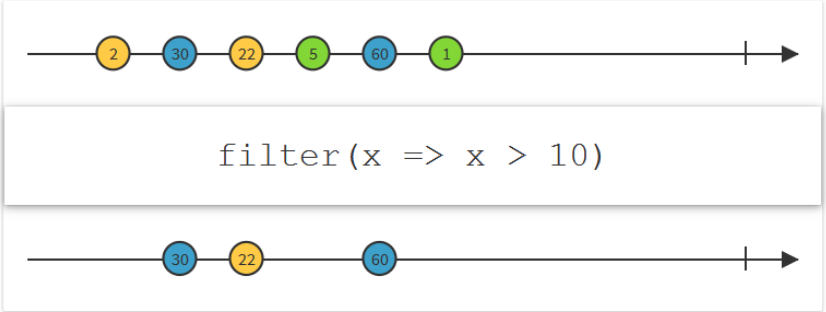

Figure 6: `Filter` marble diagram
Source: reactivex.io

3.11.5 `Merge` and `Concat`

`Merge` and `Concat` combine multiple sequences into one. `Merge` might interweave elements from different sequence whereas `Concat` emits all elements from the first sequence before turning to the next one.

3.11.6 `Buffer` and `Window`

`Buffer` and `Window` collect elements from the source sequence and emit them in groups. `Buffer` projects these elements onto arrays and emits those arrays when the buffer is closed. `Window` emits these elements in nested observables. It will emit a new inner observable when a window opens and will complete the inner observable when the window closes. Notice that there can be overlap between multiple buffers and windows if the next one opens before the last one closes. For example `Buffer` with a count of 2 and a skip of 1 will emit the last 2 elements (count 2) for every element (skip 1), so the sequence `-1-2-3-4-|` becomes `--[12][23][34][4]|`.

3.11.7 `GroupBy`

Similar to `Window`, `GroupBy` projects the sequence onto a number of inner observables but as opposite to `Window` where all windows receive the same sequence, `GroupBy` will emit elements only to one inner observable that is associated with the current element based on a key selector function. See figure 7.

3.11.8 `Amb`

The `Amb` operator (stands for ambiguous), alias `race`, subscribes to a number of observables and retrieves the first observable that yields a value, closing off all others. For example, `Amb` can automatically select the best server to download from: `Amb` listens to both servers and the first server that replies is used. See figure 8.

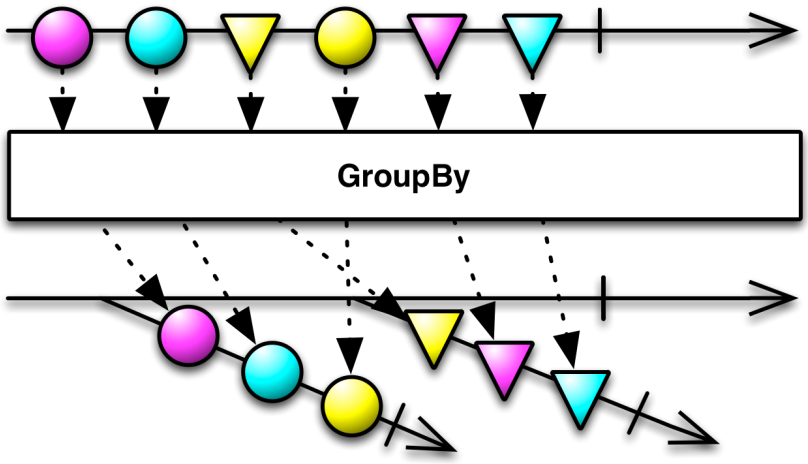

Figure 7: `GroupBy` marble diagram
Source: reactivex.io

3.11.9 `Debounce`, `Sample` and `Throttle`

The following three operators, `Debounce`, `Sample` and `Throttle` are used to rate-limit the sequence. They will filter out elements based on the timing. There is a difference in how they exactly do that. `Debounce` will delay a value when it arrives and only emits the last value in a burst of events after the set delay is over and no new event arrives during this delay. `Throttle` will emit the first event from a burst and will ignore all subsequent values that arrive during the set timeout. `Sample` will emit the latest value on a set interval or emit nothing if no new value arrived during the last interval.

3.11.10 `ObserveOn`

`ObserveOn` is Used to switch to a different scheduler. For example, to switch to the user interface scheduler (mandatory in many applications when mutating the interface).

3.12 Related work

There are various other systems that work with publisher/subscriber pattern and/or asynchronous events. It is interesting to look at these systems. There is background on other types of publisher/subscriber systems that aim at 'full decoupling in time, space, and synchronization between publishers and subscribers'[10]. Systems that deal with huge volumes of data are also interesting to examine. For example, Splice[11] is a system that works with radar data, producing high volume data streams that would quickly exceed most bandwidth.

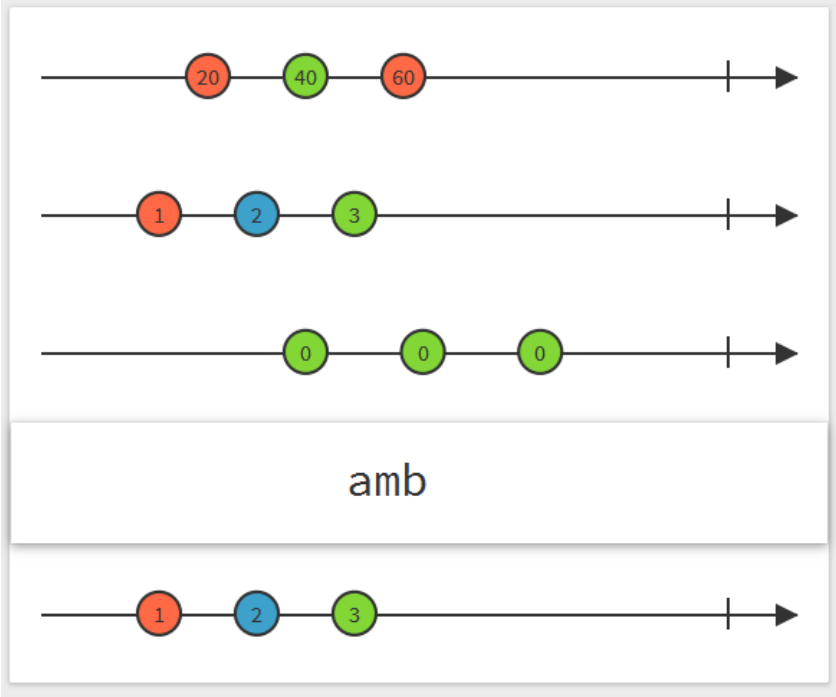

Figure 8: `Amb` marble diagram
Source: rxmarbles.com/#amb

3.12.1 TPL

Whereas Rx focuses on asynchronous data driven programming, its sibling the Task Parallel Library (TPL) [12] is used in a more imperative way. TPL is built around tasks that are small work units for a thread pool or asynchronous operations. TPL contains many functions similar to those in Rx and the signature of a Task is very similar to that of an Observable. A Task yields no result (incomplete) or a complete state with a value or error state with an error, while an observable can yield zero or more results, optionally followed by a complete or error state. The signature of Task can be considered to be a subset of that of Observable. Because of these similarities, it is very simple to convert a Task to an Observable (there is a standard function to do so).

Because both Rx and TPL focus on asynchronous programming, they supplement each other and it can be very useful to combine both paradigms.

3.12.2 iTasks

iTasks[13] is a system for task-oriented programming in pure functional language. A `Task` in iTasks is actually a unit of work with observable intermediate values. Through `Shared Data`, iTasks contains a publisher subscriber system that again can be compared to Rx observables. A `Task` can start with no value and can then produce an unstable value, i.e. a value that may still change or go back to no value. At one point, the latest value

might become stable after which it can no longer change. A `Data Share` guarantees all listeners that it broadcasts the latest known value. An Rx Observable can also update a value with every `onNext` but the latest value can also become final when `onComplete` is signalled. Going back to no value can be added by wrapping the data N in a Maybe<N> type.

4 The experiment

The experiment is based on the use case. The Aia Software use case is based on a real world scenario. In the experiment, a simulation of this scenario will be carried out.

To recap section 2.3: the Cloud Manager manages the jobs on the cloud and shares metadata about these jobs (when they start and finish) to the Failover. The Failover uses this metadata to track the health of the cloud. If the cloud behaves unreliably, the Failover can move all work to another cloud. This is an expensive and disturbing operation that will cancel and reschedule all outstanding work. Therefore, it is essential that the Failover receives all metadata reliably and without delay. Next to the Failover, it is also preferable that this metadata is shared with other monitors.

These other monitors are only used for observations and are as such not essential. Monitoring a system in this manner will cause disturbance, and it is essential that the Failover continues to work reliably while the monitors may not function reliably themselves.

Is it possible to build a system that can monitor an event stream with multiple receivers without a number of nonessential receivers disturbing the essential receivers?

In the experiment, we built a simulation of this problem. We built a server that generates an event stream of metadata that is sent to a client program that connects multiple times, with different behaviour properties. The server and the client log all relevant data of what is happening. By changing the sharing strategy in the server, we can test different solutions to the problem and compare the results.

4.1 Important properties

Subscriptions of nonessential observers may not have noticeable side-effects on essential subscribers.

- Q1: Is the solution suited?
  - Essential subscribers do not skip any messages?
  - Are essential subscribers not delayed by nonessential subscribers?
  - Are messages to essential subscribers not delayed? (Over a threshold?)
  - The system does not run out of resources? (CPU, memory)

  If the solution is suited, then:

- Q2: How well does the system perform for nonessential subscribers?
  - Well-behaving subscribers do not miss any messages?
  - Do well-behaving subscribers get any delay? (How Much? Over a threshold?)

  If the system performs satisfactorily:

- Q3: How does the system perform under high load?
  - Are messages delayed? How long?

- Is message generation delayed?
- What is the message throughput per second?

4.2 Test architecture

The test environment is loosely based on the setup of the use case with a generator that generates job messages, simulating the messages generated by the Cloud Manager. There are essential clients that simulate the Failover and nonessential clients that simulate the monitors. These clients can be configured to simulate different behaviour, depending on what scenario we want to test. For example a scenario can exists of one essential client that processes events fast and one nonessential client that processes them really slow.

4.2.1 Design

The test environment consists of two programs. One is the server that generates all messages, accepts connections from different clients (all simulated by the client program), and measures, e.g. how fast these messages are processed and how much memory is used. See figure 9. The server offers multiple implemented solution strategies on different ports. The results depend on the activated solution: if a blocking solution is used, the generator will be delayed and push-back will be observed. If a solution keeps all messages in a backlog, memory consumption will increase. If a solution drops messages, the generator may run at full speed.

The other program, i.e. the client program, simulates all clients and measures their performance, namely the number of messages received; the number of messaged missed; the delay between the generation of the messages on the server and the reception; how long the client is waiting for the server and how long the server made the client wait.

The test software requirements list can be found in the attachments.

4.2.2 Concepts

The server- and client-side of the test environment each measure their own local effects. To evaluate the solutions, it is required to aggregate different results to get meaningful data.

The server measures the push-back: the time between when the data should have been generated and when it was actually generated. The client observes time shift: the time between when the data was generated and when it was received (and processed) by the client. If we add both intervals, we know the total delay that was observed.

Clients also measure drop rate and message count. These two statistics are only useful when combined. Drop rate indicates that messages are missed but it is also possible that messages are not missed but that the client is behind. All messages have an index number and when the client receives new messages with a higher index number, it can conclude that messages where missed and not delayed. However, if no new messages are received at all, it cannot make this distinction. Therefore, it is also important to compare

how many messages should be received. The longer the delay, the more messages the client is behind.

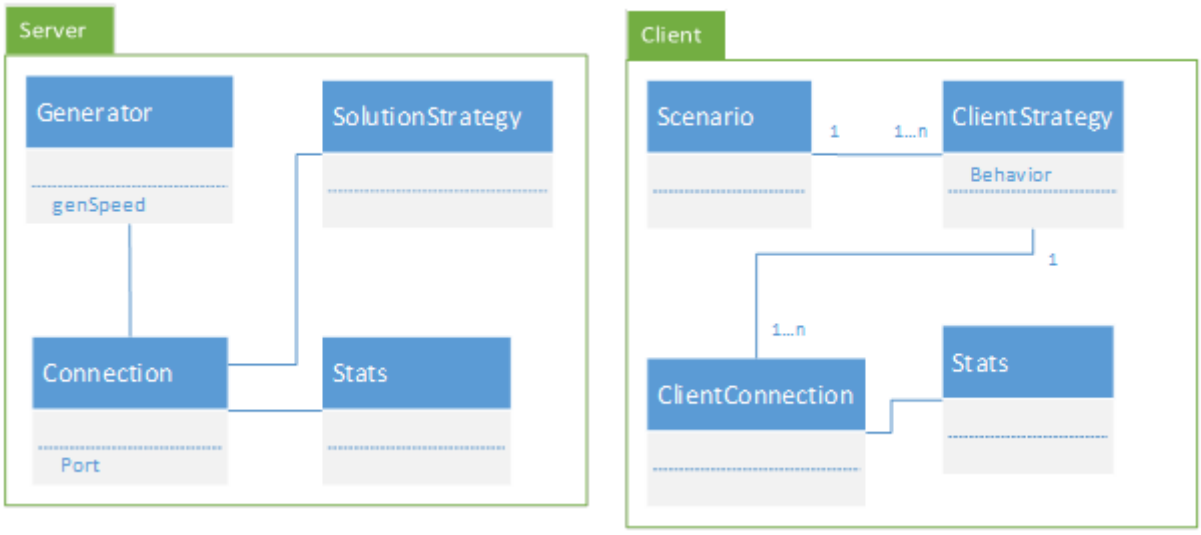

Figure 9: UML Class diagram of test program

4.3 Measurements

We measure the time that clients are inactive (sleep time); the time between creating and receiving messages (time shift); the number of messages that are lost (drop rate); the number of messages that are received (message count); the memory usage on the server and the delay in generating messages (pushback).

The sleep time is especially interesting in order to assert whether clients are fast enough. As long as they have sleep time (i.e.: they wait for the server), we know that no holdup is generated by the client. We expect the well-behaving clients to have sleep time. Measuring sleep time on the slow and malicious clients is meaningless because they intentionally stay busy and make the server wait for them.

We also expect well-behaving clients to have no message drop (especially the one marked essential). The slow and malicious clients should eventually get messages drop if the solution allows it.

The message count will show how fast the clients operate.

Because we know how fast the server should be generating messages, the message count can be compared to the expected value. By adding up the pushback, time shift (both multiplied by the generation speed), drop rate and the message count, we should get a value close to the expected number of generated messages.

4.4 Scenario setup

Scenarios are pre-defined sets of clients with different behaviour patterns. To test different behaviour against the solutions, a couple of variations of scenarios were used. Eventually, one scenario setup was chosen that exposed a broad selection of problems in the solutions.

The final scenario has a fast essential subscriber and a number of nonessential subscribers that behave in different ways, as described below:

- One nonessential subscriber with no delay (processes all events as fast as possible).
- A subscriber with a 10ms delay (processes 1 event every 10ms).
- A subscriber with a 10ms delay that disconnects and reconnects every second.
- A subscriber with a 20ms delay that crashes every 2 seconds but keeps the old connection open while also starting a new connection (that will again crash after 2 seconds).
- A subscriber with no delay, pausing every 1 second and resuming 1 second later.

The purpose of these nonessential subscribers is to put stress on the server to determine if the essential subscriber is affected.

We ran this scenario against different solutions, each time picking a solution for the essential subscribers and one for the nonessential subscribers.

It is also interesting to observe how various nonessential clients perform. The client with no delay will hopefully receive data at an equally reliable rate as the essential subscriber, while the clients that pause, will end up losing messages (depending on the buffer strategy of the solution). The goal of the disconnecting client is to put stress on the server, while the crashing client continues to consume additional connections.

4.5 Test data

The test data are generated at a speed of ten items per ms, inflated to 5-6kb per package. This means that as long as the clients are able to process ten items per millisecond, they should not fall behind.

Initially a lower generation speed (1 message per 10ms) was used but this produced less interesting results because almost all solutions were able to keep up with this speed. By increasing the speed, even the most optimal solution was put under stress.

4.5.1 Data size

Even if no buffer strategy was used in the solution, the network connection already allowed a bit of data to be buffered. In an initial setup, the test data was only 10-12 bytes large. With buffer of several hundred kilobytes, this allows thousands of items to be buffered in the network layer. Therefore it was necessary to inflate the test data with an extra field that contains 5-6Kb of random data.

5 Possible solutions

A number of operators in Rx.net in C# were examined that offer solutions to handle high load situations. In addition, solutions in other Rx implementations and other related systems were studied that avoid backpressure in various ways.

5.1 The problem

Before trying to solve backpressure, it is paramount to understand what happens when backpressure arises. It is also essential to gain insight into the reasons why this is problematic (or not).

There are two mechanisms in Rx that come into play with backpressure is observed. The first is push-back. When no scheduling is introduced, the Observable and the Observer share the same thread and a slow Observer will delay the Observable. When scheduling is introduced, Rx introduces a queue where one thread schedules new work and another picks it up. This will be an unbound (endless) queue as the library always assumes messages are processed faster than they are produced. The programmer is responsible to make sure that this holds true. If the work is not processed fast enough, the endless queue consumes all available memory and causes the program to crash.

Also, various operators are able to ignore elements based on the timing. `Sample` only takes one element every time span. `Debounce` ignores elements received in rapid succession. `Windows` and `Buffer` collects multiple elements and process them as a single item.

5.2 Solutions in RxJava

Another source of inspiration is RxJava, the Java implementation of Rx. RxJava solves the backpressure problem by introducing the Producer interface that pulls data, automatically switches to pull mode when needed and back to push mode when the observable has caught up. The producer interface adds a $onBackpressure(Strategy)$ option to specify a strategy that should be used when the buffer overflows.

Where Rx.Net uses unbound buffers, in RxJava the backpressure strategy specifies a maximum buffer size. When the buffer size is reached, the observable switches to pull based mode. Then all events are cached, up to the buffer size. While in pull mode, the producer can request events when it is ready to process them. One buffer strategy is to drop messages when they exceed the buffer size. Another is to terminate the subscription in an error. A third option is to block the source observable. All these solutions require that the source observable must support backpressure.

Because of the producer interface, RxJava is capable of signaling the Observable. It can send a pause signal back to the Observable. In Rx.Net this is not possible because there is no way to signal back at the Observable unless the Observable interface is overhauled. This would require the entire library to be updated.

5.3 Solutions requirements

We have implemented Various solutions in C#. Not all solutions are suited for both the essential and nonessential subscribers. A solution for the essential subscriber may not lose messages. The nonessential subscribers need to run against strategies that drop messages before resources are exhausted because push-back is not allowed.

We assume that an essential subscriber processes messages faster than the generator produces them. If it does not, it would be preferable if the solution slows down the generator (push-back) in order to match the speed of the essential subscription. Failing to do so will result in an out of memory exception. While push-back is an optional requirement, it may still be interesting to try different solutions with the essential subscriber to see how they perform in combination with strategies for nonessential subscribers.

5.4 Solutions implemented in experiment

Nothing connects the observers directly to the source and follows standard Rx behaviour. It is also known as the null-solution. Because observers are connected directly to the source, they substantially influence production. All elements are processed single threaded and a slowdown in one message immediately results in a holdup for the entire sequence. That makes this solution suited for essential observers. Even when they are slower, the generator will slow down (push-back).

WithObserveOn connects the observers through a thread pool (or other scheduler specified in the `ObserveOn` operator). As long as the scheduler introduces concurrency, observers will not interfere with each other. However, the system no longer has control over the generator and uses an unbounded queue internally. Potentially consuming all available memory. This solution is suited for the essential observers as long as the essential observer is fast enough.

LockQueue works also on a thread pool but locks the generator when there are too many elements in the queue. This spreads the processing time out over a longer period, without locking the generator for every small hiccup.

WithSample uses the (in Rx) recommended operator `Sample` to prevent backpressure. `Sample` only processes the latest element every time the sampler ticks (the sampler could be a set interval). Because the `Sample` operator only holds at most one element, it will ignore the elements arriving faster than the sampler interval and also ignore the elements arriving faster than the processing speed of the observer. By setting the time span for the sampler to zero, all elements will be processed as long as the observer is fast enough.

Interestingly enough, `Sample` does have push-back. When an element is processed, this also ties up the current thread that may have been used by the generator as well. However, `Sample` combines very well with the `ObserveOn` operator before it in the queue, while `ObserveOn` introduces threading, WithSample will stop the unbound queue in `ObserveOn`

from growing indefinitely. WithSample is not suited for essential observers because it loses elements.

The following example of `Sample` shows that element 2 is ignored because the sampler did not tick. `*` is used in the marble diagram to indicate the observer was busy.

```
-1---2----3-----|
---s---------s--|
   ↓          ↓
---1*--------3***|
```

In the next example, interesting behaviour can be observed when the sampler does tick multiple times but the observer is not ready because it is busy.

```
-4--5--7--3----|
---s--s--s--s--|
   ↓  \-↓   \-↓
---4****5*****3***|
```

Element 7 is ignored because the observer was not ready.

DropQueue takes elements and queues them on another scheduler. When there are too many elements in the queue, the oldest element is skipped. The dropQueue can also skip elements when they are older than a certain interval.

DropQueue is not suited for essential observers because it can skip messages. However, it is especially designed to serve nonessential observers.

In earlier attempts to implement DropQueue, the newest element was skipped. This resulted in a problem when the DropQueue was used multiple times for different observers; every DropQueue ended up keeping a unique list of elements when observers hang at different moments in time, adding up to high memory pressure. By keeping the latest element (and skipping the oldest), all DropQueue instances keep references to only the most recent elements, allowing the program to garbage collect older elements.

5.5 Usefulness of solutions

As is clear from table 1, not all solutions have the required properties for the essential subscribers. Essential subscribers cannot lose elements but can (and probably should) have push-back. Nonessential subscribers probably should have message drop but cannot have push-back. This eliminates the solution Nothing, LockQueue and WithSample for nonessential subscribers.

WithSample with threading is also not viable. While it has no push-back for new elements, it drops messages on processing thread so new messages pile up if the processing thread is tied up in a long processing job.

Solution	Push-back	Loses elements
Nothing	yes	no
WithObserveOn	no	no
LockQueue	yes	no
WithSample (no threading)	yes	yes
WithSample	no	yes
DropQueue	no	yes

Table 1: Overview of solutions

6 Results

Various solutions were run against a number of scenarios. This resulted in data on how much data throughput was available in different solutions, how much memory was used and to what extent the various clients interfered with each other.

Figures 10 to 16 show the six subscribers described in the scenario in section 4.4. The first part of the name indicates their behavior:

- Imp for the fast essential subscriber;
- Norm for nonessential subscribers;
- Disc for the subscriber that disconnects each time and
- Crash for the subscriber that cashes each time repeatedly.

The second part indicates the delay for each event, for example 10ms if it processes 1 event every 10ms or 0ms if it has no delay. Subscribers with repeating behaviour (e.g. a disconnecting or crashing subscriber) indicate how often this is the case at the end of their names. For example, the subscriber that crashes every 2 seconds and processes a message every 20ms, is named Crash20ms2s.

Figures 16, 20, 24 and 28 show the number of processed messages by the essential subscriber per solution.

Figures 17, 21, 25 and 29 show the number of processed messages by the nonessential subscriber per solution.

Figures 18, 22, 26 and 30 show the message delay for essential and nonessential subscribers per solution.

Figures 19, 23, 27 and 31 show the memory usage on the server per solution.

To see how essential subscribers are affected by nonessential subscribers under different solutions, multiple runs are compared. Each time the essential subscriber is hooked to the null solution. The null solution has no buffering or smoothing, so any hiccup from the nonessential subscribers has a maximal effect on the output of the essential one. The setup described in section 4.4 is used. The nonessential subscribers are all hooked to the same solution. By changing the solution for the nonessential subscribers; it is possible to compare the way in which the different solutions perform.

The following sections show the results from the test runs and answer how well the various solutions performed based on the questions from section 4.1.

6.1 Solution: Nothing

When not using any solution, data is transmitted on the same thread that generates the events. Since the generator used in the experiment does not introduce concurrency itself and uses one thread to announce event to all observers, pushback is automatically introduced.

It is expected that even a small delay in sending messages will also delay the generator and all other observables.

Figure 10: Null solution.

In figure 10 we connected both the essential and nonessential subscribers to the null solution (Nothing). The process hangs after 189 messages as the nonessential crashing client is no longer processing messages and all other clients wait for the now crashed client.

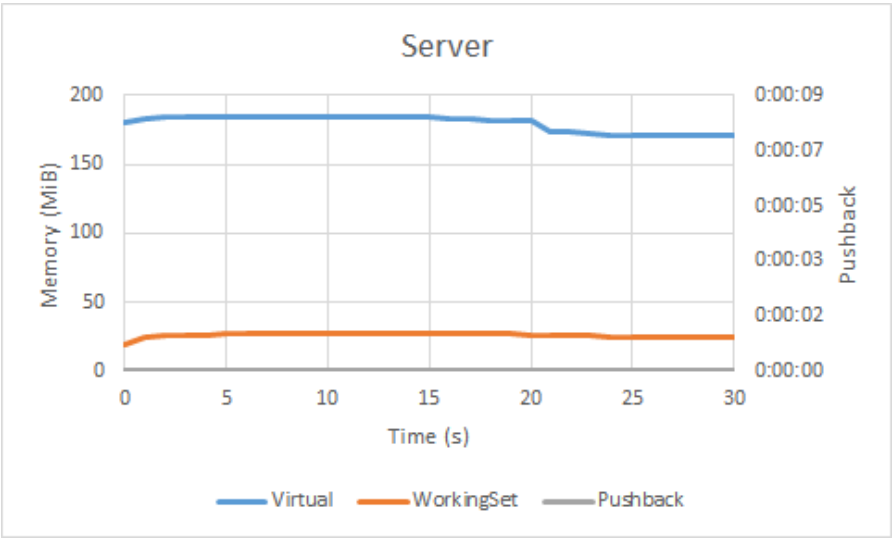

Figure 11: Null solution.

In figure 11, memory usage is stable and no pushback is observed. The graph ends after 30 seconds. This is because the server hangs at this point.

It is observed that even while there is no buffering implemented, there is some smoothing going on. Closer examination shows that this take place in the network layer, see also 4.5.1.

- Q1: Is the solutions suited?

– No, essential subscribers hang after 189 messages and 3 seconds. See figure 16.

6.2 Solution: WithObserveOn

Because this solution queues all work on an endless queue, it is not suited for nonessential subscribers. As is clear from figure 13, memory usage on the server increases linearly as all old elements are kept alive. The graph ends after 30 seconds because the server has crashed.

For essential subscribers, the assumption is that they will be fast enough to process all elements so it would be acceptable to store some elements for a short period of time. Still, it is possible that the server runs out of memory if the essential subscriber does not keep up.

Figure 12 shows the results of a run with ObserveOn for nonessential subscribers and nothing for essential subscribers. Because the server crashed after 30 seconds, the clients no longer receive data. Only the Norm10ms client continues to process some data but this probably concerns messages that are still remaining in a buffer somewhere.

Figure 12: WithObserveOn for nonessential subscriber; Null solution for essential subscriber.
The same graph is displayed twice with different y-axes because three clients process significantly more messages. These are visible in the top graph and quickly run out of the bottom graph.

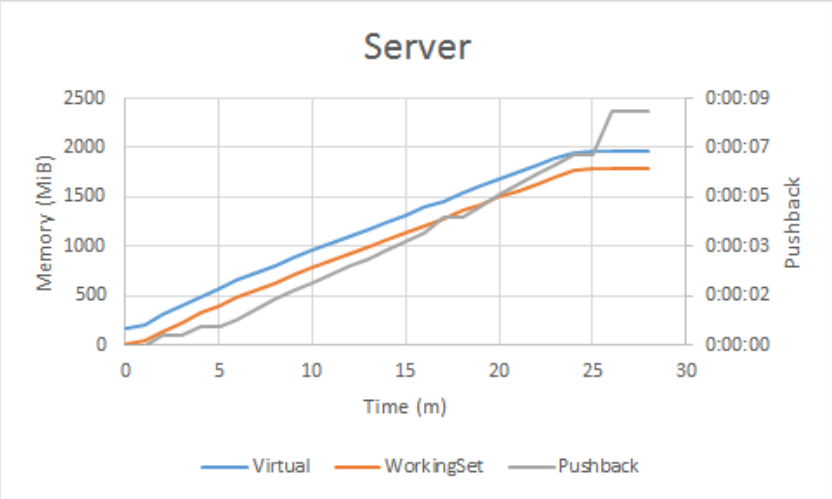

Figure 13: ObserveOn for nonessential subscriber; Null solution for essential subscriber.

- Q1: Is the solutions suited?
  - No, the system runs out of resources (memory). In Figure 23 memory goes up to 1972 MiB before the server crashes with an out of memory error. Figure 20 also shows that the essential subscriber only receives 171742 messages in 31 seconds before messages stop.

6.3 Solution: LockQueue

This solution is very similar to the previous one but has an upper limit in elements that can be queued. As long as the queue size is large enough, this removes hiccups from the essential subscribers but it is still unsuited for nonessential subscribers as these will lock up the system at one point.

- Q1: Is the solutions suited?
  - No, because nonessential subscribers can still hold up the essential one. This solution would actually be suited for essential subscribers to add some smoothing to the system.

6.4 Solution: WithSample

WithSample is only interesting when combined with threading. Without threading, it cannot be used for nonessential subscribers because it has pushback but neither can it be used for essential subscribers because it loses messages.

When combined with threading, WithSample works well in communicating with observers that are too slow. However, WithSample surprisingly still failed on the crashing observable because it only processes new elements once the previous one is finished. The crashing observable simply stops processing messages so the queue grows

forever. A fix for this problem is to add a timeout for the observables; if no new messages are accepted for some time on the connection, the observable disconnect. This does mean that all messages produced during one timeout period are buffered in the memory, while WithSample will only send the newest element if it continues and, therefore, will only need to buffer the newest element.

Another disadvantage of WithSample is that it is prone to losing messages because it skips to the newest element each time.

- Q1: Is the solutions suited?
  - Yes, it is suited. But with some remarks. The memory usage is rather bumpy, see figure 27. It is also clear from figure 24 that the essential observable starts off well but then slows down considerably as the system waits for hanging subscribers to be disconnected and for the thread pool to be extended.
- Q2: How well does the system perform for nonessential subscribers?
  - As seen in figure 25, the nonessential observer receives even less messages than the essential one and 76% of the messages is dropped. The essential observer receives 101,887 messages and the nonessential one 25,680.
- Q3: How does the system perform under high load?
  - The 101k received messages in figure 24 are far less than the 15m that could have been generated. (i.e. 10 messages per ms[7] means 10,000 per second means 15 million per 1500 seconds)
  - As is clear from figure 26, there is considerable message delay, increasing with big bumps of 300 seconds every 5 minutes. It is not surprising that the message delay is almost equal to the run time because the client only processes a fraction of the messages that it could have.

6.5 Solution: DropQueue

The DropQueue addresses some of the problems of WithSample. Mainly it dequeues excess messages on the producing thread instead of the processing thread causing the queue to grow up to the max queue size at most. It also allows smoothing the output a little by allowing an actual buffer size. A small hiccup in the observer will not immediately result in message drop but when an observable lags too far behind, it does skip messages. It is also important not to set the buffer size too high as then it would send out old messages.

In figure 14 and 15, we see the program runs smoothly with the DropQueue solution. There is some pushback because the essential observer runs on the null solution and easily adds pushback. When the essential subscriber disconnects at second 600, memory usage becomes a little but more erratic because the generator is now generating messages at full speed. Because the drop queue loses excessive messages at the same speed, memory

[7] section 4.5

usage does not go out of control and the program did not crash as it did with previous solutions.

- Q1: Is the solutions suited?
  - Yes. The essential subscriber receives a large number of messages, Figure 28 and the memory usage stays stable, see figure 31
- Q2: How well does the system perform for nonessential subscribers?
  - It shows good performance for well-behaving subscribers. See figure 29, the nonessential subscriber receives the same number of messages as the essential one.
- Q3: How does the system perform under high load?
  - About 2.53 million messages are delivered in 609 seconds (10,000 generated messages per second[8] would result in 6.09 million messages in 609 seconds). This means that 41% of the messages is delivered (2.53 million out of 6.09 million). Message delay (Figure 30) is 347 seconds out of 609, which is entirely due to pushback

[8] section 4.5

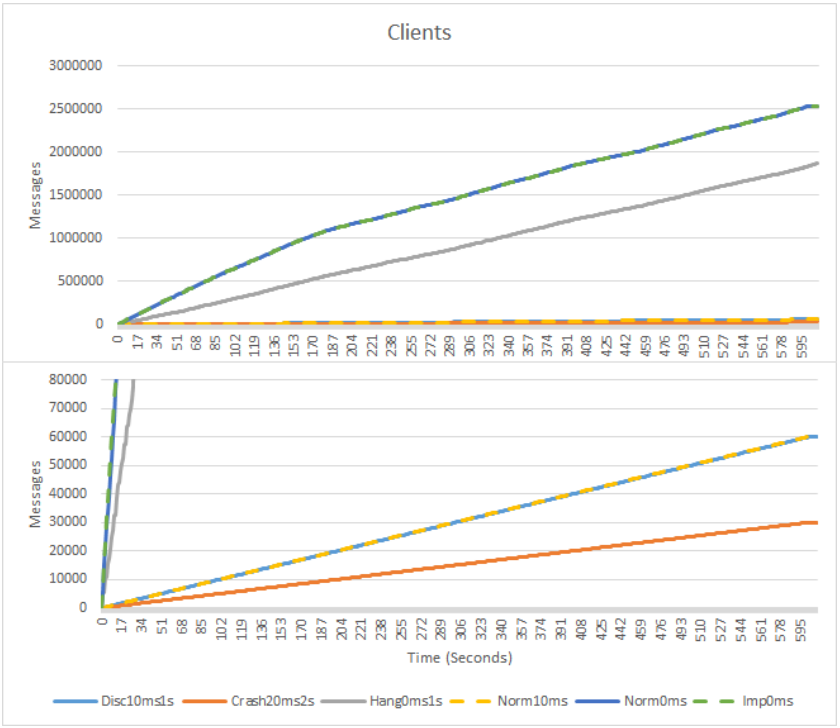

Figure 14: DropQueue for nonessential subscribers, Null solution for essential subscriber. The same graph is displayed twice with different y-axes because three clients process significant more messages. These are visible in the top graph and quickly run out of the bottom graph.

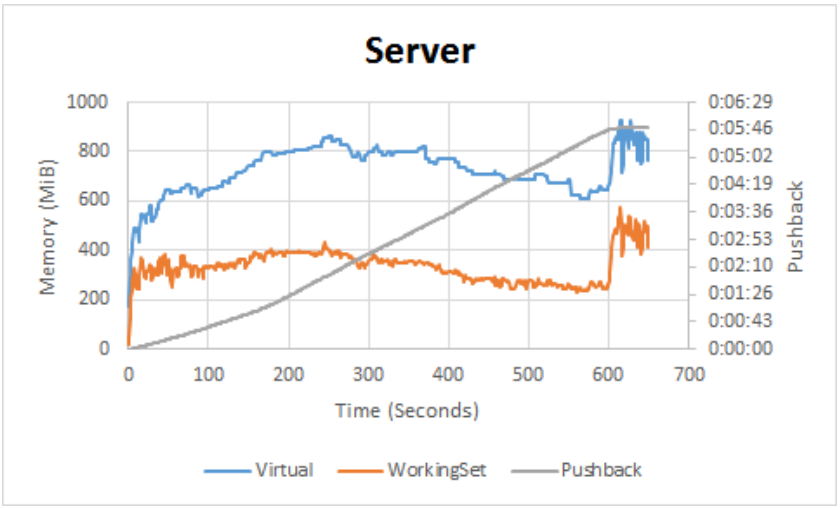

Figure 15: DropQueue for nonessential subscribers, Null solution for essential subscriber.

7 Conclusions

Is it is possible to monitor event streams? By monitoring an event stream, it is also affected. At the very least it may slow down. In our scenario, the event monitor could not keep up with the speed of the stream. At the same time we did not control our monitors. Therefore, it was required to implement a strategy to multicast the event stream without affecting it.

We selected different strategies and compared them in a simulation. Because the stream was multicast, different strategies could be used for every receiver. Receivers were split into two groups, essential and nonessential.

The nonessential group (i.e. the newly added (unreliable) monitors) should not affect the original source or receiver. There will always be some effect on the stream but we compared strategies on how severe they affected the stream.

The essential group (i.e. the original receiver) should receive the original stream unaffected but should also continue to affect the source. It is important that the original receiver can slow down (pushback) the source; in case the source produces events faster than the receiver can handle, all messages have to be stored. Eventually memory will run out and the delay between producing and processing messages may become too large even earlier. This property is only of secondary importance because we assume that our reliable receiver is fast enough.

The most naive strategy is to do nothing (null solution). This means that the receiver is directly connected to the source. This also implies that any delay in the receiver will delay the source. This strategy gave good results for the original receiver but failed for the unreliable receivers that forwarded their unreliable behaviour.

The second strategy was to copy and buffer the entire stream. This works for the original receiver and the monitors. However, it has the disadvantage that as soon as the receivers are processing messages slower than they are generated by the source, the receivers will receive older and older messages because the buffer grows. Eventually the system runs out of memory.

Next we wrote the LockQueue that buffers only a limited number of elements from the stream. Once the buffer is full, the source is delayed. The buffer gave a smoothing effect to the original receiver, while still allowing it to delay the source. This slightly decreases the total pushback when the original receiver has a hiccup. This solution was not acceptable for the nonessential subscribers because they forward their unreliable behaviour just like the null-solution.

Our fourth option was WithSample, an operator natively available in Rx. In theory, it is a good operator but it is not suited because of its implementation. It became apparent that WithSample uses the processing thread to process messages. Because this is the thread that locks up when receivers are too slow, it ended up using a lot of memory and only forwards a small percentage of the messages under high load.

The last strategy was a technique based on a combination of WithSample and LockQueue. A number of messages are kept in memory to smooth out short hiccups. If the queue size is exceeded, the receiving thread is used to dequeue (drop) excessive messages so that the number of stored messages never exceeds the queue size. This

solution worked quite well and was able to process 41% of the messages under high load.

References

[1] Erik Meijer. Your mouse is a database. *Queue*, 10(3):20, 2012.

[2] `http://reactivex.io/`, 2016. [Online; accessed September 2016].

[3] Linq (language-integrated query). `https://msdn.microsoft.com/en-us/library/bb397926.aspx`, 2016. [Online; accessed September 2016].

[4] Erik Meijer, Microsoft Corporation, emeijer@microsoft.com. Subject/Observer is Dual to Iterator. `csl.stanford.edu/~christos/pldi2010.fit/meijer.duality.pdf`, 2010.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[6] Continuation-passing style. `http://tunes.org/wiki/continuation-passing_20style.html`, 2016. [Online; accessed September 2016].

[7] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 51–64. ACM, 2002.

[8] `https://blogs.msdn.microsoft.com/rxteam/2010/10/28/rx-design-guidelines/`, 2016. [Online; accessed September 2016].

[9] Reactive streams with rx at javaone 2014. `https://speakerdeck.com/benjchristensen/reactive-streams-with-rx-at-javaone-2014?slide=67`, 2016. [Online; accessed October 2016].

[10] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[11] Prism Technologies. OpenSplice Data Distribution Service. www.prismtechnologies.com/, 2006.

[12] Task parallel library (tpl). `https://msdn.microsoft.com/en-us/library/dd460717(v=vs.110).aspx`, 2016. [Online; accessed October 2016].

[13] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. itasks: Executable specifications of interactive work flow systems for the web. *SIGPLAN Not.*, 42(9):141–152, October 2007.

Attachments

Requirements

The test environment simulates the use case. The test environment has the following requirements:

- Server
  - Generates example data from the CM.
  - Takes inbound connections.
  - Offers different solution.
  - Accept multiple connections to the same solution.
  - Logs the result:
    - How much memory used.
    - How many messages processed.
    - How much delay (push-back) while sending.
    - Etc.
- Scenarios
  - Multiple scenarios are supported.
  - Contains multiple clients with unique behaviour.
- Clients
  - Can show different (configurable) behaviours.
    - Connect as essential or nonessential client.
    - Request a specific solution form the server.
    - Process messages at a certain speed.
    - Can change speed.
    - Can simulate lock up (they wait).
    - Can disconnect.
    - Can repeat a list of actions.
  - Logs the results:
    - Wait time on server.
    - Time in wait state.
    - Number of messages processed.
    - Number of messages missed.
    - Delay between message generated and received.
    - Etc.

Graphs

The result graphs.

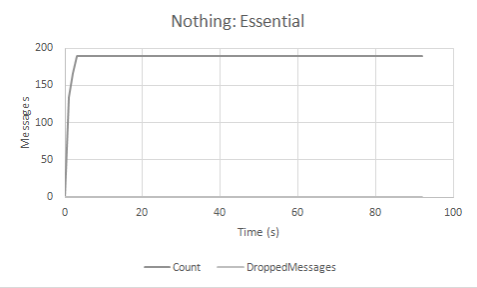

Figure 16: Nothing, essential subscribers.

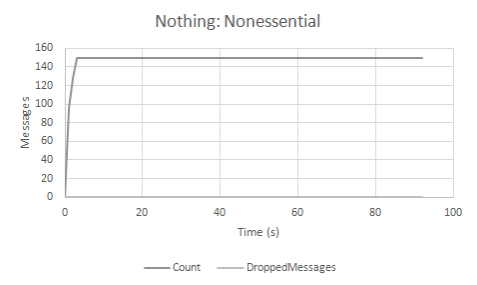

Figure 17: Nothing, nonessential subscribers.

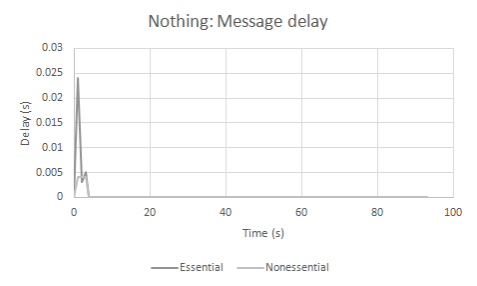

Figure 18: Nothing, message delay.

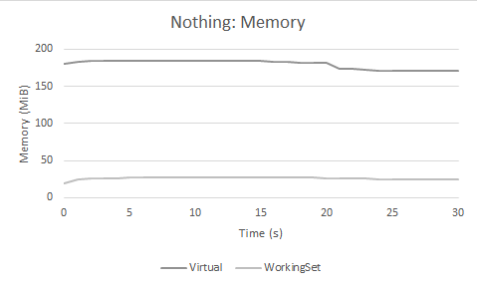

Figure 19: Nothing, server memory.

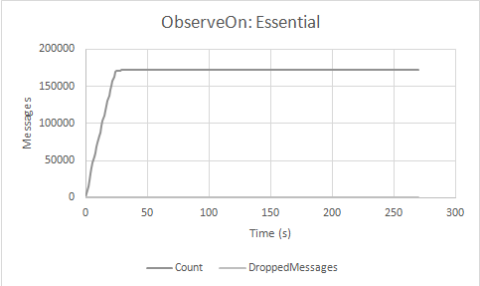

Figure 20: WithObserveOn, essential subscribers.

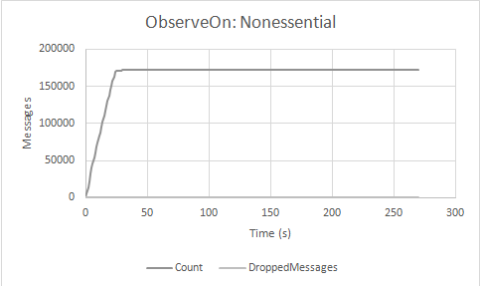

Figure 21: WithObserveOn, nonessential subscribers.

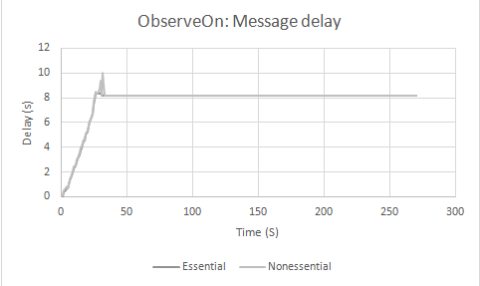

Figure 22: WithObserveOn, message delay.

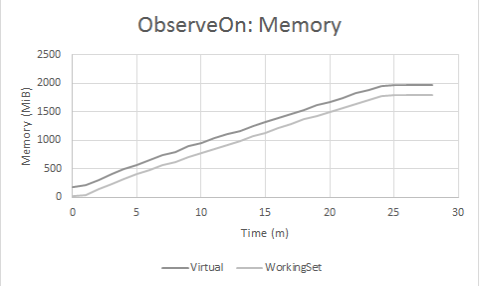

Figure 23: WithObserveOn, server memory.

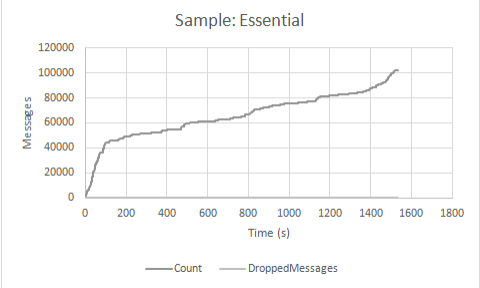

Figure 24: WithSample, essential subscribers.

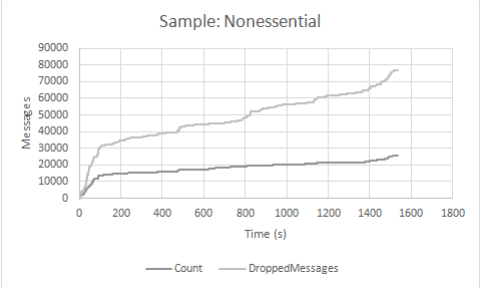

Figure 25: WithSample, nonessential subscribers.

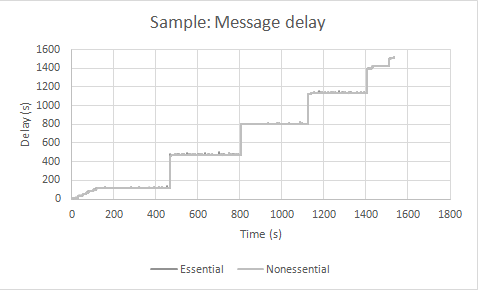

Figure 26: WithSample, message delay.

Figure 27: WithSample, server memory.

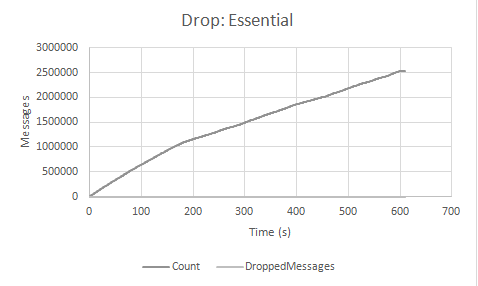

Figure 28: DropQueue, essential subscribers.

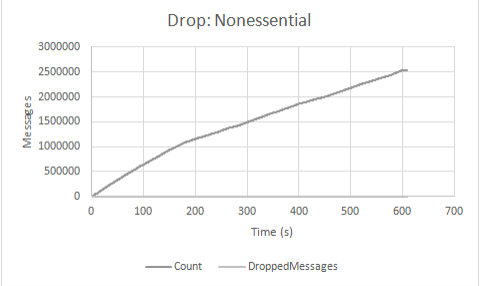

Figure 29: DropQueue, nonessential subscribers.

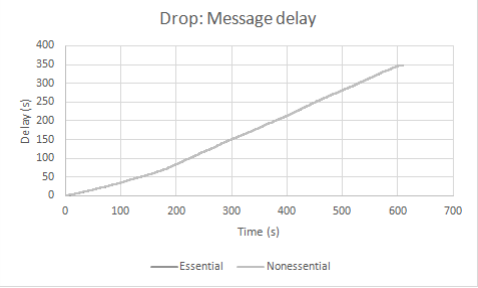

Figure 30: DropQueue, message delay.

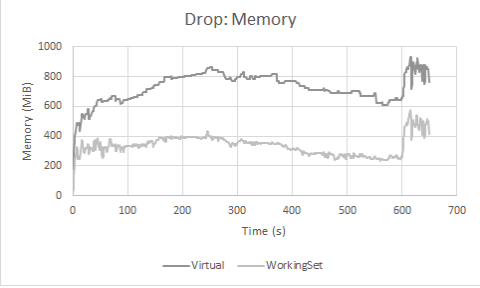

Figure 31: DropQueue, server memory.