Introduction

In RxJava it is not difficult to get into a situation in which an Observable is emitting items more rapidly than an operator or subscriber can consume them. This presents the problem of what to do with such a growing backlog of unconsumed items.

For example, imagine using the `zip` operator to zip together two infinite Observables, one of which emits items twice as frequently as the other. A naive implementation of the `zip` operator would have to maintain an ever-expanding buffer of items emitted by the faster Observable to eventually combine with items emitted by the slower one. This could cause RxJava to seize an unwieldy amount of system resources.

There are a variety of strategies with which you can exercise flow control and backpressure in RxJava in order to alleviate the problems caused when a quickly-producing Observable meets a slow-consuming observer. This page explains some of these strategies, and also shows you how you can design your own Observables and Observable operators to respect requests for flow control.

Hot and cold Observables, and multicasted Observables

A *cold* Observable emits a particular sequence of items, but can begin emitting this sequence when its Observer finds it to be convenient, and at whatever rate the Observer desires, without disrupting the integrity of the sequence. For example if you convert a static Iterable into an Observable, that Observable will emit the same sequence of items no matter when it is later subscribed to or how frequently those items are observed. Examples of items emitted by a cold Observable might include the results of a database query, file retrieval, or web request.

A *hot* Observable begins generating items to emit immediately when it is created. Subscribers typically begin observing the sequence of items emitted by a hot Observable from somewhere in the middle of the sequence, beginning with the first item emitted by the Observable subsequent to the establishment of the subscription. Such an Observable emits items at its own pace, and it is up to its observers to keep up. Examples of items emitted by a hot Observable might include mouse & keyboard events, system events, or stock prices.

When a cold Observable is *multicast* (when it is converted into a `ConnectableObservable` and its `connect()` method is called), it effectively becomes *hot* and for the purposes of backpressure and flow-control it should be treated as a hot Observable.

Cold Observables are ideal for the reactive pull model of backpressure described below. Hot Observables typically do not cope well with a reactive pull model, and are better candidates for some of the other flow control strategies discussed on this page, such as the use of the `onBackpressureBuffer` or `onBackpressureDrop` operators, throttling, buffers, or windows.

Useful operators that avoid the need for backpressure

Your first line of defense against the problems of over-producing Observables is to use some of the ordinary set of Observable operators to reduce the number of emitted items to a more manageable number. The examples in this section will show how you might use such operators to handle a bursty Observable like the one illustrated in the following marble diagram:

By fine-tuning the parameters to these operators you can ensure that a slow-consuming observer is not overwhelmed by a fast-producing Observable.

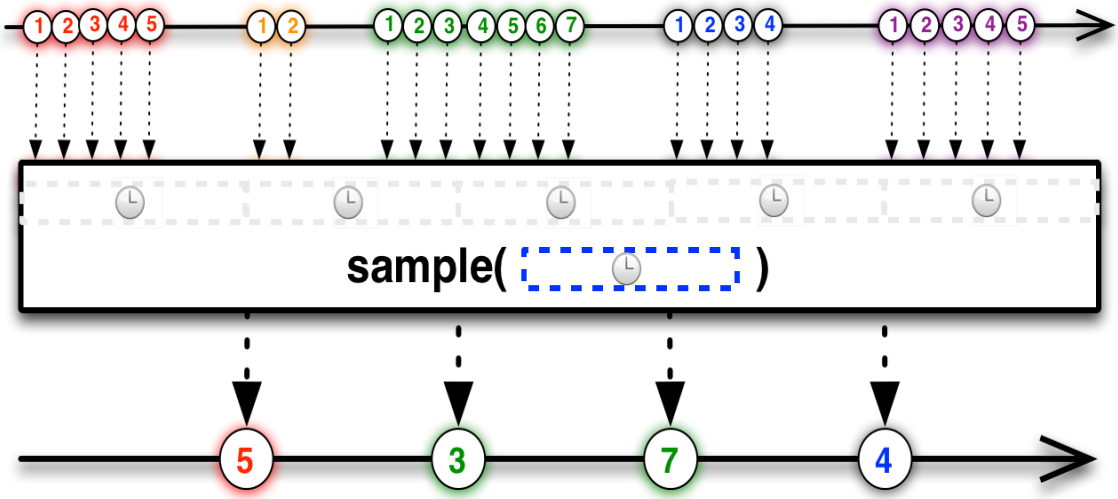
Throttling

Operators like `sample()` or `throttleLast()`, `throttleFirst()`, and `throttleWithTimeout()` or `debounce()` allow you to regulate the rate at which an Observable emits items.

The following diagrams show how you could use each of these operators on the bursty Observable shown above.

sample (or throttleLast)

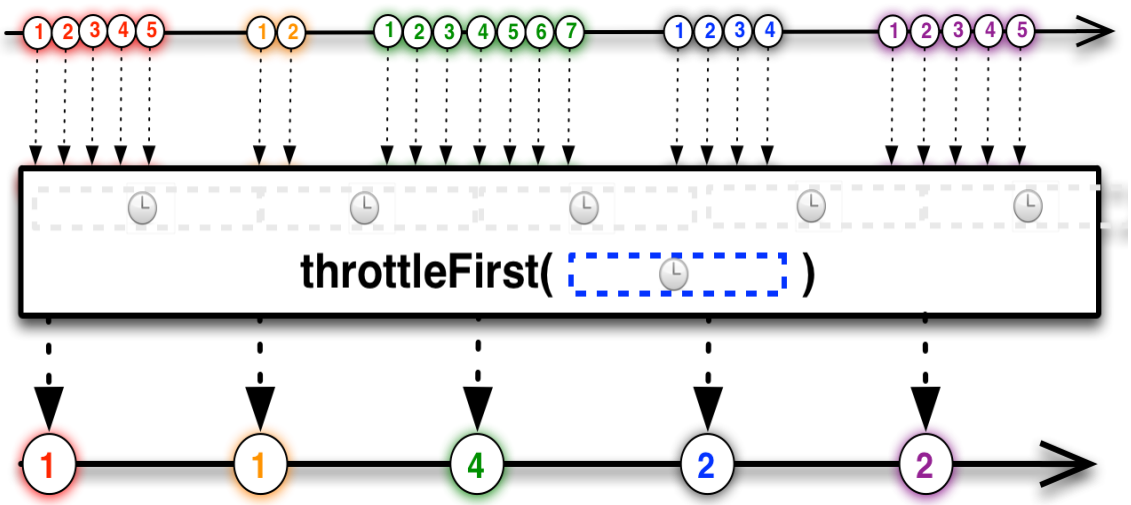
The `sample` operator periodically "dips" into the sequence and emits only the most recently emitted item during each dip:




```
Observable<Integer> burstySampled = bursty.sample(500, TimeUnit.MILLISECONDS);
```

throttleFirst

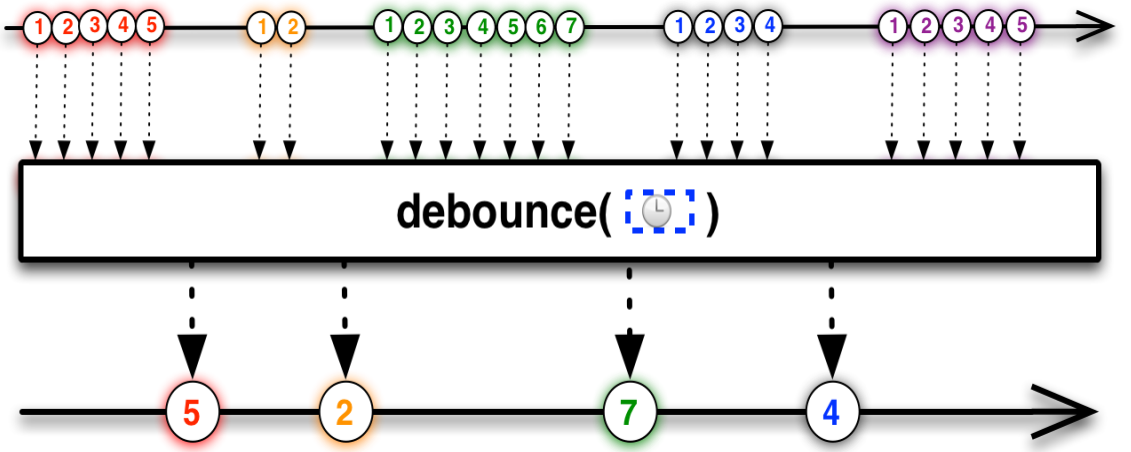
The `throttleFirst` operator is similar, but emits not the most recently emitted item, but the first item that was emitted after the previous "dip":




```
Observable<Integer> burstyThrottled = bursty.throttleFirst(500,
TimeUnit.MILLISECONDS);
```

debounce (or throttleWithTimeout)

The `debounce` operator emits only those items from the source Observable that are not followed by another item within a specified duration:




```
Observable<Integer> burstyDebounced = bursty.debounce(10, TimeUnit.MILLISECONDS);
```

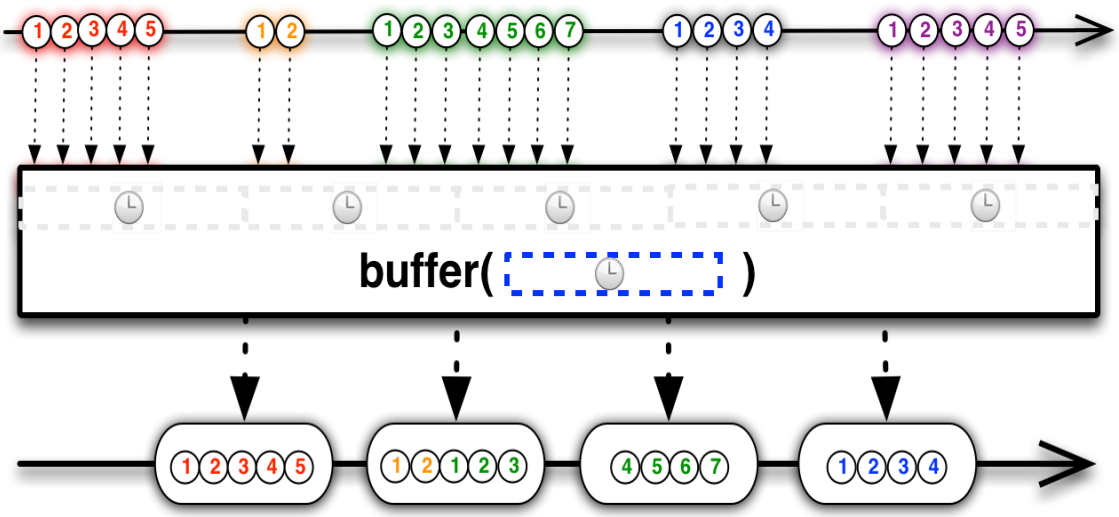
Buffers and windows

You can also use an operator like `buffer()` or `window()` to collect items from the over-producing Observable and then emit them, less-frequently, as collections (or Observables) of items. The slow consumer can then decide whether to process only one particular item from each collection, to process some combination of those items, or to schedule work to be done on each item in the collection, as appropriate.

The following diagrams show how you could use each of these operators on the bursty Observable shown above.

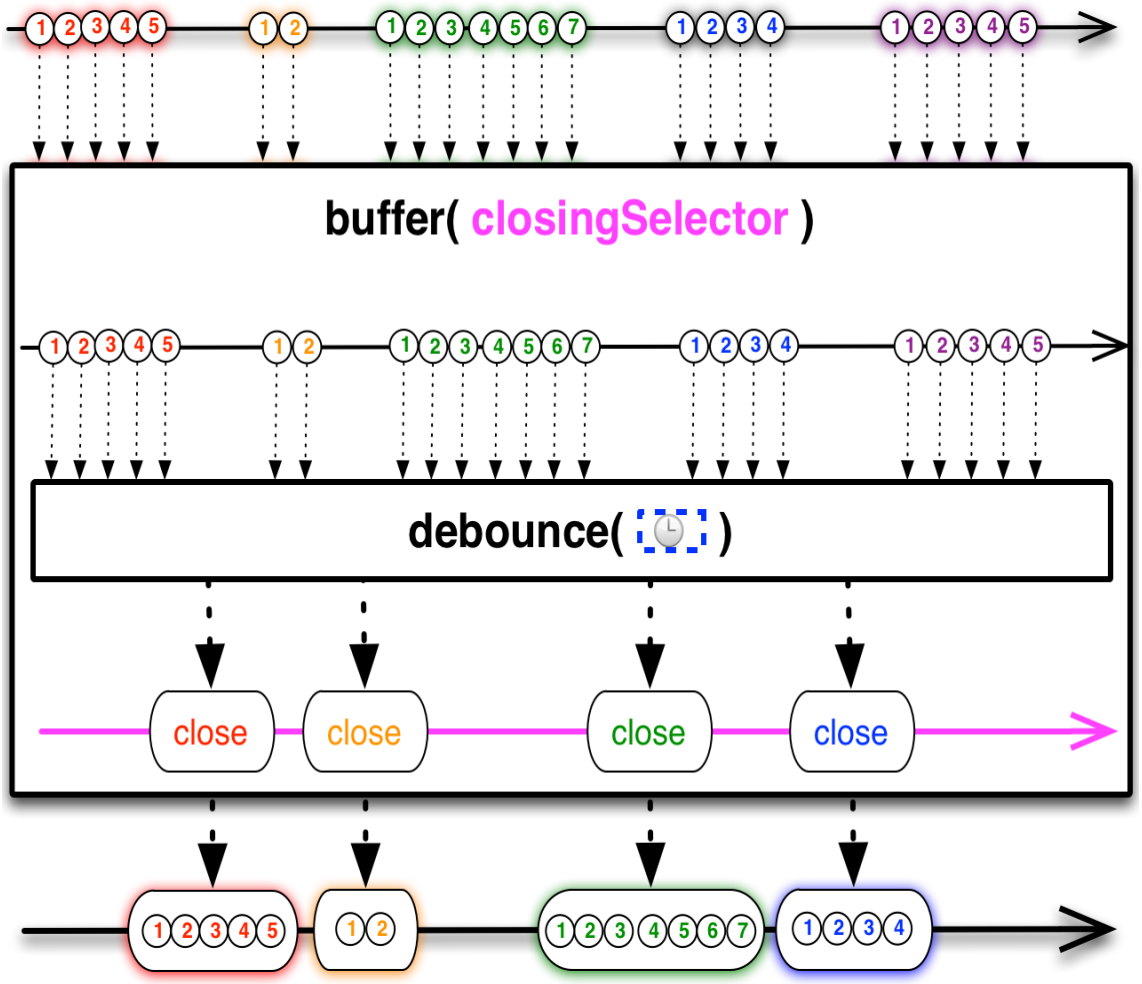
buffer

You could, for example, close and emit a buffer of items from the bursty Observable periodically, at a regular interval of time:




```
Observable<List<Integer>> burstyBuffered = bursty.buffer(500,
TimeUnit.MILLISECONDS);
```

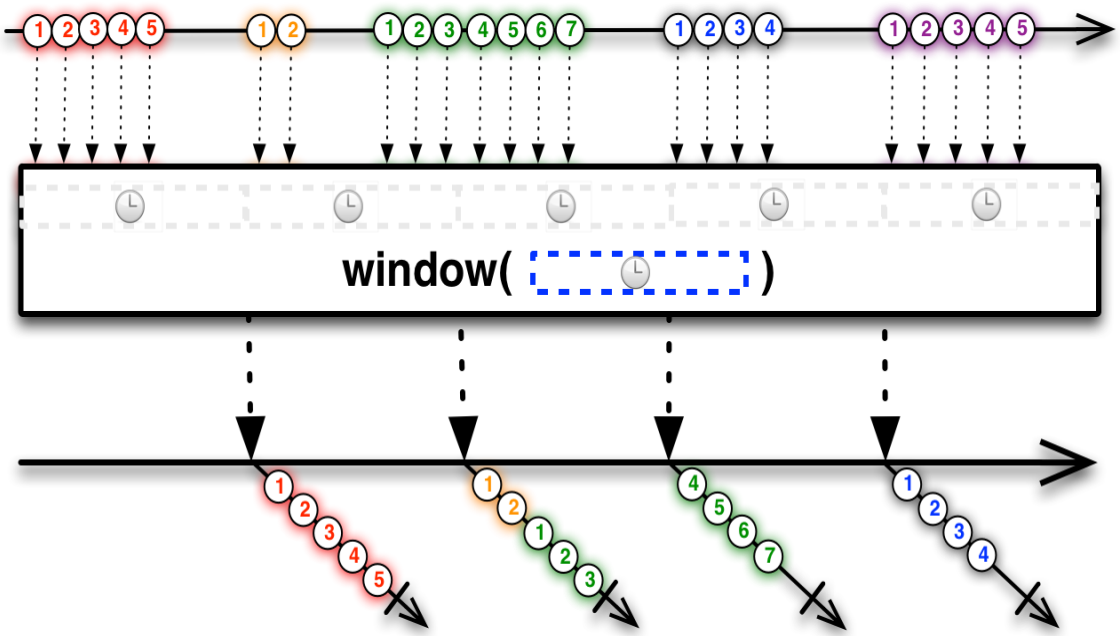
Or you could get fancy, and collect items in buffers during the bursty periods and emit them at the end of each burst, by using the `debounce` operator to emit a buffer closing indicator to the `buffer` operator:




```
// we have to multicast the original bursty Observable so we can use it
// both as our source and as the source for our buffer closing selector:
Observable<Integer> burstyMulticast = bursty.publish().refCount();
// burstyDebounced will be our buffer closing selector:
Observable<Integer> burstyDebounced = burstMulticast.debounce(10,
TimeUnit.MILLISECONDS);
// and this, finally, is the Observable of buffers we're interested in:
Observable<List<Integer>> burstyBuffered = burstyMulticast.buffer(burstyDebounced);
```

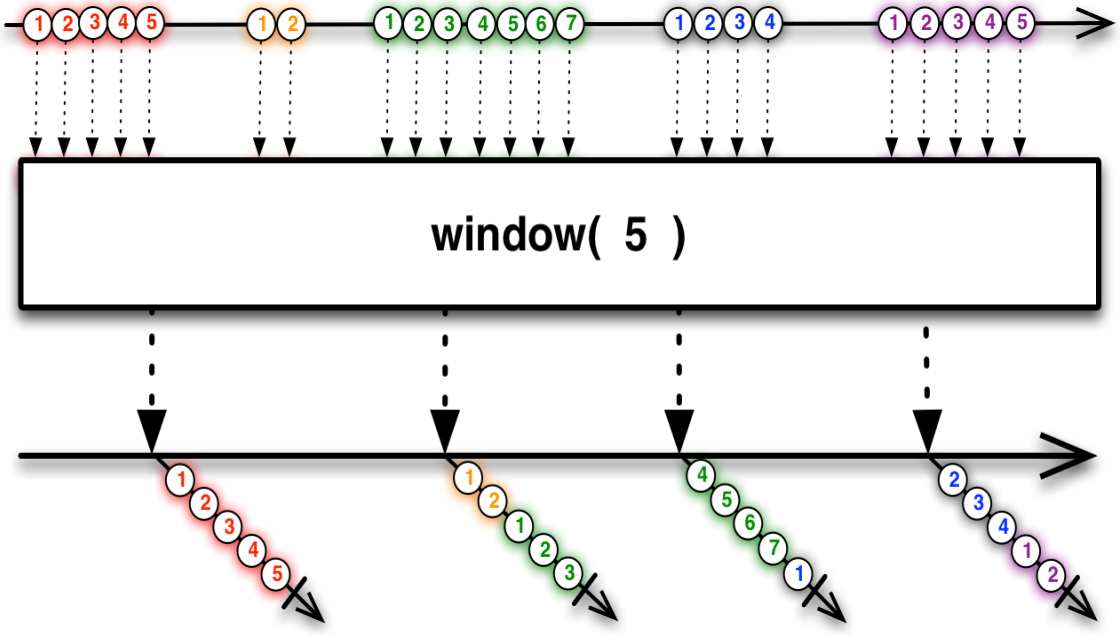
window

`window` is similar to `buffer`. One variant of `window` allows you to periodically emit Observable windows of items at a regular interval of time:




```
Observable<Observable<Integer>> burstyWindowed = bursty.window(500,
TimeUnit.MILLISECONDS);
```

You could also choose to emit a new window each time you have collected a particular number of items from the source Observable:




```
Observable<Observable<Integer>> burstyWindowed = bursty.window(5);
```

Callstack blocking as a flow-control alternative to backpressure

Another way of handling an overproductive Observable is to block the callstack (parking the thread that governs the overproductive Observable). This has the disadvantage of going against the "reactive" and non-blocking model of Rx. However this can be a viable option if the problematic Observable is on a thread that can be blocked safely. Currently RxJava does not expose any operators to facilitate this.

If the Observable, all of the operators that operate on it, and the observer that is subscribed to it, are all operating in the same thread, this effectively establishes a form of backpressure by means of callstack blocking. But be aware that many Observable operators do operate in distinct threads by default (the javadocs for those operators will indicate this).

How a subscriber establishes "reactive pull" backpressure

When you subscribe to an `Observable` with a `Subscriber`, you can request reactive pull backpressure by calling `Subscriber.request(n)` in the `Subscriber`'s `onStart()` method (where *n* is the maximum number of items you want the `Observable` to emit before the next `request()` call).

Then, after handling this item (or these items) in `onNext()`, you can call `request()` again to instruct the `Observable` to emit another item (or items). Here is an example of a `Subscriber` that requests one item at a time from `someObservable`:

```
someObservable.subscribe(new Subscriber<T>() {
    @Override
    public void onStart() {
      request(1);
    }

    @Override
    public void onCompleted() {
      // gracefully handle sequence-complete
    }

    @Override
    public void onError(Throwable e) {
      // gracefully handle error
    }

    @Override
    public void onNext(T n) {
      // do something with the emitted item "n"
      // request another item:
      request(1);
    }
});
```

You can pass a magic number to `request`, `request(Long.MAX_VALUE)`, to disable reactive pull backpressure and to ask the Observable to emit items at its own pace. `request(0)` is a legal call, but has no effect. Passing values less than zero to `request` will cause an exception to be thrown.

Reactive pull backpressure isn't magic

Backpressure doesn't make the problem of an overproducing Observable or an underconsuming Subscriber go away. It just moves the problem up the chain of operators to a point where it can be handled better.

Let's take a closer look at the problem of the uneven `zip`.

You have two Observables, *A* and *B*, where *B* is inclined to emit items more frequently than *A*. When you try to `zip` these two Observables together, the `zip` operator combines item *n* from *A* and item *n* from *B*, but meanwhile *B* has also emitted items *n*+1 to *n*+*m*. The `zip` operator has to hold on to these items so it can combine them with items *n*+1 to *n*+*m* from *A* as they are emitted, but meanwhile *m* keeps growing and so the size of the buffer needed to hold on to these items keeps increasing.

You could attach a throttling operator to *B*, but this would mean ignoring some of the items *B* emits, which might not be appropriate. What you'd really like to do is to signal to *B* that it needs to slow down and then let *B* decide how to do this in a way that maintains the integrity of its emissions.

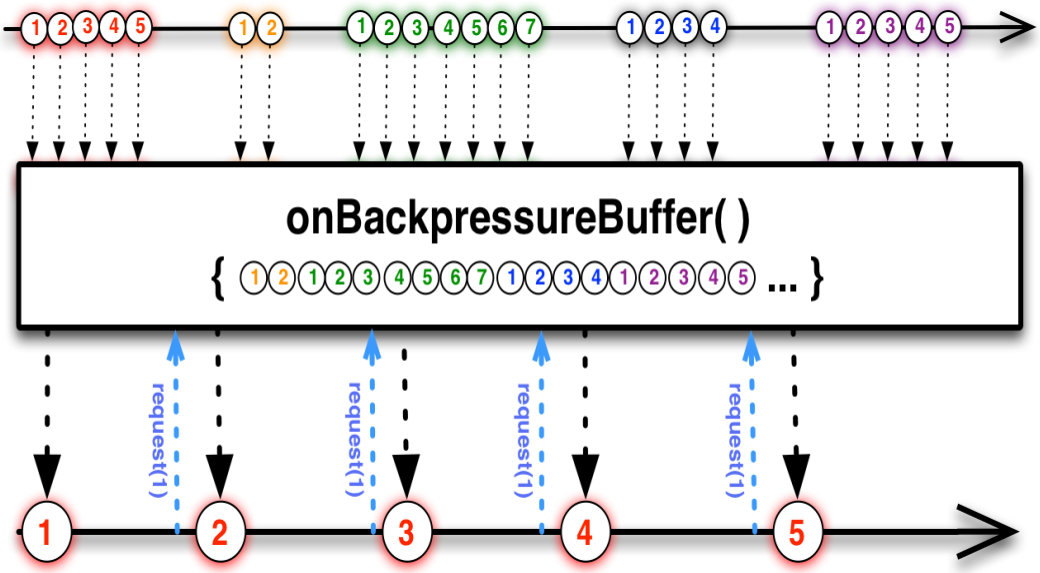
The reactive pull backpressure model lets you do this. It creates a sort of active pull from the Subscriber in contrast to the normal passive push Observable behavior.

The `zip` operator as implemented in RxJava uses this technique. It maintains a small buffer of items for each source Observable, and it requests no more items from each source Observable than would fill its buffer. Each time `zip` emits an item, it removes the corresponding items from its buffers and requests exactly one more item from each of its source Observables.

(Many RxJava operators exercise reactive pull backpressure. Some operators do not need to use this variety of backpressure, as they operate in the same thread as the Observable they operate on, and so they exert a form of blocking backpressure simply by not giving the Observable the opportunity to emit another item until they have finished processing the previous one. For other operators, backpressure is inappropriate as they have been explicitly designed to deal with flow control in other ways. The RxJava javadocs for those operators that are methods of the Observable class indicate which ones do not use reactive pull backpressure and why.)

For this to work, though, Observables *A* and *B* must respond correctly to the `request()`. If an Observable has not been written to support reactive pull backpressure (such support is not a requirement for Observables), you can apply one of the following operators to it, each of which forces a simple form of backpressure behavior:

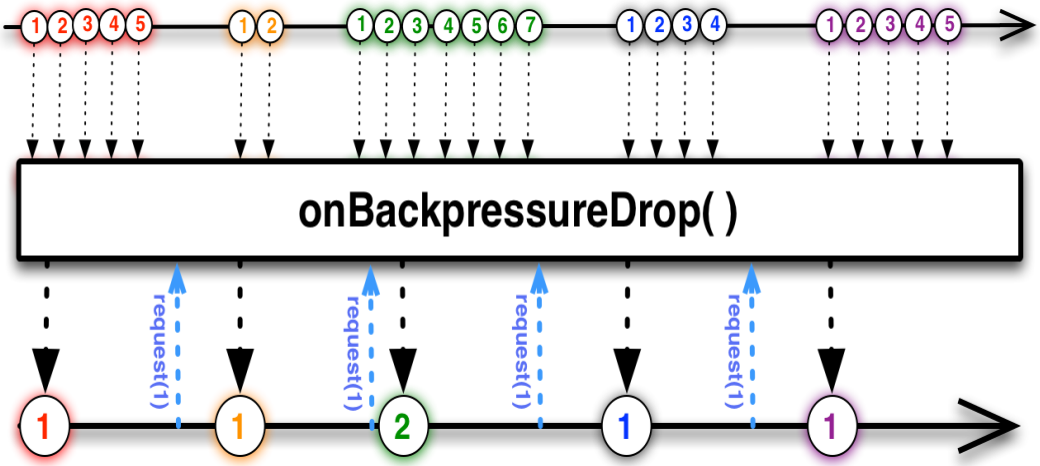
`onBackpressureBuffer`
: maintains a buffer of all emissions from the source Observable and emits them to downstream Subscribers according to the `request`s they generate



an experimental version of this operator (not available in RxJava 1.0) allows you to set the capacity of the buffer; applying this operator will cause the resulting Observable to terminate with an error if this buffer is overrun

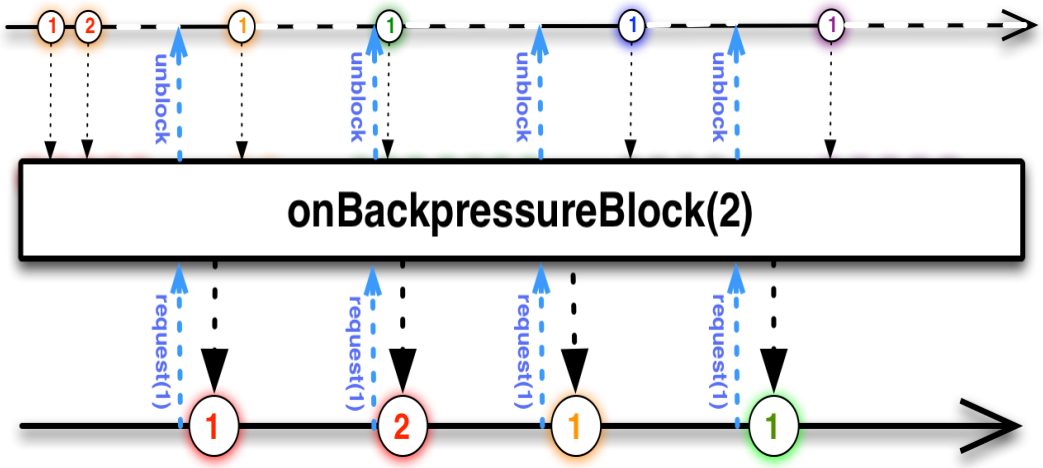
`onBackpressureDrop`

drops emissions from the source Observable unless there is a pending `request` from a downstream Subscriber, in which case it will emit enough items to fulfill the request



`onBackpressureBlock` *(experimental, not in RxJava 1.0)*

blocks the thread on which the source Observable is operating until such time as a Subscriber issues a `request` for items, and then unblocks the thread only so long as there are pending requests



If you do not apply any of these operators to an Observable that does not support backpressure, *and* if either you as the Subscriber or some operator between you and the Observable attempts to apply reactive pull backpressure, you will encounter a `MissingBackpressureException` which you will be notified of via your `onError()` callback.

Further reading

If the standard operators aren't providing the expected behavior, one can write custom operators in RxJava.

See also

- RxJava 0.20.0-RC1 release notes