acmqueue

Your Mouse is a Database

Web and mobile applications are increasingly composed of asynchronous and realtime streaming services and push notifications.

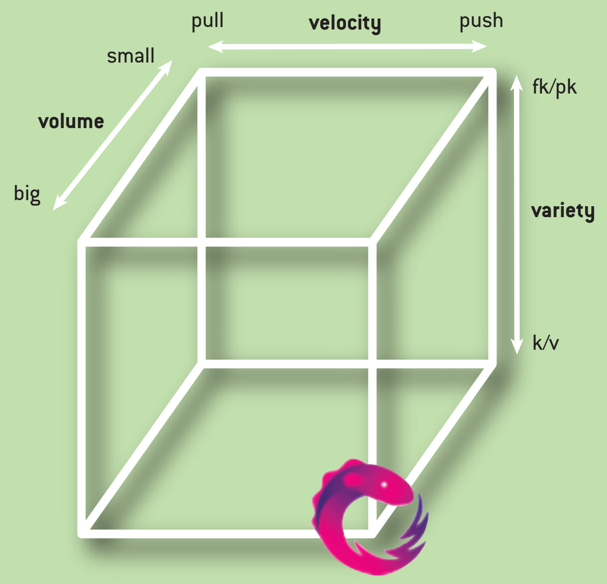
Erik Meijer

Among the hottest buzzwords in the IT industry these days is "big data," but the "big" is something of a misnomer: big data is not just about volume, but also about velocity and variety:[4]

- The *volume* of data ranges from a small number of items stored in the closed world of a conventional RDMS (relational database management system) to a large number of items spread out over a large cluster of machines or across the entire World Wide Web.
- The *velocity* of data ranges from the consumer synchronously pulling data from the source to the source asynchronously pushing data to its clients, at different speeds, ranging from millisecond-latency push-based streams of stock quotes to reference data pulled by an application from a central repository once a month.
- The *variety* of data ranges from SQL-style relational tuples with foreign-/primary-key relationships to coSQL-style objects or graphs with key-value pointers, or even binary data such as videos and music.

If we draw a picture of the design space for big data along these three dimensions of volume, velocity, and variety, then we get the big-data cube shown in figure 1. Each of the eight corners of

FIGURE 1 **Big Data Cube**



the cube corresponds to a (well-known) database technology. For example, the traditional RDMS is at the top-back corner with coordinates (small, pull, fk/pk), meaning that the data sets are small; it assumes a closed world that is under full control by the database, clients synchronously pull rows out of the database after they have issued a query, and the data model is based on Codd's relational model. Hadoop-based systems such as HBase are on the front-left corner with coordinates (big, pull, fk/pk). The data model is still rectangular with rows, columns, and primary keys, and results are still pulled by the client out of the store, but the data is stored on a cluster of machines using some partitioning scheme.

When moving from the top plane to the bottom plane, the data model changes from rows with primary and foreign keys to objects and pointers. On the bottom-left corner at coordinates (small, pull, k/v) are traditional O/R (object/relational) mapping solutions such as LINQ to SQL and Hibernate, which put an OO (object-oriented) veneer on top of relational databases. In the front of the cube is LINQ to Objects with coordinates (big, pull, k/v). It virtualizes the actual data source using the `IEnumerable<T>` interface, which allows for an infinite collection of items to be generated on the fly. To the right, the cube changes from batch processing to streaming or realtime data where the data source asynchronously pushes data to its clients. Streaming database systems with a rows-and-columns data model such as Percolator, StreamBase, and StreamInsight occupy the top-right axis.

Finally, on the bottom right at coordinates (big, push, k/v), is Rx (Reactive Extensions), or as it is sometimes called, LINQ to Events, which is the topic of this article.

The goal of Rx is to coordinate and orchestrate event-based and asynchronous computations such as low-latency sensor streams, Twitter and social media status updates, SMS messages, GPS coordinates, mouse moves and other UI events, Web sockets, and high-latency calls to Web services using standard object-oriented programming languages such as Jav, C#, or Visual Basic.

There are many ways to derive Rx, some involving category theory and appealing to mathematical duality, but this article shows how every developer could have invented Rx by crossing the standard JDK (Java Development Kit) `Future<T>` interface with the GWT (Google Web Toolkit) `AsyncCallBack<T>` interface to create the pair of interfaces `IObservable<T>` and `IObserver<T>` that model asynchronous data streams with values of type `T`. This corresponds to the well-known Subject/Observer design pattern. The article then shows how to write a simple Ajax-style application by exposing UI events and Web services as asynchronous data streams and composing them using a fluent API.

CROSSING FUTURES AND CALLBACKS

The GWT developer documentation contains a slightly apologetic section called "Getting Used to Asynchronous Calls,"[3] which explains that while asynchronous calls at first sight look cruel and unnatural to the developer, they are a necessary evil to prevent the UI from locking up and allow a client to have multiple concurrent outstanding server requests.

In GWT the asynchronous counterpart of a synchronous method, say `Person[] getPeople(…)`, that makes a synchronous cross-network call and blocks until it returns an array of `Person`, would return `void` and take an additional callback argument `void getPeople(…, AsyncCallback<Person[]> callback)`. The callback interface has two methods: `void onFailure(Throwable error)`, which is called when the asynchronous call throws an exception; and `void onSuccess(T result)`, which is called when the asynchronous call successfully returns a result value. Given an asynchronous function such as getPeople, an invocation typically passes an anonymous interface implementation that handles the success and failure callbacks, respectively, as follows:

```
service.getPeople(startRow, maxRows, new AsyncCallback<Person[]>() {
   void onFailure(Throwable error) { …code to handle failure… }
   void onSuccess(Person[] result) { …code to handle success… }
});
```

While the commitment to asynchrony in GWT is laudable, GWT misses a huge opportunity by not further refining and unifying the asynchronous programming model across the entire framework. For example, the RequestBuilder class for making direct HTTP calls uses the RequestCallback interface, which has two methods onError and onResponseReceived that are virtually isomorphic to the methods of the AsyncCallback interface previously discussed.

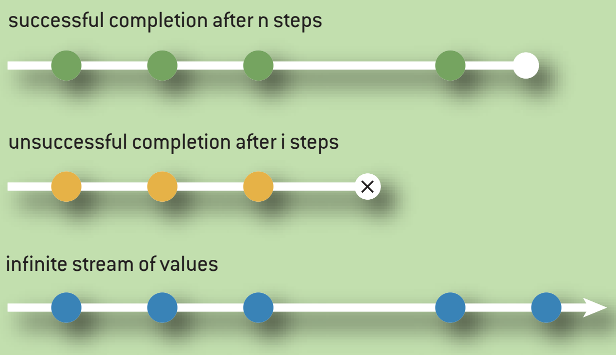
Both the AsyncCallback and RequestCallback interfaces assume that asynchronous calls deliver their results in one shot. In the example, however, returning the elements of the Person array incrementally in a streaming fashion makes perfect sense, especially when the result set is large or even infinite. You can asynchronously stream back results by allowing the onSuccess method to be called multiple times, once for each additional chunk of the result array, and by adding a method void onCompleted(), which is called when all chunks have been delivered successfully. Let's call this derived interface Observer<T> to indicate that it can observe multiple T values before completing and to reflect the standard Java nongeneric observer interface.

With Observer<T> instead of AsyncCallback<T>, the possible sequences of interaction between an asynchronous computation and its client are: (a) successful termination after $n \geq 0$ values; (b) unsuccessful termination after i values; or (c) an infinite stream of values that never completes, as shown in figure 2.

Another downside of passing callbacks directly as parameters to asynchronous methods is that revoking a callback from being invoked is tricky once the call has been made, leaving you with just a void in your hands. Suppose, for example, that the function getPeople streams back the

FIGURE 2

Possible Sequences of Interaction When Using Observer<T>


successful completion after n steps
unsuccessful completion after i steps
infinite stream of values

names of the people who have signed up for a marketing promotion every minute, but that you are not interested in receiving more than the first thousand names. How do you achieve this later if you did not anticipate this pattern when you made the call and received back `void`? Even if the asynchronous call delivers at most one value, you may choose later to ignore or cancel the call by timing out after not receiving a result within a certain time interval. Again, this is possible if you anticipated this when passing the callback into `getPeople`, but you cannot change your mind later.

These hitches are symptoms of the fact that asynchronous computations and streams are not treated as first-class values that can be returned from methods, stored in variables, etc. The next section shows how to make asynchronous data streams by introducing an additional container interface that represents the asynchronous computation itself and on which you register the callbacks to be notified about the results of the computation. Now asynchronous methods can return a value that represents the pending asynchronous computation or stream instead of just `void`, which you can treat just like any other regular value. In particular, it allows you to change your mind after making the call and filter, manipulate, or transform the computation at will.

The Java SDK already provides (single-shot) asynchronous computations as first-class values in the form of the `Future<T>` interface, whose principal method `T get()` retrieves the result of the computation and blocks when the underlying computation has not yet terminated:

```
interface Future<T>
{
   boolean cancel(boolean mayInterruptIfRunning);
   T get();
   T get(long timeout, TimeUnit unit);
   boolean isCancelled();
   boolean isDone();
}
```

Note that in principle, `Future<T>` could be used to produce multiple values. In this case each call to `get()` would block and return the next value once it is available, as long as `isDone()` is not true. This is similar to the `iterable` interface. In the rest of this article, asynchronous computations are assumed to return streams of multiple results.

While futures do provide a first-class representation of asynchronous computations, the `get` method is blocking. Fortunately, you can make the JDK interface `Future<T>` nonblocking by supplying the `T get()` method with a callback of type `Observer<T>` (the interface introduced to extend the `AsyncCallback<T>` interface of GWT). Note that the blocking `isCancelled` and `isDone` methods are no longer needed because that information is transmitted via the callback as well. For simplicity, ignore the second overload of `get` since you can easily reconstruct that later. Applying these changes, the nonblocking version of the `Future<T>` interface looks like this:

```
interface Future<T>
{
   boolean cancel(boolean mayInterruptIfRunning);
   void get(Observer<T> callback);
}
```

You are not yet done refactoring. Instead of cancelling the future as a whole via the `cancel` method, it makes more sense to cancel just the particular outstanding call to `get` per observer. This can be achieved by letting `get` return an interface that represents a cancellable resource. Moreover, since you have already called `get`, there is no need to specify the `mayInterruptIfRunning` parameter, because the computation is already running at that point and you can encode the Boolean by deciding whether or not to call `cancel`. Lastly, you can make the `cancel` method nonblocking by returning `void` instead of `Boolean`. You could try to make `cancel` return a `Future<boolean>` instead, but then you would fall into an endless recursive rabbit hole of asynchrony. As it turns out, the `java.io.Closable` interface precisely fits the bill, resulting in the following mutation of `Future<T>`:

```
interface Future<T> { Closable get(Observer<T> callback); }
```

Note that calling the `close()` method of the `Closable` interface returned by a subscription may or may not actually cancel the underlying computation, because a single observable may have multiple observers (disposing, say, of the subscription to mouse moves, which should not stop your mouse from working). Since that particular observer is not notified of any further values, however, from its perspective the computation has terminated. If needed, the class that implements `IObservable<T>` could cancel the computation in some other way.

Instead of `Future<T>` and `Observer<T>`, .NET has the standard `IObservable<T>` and `IObserver<T>` interfaces; and instead of `Closable`, it has `IDisposable`. Values of type `IObservable<T>` (or `Observer<T>` depending on your preferred programming language) represent asynchronous data streams, or event streams, with values of type `T`.

```
interface IObservable<T>
{
    IDisposable Subscribe(IObserver<T> observer);
}
interface IObserver<T>
{
    void OnNext(T value);
    void OnError(Exception error);
    void OnCompleted();
}
interface IDisposable
{
    void Dispose();
}
```

A closer inspection of the resulting interface trinity reveals a generic variation of the classic Subject/Observer interface[2] for the publish/subscribe pattern, a staple in the tool chest of object-oriented programmers for decades for dealing with event-based systems. JDK 1.0 already supports this pattern via the (nongeneric) `Observable` class and the `Observer` interface. In .NET, the Rx library supports the pattern.

The Rx library makes some additional behavioral assumptions about the `IObserver<T>` and `IObservable<T>` interfaces that are not expressed by their (syntactic) type signatures:

- The sequence of calls to an instance of the `IObserver<T>` interface should follow the regular expression `OnNext(t)*` (`OnCompleted()` | `OnError(e)`)?. In other words, after zero or more `OnNext` calls, either one of `OnCompleted` or `OnError` will optionally be called.
- Implementations of `IObserver<T>` can assume to be synchronized; conceptually they run under a lock, similar to regular .NET event handlers, or the reactor pattern.[9]
- All resources associated with an observer should be cleaned up at the moment `OnError` or `OnCompleted` is called. In particular, the subscription returned by the `Subscribe` call of the observer will be disposed of by the observable as soon as the stream completes. In practice this is implemented by closing over the `IDisposable` returned by `Subscribe` in the implementation of the `OnError` and `OnCompleted` methods.
- When a subscription is disposed of externally, the `IObservable<T>` stream should make a *best-effort* attempt to stop all outstanding work for that subscription. Any work already in progress might still complete as it is not always safe to abort work in progress but should not be signaled to unsubscribed observers.

This contract ensures it is easy to reason about and prove the correctness of operators and user code.

FLUENTLY CREATING, COMPOSING, AND CONSUMING ASYNCHRONOUS DATA STREAMS

To create an instance of an `Observable<T>` in Java, you would use anonymous inner classes and define an abstract base class `ObservableBase<T>` that takes care of enforcing the Rx contract. It is specialized by providing an implementation of the `subscribe` method:

```
Observable<T> observable = new ObservableBase<T>()
{
   Closable subscribe(Observer<T> observer) { … }
};
```

Since .NET lacks anonymous interfaces, it instead uses a factory method `Observable.Create` that creates a new observable instance from an anonymous delegate of type `Func<IObservable<T>, IDisposable>` that implements the `Subscribe` function:

```
IObservable<T> observable = Observable.Create<T>(
   IObserver<T> observer => { … }
);
```

Just as in the Java solution, the concrete type returned by the `Create` method enforces the required Rx behavior.

Once you have a single interface to represent asynchronous data streams, you can expose existing event- and callback-based abstractions such as GUI controls as sources of asynchronous data streams. For example, you can wrap the text-changed event of a `TextField` control in Java as an asynchronous data stream using the following delicious token salad:

```
Observable<string> TextChanges(JTextField tf){
   return new ObservableBase<string>(){
      Closable subscribe(Observer<string> o){
         DocumentListener l = new DocumentListener(){
            void changedUpdate(DocumentEvent e {
                o.OnNext(tf.getText());};
             tf.addDocumentListener (l);
             return new Closable() {
                void close(){tf.removeDocumentListener(l);}}}}}
```

Every time the `changedUpdate` event fires, the corresponding asynchronous data stream of type `Observable<string>` pushes a new string to its subscribers, representing the current content of the text field.

Likewise, you can expose objects with setters as sinks of asynchronous data streams by wrapping them as observers. For example, expose a `javax.swing.JList<T>` list into an `Observer<T[]>` by setting the `listData` property to the given array whenever `onNext` is called:

```
Observer<T[]> ObserveChanges(javax.swing.JList<T> list){
   return new ObserverBase<T[]>() {
      void onNext(T[] values){ list.setListData(values); }}}
```

You can thus view a UI control, the mouse, a text field, or a button as a streaming database that generates an infinite collection of values for each time the underlying control fires an event. Conversely, objects with settable properties such as lists and labels can be used as observers for such asynchronous data streams.

Asynchronous data streams represented by the `IObservable<T>` interface (or `Observable<T>` in Java) behave as regular collections of values of type `T`, except that they are push-based or streaming instead of the usual pull-based collections such as arrays and lists that implement the `IEnumerable<T>` interface (or in Java `iterable<T>`). This means that you can wire asynchronous data streams together using a fluent API of standard query operators to create complex event-processing systems in a highly composable and declarative way.

For example, the `Where` operator takes a predicate of type `Func<S,bool>` and filters out all values for which the predicate does not hold an input-observable collection of type `IObservable<S>`, precisely the same as its cousin that works on pull-based `IEnumerable<T>` collections. Figure 3 illustrates this.

FIGURE 3 **The Where Operator**

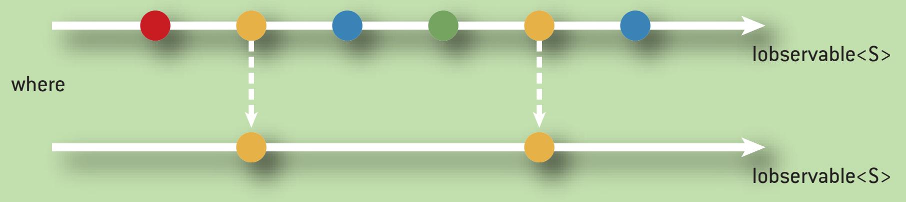


FIGURE 4 **The Select Operator**

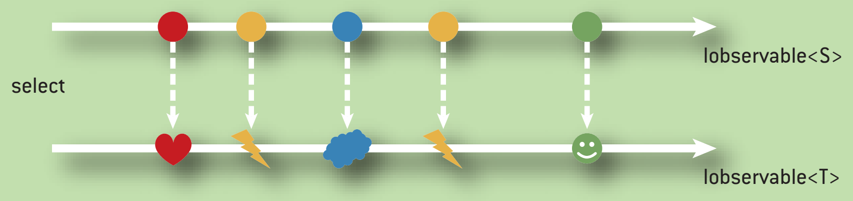
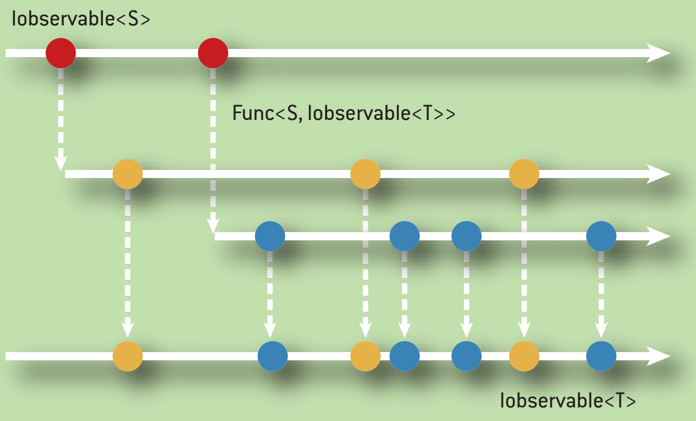


FIGURE 5 **The SelectMany Operator**



Using this operator, you can cleanse a text field input exposed as `IObservable<string>` stream and remove all empty and null strings using the query expression `input.Where(s=>!string.IsNullOrEmpty(s))`.

In Java 8 with lambda expressions and defender methods, the code would look very similar to the C# code shown here, just using `->` instead of `=>` for lambdas and different casing of variable names. Even without those upcoming Java language features, however, you can approximate a fluent interface in Java—as in FlumeJava[1] or Reactive4Java[8]—for manipulating event streams using standard query operators. For example, by having the operators as methods on `ObservableBase<T>`, you can write the filter example as:

```
input.Where<T>(new Func<string,T>{
  Invoke(string s){
    return !(s == null || s.length() == 0); }}
```

To save us all from too much typing, however, the next couple of examples are provided only in C#, even though nothing is C# or .NET specific.

The Select operator takes a transformation function Func<S,T> to transform each value in the input data stream of type IObservable<S>. This produces a new asynchronous result stream of type IObservable<T>, again exactly like the IEnumerable<T>-based version, as seen in figure 4.

The SelectMany operator is often used to wire together two data streams, pull-based or push-based. SelectMany takes a source stream of type IObservable<S> and an inflator function of type Func<S, IObservable<T>>, and from each element in the original source stream generates a new nested stream of zero, one, or more elements. It then merges all intermediate asynchronous data streams into a single output stream of type IObservable<T>, as shown in figure 5.

The SelectMany operator clearly shows the difference between the asynchronous nature of IObservable<T> and the synchronous nature of IEnumerable<T>. As figure 5 shows, values on the source stream appear asynchronously, even as you are still producing values from a previous inflator function. In the IEnumerable<T> case, the next value is pulled from the source stream only after all values from the inflator function have been produced (i.e., the output stream is the concatenation of all subsequent inflator-produced streams, not the nondeterministic interleaving), as shown in figure 6.

Sometimes it is convenient to generate the output stream of an asynchronous stream using a more sequential pattern. As shown in figure 7, the Switch operator takes a nested asynchronous data stream IObservable<IObservable<T>> and produces the elements of the most recent inner asynchronous data stream that has been received up to that point. This produces a new non-nested asynchronous data stream of type IObservable<T>. It allows later streams to override earlier streams, always yielding the "latest possible results," rather like a scrolling news feed.

At this point, the determined reader may attempt to create his or her own implementation of Switch, which, as it turns out, is surprisingly tricky, especially dealing with all edge conditions while also satisfying the Rx contract.

Using the fluent API previously introduced, you can program the prototypical Ajax "dictionary suggest" program in a few lines. Assume you have a dictionary-lookup Web service that, given a word, asynchronously returns an array of completions for that word via the following method:

FIGURE 6

Concatenation of All Subsequent Inflator-Produced Streams

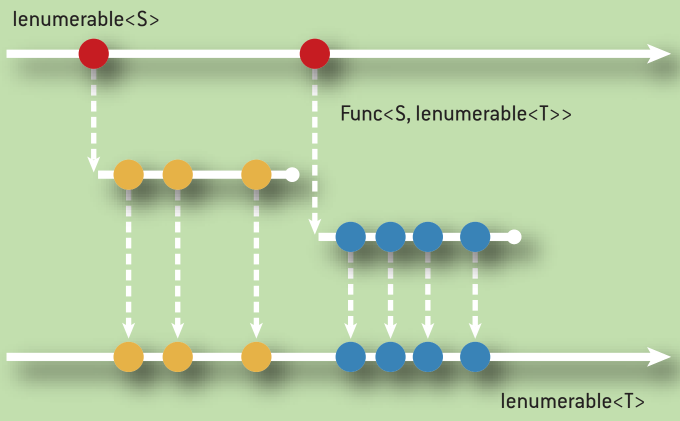
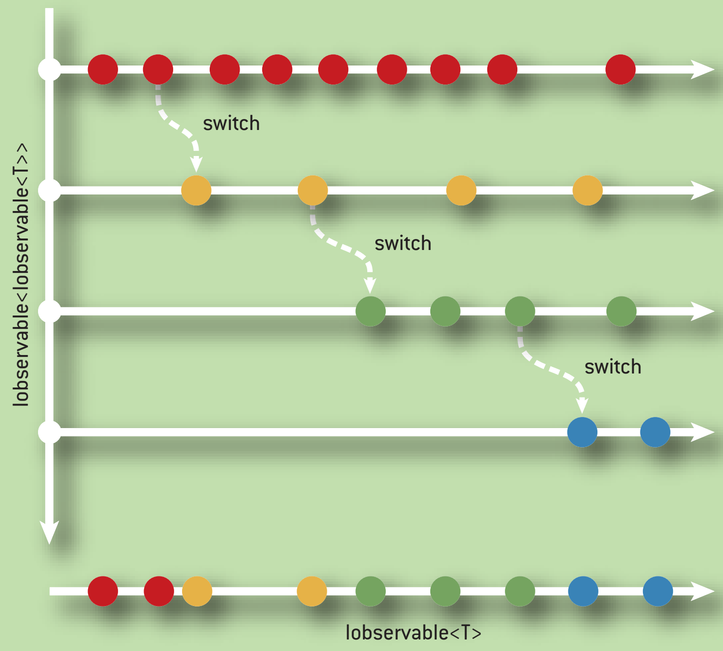


FIGURE 7

The Switch Operator



```
IObservable<string[]> Completions(string word){ … }
```

Assume also that using these helpers, you have exposed a GUI text field `input` as an `IObservable<string>` to produce the value of the text field every time the input changes, and you have wrapped a label `output` as an `IObserver<string[]>` to display an asynchronous data stream of string arrays. Then you can wire up a pipeline that asynchronously calls the `Completions` service for every partial word typed into the text field but displays only the most recent result on the label:

```
TextChanges(input)
.Select(word⇒Completions(word))
.Switch()
.Subscribe(ObserveChanges(output));
```

The effect of the `Switch` operator is that every time another asynchronous call is made to `Completions` in response to a change in the input, the result is switched to receive the output of this latest call, as shown in figure 8, and the results from all previous calls that are still outstanding are ignored.

This is not just a performance optimization. Without using `Switch`, there would be multiple outstanding requests to the `Completion` service, and since the stream is asynchronous, results could come back in arbitrary order, possibly updating the UI with results of older requests.

This basic dictionary example can be improved by inserting a few more operators into the query. The first operator is `IObservable<T> DistinctUntilChanged(IObservable<T> source)`, which ensures that an asynchronous data stream contains only distinct contiguous elements—in other words, it removes adjunct elements that are equivalent. For the example, this ensures that `Completions` is called only when the input has actually changed.

Second, if the user types faster than you can make calls to the Web service, a lot of work will go to waste since you are firing off many requests, only to cancel them immediately when the input changes again before the previous result has come back. Instead, you want to wait at least `N` milliseconds since the last change using the operator `IObservable<T> Throttle(IObservable<T> source, TimeSpan delay)`. The throttle operator samples an asynchronous data stream by ignoring values that are followed by another value before the specified delay, as shown in figure 9.

The throttle operator drops events that come in at too high a rate; however, one can easily define other operators that aggregate events in (tumbling) windows or sample the input stream at certain intervals.

FIGURE 8

Asynchronous Call Made to Completions in Response to a Change in the Input

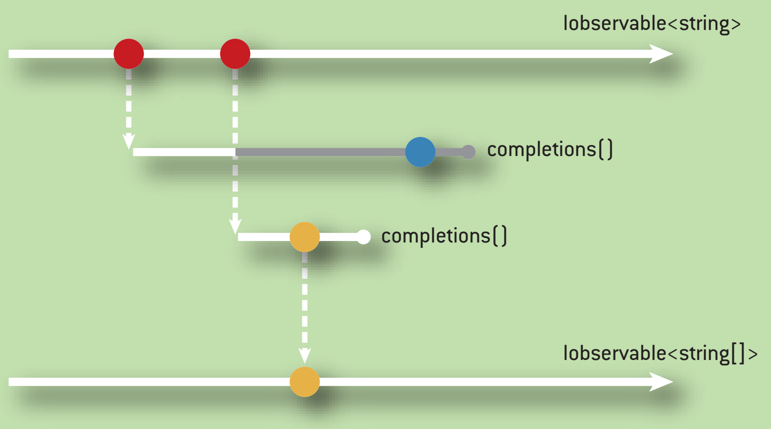
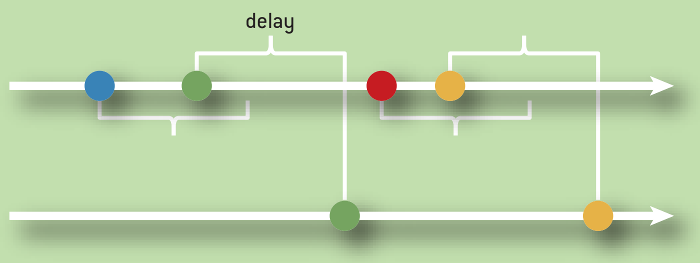


FIGURE 9

The Throttle Operator



The final Ajax program presented here is a dataflow pipeline that invokes the dictionary service only if there has not been a new distinct value fired by `input` in the last 10 milliseconds; and it ignores the result of the service if a new input value appears before the completion of the previous value has returned:

```
        TextChanges(input)
.DistinctUntilChanged()
.Throttle(TimeSpan.FromMilliSeconds(10))
        .Select(word⇒Completions(word))
.Switch()
        .Subscribe(ObserveChanges(output));
```

Of course, the `IObservable<T>` interface is not restricted to UI events and asynchronous computations but applies equally well to any push-based data stream such as tweets, stock stickers, GPS position, etc., and of course the kinds of asynchronous service invocations for GWT we started off with. For example, you can model a Twitter stream that filters on a given hashtag as a method:

```
IObservable<Tweet> TweetsByHashTag(string hashtag, …)
```

This will push an (infinite) stream of tweets to the client that called the function. Further, looking inward (rather than outward), observers are natural expressions of "completion ports" for asynchronous I/O and coprocessor operations such as those from DSPs (digital signal processors) and GPUs (graphics processing units).

SHOW ME THE (CO)MONADS!

So far we have been able to avoid the "M" word (and the "L" word as well), but there's no more hiding it. If we ignore operational concerns such as exceptions, termination, and canceling subscriptions and boil things down to their essence, the `IObservable<T>` and `IObserver<T>` interfaces represent functions of type `(T->())->()`, which is the continuation monad, the mother of all monads, and a co-monad.

Historically, we did not discover the Rx interfaces by the refactorings performed in this article. Instead we applied the definition of *categorical duality* from Wikipedia literally to the `IEnumerable<T>` and `IEnumerator<T>` interfaces for pull-based collections, and thus derived the `IObservable<T>` and `IObserver<T>` interfaces completely mechanically by swapping the arguments and results of all method signatures, not guided by any operational intuition in the process.

Note that our model of asynchronous data streams makes no special assumptions about time. This makes the approach different from the typical reactive programming approaches in functional programming such as Fran or FlapJax that emphasize (continuous) time-varying values, called behaviors, and SQL-based complex event-processing systems such as StreamBase and StreamInsight that also emphasize time in their semantic model. Instead clocks and timers are treated just as regular asynchronous data streams of type `IObservable<DateTimeOffset>`. We parameterize over concurrency and logical clocks by another interface `IScheduler` (slightly simplified here), which represents an execution context that has a local notion of time on which work can be scheduled in the future:

```
interface IScheduler
{
  DateTimeOffset Now { get; }
  IDisposable Schedule(Action work, TimeSpan dueTime)
}
```

Java programmers will immediately see the correspondence with the `executor` interface that in the Java SDK plays the same role of abstracting over the precise introduction of concurrency.

CONCLUSION

Web and mobile applications are increasingly composed of asynchronous and realtime streaming services and push notifications, a particular form of big data where the data has positive velocity. This article has shown how to expose asynchronous data streams as push-based collections of type `IObservable<T>` (in contrast to pull-based collections of type `IEnumerable<T>`) and how to query asynchronous data streams using the fluent API operators provided by the Rx library. This popular library is available for .NET and JavaScript (including bindings for prevalent frameworks such as JQuery and Node) and also ships in the ROM of Windows Phone. F#'s first-class events are based on Rx, and alternative implementations for other languages such as Dart [7] or Haskell[6] are created by the community.

To learn more about LINQ in general and Rx in particular, read the short textbook *Programming Reactive Extensions and LINQ.*[5]

REFERENCES

1. Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R. R., Bradshaw, R., Weizenbaum, N. 2010. FlumeJava: easy, efficient, data- parallel pipelines. *Proceedings of the ACM SIGPLAN Conference on Programming Design and Implementation*; https://dl.acm.org/citation.cfm?id=1806638.
2. Eugster, P. Th., Felber, P. A., Guerraiou, R., Kermarrec, A-M. 2003. The many faces of publish/subscribe. *ACM Computing Surveys 35(2):114-131; https://dl.acm.org/citation.cfm?id=857076.857078*.
3. Google Web Toolkit. 2007. Getting used to asynchronous calls; http://www.gwtapps.com/doc/html/com.google.gwt.doc.DeveloperGuide.RemoteProcedureCalls.GettingUsedToAsyncCalls.html.
4. Laney, D. 2001. 3D data management: Controlling data volume, velocity, and variety. Application Delivery Strategies; http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf.
5. Liberty, J., Betts, P. 2011. *Programming Reactive Extensions and LINQ*. New York: Apress; http://www.apress.com/9781430237471.
6. Reactive-bacon; https://github.com/raimohanska/reactive-bacon.
7. Reactive-Dart; https://github.com/prujohn/Reactive-Dart.
8. Reactive4java; https://code.google.com/p/reactive4java/.
9. Wikipedia. Reactor pattern; https://en.wikipedia.org/wiki/Reactor_pattern.

LOVE IT, HATE IT? LET US KNOW
feedback@queue.acm.org

ERIK MEIJER (emeijer@microsoft.com) has been working on "Democratizing the Cloud" for the past 15 years. He is perhaps best known for his work on the Haskell language and his contributions to LINQ and the Rx Framework.