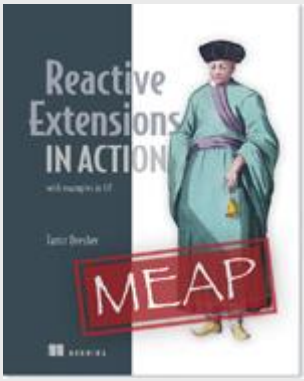

Reactive Extensions: What is Asynchronicity?

By Tamir Dresher

In this article, excerpted from Reactive Extensions in Action, I define asynchronicity and talk about why it's so important to a Reactive application.

Asynchronous message passing is a key trait of a Reactive system, but what exactly is asynchronicity means and why is it so important to a Reactive application?

Our lives are made up of many asynchronous tasks. You may not be aware of it, but your everyday activities woud be annoying if they weren't asynchronous by nature.

To understand what asynchronicity is we first need to understand non-asynchronous execution, or synchronous execution.

Synchronous: (Merriam Webster) Happening, existing, or arising at precisely the same time.

Synchronous execution means that you have to wait for a task to complete before you can continue to the next task. A real-life example of synchronous execution could be the way you approach the staff at the counter, decide what to order while the clerk waits, wait until the meal is ready, and the clerk waits until you hand the payment. Only then you could can to the next task of going to your table to eat. This sequence is shown in Figure 1.

Figure 1 Synchronous food order in which every step must be completed before going to the next one.

This type of sequence feels like a waste of time (or, better said, a waste of resources), so imagine how your applications feel when you do the same for them. The next section will demonstrate this.

It's all about resource utilization

Imagine how your life would be if you had to wait for every single operation to complete before you could do something else. Think of the resource that would be waiting and utilized at that time. The same issues are also relevant in computer science:

```
resultA=LongOperationA();
resultB=LongOperationB();
resultC=LongOpertionsC();
```

In this synchronous code fragment, `LongOperationC()` won't start execution until `LongOprationB()` and `LongOperationA)` completes. During the time that each of these methods are executed the calling thread is blocked and the resources it holds are practically wasted and cannot be utilized to serve other requests or handle other events. If this was happening on the UI thread then the application would look frozen until the execution finish.

If this was happening on a server application, then at some point we might run out of free threads and requests would start being rejected. In both of those cases the application stops being responsive.

The total time it takes to run the code fragment above is

$total\_time = LongOperationA_{time} + LongOperationB_{time} + LongOperationC_{time}$

The total completion time is the sum of the completion time of its components.

If we could start an operation without waiting for a previous operation to complete, we could utilize our resources much better, this is what **asynchronous execution** is for.

Asynchronous execution means that an operation is started, but its execution is happening in the background and the caller isn't blocked. Instead the caller is notified when the operation is completed. In that time the caller can continue to do useful work.

In the food ordering example, an asynchronous approach will be similar to sitting at the table and being served by a waiter.

First, you sit at the table, the waiter comes to hand you the menu and leaves. While you're deciding what to order the waiter is still available to other customers. When you decide what meal you want, the waiter comes back and takes your order. While the food is being prepared, you're free to chat, use your phone, or enjoy the view. You're not blocked (and neither is the waiter). When the food is ready, the waiter brings it to your table and goes back to serve other customers until you request the bill and pay.

This model is asynchronous, tasks are executed concurrently, and the time of execution is different from the time of the request, this way the resources (such as the waiter) are free to handle more requests.

Where the asynchronous execution happens

In a computer program, we can differentiate between two types of asynchronous operations, IO based and CPU based.

CPU-based operation means that the asynchronous code will run on another thread and the result will be returned when the execution on the other thread finishes.

IO-based operation means that the operation is made on an IO device such as a hard drive or network. If network is the case, a request was made to another machine (by using TCP or UDP or other network protocol) and when the operating system on your machine gets a signal from the network hardware by an interrupt that the result came back then the operation will be completed.

The calling thread in both of the cases is free to execute other tasks and handle other requests and event.

There's more than one way to run code asynchronously, and it depends on the language that's used. For now let's look at one example of doing asynchronous work using the .NET implementation of Futures – the `Task` class.

The asynchronous version of the code fragment above will look like the following code:

```
taskA=LongOperationAAsync();
taskB=LongOperationBAsync();
taskC=LongOpertionsCAsync();

Task.WaitAll(taskA, taskB, taskC).Wait()
```

In this version, each of the methods return a `Task<T>`. This class represents an operation that's being executed in the background. When each of the methods is called, the calling thread isn't blocked, and the method returns immediately and then the next method is called while the previous method is still executing. When all the methods are called we wait for their completion by using the `Task.WaitAll(…)` method that gets a collection of tasks and blocks until all of them are completed. Another way we could write this is:

```
taskA=LongOperationAAsync();
taskB=LongOperationBAsync();
taskC=LongOpertionsCAsync();

 taskA.Wait();
taskB.Wait();
taskC.Wait();
```

This way we get the same result, we wait for each of the tasks to complete (while they're still running in the background). If a task was already completed when we called the `Wait()` method, then it will return immediately.

The total time it takes to run the asynchronous version of the code fragment is shown as: *total_time = MAX(LongOperationA$_{time}$, LongOperationB$_{time}$, LongOperationC$_{time}$)*

Because all of the methods are running concurrently (and maybe even parallel) the time it takes to run the code will the time of the longest operation.

Asynchronicity and Rx

Asynchronous execution isn't limited to only being handled by using `Task<T>`.

Looking back at the Rx representation of time-variant variable – the `IObservable<T>` – we can use it to represent any asynchronous pattern, so when the asynchronous execution completes (successfully or with an error) the chain of execution will run and the dependencies will be evaluated. Rx provides methods for "casting" the different types of asynchronous execution (like `Task<T>`) to IObservable.

For example, in the Shoppy app, we want to get new discounts when our location changes. The call to the Shoppy webservice is done in an asynchronous way, and when it completes we want to update our view to show the new items.

```
IObservable<Connectivity> myConnectivity=...
IObservable<IEnumerable<Discount>> newDiscounts =
from connectiviy in myConnectivity
where connectiviy == Connectivity.Online
from discounts in GetDiscounts()//#A
select discounts;

newDiscounts.Subscribe(discounts => RefreshView(discounts));

private Task<IEnumerable<Discount>> GetDiscounts() {


   //Send request to the server and receives the collection of discounts

}
```

#A GetDiscounts() is returning a Task that is implicitly converted to an observable.

In this example we're reacting to the connectivity changes that are carried on the `myConnectivity` observable. Each time there's a change in connectivity, we check to see if it's because we're online and, if so, we call the asynchronous `GetDiscounts` method. When the method execution is complete, we select the result that was returned. This result is what will be pushed to the observers of the `newDiscounts` observable that was created from our code.

Events and streams

In a software system, an event is a type of message that's used to indicate that something has happened. The event might represent a technical occurrence--for example, in a GUI application, we might see events on each key that was pressed or mouse movements. The event can also represent a business occurrence such a money transaction that was completed in a financial system.

An event is raised by an **event source** and consumed by an **event handler**.

As we saw events are one way to represent time-variant values. And in Rx, the event source can be represented by the observable, and an event handler can be represented by the observer. But what about the simple data that our application is using, such as the one sitting in the database or fetched from a webserver? Does it have a place in the Reactive world?

TYPES OF DATA

The application you write will ultimately deal with some kind of data. Data can be of two types: data-in-motion and data-at-rest. Data-at-rest is data that's stored in a digital format and that you usually read from some persisted storage such as a database or files. Data-in-motion is data that's moving on the network (or other medium) and is being pushed to your application or pulled by your application from any external source.

Many technologies use events as a way to handle data-in-motion. Usually the application registers to a source and an event is raised when new data arrives or when new data is available to retrieve. Think of it like a mailbox that has a bell and is ringing whenever someone puts something inside the box.

When dealing with data in motion, it's easier to look at it as a stream of data (or stream of events), like a hose with data packets going through it, just like the one you see in Figure 2. When using a water hose, there are many things you can do with it, like putting filters at the

end, adding different hose-heads that give different functionality. You can add pressure monitors on the hose to help you regulate the flow. The same thing is what you would wish to do with your data stream. You'll want to build a pipeline that the flow through to eventually give an end result that suits your logic, this include filtering, transformations, grouping, merging, and so on.

Figure 2 Data stream is like a hose, every drop of water is a data packet that needs to go through different stations until it reaches the end. Your data as well needs to be filtered and transformed until it gets to the real handler that does something useful with it.

The data and event streams are a perfect fit for Rx observables; when abstracting them with an IObservable we get the possibility to make composition of the operators and create the complex pipeline of execution.