Lab 4 - **Futures and continuations**

Goal

The goal of this lab is to use C# TPL futures and continuations in a more complex scenario, in conjunction with waiting for external events.

Requirement

Write a program that is capable of simultaneously downloading several files through HTTP. Use directly the BeginConnect()/EndConnect(), BeginSend()/EndSend() and BeginReceive()/EndReceive() Socket functions, and write a simple parser for the HTTP protocol (it should be able only to get the header lines and to understand the *Content-lenght:* header line).

Try three implementations:

1. Directly implement the parser on the callbacks (event-driven);
2. Wrap the connect/send/receive operations in tasks, with the callback setting the result of the task;
3. Like the previous, but also use the async/await mechanism.

Direct Callbacks

Each method is given a method to call when it finished the execution, aka “back-to-back” execution.

Task Mechanism

The Task Parallel Library (TPL) is based on the concept of a task, which represents an asynchronous operation. In some ways, a task resembles a thread or ThreadPool work item, but at a higher level of abstraction. The term task parallelism refers to one or more independent tasks running concurrently. Tasks provide two primary benefits:

- More efficient and more scalable use of system resources.
  Behind the scenes, tasks are queued to the ThreadPool, which has been enhanced with algorithms that determine and adjust to the number of threads and that provide load balancing to maximize throughput. This makes tasks relatively lightweight, and you can create many of them to enable fine-grained parallelism.
- More programmatic control than is possible with a thread or work item.
  Tasks and the framework built around them provide a rich set of APIs that support waiting, cancellation, continuations, robust exception handling, detailed status, custom scheduling, and more.

For both of these reasons, TPL is the preferred API for writing multi-threaded, asynchronous, and parallel code in .NET.

Async/Await

The Task asynchronous programming model (TAP) provides an abstraction over asynchronous code. You write code as a sequence of statements, just like always. You can read that code as though each statement completes before the next begins. The compiler performs a number of transformations because some of those statements may start work and return a Task that represents the ongoing work.