**Lecture 5 - Futures with continuations**

This is an easier mechanism to work with, compared to the condition variables. Essentially, we have an object that exposes two interfaces:

* On the *promise* interface, a thread can, a single time, set a value. This actions also marks the completion of an activity (a *task*), and the value is the result of that computation.
* On the *future* interface, a thread can wait for the value to become available, and retrieve that value. Thus, a future is a result of some future computation.
* It is also possible to set a *continuation* on a future. This is done in the C# TPL (*Task Parallel Library*). The continuation is some computation that will be executed when the future will get a value (and the computation can use that value). It is also possible to set a continuation when all the futures in some set get values, or when any of the futures in some set gets a value.

Examples:

[***futures-demo1.cpp***](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/futures-demo1.cpp)

using futures to get the result from asynchronous tasks

[***futures-demo1-with-impl.cpp***](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/futures-demo1-with-impl.cpp)

as above, but with a possible implementation for futures and the async() call

[***futures-demo1.cs***](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/futures-demo1.cs)

as above, but in C#

[***futures-demo2-cascade1.cs***](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/futures-demo2-cascade1.cs)

cascading tasks through the ContinueWith() mechanism

[***futures-demo2-cascade2.cs***](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/futures-demo2-cascade2.cs)

same as above, but showing that the actual execution is de-coupled from the setup

[***futures-demo3-when-all.cs***](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/futures-demo3-when-all.cs)

using the WhenAll() mechanism to start a task only after all its input data is computed (by other tasks)

**Handling operations that depend on external events**

### Blocking calls

recv(sd, data, len); // blocks until data is available

... // process

send(sd, result, len);

...

* the current thread gets blocked in recv() / send() calls until the data is received / sent from / to the connection;
* the processing is easy to understand, since the current instruction and the execution stack contains the current state of the interaction with the single client;
* needs one thread for each client (or for each blocking operation to be executed in parallel)

### Event-driven, select()

while(true) {

select(nr, readFds, writeFds, exceptFds, nullptr);

if(FD\_ISSET(sd, readFds) {

recv(sd, data, len); // a single read is guaranteed not to block

...// process

}

}

...

* the only point where it blocks is in select();
* hard to combine differnet libraries, since everything that may have to wait for external events needs to be dispatched from the same central point;
* a single thread servers all clients;
* the state is harder to manage (event-driven);

### Event-driven, based on callbacks:

There is a begin...() call that initiates the operation and sets a callback that will be executed on completion. The begin...() operation returns immediatey an identifier of the asynchronous operation. The callback is called by the library when the operation completes. The callback (or someother thread) needs to call the corresponding end...() operation, that returns the results of the operation and frees the associated resources in the library.

class Receiver {

void Callback(IAsyncResult ar) {

int receivedBytes = sd.EndReceive(ar);

// process data

if(expectMoreData) {

sd.BeginReceive(m\_buf, m\_offset, m\_bufSize, 0, Callback, null);

}

}

void Start() {

// ...

sd.BeginReceive(m\_buf, m\_offset, m\_bufSize, 0, Callback, null);

}

}

...

A complete server implementation (for a very simple server) is given in [srv-begin-end.cs](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/srv-begin-end.cs).

Features:

* no need to explicitly create threads;
* easier to work with, compared to select() (no single central event loop and dispatcher), but still harder compared to one thread per client (still event-driven);
* need to know exactly what operations can be executed on which callbacks.

### Combining callbacks with futures and continuations

The idea is that the begin...() call is inside a function that returns a future, and the callback completes that future.

The previous server re-implemented using futures: [srv-task.cs](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/srv-task.cs). In this phase, the continuations are used as a way to trigger callbacks, like in the callbacks based implementation.

To have a nicer implementation, we need a mechanism to to better compose the asynchronous operations. Especially important is a mechanism to create a loop executing an asynchronous operation.

The previous server re-implemented again using the loop mechanism and composing asynchronous operations is [srv-tasks-loop.cs](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/srv-tasks-loop.cs).

As a note, an experimental framework for C++ for composing asynchronous operations is available at <https://github.com/rlupsa/futures-demo>. A more developed framework, including both futures with continuations and C++20 coroutines is available at <https://github.com/rlupsa/carpal>.

### The async-await mechanism

* The programmer writes the code (almost) like code-driven (not event-driven), with blocking like calls;
* The functions with pseudo-blocking calls are declared async and need to return futures; the called pseudo-blocking functions must be functions returning futures and the calls are marked with await;
* The compiler will generate, for the async function, a coroutine. The initial call goes up to the first await; at that point, it returns the control to the caller, returning a future as the result. Just before that, however, it calls the await function and enqueues a continuation to the resulting future. That continuation will execute the next part of the async calling function, up to either a return - that would complete the future - or to another await call - that would call that function and enqueue the next part of the caller as continuation.

The previous server re-implemented using *async-await*: [srv-await.cs](https://www.cs.ubbcluj.ro/~rlupsa/edu/pdp/progs/srv-await.cs).