STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXVIII**, Number 2, 2023
DOI: 10.24193/subbi.2023.2.03

ON COMPOSING ASYNCHRONOUS OPERATIONS

RADU LUPŞA AND DANA LUPŞA

Abstract. Asynchronous operations are very useful for actions that wait for an external event or work for a long time, to avoid blocking the thread that launches them. Unfortunately, whether they report their termination via callbacks or via completing a future, composing several asynchronous calls is difficult and error prone. The continuations mechanism (provided, for example, in C# Task Parallel Library via `ContinueWith()`) offers limited support for scheduling a sequence of operations. In this paper we try to improve this mechanism with better support for sequencing operations and exceptions, and with support for conditionals and loops, while covering the specifics of a C++ implementation. The most recent version of our source code is at [14].

1. Introduction

Asynchronous operations are operations that are started by a thread via a function call, but that continue after the initiating call returns.

They are essential in exploiting the parallelism, either between the CPU and the peripherals or external events, or between CPU cores.

Asyncronous communication and future objects are useful mechanisms to tolerate high latencies and improve overall performance. Automatic continuation and the mechanism used for updating result values can be used to further increase applicability and performance in different application and deployment scenarios [11] [15].

Research in the domain include: analysis of different strategies for updating future objects[15], proposal for architectures to support asynchronous messages using future objects while preserving the separation between the logic and the control aspects in the implementation [10], design for asynchronous

Received by the editors: 28 February 2023.

2010 *Mathematics Subject Classification.* 68N19, 68Q85.

1998 *CR Categories and Descriptors.* D.3.3 [**Software**]: Programming Languages – *Language Constructs and Features*; D.1.3 [**Software**]: Programming Techniques – *Language Concurrent Programming*; D.2.2 [**Software**]: Software Engineering – *Design Tools and Techniques* .

Key words and phrases. asynchronous programming.

stream generators, extending previous facilities [6], new framework to mimic simple synchronous programming but able to achieve fullflow processing asynchronously [4].

Combining several asynchronous operations to form a program is a difficult task. The goal of this paper is to study how to create patterns similar to structured programming. We try to improve the continuation mechanism with better support for sequencing operations and exceptions, and with support for conditionals and loops, while covering the specifics of a C++ implementation.

The rest of the paper is organised as follows: section 2 reviews the mechanisms available for getting the results for asynchronous operations. Section 3 discusses in more details the futures with continuation mechanisms. Section 4 describes the proposed framework allowing to combine simpler asynchronous operations into a more complex ones in a way similar to structured programming. The paper ends with conclusions.

2. Asynchronous operations result and chaininig

For any asynchronous operation, the caller needs to have some mechanism allowing it find out when the asynchronous operation has finished, and to retrieve any data produced by that asynchronous operation.

There are two basic mechanisms used by frameworks that offer asynchronous operations:

callbacks: When the asynchronous operation ends, a callback provided by the application is called.

futures: The call that initiates the asynchronous operation returns an object (a future or something that can be used as such) that can be polled by the application to find out if the asynchronous operation ended and, possibly, to wait until the operation ends.

2.1. Callbacks.

It should be noted that there are lots of ways in which various libraries offer the callback mechanism. For some, the callback is given as an argument to the function that starts the asynchronous operation, for others the callback is registered ahead of time to be called when any operation of a certain type completes. Also, for some libraries, the callback receives as an argument the result of the just finished asynchronous operation; for others, the callback is supposed to call some function to retrieve the asynchronous operation result and also to free any resources associated to the asynchronous operation.

Using callbacks is similar to *goto*-based programming. It is extremely flexible and can be used to build structures like sequence, *if-then-else*, or loops, but it is tedious and error-prone to use directly.

Another issue with callbacks is that they execute sometimes on some thread spawn by the library providing the asynchronous operations, and sometimes from within functions of that library called by the application. In either case, the thread on which the callback is called may hold some mutexes or may have to do some more work inside the library after the callback returns. As a result, there may be restrictions upon what library functions may be called from within the callback; calling forbidden functions may lead to the call being rejected, to a deadlock, or even to undefined behavior.

2.2. **Futures.** Futures were introduced by Liskov and Shrira [9]; they call them promises. They are available in most mainstream languages and recommended for asynchronous operations. Meyers [12] gives a very good explanation of the futures mechanism in C++11/14, while [5] discusses various future related issues and their approach in Kotlin.

Futures are better fit for structured programming mechanisms, and the main pattern is to start several asynchronous operations, that will then proceed in parallel, and then wait for all to finish and gather and using their results.

Having several operations executing one after another or one operation executing several times in a cycle requires however a thread that waits for the future corresponding to the previous operation and then calls the next one. This requires a thread that gets blocked.

3. Futures with continuations

The *futures with continuations* mechanism was introduced in C#/.NET Task Parallel Library (TPL) [3], and is also available in Boost, in a C++ standard proposal [1] or in the `stlab` library[8].

Extending the futures mechanism with continuations in C++ standard library is a debated issue. There are proponents, like [13], which proposes this as an extension to the basic C++ `std::future` mechanism. There are also opponents, like [7], which argue that `std::future` should remain a simple, wait-only type that serves a concrete purpose of synchronously waiting on potentially asynchronous work and they find that it should not be extended with continuations.

Basically, with futures with continuations, the asynchronous function returns a future that will complete when the asynchronous operation completes. However, beside the possibility to check whether the future has completed or to wait for its completion, the caller can also set a callback to be called when the future completes. The callback typically executes on some thread pool, usually called an *executor*.

The basic feature of the *future with continuation* mechanism is that it decouples the registering of the callback (the continuation) from the asynchronous

call itself. With the classical callback mechanism, the callback is supplied as an argument to the asynchronous call. With the *future with continuation*, the asynchronous call returns a future — called `Task` in C# TPL — which can be used in all 3 ways of getting the result from the asynchronous operation:

wait for completion: call `Wait()`,
poll: examine the `IsCompleted` property,
register a callback: call `ContinueWith()`.

Decoupling of the callback (continuation) from the asynchronous operation has several benefits.

First, the continuation is registered after the asynchronous operation is started, at any convenient time for the caller. The classical callback needs to be prepared beforehand and, in extreme cases, it might get executed even before the control returns from the call that initiates the asynchronous operation.

Second, the classical callback executes on a thread controlled by the framework providing the asynchronous operation. This usually poses some restrictions regarding which functions (especially, from the same framework) can be called from within the callback; disobeying those restrictions may lead to calls being rejected, deadlocks, or, in extreme cases, undefined behavior. The continuations, on the other side, execute on a thread in a thread pool provided by the *future with continuation* framework; thus, no restrictions exist with respect to which functions can be called from within the continuation.

Finally, continuations are more flexible, as their handling is independent of the asynchronous operation. It is possible to add multiple continuations to the same asynchronous operation (by calling `ContinueWith()` multiple times). It is also possible to add a continuation that is to be invoked when all operations from a set of asynchronous operations complete (by calling `WhenAll()`).

4. Composable asynchronous operations

The already existing mechanisms presented in the previous section evolved in a bottom-up manner, being provided *as-is*, and to be used as the programmer sees fit.

In this paper, we try a more systematic approach: we examine how to write an asynchronous function as a composition of smaller asynchronous functions. The construction should thus be similar to the way a classical, sequential, function is constructed by composing smaller functions.

So, our goal in this section is to create a framework allowing to compose asynchronous functions in the classical programming structures: sequence, conditional (*if-then-else*), and loop, as well as a *try-catch* mechanism. The result of composing asynchronous functions should be a new asynchronous function.

At the same time, we want not to lose the possibility of executing the asynchronous operations in parallel, when appropriate.

One trivial way to compose asynchronous functions is to treat them as synchronous: just call the function and then wait for the asynchronous operation to complete, blocking the execution of the current thread. This defeats the purpose of asynchronous operations. It needs to create a potentially large number of threads, and then, each time a thread blocks waiting for an asynchronous operation, it must switch to a new thread. Creation of a thread and switching from a thread to another are expensive operations because they involve going through an operating system call. So, instead of blocking threads, the finishing of each asynchronous operation needs to trigger a callback that would call the function that launches the next asynchronous operation.

Finally, the framework must be a pure library, without needing any special support from the programming language, such as coroutines. Coroutines are indeed very useful for describing an asyncronous function that calls other asynchronous function, and languages like C# and Python support this via the *async-await* mechanism: the asynchronous function is declared *async*, so that the compiler or interpreter knows it should be implemented as a coroutine, and, when the function calls some other asyncrounous function, its coroutine gets suspended until the called asynchronous operation completes, at which time the coroutine resumes. This allows the programmer to write code almost as if it were ordinary sequential code. However, some languages do not support coroutines and, even though recent C++20 does, there are systems where, for various reasons, upgrading to C++20 is not possible.

We choose C++ language for implementing the framework.

4.1. **Basic building blocks.** First, the basic building blocks will be asynchronous functions. Each asynchronous function will return a future, that completes when the asynchronous operation completes.

The future mechanism needs to allow the possibility to hook to a future a callback that gets executed when the future completes.

4.2. **Sequence.** In a sequence of asynchronous calls, each asynchronous operation would start after the previous one finishes. The full sequence becomes an asynchronous operation that completes when the last asynchronous operation of the sequence completes.

The basic support for a sequence is the future with continuation mechanism, described in section 3.

The C# `ContinueWith()` operation, or its equivalent `then()` in the C++ standard proposal, takes a future, representing the result of a first asynchronous operation, and a function and returns a future that will get the result of the second operation.

However, looking at the continuation enqueueing operation from the composability perspective, there are two aspects to be noticed.

First, the function for the second operation in the sequence takes as argument a future (a `Task`, in C#) instead of its value.

Second, the original `ContinueWith()` returns a future that completes when the function passed as argument returns. This works if the continuation is a synchronous function. However, if the function is asynchronous, the result of `ContinueWith()` is a future that completes when the second operation starts. The value of that future is a second (inner) future and its value is what the user code is interested in. C# provides a function called `Unwrap()` that creates and returns a future that completes when the inner future completes.

The C++ proposed `then()` function does the *unwrap* automatically, if the continuation function returns a future.

As an example, consider an asynchronous function that looks up in some database. For simplicity, let both the key and the value be of type `int`, and that we want the asynchronous equivalent of a synchronous code like `lookup(lookup(k))`. Then, the asynchronous version of lookup needs to be declared as

```
Task<int> AsyncLookup(int)
```

and the usage would be

```
Task<int> result = AsyncLookup(k).
    ContinueWith((Task<int> arg) => AsyncLookup(arg.Result)).
    Unwrap()
```

The last `Unwrap()` is needed because the future returned `ContinueWith()` completes when the second lookup starts. Its value is a second future, that completes when the second lookup completes, and the value of that inner future is the result of that second lookup — which is what we are interested in.

Java's `CompletableFuture` [2] offers two distinct functions for adding a continuation to a future, `thenApply()` and `thenCompose()`, the first behaving like C#'s `ContinueWith()` and the second like `ContinueWith()` followed by `Unwrap()`.

Standard C++ futures do not offer continuations, but there is a proposed `then()` function on a future that does an automatic `Unwrap()` if and only if the continuation function returns a future, thus being assumed to be an asynchronous function.

The equivalent implementation of the double lookup above using the C++ standard proposal would be:

```
std::experimental::future<int> result = AsyncLookup(k).
    then([](std::experimental::future<int> arg) {
        return AsyncLookup(arg.get());
    });
```

We propose here some small modifications that, while mostly cosmetic, emphasize on composability. Our continuation enqueueing operation is declared as:

```
template<typename R, typename Func, typename Arg>
Future<R>
        addAsyncContinuation(Executor& executor, Func func, Future<Arg> fArg)
```

Aside from the explicit specification of the thread pool used for executing the continuation (`executor`) and the fact that `addAsyncContinuation()` is a stand-alone function (not a class member), the differences are that `func` takes a simple value of type `Arg` (not `Future<Arg>`), and the returned `Future<R>` completes when the asynchronous operation completes.

The above example becomes:

```
Future<int> tmp = AsyncLookup(k);
Future<int> result = addAsyncContinuation<int>(executor, AsyncLookup, tmp);
```

4.3. **Conditional.** Implementing an *if-then-else* can be done in a straightforward way even without framework support. An asynchronous function implementing an *if-then-else* could have the following structure:

```
Future<int> conditional() {
  Future<int> f1 = foo();
  Future<int> f2 = addAsyncContinuation<int>(executor,
    [](int v) -> Future<int> {
      if(v>0) {
        return thenFunc(v);
      } else {
        return elseFunc(v);
      }
    }, f1);
}
```

4.4. **Loop.** Creating a loop around an asynchronous function is the main contribution of this paper. While the other structured programming constructs are relatively easy to obtain from the continuation enqueueing operation, creating a loop is much harder.

The need for loops arise in many places. For example, consider reading and parsing data coming via a connection. Suppose that reading bytes is provided as an asynchronous operation. Also, suppose that one needs to implement parsing as an asynchronous operation, that returns a parsed value (an integer, or some more complex object). To obtain that, the parsing operation would have a loop where it starts a read and, when the read completes, parses the read data and, if not complete, starts a new read and repeats.

As another example, handling a client from a server is also a loop where a (parsed) request is read, the response is sent, and the cycle repeats until the connection is closed or the request is to terminate the connection.

The basic loop construction needs a loop condition and a loop body.

Similarly with the case of the sequence, the body of the loop will be a function that launches an asynchronous operation and returns a future.

The asynchronous operation for each iteration is started after the asynchronous operation for the previous operation completes. Additionally, to pass information from one iteration to the next, the function acting as the loop body takes a value of some type and returns a future of the same type, whose value is passed to the loop body for the next iteration.

For the loop condition, we will use a synchronous function, taking as the value the value passed from one iteration to the next.

Putting all together, the result is a framework function declared as follows:

```
template<typename R, typename LoopFunc, typename PredicateFunc>
Future<R> executeAsyncLoop(Executor& executor, PredicateFunc loopingPredicate,
    LoopFunc loopFunc, R const& startValue)
```

The `startValue` argument is the initial value to be checked by the predicate function and to be passed to the loop body function for the first iteration. Consequently, `executeAsyncLoop()` creates an asynchronous function, taking a value — that is passed as the initial value for the loop body — and returning a future — that completes when the loop ends and receiving the returned value from the last iteration.

The following is a simple example of how a loop can be created. The function `delayedResult()` returns a future that is completed with the value given as the last argument after a time given as the second argument (1000) after it starts. The loop will thus count up to 10 with each step taking the given amount of time.

```
Future<int> f = executeAsyncLoop<int>(executor,
    [](int v)->bool {return v < 10;},
    [&alarmClock](int const& v)->Future<int> {
        return delayedResult(alarmClock, 1000, v + 1); },
    0);
```

To test in a slightly more realistic scenario, a small demonstrative server was implemented. The server repeatedly reads two integers (as sequences of digits) and responds with their sum. Below is a small excerpt that demonstrates the usage of the asynchronous loop:

```
Future<bool> executeOneRequest() {
  Future<int> fa = m_reader.readInt();
  Future<int> fb = addAsyncContinuation<int>(*m_pExecutor,
    [this](int a)->Future<int> {
      if(a > 0) return m_reader.readInt();
```

```
      return completedFuture<int>(0);
    }, fa);
    Future<bool> result = addAsyncContinuation<bool>(*m_pExecutor,
      [this,fa](int b) -> Future<bool> {
        if(fa.get() > 0) {
          int sum = fa.get() + b;
          std::shared_ptr<std::string> pSumStr =
                  std::make_shared<std::string>(std::to_string(sum) + "\n");
          return m_pSocket->send(pSumStr);
        } else {
          return completedFuture<bool>(false);
        }
    }, fb);
    return result;
}

Future<bool> run() {
  return executeAsyncLoop<bool>(*m_pExecutor,
    [](bool b){return b;},
    [this](bool b){return executeOneRequest();},
    true);
}
```

The the function `executeOneRequest()` launches an asynchronous operation that reads two integers over the socket and sends back their sum. It uses in turn other asynchronous functions for receiving and for sending data over the socket (`readInt()` and `send()`). The function immediately returns a boolean future that completes with *true* after the sum has been sent. The future completes with *false* if the client closes the connection or if an error occurs. Then, the `run()` function does the complete handling of a client: it returns a future that completes when the handling of the client is over (either because the client disconnects or because an error occurs.

The pattern solved by the `executeAsyncLoop()` is thus repeatedly calling an asynchronous operation that pulls data from a source (a connection, for instance) or pushes data into a sink.

Obtaining the same effect without `executeAsyncLoop()` is possible, but tedious. The implementation of the function `run()` from above, with C# TPL, would be (a helper function, `loopBody()`, is needed):

```
void loopBody(TaskCompletionSource result) {
  Task.Factory.StartNew(executeOneRequest)
    .ContinueWith((Task<bool> execResult) => {
      if(execResult.Result) loopBody(result);
      else result.SetResult();
    }
}
```

```
Task run() {
  TaskCompletionSource ret = new TaskCompletionSource();
  loopBody(ret);
  return ret;
}
```

One difficulty that should be noted about this example is that there are a lot of shared pointers. They are needed because of the asynchronous nature of the code. The actual functions usually only do some setup, so local variables will be long gone when the actual computation happens. Note that this is not a characteristic of the framework, but rather a common issue of asynchronous functions.

4.5. **Exceptions.** In regular programming, exceptions provide a mechanism for easily exiting from the structures.

Providing the same mechanism for asynchronous programming requires several elements which we will present below.

First, the futures can complete in two modes: with a value or with an exception.

Next, the sequence stops if a step completes with an exception. The next steps are skipped, but the sequence result completes with the same exception. Concretely, this means that, for `addAsyncContinuation()`, if the future given as argument completes with an exception, the returned future completes with that exception without the function argument being called.

Note that this behavior is quite distinct from what C# TPL is doing. In C#, if a `Task` completes with an exception and its continuations are set to execute only on normal completion, then the continuations resulting `Task`s complete as canceled. This means that, in order to get the exception, one needs to examine the `Task` corresponding to the failed operation; subsequent `Task`s have no information on the exception.

Similarly to `addAsyncContinuation()`, for `executeAsyncLoop()`, if the body completes with an exception, the loop stops and the future returned by `executeAsyncLoop()` completes with that exception.

Finally, the equivalent of the *try-catch* mechanism is also needed. Our framework provides a function declared as

```
template<typename T, typename CatchFunc>
Future<T>
        catchAsync(Executor& executor, CatchFunc catchFunc, Future<T> value);
```

Its `catchFunc` argument must be an asynchronous function taking an `std::exception_ptr` and returning `Future<T>` and gets the role of the *catch* block. The semantic of `catchAsync()` is the following:

- It immediately returns a future;

- If `value` completes normally, the future returned from `catchAsync()` completes with the same value;
- If `value` completes with an exception, `catchFunc()` is called with that exception as argument and the future returned by `catchAsync()` will complete with the value (or exception) returned (respectively thrown) by `catchFunc()`.

Using exceptions, the small server of the previous section can be re-written in a simpler way, eliminating the repeated checks that the state of handling the client is still normal, and instead relying on the "fast exit" mechanism of the exceptions:

```
Future<bool> executeOneRequest() {
    Future<int> fa = m_reader.readInt();
    Future<int> fb = addAsyncContinuation<int>(*m_pExecutor,
        [this](int a)->Future<int> {
            if(a < 0) {
                throw Flag::client_disconnected;
            }
            return m_reader.readInt();
        }, fa);
    Future<bool> result = addAsyncContinuation<bool>(*m_pExecutor,
        [this,fa](int b) -> Future<bool> {
            if(b < 0) {
                throw Flag::invalid_input;
            }
            int sum = fa.get() + b;
            std::shared_ptr<std::string> pSumStr =
                 std::make_shared<std::string>(std::to_string(sum)+"\n");
            return m_pSocket->send(pSumStr);
        }, fb);
    return result;
}
```

5. Conclusions

We developed, and presented here, a framework for using the futures with continuations mechanism in a way very similar to the classical, structured style, programming. This way, programs using asynchronous calls look reasonably similar to regular programs, and this is probably the best that can be achieved without language support like coroutines. Its central point is the asynchronous loop mechanism.

As a limitation, the lifetimes of the variables are not very obvious, and shared pointers are required almost everywhere because of this. This is not a limitation of the framework per se, but a result of the asynchronous work

model. A study of possible improvements in this area may come as a future work.

Yet another future direction would be to attempt to use the looping mechanism to produce or to consume a stream of values, in the reactive programming style.

References

[1] C++ reference. extensions for concurrency. `https://en.cppreference.com/w/cpp/experimental/future/then`. Accessed: 2023.

[2] `Java Platform, Standard Edition 8 API Specification, CompletableFuture` . `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html`. Accessed: 2023.

[3] David Pine, e. a. Task parallel library (tpl). `https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl`, 2022. Accessed: 2022.

[4] Duan, J., Yi, X., Wang, J., Wu, C., and Le, F. Netstar: A future/promise framework for asynchronous network functions. *IEEE Journal on Selected Areas in Communications 37*, 3 (2019), 600–612.

[5] Elizarov, R., Belyaev, M., Akhin, M., and Usmanov, I. Kotlin coroutines: Design and implementation. Onward! 2021, Association for Computing Machinery, p. 68–84.

[6] Haller, P., and Miller, H. A reduction semantics for direct-style asynchronous observables. *Journal of Logical and Algebraic Methods in Programming 105* (03 2019).

[7] Howes, L., Grynenko, A., and Feldblum, J. Continuations without overcomplicating the future. `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0783r0.html`, 2017. Accessed: 2022.

[8] Lab, A. S. T. stlab: Api documentation. futures. `https://stlab.cc/libraries/concurrency/future/`. Accessed: 2023.

[9] Liskov, B., and Shrira, L. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1988), PLDI '88, Association for Computing Machinery, p. 260–267.

[10] Manolescu, D. A. Workflow enactment with continuation and future objects. 40–51.

[11] Marshall Cline, e. a. A unified futures proposal for c++. `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1054r0.html`, 2018. Accessed: 2022.

[12] Meyers, S. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*, 1st ed. O'Reilly Media, Inc., 2014.

[13] N., G., A., L., H., S., and S., M. A standardized representation of asynchronous operations, tecnical report n3538. `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3558.pdf`, 2013. Accessed: 2022.

[14] Radu, L. futures-demo `https://github.com/rlupsa/futures-demo`, 2023.

[15] Ranaldo, N., and Zimeo, E. Analysis of different future objects update strategies in `ProActive`. In *2007 IEEE International Parallel and Distributed Processing Symposium* (2007), pp. 1–7.

Computer Science Department, Babeş Bolyai University, Cluj-Napoca, Romania

Email address: `radu.lupsa@ubbcluj.ro, dana.lupsa@ubbcluj.ro`