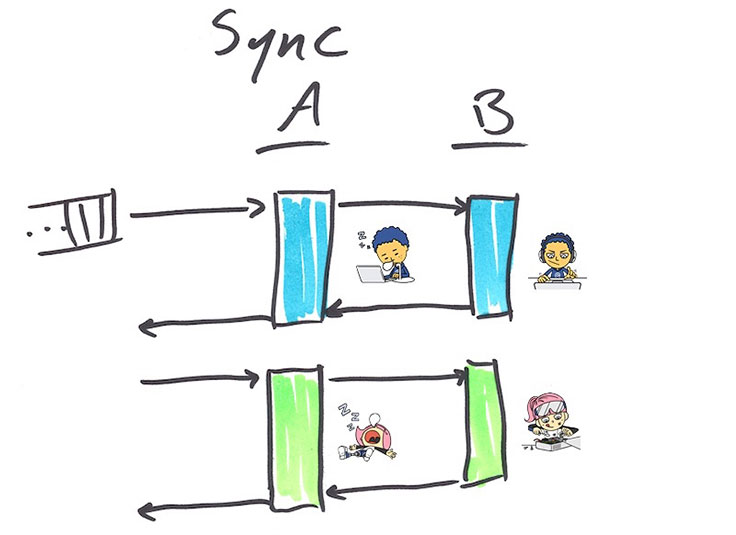
Futures for C++11 at Facebook

[https://fb-s-d-a.akamaihd.net/h-ak-fbx/v/t1.0-1/c34.0.320.320/s32x32/11403165_10204572813019805_550684040030943819_n.jpg?oh=d27c8c720838055e03d08f246d1217b1&oe=59F03CED&__gda__=1509284056_e2dab35771cef4ab15c67185a2f0d214Hans Fugal](https://www.facebook.com/fugalh)

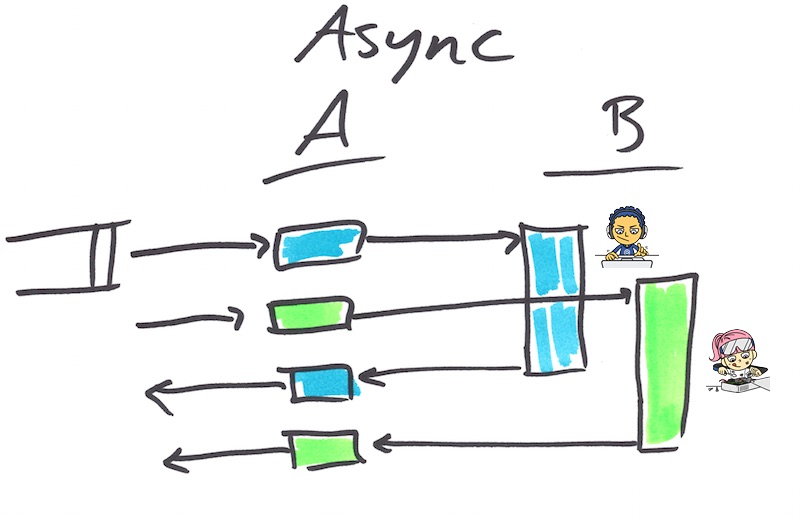
Futures are a pattern for expressing asynchronous computations in a natural and composable way. This blog post introduces [**Folly Futures**](https://l.facebook.com/l.php?u=https%3A%2F%2Fgithub.com%2Ffacebook%2Ffolly%2Ftree%2Fmaster%2Ffolly%2Ffutures&h=ATNm_Ckha6aRDPldJAY2Z4TO39vkMt0mbx59Vc9u0hkXjH6ZdiA_Hn-r0Ta4EuBpN_UaqdBfg5JnuGW3DYNjwJfV-W0ZBBapcL4hgrXXsnDuXKVfBvavxYmLD7asgW9ERA&s=1), an implementation of futures for C++11 that we use at Facebook.

Why asynchrony?

Consider a service *A* that talks to another service *B*. If *A* blocks while waiting for a reply from *B*, then *A* is *synchronous*. A blocked thread is idle; it cannot service other requests. Threads are heavyweight — switching threads is inefficient, they have considerable memory overhead, and the OS will bog down if you make too many of them. The result is wasted resources, reduced throughput, and increased latency (because requests are in a queue, waiting to be serviced).



It is more efficient to make service *A asynchronous*, meaning that while *B* is busy computing its answer, *A* has moved on to service other requests. When the answer from *B* becomes available, *A*will use it to finish the request.



Synchronous code and asynchronous code compared

Let's consider a function fooSync that does a basic computation foo fully synchronously, and a function fooAsync that does the same work asynchronously. fooAsync takes both an input and a *callback*, which is called once the answer is available.

template <typename T> using Callback = std::function<void(T)>;

Output fooSync(Input);

void fooAsync(Input, Callback<Output>);

This is the traditional way of expressing asynchronous computation. (Older C/C++ asynchronous libraries would take a function pointer and a void\* context argument, but now that C++11 supports closures, an explicit context argument is not required.)

Traditional asynchronous code is more efficient than synchronous code, but it is not as easy to read. Compare the synchronous and asynchronous versions of a function that implements a multiFoocomputation, which does foo for every element in an input vector:

using std::vector;

vector<Output> multiFooSync(vector<Input> inputs) {

vector<Output> outputs;

for (auto input : inputs) {

outputs.push\_back(fooSync(input));

}

return outputs;

}

void multiFooAsync(

vector<Input> inputs,

Callback<vector<Output>> callback)

{

struct Context {

vector<Output> outputs;

std::mutex lock;

size\_t remaining;

};

auto context = std::make\_shared<Context>();

context->remaining = inputs.size();

for (auto input : inputs) {

fooAsync(

input,

[=](Output output) {

std::lock\_guard<std::mutex> guard(context->lock);

context->outputs->push\_back(output);

if (--context->remaining == 0) {

callback(context->outputs);

}

});

}

}

The asynchronous version is much more complicated. It has to worry about setting up a shared context object, thread safety, and bookkeeping, so it will know when the overall computation is complete. Even worse, though it's not obvious in this example, is that when the computation graph becomes complex, it becomes very difficult to follow the execution path. Programmers need to build up a mental model of the whole service state machine and how it acts with different inputs, and computation flow is not expressed in one place in the code where it can be reviewed. This state of affairs is affectionately referred to as "callback hell."

Futures

A *future* is an object that represents the result of an asynchronous computation, which may not yet be available. When the computation is complete, the future may hold either a value or an exception. As an example:

#include <folly/futures/Future.h>

using folly::Future;

// Do foo asynchronously; immediately return a Future for the output

Future<Output> fooFuture(Input);

Future<Output> f = fooFuture(input);

// f may not have a value (or exception) yet. But eventually it will.

f.isReady(); // Maybe, maybe not.

f.wait(); // You can synchronously wait for futures to become ready.

f.isReady(); // Now this is guaranteed to be true.

Output o = f.value(); // If f holds an exception, this will throw that exception.

So far we haven't done anything [**std::future**](https://l.facebook.com/l.php?u=http%3A%2F%2Fen.cppreference.com%2Fw%2Fcpp%2Fthread%2Ffuture&h=ATOs8BMXkK5fLG2BMxU8PPSiN1SSb7APkldy2mfY3Cv-tDNu-L2K1IhgGjTyglCl2ifXzoHoLjB8-d2yw9Qym_i9KhrcKkfwwVIekfTFd96c-RrAckT0TTTPvC7HiVMMtg&s=1) can't do. But a powerful aspect of the future pattern is being able to chain callbacks, which std::future doesn't yet support. We express this with the method [**Future::then**](https://l.facebook.com/l.php?u=https%3A%2F%2Fgithub.com%2Ffacebook%2Ffolly%2Fblob%2Fmaster%2Ffolly%2Ffutures%2FREADME.md%23setting-callbacks-with-then-&h=ATNzt2Yv6Dty3E8xDojBLh-LpvaQA5guWcToYQa-e5hWGr-uVSA5Bp12eFAsxPlGCqY2wYKCMcPqcyWCm1r6P_DSLJggyycgROirT4fECRIsX19HMEi-_As6DeES7t-XEw&s=1):

Future<double> f =

fooFuture(input)

.then([](Output o) {

return o \* M\_PI;

})

.onError([](std::exception const& e) {

cerr << "Oh bother, " << e.what()

<< ". Returning pi instead." << endl;

return M\_PI;

});

// get() first waits, and then returns the value

cout << "Result: " << f.get() << endl;

Here we chained then as well as [**onError**](https://l.facebook.com/l.php?u=https%3A%2F%2Fgithub.com%2Ffacebook%2Ffolly%2Fblob%2Fmaster%2Ffolly%2Ffutures%2FREADME.md%23using-onerror-&h=ATOtHrtgfn-EfhiBG36i4w8V5aPzshmAuq14nfw7o3SggLApvBGpmMYVlyUHm6R11IK2-JIKDpzO1osQLaEfmJS_XF_jXVKqViM3YSk0mN8bhkJNKCYmuUvrA9jXA8e2eg&s=1) to handle any exception that may have been generated. Being able to chain futures turns out to be an important capability. It allows us to compose sequential and parallel computations, express them in one place, and provide clean error handling.

Sequential composition

If you want to do computations a, b, c, and d asynchronously in that order, with traditional callback programming you end up with callback hell — or, in a language with first-class anonymous functions (like C++11), perhaps the “callback pyramid”:

// the callback pyramid is syntactically annoying

void asyncA(Output, Callback<OutputA>);

void asyncB(OutputA, Callback<OutputB>);

void asyncC(OutputB, Callback<OutputC>);

void asyncD(OutputC, Callback<OutputD>);

auto result = std::make\_shared<double>();

fooAsync(input, [=](Output output) {

// ...

asyncA(output, [=](OutputA outputA) {

// ...

asyncB(outputA, [=](OutputB outputB) {

// ...

asyncC(outputB, [=](OutputC outputC) {

// ...

asyncD(outputC, [=](OutputD outputD) {

\*result = outputD \* M\_PI;

});

});

});

});

});

// As an exercise for the masochistic reader, express the same thing without

// lambdas. The result is called callback hell.

With futures, sequentially composed using then, the code cleans up nicely:

Future<OutputA> futureA(Output);

Future<OutputB> futureB(OutputA);

Future<OutputC> futureC(OutputB);

// then() automatically lifts values (and exceptions) into a Future.

OutputD d(OutputC) {

if (somethingExceptional) throw anException;

return OutputD();

}

Future<double> fut =

fooFuture(input)

.then(futureA)

.then(futureB)

.then(futureC)

.then(d)

.then([](OutputD outputD) { // lambdas are ok too

return outputD \* M\_PI;

});

Parallel composition

Let's go back to our multiFoo example. Here's what it looks like with futures:

using folly::futures::collect;

Future<vector<Output>> multiFooFuture(vector<Input> inputs) {

vector<Future<Output>> futures;

for (auto input : inputs) {

futures.push\_back(fooFuture(input));

}

return collect(futures);

}

[**collect**](https://l.facebook.com/l.php?u=https%3A%2F%2Fgithub.com%2Ffacebook%2Ffolly%2Fblob%2Fmaster%2Ffolly%2Ffutures%2FREADME.md%23collect-&h=ATMY1i_uEvKo95DOljpGnS8Pt5OL74CbCSSee-0bOIt63vdxLkVAhJWjGmJvtPP6BZ1ukFEPV16sMR0_yyJhIaQUwLDBIhKUuVNCda9ubAMIdQ8ylhK1kpro23G4ZgZaKA&s=1) is one of the compositional building blocks we provide, which takes a collection of Future<T> and returns a Future<vector<T>> that will complete when all those futures are complete. (The implementation of collect relies on — you guessed it — then.) There are many other [**compositional building blocks**](https://l.facebook.com/l.php?u=https%3A%2F%2Fgithub.com%2Ffacebook%2Ffolly%2Fblob%2Fmaster%2Ffolly%2Ffutures%2FREADME.md%23compositional-building-blocks&h=ATOOLG9-ERofgXlNzyA80wV8Mv1uZeg8kCd-EVzmCfWQrQh22vjjVw3llJmbp-kSYEgCIPnDdjznnhmoc5_L0wy9wNZrrQQQwzylrm_ezhoAw8VRvCP7gRSdzkroLAdN0A&s=1), including collectAny, collectN, map, and reduce.

Notice how this code looks very similar to the synchronous version multiFooSync, above. We don't need to worry about passing context around or thread safety; it is all handled transparently for us by the framework.

Execution context

Some futures frameworks in other languages provide a thread pool for executing callbacks, so you don't need to worry about the execution context other than to know that it's in some other thread. But C++ developers tend to be writing C++ because they need to control low-level details for performance optimization, and this is certainly true at Facebook. So we provide a flexible mechanism for explicitly controlling the execution context of callbacks, using a simple [**Executor interface**](https://l.facebook.com/l.php?u=https%3A%2F%2Fgithub.com%2Ffacebook%2Ffolly%2Fblob%2Fmaster%2Ffolly%2FExecutor.h&h=ATONLLIDwYeF_NDa54p4y6pkTO-xurJW-OFgCmtoQyv8YJFVAVwG0SGL6VV-F02gcf2MCPklckQ4pEgoMYeuJRc3jCA_Qqm0YgpwgScq_-rimaF_in_6fLPFMoGwTPj5Kw&s=1):

struct Executor {

using Func = std::function<void()>;

virtual void add(Func) = 0;

};

You can pass an executor to then to dictate that its callback will execute via that executor.

a(input).then(executor, b);

In this code, b will be executed via executor, which might be a specific thread, a thread pool, or [**something even more intriguing**](https://l.facebook.com/l.php?u=https%3A%2F%2Fgithub.com%2Ffacebook%2Ffolly%2Fblob%2Fmaster%2Ffolly%2Ffutures%2FREADME.md%23executor-implementations&h=ATNf410h5pt5YaSRckBh5btuaN77m6H7CzLxcVZ2wINdrxQpqvaY3FKxWs4ZE9QeabvAqMVBQ0xgAIIddDeDmdpxRvwt-jmf6-eAgHazRVELDe_Fix5dQ4nZwf7E-qsiPA&s=1). A common use case for this is to move significant CPU work off of the I/O thread to avoid queueing delay for other requests.

Futures mean never having to forget you're sorry

One pervasive problem with traditional callback code is that it is extremely easy to lose track of errors or exceptional conditions. Programmers must exercise great (even superhuman) discipline to check for errors and take appropriate action, to say nothing of what happens when an unexpected exception is thrown. Futures address this problem by containing either a value or an exception, and exceptions behave like you would expect exceptions to when composing futures, except that it remains within the future monad until handled with onError or released synchronously with, for example, value or get. It's harder (though not impossible) to lose track of an error that should be handled.

You make me promises, promises

We've seen an overview of how to use futures, but how do we make them? If you need to lift a value into Future, use makeFuture:

using folly::makeFuture;

std::runtime\_error greatScott("Great Scott!");

Future<double> future = makeFuture(1.21e9);

Future<double> future = makeFuture<double>(greatScott);

But if you are wrapping an asynchronous operation, you use a Promise:

using folly::Promise;

Promise<double> promise;

Future<double> future = promise.getFuture();

When you're ready to fulfill the promise, use setValue, setException, or setWith:

promise.setValue(1.21e9);

promise.setException(greatScott);

promise.setWith([]{

if (year == 1955 || year == 1885) throw greatScott;

return 1.21e9;

});

All together, then, making a long-running synchronous operation asynchronous by spawning another thread might look like this:

double getEnergySync(int year) {

auto reactor = ReactorFactory::getReactor(year);

if (!reactor) // It must be 1955 or 1885

throw greatScott;

return reactor->getGigawatts(1.21);

}

Future<double> getEnergy(int year) {

auto promise = make\_shared<Promise<double>>();

std::thread([=]{

promise->setWith(std::bind(getEnergySync, year));

}).detach();

return promise->getFuture();

}

You often don't need promises, even if at first blush it seems like you do. For example, if you already have an executor for your thread pool or can get your hands on one easily, it's easier to just do this:

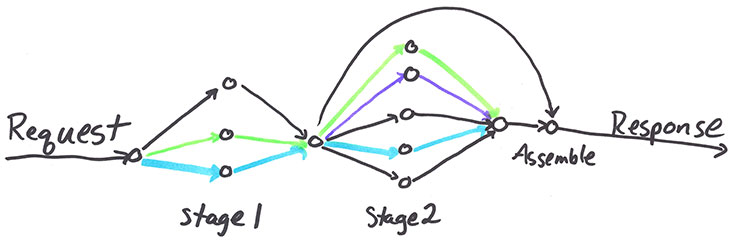
Future<double> future = folly::via(executor, std::bind(getEnergySync, year));

Case studies

We offer two case studies for how using futures has improved latency, robustness, and code readability at Facebook and Instagram.

Instagram improved the infrastructure for their recommendation service by converting it from synchronous to asynchronous, using futures. The result was a significant drop in tail latencies, and tenfold fewer servers required to achieve the same throughput. They describe the changes they made and the benefits in more detail in their [**blog post**](https://l.facebook.com/l.php?u=http%3A%2F%2Finstagram-engineering.tumblr.com%2Fpost%2F121930298932%2Fc-futures-at-instagram&h=ATMGfmiZpsL3C5p7MVc3f215p29JKWVdAOUQPaA-68oHrRN0h02QVCD9zRpT0numaFYSAsYpL3pz76uZdqHn894lppuwxjtntjOSZiSwiAUA3MAfC_Ln2zyOPtDQLWXlXg&s=1).

The next case study is a real service that is one piece of constructing the Facebook News Feed. It has a two-stage leaf-aggregate pattern, where a request is broken into many leaf requests that shard to different leaf servers, and then we do the same thing but shard differently based on the results of the first aggregate. Finally, we take both sets of results and reduce them to a single response.



Here is a simplified version of the relevant code:

Future<vector<LeafResponse>> fanout(

const map<Leaf, LeafReq>& leafToReqMap,

chrono::milliseconds timeout)

{

vector<Future<LeafResponse>> leafFutures;

for (const auto& kv : leafToReqMap) {

const auto& leaf = kv.first;

const auto& leafReq = kv.second;

leafFutures.push\_back(

// Get the client for this leaf and do the async RPC

getClient(leaf)->futureLeafRPC(leafReq)

// If the request times out, use an empty response and move on.

.onTimeout(timeout, [=] { return LeafResponse(); })

// If there's an error (e.g. RPC exception),

// use an empty response and move on.

.onError([=](const exception& e) { return LeafResponse(); }));

}

// Collect all the individual leaf requests into one Future

return collect(leafFutures);

}

// Some sharding function; possibly dependent on previous responses.

map<Leaf, LeafReq> buildLeafToReqMap(

const Request& request,

const vector<LeafResponse>& responses);

// This function assembles our final response.

Response assembleResponse(

const Request& request,

const vector<LeafResponse>& firstFanoutResponses,

const vector<LeafResponse>& secondFanoutResponses);

Future<Response> twoStageFanout(shared\_ptr<Request> request) {

// Stage 1: first fanout

return fanout(buildLeafToReqMap(\*request, {}),

FIRST\_FANOUT\_TIMEOUT\_MS)

// Stage 2: With the first fanout completed, initiate the second fanout.

.then([=](vector<LeafResponse>& responses) {

auto firstFanoutResponses =

std::make\_shared<vector<LeafResponse>>(std::move(responses));

// This time, sharding is dependent on the first fanout.

return fanout(buildLeafToReqMap(\*request, \*firstFanoutResponses),

SECOND\_FANOUT\_TIMEOUT\_MS)

// Stage 3: Assemble and return the final response.

.then([=](const vector<LeafResponse>& secondFanoutResponses) {

return assembleResponse(\*request, \*firstFanoutResponses, secondFanoutResponses);

});

});

}

The legacy version of this service was utilizing an asynchronous framework that allowed only an overall timeout and used the traditional "callback hell" style. Futures made it natural to express the asynchronous computation and use granular timeouts to take more proactive action when some piece is slow. As a result, the service's average latency was reduced by two-thirds, and tail latencies were reduced tenfold. Overall timeout errors were reduced significantly. The code is much easier to read and reason about, and as a result, it is more maintainable.

When developers have the tools to understand and express asynchrony better, they write lower-latency services that are easier to maintain.

Conclusion

Folly Futures bring robust, powerful, and performant futures to C++11. We hope that you enjoy using them as much as we do. For more information, consult the [**documentation**](https://l.facebook.com/l.php?u=https%3A%2F%2Fgithub.com%2Ffacebook%2Ffolly%2Fblob%2Fmaster%2Ffolly%2Ffutures%2FREADME.md&h=ATMbjiOolr1XIHVv0CqPe_Ua-cYLe3-XEooMJghEyRndBGPtI915R4T0LONUCW07Y-pANaVmwso_IXB9Rh521jNXXfK32alVVdUz1Mtbt8MMhPsr9acA8UJgRfo3DqzPXw&s=1), [**docblocks, and code**](https://l.facebook.com/l.php?u=https%3A%2F%2Fgithub.com%2Ffacebook%2Ffolly%2Ftree%2Fmaster%2Ffolly%2Ffutures&h=ATMwWcZPu99eDRfuzpyY6siiL2R_S6RUbG3HldjQKwPx0eWwYMoNYPfTFDEcruQJPFNW03bzDQf9B8cx6ugKTINOXDJEdqtaQoOnSygD4DzFsNNr22vOFALJV4KT_7DEgA&s=1)on GitHub.

Acknowledgements

The team that wrangled Folly Futures into existence is Hans Fugal, Dave Watson, James Sedgwick, Hannes Roth, and Blake Matheny, with contributions from many others along the way. We would like to thank Twitter and particularly [**Marius**](https://l.facebook.com/l.php?u=http%3A%2F%2Fmonkey.org%2F%7Emarius%2F&h=ATNX9YlWpYpWWz6ePv5UK2EXdWqRuDhxU2T0h80p_-q_YyufdGjSSEfHyF1ZluQTnAkIGQOH_AX0FOEgkYr5mBlJFWqh4doRDLQhXPiUxfuRWqMVWxXkt_cOqPGGVPfWhg&s=1), whose tech talk at Facebook on [**Finagle**](https://l.facebook.com/l.php?u=http%3A%2F%2Ftwitter.github.io%2Ffinagle%2F&h=ATP7nCMlNpBmXnwmu35kvAhFr7ypP6QindTrJ1T6zZ9iJ-Za_JyC-_CxPY0tnWZHnmD9_HBuJ1MTondT2SLwei850XLyIBmz_6xhQmkuee7Z_L_KAawhLuNiFcxsCmBY5g&s=1) and [**Futures**](https://l.facebook.com/l.php?u=http%3A%2F%2Ftwitter.github.io%2Ffinagle%2Fguide%2FFutures.html&h=ATMtUMtnmKpQJG1yKa26TmVR46D6O6vluaeOCJi0qcIdtA9gsRK8XhRPsVgOGG2Lrsm4XdvuJmJN6Gq1j2lT2rsd7z8g0YL5lYjweKSYtnPJicKY30l4GCcYcxR1Zs5A9w&s=1)inspired the project.