Boost Asynchronous

Christophe Henry

Boost Asynchronous

Christophe Henry

Table of Contents

List of Tables

Preface

Note: Asynchronous is not part of the Boost library. It is planed to be offered for Review in 2014. At the moment it is still in development.

Herb Sutter wrote in an article [http://www.gotw.ca/publications/concurrency-ddj.htm] "The Free Lunch Is Over", meaning that developpers will be forced to learn to develop multi-threaded applications. This, however, brings a fundamental issue: multithreading is hard, it's full of ugly beasts waiting hidden for our mistakes: races, deadlocks, all kinds of subtle bugs. Worse yet, these bugs are hard to find because they are never reproducible when you are looking for them, which leaves us with backtrace analysis, and this is when we are lucky enough to have a backtrace in the first place.

This is not even the only danger. CPUs are a magnitude faster than memory, I/O operations, network communications, which all stall our programms and degrade our performance, which means long sessions with coverage or analysis tools.

Well, maybe the free lunch is not completely over yet, or at least maybe we can still get one a bit longer for a bargain. This is what Boost Asynchronous is helping solve.

Boost Asynchronous is a library making it easy to write asynchronous code similar to the Proactor pattern [http://www.cs.wustl.edu/~schmidt/PDF/proactor.pdf]. To achieve this, it offers tools for asynchronous designs: ActiveObject, threadpools, servants, proxies, queues, algorithms, etc.

Asynchronous programming has the advantage of making it easier to design your code nonblocking, single-threaded while still getting your cores to work at full capacity. And all this while forgetting what a mutex is. Incorrect mutex usage is a huge source of bugs and Asynchronous helps you avoid them.

However, the goal of this library is not to help you write massively parallel code. There are other solutions for this. This library is for the other 99% of us who happen to work on 4-12 cores hardware because 1000 cores are not affordable, but would still like to get the best of it while avoiding ugly bugs, get better diagnostic of what our application is doing and planing ourselves what our cores are doing. Asynchronous is not:

- std/boost::async: both are asynchronous in a very limited way. One posts asynchronously some work, but then? Well, then the choice is between blocking for the future result (taboo) or polling from time to time, as in the (bad) old times. Furthermore, one has no control on the scheduler.

- Intel TBB: this is a wonderful parallel library. But it's not asynchronous as one needs to wait for the end of a parallel_xxx call. Sure, you have pipelines, but when your application is complex, good luck to understand the code. Again, one has limited control on the scheduler.

- N3428: this is an interesting approach as it was at least recognized that std::async is not asynchronous. So now, you get a kind of state machine hidden behind a .then, when_any, when_all, which are a poor man's state machines, besides ugly limitations like finding out which when_all threw an exception. When did we give up writing design diagrams to document and understand our code later? When a 15+ states state machine with guards, event deferring and control flow has to be written with .then, please don't ask us to review the code.

Let's have a quick look at code using futures and .then (taken from N3428):

```
future<int> f1 = async([]() { return 123; });
future<string> f2 = f1.then([](future<int> f) {return f.get().to_string();}); /
f2.get(); // just a "small get" at the end?
```

Saying that there is only a "small get" at the end is, for an application with real-time constraints, equivalent to saying at a lockfree conference something like "what is all the fuss about? Can't we just add a small lock at the end?". Just try it...

This brings us to a central point of Asynchronous: if we build a system with strict real-time constraints, there is no such thing as a small get(). We need to be able to react to any event in the system in a timely manner. And we can't afford to have lots of functions potentially waiting too long everywhere in our code. Therefore, .then is only good for an application of a few hundreds of lines. What about using a timed_wait instead? Nope. Either we wait too long before handling an error (this just limits the amount of time we waste waiting, that's all), or we wait not enough and we poll. In any case, while waiting, our thread cannot react to other events.

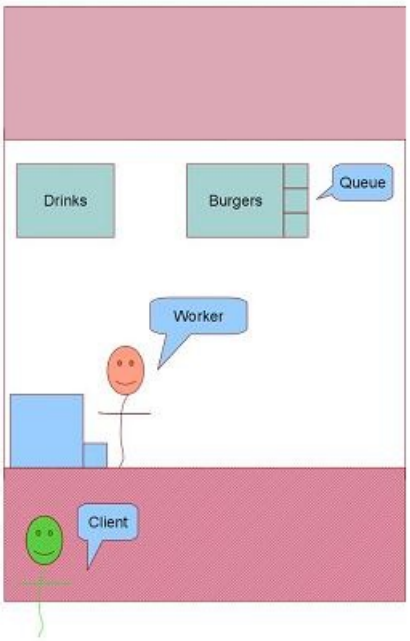
All these libraries also have the disadvantage of working with functions, not classes. But we, normal developers, do use classes. And we want them safe. And you can hardly make a class safe if you simply pass it to another thread. Consider the following example:

```
struct Bad : public boost::signals::trackable
{
   int foo();
};
boost::shared_ptr<Bad> b;
future<int> f = async([b](){return b->foo()});
```

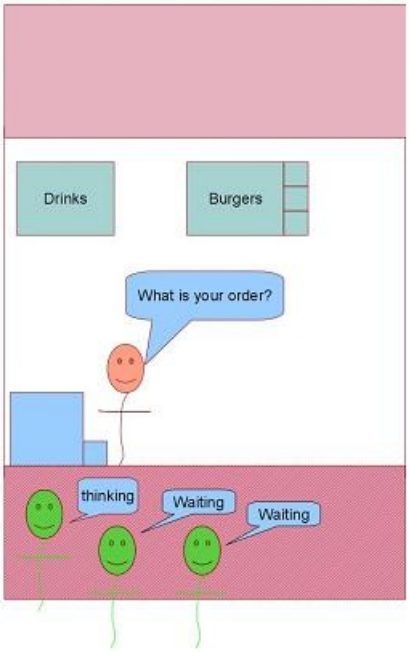
Now you have the ugly problem of not knowing in which thread Bad will be destroyed. And as it's pretty hard to have a thread-safe destructor, you find yourself with a race condition in it. Maybe you'll find a solution for signals, but are you sure nobody is ever going to do something dangerous in the destructor? This clearly is not thinking in the future sense.

Another particularity of Asynchronous is that it's not hiding anything: one gets to pick a pool for a particular application, choose how many threads are to be used, and the type of queue which best corresponds to the job to do, which allows finer tuning.

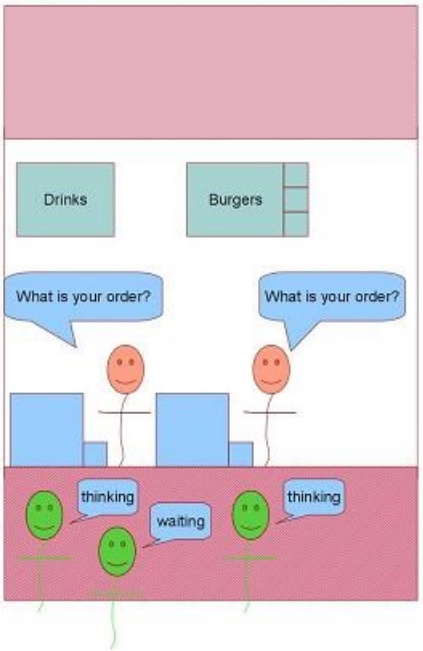
An image being more worth than thousand words, the following story will explain in a few minutes what Asynchronous is about. Consider some fast-food restaurant:



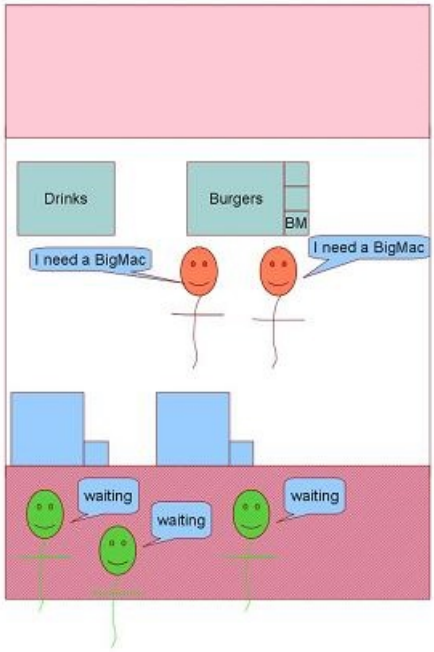
This restaurant has a single employee, Worker, who delivers burgers through a burger queue and drinks. A Customer comes. Then another, who waits until the first customer is served.



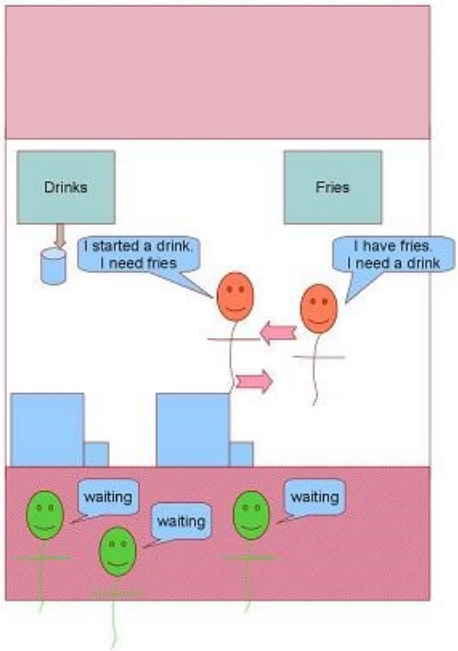
To keep customers happy by reducing waiting time, the restaurant owner hires a second employee:



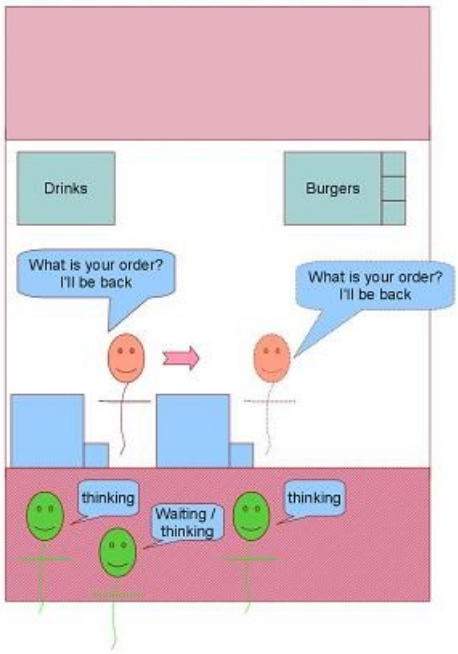
Unfortunately, this brings chaos in the restaurant. Sometimes, employes fight to get a burger to their own customer first:



And sometimes, they stay in each other's way:



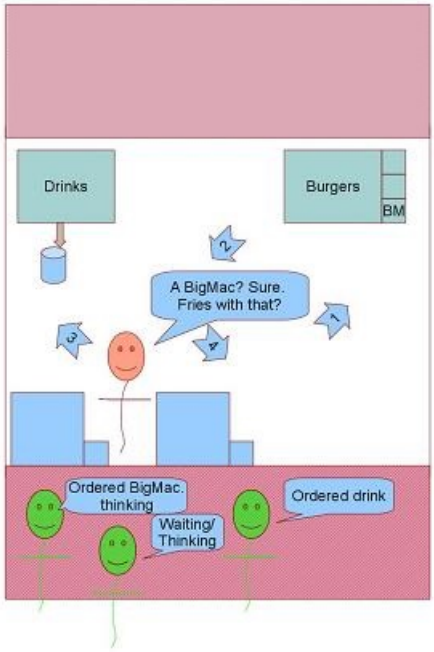
This clearly is a not an optimal solution. Not only the additional employee brings additional costs, but both employees now spend much more time waiting. It also is not a scalable solution if even more customers want to eat because it's lunch-time right now. Even worse, as they fight for resources and stay in each other's way, the restaurant now serves people less fast than before. Customers flee and the restaurant gets bankrupt. A sad story, isn't it? To avoid this, the owner decides to go asynchronous. He keeps a single worker, who runs in zero time from cash desk to cash desk:



The worker never waits because it would increase customer's waiting time. Instead, he runs from cash desks to the burger queue, beverage machine using a self-made strategy:

- ask what the customer wants and keep an up-to-date information of the customer's state.
- if we have another customer at a desk, ask what he wants. For both customers, remember the state of the order (waiting for customer choice, getting food, getting drink, delivering, getting payment, etc.)
- as soon as some new state is detected (customer choice, burger in the queue, drink ready), handle it.
- priorities are defined: start the longest-lasting tasks first, serve angry-looking customers first, etc.

The following diagram shows us the busy and really really fast worker in action:



Of course the owner needs a worker who runs fast, and has a pretty good memory so he can remember what customers are waiting for.

This is what Asynchronous is for. A worker (thread) runs as long as there are waiting customers, following a precisely defined algorithm, and lots of state machines to manage the asynchronous behaviour. In case of customers, we could have a state machine: Waiting -> PickingMenu -> WaitingForFood -> Paying.

We also need some queues (Burger queue, Beverage glass positioning) and some Asynchronous Operation Processor (for example a threadpool made of workers in the kitchen), event of different types (Drinks delivery). Maybe you also want some work stealing (someone in the kitchen serving drinks as he has no more burger to prepare. He will be slower than the machine, but still bring some time gain).

To make this work, the worker must not block, never, ever. And whatever he's doing has to be as fast as possible, otherwise the whole process stalls.

Part I. Concepts

Table of Contents

Chapter 1. Related designs: std::async, Active Object, Proactor

std::async

What is wrong with it

The following code is a classical use of std::async as it can be found in articles, books, etc.

```
std::future<int> f = std::async([](){return 42;}); // executes asynchronously
int res = f.get(); // wait for result, block until ready
```

It looks simple, easy to use, and everybody can get it. The problem is, well, that it's not really asynchronous. True, our lambda will execute in another thread. Actually, it's not even guaranteed either. But then, what do we do with our future? Do we poll it? Or call get() as in the example? But then we will block, right? And if we block, are we still asynchronous? We will postulate that no, we're not. If we block, we cannot react to any event happening in our system any more, we are unresponsive for a while (are we back to the old times of freezing programs, the old time before threads?). We also probably miss some parallelizing opportunities as we could be doing something more useful at the same time, as in our fast-food example. And diagnostics are looking bad too as we are blocked and cannot return any. What is left to us is polling. Polling? No, it cannot be true, this is what the C++ standard offers us? And if we get more and more futures, do we carry a bag of them with us at any time and check them from time to time? Do we need some functions to, at a given point, wait for all futures or any of them to be ready?

Wait, yes they exist, `wait_for_all` and `wait_for_any`...

And what about this example from an online documentation?

```
{
   std::async(std::launch::async, []{ f(); });
   std::async(std::launch::async, []{ g(); });
}
```

Every std::async returns you a future, a particularly mean one which blocks upon destruction. This means that the second line will not execute until f() completes. Now this is not only not asynchronous, it's also much slower than calling sequentially f and g while doing the same.

No, really, this does not look good. Do we have alternatives?

N3558 / N3650

Of course it did not go unnoticed that std::async has some limitations. And so do we see some tries to save it instead of giving it up. Usually, it goes around the lines of blocking, but later.

```
future<int> f1 = async([]() { return 123; });
future<string> f2 = f1.then([](future<int> f)
{
  return f.get().to_string(); // here .get() won't block
});
// and here?
string s= f2.get();
```

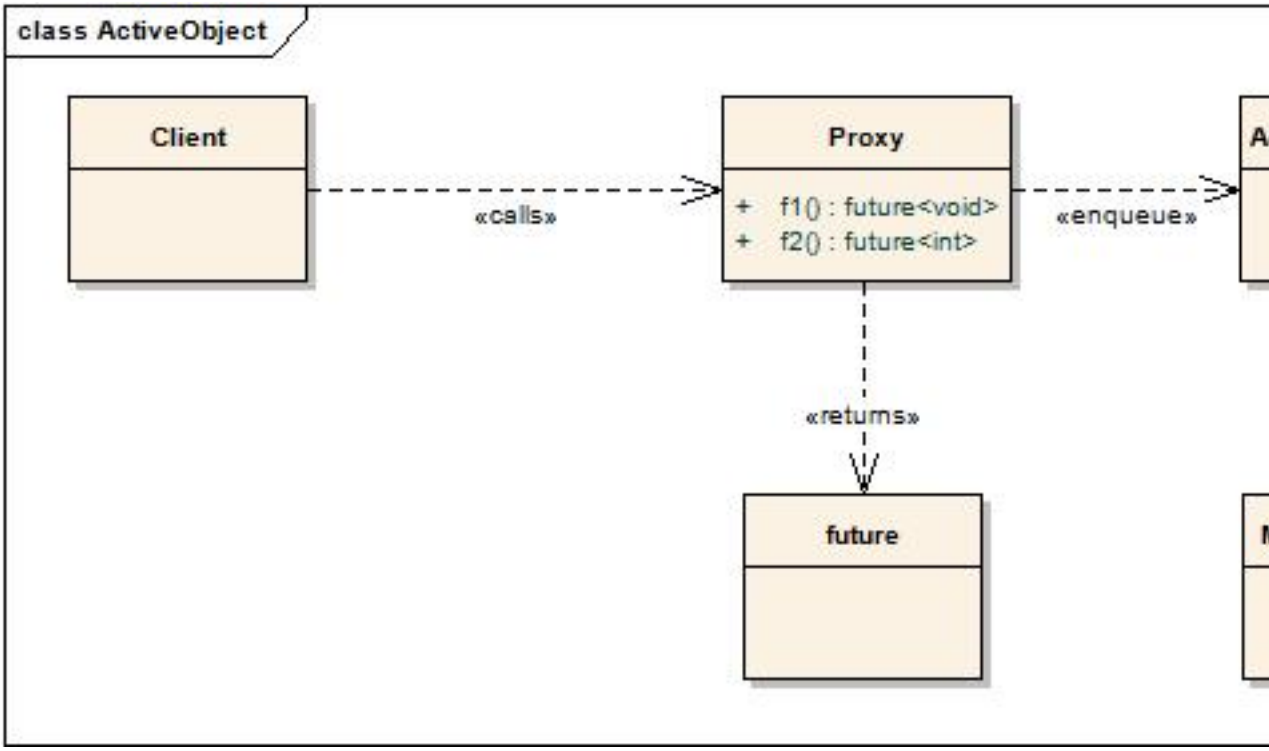
The idea is to make std::async more asynchronous (oh my, this already just sounds bad) by adding something (.then) to be called when the asynchronous action finishes. It still does not fly:

- at some point, we will have to block, thus ending our asynchronous behavior
- This works only for very small programs. Do we imagine a 500k lines program built that way?

And what about the suggestion of adding new keywords, async and await, as in N3650? Nope. First because, as await suggests, someone will need, at some point, to block waiting. Second because as we have no future, we also lose our polling option.

Active Object

Design



This simplified diagram shows a possible design variation of an Active Object pattern.

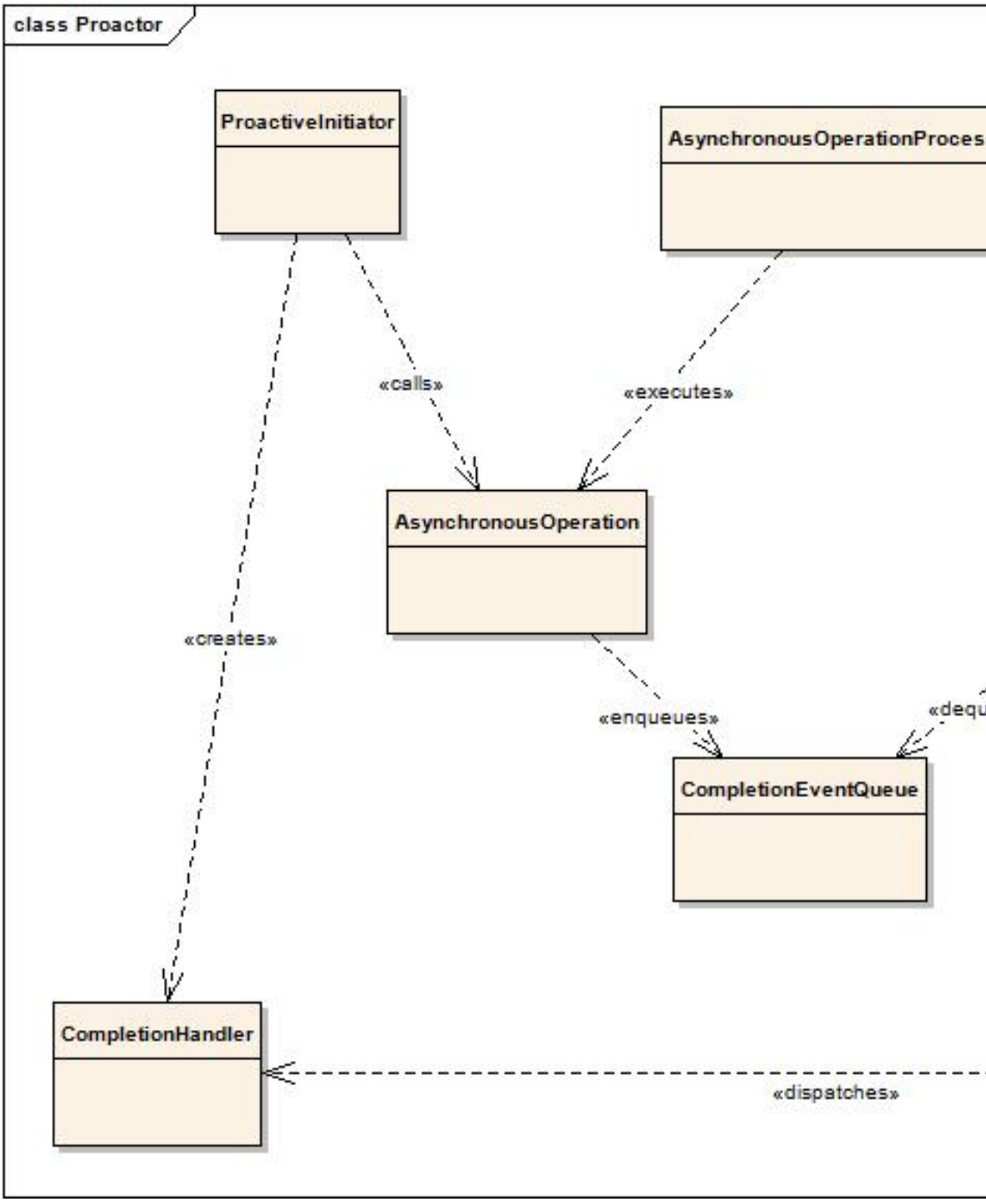
A thread-unsafe Servant is hidden behind a Proxy, which offers the same members as the Servant itself. This Proxy is called by clients and delivers a future object, which will, at some later point, contain the result of the corresponding member called on the servant. The Proxy packs a MethodRequest corresponding to a Servant call into the ActivationQueue. The Scheduler waits permanently for MethodRequests in the queue, dequeues them, and executes them. As only one scheduler waits for requests, it serializes access to the Servant, thus providing thread-safety.

However, this pattern presents some liabilities:

- Performance overhead: depending on the system, data moving and context switching can be a performance drain.
- Memory overhead: for every Servant, a thread has to be created, consuming resources.
- Usage: getting a future doesn't bring as much asynchronous behaviour as one might think. Usually, docs tell you to do something else and check it later. But most cases simply mean that the client will earlier or later block until the future is ready. This also applies to std/boost::async.

Proactor

Design


class Proactor
ProactiveInitiator
AsynchronousOperationProces
«calls»
«executes»
AsynchronousOperation
«creates»
«enqueues»
«dequ
CompletionEventQueue
CompletionHandler
«dispatches»

TODO

Chapter 2. Features

Active Component

Extending Active Objects with more servants within a thread context

A commonly cited drawback of Active Objects is that they are awfully expensive. A thread per object is really a waste of ressources. Boost.Asynchronous extends this concept by allowing an unlimited number of objects to live within a single thread context, thus amortizing the costs.

This brings another difference with ActiveObjects. As many objects are potentially living in a thread context, none should be allowed to process long-lasting tasks as it would reduce reactivity of the whole component. In this aspect, Asynchronous' philosophy is closer to a Proactor.

As long-lasting tasks do happen, Boost.Asynchronous provides several implementations of threadpools and the infrastructure to make it safe to post work to threadpools and get aynchronously a safe callback. It also provides safe mechanisms to shutdown Active Components and threadpools.

Shutting down

Shutting down a thread turns out to be harder in practice than expected, as shown by several posts of surprise on the Boost mailing lists when Boost.Thread tried to match the C++ Standard. Asynchronous hides all these ugly details. What users see is a proxy object, which can be shared by any number of objects executing within any number of threads.

When the last instance of the inner-ActiveComponent scheduler object is destroyed, the scheduler thread is stopped. When the last instance of a scheduler proxy is destroyed, the scheduler thread is joined. It's as simple as that. This makes threads shared objects.

Object lifetime

There are subtle bugs when living in a multithreaded world. Consider the following class:

```
struct Unsafe
{
  void foo()
  {
    m_mutex.lock();
    // call private member
    m_mutex.unlock();
  }
private:
  void foobar()
  {
    //we are already locked when called, do something while locked
  }
  boost::mutex m_mutex;
};
```

This is called a thread-safe interface pattern. Public members lock, private do not. Simple enough, right? Unfortunately, it doesn't fly.

First you have the risk of deadlock if a private member calls a public one while being called from another public member. Forget to check one path of execution within your class implementation and you get a nice deadlock. You'll have to test every single path of execution to prove your code is correct. And this at every change. Hmmm, does not sound reassuring.

Anyway, let's face it, for any complex class, where there's a mutex, there is a race or a deadlock...

But even worse, the principle itself is not correct in C++. It supposes that a class can protect itself. Well, no, it can't. Why? You can't protect the destructor. If the object (and the mutex) gets destroyed when a thread waits for it in foo(), we're toast. Ok, then we can use a shared_ptr, and all is good, right? Then you have no destructor called as someone keeps the object alive, right? Well, you still have a risk of a signal, callback, etc.

What you need is protect your object with a shared_ptr and have no other way to access the object. Asynchronous provides this.

There are more lifetime issues, even without mutexes. If you have ever used Boost.Asio, a common mistake and an easy one is when a callback is called in the proactor thread after an asynchronous operation, but the object called is long gone and the callback invalid. Asynchronous provides **trackable_servant** which makes sure that a callback is not called if the object which called the asynchronous operation is gone. It also prevents a task posted in a threadpool to be called if this condition occurs, which improves performance. Some helper members even make sure you get no crash, even while facing a Boost.Asio callback.

Servant Proxies

Asynchronous offers `servant_proxy`, which make the outside world call members of a servant as if it was not living in an ActiveObject. It looks like a thread-safe interface, but safe from deadlock and race conditions unles you really try hard.

Interrupting

Or how to catch back if you're drowning.

Let's say you posted so many tasks to your threadpool that all your cores are full, still, your application is slipping more and more behind plan. You need to give up some tasks to catch back a little.

Asynchronous can give you an interruptible cookie when you post a task to a scheduler, and you can use it to **stop a posted task**. If not running yet, the task will not start, if running, it will stop at the next interruption point, which are documented in the Boost.Thread documentation [http://www.boost.org/doc/libs/1_54_0/doc/html/thread/thread_management.html#thread.thread_management.tutorial.interruption]. Diagnostics will show that a task was interrupted.

Diagnostics

Finding out how good your software is doing is not an easy task. You need to add lots of logging to find out which function call takes too long and becomes a bottleneck. Finding out the minimum required hardware to run your application is even harder.

Asynchronous design helps here too. By logging the required time and the frequency of tasks, it is easy to find out how many cores are needed. Bottlenecks can be found by logging what the Active Component is doing and how long. Finally, designing the asynchronous Active Component as state machines and logging state changes will allow a better understanding of your system and make visible potential for concurrency. Even for non-parallel algorithms, finding out, using a state machine, the earliest point a task can be thrown to a threadpool will give some easy concurrency. Throw enough tasks to the threadpool and manage this with a state machine and you might use your cores with little effort. Parallelization can then be used later on by logging which tasks are long enough to be parallelized.

Continuations

Callback are great when you have a complex flow of operations which require a state machine for management, however there are cases where callbacks are not an ideal solution. Either because your

application requires a constant switching of context between single-threaded and parallel parts, or because the single-threaded part might be busy, which would delay completion of the algorithm. A known example of this is a parallel fibonacci. In this case, one can register a **continuation**, which is to be executed upon completion of one or several tasks.

This mechanism is flexible so that you can use it with futures coming from another library, thus removing any need for a `wait_for_all(futures...)` or a `wait_for_any(futures...)`.

Want more power? What about extra machines?

What to do if your threadpools are using all of your cores but there simply are not enough cores for the job? Buy more cores? Unfortunately, the number of cores a single-machine can use is limited, unless you have unlimited money. A dual 6-core Xeon, 24 threads with hyperthreading will cost much more than 2 x 6-core i7, and will usually have a lesser clock frequency and an older architecture.

The solution could be: start with the i7, then if you need more power, add some more machines which will steal jobs from your threadpools with **TCP**. This can be done quite easily with Asynchronous.

Want to build your own hierarchical network of servers? It's hard to make it easier.

Parallel algorithms

The library also comes with **non-blocking algorithms** with iterators or ranges, support for TCP, which fit well in the asynchronous system, with more to come. If you want to contribute some more, be welcome. At the moment, the library offers:

- parallel_for:
- parallel_reduce
- parallel_extremum
- parallel_count
- parallel_find_all
- parallel_invoke

Task Priority

Asynchronous offers this possibility for all schedulers at low performance cost. This means you not only have the possibility to influence task execution order in a threadpool but also in Active Objects.

This is achieved by posting a task to the queue with the corresponding priority. It is also possible to get it even more fine-grained by using a sequence of queues, etc.

Integrating with Boost.Asio

Asynchronous offers a Boost.Asio based **scheduler** allowing you to easily write a Servant using Asio, or an Asio based threadpool. An advantage is that you get safe callbacks and easily get your Asio application to scale. Writing a server has never been easier.

Asynchronous also uses Boost.Asio to provide a timer with callbacks.

Integrating with Qt

What about getting the power of Asynchronous within a Qt application? Use Asynchronous' threadpools, algorithms and other cool features easily.

Work Stealing

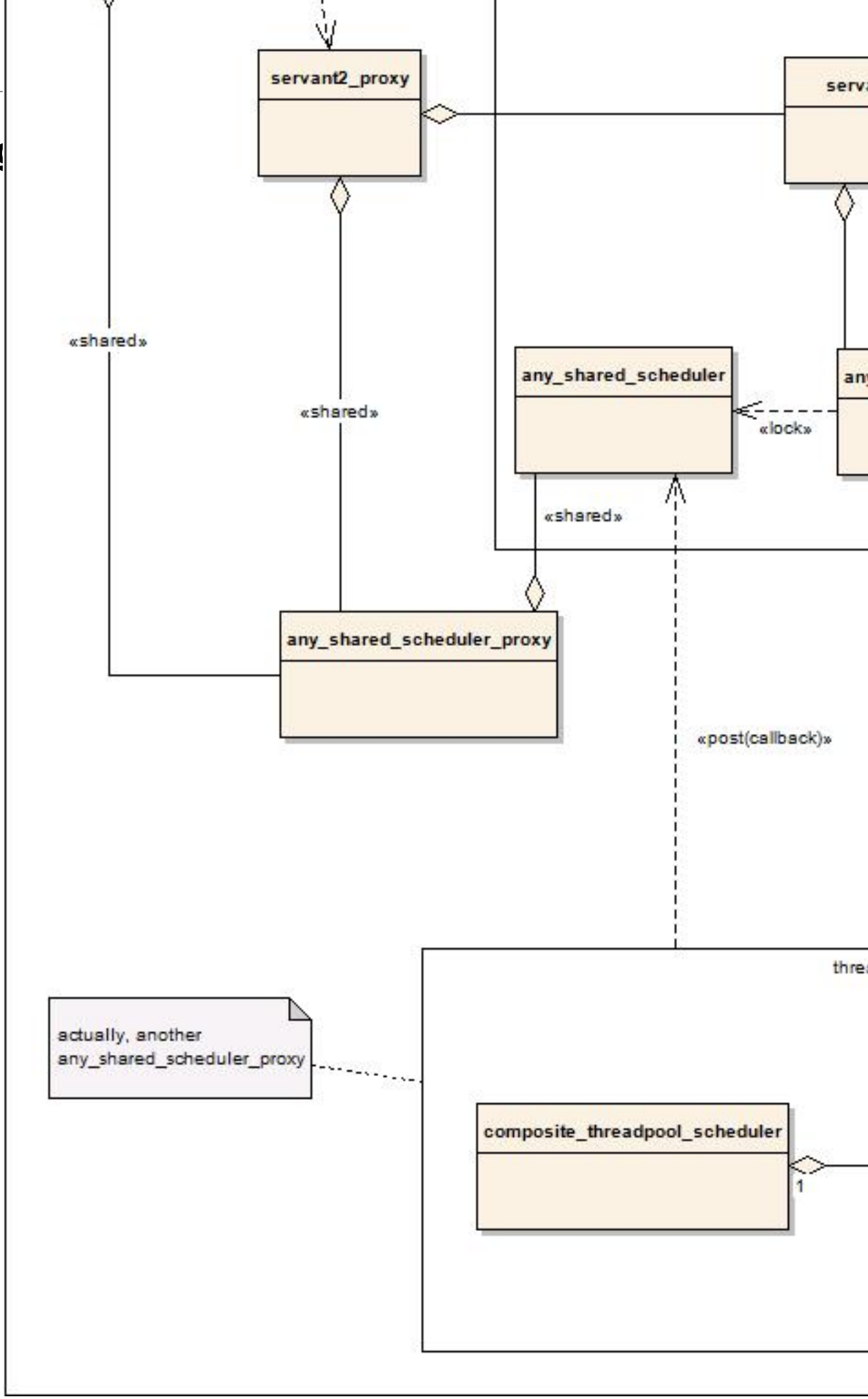
Work stealing is supported both within the threads of a threadpool but also between different threadpools. Please have a look at Asynchronous' composite scheduler.

Extending the library

Asynchronous has been written with the design goal of allowing anybody to extend the library. In particular, the author is hoping to be offered the following extensions:

- Schedulers, Threadpools
- Queues
- Parallel algorithms
- Integration with other libraries

Desig


servant2_proxy
«shared»
«shared»
any_shared_scheduler
«lock»
«shared»
any_shared_scheduler_proxy
«post(callback)»
actually, another
any_shared_scheduler_proxy
composite_threadpool_scheduler
1

This diagram shows an overview of the design behind Asynchronous. One or more Servant objects live in a single-theaded world, communicating with the outside world only through one or several queues, from which the single-threaded scheduler pops tasks. Tasks are pushed by calling a member on a proxy object.

Like an Active Object, a client uses a proxy (a shared object type), which offers the same members as the real servant, with the same parameters, the only difference being the return type, a boost::future<R>, with R being the return type of the servant's member. All calls to a servant from the client side are posted, which includes the servant constructor and destructor. When the last instance of a servant is destroyed, be it used inside the Active Component or outside, the servant destructor is posted.

any_shared_scheduler is the part of the Active Object scheduler living inside the Active Component. Servants do not hold it directly but hold an any_weak_scheduler instead. The library will use it to create a posted callback when a task executing in a worker threadpool is completed.

Shutting down an Active Component is done automatically by not needing it. It happens in the following order:

- While a servant proxy is alive, no shutdown
- When the last servant proxy goes out of scope, the servant destructor is posted.
- if jobs from servants are running in a threadpool, they get a chance to stop earlier by running into an interruption point or will not even start.
- threadpool(s) is (are) shut down.
- The Active Component scheduler is stopped and its thread terminates.
- The last instance of any_shared_scheduler_proxy goes out of scope with the last servant proxy and joins.

It is usually accepted that threads are orthogonal to an OO design and therefore are hard to manage as they don't belong to an object. Asynchronous comes close to this: threads are not directly used, but instead owned by a scheduler, in which one creates objects and tasks.

Part II. User Guide

Table of Contents

Chapter 3. Using Asynchronous

Hello, asynchronous world

The following code shows a very basic usage (a complete example here [examples/example_post_future.cpp]), this is not really asynchronous yet:

```
#include <boost/asynchronous/scheduler/threadpool_scheduler.hpp>
#include <boost/asynchronous/queue/lockfree_queue.hpp>
#include <boost/asynchronous/scheduler_shared_proxy.hpp>
#include <boost/asynchronous/post.hpp>
struct void_task
{
    void operator()()const
    {
        std::cout << "void_task called" << std::endl;
    }
};
struct int_task
{
    int operator()()const
    {
        std::cout << "int_task called" << std::endl;
        return 42;
    }
};

// create a threadpool scheduler with 3 threads and communicate with it using a
// we use auto as it is easier than boost::asynchronous::any_shared_scheduler_p
auto scheduler = boost::asynchronous::create_shared_scheduler_proxy(
                            new boost::asynchronous::threadpool_scheduler<
                                boost::asynchronous::lockfree_queue<> >(3));
// post a simple task and wait for execution to complete
boost::shared_future<void> fuv = boost::asynchronous::post_future(scheduler, vo
fuv.get();
// post a simple task and wait for result
boost::shared_future<int> fui = boost::asynchronous::post_future(scheduler, int_
int res = fui.get();
```

Of course this works with C++11 lambdas:

```
auto scheduler = boost::asynchronous::create_shared_scheduler_proxy(
                            new boost::asynchronous::threadpool_scheduler<
                                boost::asynchronous::lockfree_queue<> >(3));
// post a simple task and wait for execution to complete
boost::shared_future<void> fuv =
                boost::asynchronous::post_future(scheduler, [](){std::cout << "
fuv.get();
// post a simple task and wait for result
boost::shared_future<int> fui =
                boost::asynchronous::post_future(scheduler, [](){std::cout << "
int res = fui.get();
```

boost::asynchronous::post_future posts a piece of work to a threadpool scheduler with 3 threads and using a lockfree_queue. We get a boost::future<the type of the task return type>.

This looks like much std::async, but we're just getting started. Let's move on to something more asynchronous.

A servant proxy

We now want to create a single-threaded scheduler, populate it with some servant(s), and exercise some members of the servant from an outside thread. We first need a servant:

```
struct Servant
{
    // optional: the servant has such an easy constructor, no need to post it
    typedef int simple_ctor;
    Servant(int data): m_data(data){}
    int doIt()const
    {
        std::cout << "Servant::doIt with m_data:" << m_data << std::endl;
        return 5;
    }
    void foo(int& i)const
    {
        std::cout << "Servant::foo with int:" << i << std::endl;
        i = 100;
    }
    void foobar(int i, char c)const
    {
        std::cout << "Servant::foobar with int:" << i << " and char:" << c <<st
    }
    int m_data;
};
```

We now create a proxy type to be used in other threads:

```
class ServantProxy : public boost::asynchronous::servant_proxy<ServantProxy,Ser
{
public:
    // forwarding constructor. Scheduler to servant_proxy, followed by argument
    template <class Scheduler>
    ServantProxy(Scheduler s, int data):
        boost::asynchronous::servant_proxy<ServantProxy,Servant>(s, data)
    {}
    // the following members must be available "outside"
    // foo and foobar, just as a post (no interesting return value)
    BOOST_ASYNC_POST_MEMBER(foo)
    BOOST_ASYNC_POST_MEMBER(foobar)
    // for doIt, we'd like a future
    BOOST_ASYNC_FUTURE_MEMBER(doIt)
};
```

Let's use our newly defined proxy:

```
int something = 3;
{
    // with c++11
    auto scheduler = boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::single_thread_scheduler<
                          boost::asynchronous::lockfree_queue<> >);

    {
        // arguments (here 42) are forwarded to Servant's constructor
```

```
        ServantProxy proxy(scheduler,42);
        // post a call to foobar, arguments are forwarded.
        proxy.foobar(1,'a');
        // post a call to foo. To avoid races, the reference is ignored.
        proxy.foo(something);
        // post and get a future because we're interested in the result.
        boost::shared_future<int> fu = proxy.doIt();
        std::cout<< "future:" << fu.get() << std::endl;
    }// here, Servant's destructor is posted
}// scheduler is gone, its thread has been joined
std::cout<< "something:" << something << std::endl; // something was not change
```

We can call members on the proxy, almost as if they were called on Servant. The library takes care of the posting and forwarding the arguments. When required, a future is returned. Stack unwinding works, and when the servant proxy goes out of scope, the servant destructor is posted. When the scheduler goes out of scope, its thread is stopped and joined. The queue is processed completely first. Of course, as many servants as desired can be created in this scheduler context. Please have a look at the complete example [examples/example_simple_servant.cpp].

Using a threadpool

If you remember the principles of Asynchronous, blocking a single-thread scheduler is taboo as it blocks the thread doing all the management of a system. But what to do when one needs to execute long tasks? Asynchronous provides a whole set of threadpools. A servant posts something to a threadpool, provides a callback, then gets a result. Wait a minute. Callback? Is this not thread-unsafe? Why not threadpools with futures, like usual? Because in a perfectly asynchronous world, waiting for a future means blocking a servant scheduler. One would argue that it is possible not to block on the future, and instead ask if there is a result. But then, what if not? Is the alternative to poll? Like in the "good" all times?

If we accept the future argument, but what about thread-safety? Asynchronous takes care of this. A callback is never called from a threadpool, but instead posted back to the queue of the scheduler which posted the work. All the servant has to do is to do nothing and wait until the callback is executed. Note that this is not the same as a blocking wait, the servant can still react to events.

Clearly, this brings some new challenges as the flow of control gets harder to follow. This is why a servant is often written using state machines. The (biased) author suggests to have a look at the Meta State Machine library [http://svn.boost.org/svn/boost/trunk/libs/msm/doc/HTML/index.html] , which plays nicely with Asynchronous.

But what about the usual proactor issues (crashes) when the servant has long been destroyed when the callback is posted. Gone. Asynchronous provides `trackable_servant` which will ensure that a callback is not called if the servant is gone. Better even, if the servant has been destroyed, an unstarted posted task will not be executed.

Again comes another issue. What if I post a task, say a lambda, which captures a shared_ptr to an object per value, and this object is a boost::signal? Then when the task object has been executed and is destroyed, I'll get a race on the signal deregistration. Good point, but again no. Asynchronous ensures that a task created within a scheduler context gets destroyed in this context.

This is about the best protection you can get. What Asynchronous cannot protect you from are self-made races within a task (if you post a task with a pointer to the servant, you're on your own and have to protect your servant). A good rule of thumb is to consider data passed to a task as moved. To support this, Asynchronous does not copy tasks but moves them.

Armed with these protections, let's give a try to a threadpool, starting with the most basic one, `threadpool_scheduler` (more to come):

```
struct Servant : boost::asynchronous::trackable_servant<>
{
```

```
    Servant(boost::asynchronous::any_weak_scheduler<> scheduler)
        : boost::asynchronous::trackable_servant<>(scheduler,
                                                   // threadpool with 3 threads and
                                                   boost::asynchronous::create_share
                                                       new boost::asynchronous::thre
                                                               boost::asynchronous:
    // call to this is posted and executes in our (safe) single-thread schedule
    void start_async_work()
    {
       //ok, let's post some work and wait for an answer
       post_callback(
                   [](){std::cout << "Long Work" << std::endl;}, //work, do no
                   [/*this*/](boost::future<void>){...}// callback. Safe to us
        );
    }
};
```

We now have a servant, ready to be created in its own thread, which posts some long work to a three-thread-threadpool and gets a callback, but only if still alive. Similarly, the long work will be executed by the threadpool only if Servant is alive by the time it starts. Everything else stays the same, one creates a proxy for the servant and posts calls to its members, so we'll skip it for conciseness, the complete example can be found here [examples/example_post_trackable_threadpool.cpp].

A servant using another servant proxy

Often, in a layered design, you'll need that a servant in a single-threaded scheduler calls a member of a servant living in another one. And you'll want to get a callback, not a future like in our previous example, because you absolutely refuse to block waiting for a future (and you'll be very right of course!). Ideally, except for main(), you won't want any of your objects to wait for a future. There is another servant_proxy macro for this, `BOOST_ASYNC_UNSAFE_MEMBER`(unsafe because you get no thread-safety from if and you'll take care of this yourself, or better, `trackable_servant` will take care of it for you, as follows):

```
// Proxy for a basic servant
class ServantProxy : public boost::asynchronous::servant_proxy<ServantProxy,Ser
{
public:
    template <class Scheduler>
    ServantProxy(Scheduler s, int data):
        boost::asynchronous::servant_proxy<ServantProxy,Servant>(s, data)
    {}
    BOOST_ASYNC_UNSAFE_MEMBER(foo)
    BOOST_ASYNC_UNSAFE_MEMBER(foobar)
};

// Servant using the first one
struct Servant2 : boost::asynchronous::trackable_servant<>
{
    Servant2(boost::asynchronous::any_weak_scheduler<> scheduler,ServantProxy w
        :boost::asynchronous::trackable_servant<>(scheduler)
        ,m_worker(worker) // the proxy allowing access to Servant
    boost::shared_future<void> doIt
    {
         call_callback(m_worker.get_proxy(), // Servant's outer proxy, for post
                       m_worker.foo(), // what we want to call on Servant
                      // callback functor, when done.
                      [](boost::future<int> result){...} );// future<return typ
    }
```

```
};
```

Call of `foo()` will be posted to `Servant`'s scheduler, and the callback lambda will be posted to `Servant2` when completed. All this thread-safe of course. Destruction is also safe. When `Servant2` goes out of scope, it will shutdown `Servant`'s scheduler, then will his scheduler be shutdown (provided no more object is living there), and all threads joined. The complete example [examples/example_two_simple_servants.cpp] shows a few more calls too.

Interrupting tasks

Imagine a manager object (a state machine for example) posted some long-lasting work to a threadpool, but this long-lasting work really takes too long. As we are in an asynchronous world and non-blocking, the manager object realizes there is a problem and decides the task must be stopped otherwise the whole application starts failing some real-time constraints. This is made possible by using another version of posting, getting a handle, on which one can require interruption. As Asynchronous does not kill threads, it means that we'll have to use one of Boost.Thread predefined interruption points. Supposing we have well-behaved tasks, they will be interrupted at the next interruption point if they started, or if they did not start yet because they are waiting in a queue, then they will never start. In this example [examples/example_interrupt.cpp], we have very little to change but the post call:

```
struct Servant : boost::asynchronous::trackable_servant<>
{
     ... // as usual
    void start_async_work()
    {
        // start long interruptible tasks
        // we get an interruptible handler representing the task
        boost::asynchronous::any_interruptible interruptible =
        interruptible_post_callback(
                // interruptible task
               [](){
                    std::cout << "Long Work" << std::endl;
                    boost::this_thread::sleep(boost::posix_time::milliseconds(1
               // callback functor.
               [](boost::future<void> ){std::cout << "Callback will most likely
        );
        // let the task start (not sure but likely)
        // if it had no time to start, well, then it will never.
        boost::this_thread::sleep(boost::posix_time::milliseconds(100));
        // actually, we changed our mind and want to interrupt the task
        interruptible.interrupt();
        // the callback will never be called as the task was interrupted
    }
};
```

Logging tasks

Developers are notoriously famous for being bad at guessing which part of their code is inefficient or has potential for long execution time. This is bad in itself, but even worse for a control class like our post-callback servant as it reduces responsiveness. Knowing how long a posted call or a callback lasts is therefore very useful. Knowing how long take tasks executing in the threadpools is also essential to plan what hardware one needs for an application(4 cores? Or 100?). We need to know what our program is doing. Asynchronous provides some logging per task to help there. Let's have a look at some code. It's also time to start using our template parameters for `trackable_servant`, in case you wondered why they are here.

```
// we will be using loggable jobs internally
typedef boost::asynchronous::any_loggable<boost::chrono::high_resolution_clock>
```

```
// the type of our log
typedef std::map<std::string,std::list<boost::asynchronous::diagnostic_item<boo
// we log our scheduler and our threadpool scheduler (both use servant_job)
struct Servant : boost::asynchronous::trackable_servant<servant_job,servant_job
{
    Servant(boost::asynchronous::any_weak_scheduler<servant_job> scheduler) //s
        : boost::asynchronous::trackable_servant<servant_job,servant_job>(sched
                                                boost::asynchronous::create_shar
                                                    // threadpool with 3 threads
                                                    // Furthermore, it logs post
                                                    new boost::asynchronous::thr
                                                            //servant_job is our
                                                            boost::asynchronous:
    void start_async_work()
    {
         post_callback(
               // task posted to threadpool
               [](){...}, // will return an int
               [](boost::future<int> res){...},// callback functor.
               // the task / callback name for logging
               "int_async_work"
        );
    }
    // we happily provide a way for the outside world to know what our threadpo
    // get_worker is provided by trackable_servant and gives the proxy of our t
    diag_type get_diagnostics() const
    {
        return (*get_worker()).get_diagnostics();
    }
};
```

The proxy is also slightly different, using a _LOG macro and an argument representing the name of the task.

```
class ServantProxy : public boost::asynchronous::servant_proxy<ServantProxy,Ser
{
public:
    template <class Scheduler>
    ServantProxy(Scheduler s):
        boost::asynchronous::servant_proxy<ServantProxy,Servant,servant_job>(s)
    {}
    // the _LOG macros do the same as the others, but take an extra argument, t
    BOOST_ASYNC_FUTURE_MEMBER_LOG(start_async_work,"proxy::start_async_work")
    BOOST_ASYNC_FUTURE_MEMBER_LOG(get_diagnostics,"proxy::get_diagnostics")
};
```

We now can get diagnostics from both schedulers, the single-threaded and the threadpool (as external code has no access to it, we ask Servant to help us there through a get_diagnostics() member).

```
// create a scheduler with logging
auto scheduler = boost::asynchronous::create_shared_scheduler_proxy(
                                new boost::asynchronous::single_thread_schedule
                                    boost::asynchronous::lockfree_queue<servant_
// create a Servant
ServantProxy proxy(scheduler);
...
// let's ask the single-threaded scheduler what it did.
diag_type single_thread_sched_diag = (*scheduler).get_diagnostics();
for (auto mit = single_thread_sched_diag.begin(); mit != single_thread_sched_di
```

```
{
     std::cout << "job type: " << (*mit).first << std::endl;
     for (auto jit = (*mit).second.begin(); jit != (*mit).second.end();++jit)
     {
          std::cout << "job waited in us: " << boost::chrono::nanoseconds((*jit
          std::cout << "job lasted in us: " << boost::chrono::nanoseconds((*jit
          std::cout << "job interrupted? "  << std::boolalpha << (*jit).is_inte
     }
}
```

It goes similarly with the threapool scheduler, with the slight difference that we ask the Servant to deliver diagnostic information through a proxy member. The complete example [examples/example_log.cpp] shows all this, plus an interrupted job (you might have noticed in the previous listing that a diagnostic offers an `is_interrupted` member).

Queue container with priority

Sometimes, all jobs posted to a scheduler do not have the same priority. For threadpool schedulers, `composite_threadpool_scheduler` is an option. For a single-threaded scheduler, Asynchronous does not provide a priority queue but a queue container, which itself contains any number of queues, of different types if needed. This has several advantages:

- Priority is defined simply by posting to the queue with the desired priority, so there is no need for expensive priority algorithms.
- Ones gets also reduced contention if many threads of a threadpool post something to the queue of a single-threaded scheduler. If no priority is defined, one queue will be picked, according to a configurable policy, reducing contention on a single queue.
- It is possible to mix queues to get the best of each.
- One can build a queue container of queue containers, etc.

Note: This applies to any scheduler. We'll start with single-threaded schedulers used by managing servants for simplicity, but it is possible to have composite schedulers using queue containers for finest granularity and least contention.

First, we need to create a single-threaded scheduler with several queues for our servant to live in, for example, one threadsafe list and three lockfree queues:

```
boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
                              boost::asynchronous::create_shared_scheduler_proxy(
                                   new boost::asynchronous::single_thread_schedule
                                           boost::asynchronous::any_queue_containe
                                (boost::asynchronous::any_queue_container_config<boost:
                                 boost::asynchronous::any_queue_container_config<boost:
                                 ));
```

`any_queue_container` takes as constructor arguments a variadic sequence of `any_queue_container_config`, with a queue type as template argument, and in the constructor the number of objects of this queue (in the above example, one `threadsafe_list` and 3 `lockfree_queue` instances, then the parameters that these queues require in their constructor (100 is the capacity of the underlying `boost::lockfree_queue`). This means, that our `single_thread_scheduler` has 4 queues:

- a threadsafe_list at index 1
- lockfree queues at indexes 2,3,4
- >= 4 means the queue with the least priority.
- 0 means "any queue" and is the default

The scheduler will handle these queues as having priorities: as long as there are work items in the first queue, take them, if there are no, try in the second, etc. If all queues are empty, the thread gives up his time slice and sleeps until some work item arrives. If no priority is defined by posting, a queue will be chosen (by default randomly, but you can configure this with a policy). This has the advantage of reducing contention of the queue, even when not using priorities. The servant defines the priority of the tasks it provides. While this might seem surprising, it is a design choice to avoid someone using a servant proxy interface to think about it, as you will see in the second listing. To define a priority for a servant proxy, there is a second field in the macros:

```
class ServantProxy : public boost::asynchronous::servant_proxy<ServantProxy,Serv
{
public:
    template <class Scheduler>
    ServantProxy(Scheduler s):
        boost::asynchronous::servant_proxy<ServantProxy,Servant>(s)
    {}
    BOOST_ASYNC_SERVANT_POST_CTOR(3)
    BOOST_ASYNC_SERVANT_POST_DTOR(4)
    BOOST_ASYNC_FUTURE_MEMBER(start_async_work,1)
};
```

BOOST_ASYNC_FUTURE_MEMBER and other similar macros can be given an optional priority parameter, in this case 1, which is our threadsafe list. Notice how you can then define the priority of the posted servant constructor and destructor.

```
ServantProxy proxy(scheduler);
boost::future<boost::future<int>> fu = proxy.start_async_work();
```

Calling our proxy member stays unchanged because the macro defines the priority of the call.

We also have an extended version of `post_callback`, called by a servant posting work to a threadpool:

```
post_callback(
       [](){return 42;},// work
       [this](boost::future<int> res){}// callback functor.
       ,"",
       2,2
);
```

Note the two added priority values: the first one for the task posted to the threadpool, the second for the priority of the callback posted back to the servant scheduler. The string is the log name of the task, which we choose to ignore here.

The priority is in any case an indication, the scheduler is free to ignore it if not supported. In the example [examples/example_queue_container.cpp], the single threaded scheduler will honor the request, but the threadpool has a normal queue and cannot honor the request, but a threadpool with an `any_queue_container` or a `composite_threadpool_scheduler` can. The same example [examples/example_queue_container_log.cpp] can be rewritten to make use of the logging mechanism.

`any_queue_container` has two template arguments. The first, the job type, is as always by default, a callable (`any_callable`) job. The second is the policy which Asynchronous uses to find the desired queue for a job. The default is `default_find_position`, which is as described above, 0 means any position, all other values map to a queue, priorities >= number of queues means last queue. Any position is by default random (`default_random_push_policy`), but you might pick `sequential_push_policy`, which keeps an atomic counter to post jobs to queues in a sequential order.

If your idea is to build a queue container of queue containers, you'll probably want to provide your own policy.

Multiqueue Schedulers' priority

TODO

Threadpool Schedulers with several queues

A queue container has advantages (different queue types, priority for single threaded schedulers) but also disadvantages (takes jobs from one end of the queue, which means potential cache misses, more typing work). If you don't need different queue types for a threadpool but want to reduce contention, multiqueue schedulers are for you. A normal `threadpool_scheduler` has x threads and one queue, serving them. A `multiqueue_threadpool_scheduler` has x threads and x queues, each serving a worker thread. Each thread looks for work in its queue. If it doesn't find any, it looks for work in the previous one, etc. until it finds one or inspected all the queues. As all threads steal from the previous queue, there is little contention. The construction of this threadpool is very similar to the simple `threadpool_scheduler`:

```
boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::create_shared_scheduler_proxy(
                // 4 threads and 4 lockfree queues of 10 capacity
                new boost::asynchronous::multiqueue_threadpool_scheduler<boost:
```

The first argument is the number of worker threads, which is at the same time the number of queues. As for every scheduler, if the queue constructor takes arguments, they come next and are forwarded to the queue.

This is the advised scheduler for standard cases as it offers lesser contention and task stealing between the queues it uses for task transfer.

There is a limitation, these schedulers cannot have 0 thread like their single-queue counterparts.

Composite Threadpool Scheduler

When a project becomes more complex, having a single threadpool for the whole application does not offer enough flexibility in load planning. It is pretty hard to avoid either oversubscription (more busy threads than available hardware threads) or undersubscription. One would need one big threadpool with exactly the number of threads available in the hardware. Unfortunately, if we have a hardware with, say 12 hardware threads, parallelizing some work using all 12 might be slowlier than using only 8. One would need different threadpools of different number of threads for the application. This, however, has the serious drawback that there is a risk that some threadpools will be in overload, while others are out of work unless we have work stealing between different threadpools.

The second issue is task priority. One can define priorities with several queues or a queue container, but this ensures that only highest priority tasks get executed if the system is coming close to overload. Ideally, it would be great if we could decide how much compute power we give to each task type.

This is what `composite_threadpool_scheduler` solves. This pool supports, like any other pool, the `any_shared_scheduler_proxy`concept so you can use it in place of the ones we used so far. The pool is composed of other pools (`any_shared_scheduler_proxy` pools). It implements work stealing between pools if a) the pools support it and b) the queue of a pool also does. For example, if we define as worker of a servant inside a single-threaded scheduler:

```
// create a composite threadpool made of:
// a multiqueue_threadpool_scheduler, 1 thread, with a lockfree_queue of capaci
// This scheduler does not steal from other schedulers, but will lend its queue
boost::asynchronous::any_shared_scheduler_proxy<> tp = boost::asynchronous::crea
               new boost::asynchronous::multiqueue_threadpool_scheduler<boost::a

// a stealing_multiqueue_threadpool_scheduler, 3 threads, each with a threadsafe
```

```
// this scheduler will steal from other schedulers if it can. In this case it w
boost::asynchronous::any_shared_scheduler_proxy<> tp2 = boost::asynchronous::cr
                    new boost::asynchronous::stealing_multiqueue_threadpool_sch

// a multiqueue_threadpool_scheduler, 4 threads, each with a lockfree_spsc_queu
// this works because there will be no stealing as the queue can't, and only th
boost::asynchronous::any_shared_scheduler_proxy<> tp3 = boost::asynchronous::cr
               new boost::asynchronous::multiqueue_threadpool_scheduler<boost::a

// create a composite pool made of the 3 previous ones
boost::asynchronous::any_shared_scheduler_proxy<> tp_worker =
             boost::make_shared<boost::asynchronous::composite_threadpool_sched
```

We can use this pool:

- As a big worker pool. In this case, the priority argument we use for posting refers to the (1-based) index of the subpool (post_callback(func1,func2,"task name",**1**,0);). "1" means post to the first pool. But another pool could steal the work.
- As a pool container, but different parts of the code will get to see only the subpools. For example, the pools tp, tp2 and tp3 can still be used independently as a worker pool. Calling composite_threadpool_scheduler<>::get_scheduler(std::size_t index_of_pool) will also give us the corresponding pool (1-based, as always).

A good example of why to use this pool is if you have a threadpool for an asio-based communication. Using such a pool inside the composite pool will allow the threads of this pool to help (steal) other pools if they have nothing to do.

Stealing is done with priority. A stealing poll first tries to steal from the first pool, then from the second, etc.

The following example [examples/example_composite_threadpool.cpp] shows a complete servant implementation, and the **ASIO section** will show how an ASIO pool can steal.

The threadpool schedulers we saw so far are not stealing from other pools. The single-queue schedulers are not stealing, and the multiqueue schedulers steal from the queues of other threads of the same pool. The stealing schedulers usually indicate this by appending a `stealing_` to their name:

- `stealing_threadpool_scheduler` is a `threadpool_scheduler` which steals from other pools.
- `stealing_multiqueue_threadpool_scheduler` is a `multiqueue_threadpool scheduler` which steals from other pools.
- `asio_scheduler steals`.

The only difference with their not stealing equivalent is that they need a composite_scheduler to tell them from which queues from other pools they can steal.

Not all schedulers offer a queue to steal from. A `single_thread_scheduler` does not as it would likely bring race conditions in active objects. If you do want to allow stealing, use a threadpool with 1 thread. An `asio_scheduler` also offers no queue to steal from although it can steal from other queues because Boost.Asio does not offer this in its interface. Future extensions will overcome this.

TODO priority posting

Another interesting usage will be when planning for extra machines to help a threadpool by processing some of the work: jobs can be stolen from a threadpool by a **tcp_server_scheduler** from which other machines can get them. Just pack both pools in a `composite_threadpool_scheduler` and you're ready to go.

asio_scheduler

Asynchronous supports the possibility to use Boost.Asio as a threadpool provider. This has several advantages:

- asio_scheduler is delivered with a way to access Asio's io_service from a servant object living inside the scheduler.
- asio_scheduler handles the necessary work for creating a pool of threads for multithreaded-multi-io_service communication.
- asio_scheduler threads implement work-stealing from other Asynchronous schedulers. This allows communication threads to help other threadpools when no I/O communication is happening. This helps reducing thread oversubscription.
- One has all the usual goodies of Asynchronous: safe callbacks, object tracking, servant proxies, etc.

Let's create a simple but powerful example to illustrate its usage. We want to create a TCP client, which connects several times to the same server, gets data from it (in our case, the Boost license will do), then checks if the data is coherent by comparing the results two-by-two. Of course, the client has to be perfectly asynchronous and never block. We also want to guarantee some threads for the communication and some for the calculation work. We also want to communication threads to "help" by stealing some work if necessary.

Let's start by creating a TCP client using Boost.Asio. A slightly modified version of the async TCP client from the Asio documentation will do. All we change is pass it a callback which it will call when the requested data is ready. We now pack it into an Asynchronous trackable client:

```
// Objects of this type are made to live inside an asio_scheduler,
// they get their associated io_service object from TLS
struct AsioCommunicationServant : boost::asynchronous::trackable_servant<>
{
    AsioCommunicationServant(boost::asynchronous::any_weak_scheduler<> schedule
                             const std::string& server, const std::string& path
        : boost::asynchronous::trackable_servant<>(scheduler)
        , m_client(*boost::asynchronous::get_io_service<>(),server,path)
    {}
    void test(std::function<void(std::string)> cb)
    {
        // just forward call to asio asynchronous http client
        // the only change being the (safe) callback which will be called when
        m_client.request_content(cb);
    }
private:
    client m_client; //client is from Asio example
};
```

The main noteworthy thing to notice is the call to **boost::asynchronous::get_io_service<>()**, which, using thread-local-storage, gives us the io_service associated with this thread (one io_service per thread). This is needed by the Asio TCP client. Also noteworthy is the argument to `test()`, a callback when the data is available.

Wait a minute, is this not unsafe (called from an asio worker thread)? It is but it will be made safe in a minute.

We now need a proxy so that this communication servant can be safely used by others, as usual:

```
class AsioCommunicationServantProxy: public boost::asynchronous::servant_proxy<A
{
public:
```

```
    // ctor arguments are forwarded to AsioCommunicationServant
    template <class Scheduler>
    AsioCommunicationServantProxy(Scheduler s,const std::string& server, const
        boost::asynchronous::servant_proxy<AsioCommunicationServantProxy,AsioCo
    {}
    // we offer a single member for posting
    BOOST_ASYNC_POST_MEMBER(test)
};
```

A single member, `test`, is used in the proxy. The constructor takes the server and relative path to the desired page. We now need a manager object, which will trigger the communication, wait for data, check that the data is coherent:

```
struct Servant : boost::asynchronous::trackable_servant<>
{
    Servant(boost::asynchronous::any_weak_scheduler<> scheduler,const std::stri
        : boost::asynchronous::trackable_servant<>(scheduler)
        , m_check_string_count(0)
    {
        // as worker we use a simple threadpool scheduler with 4 threads (0 wou
        auto worker_tp = boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::threadpool_scheduler<boost::asynch

        // for tcp communication we use an asio-based scheduler with 3 threads
        auto asio_workers = boost::asynchronous::create_shared_scheduler_proxy(

        // we create a composite pool whose only goal is to allow asio worker t
        m_pools = boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::composite_threadpool_scheduler<> (

        set_worker(worker_tp);
        // we create one asynchronous communication manager in each thread
        m_asio_comm.push_back(AsioCommunicationServantProxy(asio_workers,server
        m_asio_comm.push_back(AsioCommunicationServantProxy(asio_workers,server
        m_asio_comm.push_back(AsioCommunicationServantProxy(asio_workers,server
    }
... //to be continued
```

We create 3 pools:

- A worker pool for calculations (page comparisons)
- An asio threadpool with 3 threads in which we create 3 communication objects.
- A composite pool which binds both pools together into one stealing unit. You could even set the worker pool to 0 thread, in which case the worker will get its work done when the asio threads have nothing to do. Only non- multiqueue schedulers support this. This composite pool is now made to be the worker pool of this object using `set_worker()`.

We then create our communication objects inside the asio pool.

Note: at the moment, asio pools can steal from other pools but not be stolen from. Let's move on to the most interesting part:

```
void get_data()
{
    // provide this callback (executing in our thread) to all asio servants as
    std::function<void(std::string)> f =
...
```

```
    m_asio_comm[0].test(make_safe_callback(f));
    m_asio_comm[1].test(make_safe_callback(f));
    m_asio_comm[2].test(make_safe_callback(f));
}
```

We skip the body of f for the moment. f is a task which will be posted to each communication servant so that they can do the same work:

- call the same http get on an asio servants
- at each callback, check if we got all three callbacks
- if yes, post some work to worker threadpool, compare the returned strings (should be all the same)
- if all strings equal as they should be, cout the page

All this will be doine in a single functor. This functor is passed to each communication servant, packed into a make_safe_callback, which, as its name says, transforms the unsafe functor into one which posts this callback functor to the manager thread and also tracks it to check if still alive at the time of the callback. By calling `test()`, we trigger the 3 communications, and f will be called 3 times. The body of f is:

```
std::function<void(std::string)> f =
                [this](std::string s)
                {
                   this->m_requested_data.push_back(s);
                   // poor man's state machine saying we got the result of our a
                   if (this->m_requested_data.size() == 3)
                   {
                       // ok, this has really been called for all servants, comp
                       // but it could be long, so we will post it to threadpoo
                       std::cout << "got all tcp data, parallel check it's corre
                       std::string s1 = this->m_requested_data[0];
                       std::string s2 = this->m_requested_data[1];
                       std::string s3 = this->m_requested_data[2];
                       // this callback (executing in our thread) will be calle
                       auto cb1 = [this,s1](boost::future<bool> res)
                       {
                          if (res.get())
                              ++this->m_check_string_count;
                          else
                              std::cout << "uh oh, the pages do not match, data
                          if (this->m_check_string_count ==2)
                          {
                              // we started 2 comparisons, so it was the last o
                              std::cout << "data has been confirmed, here it is
                              std::cout << s1;
                          }
                       };
                       auto cb2=cb1;
                       // post 2 string comparison tasks, provide callback where
                       this->post_callback([s1,s2](){return s1 == s2;},std::move
                       this->post_callback([s2,s3](){return s2 == s3;},std::move
                   }
                };
```

We start by checking if this is the third time this functor is called (this, the manager, is nicely serving as holder, kind of poor man's state machine counting to 3). If yes, we prepare a call to the worker pool to compare the 3 returned strings 2 by 2 (cb1, cb2). Again, simple state machine, if the callback is

called twice, we are done comparing string 1 and 2, and 2 and 3, in which case the page is confirmed and cout'ed. The last 2 lines trigger the work and post to our worker pool (which is the threadpool scheduler, or, if stealing happens, the asio pool) two comparison tasks and the callbacks.

Our manager is now ready, we still need to create for it a proxy so that it can be called from the outside world asynchronously, then create it in its own thread, as usual:

```
class ServantProxy : public boost::asynchronous::servant_proxy<ServantProxy,Serv
{
public:
    template <class Scheduler>
    ServantProxy(Scheduler s,const std::string& server, const std::string& path
        boost::asynchronous::servant_proxy<ServantProxy,Servant>(s,server,path)
    {}
    // get_data is posted, no future, no callback
    BOOST_ASYNC_POST_MEMBER(get_data)
};
...
auto scheduler = boost::asynchronous::create_shared_scheduler_proxy(
                                 new boost::asynchronous::single_thread_schedule
                                      boost::asynchronous::threadsafe_list<> >);
{
   ServantProxy proxy(scheduler,"www.boost.org","/LICENSE_1_0.txt");
   // call member, as if it was from Servant
   proxy.get_data();
   // if too short, no problem, we will simply give up the tcp requests
   // this is simply to simulate a main() doing nothing but waiting for a termi
   boost::this_thread::sleep(boost::posix_time::milliseconds(2000));
}
```

As usual, here the complete, ready-to-use example [examples/example_asio_http_client.cpp] and the implementation of the Boost.Asio HTTP client [examples/asio/asio_http_async_client.hpp].

Timers

Very often, an Active Object servant acting as an asynchronous dispatcher will post tasks which have to be done until a certain point in the future, or which will start only at a later point. State machines also regularly make use of a "time" event.

For this we need a timer, but a safe one:

- The timer callback has to be posted to the Active Object thread to avoid races.
- The timer callback shall not be called in the servant making the request has been deleted (it can be an awfully long time until the callback).

Asynchronous itself has no timer, but Boost.Asio has, so the library provides a wrapper around it and will allow us to create a timer using an io_service running in its own thread or in an asio threadpool, also provided by the library.

Constructing a timer

One first needs an `asio_scheduler` with at least one thread:

```
boost::asynchronous::any_shared_scheduler_proxy<> asio_sched = boost::asynchron
```

The Servant living in its ActiveObject thread then creates a timer (as attribute to keep it alive as destroying the object will cancel the timer) using this scheduler and a timer value:

```
boost::asynchronous::asio_deadline_timer_proxy m_timer (asio_sched,boost::posi
```

It can now start the timer using `trackable_servant` (its base class)`::async_wait`, passing it a functor call when timer expires / is cancelled:

```
async_wait(m_timer,
           [](const ::boost::system::error_code& err)
           {
               std::cout << "timer expired? "<< std::boolalpha << (bool)err <<
           }
           );
```

Canceling the timer means destroying (and possibly recreating) the timer object:

```
m_timer =  boost::asynchronous::asio_deadline_timer_proxy(get_worker(),boost::
```

The following example [examples/example_asio_deadline_timer.cpp] displays a servant using an asio scheduler as a thread pool and creating there its timer object. Not how the timer is created using `trackable_servant` (its base class)`::get_worker()`.

Continuation tasks

A common limitation of threadpools is support for recursive tasks: tasks start other tasks, which start other tasks, until all threads in the threadpool are busy waiting. At this point, one could add more threads, but threads are expensive. Similarly, you might post a task which posts more tasks and wait for them to complete to do a merge of the part-results. Of course you can achieve this with a controller object or state machine in a single-threaded scheduler waiting for callbacks, but for very small tasks, using callbacks might just be too expensive. In such cases, Asynchronous provides continuations: a task executes, does something then creates a continuation which will wake up when ready.

A common example of recursive tasks is a parallel fibonacci. Usually, this means a task calculating fib(n) will start a fib(n-1) and fib(n-2) and blocks until both are done. These tasks will start more tasks, etc. until a cutoff number, at which point recursion stops and fibonacci is calculated serially. This approach has some problems: to avoid thread explosion, we would need fibers, which are not available in Boost at the time of this writing, and even in fibers, tasks would block, which means interrupting them is not possible, which we would want to avoid. In any case, blocking simply isn't part of the asynchronous philosophy of the library. Let's have a look how continuation tasks let us implement a parallel fibonacci.

First of all, we need a serial fibonacci to use for the cutoff. This is a classical one:

```
long serial_fib( long n ) {
   if( n<2 )
       return n;
   else
       return serial_fib(n-1)+serial_fib(n-2);
}
```

We now need a recursive fibonacci task:

```
// our recursive fibonacci tasks. Needs to inherit continuation_task<value type
struct fib_task : public boost::asynchronous::continuation_task<long>
{
    fib_task(long n,long cutoff):n_(n),cutoff_(cutoff){}
    // called inside of threadpool
    void operator()()const
    {
        // the result of this task, will be either set directly if < cutoff, ot
```

```
        boost::asynchronous::continuation_result<long> task_res = this_task_resu
        if (n_<cutoff_)
        {
            // n < cutoff => execute immediately
            task_res.set_value(serial_fib(n_));
        }
        else
        {
            // n>= cutoff, create 2 new tasks and when both are done, set our re
            boost::asynchronous::create_continuation(
                        // called when subtasks are done, set our result
                        [task_res](std::tuple<boost::future<long>,boost::future<
                        {
                            long r = std::get<0>(res).get() + std::get<1>(res).g
                            task_res.set_value(r);
                        },
                        // recursive tasks
                        fib_task(n_-1,cutoff_),
                        fib_task(n_-2,cutoff_));
        }
    }
    long n_;
    long cutoff_;
};
```

Our task need to inherit `boost::asynchronous::continuation_task<R>` where R is the type later returned. This class provides us with `this_task_result()` where we set the task result. This is done either immediately if n < cutoff (first if clause), or (else clause) using a continuation.

If n>= cutoff, we create a continuation task. This is a sleeping task, which will get activated when all required tasks complete. In this case, we have two fibonacci sub tasks. The template argument is the return type of the continuation. We create two sub-tasks, for n-1 and n-2 and when they complete, the completion functor passed as first argument is called.

Note that `boost::asynchronous::create_continuation` is a variadic function, there can be any number of sub-tasks. The completion functor takes as single argument a tuple of futures, one for each subtask. The template argument of the future is the template argument of `boost::asynchronous::continuation_task` of each subtask. In this case, all are long, but it's not a requirement.

When this completion functor is called, we set our result to be result of first task + result of second task and return.

The main particularity of this solution is that a task does not block until sub-tasks complete but instead provides an asynchronous functor.

All what we still need to do is create the first task. In the tradition of Asynchronous, we do it inside an asynchronous servant which posts the first task and waits for a callback:

```
struct Servant : boost::asynchronous::trackable_servant<>
{
...
   void calc_fibonacci(long n,long cutoff)
   {
      post_callback(
            [n,cutoff]()
            {
                // a top-level continuation is the first one in a recursive seri
                // Its result will be passed to callback
                return boost::asynchronous::top_level_continuation<long>(fib_tas
```

```
            }// work
            ,
            // callback with fibonacci result.
            [](boost::future<long> res){...}// callback functor.
        );
    }
};
```

We call `post_callback`, which, as usual, ensures that the callback is posted to the right thread and the servant lifetime is tracked. The posted task calls `boost::asynchronous::top_level_continuation<task-return-type>` to create the first, top-level continuation, passing it a first fib_task. This is non-blocking, a special version of `post_callback` recognizes a continuation and will call its callback (with a `future<task-return-type>`) only when the calculation is finished.

As usual, calling get() on the future is non-blocking, one gets either the result or an exception if thrown by a task.

Please have a look at the complete example [examples/example_fibonacci.cpp].

And what about logging? We don't want to give up this feature of course and would like to know how long all these fib_task took to complete. This is done through minor changes. As always we need a job:

```
typedef boost::asynchronous::any_loggable<boost::chrono::high_resolution_clock>
```

We give the logged name of the task in the constructor of fib_task, for example fib_task_xxx:

```
fib_task(long n,long cutoff)
        : boost::asynchronous::continuation_task<long>("fib_task_" + boost::lexi
        ,n_(n),cutoff_(cutoff){}
```

And call `boost::asynchronous::create_continuation_job` instead of `boost::asynchronous::create_continuation`:

```
boost::asynchronous::create_continuation_job<servant_job>(
                        [task_res](std::tuple<boost::future<long>,boost::future<
                        {
                            long r = std::get<0>(res).get() + std::get<1>(res).g
                            task_res.set_value(r);
                        },
                        fib_task(n_-1,cutoff_),
                        fib_task(n_-2,cutoff_)
);
```

Inside the servant we might optionally want the version of post_callback with name, and we need to use `top_level_continuation_job` instead of `top_level_continuation`:

```
post_callback(
              [n,cutoff]()
              {
                   return boost::asynchronous::top_level_continuation_job<long,s
              }// work
              ,
              // the lambda calls Servant, just to show that all is safe, Servan
              [this](boost::future<long> res){...},// callback functor.
              "calc_fibonacci"
        );
```

The previous example has been rewritten with logs and a display of all tasks [examples/example_fibonacci_log.cpp] (beware, with higher fibonacci numbers, this can become a long list).

Limitation: in the current implementation, tasks are logged, but the continuation callback is not. If it might take long, post a (loggable) task.

Distributing work among machines

At the time of this writing, a core i7-3930K with 6 cores and 3.2 GHz will cost $560, so say $100 per core. Not a bad deal, so you buy it. Unfortunately, some time later you realize you need more power. Ok, there is no i7 with more cores and an Extreme Edition will be quite expensive for only a little more power so you decide to go for a Xeon. A 12-core E5-2697v2 2.7GHz will go for almost $3000 which means $250 per core, and for this you have a lesser frequency. And if you need later even more power, well, it will become really expensive. Can Asynchronous help us use more power for cheap, and at best, with little work? It does, as you guess ;-)

Asynchronous provides a special pool, `tcp_server_scheduler`, which will behave like any other scheduler but will not execute work itself, waiting instead for clients to connect and get some work. The client execute the work on behalf of the `tcp_server_scheduler` and send it back the results.

For this to work, there is however a condition: jobs must be (boost) serializable to be transferred to the client. So does the returned value.

Let's start with a simplest example [examples/example_tcp_server.cpp]:

```
// notice how the worker pool has a different job type
struct Servant : boost::asynchronous::trackable_servant<boost::asynchronous::any
{
  Servant(boost::asynchronous::any_weak_scheduler<> scheduler)
        : boost::asynchronous::trackable_servant<boost::asynchronous::any_callab
  {
        // let's build our pool step by step. First we need a worker pool
        // possibly for us, and we want to share it with the tcp pool for its s
        boost::asynchronous::any_shared_scheduler_proxy<> workers = boost::asyn
            new boost::asynchronous::threadpool_scheduler<boost::asynchronous::

        // we use a tcp pool using the 3 worker threads we just built
        // our server will listen on "localhost" port 12345
        auto pool= boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::tcp_server_scheduler<
                            boost::asynchronous::lockfree_queue<boost::asynchro
                                (workers,"localhost",12345));
        // and this will be the worker pool for post_callback
        set_worker(pool);
  }
};
```

We start by creating a worker pool. The `tcp_server_scheduler` will delegate to this pool all its serialization / deserialization work. For maximum scalability we want this work to happen in more than one thread.

Note that our job type is no more a simple callable, it must be (de)serializable too (**boost::asynchronous::any_serializable**).

Then we need a `tcp_server_scheduler` listening on, in this case, localhost, port 12345. We now have a functioning worker pool and choose to use it as our worker pool so that we do not execute jobs ourselves (other configurations will be shown soon). Let's exercise our new pool. We first need a task to be executed remotely:

```
struct dummy_tcp_task : public boost::asynchronous::serializable_task
{
    dummy_tcp_task(int d):boost::asynchronous::serializable_task("dummy_tcp_tas
```

```
    template <class Archive>
    void serialize(Archive & ar, const unsigned int /*version*/)
    {
        ar & m_data;
    }
    int operator()()const
    {
        std::cout << "dummy_tcp_task operator(): " << m_data << std::endl;
        boost::this_thread::sleep(boost::posix_time::milliseconds(2000));
        std::cout << "dummy_tcp_task operator() finished" << std::endl;
        return m_data;
    }
    int m_data;
};
```

This is a minimum task, only sleeping. All it needs is a `serialize` member to play nice with Boost.Serialization and it must inherit `serializable_task`. Giving the task a name is essential as it will allow the client to deserialize it. Let's post to our TCP worker pool some of the tasks, wait for a client to pick them and use the results:

```
// start long tasks in threadpool (first lambda) and callback in our thread
for (int i =0 ;i < 10 ; ++i)
{
    std::cout << "call post_callback with i: " << i << std::endl;
    post_callback(
           dummy_tcp_task(i),
           // the lambda calls Servant, just to show that all is safe, Servant
           [this](boost::future<int> res){
                  try{
                        this->on_callback(res.get());
                  }
                  catch(std::exception& e)
                  {
                       std::cout << "got exception: " << e.what() << std::endl;
                       this->on_callback(0);
                  }
            }// callback functor.
    );
}
```

We post 10 tasks to the pool. For each task we will get, at some later undefined point (provided some clients are around), a result in form of a (ready) future, possibly an exception if one was thrown by the task.

Notice it is safe to use `this` in the callback lambda as it will be only called if the servant still exists.

We still need a client to execute the task, this is pretty straightforward (we will extend it soon):

```
int main(int argc, char* argv[])
{
    std::string server_address = (argc>1) ? argv[1]:"localhost";
    std::string server_port = (argc>2) ? argv[2]:"12346";
    int threads = (argc>3) ? strtol(argv[3],0,0) : 4;
    cout << "Starting connecting to " << server_address << " port " << server_p

    auto scheduler = boost::asynchronous::create_shared_scheduler_proxy(
                new boost::asynchronous::asio_scheduler<>);
    {
        std::function<void(std::string const&,boost::asynchronous::tcp::server_
```

```
        executor=
        [](std::string const& task_name,boost::asynchronous::tcp::server_reponse
           std::function<void(boost::asynchronous::tcp::client_request const&)>
        {
            if (task_name=="dummy_tcp_task")
            {
                dummy_tcp_task t(0);
                boost::asynchronous::tcp::deserialize_and_call_task(t,resp,when_
            }
            else
            {
                std::cout << "unknown task! Sorry, don't know: " << task_name <<
                throw boost::asynchronous::tcp::transport_exception("unknown tas
            }
        };

        auto pool = boost::asynchronous::create_shared_scheduler_proxy(
                        new boost::asynchronous::threadpool_scheduler<
                            boost::asynchronous::lockfree_queue<boost::asynchron
        boost::asynchronous::tcp::simple_tcp_client_proxy proxy(scheduler,pool,s
                                                                0/*ms betwee
        boost::future<boost::future<void> > fu = proxy.run();
        boost::future<void> fu_end = fu.get();
        fu_end.get();
    }
    return 0;
}
```

We start by taking as command-line arguments the server address and port and the number of threads the client will use to process stolen work from the server.

We create a single-threaded `asio_scheduler` for the communication (in our case, this is sufficient, your case might vary) to the server.

The client then defines an executor function. This function will be called when a job is stolen by the client. As Asynchronous does not know what the job is, you will need to "help" by creating an instance of the task using its name. Calling `deserialize_and_call_task` will, well, deserialize the task data into our dummy task, then call it. We also choose to return an exception is the task is not known to us.

Next, we need a pool of threads to execute the work. Usually, you will want more than one thread. Remember, we want to use our several 6-core-i7s, right?

The simplest client that Asynchronous offers is a `simple_tcp_client_proxy` proxy. We say simple, because it is only a client. Later on, we will see a more powerful tool. `simple_tcp_client_proxy` will require the asio pool for communication, the server address and port, our executor and a parameter telling it how often it should try to steal work from a server.

We are now done, the client will run until killed.

Let's sum up what we got in these few lines of code:

- a pool behaving like any other pool, which can be stolen from
- a server which does no work itself, but still scales well as serialization is using whatever threads it is given
- a trackable servant working with `post_callback`, like always
- a multithreaded client, which can be tuned precisely to use a given pool for the communication and another (or the same btw.) for work processing.

Interestingly, we have a very versatile client. It is possible to reuse the work processing and communication pools, within the same client application, for a different `simple_tcp_client_proxy` which would be connecting to another server.

The server is also quite flexible. It scales well and can handle as many clients as you wish.

This is only the beginning of our distributed chapter. Let's now revisit our Fibonacci example to make it distributed too.

A distributed, parallel Fibonacci

Lets's revisit our parallel Fibonacci example. We realize that with higher Fibonacci numbers, our CPU power doesn't suffice any more. We want to distribute it among several machines while our main machine still does some calculation work. To do this, we'll start with our previous example, and rewrite our Fibonacci task to make it distributable.

We remember that we first had to call `boost::asynchronous::top_level_continuation` in our post_callback to make Asynchronous aware of the later return value. The difference now is that even this one-liner lambda could be serialized and sent away, wo we need to make it a `serializable_task`:

```
struct serializable_fib_task : public boost::asynchronous::serializable_task
{
    serializable_fib_task(long n,long cutoff):boost::asynchronous::serializable
    template <class Archive>
    void serialize(Archive & ar, const unsigned int /*version*/)
    {
        ar & n_;
        ar & cutoff_;
    }
    auto operator()()const
        -> decltype(boost::asynchronous::top_level_continuation_log<long,boost:
                    (tcp_example::fib_task(long(0),long(0))))
    {
        auto cont =  boost::asynchronous::top_level_continuation_job<long,boost
                (tcp_example::fib_task(n_,cutoff_));
        return cont;
    }
    long n_;
    long cutoff_;
};
```

We need to make our task serializable and give it a name so that the client application can recognize it. We also need a serialize member, as required by Boost.Serialization. And we need an operator() so that the task can be executed. There is in C++11 an ugly decltype, but C++14 will solve this if your compiler supports it. We also need a few changes in our Fibonacci task:

```
// our recursive fibonacci tasks. Needs to inherit continuation_task<value type
struct fib_task : public boost::asynchronous::continuation_task<long>
                , public boost::asynchronous::serializable_task
{
    fib_task(long n,long cutoff)
        :  boost::asynchronous::continuation_task<long>()
        , boost::asynchronous::serializable_task("serializable_sub_fib_task")
        ,n_(n),cutoff_(cutoff)
    {
    }
    template <class Archive>
    void save(Archive & ar, const unsigned int /*version*/)const
```

```
    {
        ar & n_;
        ar & cutoff_;
    }
    template <class Archive>
    void load(Archive & ar, const unsigned int /*version*/)
    {
        ar & n_;
        ar & cutoff_;
    }
    BOOST_SERIALIZATION_SPLIT_MEMBER()
    void operator()()const
    {
        // the result of this task, will be either set directly if < cutoff, otherwise
        boost::asynchronous::continuation_result<long> task_res = this_task_result();
        if (n_<cutoff_)
        {
            // n < cutoff => execute ourselves
            task_res.set_value(serial_fib(n_));
        }
        else
        {
            // n>= cutoff, create 2 new tasks and when both are done, set our result
            boost::asynchronous::create_continuation_log<boost::asynchronous::any_
                        // called when subtasks are done, set our result
                        [task_res](std::tuple<boost::future<long>,boost::future<long
                        {
                            long r = std::get<0>(res).get() + std::get<1>(res).get
                            task_res.set_value(r);
                        },
                        // recursive tasks
                        fib_task(n_-1,cutoff_),
                        fib_task(n_-2,cutoff_));
        }
    }
    long n_;
    long cutoff_;
};
```

The few changes are highlighted. It needs to be a serializable task with its own name in the constructor, and it needs serialization members. That's it, we're ready to distribute!

As we previously said, we will reuse our previous TCP example, using `serializable_fib_task` as the main posted task. This gives us this example [examples/example_tcp_server_fib.cpp].

But wait, we promised that our server would itself do some calculation work, and we use as worker pool only a `tcp_server_scheduler`. Right, let's do it now, throwing in a few more goodies. We need a worker pool, with as many threads as we are willing to offer:

```
// we need a pool where the tasks execute
auto pool = boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::threadpool_scheduler<
                    boost::asynchronous::lockfree_queue<boost::asynchronous::any
```

This pool will get the fibonacci top-level task we will post, then, if our clients connect after we start, it will get the first sub-tasks.

To make it more interesting, let's offer our server to also be a job client. This way, we can build a cooperation network: the server offers fibonacci tasks, but also tries to steal some, thus increasing homogenous work distribution. We'll talk more about this in the next chapter.

```
// a client will steal jobs in this pool
auto cscheduler = boost::asynchronous::create_shared_scheduler_proxy(new boost:
// jobs we will support
std::function<void(std::string const&,boost::asynchronous::tcp::server_reponse,
                   std::function<void(boost::asynchronous::tcp::client_request
        [](std::string const& task_name,boost::asynchronous::tcp::server_repons
           std::function<void(boost::asynchronous::tcp::client_request const&)>
        {
            if (task_name=="serializable_sub_fib_task")
            {
                tcp_example::fib_task fib(0,0);
                boost::asynchronous::tcp::deserialize_and_call_continuation_tas
            }
            else if (task_name=="serializable_fib_task")
            {
                tcp_example::serializable_fib_task fib(0,0);
                boost::asynchronous::tcp::deserialize_and_call_top_level_contin
            }
            // else whatever functor we support
            else
            {
                std::cout << "unknown task! Sorry, don't know: " << task_name <<
                throw boost::asynchronous::tcp::transport_exception("unknown ta
            }
        };
boost::asynchronous::tcp::simple_tcp_client_proxy client_proxy(cscheduler,pool,
                                                                10/*ms between ca
```

Notice how we use our worker pool for job serialization / deserialization. Notice also how we check both possible stolen jobs.

We also introduce two new deserialization functions. boost::asynchronous::tcp::**deserialize_and_call_task** was used for normal tasks, we now have boost::asynchronous::tcp::**deserialize_and_call_top_level_continuation_task** for our top-level continuation task, and boost::asynchronous::tcp::**deserialize_and_call_continuation_task** for the continuation-sub-task.

We now need to build our TCP server, which we decide will get only one thread for task serialization. This ought to be enough, Fibonacci tasks have little data (2 long).

```
// we need a server
// we use a tcp pool using 1 worker
auto server_pool = boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::threadpool_scheduler<
                            boost::asynchronous::lockfree_queue<> >(1));

auto tcp_server= boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::tcp_server_scheduler<
                            boost::asynchronous::lockfree_queue<boost::asynchro
                            boost::asynchronous::any_callable,true>
                                (server_pool,own_server_address,(unsigned int)ou
```

We have a TCP server pool, as before, even a client to steal work ourselves, but how do we get ourselves this combined pool, which executes some work or gives some away?

Wait a minute, combined pool? Yes, a `composite_threadpool_scheduler` will do the trick. As we're at it, we create a servant to coordinate the work, as we now always do:

```
// we need a composite for stealing
```

```
auto composite = boost::asynchronous::create_shared_scheduler_proxy
                (new boost::asynchronous::composite_threadpool_scheduler<boost:
                          (pool,tcp_server));

// a single-threaded world, where Servant will live.
auto scheduler = boost::asynchronous::create_shared_scheduler_proxy(
                                new boost::asynchronous::single_thread_schedule
                                     boost::asynchronous::lockfree_queue<> >);
{
      ServantProxy proxy(scheduler,pool);
      // result of BOOST_ASYNC_FUTURE_MEMBER is a shared_future,
      // so we have a shared_future of a shared_future(result of start_async_wo
      boost::shared_future<boost::shared_future<long> > fu = proxy.calc_fibonac
      boost::shared_future<long> resfu = fu.get();
      long res = resfu.get();
}
```

Notice how we give only the worker "pool" to the servant. This means, the servant will post the top-level task to it, it will immediately be called and create 2 Fibonacci tasks, which will create each one 2 more, etc. until at some point a client connects and steals one, which will create 2 more, etc.

The client will not steal directly from this pool, it will steal from the `tcp_server` pool, which, as long as a client request comes, will steal from the worker pool, as they belong to the same composite. This will continue until the composite is destroyed, or the work is done. For the sake of the example, we do not give the composite as the Servant's worker pool but keep it alive until the end of calculation. Please have a look at the complete example [examples/example_tcp_server_fib2.cpp].

In this example, we start taking care of homogenous work distribution by packing a client and a server in the same application. But we need a bit more: our last client would steal work so fast, every 10ms that it would starve the server or other potential client applications, so we're going to tell it to only steal if its work queues goes under a certain amount, which we will empirically determine, according to our hardware, network speed, etc.

```
int main(int argc, char* argv[])
{
    std::string server_address = (argc>1) ? argv[1]:"localhost";
    std::string server_port = (argc>2) ? argv[2]:"12346";
    int threads = (argc>3) ? strtol(argv[3],0,0) : 4;
    // 1..n => check at regular time intervals if the queue is under the given
    int job_getting_policy = (argc>4) ? strtol(argv[4],0,0):0;
    cout << "Starting connecting to " << server_address << " port " << server_p

    auto scheduler = boost::asynchronous::create_shared_scheduler_proxy(
                new boost::asynchronous::asio_scheduler<>);
    {
        std::function<void(std::string const&,boost::asynchronous::tcp::server_
        executor=
        [](std::string const& task_name,boost::asynchronous::tcp::server_reponse
           std::function<void(boost::asynchronous::tcp::client_request const&)>
        {
            if (task_name=="serializable_fib_task")
            {
                tcp_example::serializable_fib_task fib(0,0);
                boost::asynchronous::tcp::deserialize_and_call_top_level_contin
            }
            else if (task_name=="serializable_sub_fib_task")
            {
                tcp_example::fib_task fib(0,0);
                boost::asynchronous::tcp::deserialize_and_call_continuation_tas
```

```
            }
            else
            {
                std::cout << "unknown task! Sorry, don't know: " << task_name <<
                throw boost::asynchronous::tcp::transport_exception("unknown tas
            }
        };

        // guarded_deque supports queue size
        auto pool = boost::asynchronous::create_shared_scheduler_proxy(
                        new boost::asynchronous::threadpool_scheduler<
                            boost::asynchronous::guarded_deque<boost::asynchrono
        // more advanced policy
        // or simple_tcp_client_proxy<boost::asynchronous::tcp::queue_size_check
        typename boost::asynchronous::tcp::get_correct_simple_tcp_client_proxy<
                        scheduler,pool,server_address,server_port,executor,
                        0/*ms between calls to server*/,
                        job_getting_policy /* number of jobs we try to keep in
        // run forever
        boost::future<boost::future<void> > fu = proxy.run();
        boost::future<void> fu_end = fu.get();
        fu_end.get();
    }
    return 0;
}
```

The important new part is highlighted. `simple_tcp_client_proxy` gets an extra template argument, `queue_size_check_policy`, and a new constructor argument, the number of jobs in the queue, under which the client will try, every 10ms, to steal a job. Normally, that would be all, but g++ (up to 4.7 at least) is uncooperative and requires an extra level of indirection to get the desired client proxy. Otherwise, there is no change.

Notice that our standard lockfree queue offers no size() so we use a less efficient guarded_deque.

You will find in the complete example [examples/simple_tcp_client.cpp] a few other tasks which we will explain shortly.

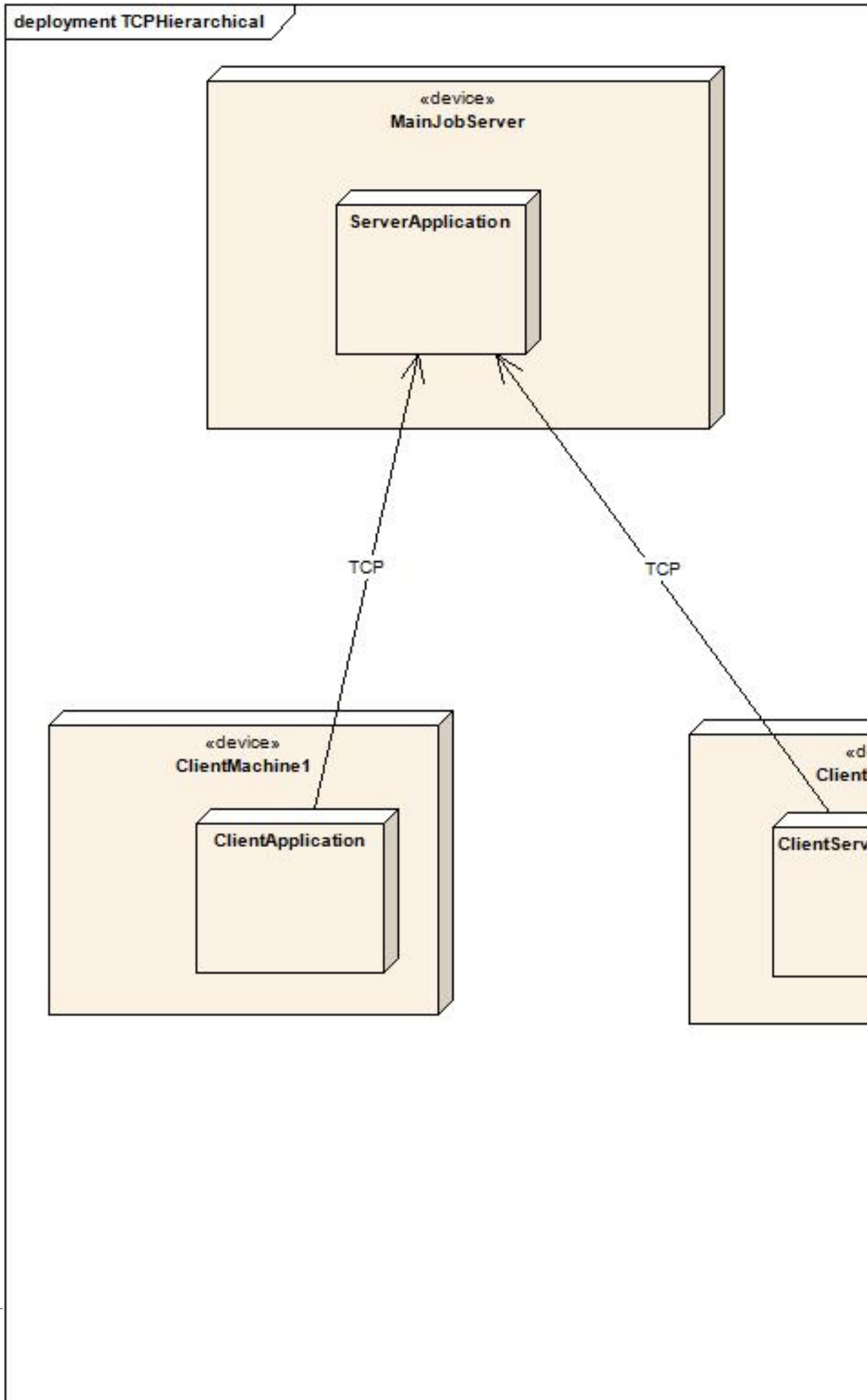
Let's stop a minute to think about what we just did. We built, with little code, a complete framework for distributing tasks homogenously among machines, by reusing standard component offered by the library: threadpools, composite pools, clients, servers. If we really have client connecting or not is secondary, all what can happen is that calculating our Fibonacci number will last a little longer.

We also separate the task (Fibonacci) from the threadpool configuration, from the network configuration, and from the control of the task (Servant), leading us to highly reusable, extendable code.

In the next chapter, we will add a way to further distribute work among not only machines, but whole networks.

Example: a hierarchical network

We already distribute and parallelize work, so we can scale a great deal, but our current model is one server, many clients, which means a potentially high network load and a lesser scalability as more and more clients connect to a server. What we want is a client/server combo application where the client steals and executes jobs and a server component of the same application which steals jobs from the client on behalf of other clients. What we want is to achieve something like this:


deployment TCPHierarchical
«device»
MainJobServer
ServerApplication
TCP
TCP
«device»
ClientMachine1
ClientApplication

We have our server application, as seen until now, called interestingly ServerApplication on a machine called MainJobServer. This machine executes work and offers at the same time a steal-from capability. We also have a simple client called ClientApplication running on ClientMachine1, which steals jobs and executes them itself without further delegating. We have another client machine called ClientMachine2 on which ClientServerApplication runs. This applications has two parts, a client stealing jobs like ClientApplication and a server part stealing jobs from the client part upon request. For example, another simple ClientApplication running on ClientMachine2.1 connects to it and steals further jobs in case ClientMachine2 is not executing them fast enough, or if ClientMachine2 is only seen as a pass-through to move jobs execution to another network. Sounds scalable. How hard is it to build? Not so hard, because in fact, we already saw all we need to build this, so it's kind of a Lego game.

```
int main(int argc, char* argv[])
{
    std::string server_address = (argc>1) ? argv[1]:"localhost";
    std::string server_port = (argc>2) ? argv[2]:"12345";
    std::string own_server_address = (argc>3) ? argv[3]:"localhost";
    long own_server_port = (argc>4) ? strtol(argv[4],0,0):12346;
    int threads = (argc>5) ? strtol(argv[5],0,0) : 4;
    cout << "Starting connecting to " << server_address << " port " << server_p
         << " listening on " << own_server_address << " port " << own_server_po

// to be continued
```

We take as arguments the address and port of the server we are going to steal from, then our own address and port. We now need a client with its communication asio scheduler and its threadpool for job execution.

```
auto scheduler = boost::asynchronous::create_shared_scheduler_proxy(new boost::a
    { //block start
        std::function<void(std::string const&,boost::asynchronous::tcp::server_
                           std::function<void(boost::asynchronous::tcp::client_
        [](std::string const& task_name,boost::asynchronous::tcp::server_reponse
           std::function<void(boost::asynchronous::tcp::client_request const&)>
        {
            if (task_name=="serializable_fib_task")
            {
                tcp_example::serializable_fib_task fib(0,0);
                boost::asynchronous::tcp::deserialize_and_call_top_level_continu
            }
            else if (task_name=="serializable_sub_fib_task")
            {
                tcp_example::fib_task fib(0,0);
                boost::asynchronous::tcp::deserialize_and_call_continuation_task
            }
            // else whatever functor we support
            else
            {
                std::cout << "unknown task! Sorry, don't know: " << task_name <<
                throw boost::asynchronous::tcp::transport_exception("unknown tas
            }
        };
        // create pools
        // we need a pool where the tasks execute
        auto pool = boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::threadpool_scheduler<
                            boost::asynchronous::lockfree_queue<boost::asynchron
        boost::asynchronous::tcp::simple_tcp_client_proxy client_proxy(schedule
                                                                       10/*ms b

// to be continued
```

We now need a server to which more clients will connect, and a composite binding it to our worker pool:

```
   // we need a server
   // we use a tcp pool using 1 worker
   auto server_pool = boost::asynchronous::create_shared_scheduler_proxy(
                   new boost::asynchronous::threadpool_scheduler<
                           boost::asynchronous::lockfree_queue<> >(1));
   auto tcp_server= boost::asynchronous::create_shared_scheduler_proxy(
                   new boost::asynchronous::tcp_server_scheduler<
                           boost::asynchronous::lockfree_queue<boost::asynchro
                           boost::asynchronous::any_callable,true>
                               (server_pool,own_server_address,(unsigned int)ou
   // we need a composite for stealing
   auto composite = boost::asynchronous::create_shared_scheduler_proxy(new boos
                                                                       (pool,tcp_se

   boost::future<boost::future<void> > fu = client_proxy.run();
   boost::future<void> fu_end = fu.get();
   fu_end.get();
} //end block

 return 0;
 } //end main
```

And we're done! The client part will steal jobs and execute them, while the server part, bound to the client pool, will steal on sub-client-demand. Please have a look at the complete code [examples/tcp_client_server.cpp].

Picking your archive

By default, Asynchronous uses a Boost Text archive (text_oarchive, text_iarchive), which is simple and efficient enough for our Fibonacci example, but inefficient for tasks holding more data.

Asynchronous supports any archive task, requires however a different job type for this. At the moment, we can use a `portable_binary_oarchive`/`portable_binary_iarchive` by selecting `any_bin_serializable` as job. If Boost supports more archive types, support is easy to add.

The previous Fibonacci server example has been rewritten [examples/example_tcp_server_fib2_bin.cpp] to use this capability. The client [examples/simple_tcp_client_bin_archive.cpp] has also been rewritten using this new job type.

Parallel Algorithms (Christophe Henry / Tobias Holl)

Asynchronous supports out of the box some asynchronous parallel algorithms, with more to come, as well as interesting combination usages. These algorithms are continuation-based for simpler usage, often faster callback-based ones will be added later. All these algorithms also support distributed calculations as long as the user-provided functors are (meaning they must be serializable).

What is the point of adding yet another set of parallel algorithms which can be found elsewhere? Because truly asynchronous algorithms are hard to find. By this we mean non-blocking. If one needs parallel algorithms, it's because they could need long to complete. And if they take long, we really do not want to block until it happens.

All of the algorithms are made for use in a worker threadpool. They represent the work part of a `post_callback`;

In the philosophy of Asynchronous, the programmer knows better the task size where he wants to start parallelizing, so all these algorithms take a cutoff. Work is cut into packets of this size.

All range algorithms also have a version taking a continuation as range argument. This allows to combine algorithms functional way, for example this (more to come):

```
return parallel_for(parallel_for(parallel_for(...)));
```

parallel_for

There are four versions of this algorithm:

```
template <class Iterator, class Func, class Job=boost::asynchronous::any_callab
boost::asynchronous::detail::continuation<void,Job>
parallel_for(Iterator beg, Iterator end,Func func,long cutoff,const std::string&
```

The parallel_for version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

The third argument is the predicate applied on each element of the algorithm.

The fourth argument is the cutoff, meaning in this case the max. number of elements of the input range in a task.

The optional fifth argument is the name of the tasks used for logging.

The optional sixth argument is the priority of the tasks in the pool.

The return value is a void continuation containing either nothing or an exception if one was thrown from one of the tasks.

Example:

```
struct Servant : boost::asynchronous::trackable_servant<>
{
...
    void start_async_work()
    {
        // start long tasks in threadpool (first lambda) and callback in our th
        post_callback(
               [this](){
                        return boost::asynchronous::parallel_for(this->m_data.b
                                                                 [](int const& i
                                                                 {
                                                                    const_cast<i
                                                                 },1500);
                      },// work
               // the lambda calls Servant, just to show that all is safe, Serva
               [](boost::future<void> /*res*/){
                           ...
               }// callback functor.
        );
    }
    std::vector<int> m_data;
};
```

The most important parts are highlighted. Do not forget the return statement as we are returning a continuation and we do not want the lambda to be interpreted as a void lambda. The caller has responsibility of the input data, given in the form of iterators. We use a non-legal modifying functor for the sake of the example.

The call will do following:

- start tasks in the current worker pool of max 1500 elements of the input data
- add 2 to each element in parallel
- return a continuation
- Execute the callback lambda when all tasks complete. The future will be either set or contain an exception

Please have a look at the complete example [examples/example_parallel_for.cpp].

The second version is very similar and takes a range per reference. Again, the range has to stay valid during the call. As previously, the return value is a void continuation.

```
template <class Range, class Func, class Job=boost::asynchronous::any_callable>
boost::asynchronous::detail::continuation<void,Job>
parallel_for(Range const& range,Func func,long cutoff,const std::string& task_na
```

The third version takes a range per rvalue reference. This is signal given to Asynchronous that it must take ownership of the range. The return value is then a continuation of the given range type:

```
template <class Range, class Func, class Job=boost::asynchronous::any_callable>
boost::asynchronous::detail::continuation<Range,Job>
parallel_for(Range&& range,Func func,long cutoff,const std::string& task_name="
```

A `post_callback` will therefore get a future<new range>, for example:

```
post_callback(
    [](){
       std::vector<int> data;
       return boost::asynchronous::parallel_for(std::move(data),
                                                []( int const& i)
                                                {
                                                  const_cast<int&>(i) += 
                                                },1500);
    },
    ](boost::future<std::vector<int>> ){}
);
```

In this case, the programmer does not need to ensure the container stays valid, Asynchronous takes care of it.

The fourth version of this algorithm takes a range continuation instead of a range as argument and will be invoked after the continuation is ready.

```
// version taking a continuation of a range as first argument
template <class Range, class Func, class Job=boost::asynchronous::any_callable>
boost::asynchronous::detail::continuation<typename Range::return_type,Job>
parallel_for(Range range,Func func,long cutoff,const std::string& task_name="",
```

This version allows chaining parallel calls. For example, it is now possible to write:

```
post_callback(
    [](){
       std::vector<int> data;
       return parallel_for(parallel_for(parallel_for(
                                                        // executed first
```

```
                                                                    std::move(data),
                                                                    [](int const& i)
                                                                    {
                                                                       const_cast<int&>
                                                                    },1500),
                                                            // executed second
                                                            [](int const& i)
                                                            {
                                                                const_cast<int&>(i) += 2;
                                                            },1500),
                                                    // executed third
                                                    [](int const& i)
                                                    {
                                                         const_cast<int&>(i) += 2;
                                                    },1500);
        },
        ](boost::future<std::vector<int>> ){} // callback
);
```

This code will be executed as follows:

- the most inner parallel_for (parallel execution)
- A kind of synchronization point will be done at this point until the parallel_for completes
- the middle parallel_for will be executed (parallel execution)
- A kind of synchronization point will be done at this point until the parallel_for completes
- the outer parallel_for will be executed (parallel execution)
- A kind of synchronization point will be done at this point until the parallel_for completes
- The callback will be called

With "kind of synchronization point", we mean there will be no blocking synchronization, it will just be waited until completion.

Finally, we also promised some distributed support, so here it is. We need, as with our Fibonacci example, a serializable sub-task which will be created as often as required by our cutoff and which will handle a part of our range:

```
struct dummy_parallel_for_subtask : public boost::asynchronous::serializable_ta
{
    dummy_parallel_for_subtask(int d=0):boost::asynchronous::serializable_task(
    template <class Archive>
    void serialize(Archive & ar, const unsigned int /*version*/)
    {
        ar & m_data;
    }
    void operator()(int const& i)const
    {
        const_cast<int&>(i) += m_data;
    }
    // some data, so we have something to serialize
    int m_data;
};
```

As always we need a serializable top-level task, creating sub-tasks:

```
struct dummy_parallel_for_task : public boost::asynchronous::serializable_task
```

```
{
    dummy_parallel_for_task():boost::asynchronous::serializable_task("dummy_para
    template <class Archive>
    void serialize(Archive & ar, const unsigned int /*version*/)
    {
        ar & m_data;
    }
    auto operator()() -> decltype(boost::asynchronous::parallel_for<std::vector<
                                        std::move(std::vector<int>()),
                                        dummy_parallel_for_subtask(2),
                                        10))
    {
        return boost::asynchronous::parallel_for
                <std::vector<int>,dummy_parallel_for_subtask,boost::asynchronous
            std::move(m_data),
            dummy_parallel_for_subtask(2),
            10);
    }
    std::vector<int> m_data;
};
```

We now need to post our top-level task inside a servant:

```
post_callback(
               dummy_parallel_for_task(),
               // the lambda calls Servant, just to show that all is safe, Serva
               [this](boost::future<std::vector<int>> res){
                 try
                 {
                    // do something
                 }
                 catch(std::exception& e)
                 {
                    std::cout << "got exception: " << e.what() << std::endl;
                 }
               }// end of callback functor.
);
```

Please have a look at the complete server example [examples/example_parallel_for_tcp.cpp].

parallel_reduce

Like parallel_for, there are four versions of this algorithm, with the same lifetime behaviour. parallel_reduce applies a predicate to all elements of a range, accumulating the result.

```
template <class Iterator, class Func, class Job=boost::asynchronous::any_callab
boost::asynchronous::detail::continuation<decltype(func(std::declval<typename I
parallel_reduce(Iterator beg, Iterator end,Func func,long cutoff,const std::stri

template <class Range, class Func, class Job=boost::asynchronous::any_callable>
boost::asynchronous::detail::continuation<decltype(func(*(range.begin()), *(rang
parallel_reduce(Range const& range,Func func,long cutoff,const std::string& tas

template <class Range, class Func, class Job=boost::asynchronous::any_callable>
boost::asynchronous::detail::continuation<decltype(func(*(range.begin()), *(rang
parallel_reduce(Range&& range,Func func,long cutoff,const std::string& task_name

// version taking a continuation of a range as first argument
```

```
template <class Range, class Func, class Job=boost::asynchronous::any_callable>
boost::asynchronous::detail::continuation<decltype(func(std::declval<typename Ra
parallel_reduce(Range range,Func func,long cutoff,const std::string& task_name=
```

Don't be worried about the return type. To keep it short, what we get is a continuation of the type returned by the given predicate, for example, using the iterator version:

```
std::vector<int> data;
post_callback(
    [this]()
    {
       return boost::asynchronous::parallel_reduce(this->data.begin(),this->data
                                                   [](int const& a, int const&
                                                   {
                                                       return a + b; // return
                                                   },
                                                   1500);
    },
    ](boost::future<int> ){} // callback gets an int
);
```

We also have a distributed version [examples/example_parallel_reduce_tcp.cpp] as an example, which strictly looks like the parallel_for version.

parallel_invoke

parallel_invoke invokes a variadic list of predicates in parallel and returns a (continuation of) tuple of futures containing the result of all of them.

```
template <class Job, typename... Args>
boost::asynchronous::detail::continuation<typename decltype(boost::asynchronous
parallel_invoke(Args&&... args);
```

Of course, the futures can have exceptions if exceptions are thrown, as in the following example:

```
post_callback(
               []()
               {
                   return boost::asynchronous::parallel_invoke<boost::asynchron
                                       boost::asynchronous::to_continuation_task(
                                       boost::asynchronous::to_continuation_task(
                }// work
                // the lambda calls Servant, just to show that all is safe, Serv
                [this](boost::future<std::tuple<boost::future<void>,boost::futu
                {
                   try
                   {
                        auto t = res.get();
                        std::cout << "got result: " << (std::get<1>(t)).get() <<
                        std::cout << "got exception?: " << (std::get<0>(t)).has_
                    }
                    catch(std::exception& e)
                    {
                        std::cout << "got exception: " << e.what() << std::endl
                     }
                }// callback functor.
);
```

Notice the use of **to_continuation_task** to convert the lambdas in continuations.

As always, the callback lambda will be called when all tasks complete and the futures are non-blocking.

Please have a look at the complete example [examples/example_parallel_invoke.cpp].

parallel_find_all

This algorithms finds and copies into a returned container all elements of a range for which a predicate returns true. Like parallel_for, we have four versions of the algorithm.

```
template <class Iterator, class Func,
          class ReturnRange=std::vector<typename std::iterator_traits<Iterator>
          class Job=boost::asynchronous::any_callable>
boost::asynchronous::detail::continuation<ReturnRange,Job>
parallel_find_all(Iterator beg, Iterator end,Func func,long cutoff,const std::s

template <class Range, class Func, class ReturnRange=Range, class Job=boost::asy
boost::asynchronous::detail::continuation<ReturnRange,Job>
parallel_find_all(Range const& range,Func func,long cutoff,const std::string& ta

template <class Range, class Func, class ReturnRange=Range, class Job=boost::asy
boost::asynchronous::detail::continuation<ReturnRange,Job>
parallel_find_all(Range&& range,Func func,long cutoff,const std::string& task_na

// version taking a continuation of a range as first argument
template <class Range, class Func, class ReturnRange=typename Range::return_type
boost::asynchronous::detail::continuation<ReturnRange,Job>
parallel_find_all(Range range,Func func,long cutoff,const std::string& task_name
```

The algorithm will find elements matching the search criteria in parallel and copy all into a new container, by default of the type given as argument:

```
std::vector<int> data;
post_callback(
    [this]()
    {
       return boost::asynchronous::parallel_find_all(this->data.begin(),this->da
                                                     [](int i)
                                                     {
                                                         return (400 <= i) && (
                                                     },
                                                     1500);
    },
    ](boost::future<std::vector<int>> ){} // callback gets an int
);
```

Please have a look at the complete example [examples/example_parallel_find_all.cpp].

parallel_extremum

parallel_extremum finds an extremum (min/max) of a range given by a predicate. It is a good example of using a prallel_reduce for writing new algorithms. We have, as usual, four versions of the algorithm:.

```
template <class Iterator, class Func,class Job=boost::asynchronous::any_callable
boost::asynchronous::detail::continuation<typename std::iterator_traits<Iterator
parallel_extremum(Iterator beg, Iterator end,Func func,long cutoff,const std::s

template <class Iterator, class Func,class Job=boost::asynchronous::any_callable
decltype(boost::asynchronous::parallel_reduce(...))
```

```
parallel_extremum(Range const& range,Func func,long cutoff,const std::string& ta

template <class Iterator, class Func,class Job=boost::asynchronous::any_callable
decltype(boost::asynchronous::parallel_reduce(...))
parallel_extremum(Range&& range,Func func,long cutoff,const std::string& task_na

// version taking a continuation of a range as first argument
template <class Iterator, class Func,class Job=boost::asynchronous::any_callable
decltype(boost::asynchronous::parallel_reduce(...))
parallel_extremum(Range range,Func func,long cutoff,const std::string& task_name
```

Please have a look at the complete example [examples/example_parallel_extremum.cpp].

parallel_count

parallel_count counts the elements of a range satisfying a predicate. As usual, we have four versions of the algorithm.

```
template <class Iterator, class Func,class Job=boost::asynchronous::any_callable
boost::asynchronous::detail::continuation<long,Job>
parallel_count(Iterator beg, Iterator end,Func func,long cutoff,const std::strin

template <class Iterator, class Func,class Job=boost::asynchronous::any_callable
boost::asynchronous::detail::continuation<long,Job>
parallel_count(Range const& range,Func func,long cutoff,const std::string& task_

template <class Iterator, class Func,class Job=boost::asynchronous::any_callable
boost::asynchronous::detail::continuation<long,Job>
parallel_count(Range&& range,Func func,long cutoff,const std::string& task_name=

// version taking a continuation of a range as first argument
template <class Iterator, class Func,class Job=boost::asynchronous::any_callable
boost::asynchronous::detail::continuation<long,Job>
parallel_count(Range range,Func func,long cutoff,const std::string& task_name=""
```

Please have a look at the complete example [examples/example_parallel_count.cpp].

Chapter 4. Tips.

Which protections you get, which ones you don't.

Asynchronous is doing much to protect developers from some ugly beasts around:

- (visible) threads
- races
- deadlocks
- crashes at the end of an object lifetime

It also helps parallelizing and improve performance by not blocking. It also helps find out where bottlenecks and hidden possible performance gains are.

There are, however, things for which it cannot help:

- cycles in design
- C++ legal ways to work around the protections if one really wants.
- blocking on a future if one really wants.
- using "this" captured in a task lambda.
- writing a not clean task with pointers or references to data used in a servant.

No cycle, never!

This is one of the first things one learns in a design class. Cycles are evil. Everybody knows it. And yet, designs are often made without care in a too agile process, dependency within an application is not thought out carefully enough and cycles happen. What we do learn in these classes is that cycles make our code monolithic and not reusable. What we however do not learn is how bad, bad, bad this is in face of threads. It becomes impossible to follow the flow of information, resource usage, degradation of performance. But the worst of all, it becomes almost impossible to prevent deadlocks and resource leakage.

Using Asynchronous will help write clean layered architectures. But it will not replace carefully crafted designs, thinking before writing code and the experience which make a good designer. Asynchronous will not be able to prevent code having cycles in a design.

Fortunately, there is an easy solution: back to the basics, well-thought designs before coding, writing diagrams, using a real development process (hint: an agile "process" is not all this in the author's mind).

No "this" within a task.

A very easy way to see if you are paving the way to a race even using Asynchronous is to have a look at the captured variables of a lambda posted to a threadpool. If you find "this", it's probably bad, unless you really know that the single-thread code will do nothing. Apart from a simple application, this will not be true. By extension, pointers, references, or even shared smart pointers pointing to data living in a single-thread world is usually bad.

Experience shows that there are only two safe way to pass data to a posted task: copy for basic types or types having a trivial destructor and move for everything else. Keep to this rule and you will be safe.

On the other hand, "this" is okay in the capture list of a callback task as Asynchronous will only call it if the servant is still alive.

Part III. Reference

Table of Contents

Chapter 5. Queues

Asynchronous provides a range of queues with different trade-offs. Use `lockfree_queue` as default for a quickstart with Asynchronous.

threadsafe_list

This queue is mostly the one presented in Anthony Williams' book, "C++ Concurrency In Action". It is made of a single linked list of nodes, with a mutex at each end of the queue to minimize contention. It is reasonably fast and of simple usage. It can be used in all configurations of pools.

Its constructor does not require any parameter forwarded from the scheduler.

Stealing: from the same queue end as pop. Will be implemented better (from the other end to reduce contention) in a future version.

Caution: crashes were noticed with gcc 4.8 while 4.7 and clang 3.3 seemed ok though the compiler might be the reason. For this reason, lockfree_queue is now the default queue.

Declaration:

```
template<class JOB = boost::asynchronous::any_callable>
class threadsafe_list;
```

lockfree_queue

This queue is a light wrapper around a `boost::lockfree::queue`, which gives lockfree behavior at the cost of an extra dynamic memory allocation. Please use this container as default when starting with Asynchronous.

The container is faster than a `threadsafe_list`, provided one manages to set the queue size to an optimum value. A too small size will cause expensive memory allocations, a too big size will significantly degrade performance.

Its constructor takes optionally a default size forwarded from the scheduler.

Stealing: from the same queue end as pop. Stealing from the other end is not supported by `boost::lockfree::queue`. It can be used in all configurations of pools.

Declaration:

```
template<class JOB = boost::asynchronous::any_callable>
class lockfree_queue;
```

lockfree_spsc_queue

This queue is a light wrapper around a `boost::lockfree::spsc_queue`, which gives lockfree behavior at the cost of an extra dynamic memory allocation.

Its constructor requires a default size forwarded from the scheduler.

Stealing: None. Stealing is not supported by `boost::lockfree::spsc_queue`. It can only be used Single-Producer / Single-Consumer, which reduces its typical usage to a queue of a `multiqueue_threadpool_scheduler` as consumer, with a `single_thread_scheduler` as producer.

Declaration:

```
template<class JOB = boost::asynchronous::any_callable>
class lockfree_spsc_queue;
```

lockfree_stack

This queue is a light wrapper around a `boost::lockfree::stack`, which gives lockfree behavior at the cost of an extra dynamic memory allocation. This container creates a task inversion as the last posted tasks will be executed first.

Its constructor requires a default size forwarded from the scheduler.

Stealing: from the same queue end as pop. Stealing from the other end is not supported by `boost::lockfree::stack`. It can be used in all configurations of pools.

Declaration:

```
template<class JOB = boost::asynchronous::any_callable>
class lockfree_stack;
```

Chapter 6. Schedulers

There is no perfect scheduler. In any case it's a question of trade-off. Here are the schedulers offered by Asynchronous.

single_thread_scheduler

The scheduler of choice for all servants which are not thread-safe. Serializes all calls to a single queue and executes them in order. Using `any_queue_container` as queue will however allow it to support task priority.

This scheduler does not steal from other queues or pools, only stealing_ threadpools do this, and does not get stolen from to avoid races.

Declaration:

```
template<class Queue>
class single_thread_scheduler;
```

Creation:

```
boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::single_thread_scheduler<
                          boost::asynchronous::threadsafe_list<> >);

boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::single_thread_scheduler<
                          boost::asynchronous::lockfree_queue<> >(10)); // size
```

Or, using logging:

```
typedef boost::asynchronous::any_loggable<boost::chrono::high_resolution_clock>

boost::asynchronous::any_shared_scheduler_proxy<servant_job> scheduler =
    boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::single_thread_scheduler<
                          boost::asynchronous::threadsafe_list<servant_job> >);

boost::asynchronous::any_shared_scheduler_proxy<servant_job> scheduler =
    boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::single_thread_scheduler<
                          boost::asynchronous::lockfree_queue<servant_job> >(10
```

Table 6.1. #include <boost/asynchronous/scheduler/single_thread_scheduler.hpp>

Characteristics	
Number of threads	1
Can be stolen from?	No
Can steal from other threads in this pool?	N/A (only 1 thread)
Can steal from other threads in other pools?	No

threadpool_scheduler

The simplest and easiest threadpool using a single queue, though multiqueue behavior could be done using `any_queue_container`. The advantage is that it allows the pool to be given 0 thread and only be stolen from. The cost is a slight performance loss due to higher contention on the single queue.

This pool does not steal from other pool's queues.

Use this pool as default for a quickstart with Asynchronous.

Declaration:

```
template<class Queue>
class threadpool_scheduler;
```

Creation:

```
boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::threadpool_scheduler<
                          boost::asynchronous::threadsafe_list<> >(4)); // 4 th

boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::threadpool_scheduler<
                          boost::asynchronous::lockfree_queue<> >(4,10)); // si
```

Or, using logging:

```
typedef boost::asynchronous::any_loggable<boost::chrono::high_resolution_clock>

boost::asynchronous::any_shared_scheduler_proxy<servant_job> scheduler =
    boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::threadpool_scheduler<
                          boost::asynchronous::threadsafe_list<servant_job> >(4

boost::asynchronous::any_shared_scheduler_proxy<servant_job> scheduler =
    boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::threadpool_scheduler<
                          boost::asynchronous::lockfree_queue<servant_job> >(4,
```

Table 6.2. #include <boost/asynchronous/scheduler/threadpool_scheduler.hpp>

Characteristics	
Number of threads	**0**-n
Can be stolen from?	Yes
Can steal from other threads in this pool?	N/A (only 1 queue)
Can steal from other threads in other pools?	No

multiqueue_threadpool_scheduler

This is a `threadpool_scheduler` with multiple queues to reduce contention. On the other hand, this pool requires at least one thread.

This pool does not steal from other pool's queues though pool threads do steal from each other's queues.

Declaration:

```
template<class Queue,class FindPosition=boost::asynchronous::default_find_posit
class multiqueue_threadpool_scheduler;
```

Creation:

```
boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::multiqueue_threadpool_scheduler<
                          boost::asynchronous::threadsafe_list<> >(4)); // 4 th

boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::multiqueue_threadpool_scheduler<
                          boost::asynchronous::lockfree_queue<> >(4,10)); // si
```

Or, using logging:

```
typedef boost::asynchronous::any_loggable<boost::chrono::high_resolution_clock>

boost::asynchronous::any_shared_scheduler_proxy<servant_job> scheduler =
    boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::multiqueue_threadpool_scheduler<
                          boost::asynchronous::threadsafe_list<servant_job> >(4

boost::asynchronous::any_shared_scheduler_proxy<servant_job> scheduler =
    boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::multiqueue_threadpool_scheduler<
                          boost::asynchronous::lockfree_queue<servant_job> >(4,
```

Table 6.3. #include <boost/asynchronous/scheduler/multiqueue_threadpool_scheduler.hpp>

Characteristics	
Number of threads	**1**-n
Can be stolen from?	Yes
Can steal from other threads in this pool?	Yes
Can steal from other threads in other pools?	No

stealing_threadpool_scheduler

This is a `threadpool_scheduler` with the added capability to steal from other pool's queues within a `composite_threadpool_scheduler`. Not used within a `composite_threadpool_scheduler`, it is a standard `threadpool_scheduler`.

Declaration:

```
template<class Queue,bool /* InternalOnly */ = true >
class stealing_threadpool_scheduler;
```

Creation if used within a `composite_threadpool_scheduler`:

```
boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::stealing_threadpool_scheduler<
```

```
                    boost::asynchronous::threadsafe_list<> >(4)); // 4 th
```

However, if used stand-alone, which has little interest outside of unit tests, we need to add a template parameter to inform it:

```
boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::stealing_threadpool_scheduler<
                          boost::asynchronous::threadsafe_list<>,true >(4)); //
```

Table 6.4. #include <boost/asynchronous/scheduler/stealing_threadpool_scheduler.hpp>

Characteristics	
Number of threads	**0**-n
Can be stolen from?	Yes
Can steal from other threads in this pool?	N/A (only 1 queue)
Can steal from other threads in other pools?	Yes

stealing_multiqueue_threadpool_scheduler

This is a `multiqueue_threadpool_scheduler` with the added capability to steal from other pool's queues within a `composite_threadpool_scheduler` (of course, threads within this pool do steal from each other queues, with higher priority). Not used within a `composite_threadpool_scheduler`, it is a standard `multiqueue_threadpool_scheduler`.

Declaration:

```
template<class Queue,class FindPosition=boost::asynchronous::default_find_posit
class stealing_multiqueue_threadpool_scheduler;
```

Creation if used within a `composite_threadpool_scheduler`:

```
boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::stealing_multiqueue_threadpool_sch
                          boost::asynchronous::threadsafe_list<> >(4)); // 4 th
```

However, if used stand-alone, which has little interest outside of unit tests, we need to add a template parameter to inform it:

```
boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::stealing_multiqueue_threadpool_sch
                          boost::asynchronous::threadsafe_list<>,boost::asynchr
```

Table 6.5. #include <boost/asynchronous/stealing_multiqueue_threadpool_scheduler.hpp>

Characteristics	
Number of threads	**1**-n
Can be stolen from?	Yes
Can steal from other threads in this pool?	Yes

Characteristics	
Can steal from other threads in other pools?	Yes

composite_threadpool_scheduler

This pool owns no thread by itself. Its job is to contain other pools, accessible by the priority given by posting, and share all queues of its subpools among them. Only the stealing_* pools and `asio_scheduler` will make use of this and steal from other pools though.

For creation we need to create other pool of stealing or not stealing, stolen from or not, schedulers. stealing_xxx pools will try to steal jobs from other pool of the same composite, but only if these schedulers support this. Other threadpools will not steal but get stolen from. `single_thread_scheduler` will not steal or get stolen from.

```
// create a composite threadpool made of:
// a multiqueue_threadpool_scheduler, 0 thread
// This scheduler does not steal from other schedulers, but will lend its queues
auto tp = boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::threadpool_scheduler<boost::asynch

// a stealing_multiqueue_threadpool_scheduler, 3 threads, each with a threadsafe
// this scheduler will steal from other schedulers if it can. In this case it w
auto tp2 = boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::stealing_multiqueue_threadpool_sche

// composite pool made of the previous 2
auto tp_worker = boost::asynchronous::create_shared_scheduler_proxy(new boost::a
```

Declaration:

```
template<class Job = boost::asynchronous::any_callable,
         class FindPosition=boost::asynchronous::default_find_position< >,
         class Clock = boost::chrono::high_resolution_clock  >
class composite_threadpool_scheduler;
```

Table 6.6. #include <boost/asynchronous/scheduler/composite_threadpool_scheduler.hpp>

Characteristics	
Number of threads	0
Can be stolen from?	Yes
Can steal from other threads in this pool?	N/A
Can steal from other threads in other pools?	No

asio_scheduler

This pool brings the infrastructure and access to io_service for an integrated usage of Boost.Asio. Furthermore, if used withing a `composite_threadpool_scheduler`, it will steal jobs from other pool's queues.

Declaration:

```
template<class FindPosition=boost::asynchronous::default_find_position< boost::a
```

```
class asio_scheduler;
```

Creation:

```
boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::asio_scheduler<>(4)); // 4 threads
```

Table 6.7. #include <boost/asynchronous/extensions/asio/asio_scheduler.hpp>

Characteristics	
Number of threads	**1**-n
Can be stolen from?	No*
Can steal from other threads in this pool?	Yes
Can steal from other threads in other pools?	Yes

Chapter 7. Compiler

C++ 11

Asynchronous is C++11-only. Please check that your compiler has C++11 enabled (-std=c++0x or -std=c++11 in different versions of gcc)

Supported compilers

At the moment, Asynchronous has only be tested with gcc versions from 4.7 to 4.8 and clang 3.3-3.4.