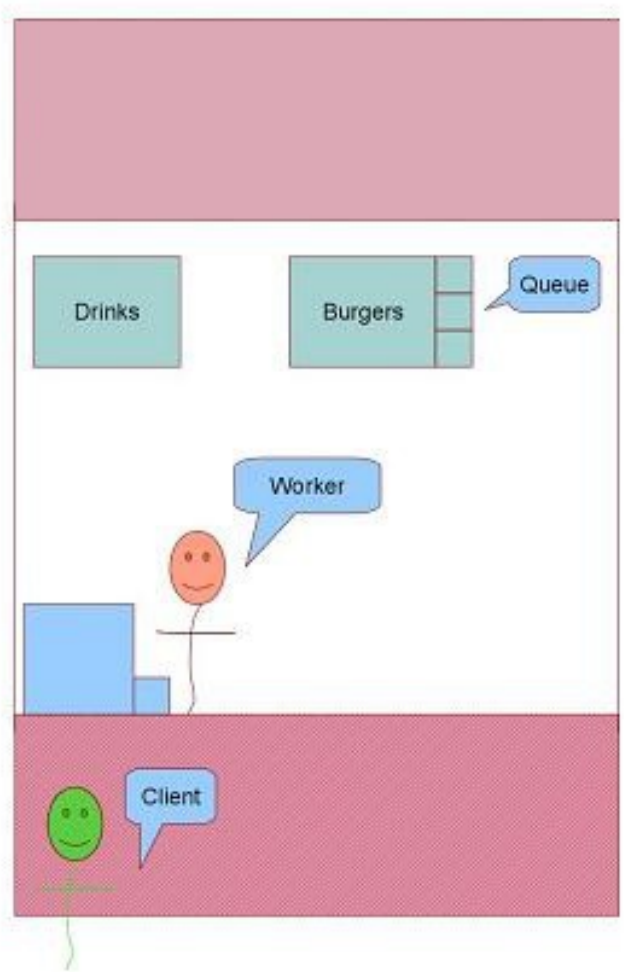
Asynchronous Programming With Boost Meta State Machine And The Asynchronous Library

Contents

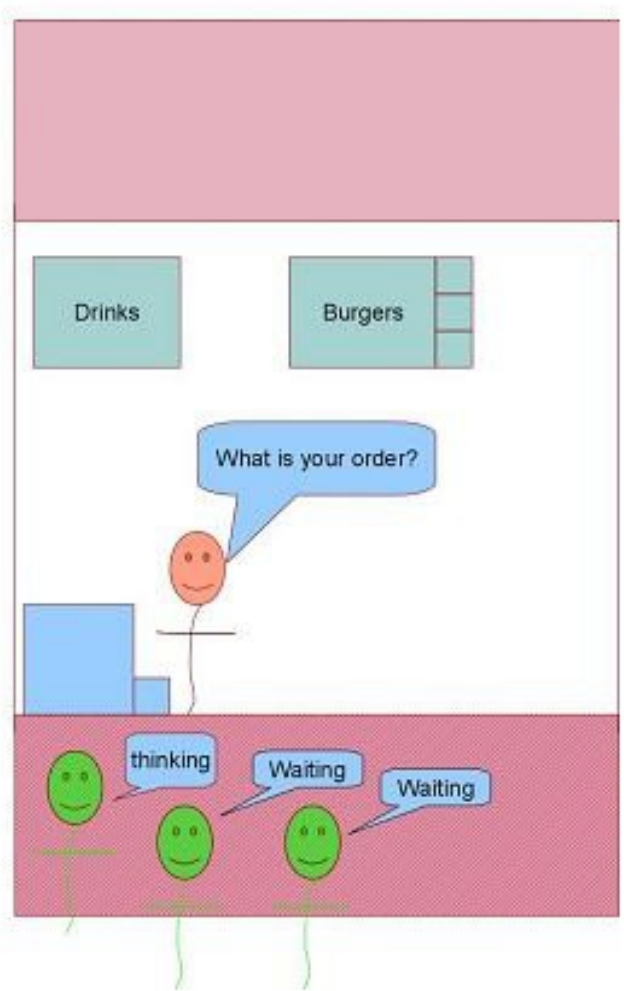
- Introducing Story
- Why State Machines?
- Our Pattern Of The Day
- Boost Meta State Machine
- Asynchronous programming
- CD Player example

Introducing Story



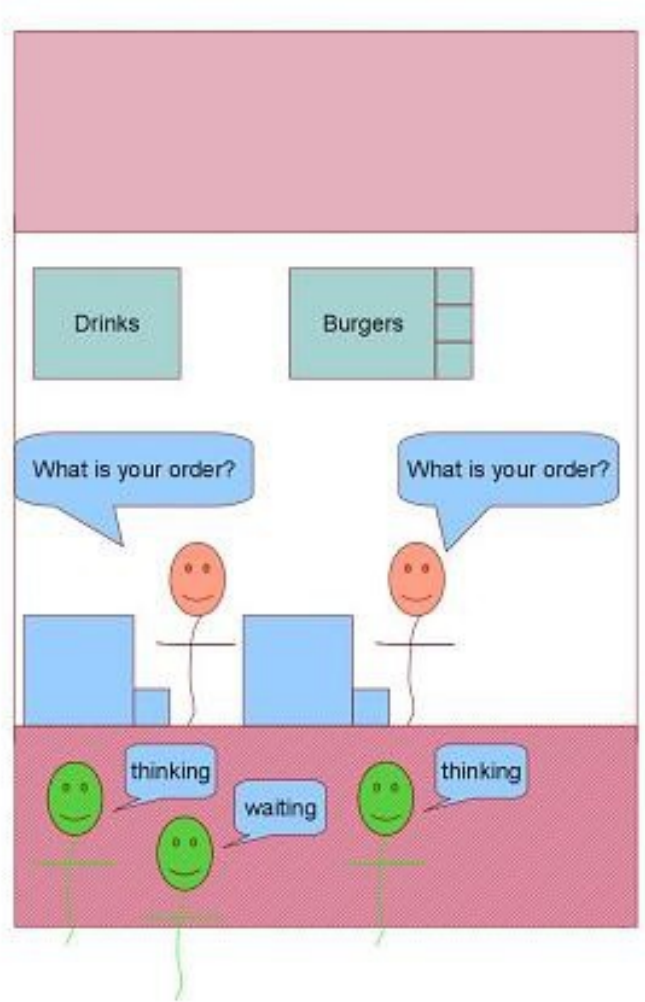
- A restaurant as a single employee
- Burgers are put in a queue
- A first customer comes

Introducing Story



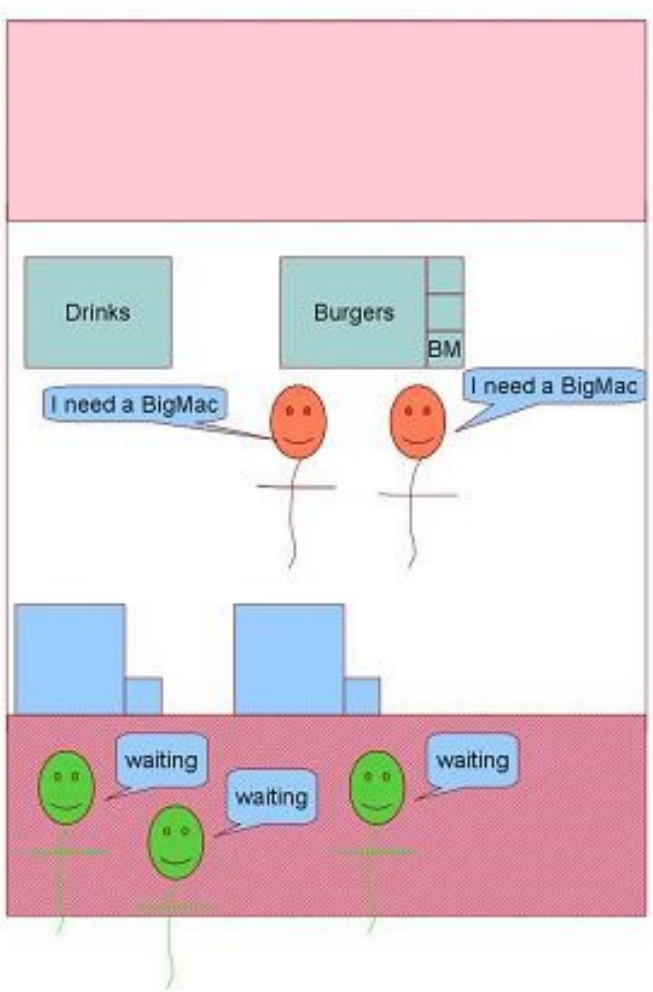
- More customers keep coming
- What to do?

Introducing Story



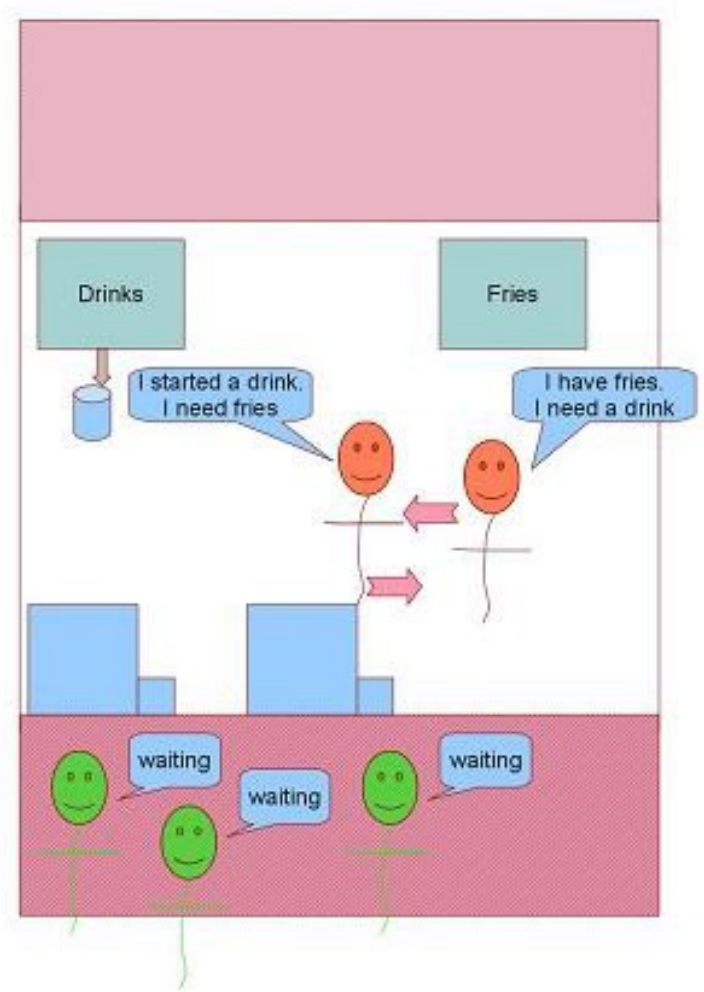
- A second employee is hired
- The owner hopes to reduce wait times

Introducing Story



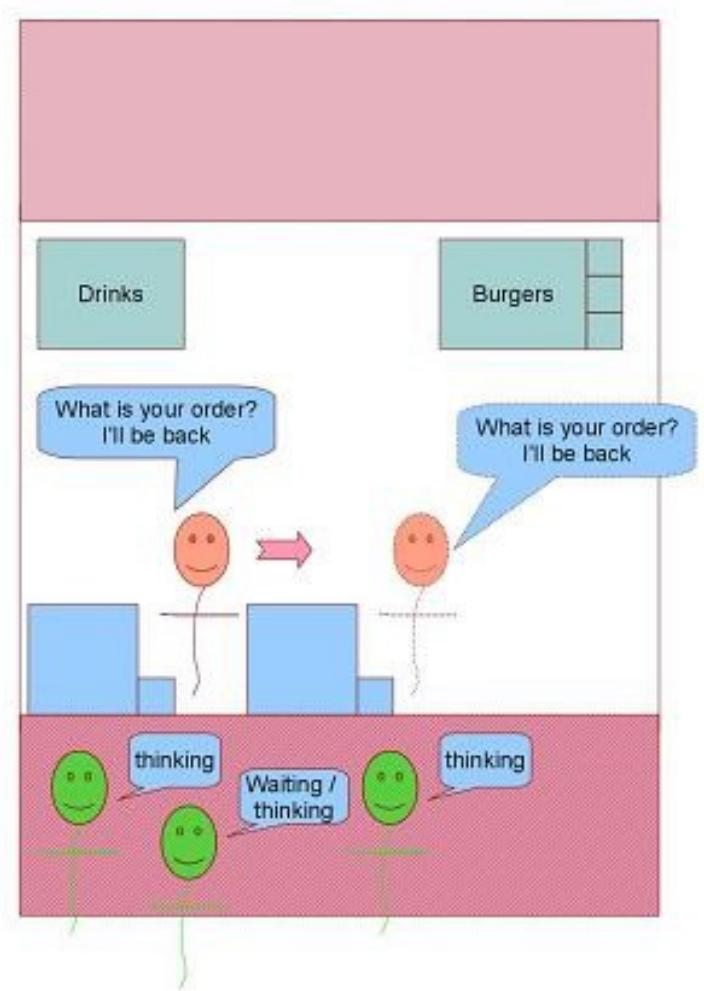
- This brings chaos
- Sometimes employees fight to get their customer a burger first

Introducing Story



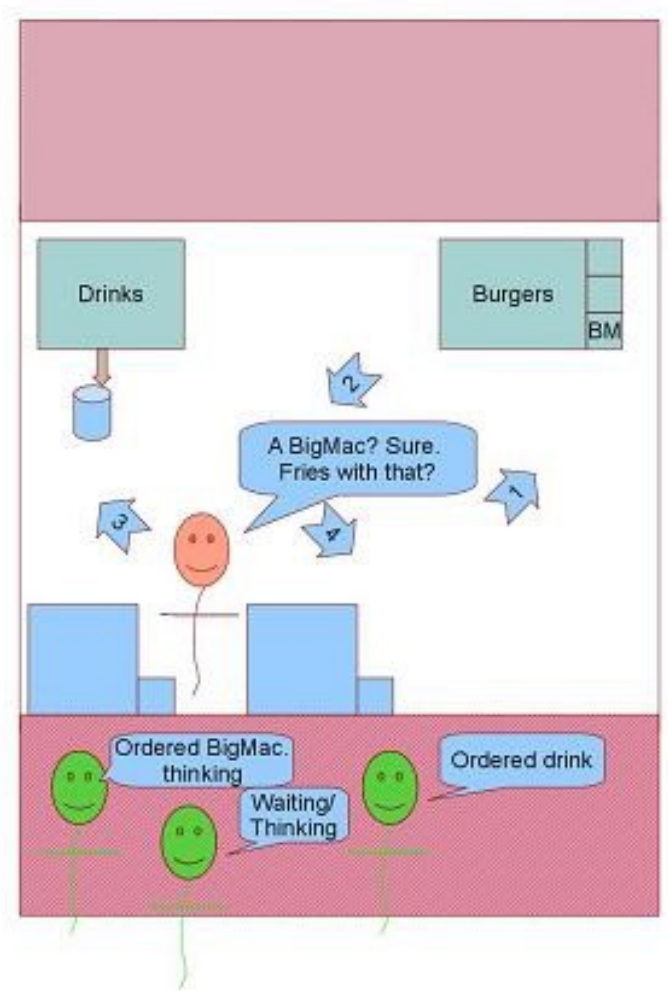
- Sometimes they get in each other's way
- The line of waiting customers grows and grows
- It just does not work
  - Costs explode
  - Wait times too
  - Customers flee the restaurant
  - The restaurant gets bankrupt
  - Can we avoid this?

Introducing Story



- We keep a single worker
- The Worker runs really fast from cash desk to cash desk, to burgers, drinks, etc.
- The Worker never ever waits
- Instead he remembers in which state of the order each customer is
- The Worker only reacts to events: burger ready, customer picked drink, etc.

Introducing Story



Our Worker in action:

Why State Machines?

State machines help us:

- Design
- Document
- Debug
- Think asynchronously

Our Pattern Of The Day

- A Manager implemented as a state machine runs in its own thread
- The Manager is non-blocking.
- Target hardware is controlled asynchronously and lives in other threads or machine.

To achieve this we need:

- A state machine library
- Infrastructure to manage asynchronous behavior.

Asynchronous Programming

std::async / boost::async

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

Simple, but...

- ► Blocking is bad for state machines (no run to completion).
- ► Blocking prevents diagnostics.
- ► Blocking makes your program less responsive.
- ► Blocking reduces opportunities for concurrency.

Waiting is ok, blocking no.

Bonus question, in which thread is lambda executed?

Asynchronous Programming

std::async / boost::async

We have for alternatives:

- *Block while waiting*
- *Poll*
- *Carry a bag of futures then do one of above*

Asynchronous Programming

std::async / boost::async

Do you spot a problem?

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

► 2nd line does not run until f() completes

Asynchronous Programming

- Better with N3558 / N3650?

```
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();
```

Asynchronous Programming

Boost.Asio. Example:

```
// won't block
boost::asio::async_write(socket_, request_,
                boost::bind(&client::handle_write_request, this,
                                boost::asio::placeholders::error));

// callback, possibly much later
void handle_write_request(const boost::system::error_code& /*error*/)
{…}
```

Asynchronous Programming

Boost.Asio. Disadvantages:

- Object lifetime
- Managing asynchrony
- Limited capabilities besides network communication

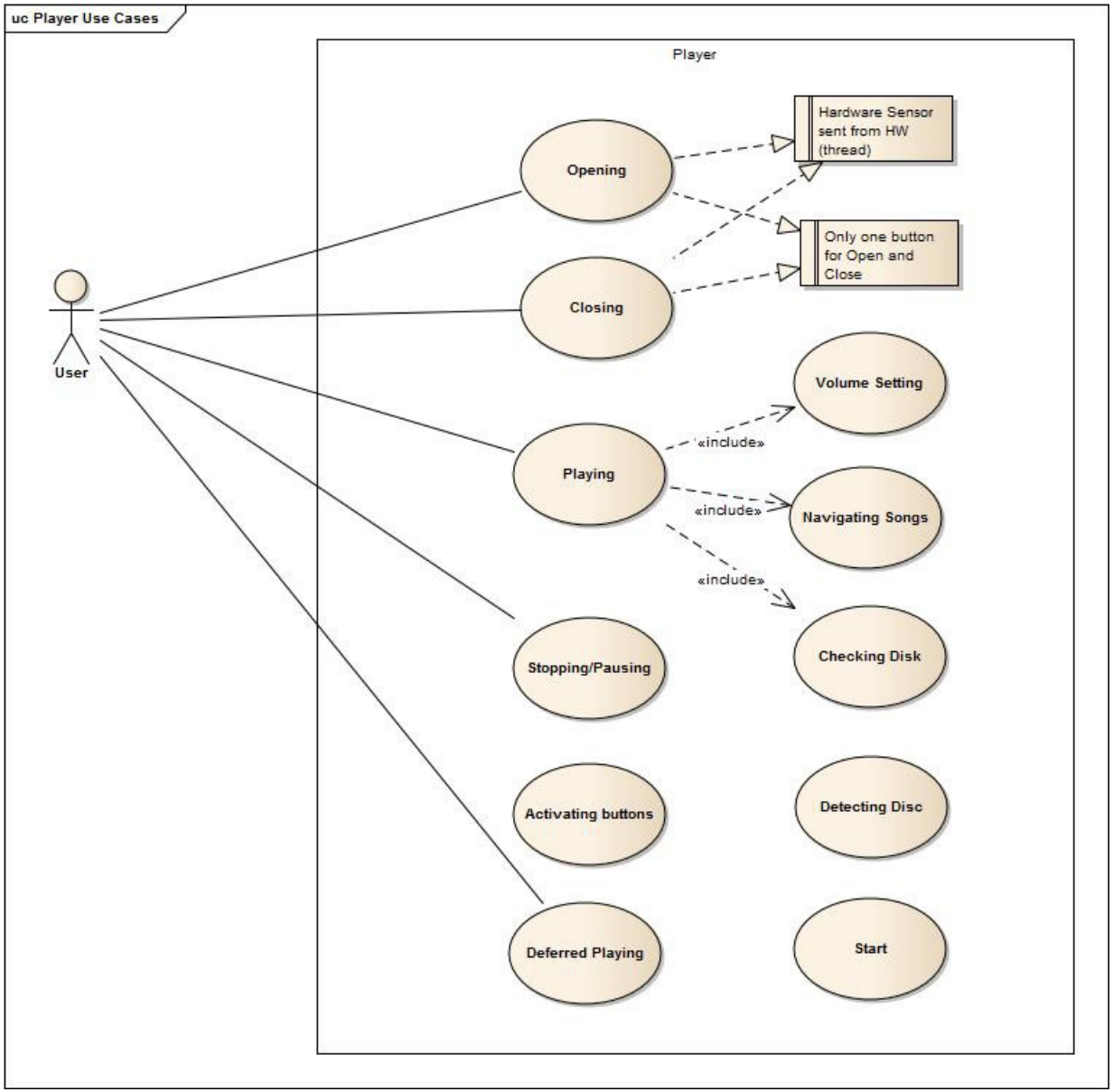
Asynchronous Programming

Maybe something like this would be better?

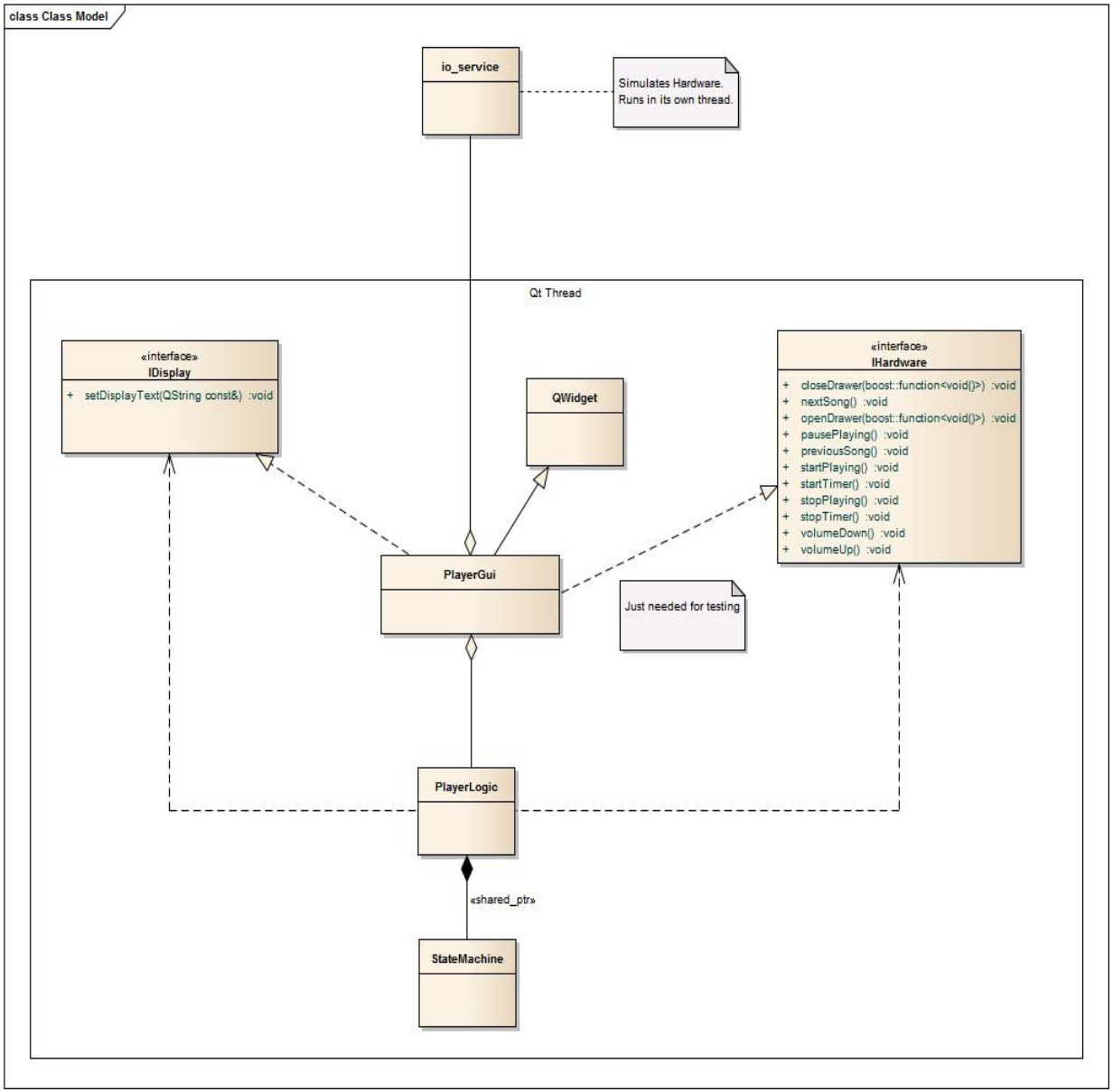
```
post_callback(
   [](){return 42;}                        // long-lasting task
   [](boost::future<int> res){...} // callback
);
```

But thread-safe, forwarding exceptions, non-blocking and taking object lifetime into consideration.

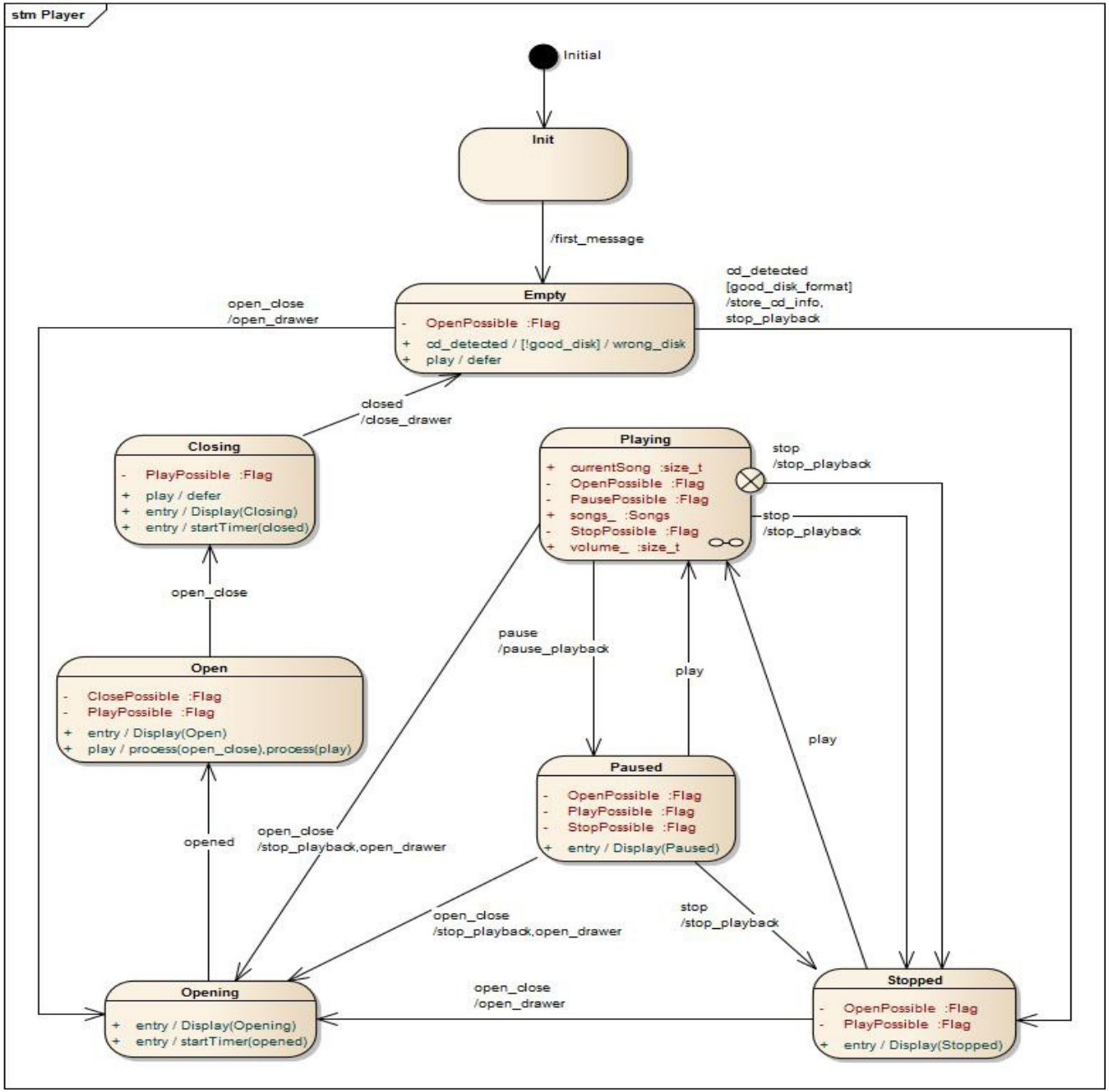
Example: CD Player



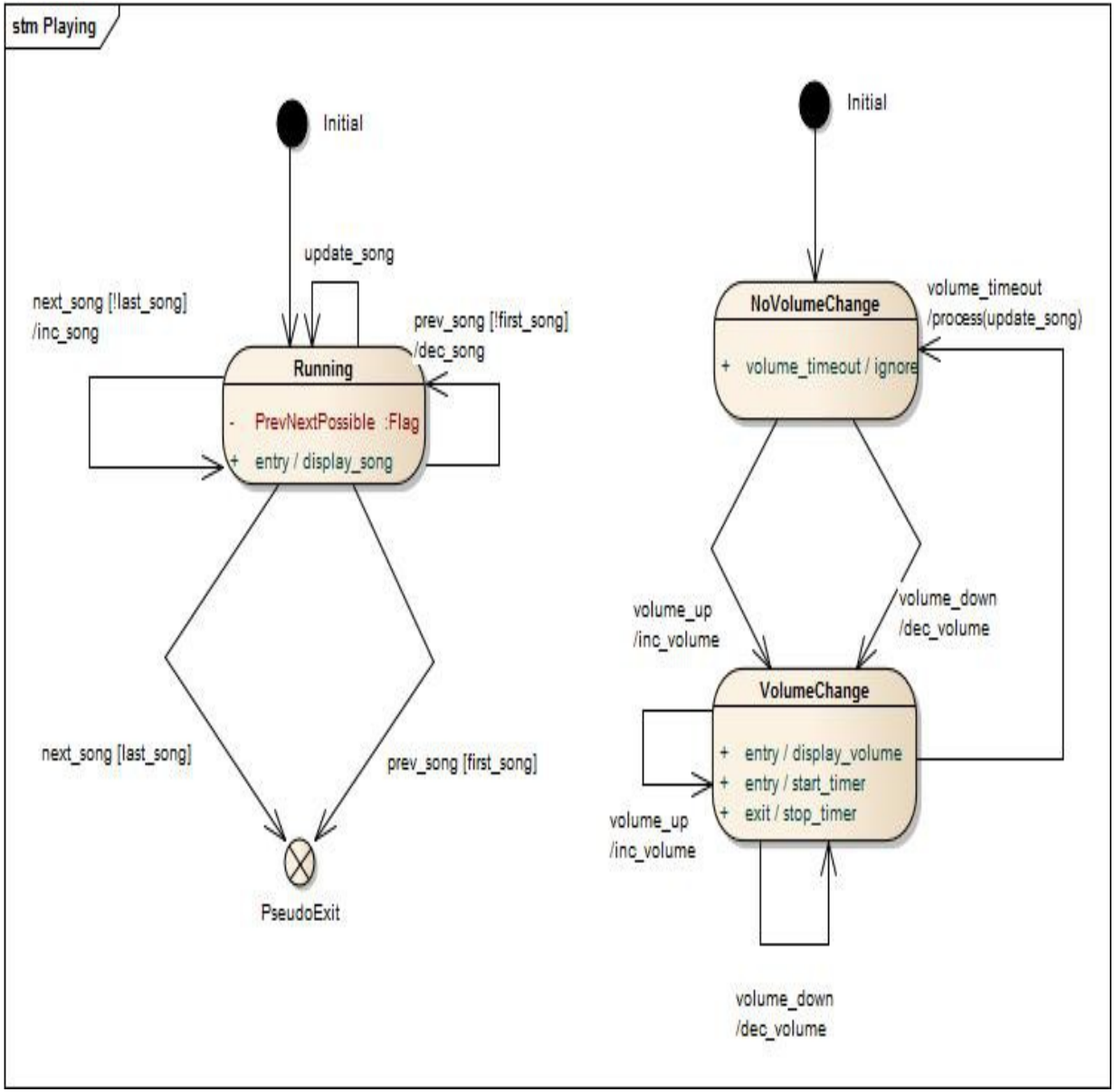
Example: CD Player



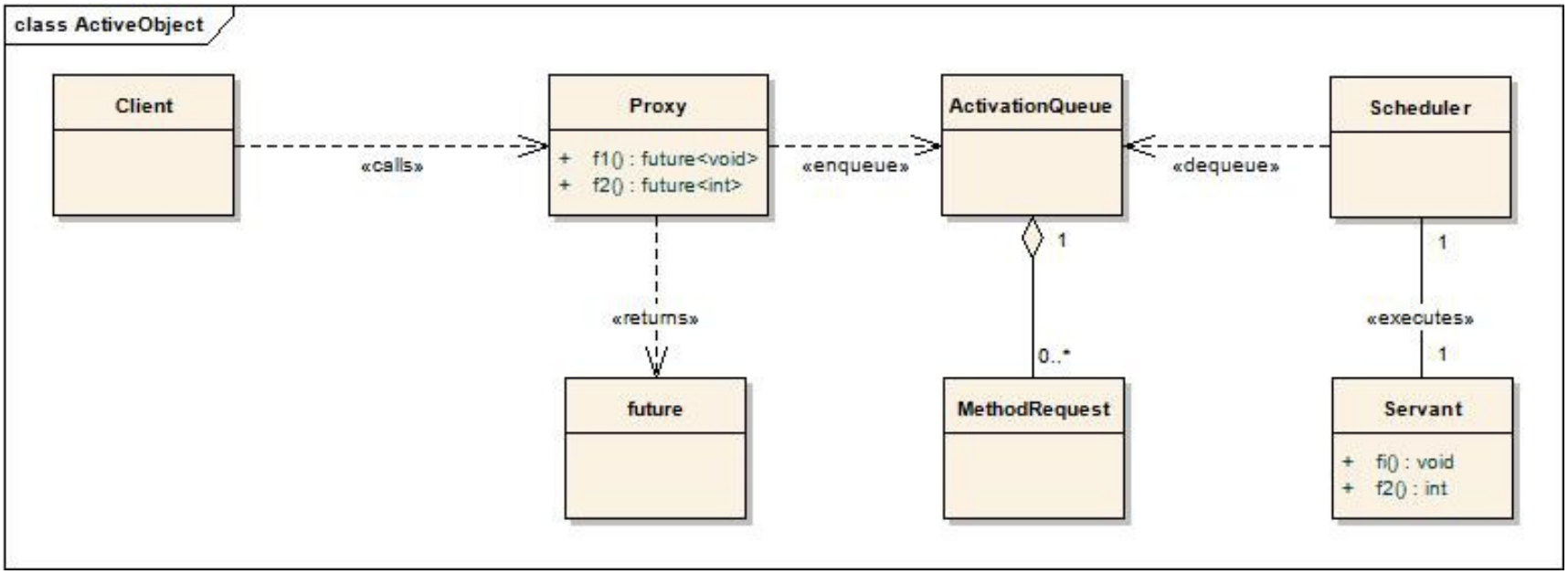
Example: CD Player



Example: CD Player

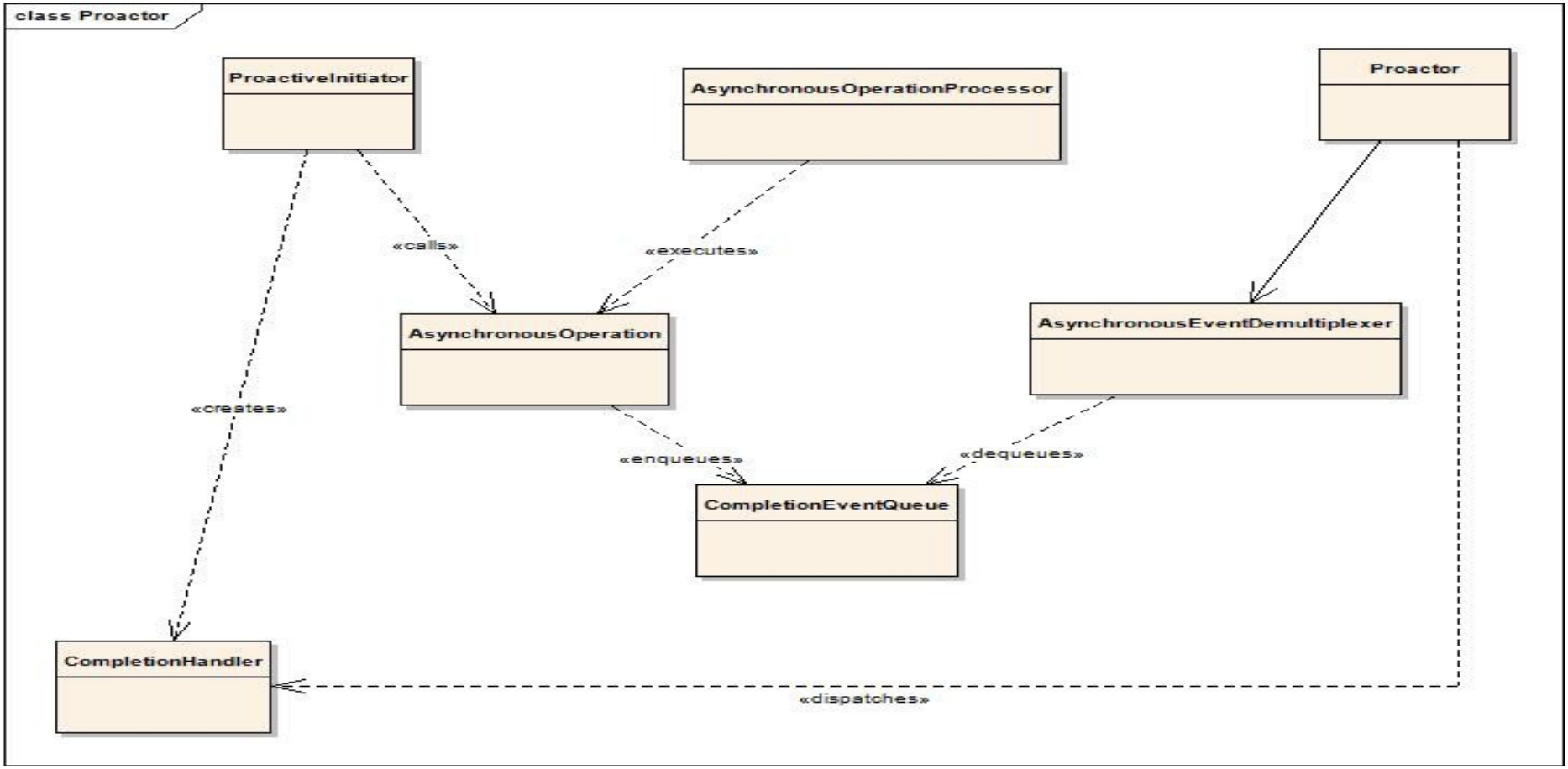


Patterns: Active Object



- Makes thread-safe a non-thread-safe object by serializing calls
- Outside world only sees a Proxy
- A Scheduler takes jobs from an ActivationQueue
- Jobs are executed within the Scheduler context
- Active Objects are expensive (a thread per object)
- Active Objects are not helping parallelize

Patterns: Proactor



- An Initiator calls an AsynchronousOperation
- A Job is enqueued into a CompletionEventQueue
- An AsynchronousOperationProcessor executes the job
- A Proactor dispatches a CompletionHandler

Patterns: Thread-Safe Interface

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

- Public members lock
- Private members do not
- Safe?

(Boost) Asynchronous

- Will be offered for review (Review Manager?)
- C++11 (will make for interesting discussions...)
- Compiles with g++ >= 4.7 && clang 3.4
- Header only. Will however require linking to Boost.Thread, Chrono, DateTime, possibly Serialization.

Asynchronous: Principles

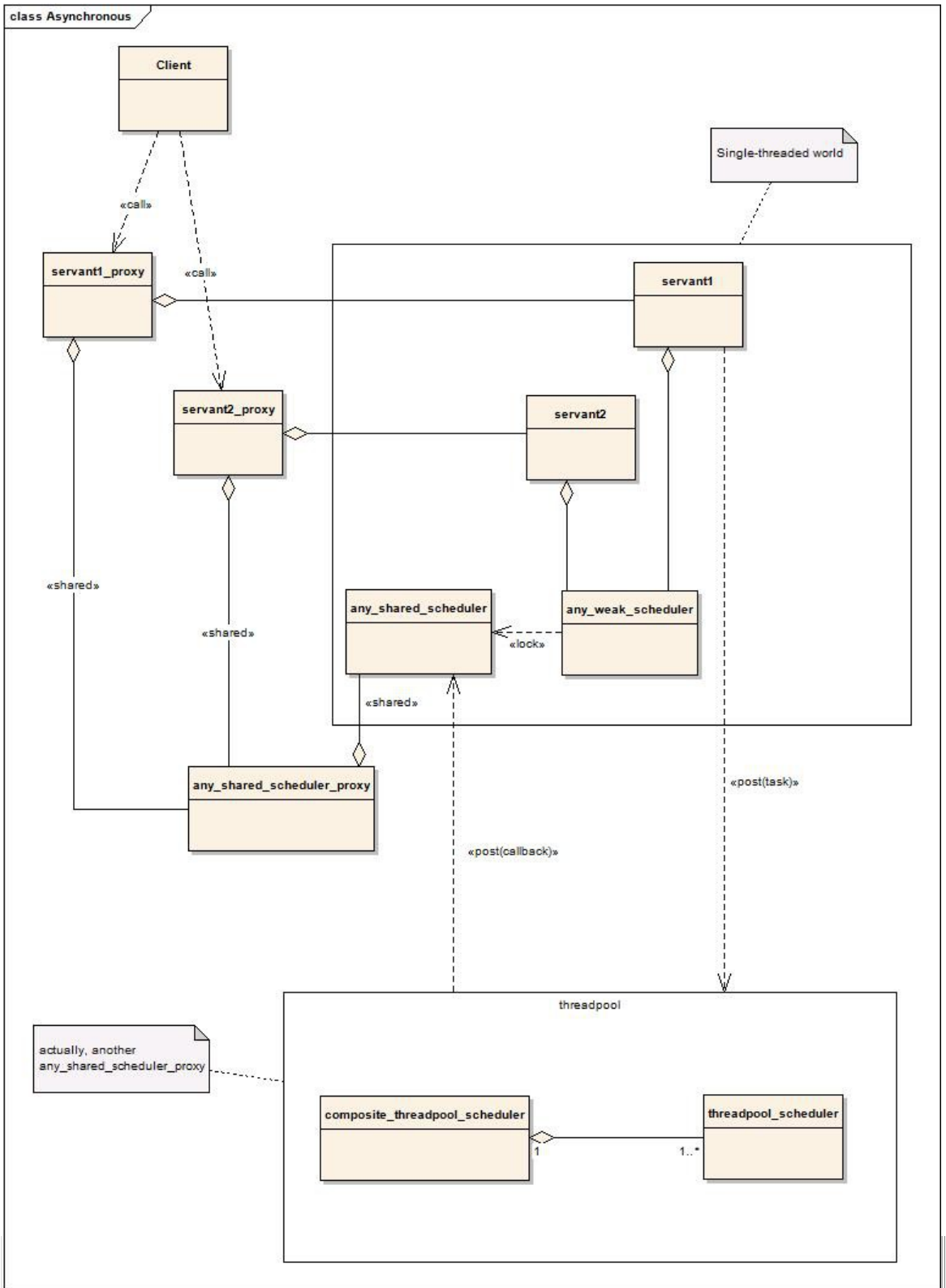
- Makes you think Tasks, not Threads
- Helps prevents races, deadlocks, crashes
- Executes Tasks asynchronously
- Result of Tasks come as callback
- No Blocking!!! Waiting yes, blocking, no.

Asynchronous: Definitions

- Scheduler: object having 0..n threads, executing jobs or callbacks. Stops threads when destroyed.
- Weak Scheduler: a weak_ptr to a scheduler.
- Servant: object living in a (single-threaded) scheduler, starting tasks and handling callbacks.
- Queue: holds jobs for a scheduler to execute
- Servant Proxy: a thread-safe object looking like a Servant and serializing calls to it
- Scheduler Shared Proxy: object holding a scheduler and interfacing with it. The last instance joins the threads of the scheduler.
- Stealing: between threads or schedulers.
- Posting: enqueueing a job into a Scheduler's queue.

Asynchronous: Lifetime

- Create any number of Servants within a single-thread context
- Servants are visible to outside world through proxies
- The last one needing it stops the thread
- The last proxy joins the thread
- Threadpools allow posting of long-lasting jobs and parallelizing
- And most of all, never ever block!!!


class Asynchronous
Client
Single-threaded world
«call»
«call»
servant1_proxy
servant1
servant2_proxy
servant2
«shared»
«shared»
any_shared_scheduler
any_weak_scheduler
«lock»
«shared»
any_shared_scheduler_proxy
«post(task)»
«post(callback)»
threadpool
actually, another
any_shared_scheduler_proxy
composite_threadpool_scheduler
threadpool_scheduler
1
1..*

Features

- Lifetime control
- Proxies
- Interrupting
- Diagnostics
- Continuations
- Distributing
- Parallel Algorithms
- Interaction with Qt / Boost.Asio
- Queues, Threadpools, Task Priority

Hello, asynchronous world

```
// a threadpool with 3 threads
auto scheduler =
   create_shared_scheduler_proxy( new threadpool_scheduler<
                                                  lockfree_queue<> >(3));

// post a simple task and wait for result
boost::shared_future<int> fui =
   boost::asynchronous::post_future(scheduler,
                                          [](){return 42;});

int res = fui.get();
```

ServantProxy

```
struct Servant
{
  Servant(int data): m_data(data){}
  int doIt()const { return 5; }
  void foobar(int i, char c)const { }
private:
  int m_data;
};
```

- Our servant is a plain, boring class
- We want it to offer two methods „outside“
- The constructor requires data

ServantProxy

```
class ServantProxy :
public servant_proxy<ServantProxy,Servant> {
public:
// forwarding constructor. Scheduler to servant_proxy,
// followed by arguments to Servant.

template <class Scheduler>
ServantProxy(Scheduler s, int data):
servant_proxy<ServantProxy,Servant>(s, data) {}

// the following members must be available "outside"
BOOST_ASYNC_POST_MEMBER(foobar)
// for doIt, we'd like a future
BOOST_ASYNC_FUTURE_MEMBER(doIt)
};
```

ServantProxy

```
{
auto scheduler = create_shared_scheduler_proxy(
                 new single_thread_scheduler<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 and get a future because we're interested in the result.
         boost::shared_future<int> fu = proxy.doIt();

      }// here, Servant's destructor is posted
}// scheduler is gone, its thread has been joined
```

Threadpool

```
struct Servant : trackable_servant<>
{
   Servant(any_weak_scheduler<> scheduler)
      : trackable_servant<>
            (scheduler,create_shared_scheduler_proxy(new
                  threadpool_scheduler<lockfree_queue<> >(3)))
//...
};
```

- We now equip our servant with a threadpool
- The threadpool has 3 threads
- The servant knows his own (weak) scheduler for callbacks
- We can now make use of post_callback

post_callback

```
post_callback(
        // possibly long work, executes in threadpool, if servant alive
        [](){return 42;},
        // callback functor. Executes in Servant's context
        // Servant is alive if this is called
        [this](boost::future<int> res){/*...*/}
);
```

- Work task is posted to threadpool
- Work task executed if Servant is still alive
- Callback executed if Servant is still alive
- Using this in callback is safe
- Return value or exception from task in future
- Future is non-blocking

Interrupting

Why to interrupt?

- Exploding algorithms
- System is drowning
- No need of result any more
- Requires support from Task itself

Interrupting

```
any_interruptible interruptible =
    interruptible_post_callback(
            // task
            [](){
                    // boost::this_thread::sleep is an interruption point
                    boost::this_thread::sleep
                        (boost::posix_time::milliseconds(1000));},
            // callback functor.
            // most likely will not be called
            [this](boost::future<void> res){/*...*/}
);
interruptible.interrupt();
```

- Work task is posted to threadpool
- Immediately after we try to interrupt
- Sleep is a documented interruption point for boost::thread and most likely will be interrupted
- Callback will not be called

Logging

Why do we need this?

- Find bottlenecks
- Find out inefficiencies in tasks
- Find concurrency opportunities
- Find out which tasks can be started earlier

Logging + state machines in an Active Object are your friends.
You will know:

- Where is your bottleneck
- Which tasks are worth parallelizing
- How long you spent in a state

Logging

```
// we need a job type
typedef any_loggable<chrono::high_resolution_clock> servant_job;

// we need our servant to make use of it
struct Servant : trackable_servant<servant_job,servant_job>

post_callback(
            [](){return 42;},
            [this](boost::future<int> res){/*...*/},
            // job / callback name
            "int_async_work"
);
// we also have a new macro
BOOST_ASYNC_FUTURE_MEMBER_LOG(foo,"foo")
```

Logging

Calling get_diagnostics() on a scheduler proxy will give us:

- get_posted_time() → Clock::time_point
- get_started_time() → Clock::time_point
- get_finished_time() → Clock::time_point
- is_interrupted() → bool

Schedulers / Stealing

Asynchronous has a small range of schedulers:

- single_thread_scheduler: one queue, one thread
- asio_scheduler: one io_service per thread
- threadpool_scheduler: one queue, 0..n threads
- multiqueue_threadpool_scheduler: 1..n queues and threads. Threads steal from each others' queues

For the last 2, we have a stealing_xxx version, for use in a composite_threadpool_scheduler, bundling them so they can steal from each other, according to their priority:

```
auto tp = create_shared_scheduler_proxy(new
    multiqueue_threadpool_scheduler<lockfree_queue<>> (1));
auto tp2 = create_shared_scheduler_proxy(new
   stealing_multiqueue_threadpool_scheduler<lockfree_queue<>> (3));
auto tp_worker = create_shared_scheduler_proxy(new
                              composite_threadpool_scheduler<> (tp,tp2,...));
```

Priorities

- Queue priority: to which queue we post a task:

```
auto scheduler =
  create_shared_scheduler_proxy(new single_thread_scheduler<
        any_queue_container<>>
            (any_queue_container_config<threadsafe_list<>>(1),
             any_queue_container_config<lockfree_queue<> >(3)));
```

- Scheduler priority: to which scheduler of a composite we post a task, either directly or through an extra post_callback argument:

```
post_callback(
            [](){},
            [](boost::future<void>){},
            "",
            1, // threadpool prio
            0  // callback prio
);
```

Interacting with Qt

- Make your servant inherit qt_servant
  struct QtServant : public QObject
  , public qt_servant<>

- Use post_callback, as always

```
post_callback(
        [](){return 42;},
        [this](boost::future<int> res){/*...*/},);
```

Advantages:

- All of Asynchronous' threadpools available
- Logging, interrupting of tasks possible
- Algorithms, Distributing, etc.

Continuations

- For recursive tasks (Fibonacci)
- Or for future(s) gotten from whatever task / library
  => create a continuation, called when all tasks are done

Advantages:

- Simple to use
- Works with futures
- Support exceptions
- recursive

Disadvantage:

- Becomes very fast messy. Solution: state machines...

Continuations

```
struct fib_task : continuation_task<long>
{
    fib_task(long n,long cutoff):n_(n),cutoff_(cutoff){}
    void operator()()const
    {
        // the result of this task
        // or an exception
        continuation_result<long> task_res = this_task_result();
        if (n_<cutoff_)
        {
            // n < cutoff => execute ourselves
            task_res.set_value(serial_fib(n_));
        }
        else
```

Continuations

```
    {
      // n_ >= cutoff create 2 new tasks
      create_continuation(
            // called when subtasks are done, set our result
            [task_res]
            (std::tuple<boost::future<long>,boost::future<long>> res)
            {
              try{
                long r = std::get<0>(res).get() + std::get<1>(res).get();
                task_res.set_value(r);
              }
              catch(std::exception& e){
                 task_res.set_exception(boost::copy_exception(e));}
            },
            // recursive tasks
            fib_task(n_-1,cutoff_),
            fib_task(n_-2,cutoff_));
    }
  }
  long n_;long cutoff_;
};
```

Continuations

```
post_callback(
        [n,cutoff]()
        {
            // a top-level continuation is the first one
            // in a recursive serie.
            // Its result will be passed to callback
            return top_level_continuation<long>(fib_task(n,cutoff));
         },
         // callback with fibonacci result.
         [this](boost::future<long> ){/*...*/}
       );
```

Continuations

We have more possibilities:

- create_continuation(
  [](std::vector<boost::future<int>>){},
  std::move(fus)); // fu is std::vector<boost::future<int>>

- create_continuation(
  [](std::tuple<boost::future<int>,boost::future<int>>){},
  std::move(fu1),std::move(fu2));

Distributing: Job Server

Preconditions:

- Task is serializable, as defined by Boost.Serialization
- Return value or exception is serializable

We have a new scheduler, used as a threadpool:

```
auto server_pool=
  create_shared_scheduler_proxy(
  new tcp_server_scheduler<lockfree_queue<any_serializable>>
          (workers,"localhost",12345));
```

Where:

- workers is a scheduler used for (de)serialization of tasks
- localhost and 12345 are the address and port of our server
- We can use post_callback, as always

Distributing: Job Server with Pool

We can use a composite scheduler instead and rely on stealing to execute part of the work in the server application itself:

```
auto composite = create_shared_scheduler_proxy
    (new composite_threadpool_scheduler<any_serializable>
            (worker_pool,server_pool));
```

Where:

- server_pool is as before
- worker_pool is any threadpool

Distributing: Simple Job Client

A simple client connects to a server regularly or when its queue is under a given size and steals job, returning result or exception:

```
simple_tcp_client_proxy
proxy(comSched,pool,server_address,server_port,executor,
        20 /*ms between calls to server*/);
```

Where:

- comSched is an asio_scheduler for communication.
- pool is any threadpool for job execution
- server_address/port: where to find the job server
- executor: functor deserializing and executing jobs

Distributing: Job Client + Server

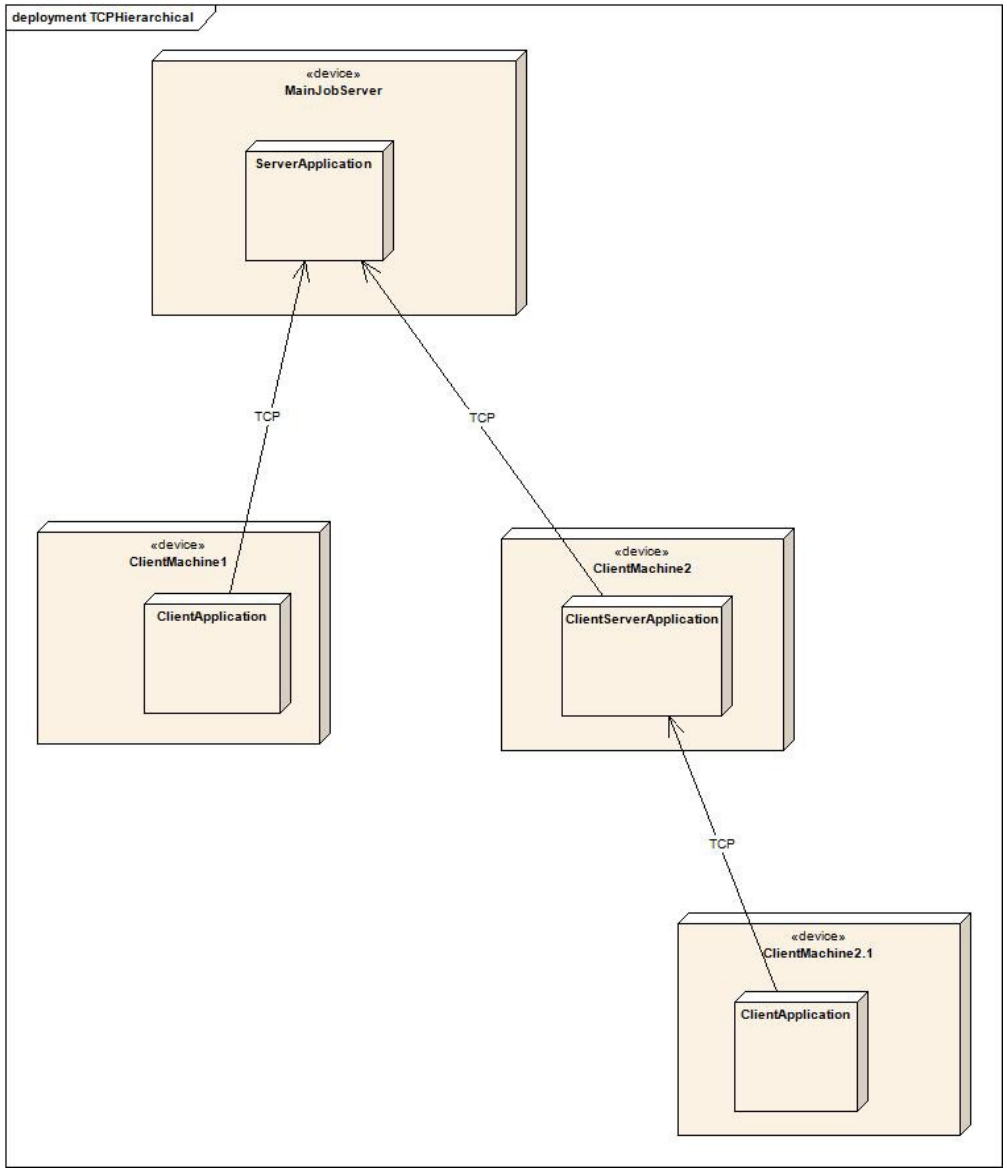
We can add a server to our client, stealing jobs on client request, using... a composite:

```
auto composite =
    create_shared_scheduler_proxy(new
        composite_threadpool_scheduler<any_serializable>
                                                (pool,tcp_server));
```

Where:

- pool is the client pool, as just defined
- tcp_server: a tcp_server_scheduler, like our job server

Building your own network



- A Server Application serves as primary job server
- ClientMachine1 is a simple job client
- ClientMachine2 executes jobs and offers a server part
- ClientMachine2.1 is a simple job client and steals from ClientMachine2
- There can be more clients connecting to MainJobServer or ClientMachine2
- Or maybe one more ClientServerApplication on ClientMachine2?

Parallel algorithms

Asynchronous offers a small range of parallel algorithms with more to come:

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

All are:

- Continuation-based
- Non-blocking
- Distributable
- Work with iterators, ranges, continuations (combinable)

Parallel algorithms examples

There are four versions of this algorithm. A version with iterators or range:

```
post_callback(
    [this]()
   {
      return parallel_for(this->m_data.begin(),this->m_data.end(),
                          [](int const& i)
                          {
                              const_cast<int&>(i) += 2;
                           },1500 /*cutoff*/);
    },
    // callback functor.
    // Servant is alive if this is called
    [this](boost::future<void> ){/*...*/}
 );
```

The caller must ensure interators stay valid until callback. We get no result in the future.

Parallel algorithms examples

Better, let Asynchronous take care of data lifetime:

```
/*std::vector<int> data;*/
post_callback(
    [data=std::move(data)]()
   {
      return parallel_for(std::move(data),
                          [](int const& i)
                          {
                              const_cast<int&>(i) += 2;
                           },1500 /*cutoff*/);
    },
    // callback functor.
    // Servant is alive if this is called
    [this](boost::future<std::vector<int>> ){/*...*/}
 );
```

The caller gets the data (possibly modified) back in the future.

Parallel algorithms examples

Let's combine!

```
/*std::vector<int> data;*/
post_callback(
    [data=std::move(data)]()
    {
       return parallel_for(parallel_for(std::move(data),
                                        [](int const& i)
                                        {
                                           const_cast<int&>(i) += 2;
                                        },1500 /*cutoff*/),
                           [](int const& i)
                           {
                              const_cast<int&>(i) += 2;
                           },1500 /*cutoff*/);
    },
    [this](boost::future<std::vector<int>> ){/*...*/}
 );
```

What happens? A parallel modification of all elements, then when done, another one. This is not only for parallel_for possible.

Parallel PI

http://goparallel.sourceforge.net/calculate-pi-with-custom-c-class/

Calculating PI in parallel is an embarassingly parallel problem. We need to Sum from 0 → N. This can be done by dividing this range in parts, and execute them in parallel.

The formula fives us a quarter of PI so we still need to multiply by 4.

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

Parallel PI

```
struct pi {
   double operator()(long n) {
       return ((double) ((((int) n)% 2 == 0)?1:-1))/((double) (2*n+1));
    }
};
post_callback(
    [this]()
    {
       return invoke(
              /*We start with a parallel_reduce calling operator +*/
              parallel_reduce(
                            /*apply pi() to numbers from 0 to COUNT*/
                            lazy_irange(0L, COUNT, pi()),
                            [](double a, double b) { return a + b; },
                            STEP_SIZE),
              /*when done we need to multiply by 4*/
              [](double a) { return a * 4.0; });
    },
    [](boost::future<double> ){/*...*/}
 );
```

Where to find Asynchronous

https://github.com/henry-ch/asynchronous