Asynchronous IO with Boost.Asio

Boost.Asio

ciere consulting

Michael Caisse

michael.caisse@ciere.com | follow @MichaelCaisse

ciere.com

Part I

Introduction

ciere.com

Outline

ciere.com

What is Asio

An Asynchronous I/O Library

- Started as a network library. Also resources:
  - Serial Ports
  - Timers
  - File Descriptors
  - Write your own! (extensible)
- Uses an efficient Proactor model
- Extremely Scalable - Easily supporting thousands of connections.
- Provides a Portable Abstraction

What is Asynchronous I/O

Daughter #1

me:	“Please make me a coffee.”
daughter:	“Sure Dad”

time passes ... I work. She makes a cappuccino.

daughter:	“Here is your coffee.”
me:	“Thanks”

What is Asynchronous I/O

Daughter #3

me:	“Please make me a coffee.”
daughter:	“I would love to!”

we both walk to the machine. I supervise (watch). She makes a cappuccino.

daughter:	“Here is your coffee.”
me:	“Thanks”

ciere.com

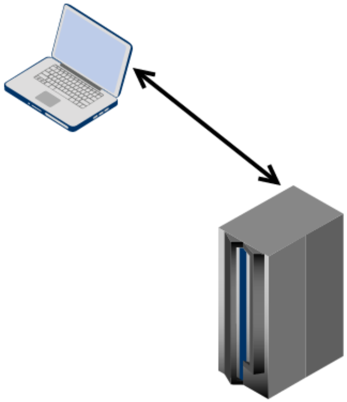
What is Asynchronous I/O

```
// completion handler
void done_reading()
{
  //...
}


read_file(filename, buffer, done_reading);
// ... do work
```

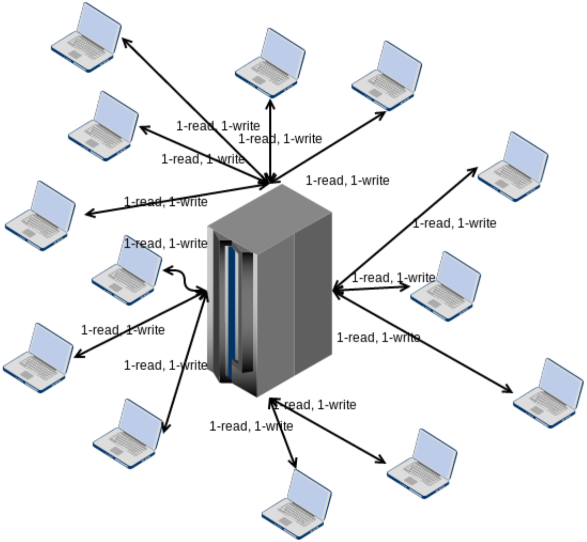
ciere.com

Why Asynchronous I/O?

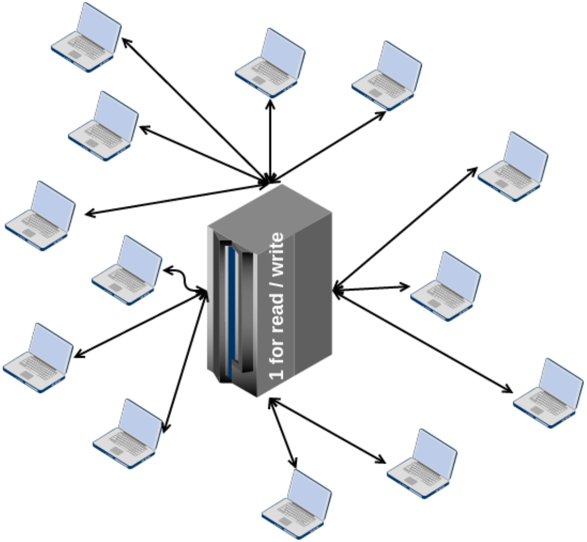


ciere.com

Why Asynchronous I/O?



Asio Asynchronous

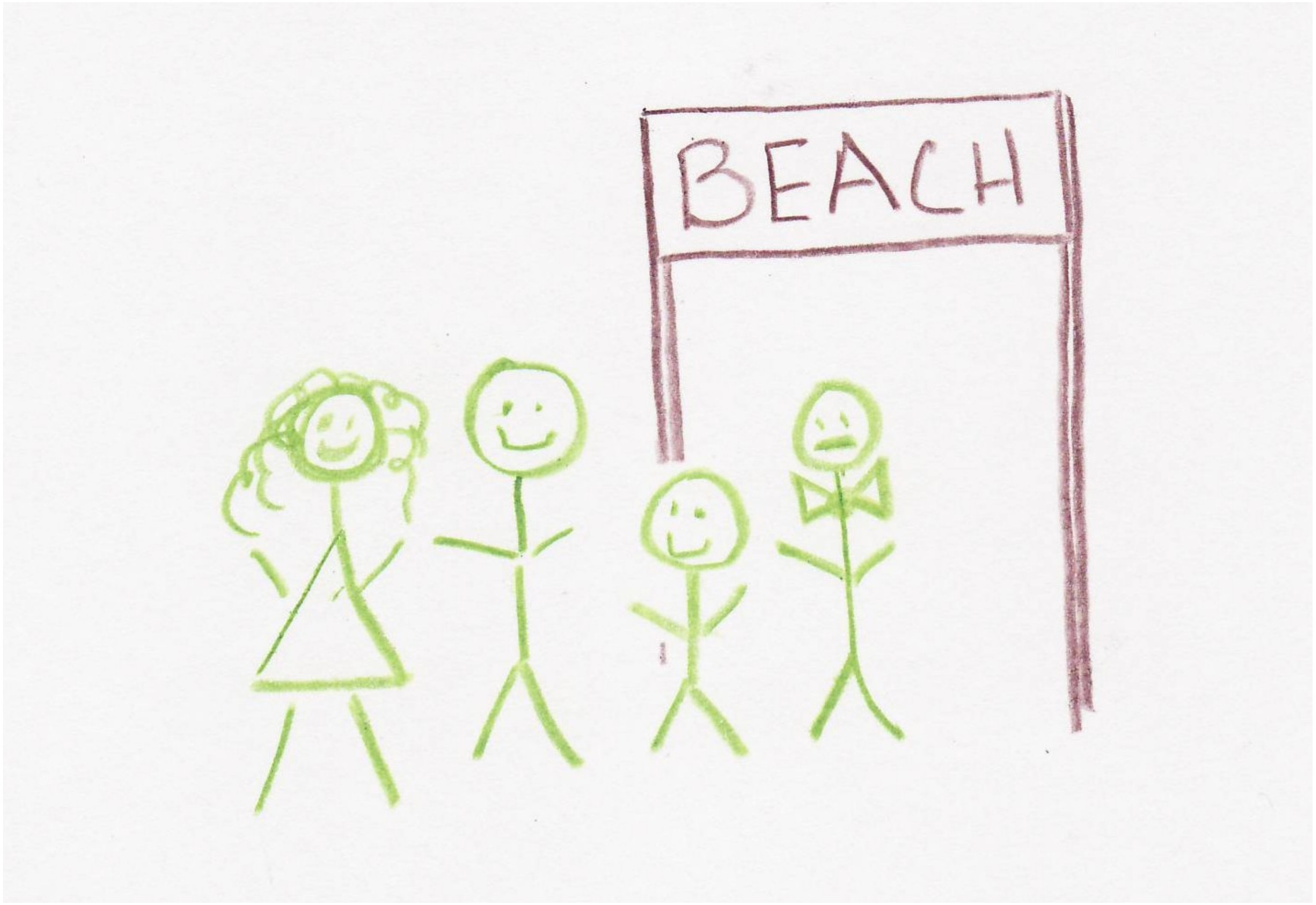


ciere.com

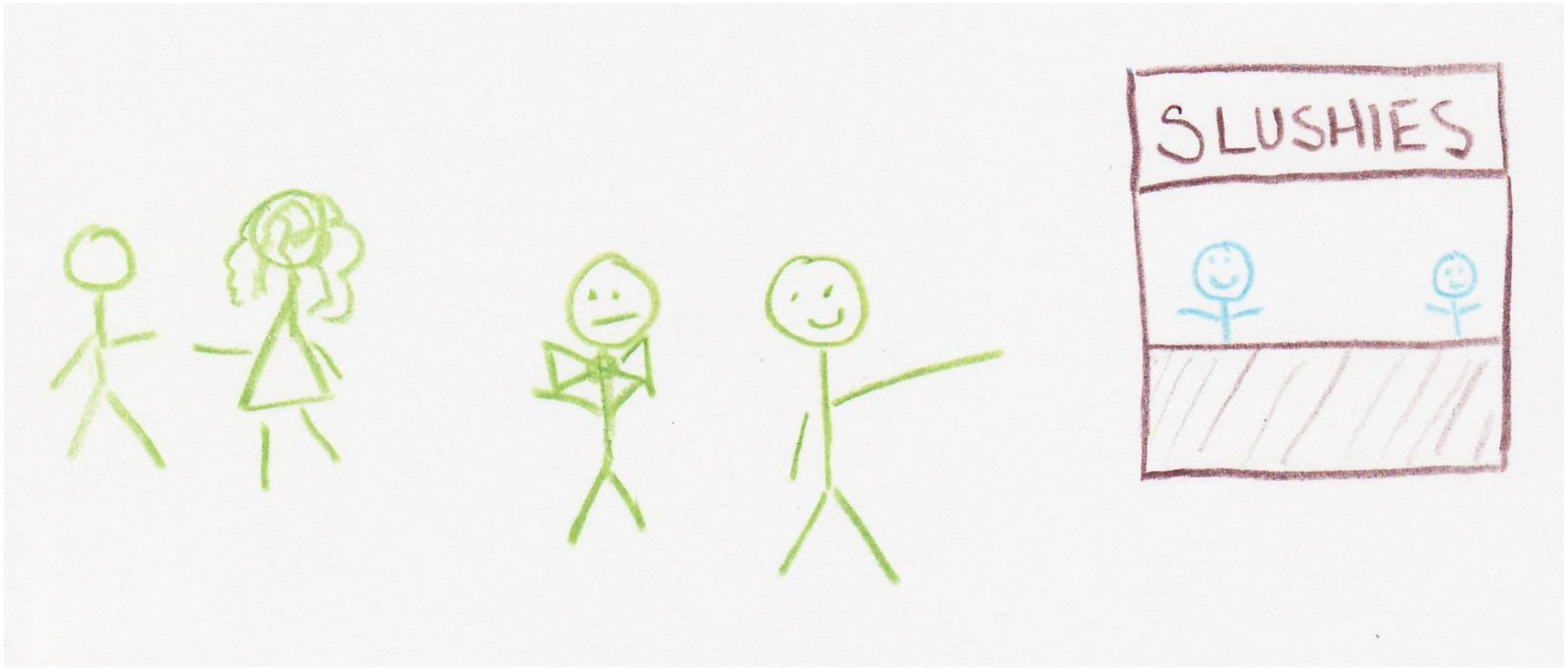
A Proactor Story

- or -

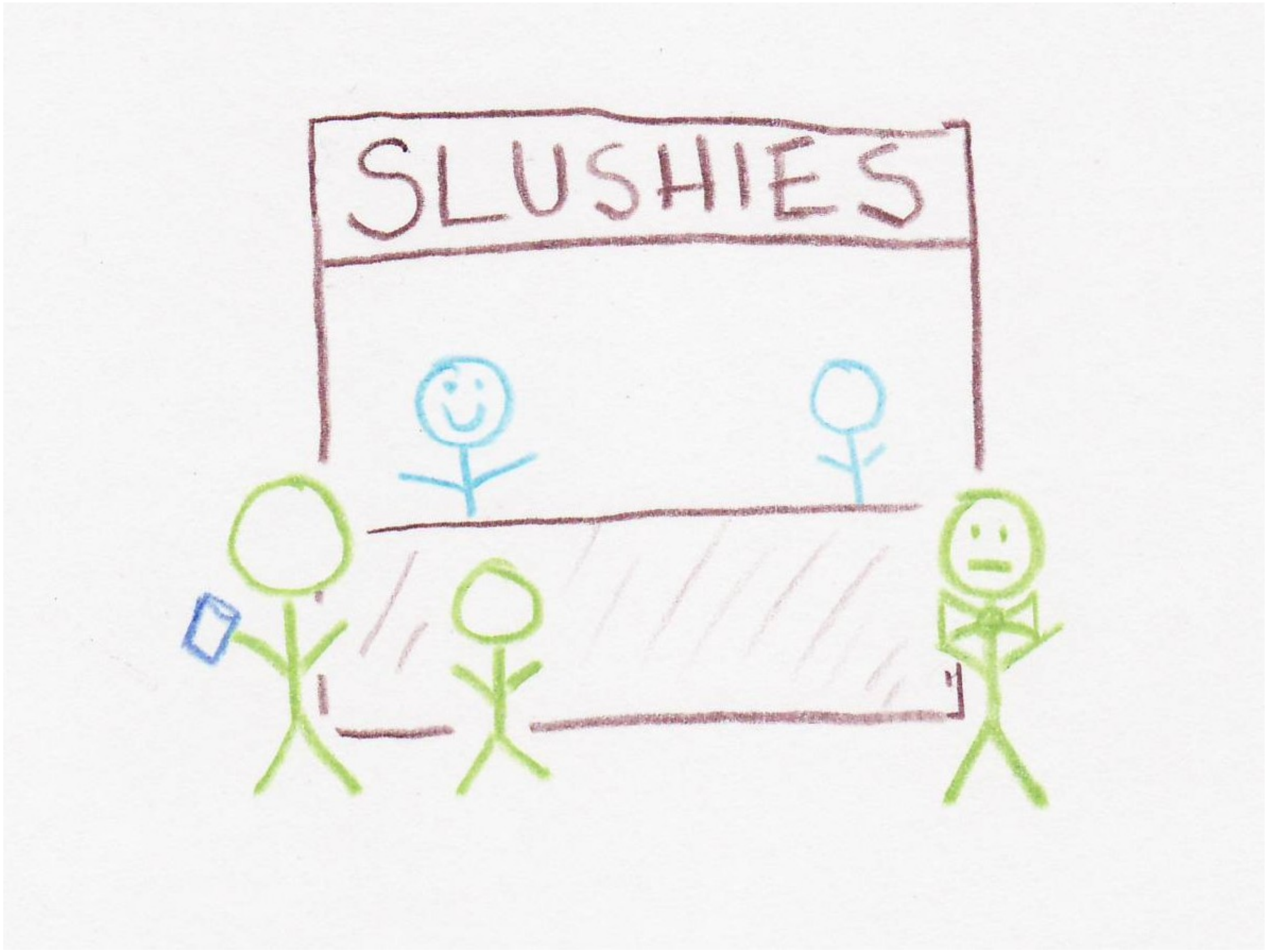
Purple Slushies, Butlers and Brain Freeze



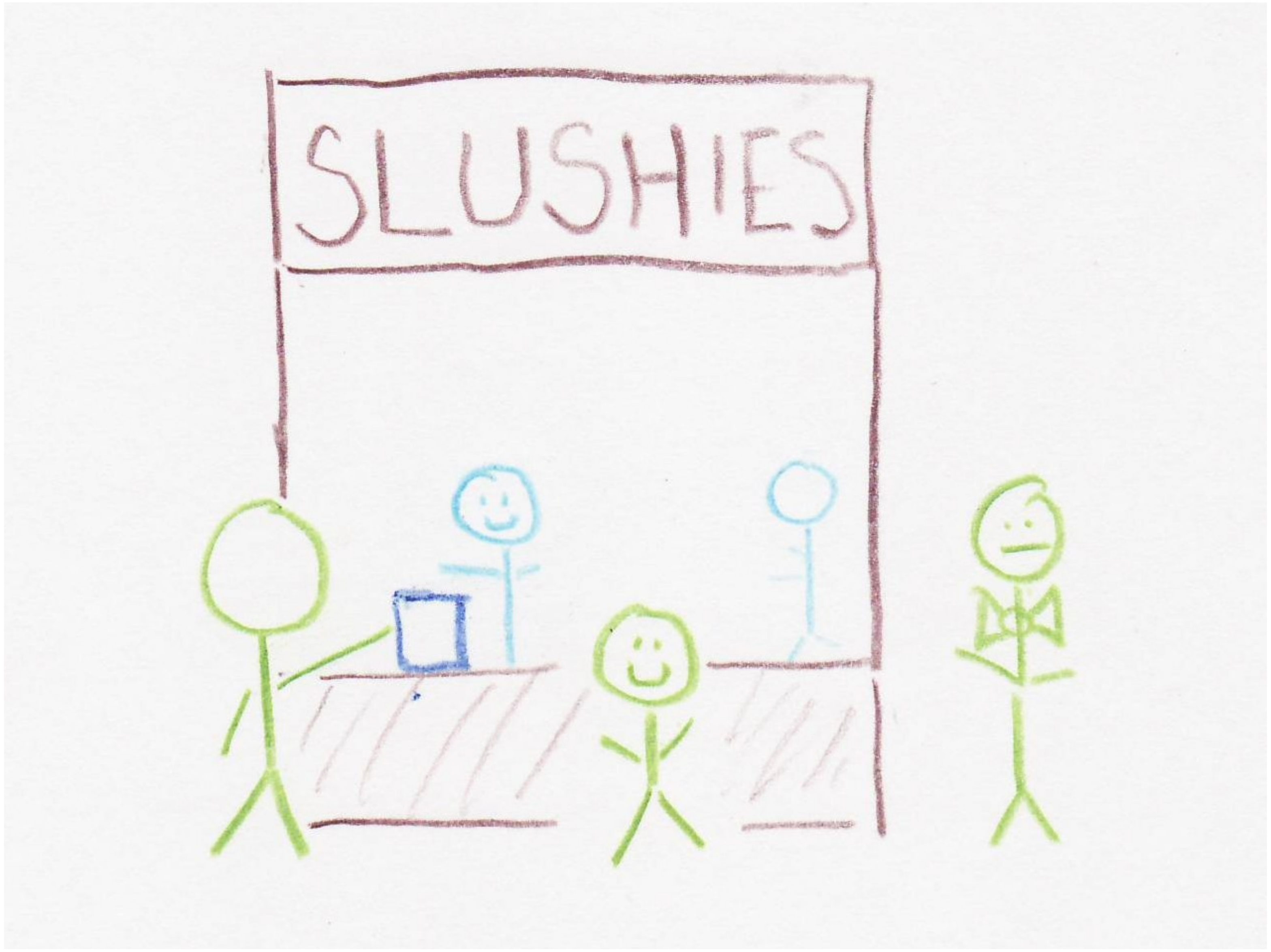
Mom, Dad, Johnny and Butler go to the beach.



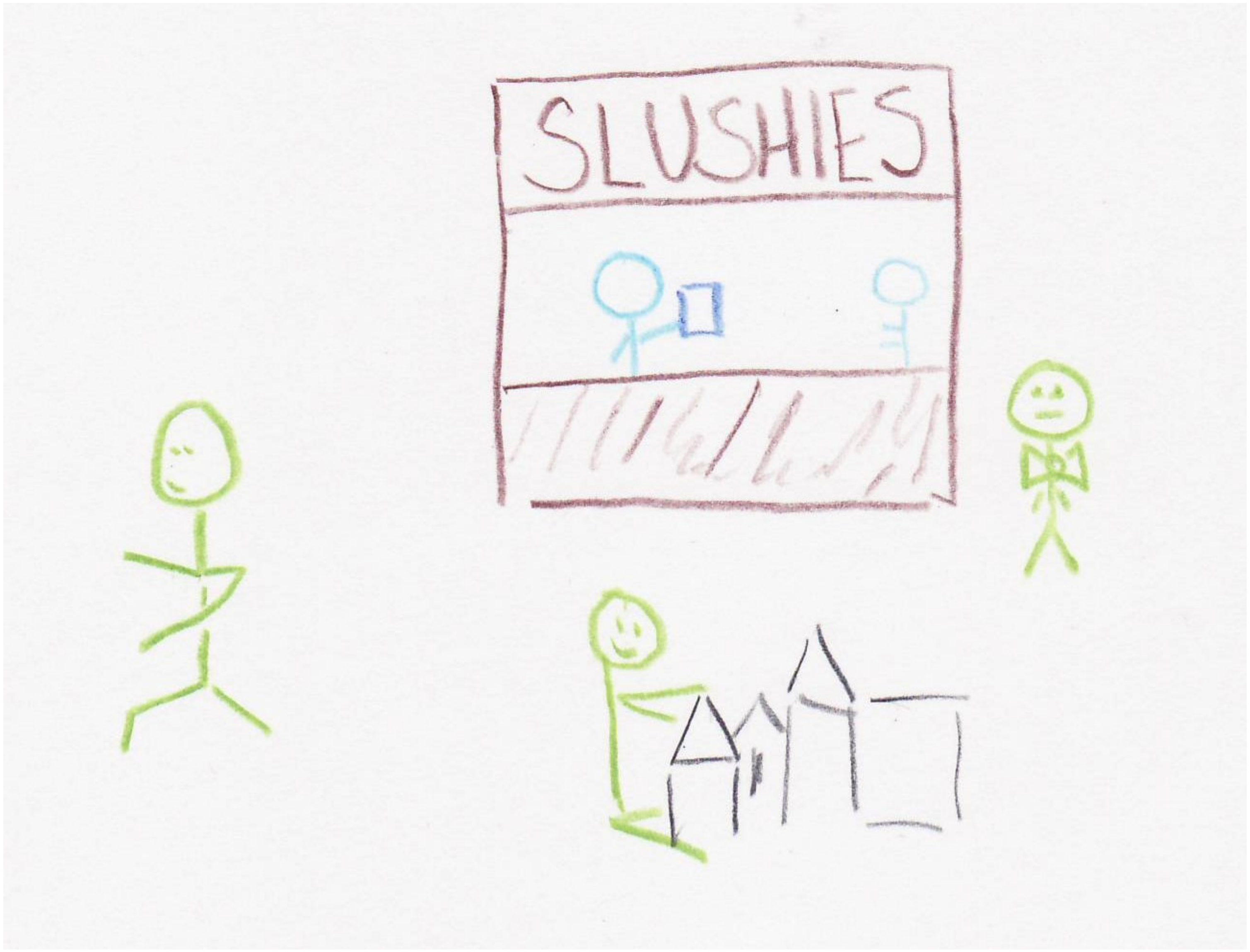
Dad tells Butler to wait at the Slushie Shack.



After some time, Dad and Johnny go to get a slushie.
Dad brings his own cup. He is greeted by the Owner.



“I would like to order a slushie. Here is my cup. Please deliver it to Johnny when it is ready.



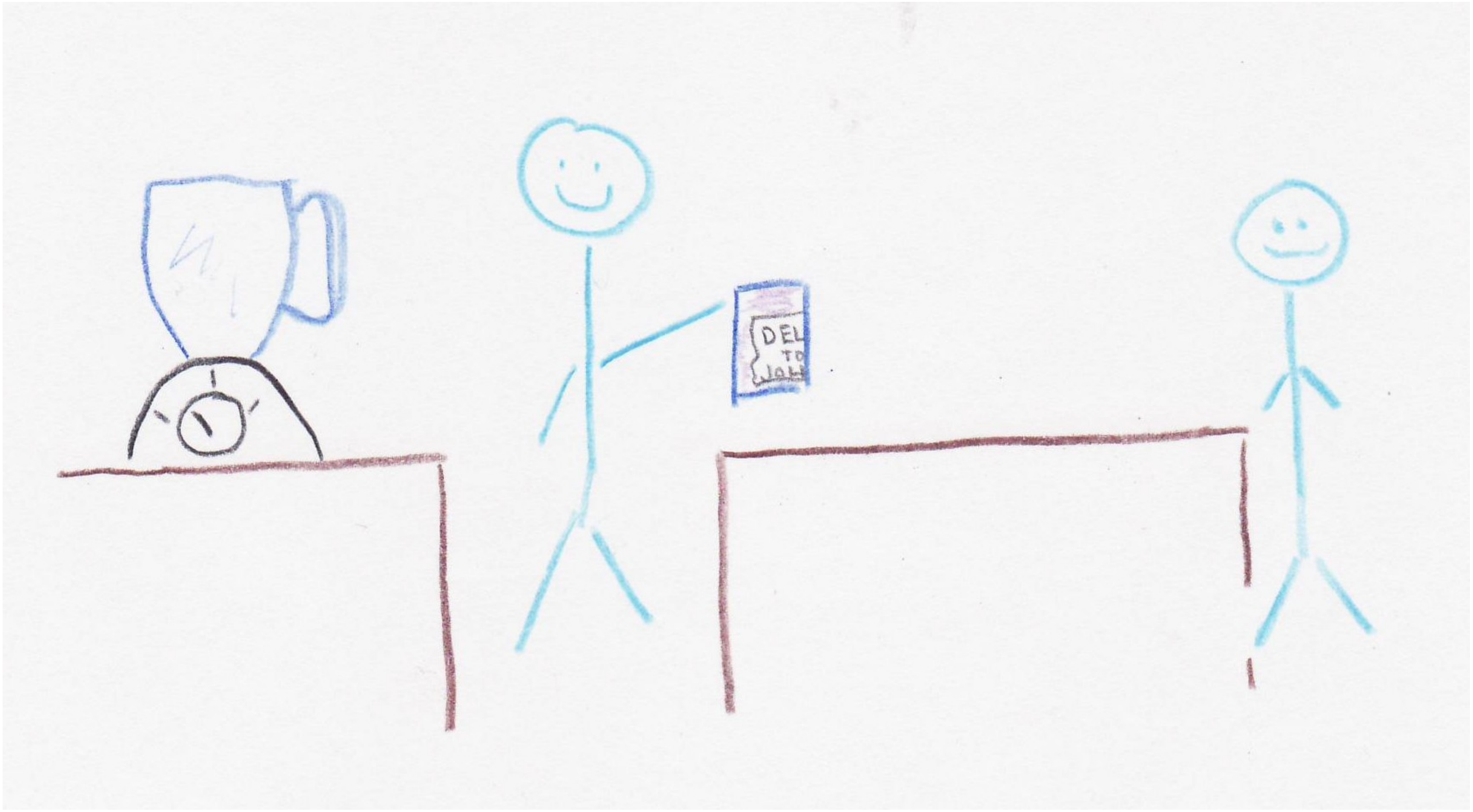
Dad heads off to explore the beach. Johnny builds a sandcastle. Owner begins to make the slushie.
And Butler waits.

DELIVER TO JOHN

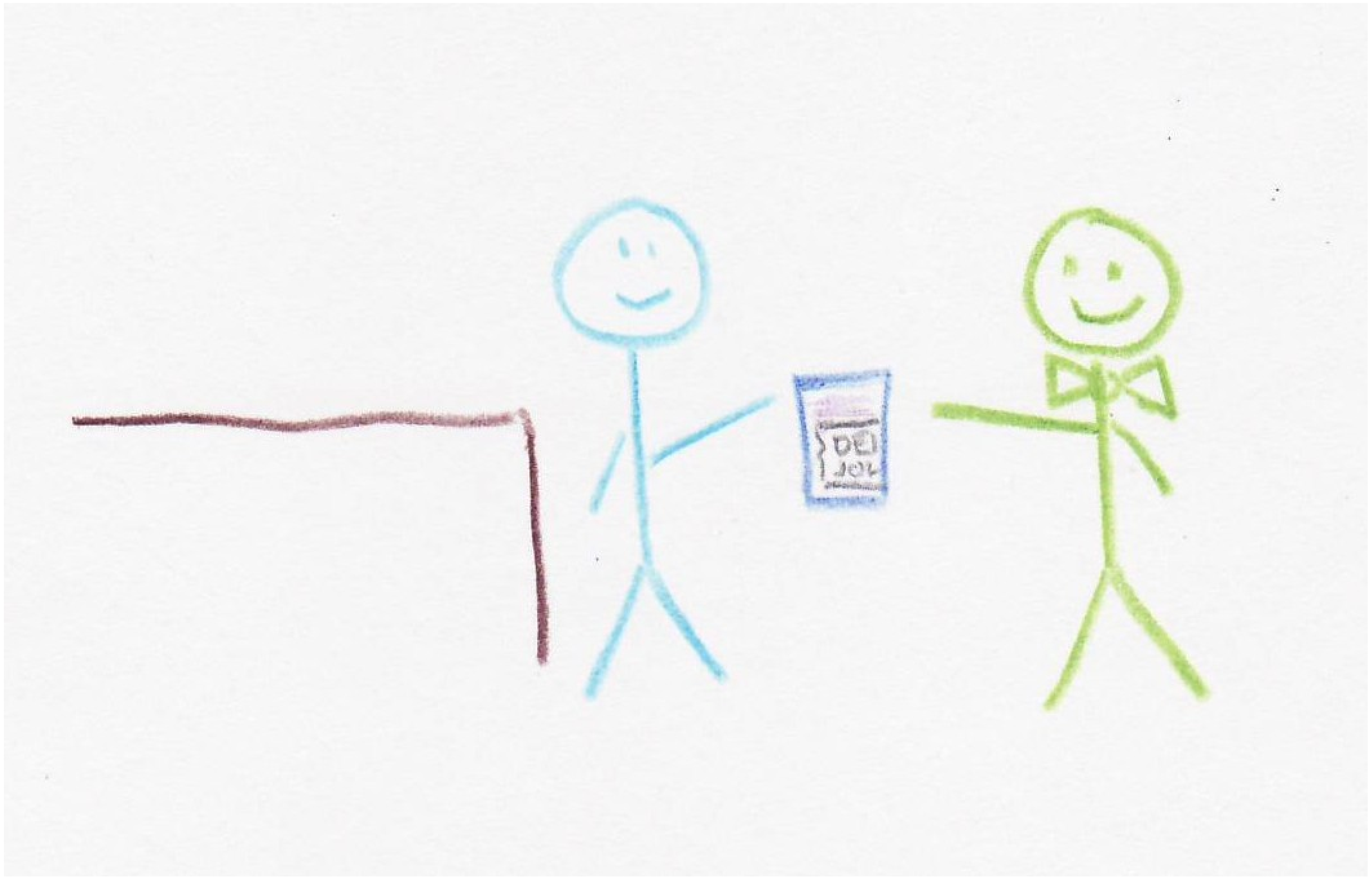
Owner starts the blender and goes back to take the next customer's order.

… *<time passes>* …

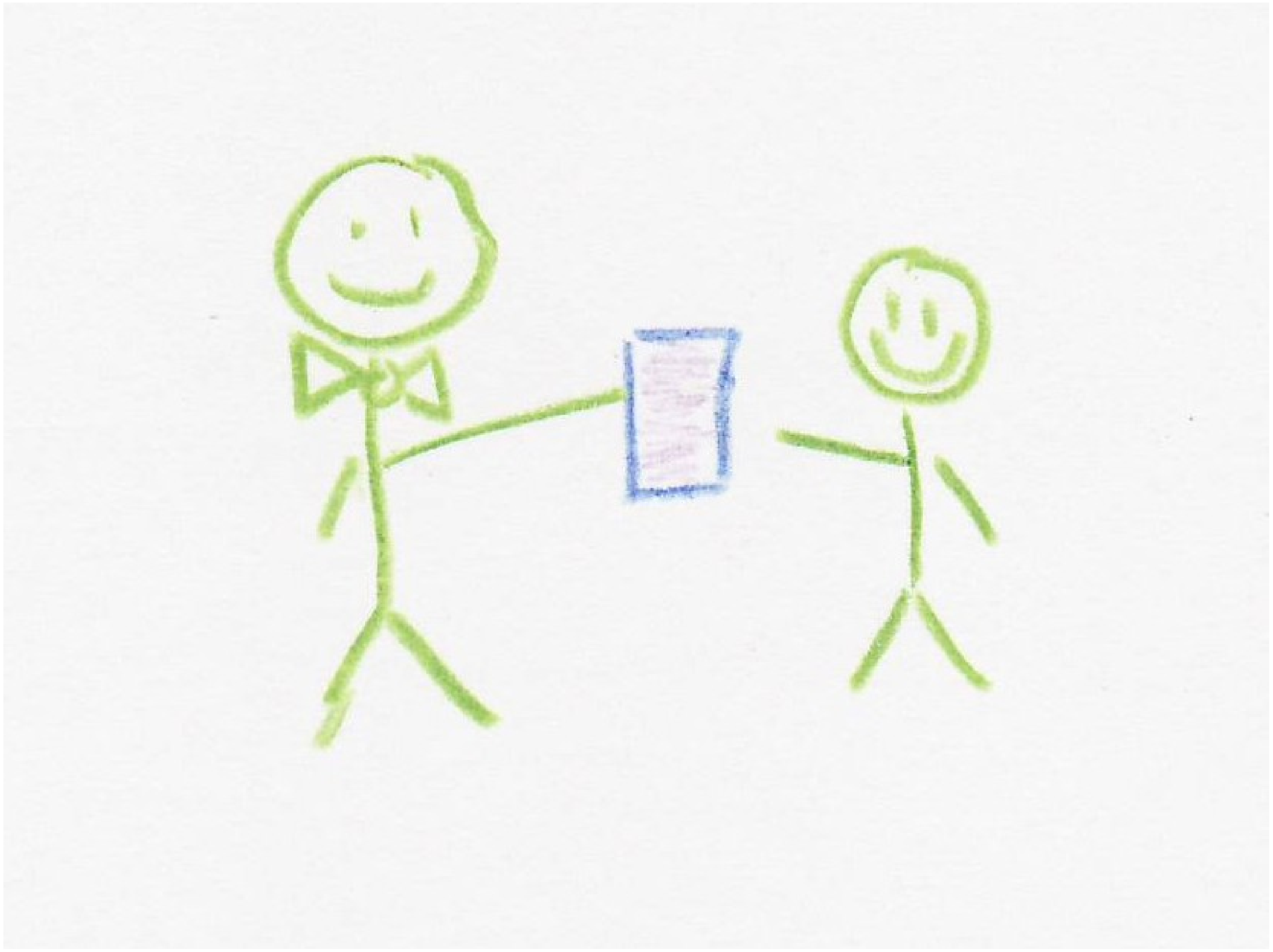
“Ding”



Slushie is ready and Owner moves the cup to the completion table where Assistant is waiting.



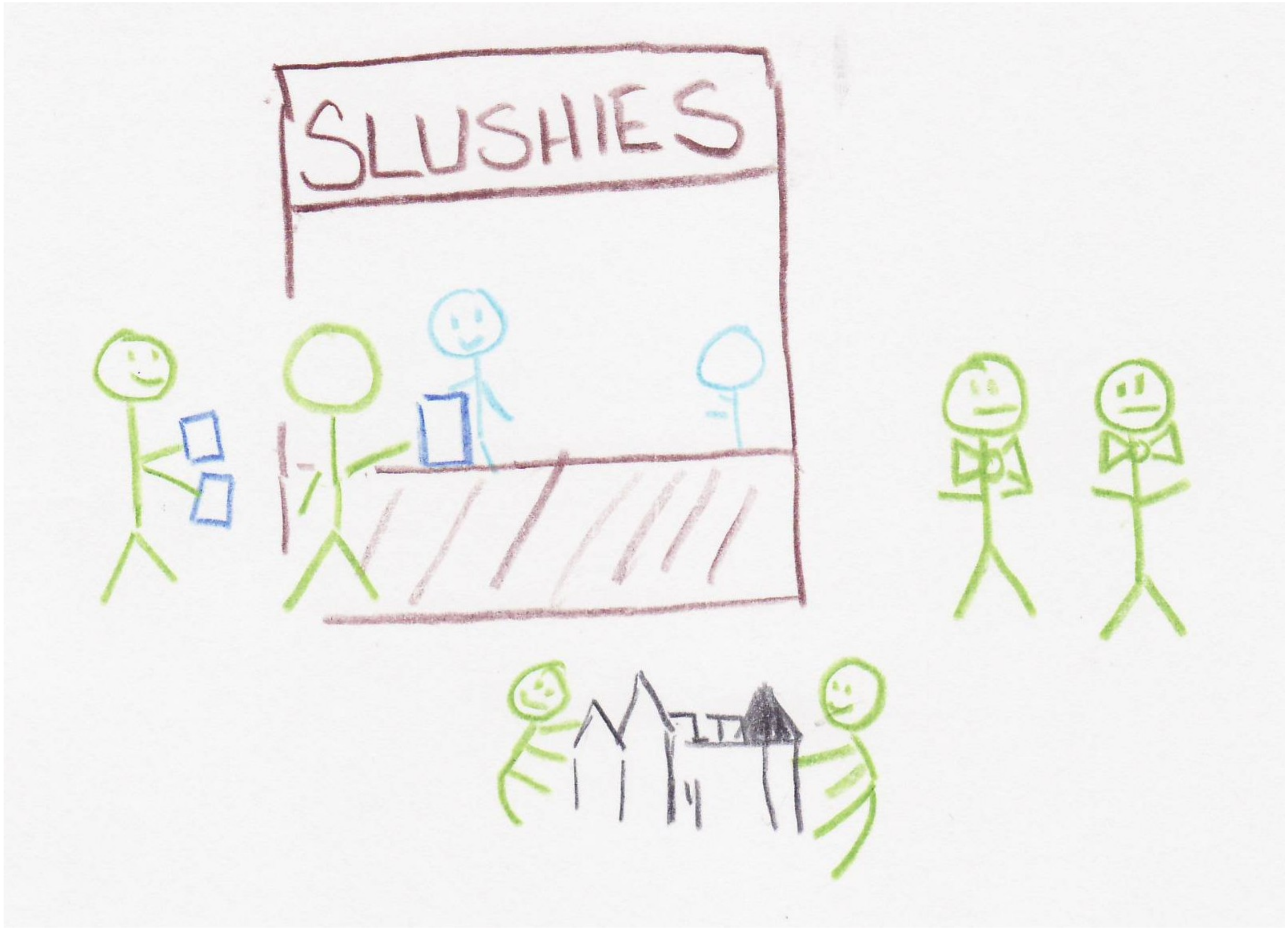
The Assistant gives the slushie to Butler for delivery to Johnny. Butler is happy to have something to do.



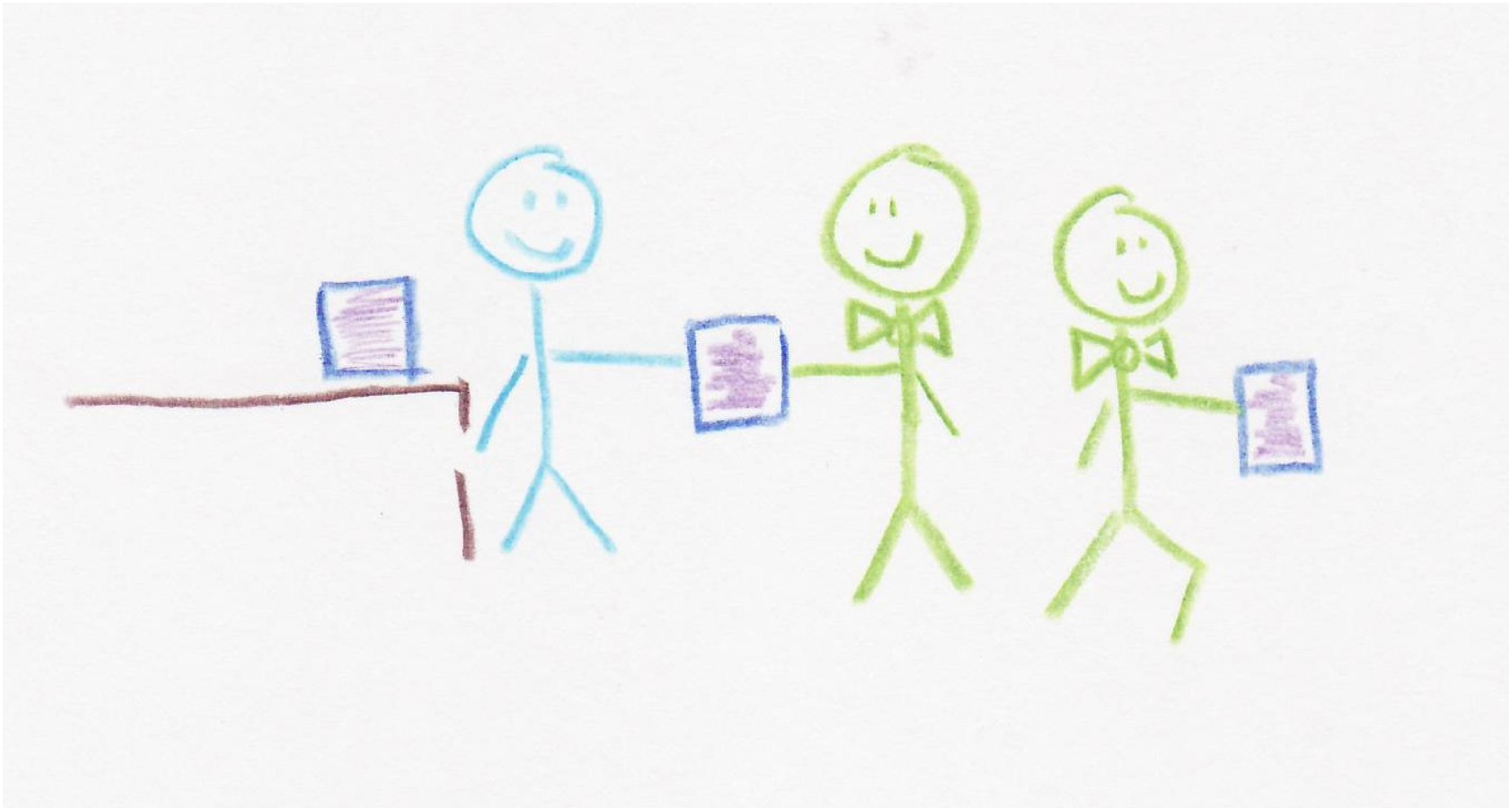
Butler delivers the slushie to Johnny who is happy too. Butler returns to the Slushie Shack and waits.

Sometimes Dad will order multiple slushies.
One for Mom and one for Johnny.

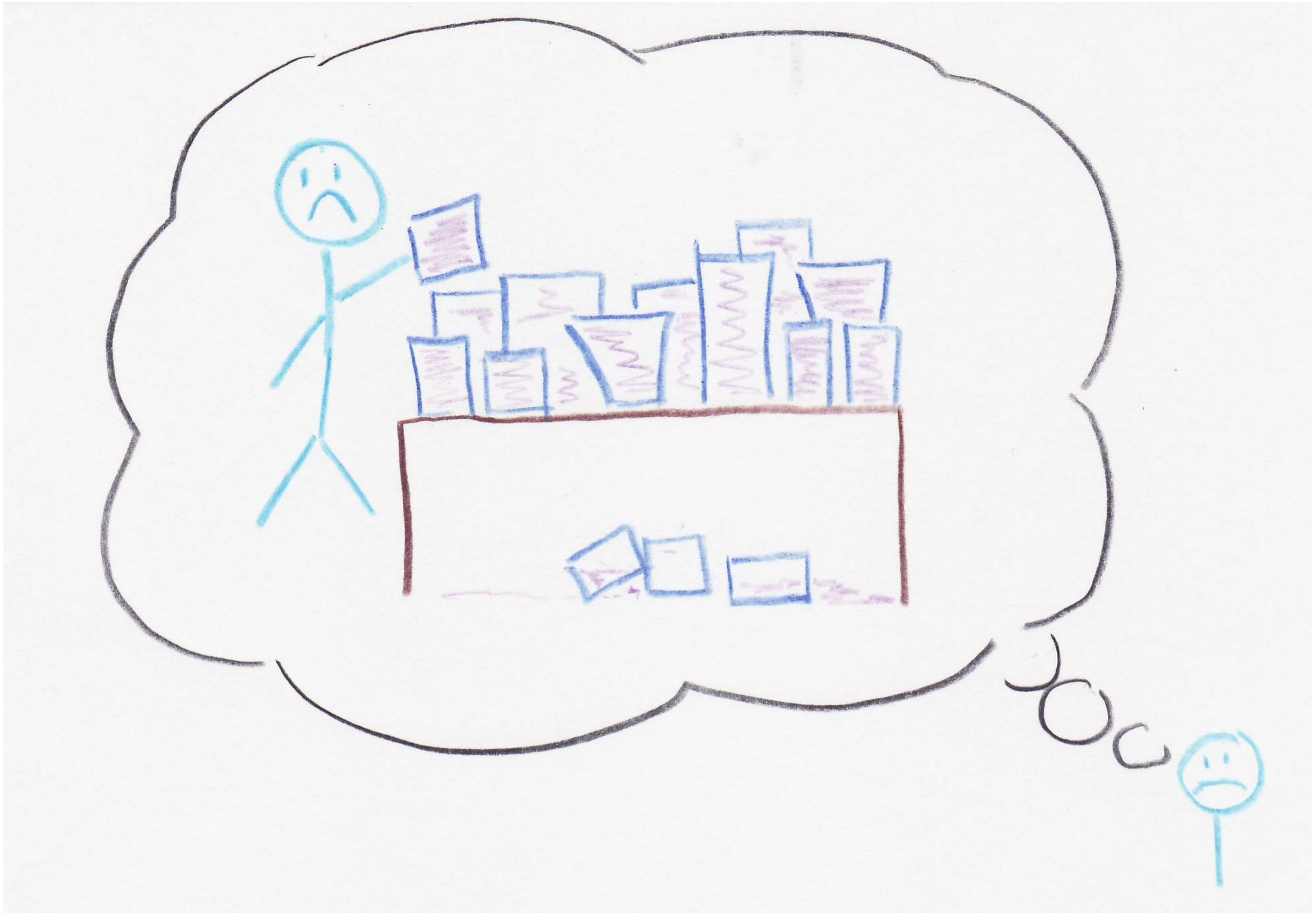
That isn't a problem. Assistant just gives the first one ready to Butler. Bulter can only deliver one at a time and returns for the second slushie.



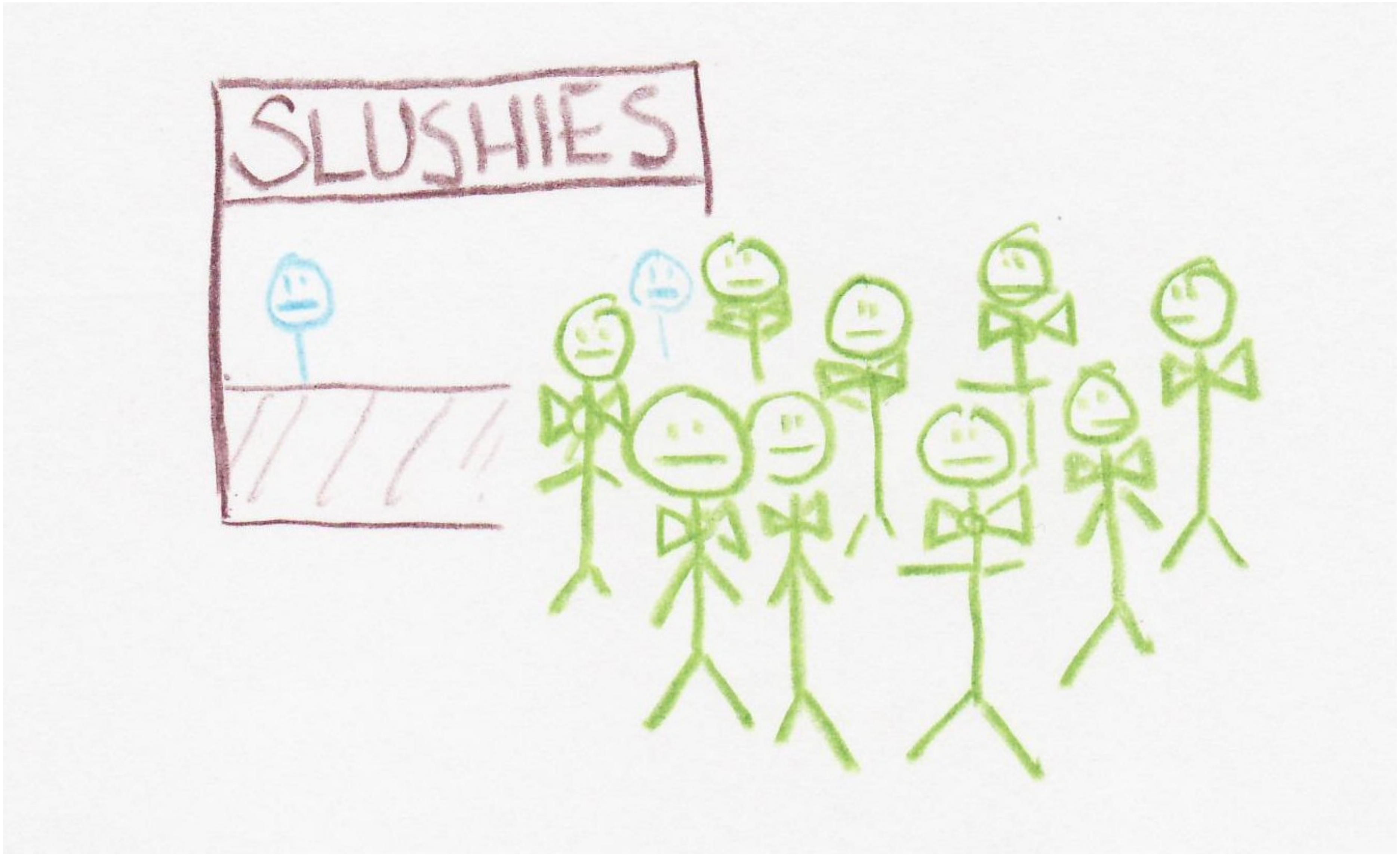
Other families come to the beach and bring their butlers who also wait in the slushie completion line.



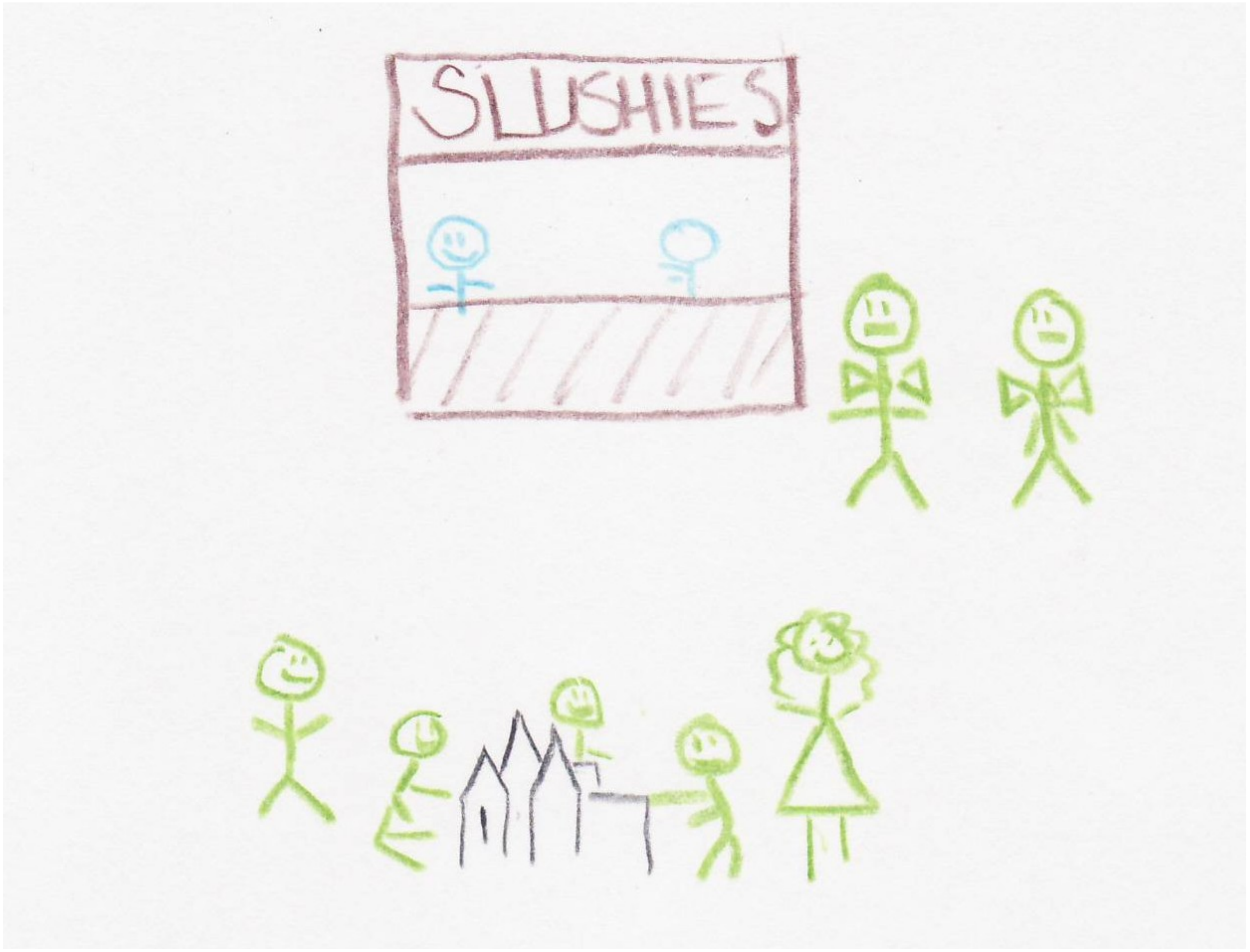
This works well because it helps keep Assistant's slushie completion table empty.



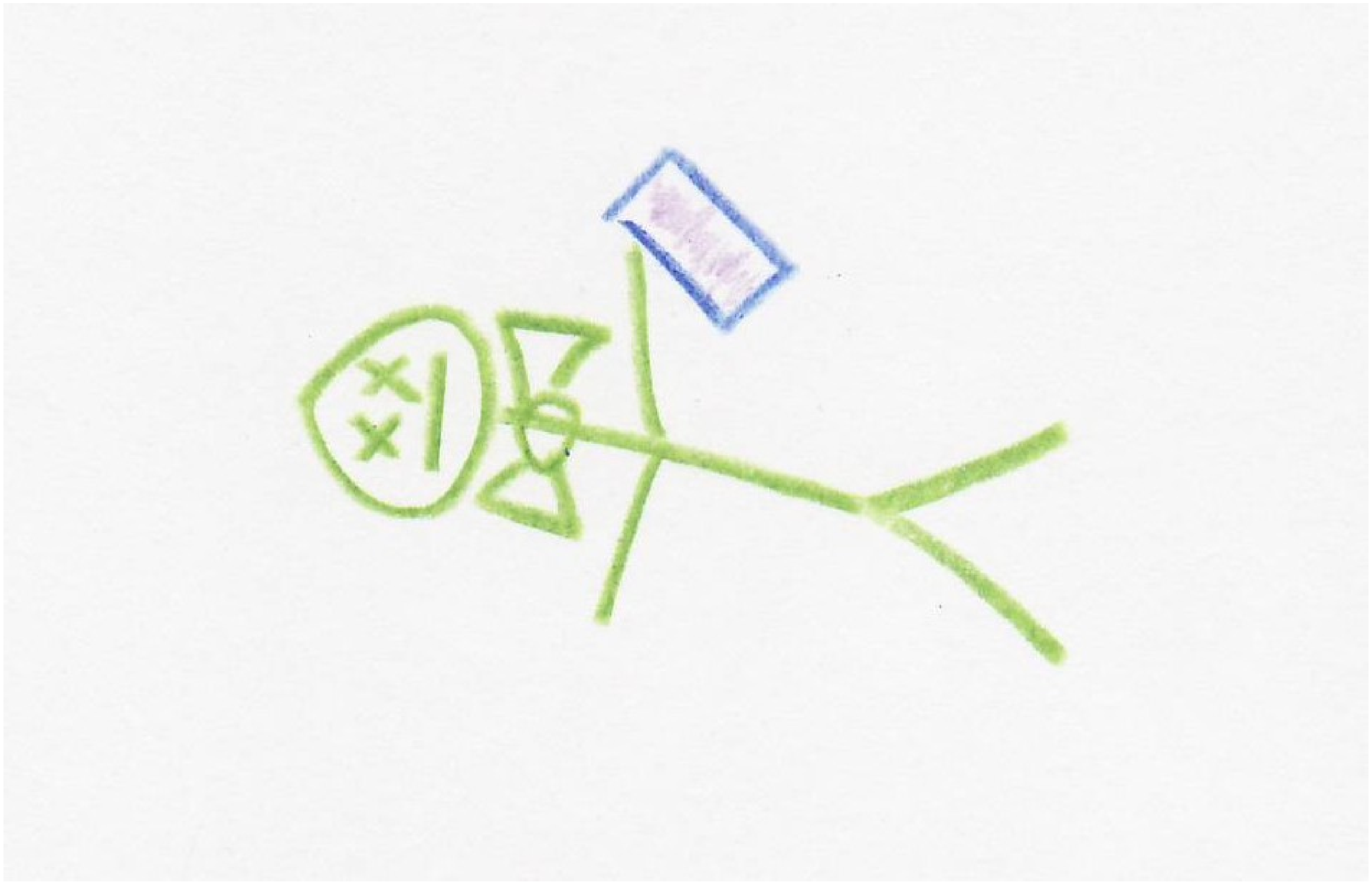
Assistant still remembers that fateful day when no butlers came to the beach.



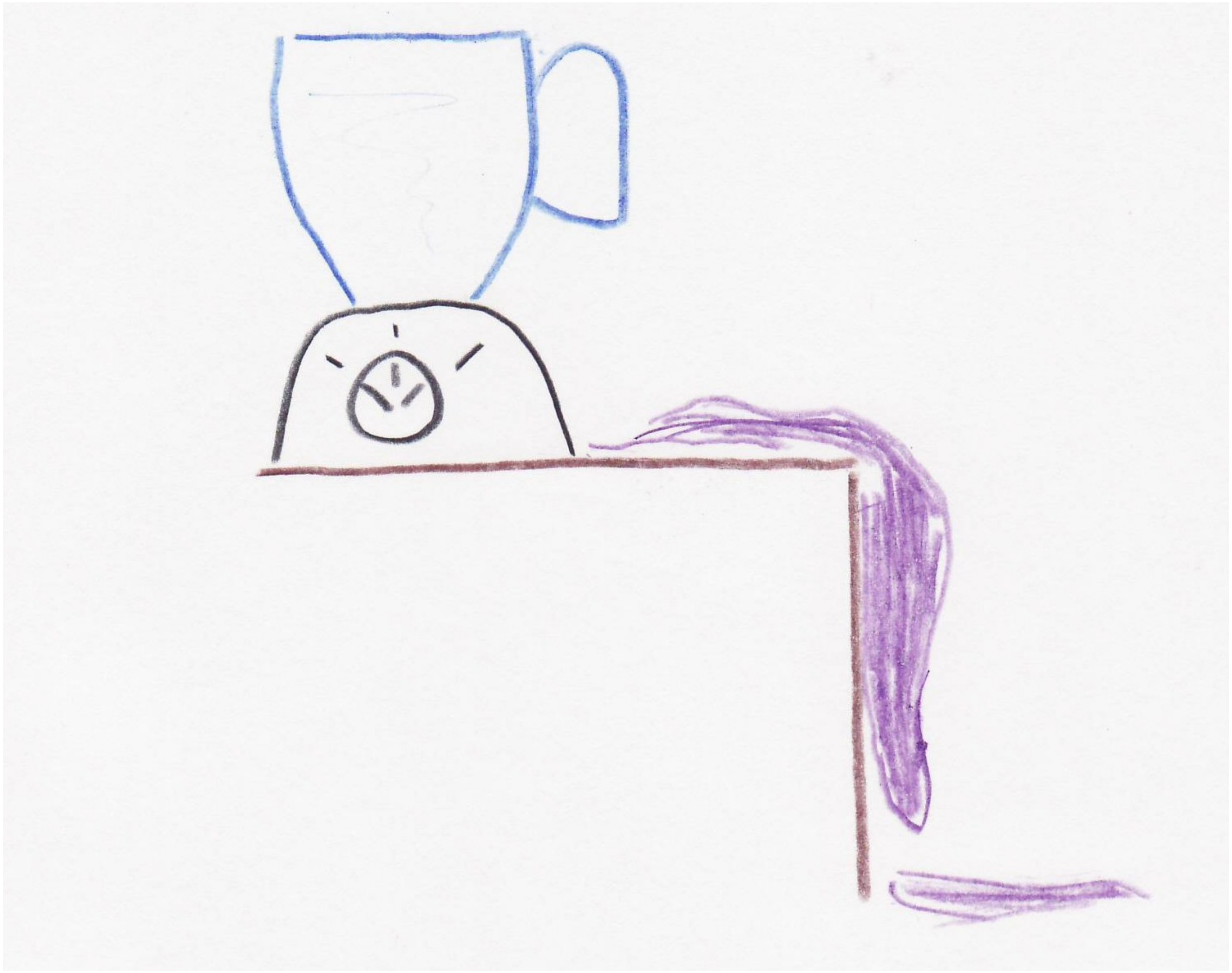
There was also the time that each kid brought a butler. Disaster! No room at the shack. Too busy, yet nothing was getting done.



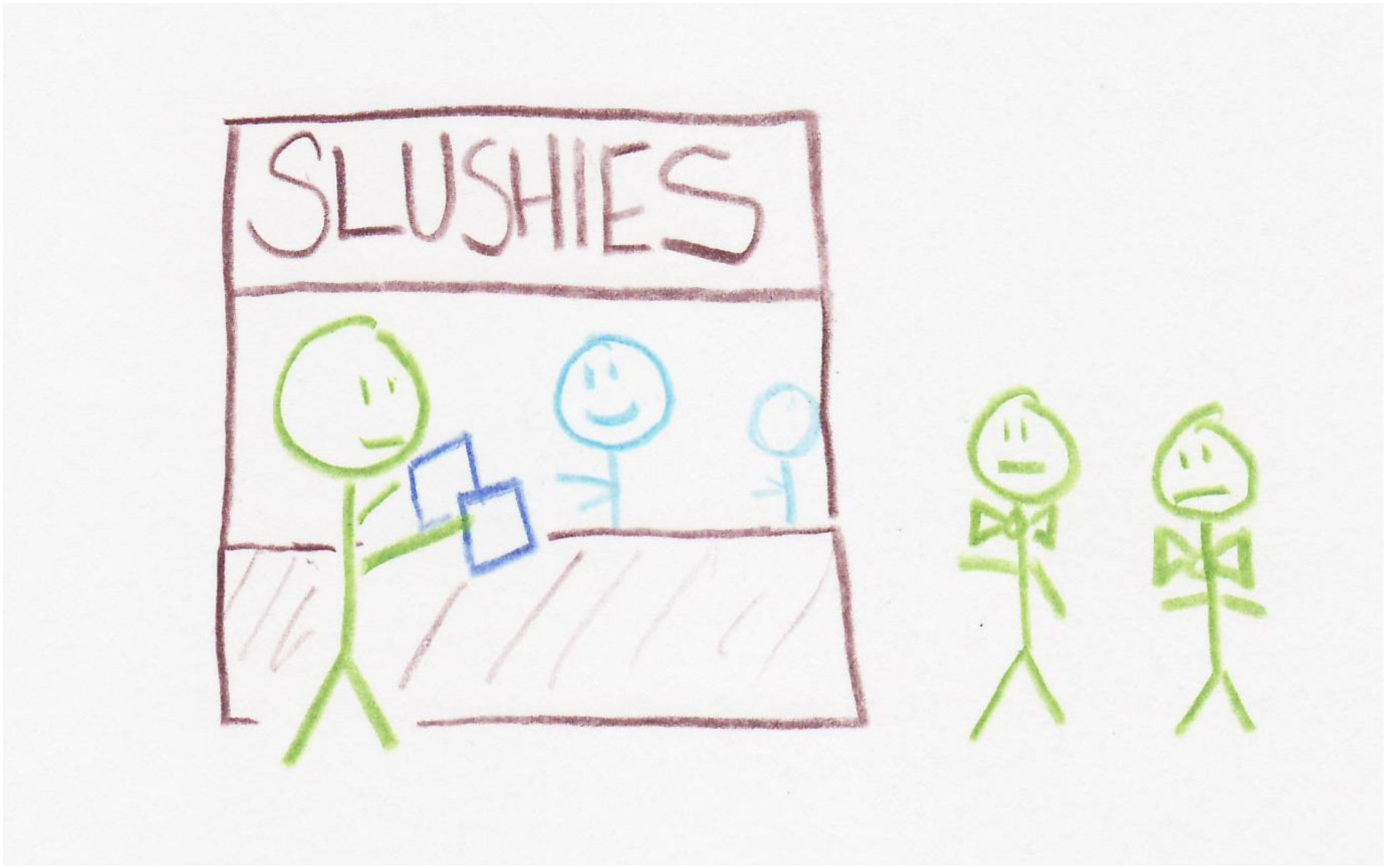
The families agreed that two butlers would be plenty for all. They now share.



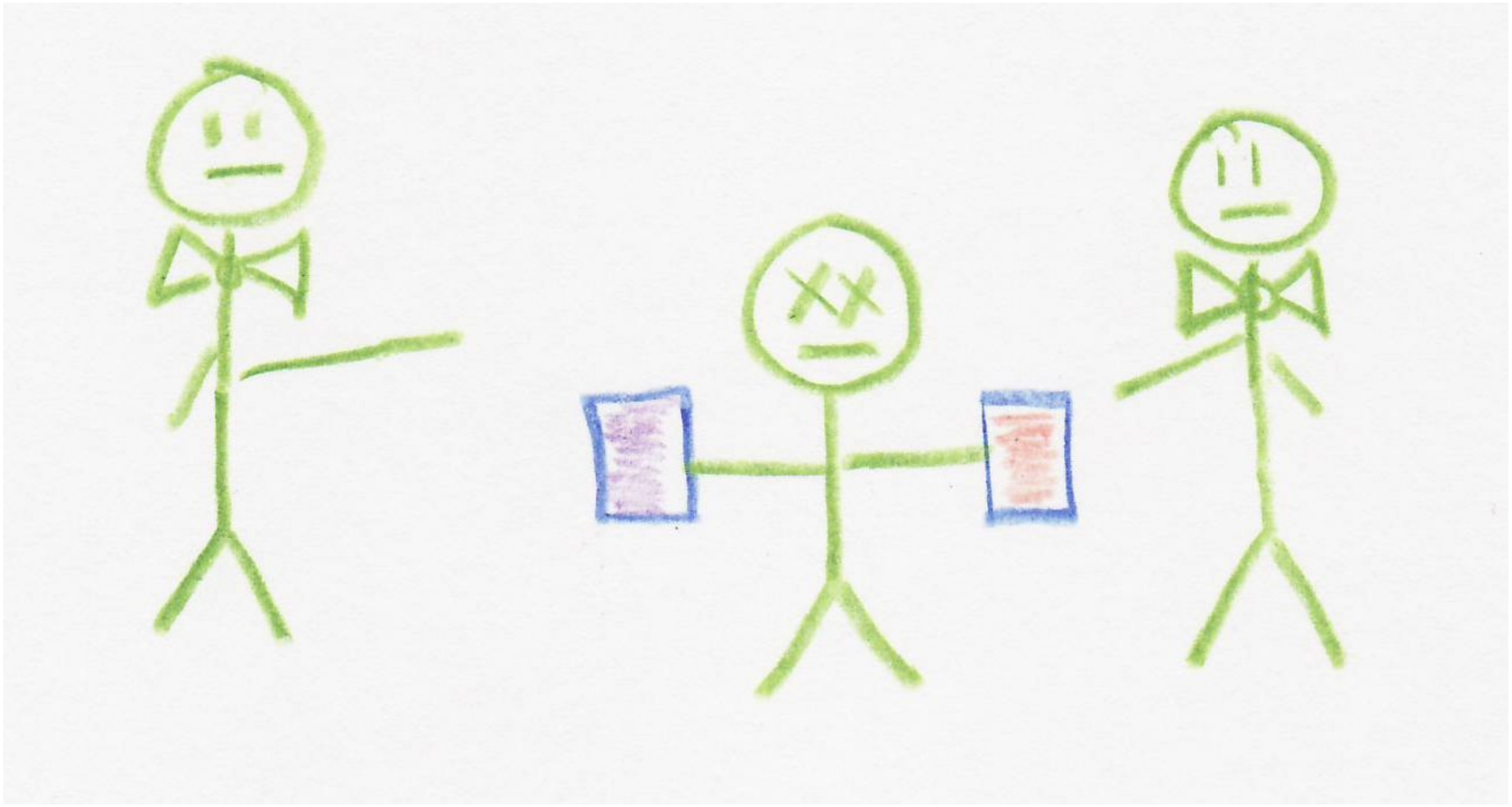
Occasionally tragedy strikes. Johnny will leave to chase waves without getting his slushie. Butler will die of exhaustion trying to find him.



… or somebody will take their cup and go home while the slushie is being made. Then it gets poured on the floor. Yuck.

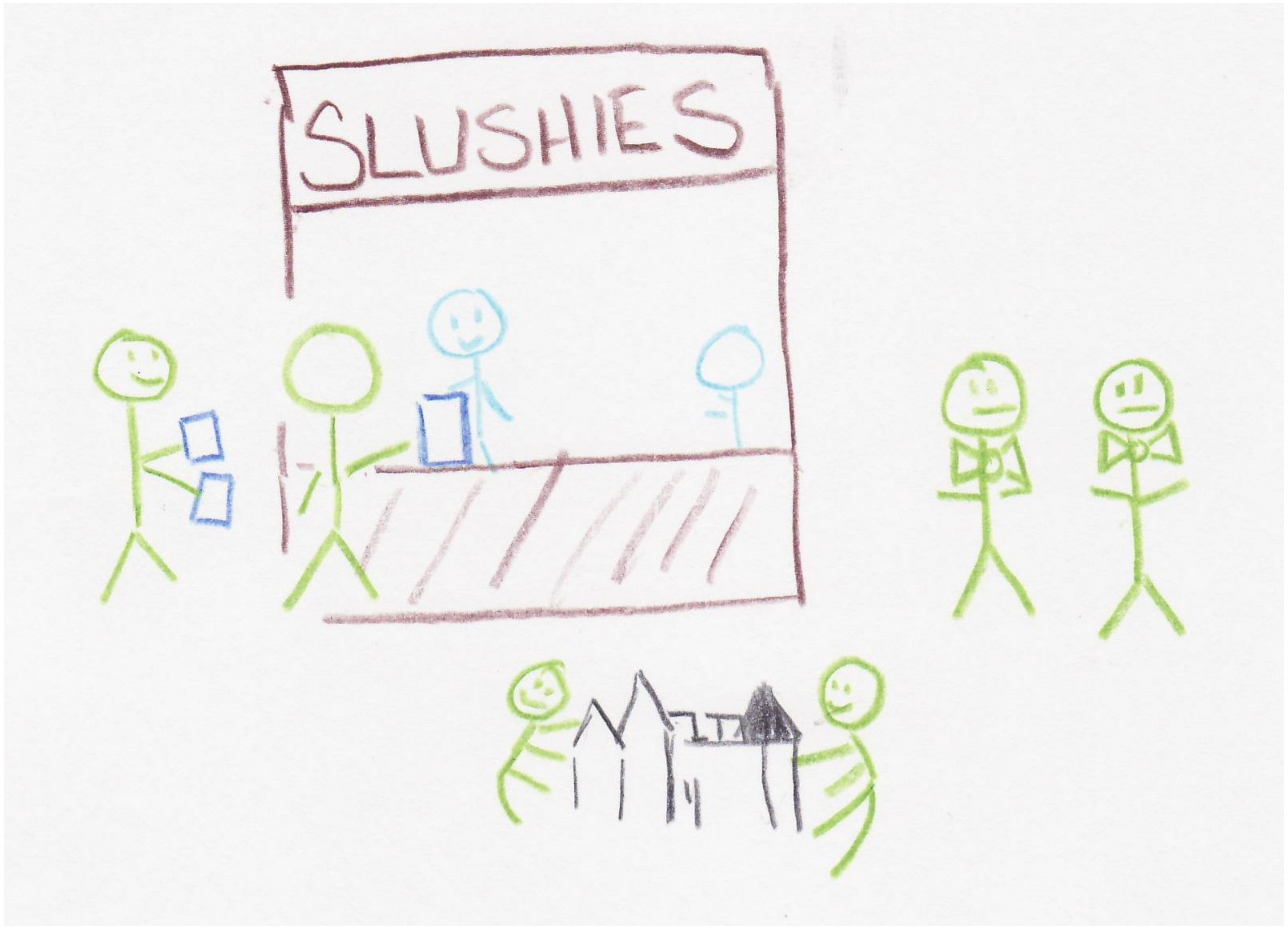


Dad is sometimes very generous. “Johnny would like one orange and one purple slushie.”



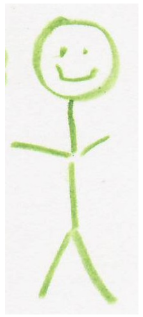
If both slushies are done at the same time and both butlers are available then Johnny gets two slushies at once. This confuses Johnny and causes brain freeze.

Susie is smarter and doesn't mind both slushies at one time.



But most often the dads are making requests to the Owner, the Assistant is monitoring the table, the kids are building sandcastles and
the butlers are waiting.

Credits



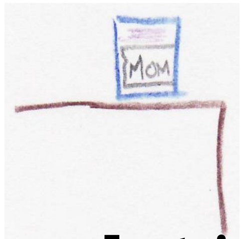
Initiator
(Dad)

Asynchronous
Operation Processor
(Owner)

Proactor
(Butler)

Asynchronous
Operation
(Blender Making
Slushies)

Asynchronous
Event Demultiplexer
(Assistant)



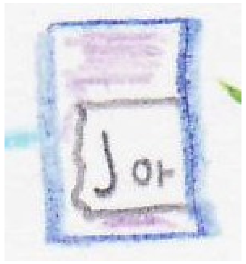
Completion
Event Queue
(Completion Table)

Completion Handler
(Johnny)

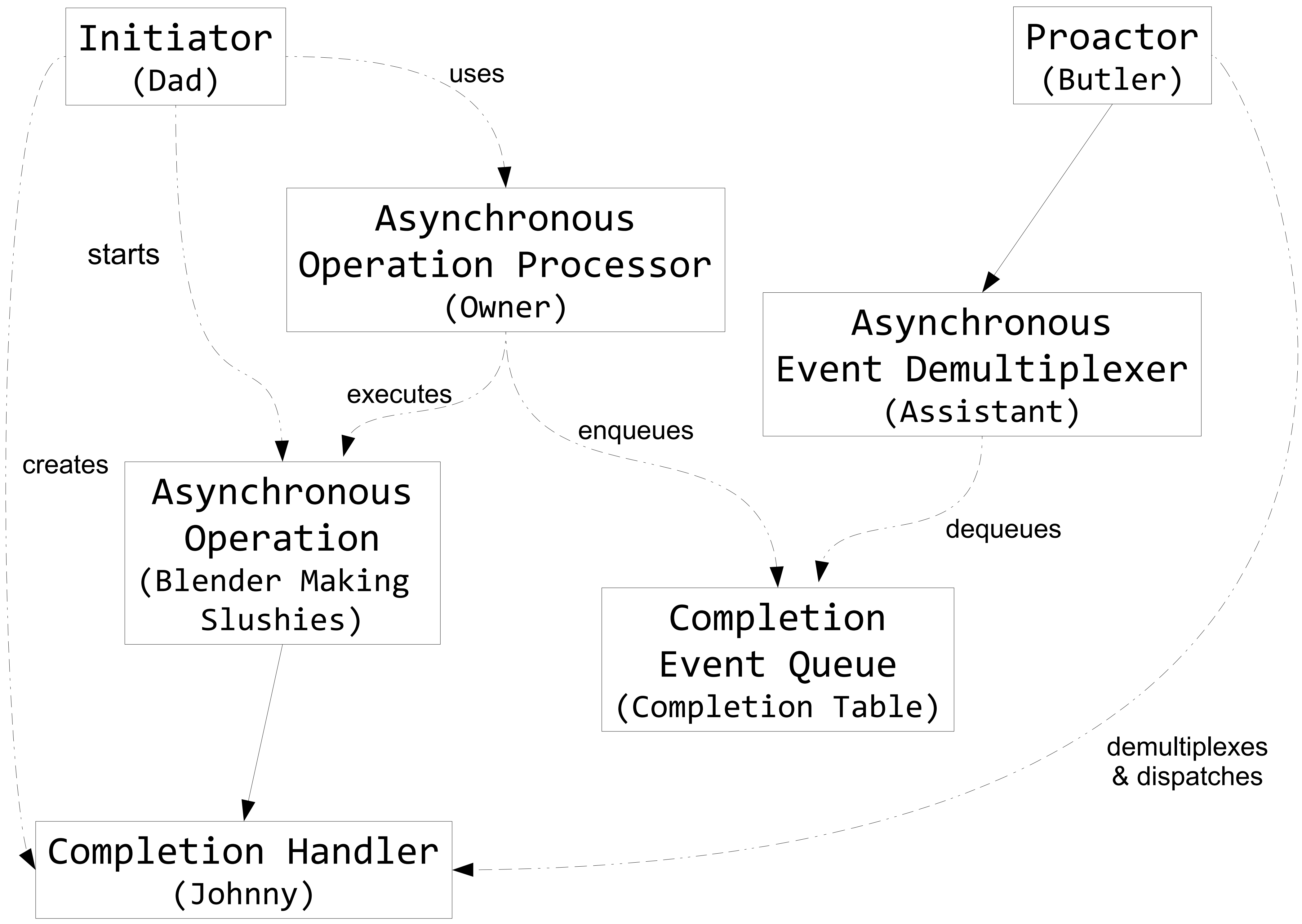
Additional Roles

Operating System
(Blender)

Memory to
be Filled
(Empty Cup)



Data in Memory
(Full Cup)


Initiator
(Dad)
uses
Proactor
(Butler)
starts
Asynchronous
Operation Processor
(Owner)
Asynchronous
Event Demultiplexer
(Assistant)
executes
enqueues
creates
Asynchronous
Operation
(Blender Making
Slushies)
dequeues
Completion
Event Queue
(Completion Table)
demultiplexes
& dispatches
Completion Handler
(Johnny)

Some Lessons

- All threads of activity in the Slushie Shack stayed in the Slushie Shack
- The Butler delivered the results to the completion handler
- The Butler (handler thread) was supplied by the family (application)
- The cup (memory) was supplied and owned by the family (application)

More Lessons

- Not all handlers (Johnny) liked having multiple results delivered at the same time
- Some handlers (Susie) didn't care if they had multiple results delivered at once
- Don't leave the beach (scope) when a slushie is being made for you
- A few handler threads (butlers) can service many completion routines

A Proactor Story

Outline

ciere.com

Simple Timer

```
int main()
{
   asio::io_service service;

   asio::deadline_timer timer(service, posix_time::seconds(5));

   timer.async_wait([](auto ... vn)
                    {
                       std::cout << system_clock::now()
                                 << " : timer expired.\n";
                    }
     );

   std::cout << system_clock::now() << " : calling run\n";

   service.run();

   std::cout << system_clock::now() << " : done.\n";
}
```

Simple Timer

Output

```
Tue Sep 20 22:54:51 2016 :  calling io_service::run
Tue Sep 20 22:54:56 2016 :  timer expired.
Tue Sep 20 22:54:56 2016 :  done.
```

```cpp
int main()
{
   asio::io_service service;

   asio::deadline_timer timer(service, posix_time::seconds(5));

   timer.async_wait([](auto ... vn)
                    {
                       std::cout << system_clock::now()
                                 << " : timer expired.\n";
                    }
      );

   std::cout << system_clock::now() << " : calling run\n";

   service.run();

   std::cout << system_clock::now() << " : done.\n";
}
```

Timer with `std::thread`

```
void timer_expired(std::string id)
{
   std::cout << now_time << " " << id << " enter.\n";
   std::this_thread::sleep_for(std::chrono::seconds(3));
   std::cout << now_time << " " << id << " leave.\n";
}



int main()
{
   asio::io_service service;

   asio::deadline_timer timer1(service, posix_time::seconds(5));
   asio::deadline_timer timer2(service, posix_time::seconds(5));

   timer1.async_wait([](auto ... vn){ timer_expired("timer1"); });
   timer2.async_wait([](auto ... vn){ timer_expired("timer2"); });

   std::thread butler( [&](){service.run();} );

   butler.join();

   std::cout << "done." << std::endl;
}
```

Timer with std::thread

```
void timer_expired(std::string id)
{
   std::cout << now_time << " " << id << " enter.\n";
   std::this_thread::sleep_for(std::chrono::seconds(3));
   std::cout << now_time << " " << id << " leave.\n";
}



int main()
{
   asio::io_service service;

   asio::deadline_timer timer1(service, posix_time::seconds(5));
   asio::deadline_timer timer2(service, posix_time::seconds(5));

   timer1.async_wait([](auto ... vn){ timer_expired("timer1"); });
   timer2.async_wait([](auto ... vn){ timer_expired("timer2"); });

   std::thread butler( [&](){service.run();} );

   butler.join();

   std::cout << "done." << std::endl;
}
```

Timer with std::thread

Output

```
Tue Sep 20 23:09:37 2016 timer1 enter.
Tue Sep 20 23:09:40 2016 timer1 leave.
Tue Sep 20 23:09:40 2016 timer2 enter.
Tue Sep 20 23:09:43 2016 timer2 leave.
done.
```

The butler can only deliver one slushie at a time.

```
void timer_expired(std::string id)
{
   std::cout << now_time << " " << id << " enter.\n";
   std::this_thread::sleep_for(std::chrono::seconds(3));
   std::cout << now_time << " " << id << " leave.\n";
}

int main()
{
   asio::io_service service;

   asio::deadline_timer timer1(service, posix_time::seconds(5));
   asio::deadline_timer timer2(service, posix_time::seconds(5));
```

Timer with two `std::thread` objects

```
void timer_expired(std::string id)
{
   std::cout << now_time << " " << id << " enter.\n";
   std::this_thread::sleep_for(std::chrono::seconds(3));
   std::cout << now_time << " " << id << " leave.\n";
}



int main()
{
   asio::io_service service;

   asio::deadline_timer timer1(service, posix_time::seconds(5));
   asio::deadline_timer timer2(service, posix_time::seconds(5));

   timer1.async_wait([](auto ... vn){ timer_expired("timer1"); });
   timer2.async_wait([](auto ... vn){ timer_expired("timer2"); });

   std::thread ta( [&](){service.run();} );
   std::thread tb( [&](){service.run();} );

   ta.join();
   tb.join();
   std::cout << "done." << std::endl;
}
```

Timer with two `std::thread` objects

```cpp
void timer_expired(std::string id)
{
   std::cout << now_time << " " << id << " enter.\n";
   std::this_thread::sleep_for(std::chrono::seconds(3));
   std::cout << now_time << " " << id << " leave.\n";
}


int main()
{
   asio::io_service service;

   asio::deadline_timer timer1(service, posix_time::seconds(5));
   asio::deadline_timer timer2(service, posix_time::seconds(5));

   timer1.async_wait([](auto ... vn){ timer_expired("timer1"); });
   timer2.async_wait([](auto ... vn){ timer_expired("timer2"); });

   std::thread ta( [&](){service.run();} );
   std::thread tb( [&](){service.run();} );

   ta.join();
   tb.join();
   std::cout << "done." << std::endl;
}
```

Timer with two `std::thread` objects

Output

```
TTuuee SSeepp 2200 2233::2211::2233 22001166

ttiimmeerr21 eenntteerr..

TTuuee SSeepp 2200 2233::2211::2266 22001166

ttiimmeerr12 lleeaavvee..

done.
```

```
void timer_expired(std::string id)
{
   std::cout << now_time << " " << id << " enter.\n";
   std::this_thread::sleep_for(std::chrono::seconds(3));
   std::cout << now_time << " " << id << " leave.\n";
}


int main()
{
   asio::io_service service;

   asio::deadline_timer timer1(service, posix_time::seconds(5));
   asio::deadline_timer timer2(service, posix_time::seconds(5));
```

Posting Work

Equivalent to the Owner placing items directly on the completion table.

```
int main()
{
   asio::io_service service;

   service.post([]{ std::cout << "eat\n"; });
   service.post([]{ std::cout << "drink\n"; });
   service.post([]{ std::cout << "and be merry!\n"; });

   std::thread butler([&]{ service.run(); });

   butler.join();

   std::cout << "done." << std::endl;
}
```

Posting Work

Output

```
eat
drink
and be merry!
done.
```

```cpp
int main()
{
   asio::io_service service;

   service.post([]{ std::cout << "eat\n"; });
   service.post([]{ std::cout << "drink\n"; });
   service.post([]{ std::cout << "and be merry!\n"; });

   std::thread butler([&]{ service.run(); });

   butler.join();

   std::cout << "done." << std::endl;
}
```

What if Johnny Can’t Handle Two Slushies

```
void timer_expired(std::string id)
{
   std::cout << now_time << " " << id << " enter.\n";
   std::this_thread::sleep_for(std::chrono::seconds(3));
   std::cout << now_time << " " << id << " leave.\n";
}


int main()
{
   asio::io_service service;

   asio::deadline_timer timer1(service, posix_time::seconds(5));
   asio::deadline_timer timer2(service, posix_time::seconds(5));

   timer1.async_wait([](auto ... vn){ timer_expired("timer1"); });
   timer2.async_wait([](auto ... vn){ timer_expired("timer2"); });

   std::thread ta( [&](){service.run();} );
   std::thread tb( [&](){service.run();} );

   ta.join();
   tb.join();
   std::cout << "done." << std::endl;
}
```

What if Johnny Can't Handle Two Slushies

Output

```
TTuuee SSeepp 2200 2233::2211::2233 22001166

ttiimmeerr21 eennetteerr..

TTuuee SSeepp 2200 2233::2211::2266 22001166

ttiimmeerr12 lleeaavvee..

done.
```

```
void timer_expired(std::string id)
{
   std::cout << now_time << " " << id << " enter.\n";
   std::this_thread::sleep_for(std::chrono::seconds(3));
   std::cout << now_time << " " << id << " leave.\n";
}

int main()
{
   asio::io_service service;

   asio::deadline_timer timer1(service, posix_time::seconds(5));
   asio::deadline_timer timer2(service, posix_time::seconds(5));
```

The io_service::strand

```
void timer_expired(std::string id)
{
   std::cout << now_time << " " << id << " enter.\n";
   std::this_thread::sleep_for(std::chrono::seconds(3));
   std::cout << now_time << " " << id << " leave.\n";
}
int main()
{
   asio::io_service service;
   asio::io_service::strand strand(service);

   asio::deadline_timer timer1(service, posix_time::seconds(5));
   asio::deadline_timer timer2(service, posix_time::seconds(5));

   timer1.async_wait(
        strand.wrap( [](auto ... vn){ timer_expired("timer1"); } )
      );
   timer2.async_wait(
        strand.wrap( [](auto ... vn){ timer_expired("timer2"); } )
      );

   std::thread ta( [&](){service.run();} );
   std::thread tb( [&](){service.run();} );

   ta.join(); tb.join();
   std::cout << "done.\n";
```

The `io_service::strand`

```
void timer_expired(std::string id)
{
   std::cout << now_time << " " << id << " enter.\n";
   std::this_thread::sleep_for(std::chrono::seconds(3));
   std::cout << now_time << " " << id << " leave.\n";
}
int main()
{
   asio::io_service service;
   asio::io_service::strand strand(service);

   asio::deadline_timer timer1(service, posix_time::seconds(5));
   asio::deadline_timer timer2(service, posix_time::seconds(5));

   timer1.async_wait(
        strand.wrap( [](auto ... vn){ timer_expired("timer1"); } )
     );
   timer2.async_wait(
        strand.wrap( [](auto ... vn){ timer_expired("timer2"); } )
     );

   std::thread ta( [&](){service.run();} );
   std::thread tb( [&](){service.run();} );

   ta.join(); tb.join();
   std::cout << "done.\n";
```

The io_service::strand

Output

```
Tue Sep 20 23:43:49 2016 timer1 enter.
Tue Sep 20 23:43:52 2016 timer1 leave.
Tue Sep 20 23:43:52 2016 timer2 enter.
Tue Sep 20 23:43:55 2016 timer2 leave.
done.
```

```
void timer_expired(std::string id)
{
   std::cout << now_time << " " << id << " enter.\n";
   std::this_thread::sleep_for(std::chrono::seconds(3));
   std::cout << now_time << " " << id << " leave.\n";
}
int main()
{
   asio::io_service service;
   asio::io_service::strand strand(service);

   asio::deadline_timer timer1(service, posix_time::seconds(5));
   asio::deadline_timer timer2(service, posix_time::seconds(5));

   timer1.async_wait(
        strand.wrap( [](auto ... vn){ timer_expired("timer1"); } )
      );
```

More io_service::strand

```
void timer_expired(std::string id)
{
   std::cout << now_time << " " << id << " enter.\n";
   std::this_thread::sleep_for(std::chrono::seconds(3));
   std::cout << now_time << " " << id << " leave.\n";
}

int main()
{
   asio::io_service service;
   asio::io_service::strand strand(service);

   asio::deadline_timer timer1(service, posix_time::seconds(5));
   asio::deadline_timer timer2(service, posix_time::seconds(5));
   asio::deadline_timer timer3(service, posix_time::seconds(6));

   timer1.async_wait(
        strand.wrap( [](auto ... vn){ timer_expired("timer1"); } ));
   timer2.async_wait(
        strand.wrap( [](auto ... vn){ timer_expired("timer2"); } ));

   timer3.async_wait( [](auto ... vn){ timer_expired("timer3"); } );

   std::thread ta([&](){service.run();}),
   std::thread tb( [&](){service.run();} );
   ta.join(); tb.join();
   std::cout << "done.\n";
}
```

More io_service::strand

Output

```
Wed Sep 21 00:03:05 2016 timer1 enter.
Wed Sep 21 00:03:06 2016 timer3 enter.
Wed Sep 21 00:03:08 2016 timer1 leave.
Wed Sep 21 00:03:08 2016 timer2 enter.
Wed Sep 21 00:03:09 2016 timer3 leave.
Wed Sep 21 00:03:11 2016 timer2 leave.
done.
```

```
void timer_expired(std::string id)
{
   std::cout << now_time << " " << id << " enter.\n";
   std::this_thread::sleep_for(std::chrono::seconds(3));
   std::cout << now_time << " " << id << " leave.\n";
}
int main()
{
   asio::io_service service;
   asio::io_service::strand strand(service);

   asio::deadline_timer timer1(service, posix_time::seconds(5));
   asio::deadline_timer timer2(service, posix_time::seconds(5));
   asio::deadline_timer timer3(service, posix_time::seconds(6));

   timer1.async_wait(
```

Part II

Communications

ciere.com

Outline

Buffers

Asio deals with memory using *buffers*.

```
using mutable_buffer = tuple<void*, std::size_t>;
using const_buffer = tuple<const void*, std::size_t>;
```

mutable_buffer $\rightarrow$ const_buffer

Asio supports scatter/gather when buffers are stored in containers.

Buffers do not own the underlying data!

ciere.com

Buffers

Asio deals with memory using *buffers*.

```
using mutable_buffer = tuple<void*, std::size_t>;
using const_buffer = tuple<const void*, std::size_t>;
```

mutable_buffer $\rightarrow$ const_buffer

Asio supports scatter/gather when buffers are stored in containers.

Buffers do not own the underlying data!

Buffers

Asio deals with memory using *buffers*.

```
class mutable_buffer;
class const_buffer;
```

mutable_buffer $\rightarrow$ const_buffer

Asio supports scatter/gather when buffers are stored in containers.

Buffers do not own the underlying data!

Buffers

Asio deals with memory using *buffers*.

```
class mutable_buffer;
class const_buffer;
```

mutable_buffer $\rightarrow$ const_buffer

Asio supports scatter/gather when buffers are stored in containers.

Buffers do not own the underlying data!

ciere.com

Buffers

Asio deals with memory using *buffers*.

```
class mutable_buffer;
class const_buffer;
```

mutable_buffer $\rightarrow$ const_buffer

Asio supports scatter/gather when buffers are stored in containers.

Buffers do not own the underlying data!

ciere.com

Buffers - Continued

It is easy to get an Asio buffer.

use

```
boost::asio::buffer(...)
```

```
socket_.send( asio::buffer(data, size) );
```

```
std::string personal_message( "dinner time!" );
socket_.send( asio::buffer(personal_message) );
```

```
std::array<uint_8,4> code = {0xde, 0xad, 0xbe, 0xef};
socket_.send( asio::buffer(code) );
```

Buffers - Scatter-Gather

```
std::array<uint_8,4> head = {0xba, 0xbe, 0xfa, 0xce};
std::string msg( "CppCon Rocks!" );
std::vector<uint8_t> data(256);

std::vector<asio::const_buffer> bufs{ asio::buffer(head)
                                    , asio::buffer(msg)
                                    , asio::buffer(data) };

socket_.send(bufs);
```

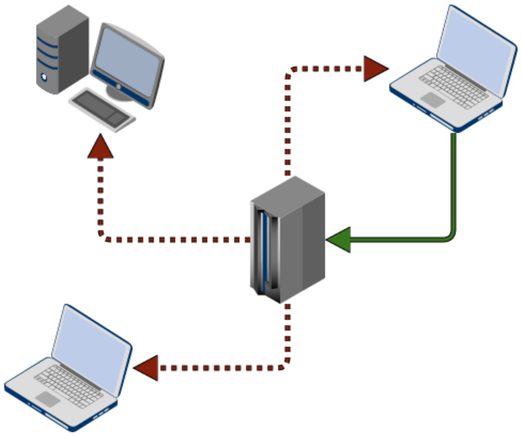
Part III

Server

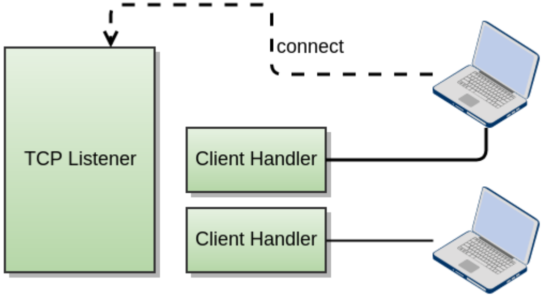
Outline

ciere.com

The Chat Server

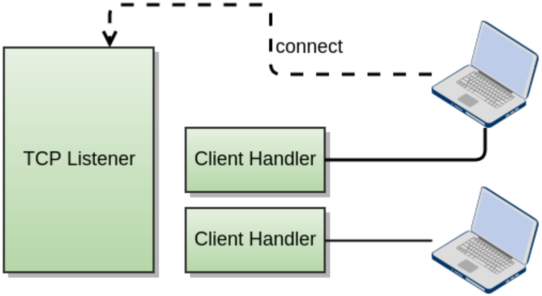


Generic Listener



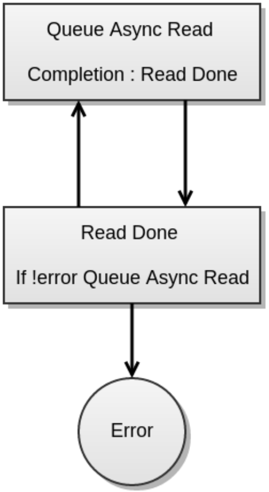
Who owns the Client Handler?

Generic Listener



Who owns the Client Handler?

Chaining Completion Handlers



Outline

ciere.com

main

```
asio_generic_server<chat_handler> server;
server.start_server(8888);
```

Generic Server

```
template <typename ConnectionHandler>
class asio_generic_server
{
   using shared_handler_t = std::shared_ptr<ConnectionHandler>;

public:

   asio_generic_server(int thread_count=1)
      : thread_count_(thread_count)
      , acceptor_(io_service_)
   {}

   void start_server(uint16_t port)
   {

   }

private:

   void handle_new_connection( shared_handler_t handler
                             , system::error_code const & error )
   {

   }

   int thread_count_;
   std::vector<std::thread> thread_pool_;
   asio::io_service io_service_;
   asio::ip::tcp::acceptor acceptor_;

};
```

Generic Server

```
template <typename ConnectionHandler>
class asio_generic_server
{
   using shared_handler_t = std::shared_ptr<ConnectionHandler>;
public:
   asio_generic_server(int thread_count=1)
      : thread_count_(thread_count)
      , acceptor_(io_service_)
   {}

   void start_server(uint16_t port)
   {

   }

private:
   void handle_new_connection( shared_handler_t handler
                             , system::error_code const & error )
   {

   }

   int thread_count_;
   std::vector<std::thread> thread_pool_;
   asio::io_service io_service_;
   asio::ip::tcp::acceptor acceptor_;
};
```

Generic Server

```
template <typename ConnectionHandler>
class asio_generic_server
{
   using shared_handler_t = std::shared_ptr<ConnectionHandler>;

public:

   asio_generic_server(int thread_count=1)
      : thread_count_(thread_count)
      , acceptor_(io_service_)
   {}

   void start_server(uint16_t port)
   {

   }

private:

   void handle_new_connection( shared_handler_t handler
                             , system::error_code const & error )
   {

   }

   int thread_count_;
   std::vector<std::thread> thread_pool_;
   asio::io_service io_service_;
   asio::ip::tcp::acceptor acceptor_;

};
```

Generic Server

```
template <typename ConnectionHandler>
class asio_generic_server
{
   using shared_handler_t = std::shared_ptr<ConnectionHandler>;
public:
   asio_generic_server(int thread_count=1)
      : thread_count_(thread_count)
      , acceptor_(io_service_)
   {}
   void start_server(uint16_t port)
   {

   }
private:
   void handle_new_connection( shared_handler_t handler
                             , system::error_code const & error )
   {

   }

   int thread_count_;
   std::vector<std::thread> thread_pool_;
   asio::io_service io_service_;
   asio::ip::tcp::acceptor acceptor_;
};
```

Generic Server

```
template <typename ConnectionHandler>
class asio_generic_server
{
   using shared_handler_t = std::shared_ptr<ConnectionHandler>;
public:
   asio_generic_server(int thread_count=1)
      : thread_count_(thread_count)
      , acceptor_(io_service_)
   {}
   void start_server(uint16_t port)
   {

   }
private:
   void handle_new_connection( shared_handler_t handler
                             , system::error_code const & error )
   {

   }
   int thread_count_;
   std::vector<std::thread> thread_pool_;
   asio::io_service io_service_;
   asio::ip::tcp::acceptor acceptor_;
};
```

Start Server

```
void start_server(uint16_t port)
{
   auto handler
      = std::make_shared<ConnectionHandler>(io_service_);

   // set up the acceptor to listen on the tcp port
   asio::ip::tcp::endpoint endpoint(asio::ip::tcp::v4(), port);
   acceptor_.open(endpoint.protocol());
   acceptor_.set_option(tcp::acceptor::reuse_address(true));
   acceptor_.bind(endpoint);
   acceptor_.listen();

   acceptor_.async_accept( handler->socket()
                         , [=](auto ec)
                           {
                              handle_new_connection(handler, ec);
                           }
      );

   // start pool of threads to process the asio events
   for(int i=0; i<thread_count_; ++i)
   {
      thread_pool_.emplace_back( [=]{io_service_.run();} );
   }
}
```

Start Server

```
void start_server(uint16_t port)
{
    auto handler
        = std::make_shared<ConnectionHandler>(io_service_);

    // set up the acceptor to listen on the tcp port
    asio::ip::tcp::endpoint endpoint(asio::ip::tcp::v4(), port);
    acceptor_.open(endpoint.protocol());
    acceptor_.set_option(tcp::acceptor::reuse_address(true));
    acceptor_.bind(endpoint);
    acceptor_.listen();

    acceptor_.async_accept( handler->socket()
                          , [=](auto ec)
                            {
                               handle_new_connection(handler, ec);
                            }
        );

    // start pool of threads to process the asio events
    for(int i=0; i<thread_count_; ++i)
    {
        thread_pool_.emplace_back( [=]{io_service_.run();} );
    }

}
```

Start Server

```
void start_server(uint16_t port)
{
   auto handler
      = std::make_shared<ConnectionHandler>(io_service_);

   // set up the acceptor to listen on the tcp port
   asio::ip::tcp::endpoint endpoint(asio::ip::tcp::v4(), port);
   acceptor_.open(endpoint.protocol());
   acceptor_.set_option(tcp::acceptor::reuse_address(true));
   acceptor_.bind(endpoint);
   acceptor_.listen();

   acceptor_.async_accept( handler->socket()
                         , [=](auto ec)
                           {
                              handle_new_connection(handler, ec);
                           }
      );

   // start pool of threads to process the asio events
   for(int i=0; i<thread_count_; ++i)
   {
      thread_pool_.emplace_back( [=]{io_service_.run();} );
   }

}
```

Start Server

```
void start_server(uint16_t port)
{
   auto handler
      = std::make_shared<ConnectionHandler>(io_service_);

   // set up the acceptor to listen on the tcp port
   asio::ip::tcp::endpoint endpoint(asio::ip::tcp::v4(), port);
   acceptor_.open(endpoint.protocol());
   acceptor_.set_option(tcp::acceptor::reuse_address(true));
   acceptor_.bind(endpoint);
   acceptor_.listen();

   acceptor_.async_accept( handler->socket()
                         , [=](auto ec)
                           {
                              handle_new_connection(handler, ec);
                           }
      );

   // start pool of threads to process the asio events
   for(int i=0; i<thread_count_; ++i)
   {
      thread_pool_.emplace_back( [=]{io_service_.run();} );
   }
}
```

Start Server

```
void start_server(uint16_t port)
{
    auto handler
        = std::make_shared<ConnectionHandler>(io_service_);

    // set up the acceptor to listen on the tcp port
    asio::ip::tcp::endpoint endpoint(asio::ip::tcp::v4(), port);
    acceptor_.open(endpoint.protocol());
    acceptor_.set_option(tcp::acceptor::reuse_address(true));
    acceptor_.bind(endpoint);
    acceptor_.listen();

    acceptor_.async_accept( handler->socket()
                          , [=](auto ec)
                            {
                               handle_new_connection(handler, ec);
                            }
        );

    // start pool of threads to process the asio events
    for(int i=0; i<thread_count_; ++i)
    {
        thread_pool_.emplace_back( [=]{io_service_.run();} );
    }
}
```

Handle Connection

```
void handle_new_connection( shared_handler_t handler
                          , system::error_code const & error )
{
   if(error){ return; }

   handler->start();

   auto new_handler
      = std::make_shared<ConnectionHandler>(io_service_);

   acceptor_.async_accept( new_handler->socket()
                         , [=](auto ec)
                           {
                              handle_new_connection( new_handler
                                                   , ec);
                           }
      );
}
```

Handle Connection

```
void handle_new_connection( shared_handler_t handler
                          , system::error_code const & error )
{
   if(error){ return; }

   handler->start();

   auto new_handler
      = std::make_shared<ConnectionHandler>(io_service_);

   acceptor_.async_accept( new_handler->socket()
                         , [=](auto ec)
                           {
                              handle_new_connection( new_handler
                                                   , ec);
                           }
      );

}
```

Handle Connection

```
void handle_new_connection( shared_handler_t handler
                          , system::error_code const & error )
{
   if(error){ return; }

   handler->start();

   auto new_handler
      = std::make_shared<ConnectionHandler>(io_service_);

   acceptor_.async_accept( new_handler->socket()
                         , [=](auto ec)
                           {
                              handle_new_connection( new_handler
                                                   , ec);
                           }
      );

}
```

Handle Connection

```
void handle_new_connection( shared_handler_t handler
                          , system::error_code const & error )
{
   if(error){ return; }

   handler->start();

   auto new_handler
      = std::make_shared<ConnectionHandler>(io_service_);

   acceptor_.async_accept( new_handler->socket()
                         , [=](auto ec)
                           {
                              handle_new_connection( new_handler
                                                   , ec);
                           }
      );

}
```

Handle Connection

```
void handle_new_connection( shared_handler_t handler
                          , system::error_code const & error )
{
   if(error){ return; }

   handler->start();

   auto new_handler
      = std::make_shared<ConnectionHandler>(io_service_);

   acceptor_.async_accept( new_handler->socket()
                         , [=](auto ec)
                           {
                              handle_new_connection( new_handler
                                                   , ec);
                           }
      );
}
```

Chat Handler

```
class chat_handler
    : public std::enable_shared_from_this<chat_handler>
{
public:
    chat_handler(asio::io_service& service)
        : service_(service)
        , socket_(service)
        , write_strand_(service)
    {}

    boost::asio::ip::tcp::socket& socket()
    {
        return socket_;
    }

    void start()
    {
        read_packet();
    }

private:
    asio::io_service& service_;
    asio::ip::tcp::socket socket_;
    asio::io_service::strand write_strand_;
    asio::streambuf in_packet_;
    std::deque<std::string> send_packet_queue;
};
```

Chat Handler

```
class chat_handler
    : public std::enable_shared_from_this<chat_handler>
{
public:
    chat_handler(asio::io_service& service)
        : service_(service)
        , socket_(service)
        , write_strand_(service)
    {}

    boost::asio::ip::tcp::socket& socket()
    {
        return socket_;
    }

    void start()
    {
        read_packet();
    }

private:
    asio::io_service& service_;
    asio::ip::tcp::socket socket_;
    asio::io_service::strand write_strand_;
    asio::streambuf in_packet_;
    std::deque<std::string> send_packet_queue;
};
```

Chat Handler

```
class chat_handler
   : public std::enable_shared_from_this<chat_handler>
{
public:
   chat_handler(asio::io_service& service)
      : service_(service)
      , socket_(service)
      , write_strand_(service)
   {}

   boost::asio::ip::tcp::socket& socket()
   {
      return socket_;
   }

   void start()
   {
      read_packet();
   }
private:
   asio::io_service& service_;
   asio::ip::tcp::socket socket_;
   asio::io_service::strand write_strand_;
   asio::streambuf in_packet_;
   std::deque<std::string> send_packet_queue;
};
```

Chat Handler

```
class chat_handler
    : public std::enable_shared_from_this<chat_handler>
{
public:
    chat_handler(asio::io_service& service)
        : service_(service)
        , socket_(service)
        , write_strand_(service)
    {}

    boost::asio::ip::tcp::socket& socket()
    {
        return socket_;
    }

    void start()
    {
        read_packet();
    }

private:
    asio::io_service& service_;
    asio::ip::tcp::socket socket_;
    asio::io_service::strand write_strand_;
    asio::streambuf in_packet_;
    std::deque<std::string> send_packet_queue;
};
```

Chat Handler

```
class chat_handler
    : public std::enable_shared_from_this<chat_handler>
{
public:
    chat_handler(asio::io_service& service)
        : service_(service)
        , socket_(service)
        , write_strand_(service)
    {}

    boost::asio::ip::tcp::socket& socket()
    {
        return socket_;
    }

    void start()
    {
        read_packet();
    }
private:
    asio::io_service& service_;
    asio::ip::tcp::socket socket_;
    asio::io_service::strand write_strand_;
    asio::streambuf in_packet_;
    std::deque<std::string> send_packet_queue;
};
```

Chat Handler - read

```
void read_packet()
{
   asio::async_read_until( socket_,
                           in_packet_,
                           '\0',
                           [me=shared_from_this()]
                           ( system::error_code const & ec
                           , std::size_t bytes_xfer)
                           {
                             me->read_packet_done(ec, bytes_xfer);
                           } );
}

void read_packet_done( system::error_code const & error
                     , std::size_t bytes_transferred )
{
   if(error){ return; }

   std::istream stream(&in_packet_);
   std::string packet_string;
   stream >> packet_string;

   // do something with it

   read_packet();
}
```

Chat Handler - read

```
void read_packet()
{
   asio::async_read_until( socket_,
                           in_packet_,
                           '\0',
                           [me=shared_from_this()]
                           ( system::error_code const & ec
                           , std::size_t bytes_xfer)
                           {
                              me->read_packet_done(ec, bytes_xfer);
                           } );
}


void read_packet_done( system::error_code const & error
                     , std::size_t bytes_transferred )
{
   if(error){ return; }

   std::istream stream(&in_packet_);
   std::string packet_string;
   stream >> packet_string;

   // do something with it

   read_packet();
}
```

Chat Handler - read

```
void read_packet()
{
   asio::async_read_until( socket_,
                           in_packet_,
                           '\0',
                           [me=shared_from_this()]
                           ( system::error_code const & ec
                           , std::size_t bytes_xfer)
                           {
                             me->read_packet_done(ec, bytes_xfer);
                           } );
}


void read_packet_done( system::error_code const & error
                     , std::size_t bytes_transferred )
{
   if(error){ return; }

   std::istream stream(&in_packet_);
   std::string packet_string;
   stream >> packet_string;

   // do something with it

   read_packet();
}
```

Chat Handler - read

```
void read_packet()
{
   asio::async_read_until( socket_,
                           in_packet_,
                           '\0',
                           [me=shared_from_this()]
                           ( system::error_code const & ec
                           , std::size_t bytes_xfer)
                           {
                             me->read_packet_done(ec, bytes_xfer);
                           } );
}


void read_packet_done( system::error_code const & error
                     , std::size_t bytes_transferred )
{
   if(error){ return; }

   std::istream stream(&in_packet_);
   std::string packet_string;
   stream >> packet_string;

   // do something with it

   read_packet();
}
```

Chat Handler - send

```
class chat_handler
   : public std::enable_shared_from_this<chat_handler>
{

public:

   void send(std::string msg)
   {
      service_.post( write_strand_.wrap( [me=shared_from_this()]()
                                         {
                                            me->queue_message(msg);
                                         } ));
   }

private:

   void queue_message(std::string message)
   {
      bool write_in_progress = !send_packet_queue.empty();
      send_packet_queue.push_back(std::move(message));

      if(!write_in_progress)
      {
         start_packet_send();
      }
   }

};
```

Chat Handler - send

```
class chat_handler
   : public std::enable_shared_from_this<chat_handler>
{
public:

   void send(std::string msg)
   {
      service_.post( write_strand_.wrap( [me=shared_from_this()]()
                                         {
                                            me->queue_message(msg);
                                         } ));
   }

private:

   void queue_message(std::string message)
   {
      bool write_in_progress = !send_packet_queue.empty();
      send_packet_queue.push_back(std::move(message));

      if(!write_in_progress)
      {
         start_packet_send();
      }
   }

};
```

Chat Handler - send

```
class chat_handler
    : public std::enable_shared_from_this<chat_handler>
{

public:

    void send(std::string msg)
    {
        service_.post( write_strand_.wrap( [me=shared_from_this()]()
                                           {
                                              me->queue_message(msg);
                                           } ));
    }

private:

    void queue_message(std::string message)
    {
        bool write_in_progress = !send_packet_queue.empty();
        send_packet_queue.push_back(std::move(message));

        if(!write_in_progress)
        {
            start_packet_send();
        }
    }

};
```

Chat Handler - send

```
void start_packet_send()
{
   send_packet_queue.front() += "\0";
   async_write( socket_
              , asio::buffer(send_packet_queue.front())
              , write_strand_.wrap( [me=shared_from_this()]
                                    ( system::error_code const & ec
                                    , std::size_t)
                                    {
                                      me->packet_send_done(ec);
                                    }
               ));
}


void packet_send_done(system::error_code const & error)
{
   if(!error)
   {
      send_packet_queue.pop_front();
      if(!send_packet_queue.empty()){ start_packet_send(); }
   }
}
```

Chat Handler - send

```
void start_packet_send()
{
   send_packet_queue.front() += "\0";
   async_write( socket_
              , asio::buffer(send_packet_queue.front())
              , write_strand_.wrap( [me=shared_from_this()]
                                    ( system::error_code const & ec
                                    , std::size_t)
                                    {
                                       me->packet_send_done(ec);
                                    }
                ));
}


void packet_send_done(system::error_code const & error)
{
   if(!error)
   {
      send_packet_queue.pop_front();
      if(!send_packet_queue.empty()){ start_packet_send(); }
   }
}
```

More...

- Layered design!
- Use as a processing queue
- Add your own services
- Combine with MSM and Spirit