Promises in C++: The Universal Glue for Asynchronous Programs

David Sankel ⟨dsankel@bloomberg.net⟩
Bloomberg
C++Now 2017

Code for this talk

https://github.com/camio/superpromise

(See c++11 branch for no concepts)

What's the problem?

Concurrently executing tasks with complex interdependencies

How is it solved?

Blocking threads

```
for(;;) {
    Request r = getRequest();
    std::thread t( [r]{
        Item i = lookupInDatabase(r.query);
        sendResponse(r.origin, i.data);
    });
    t.detach()
}
```

Blocking threads

- Threads are expensive and are required at every branch.
- Don't harmonize well with async libraries.
- Communication between threads is hard to get right.

State machines

```
class RequestHandler {
  RequestHandler(Request r) : m_request(r) {
    lookupInDatabase(r.query, [this](int errorCode, Item i) {
      handleReceivedItem(errorCode, i);
    });
  }
  void handleReceivedItem(int errorCode, Item i) {
    if(errorCode == 0)
      sendResponse(m_request.origin, i.data,
        [this](int errorCode) {
          handleSentResponse(errorCode);
        });
    else  // ...
  }
  void handleSentResponse(int errorCode) {
    //
```

State machines

- Fast
- Harmonize well with async libraries
- Tricky to follow execution path
- Error handling is clunky

Nested callbacks

```
service.listen( [](Request r) {
  lookupInDatabase(r.query, [r](int errorCode, Item i) {
    if(errorCode == 0)
      sendResponse(r.origin, i.data,
        [](int errorCode) {
          if(errorCode != 0)
           // ...
        });
    else  // ...
  });
});
```

Nested callbacks

- Flow followable
- Nesting gets deep

Other things

- Event systems

Large systems have scale issues

- Threads
- State machines
- Nested callbacks

The crippled C++ `std::future`

```
std::future<Item> lookupInDatabase(const std::string& query);
// ...

std::future<Item> itemF = lookupInDatabase(r.query);
Item i = itemF.get();
//...
```

`std::future`

- Exceptions for errors
- Still essentially a blocking interface
- `std::shared_future`, `std::future`, and `std::promise` interaction

dplp::Promise

```
service.listen( [](Request r) {
  dplp::Promise<Item> itemP = lookupInDatabase(r.query);
  return itemP.then([r](Item i) {
      return sendResponse(r.origin, i.data);
  });
});
```

`dplp::Promise`

- Concise
- Error handling code only when needed
- Simply one type (with one member function)
- Powerful

`dplp::Promise`'s fundamental operations

Constructor and `then`:

```
const dplp::Promise<int> five(
  [](auto fulfil, auto reject) { fulfill(5); } );
const dplp::Promise<int> six = five.then(
  [](int i) { return i+1; } );
```

`first` and `all`:

```
dplp::Promise<std::string> foo = /*...*/;
dplp::Promise<int> bar = /*...*/;

dplp::Promise<std::string, int> foobar = dplp::all(foo, bar);
dplp::Promise<std::variant<std::string, int>>
  foo_or_bar = dplp::first(foo, bar);
```

`dplp::Promise` constructor

```
dplp::Promise<std::string> foo( [](auto fulfil, auto reject) {
  //...
};
```

- `fulfil` takes in a `std::string`
- `reject` takes in a `std::exception_ptr`
- Only one of `fulfil` or `reject` may be called (at most once)
- `fulfil` and `reject` can be stored for later use
- The resolver function is executed immediately

`dplp::Promise` construction helpers

- `dplp::makeFulfilledPromise`. Create a promise in the fulfilled state.
- `dplp::makeRejectedPromise`. Create a promise in the rejected state.

```
dplp::Promise<int> five
  = dplp::makeFulfilledPromise(5);

dplp::Promise<int> reject
  = dplp::makeRejectedPromise<int>(
      std::make_exception_ptr(std::runtime_error("Boo")));
```

dplp::Promise's then

```
p.then(cf)
p.then(cf, ef)
```

```
dplp::Promise<std::string> stringP
  = dplp::makeFulfilledPromise(5).then(
      [](int i) { return std::to_string(i); } );
```

Special continuation return types

- `void`
- `dplp::Promise<T>`
- `std::tuple<T,U,V>`

`void` continuation → `dplp::Promise<>`

```
dplp::Promise<> p = dplp::makeFulfilledPromise(5).then( [](int i)
  std::cout << i << std::endl;
});
```

dplp::Promise<T> continuation → dplp::Promise<T>

```
dplp::Promise<int> p = dplp::makeFulfilledPromise().then( []() {
  return dplp::makeFulfilledPromise(3);
});
```

std::tuple<T,U,V>
continuation →
dplp::Promise<T,U,V>

```
dplp::Promise<int, int> p = dplp::makeFulfilledPromise(1, 2, 3)
  .then( [](int a, int b, int c) {
    return std::make_tuple(a, b);
  });
```

Asynchronous API Example

Client

```
class Client {
public:
  friend dplp::Promise<Client> connect(std::string serverAddress)

  dplp::Promise<int> lookupInt(std::string key);

  dplp::Promise<std::string> lookupString(std::string key);

  dplp::Promise<> setForwardingStatus(bool forward);
};
```

```
dplp::Promise<> baz = dplp::all(connect("foo"), connect("bar"))
  .then( [](Client foo, Client bar) {
    return foo.lookupInt("forwardSetting")
      .then( [=](int forwardSetting) {
        return bar.setForwardingStatus(bool(forwardSetting));
      });
  });
```

Server

```
class Server {
public:
  dplp::Promise<> listen();
    // Listen for an incoming connection and handle it. Return a
    // promise that is fulfilled when the handling is complete.
};
```

```
dplp::Promise<> listenUntilError(Server s) {
  return s.listen().then([s]{listenUntilError(s);});
}
```

Wrapping a callback library

```
class CallbackClient {
public:
  friend void callbackConnect(
    std::string serverAddress,
    std::function<void (std::error_code, CallbackClient)> cb );
  void lookupInt(
    std::string key,
    std::function<void (std::error_code, int)> cb);
  void lookupString(
    std::string key,
    std::function<void (std::error_code, std::string)> cb);
  void setForwardingStatus(
    bool forward,
    std::function<void (std::error_code)> cb);
};
```

Client

```
class Client {
  //...
private:
  Client(CallbackClient);
  CallbackClient m_callbackClient;
};
```

connect

```
dplp::Promise<Client> connect(std::string serverAddress) {
  return dplp::Promise<Client>([&](auto fulfill, auto reject) {
    callbackConnect(
      serverAddress,
      [=](std::error_code error, CallbackClient cc) {
        if(!error) {
          fulfill(Client(cc));
        } else {
          reject(std::make_exception(std::system_error(error)));
        }
      });
  });
}
```

lookupInt

```
dplp::Promise<int> Client::lookupInt(std::string key) {
  return dplp::Promise<int>([&](auto fulfill, auto reject) {
    m_callbackClient.lookupInt(
      key,
      [=](std::error_code error, int i) {
        if(!error) {
          fulfill(i);
        } else {
          reject(std::make_exception(std::system_error(error)));
        }
      });
  });
}
```

setForwardingStatus

```
dplp::Promise<> Client::setForwardingStatus(bool forward) {
  return dplp::Promise<>([&](auto fulfill, auto reject) {
    m_callbackClient.setForwardingStatus(
      key,
      [=](std::error_code error) {
        if(!error) {
          fulfill();
        } else {
          reject(std::make_exception(std::system_error(error)));
        }
      });
  });
}
```

Wrapping single callback systems

Complex example: zookeeper's `zoo_aget_children2`

```
// Lookup the children of 'path'. Call 'completion' with
// the list of contents and 'Stat' of 'path'. Call 'watcher'
// in the event the contents change.
int zoo_awget_children2(
  zhandle_t *zh,
  string path,
  function<void ()> watcher,
  function<void (int rc, vector<string>, Stat)> completion );
```

Wrapped

```
dplp::Promise<vector<string>,
              Stat,
              dplp::Promise<> >
getChildren(zhandle_t *zh, string path) const;
```

ASIO wrapper

```
dplp::Promise<> echo(apltcp::channel c) {
  return c.readUntil('\n')
      .then([c](const std::string &msg) {
          return c.send(msg);
      });
}

//...
apltcp::server s(context, listenAddress);
s.listen().then(echo);
```

Comparison to other promise-like libraries

- One type instead of several (`shared_future`, `promise`, `future`).
- No blocking functions (like `get`). Just `then`.
- Resolver-based construction.
- `Promise<>` instead of `promise<void>`.
- Multi-type promises (e.g. `Promise<T1, T2, ...>`).
- Direct Continuations.

Direct continuations: benefits and drawbacks

```
foo.then( // dplp style
  [](int i) { /* ... */ },
  [](std::exception_ptr e) {
    try {
      std::rethrow_exception(e);
    } catch ( /*...*/ ) { /*...*/ }
  });

foo.then( // Boost style
  [](future<int> i) {
    try { /* ... i.get() ... */ }
    catch ( /*...*/ ) { /*...*/ }
  });
```

Comparison to fibers

```
class Client {
public:
  friend Client connect(std::string serverAddress);
  int lookupInt(std::string key);
  std::string lookupString(std::string key);
  void setForwardingStatus(bool forward);
};
```

Comparison to fibers

```
dplp::Promise<> baz = dplp::all(connect("foo"), connect("bar"))
  .then( [](Client foo, Client bar) {
    return foo.lookupInt("forwardSetting")
      .then( [=](int forwardSetting) {
        return bar.setForwardingStatus(bool(forwardSetting));
      });
  });

std::function<void ()> baz = [] {
  auto [foo, bar] = fiber_all([]{ connect("foo");}, []{ connect("
  int forwardSetting = foo.lookupInt("forwardSetting");
  bar.setForwardingStatus(bool(forwardSetting));
};
```

Are Promises Enough?

PList

```
template<typename T>
struct PListImp {
  std::optional<std::tuple<T, dplp::Promise<PList> > > data;

  template<typename U>
  U expand(
    std::function<U ()> handleEnd,
    std::function<U (T, dplp::Promise<PList>)> handleFront );
};

template<typename T>
using PList<T> = dplp::Promise<PListImp<T>>;
```

```
PList<int> getResponses();

PList<Channel> listen();
```

Other Abstractions

- RxCpp (Reactive Programming)
- sfrp (Functional Reactive Programming)

Promises in C++: The Universal Glue for Asynchronous Programs

https://github.com/camio/superpromise

David Sankel ⟨dsankel@bloomberg.net⟩
Bloomberg
C++Now 2017