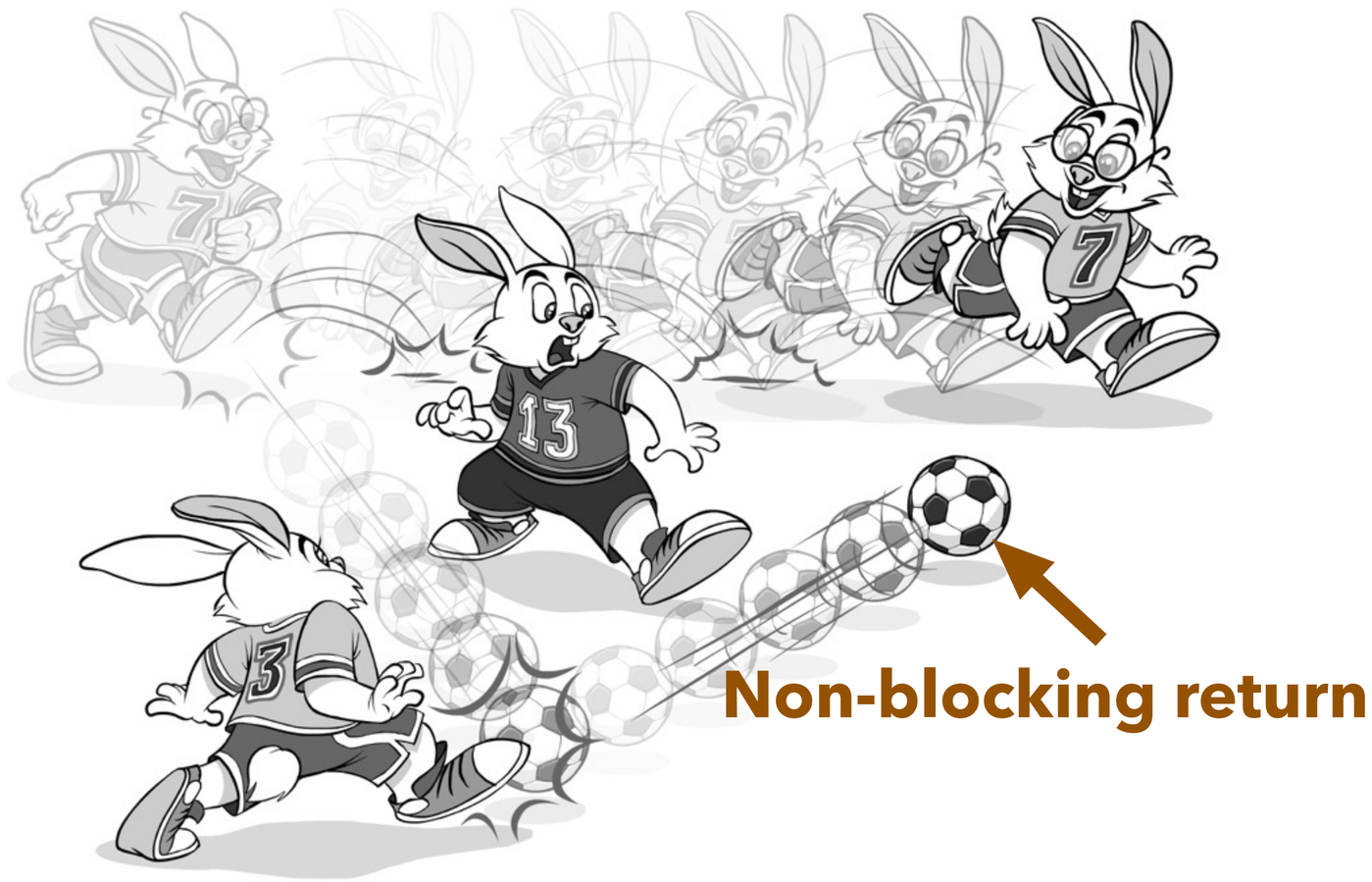
by 'No Bugs' Hare, nobugs@ithare.com



Eight Ways to Handle Non-blocking Returns in Message-passing Programs

from C++98 via C++11 to C++2a

The Guy to Blame

‘NO BUGS’ HARE

Part 0. Context

Message-Passing and (Re)Actors

- *Same-Thread Processing, no thread sync within*
- *Benefits*

Mostly non-blocking Processing

- *Non-blocking CAN be simpler than blocking*
- *Interactions between main program flow and return processing*

Do not communicate by sharing memory; instead, share memory by communicating.

– Effective Go

Implications:

- *single-thread processing*
- *no mutexes*
- *exchanging messages*

Message-Passing Allows For:

- *simpler programming*
    - *in particular, avoiding cognitive overload when trying to deal with both business logic and thread sync simultaneously*
- *determinism, which in turn allows for:*
    - *testability*
    - *production post-mortem analysis*
    - *replay-based regression testing*
- *better performance, scalability, and concurrency*
    - *no contention points*
    - *avoiding expensive thread context switches*
    - *better temporal locality*
    - *Shared-Nothing rulezz*

(Re)Actors:

- one way to implement message passing
- *a.k.a. Actors, Reactors, Event-Driven Programs, and ad-hoc Finite State Machines*
- *widely used*
  - *GUI, gamedev, HPC*
  - *from WM_* to Node.js*

While from now on we'll be speaking only about (Re)Actors - most of our findings are generalisable to more generic message passing.

Exception: allocator-related serialisation

Generic (Re)Actor

```
class GenericReactor {
  virtual void react(const Event& ev) = 0;
};
```

Infrastructure Code - Event Loop

```
GenericReactor* r =
  reactorFactory.createReactor(...);
while(true) { //event loop
  Event ev = get_event();
    //from select(), libuv, ...
  r->react(ev);
}
```

Specific (Re)Actor

```
class SpecificReactor :public GenericReactor {
  void react(const Event& ev) override;
};
```

Non-Blocking Code:

- Has a bad reputation because of perceived coding complexity
- However, we need to distinguish two very different scenarios:

1. We don't need to process anything while waiting for the result.
   - we're doing non-blocking processing ONLY for performance. Non-blocking code complexity indeed increases compared to blocking code.
2. We DO need to process events while waiting for the result.
   Example - waiting for the Internet.
   - Non-blocking code is ugly
   - Blocking code (which needs threads+sync) is even worse.

Mostly Non-Blocking Processing:

- non-blocking ONLY when we DO need to process intervening events while waiting
- blocking when we can postpone intervening events while waiting
  - example - local disk/DB accesses can often be made blocking without risking to stall forever.

It is INTERACTIONS between main control flow and processing of returned values which are of interest.

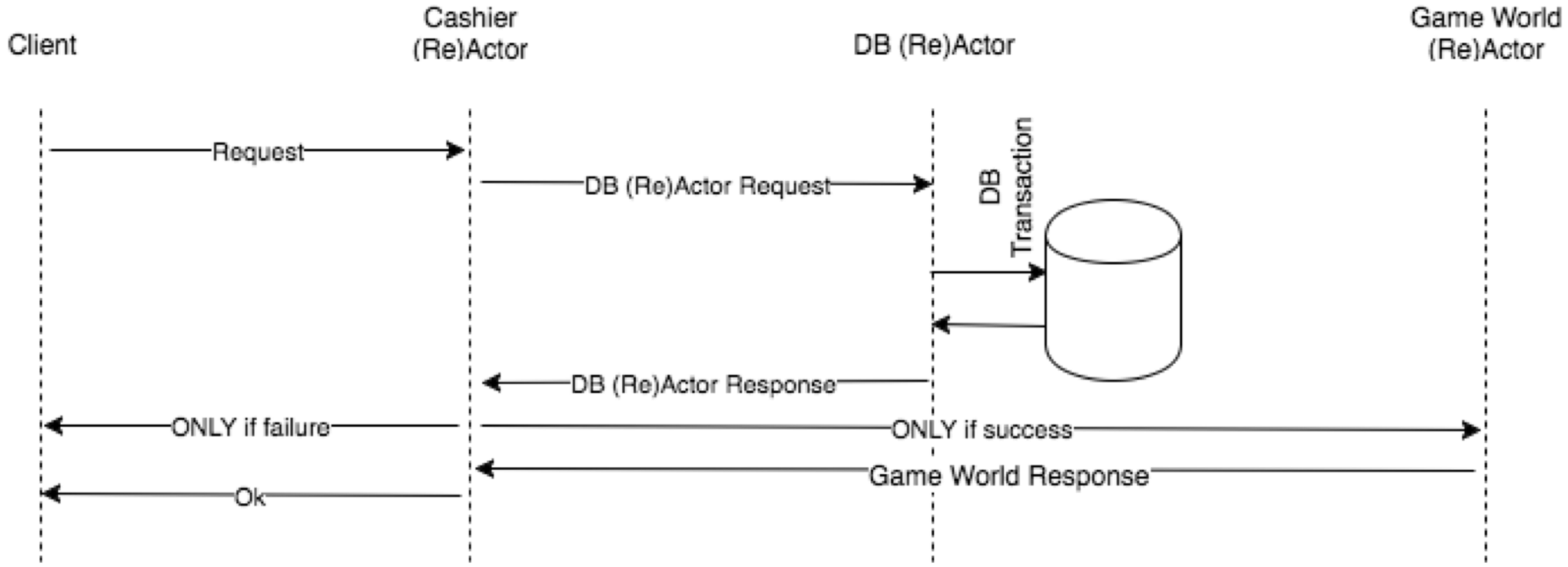
Part 1. Handling Non-Blocking Returns

Holy Grail: non-blocking looking almost like blocking

- *Caveat: Interactions.*
- *Requirement to mark potential flow interruptions.*

Take 1. Plain messages
Take 2. void RPCs
Take 3. OO-style Callbacks
Take 4. Lambda Pyramid
Take 5. Futures
Take 6. Code Builder
Take 7. Stackful Coroutines/Fibers
Take 8. co_await

“Item Purchase” Example



Blocking Code:

```
bool CashierReactor::purchaseItem(
            int item_id, int connection_id) {
  int user_id = get_user_id(connection_id);

  bool db_ok = dbPurchaseItem(db_reactor_id,
                              user_id, item_id);
  if(!db_ok)
    return false;

  REACTORID gameworld_reactor_id =
                find_gameworld_for_user(user_id);
  bool gameworld_ok = gameworldAddItem(
                        gameworld_reactor_id,
                        user_id, item_id);

  return gameworld_ok;
}
```

Stubs Generated from IDL

"Item Purchase" Example - Non-Blocking Interactions

IMPORTANT Caveat:

```
bool CashierReactor::purchaseItem(
            int item_id, int connection_id) {
  int user_id = get_user_id(connection_id);

  int some_data = this->some_data;
  bool db_ok = dbPurchaseItem(db_reactor_id,
                              user_id, item_id);
  assert(some_data == this->some_data);
  if(!db_ok)
    return false;

  REACTORID gameworld_reactor_id =
                find_gameworld_for_user(user_id);
  bool gameworld_ok = gameworldAddItem(
                          gameworld_reactor_id,
                          user_id, item_id);
  return gameworld_ok;
}
```

MAY fail in non-blocking code

"Holy Grail" Non-Blocking Code:

```
bool CashierReactor::purchaseItem(
            int item_id, int connection_id) {
  int user_id = get_user_id(connection_id);

  int some_data = this->some_data;
  bool db_ok = REENTRY dbPurchaseItem(db_reactor_id,
                                      user_id, item_id);
  assert(some_data == this->some_data);
  if(!db_ok)
    return false;

  REACTORID gameworld_reactor_id =
                find_gameworld_for_user(user_id);
  bool gameworld_ok = REENTRY gameworldAddItem(
                                gameworld_reactor_id,
                                user_id, item_id);
  return gameworld_ok;
}
```

MAY still fail, but at least we can see it in advance

“Item Purchase” Example - Take 1. Plain Messages

Take 1:

```
struct PurchaseRqData {
  enum class Status { DBRequested, GameWorldRequested };
  Status status;
  int user_request_id;
  int user_id;
  int item_id;

  PurchaseRqData(int user_request_id_,
                 int user_id_, int item_id)
 : user_request_id(user_request_id_),
   user_id(user_id_), item_id(item_id_) {
    status = Status::DBRequested;
  }
};

class CashierReactor {
  map<int,PurchaseRqData> purchase_item_requests;

  public:
  void react(const Event& ev);
};

void CashierReactor::react(const Event& ev) {
  switch( ev.type ) {
    case CASHIER_PURCHASEITEM_REQUEST:
    {
      const Msg& msg = ev.msg;
      int user_request_id, item_id;
      tie(user_request_id, item_id) =
          cashierPurchaseItem_request_parse(msg);
      int user_id = get_user_id(ev);
      int request_id = new_request_id();
      Msg msg2 =
           dbPurchaseItem_request_compose(
           request_id, user_id, item_id);
      send_msg_to(db_reactor_id, msg2);
      purchase_item_requests.insert(
           pair<int, PurchaseRqData>(request_id,
             PurchaseRqData(user_request_id,
             user_id, item_id));
      break;
    }

    case DB_PURCHASEITEM_RESPONSE:
    {
      const Msg& msg = ev.msg;
      int request_id;
      bool db_ok;
      tie(request_id, db_ok) = dbPurchaseItem_parse(msg);
      auto found =
             purchase_item_requests.find(request_id);
      MYASSERT(found != purchase_item_requests.end());
      MYASSERT(found->status ==
            PurchaseRqData::Status::DBRequested);
      if(!db_ok) {
        Msg msg3 =
          cashierPurchaseItem_response_compose(
          found->second.user_request_id, false);
        send_msg_back_to(user_id, msg3);
        purchase_item_requests.erase(found);
        break;
      }

      REACTORID gameworld_reactor_id =
          find_gameworld_for_user(
          found->second.user_id);
      Msg msg4 =
          gameworldAddItem_request_compose(
          request_id,
          found->second.user_id,
          found->second.item_id);
      send_msg_to(gameworld_reactor_id, msg4);
      found->status =
          PurchaseRqData::Status::GameWorldRequested;
      break;
    }

    case GAMEWORLD_ADDITEM_RESPONSE:
    {
      const Msg& msg = ev.msg;
      int request_id;
      bool gw_ok;
      tie(request_id, gw_ok) =
               gameworldAddItem_response_parse(msg);
      auto found = purchase_item_requests.find(
                 request_id);
      MYASSERT(found != purchase_item_requests.end());
      MYASSERT(found->status ==
           PurchaseRqData::Status::GameWorldRequested);

      Msg msg2 =
            cashierPurchaseItem_response_compose(
            found->second.user_request_id, gw_ok);

      send_msg_back_to(user_id, msg2);
      purchase_item_requests.erase(found);
      break;
    }
  }
}
```

“Item Purchase” Example - Take 1. Plain Messages

Take 1:

```
struct PurchaseRqData {
  enum class Status { DBRequested, GameWorldRequested };
  Status status;
  int user_request_id;
  int user_id;
  int item_id;

  PurchaseRqData(int user_request_id_,
                 int user_id_, int item_id)
 : user_request_id(user_request_id_),
  user_id(user_id_), item_id(item_id_) {
   status = Status::DBRequested;
  }
};

class CashierReactor {
  map<int,PurchaseRqData> purchase_item_requests;

  public:
  void react(const Event& ev);
};

void CashierReactor::react(const Event& ev) {
  switch( ev.type ) {
    case CASHIER_PURCHASEITEM_REQUEST:
    {
      const Msg& msg = ev.msg;
      int user_request_id, item_id;
      tie(user_request_id, item_id) =
          cashierPurchaseItem_request_parse(msg);
      int user_id = get_user_id(ev);
      int request_id = new_request_id();
      Msg msg2 =
           dbPurchaseItem_request_compose(
           request_id, user_id, item_id);
      send_msg_to(db_reactor_id, msg2);
      purchase_item_requests.insert(
           pair<int, PurchaseRqData>(request_id,
             PurchaseRqData(user_request_id,
             user_id, item_id));
      break;
    }

    case DB_PURCHASEITEM_RESPONSE:
    {
      const Msg& msg = ev.msg;
      int request_id;
      bool db_ok;
      tie(request_id, db_ok) = dbPurchaseItem_parse(msg);
      auto found =
             purchase_item_requests.find(request_id);
      MYASSERT(found != purchase_item_requests.end());
      MYASSERT(found->status ==
           PurchaseRqData::Status::DBRequested);
      if(!db_ok) {
        Msg msg3 =
          cashierPurchaseItem_response_compose(
          found->second.user_request_id, false);
        send_msg_back_to(user_id, msg3);
        purchase_item_requests.erase(found);
        break;
      }

      REACTORID gameworld_reactor_id =
          find_gameworld_for_user(
          found->second.user_id);
      Msg msg4 =
          gameworldAddItem_request_compose(
          request_id,
          found->second.user_id,
          found->second.item_id);
      send_msg_to(gameworld_reactor_id, msg4);
      found->status =
          PurchaseRqData::Status::GameWorldRequested;
      break;
    }

    case GAMEWORLD_ADDITEM_RESPONSE:
    {
      const Msg& msg = ev.msg;
      int request_id;
      bool gw_ok;
      tie(request_id, gw_ok) =
               gameworldAddItem_response_parse(msg);
      auto found = purchase_item_requests.find(
                request_id);
      MYASSERT(found != purchase_item_requests.end());
      MYASSERT(found->status ==
           PurchaseRqData::Status::GameWorldRequested);

      Msg msg2 =
            cashierPurchaseItem_response_compose(
            found->second.user_request_id, gw_ok);

      send_msg_back_to(user_id, msg2);
      purchase_item_requests.erase(found);
      break;
    }
  }
}
```

Boilerplate:

```
struct PurchaseRqData {
  enum class Status { DBRequested,
                      GameWorldRequested };
  Status status;
  int user_request_id;
  int user_id;
  int item_id;

  PurchaseRqData(int user_request_id_,
                 int user_id_, int item_id)
 : user_request_id(user_request_id_),
  user_id(user_id_), item_id(item_id_) {
   status = Status::DBRequested;
  }
};
```

“Item Purchase” Example - Take 1. Plain Messages

Take 1:

```
struct PurchaseRqData {
  enum class Status { DBRequested, GameWorldRequested };
  Status status;
  int user_request_id;
  int user_id;
  int item_id;

  PurchaseRqData(int user_request_id_,
                 int user_id_, int item_id)
 : user_request_id(user_request_id_),
   user_id(user_id_), item_id(item_id_) {
    status = Status::DBRequested;
  }
};

class CashierReactor {
  map<int,PurchaseRqData> purchase_item_requests;

  public:
  void react(const Event& ev);
};

void CashierReactor::react(const Event& ev) {
  switch( ev.type ) {
    case CASHIER_PURCHASEITEM_REQUEST:
    {
      const Msg& msg = ev.msg;
      int user_request_id, item_id;
      tie(user_request_id, item_id) =
          cashierPurchaseItem_request_parse(msg);
      int user_id = get_user_id(ev);
      int request_id = new_request_id();
      Msg msg2 =
           dbPurchaseItem_request_compose(
           request_id, user_id, item_id);
      send_msg_to(db_reactor_id, msg2);
      purchase_item_requests.insert(
           pair<int, PurchaseRqData>(request_id,
             PurchaseRqData(user_request_id,
             user_id, item_id));
      break;
    }

    case DB_PURCHASEITEM_RESPONSE:
    {
      const Msg& msg = ev.msg;
      int request_id;
      bool db_ok;
      tie(request_id, db_ok) = dbPurchaseItem_parse(msg);
      auto found =
             purchase_item_requests.find(request_id);
      MYASSERT(found != purchase_item_requests.end());
      MYASSERT(found->status ==
           PurchaseRqData::Status::DBRequested);
      if(!db_ok) {
        Msg msg3 =
          cashierPurchaseItem_response_compose(
          found->second.user_request_id, false);
        send_msg_back_to(user_id, msg3);
        purchase_item_requests.erase(found);
        break;
      }

      REACTORID gameworld_reactor_id =
          find_gameworld_for_user(
          found->second.user_id);
      Msg msg4 =
          gameworldAddItem_request_compose(
          request_id,
          found->second.user_id,
          found->second.item_id);
      send_msg_to(gameworld_reactor_id, msg4);
      found->status =
          PurchaseRqData::Status::GameWorldRequested;
      break;
    }

    case GAMEWORLD_ADDITEM_RESPONSE:
    {
      const Msg& msg = ev.msg;
      int request_id;
      bool gw_ok;
      tie(request_id, gw_ok) =
               gameworldAddItem_response_parse(msg);
      auto found = purchase_item_requests.find(
                 request_id);
      MYASSERT(found != purchase_item_requests.end());
      MYASSERT(found->status ==
           PurchaseRqData::Status::GameWorldRequested);

      Msg msg2 =
            cashierPurchaseItem_response_compose(
            found->second.user_request_id, gw_ok);

      send_msg_back_to(user_id, msg2);
      purchase_item_requests.erase(found);
      break;
    }
  }
}
```

Boilerplate:

```
class CashierReactor {
  map<int,PurchaseRqData>
                 purchase_item_requests;

  public:
  void react(const Event& ev);
};
```

“Item Purchase” Example - Take 1. Plain Messages

Take 1:

```
struct PurchaseRqData {
  enum class Status { DBRequested, GameWorldRequested };
  Status status;
  int user_request_id;
  int user_id;
  int item_id;

  PurchaseRqData(int user_request_id_,
                 int user_id_, int item_id)
 : user_request_id(user_request_id_),
  user_id(user_id_), item_id(item_id_) {
   status = Status::DBRequested;
  }
};

class CashierReactor {
  map<int,PurchaseRqData> purchase_item_requests;

  public:
  void react(const Event& ev);
};

void CashierReactor::react(const Event& ev) {
  switch( ev.type ) {
    case CASHIER_PURCHASEITEM_REQUEST:
    {
      const Msg& msg = ev.msg;
      int user_request_id, item_id;
      tie(user_request_id, item_id) =
          cashierPurchaseItem_request_parse(msg);
      int user_id = get_user_id(ev);
      int request_id = new_request_id();
      Msg msg2 =
           dbPurchaseItem_request_compose(
           request_id, user_id, item_id);
      send_msg_to(db_reactor_id, msg2);
      purchase_item_requests.insert(
           pair<int, PurchaseRqData>(request_id,
             PurchaseRqData(user_request_id,
             user_id, item_id));
      break;
    }

    case DB_PURCHASEITEM_RESPONSE:
    {
      const Msg& msg = ev.msg;
      int request_id;
      bool db_ok;
      tie(request_id, db_ok) = dbPurchaseItem_parse(msg);
      auto found =
             purchase_item_requests.find(request_id);
      MYASSERT(found != purchase_item_requests.end());
      MYASSERT(found->status ==
            PurchaseRqData::Status::DBRequested);
      if(!db_ok) {
        Msg msg3 =
          cashierPurchaseItem_response_compose(
          found->second.user_request_id, false);
        send_msg_back_to(user_id, msg3);
        purchase_item_requests.erase(found);
        break;
      }

      REACTORID gameworld_reactor_id =
          find_gameworld_for_user(
          found->second.user_id);
      Msg msg4 =
          gameworldAddItem_request_compose(
          request_id,
          found->second.user_id,
          found->second.item_id);
      send_msg_to(gameworld_reactor_id, msg4);
      found->status =
          PurchaseRqData::Status::GameWorldRequested;
      break;
    }

    case GAMEWORLD_ADDITEM_RESPONSE:
    {
      const Msg& msg = ev.msg;
      int request_id;
      bool gw_ok;
      tie(request_id, gw_ok) =
                gameworldAddItem_response_parse(msg);
      auto found = purchase_item_requests.find(
                 request_id);
      MYASSERT(found != purchase_item_requests.end());
      MYASSERT(found->status ==
            PurchaseRqData::Status::GameWorldRequested);

      Msg msg2 =
            cashierPurchaseItem_response_compose(
            found->second.user_request_id, gw_ok);

      send_msg_back_to(user_id, msg2);
      purchase_item_requests.erase(found);
      break;
    }
  }
}
```

Boilerplate:

Error-prone

```
void CashierReactor::react(const Event& ev) {
  switch( ev.type ) {
    case CASHIER_PURCHASEITEM_REQUEST:
    {
      const Msg& msg = ev.msg;
      int user_request_id, item_id;
      tie(user_request_id, item_id) =
          cashierPurchaseItem_request_parse(msg);
      int user_id = get_user_id(ev);
      int request_id = new_request_id();
      Msg msg2 =
           dbPurchaseItem_request_compose(
           request_id, user_id, item_id);
      send_msg_to(db_reactor_id, msg2);
      purchase_item_requests.insert(
           pair<int, PurchaseRqData>(request_id,
             PurchaseRqData(user_request_id,
             user_id, item_id));
      break;
    }
…
```

“Item Purchase” Example - Take 1. Plain Messages

Take 1:

```
struct PurchaseRqData {
  enum class Status { DBRequested, GameWorldRequested };
  Status status;
  int user_request_id;
  int user_id;
  int item_id;

  PurchaseRqData(int user_request_id_,
                 int user_id_, int item_id)
 : user_request_id(user_request_id_),
  user_id(user_id_), item_id(item_id_) {
   status = Status::DBRequested;
  }
};

class CashierReactor {
  map<int,PurchaseRqData> purchase_item_requests;

  public:
  void react(const Event& ev);
};

void CashierReactor::react(const Event& ev) {
  switch( ev.type ) {
    case CASHIER_PURCHASEITEM_REQUEST:
    {
      const Msg& msg = ev.msg;
      int user_request_id, item_id;
      tie(user_request_id, item_id) =
          cashierPurchaseItem_request_parse(msg);
      int user_id = get_user_id(ev);
      int request_id = new_request_id();
      Msg msg2 =
           dbPurchaseItem_request_compose(
           request_id, user_id, item_id);
      send_msg_to(db_reactor_id, msg2);
      purchase_item_requests.insert(
           pair<int, PurchaseRqData>(request_id,
             PurchaseRqData(user_request_id,
             user_id, item_id));
      break;
    }

    case DB_PURCHASEITEM_RESPONSE:
    {
      const Msg& msg = ev.msg;
      int request_id;
      bool db_ok;
      tie(request_id, db_ok) = dbPurchaseItem_parse(msg);
      auto found =
             purchase_item_requests.find(request_id);
      MYASSERT(found != purchase_item_requests.end());
      MYASSERT(found->status ==
            PurchaseRqData::Status::DBRequested);
      if(!db_ok) {
        Msg msg3 =
          cashierPurchaseItem_response_compose(
          found->second.user_request_id, false);
        send_msg_back_to(user_id, msg3);
        purchase_item_requests.erase(found);
        break;
      }

      REACTORID gameworld_reactor_id =
          find_gameworld_for_user(
          found->second.user_id);
      Msg msg4 =
          gameworldAddItem_request_compose(
          request_id,
          found->second.user_id,
          found->second.item_id);
      send_msg_to(gameworld_reactor_id, msg4);
      found->status =
          PurchaseRqData::Status::GameWorldRequested;
      break;
    }

    case GAMEWORLD_ADDITEM_RESPONSE:
    {
      const Msg& msg = ev.msg;
      int request_id;
      bool gw_ok;
      tie(request_id, gw_ok) =
               gameworldAddItem_response_parse(msg);
      auto found = purchase_item_requests.find(
                request_id);
      MYASSERT(found != purchase_item_requests.end());
      MYASSERT(found->status ==
           PurchaseRqData::Status::GameWorldRequested);

      Msg msg2 =
            cashierPurchaseItem_response_compose(
            found->second.user_request_id, gw_ok);

      send_msg_back_to(user_id, msg2);
      purchase_item_requests.erase(found);
      break;
    }
  }
}
```

Boilerplate:

```
…
  case DB_PURCHASEITEM_RESPONSE:
  {
    const Msg& msg = ev.msg;
    int request_id;
    bool db_ok;
    tie(request_id, db_ok) =
                    dbPurchaseItem_parse(msg);
    auto found =
     purchase_item_requests.find(request_id);
    MYASSERT(found !=
                purchase_item_requests.end());
    MYASSERT(found->status ==
        PurchaseRqData::Status::DBRequested);
…
```

“Item Purchase” Example - Take 1. Plain Messages

Take 1:

```
struct PurchaseRqData {
  enum class Status { DBRequested, GameWorldRequested };
  Status status;
  int user_request_id;
  int user_id;
  int item_id;

  PurchaseRqData(int user_request_id_,
                 int user_id_, int item_id)
 : user_request_id(user_request_id_),
  user_id(user_id_), item_id(item_id_) {
   status = Status::DBRequested;
  }
};

class CashierReactor {
  map<int,PurchaseRqData> purchase_item_requests;

  public:
  void react(const Event& ev);
};

void CashierReactor::react(const Event& ev) {
  switch( ev.type ) {
    case CASHIER_PURCHASEITEM_REQUEST:
    {
      const Msg& msg = ev.msg;
      int user_request_id, item_id;
      tie(user_request_id, item_id) =
          cashierPurchaseItem_request_parse(msg);
      int user_id = get_user_id(ev);
      int request_id = new_request_id();
      Msg msg2 =
           dbPurchaseItem_request_compose(
           request_id, user_id, item_id);
      send_msg_to(db_reactor_id, msg2);
      purchase_item_requests.insert(
           pair<int, PurchaseRqData>(request_id,
             PurchaseRqData(user_request_id,
             user_id, item_id));
      break;
    }

    case DB_PURCHASEITEM_RESPONSE:
    {
      const Msg& msg = ev.msg;
      int request_id;
      bool db_ok;
      tie(request_id, db_ok) = dbPurchaseItem_parse(msg);
      auto found =
             purchase_item_requests.find(request_id);
      MYASSERT(found != purchase_item_requests.end());
      MYASSERT(found->status ==
             PurchaseRqData::Status::DBRequested);
      if(!db_ok) {
        Msg msg3 =
          cashierPurchaseItem_response_compose(
          found->second.user_request_id, false);
        send_msg_back_to(user_id, msg3);
        purchase_item_requests.erase(found);
        break;
      }

      REACTORID gameworld_reactor_id =
          find_gameworld_for_user(
          found->second.user_id);
      Msg msg4 =
          gameworldAddItem_request_compose(
          request_id,
          found->second.user_id,
          found->second.item_id);
      send_msg_to(gameworld_reactor_id, msg4);
      found->status =
          PurchaseRqData::Status::GameWorldRequested;
      break;
    }

    case GAMEWORLD_ADDITEM_RESPONSE:
    {
      const Msg& msg = ev.msg;
      int request_id;
      bool gw_ok;
      tie(request_id, gw_ok) =
               gameworldAddItem_response_parse(msg);
      auto found = purchase_item_requests.find(
                 request_id);
      MYASSERT(found != purchase_item_requests.end());
      MYASSERT(found->status ==
           PurchaseRqData::Status::GameWorldRequested);

      Msg msg2 =
            cashierPurchaseItem_response_compose(
            found->second.user_request_id, gw_ok);

      send_msg_back_to(user_id, msg2);
      purchase_item_requests.erase(found);
      break;
    }
  }
}
```

Meaningful:

…

```
if(!db_ok) {
```

…

“Item Purchase” Example - Take 1. Plain Messages

Take 1:

```
struct PurchaseRqData {
  enum class Status { DBRequested, GameWorldRequested };
  Status status;
  int user_request_id;
  int user_id;
  int item_id;

  PurchaseRqData(int user_request_id_,
                 int user_id_, int item_id)
 : user_request_id(user_request_id_),
   user_id(user_id_), item_id(item_id_) {
    status = Status::DBRequested;
  }
};

class CashierReactor {
  map<int,PurchaseRqData> purchase_item_requests;

  public:
  void react(const Event& ev);
};

void CashierReactor::react(const Event& ev) {
  switch( ev.type ) {
    case CASHIER_PURCHASEITEM_REQUEST:
    {
      const Msg& msg = ev.msg;
      int user_request_id, item_id;
      tie(user_request_id, item_id) =
          cashierPurchaseItem_request_parse(msg);
      int user_id = get_user_id(ev);
      int request_id = new_request_id();
      Msg msg2 =
           dbPurchaseItem_request_compose(
           request_id, user_id, item_id);
      send_msg_to(db_reactor_id, msg2);
      purchase_item_requests.insert(
           pair<int, PurchaseRqData>(request_id,
             PurchaseRqData(user_request_id,
             user_id, item_id));
      break;
    }

    case DB_PURCHASEITEM_RESPONSE:
    {
      const Msg& msg = ev.msg;
      int request_id;
      bool db_ok;
      tie(request_id, db_ok) = dbPurchaseItem_parse(msg);
      auto found =
             purchase_item_requests.find(request_id);
      MYASSERT(found != purchase_item_requests.end());
      MYASSERT(found->status ==
           PurchaseRqData::Status::DBRequested);
      if(!db_ok) {
        Msg msg3 =
          cashierPurchaseItem_response_compose(
          found->second.user_request_id, false);
        send_msg_back_to(user_id, msg3);
        purchase_item_requests.erase(found);
        break;
      }

      REACTORID gameworld_reactor_id =
          find_gameworld_for_user(
          found->second.user_id);
      Msg msg4 =
          gameworldAddItem_request_compose(
          request_id,
          found->second.user_id,
          found->second.item_id);
      send_msg_to(gameworld_reactor_id, msg4);
      found->status =
          PurchaseRqData::Status::GameWorldRequested;
      break;
    }

    case GAMEWORLD_ADDITEM_RESPONSE:
    {
      const Msg& msg = ev.msg;
      int request_id;
      bool gw_ok;
      tie(request_id, gw_ok) =
                gameworldAddItem_response_parse(msg);
      auto found = purchase_item_requests.find(
                request_id);
      MYASSERT(found != purchase_item_requests.end());
      MYASSERT(found->status ==
           PurchaseRqData::Status::GameWorldRequested);

      Msg msg2 =
            cashierPurchaseItem_response_compose(
            found->second.user_request_id, gw_ok);

      send_msg_back_to(user_id, msg2);
      purchase_item_requests.erase(found);
      break;
    }
  }
}
```

Boilerplate:

```
…
  Msg msg3 =
    cashierPurchaseItem_response_compose(
      found->second.user_request_id, false);
  send_msg_back_to(user_id, msg3);
  purchase_item_requests.erase(found);
  break;
…
```

“Item Purchase” Example - Take 1. Plain Messages

Take 1 (70 LoC):

```
struct PurchaseRqData {
  enum class Status { DBRequested, GameWorldRequested };
  Status status;
  int user_request_id;
  int user_id;
  int item_id;

  PurchaseRqData(int user_request_id_,
                 int user_id_, int item_id)
 : user_request_id(user_request_id_),
   user_id(user_id_), item_id(item_id_) {
    status = Status::DBRequested;
  }
};

class CashierReactor {
  map<int,PurchaseRqData> purchase_item_requests;

  public:
  void react(const Event& ev);
};

void CashierReactor::react(const Event& ev) {
  switch( ev.type ) {
    case CASHIER_PURCHASEITEM_REQUEST:
    {
      const Msg& msg = ev.msg;
      int user_request_id, item_id;
      tie(user_request_id, item_id) =
          cashierPurchaseItem_request_parse(msg);
      int user_id = get_user_id(ev);
      int request_id = new_request_id();
      Msg msg2 =
           dbPurchaseItem_request_compose(
           request_id, user_id, item_id);
      send_msg_to(db_reactor_id, msg2);
      purchase_item_requests.insert(
           pair<int, PurchaseRqData>(request_id,
             PurchaseRqData(user_request_id,
             user_id, item_id));
      break;
    }

    case DB_PURCHASEITEM_RESPONSE:
    {
      const Msg& msg = ev.msg;
      int request_id;
      bool db_ok;
      tie(request_id, db_ok) = dbPurchaseItem_parse(msg);
      auto found =
             purchase_item_requests.find(request_id);
      MYASSERT(found != purchase_item_requests.end());
      MYASSERT(found->status ==
           PurchaseRqData::Status::DBRequested);
      if(!db_ok) {
        Msg msg3 =
          cashierPurchaseItem_response_compose(
          found->second.user_request_id, false);
        send_msg_back_to(user_id, msg3);
        purchase_item_requests.erase(found);
        break;
      }

      REACTORID gameworld_reactor_id =
          find_gameworld_for_user(
          found->second.user_id);
      Msg msg4 =
          gameworldAddItem_request_compose(
          request_id,
          found->second.user_id,
          found->second.item_id);
      send_msg_to(gameworld_reactor_id, msg4);
      found->status =
          PurchaseRqData::Status::GameWorldRequested;
      break;
    }

    case GAMEWORLD_ADDITEM_RESPONSE:
    {
      const Msg& msg = ev.msg;
      int request_id;
      bool gw_ok;
      tie(request_id, gw_ok) =
             gameworldAddItem_response_parse(msg);
      auto found = purchase_item_requests.find(
                request_id);
      MYASSERT(found != purchase_item_requests.end());
      MYASSERT(found->status ==
           PurchaseRqData::Status::GameWorldRequested);

      Msg msg2 =
            cashierPurchaseItem_response_compose(
            found->second.user_request_id, gw_ok);

      send_msg_back_to(user_id, msg2);
      purchase_item_requests.erase(found);
      break;
    }
  }
}
```

“Holy Grail” (10 LoC):

```
bool CashierReactor::purchaseItem(
            int item_id, int connection_id) {

  int user_id = get_user_id(connection_id);

  bool db_ok = REENTRY
       dbPurchaseItem(db_reactor_id,
                      user_id, item_id);

  if(!db_ok)

    return false;

  REACTORID gameworld_reactor_id =
         find_gameworld_for_user(user_id);

  bool gameworld_ok = REENTRY
                      gameworldAddItem(
                      gameworld_reactor_id,
                      user_id, item_id);

 return gameworld_ok;

}
```

“Item Purchase” Example - Take 2. Void-only RPC calls.

Take 2:

```
struct PurchaseRqData { // same as for Take 1
  enum class Status { DBRequested, GameWorldRequested };
  Status status;
  int user_request_id;
  int user_id;
  int item_id;

  PurchaseRqData(int user_request_id_,
                 int user_id_, int item_id)
  : user_request_id(user_request_id_),
    user_id(user_id_), item_id(item_id_) {
    status = Status::DBRequested;
  }
};

class CashierReactor {
  map<int,PurchaseRqData> purchase_item_requests;

  public:
  void cashierPurchaseItemRequest(REACTORID peer_reactor,
       int request_id, int item_id );
  //...
};

void CashierReactor::cashierPurchaseItemRequest(
     REACTORID peer_reactor, int request_id,
     int item_id ) {
  int user_id = get_user_id(peer_reactor);
  int request_id = new_request_id();
  dbPurchaseItemRequest(db_reactor_id,
            request_id,
            user_id, int item_id);

  purchase_item_requests.insert(
    pair<int, PurchaseRqData>(request_id,
        PurchaseRqData(user_request_id,
        user_id, item_id));
}

void CashierReactor::dbPurchaseItemResponse(
     REACTORID peer_reactor, int request_id,
     bool db_ok) {
  auto found = purchase_item_requests.find(request_id);
  MYASSERT(found != purchase_item_requests.end());
  MYASSERT(found->status ==
           PurchaseRqData::Status::DBRequested);
  if(!db_ok) {
    REACTORID user_reactor =
          find_user_reactor_id(found->second.user_id);
    cashierPurchaseItemResponse(user_reactor,
                found->second.user_request_id, false);
    purchase_item_requests.erase(found);
    return;
  }

  REACTORID gameworld_reactor_id =
          find_gameworld_for_user(found->second.user_id);
  gameworldAddItemRequest(gameworld_reactor_id, request_id,
           found->second.user_id, found->second.item_id);
  found->status =
           PurchaseRqData::Status::GameWorldRequested;
}

void CashierReactor::gameworldAddItemResponse(
     REACTORID peer_reactor, int request_id,
     bool gw_ok) {
  auto found = purchase_item_requests.find(request_id);
  MYASSERT(found != purchase_item_requests.end());
  MYASSERT(found->status ==
           PurchaseRqData::Status::GameWorldRequested);

  REACTORID user_reactor =
          find_user_reactor_id(found->second.user_id);
  cashierPurchaseItemResponse(user_reactor,
                 found->second.user_request_id, gw_ok);
  purchase_item_requests.erase(found);
}
```

“Item Purchase” Example - Take 2. Void-only RPC calls.

Take 2:

```
struct PurchaseRqData { // same as for Take 1
  enum class Status { DBRequested, GameWorldRequested };
  Status status;
  int user_request_id;
  int user_id;
  int item_id;

  PurchaseRqData(int user_request_id_,
                 int user_id_, int item_id)
  : user_request_id(user_request_id_),
    user_id(user_id_), item_id(item_id_) {
    status = Status::DBRequested;
  }
};
```

```
class CashierReactor {
  map<int,PurchaseRqData> purchase_item_requests;

  public:
  void cashierPurchaseItemRequest(REACTORID peer_reactor,
       int request_id, int item_id );
  //...
};

void CashierReactor::cashierPurchaseItemRequest(
     REACTORID peer_reactor, int request_id,
     int item_id ) {
  int user_id = get_user_id(peer_reactor);
  int request_id = new_request_id();
  dbPurchaseItemRequest(db_reactor_id,
            request_id,
            user_id, int item_id);

  purchase_item_requests.insert(
    pair<int, PurchaseRqData>(request_id,
        PurchaseRqData(user_request_id,
        user_id, item_id));
}

void CashierReactor::dbPurchaseItemResponse(
     REACTORID peer_reactor, int request_id,
     bool db_ok) {
  auto found = purchase_item_requests.find(request_id);
  MYASSERT(found != purchase_item_requests.end());
  MYASSERT(found->status ==
           PurchaseRqData::Status::DBRequested);
  if(!db_ok) {
    REACTORID user_reactor =
          find_user_reactor_id(found->second.user_id);
    cashierPurchaseItemResponse(user_reactor,
                found->second.user_request_id, false);
    purchase_item_requests.erase(found);
    return;
  }

  REACTORID gameworld_reactor_id =
          find_gameworld_for_user(found->second.user_id);
  gameworldAddItemRequest(gameworld_reactor_id, request_id,
           found->second.user_id, found->second.item_id);
  found->status =
           PurchaseRqData::Status::GameWorldRequested;
}

void CashierReactor::gameworldAddItemResponse(
     REACTORID peer_reactor, int request_id,
     bool gw_ok) {
  auto found = purchase_item_requests.find(request_id);
  MYASSERT(found != purchase_item_requests.end());
  MYASSERT(found->status ==
           PurchaseRqData::Status::GameWorldRequested);

  REACTORID user_reactor =
           find_user_reactor_id(found->second.user_id);
  cashierPurchaseItemResponse(user_reactor,
                 found->second.user_request_id, gw_ok);
  purchase_item_requests.erase(found);
}
```

Boilerplate:

```
struct PurchaseRqData {//same as for Take 1
  enum class Status { DBRequested,
                      GameWorldRequested };
  Status status;
  int user_request_id;
  int user_id;
  int item_id;

  PurchaseRqData(int user_request_id_,
                 int user_id_, int item_id)
 : user_request_id(user_request_id_),
   user_id(user_id_), item_id(item_id_) {
    status = Status::DBRequested;
  }
};
```

“Item Purchase” Example - Take 2. Void-only RPC calls.

Take 2:

```
struct PurchaseRqData { // same as for Take 1
  enum class Status { DBRequested, GameWorldRequested };
  Status status;
  int user_request_id;
  int user_id;
  int item_id;

  PurchaseRqData(int user_request_id_,
                 int user_id_, int item_id)
  : user_request_id(user_request_id_),
    user_id(user_id_), item_id(item_id_) {
    status = Status::DBRequested;
  }
};

class CashierReactor {
  map<int,PurchaseRqData> purchase_item_requests;

  public:
  void cashierPurchaseItemRequest(REACTORID peer_reactor,
      int request_id, int item_id );
  //...
};

void CashierReactor::cashierPurchaseItemRequest(
     REACTORID peer_reactor, int request_id,
     int item_id ) {
  int user_id = get_user_id(peer_reactor);
  int request_id = new_request_id();
  dbPurchaseItemRequest(db_reactor_id,
           request_id,
           user_id, int item_id);

  purchase_item_requests.insert(
    pair<int, PurchaseRqData>(request_id,
        PurchaseRqData(user_request_id,
        user_id, item_id));
}

void CashierReactor::dbPurchaseItemResponse(
     REACTORID peer_reactor, int request_id,
     bool db_ok) {
  auto found = purchase_item_requests.find(request_id);
  MYASSERT(found != purchase_item_requests.end());
  MYASSERT(found->status ==
           PurchaseRqData::Status::DBRequested);
  if(!db_ok) {
    REACTORID user_reactor =
           find_user_reactor_id(found->second.user_id);
    cashierPurchaseItemResponse(user_reactor,
                found->second.user_request_id, false);
    purchase_item_requests.erase(found);
    return;
  }

  REACTORID gameworld_reactor_id =
          find_gameworld_for_user(found->second.user_id);
  gameworldAddItemRequest(gameworld_reactor_id, request_id,
           found->second.user_id, found->second.item_id);
  found->status =
           PurchaseRqData::Status::GameWorldRequested;
}

void CashierReactor::gameworldAddItemResponse(
     REACTORID peer_reactor, int request_id,
     bool gw_ok) {
  auto found = purchase_item_requests.find(request_id);
  MYASSERT(found != purchase_item_requests.end());
  MYASSERT(found->status ==
           PurchaseRqData::Status::GameWorldRequested);

  REACTORID user_reactor =
           find_user_reactor_id(found->second.user_id);
  cashierPurchaseItemResponse(user_reactor,
                found->second.user_request_id, gw_ok);
  purchase_item_requests.erase(found);
}
```

Boilerplate:

```
class CashierReactor {
  map<int,PurchaseRqData>
purchase_item_requests;

  public:
  void
cashierPurchaseItemRequest(REACTORID
        peer_reactor,
        int request_id, int item_id );
  //...
};
```

“Item Purchase” Example - Take 2. Void-only RPC calls.

Take 2:

```
struct PurchaseRqData { // same as for Take 1
  enum class Status { DBRequested, GameWorldRequested };
  Status status;
  int user_request_id;
  int user_id;
  int item_id;

  PurchaseRqData(int user_request_id_,
                 int user_id_, int item_id)
  : user_request_id(user_request_id_),
    user_id(user_id_), item_id(item_id_) {
    status = Status::DBRequested;
  }
};

class CashierReactor {
  map<int,PurchaseRqData> purchase_item_requests;

  public:
  void cashierPurchaseItemRequest(REACTORID peer_reactor,
       int request_id, int item_id );
  //...
};

void CashierReactor::cashierPurchaseItemRequest(
     REACTORID peer_reactor, int request_id,
     int item_id ) {
  int user_id = get_user_id(peer_reactor);
  int request_id = new_request_id();
  dbPurchaseItemRequest(db_reactor_id,
            request_id,
            user_id, int item_id);

  purchase_item_requests.insert(
    pair<int, PurchaseRqData>(request_id,
        PurchaseRqData(user_request_id,
        user_id, item_id));
}

void CashierReactor::dbPurchaseItemResponse(
     REACTORID peer_reactor, int request_id,
     bool db_ok) {
  auto found = purchase_item_requests.find(request_id);
  MYASSERT(found != purchase_item_requests.end());
  MYASSERT(found->status ==
           PurchaseRqData::Status::DBRequested);
  if(!db_ok) {
    REACTORID user_reactor =
          find_user_reactor_id(found->second.user_id);
    cashierPurchaseItemResponse(user_reactor,
                found->second.user_request_id, false);
    purchase_item_requests.erase(found);
    return;
  }

  REACTORID gameworld_reactor_id =
         find_gameworld_for_user(found->second.user_id);
  gameworldAddItemRequest(gameworld_reactor_id, request_id,
           found->second.user_id, found->second.item_id);
  found->status =
           PurchaseRqData::Status::GameWorldRequested;
}

void CashierReactor::gameworldAddItemResponse(
     REACTORID peer_reactor, int request_id,
     bool gw_ok) {
  auto found = purchase_item_requests.find(request_id);
  MYASSERT(found != purchase_item_requests.end());
  MYASSERT(found->status ==
           PurchaseRqData::Status::GameWorldRequested);

  REACTORID user_reactor =
           find_user_reactor_id(found->second.user_id);
  cashierPurchaseItemResponse(user_reactor,
                found->second.user_request_id, gw_ok);
  purchase_item_requests.erase(found);
}
```

Error-prone

Boilerplate:

```
int request_id = new_request_id();
dbPurchaseItemRequest(db_reactor_id,
            request_id,
            user_id, int item_id);

purchase_item_requests.insert(
  pair<int, PurchaseRqData>(request_id,
      PurchaseRqData(user_request_id,
      user_id, item_id));
```

Boilerplate:

```
auto found =
  purchase_item_requests.find(request_id);
MYASSERT(found !=
         purchase_item_requests.end());
MYASSERT(found->status ==
    PurchaseRqData::Status::DBRequested);
```

“Item Purchase” Example - Take 2. Void-only RPC calls.

Take 2 (50 LoC):

```
struct PurchaseRqData { // same as for Take 1
  enum class Status { DBRequested, GameWorldRequested };
  Status status;
  int user_request_id;
  int user_id;
  int item_id;

  PurchaseRqData(int user_request_id_,
                 int user_id_, int item_id)
  : user_request_id(user_request_id_),
    user_id(user_id_), item_id(item_id_) {
    status = Status::DBRequested;
  }
};

class CashierReactor {
  map<int,PurchaseRqData> purchase_item_requests;

  public:
  void cashierPurchaseItemRequest(REACTORID peer_reactor,
       int request_id, int item_id );
  //...
};

void CashierReactor::cashierPurchaseItemRequest(
     REACTORID peer_reactor, int request_id,
     int item_id ) {
  int user_id = get_user_id(peer_reactor);
  int request_id = new_request_id();
  dbPurchaseItemRequest(db_reactor_id,
            request_id,
            user_id, int item_id);

  purchase_item_requests.insert(
    pair<int, PurchaseRqData>(request_id,
        PurchaseRqData(user_request_id,
        user_id, item_id));
}

void CashierReactor::dbPurchaseItemResponse(
     REACTORID peer_reactor, int request_id,
     bool db_ok) {
  auto found = purchase_item_requests.find(request_id);
  MYASSERT(found != purchase_item_requests.end());
  MYASSERT(found->status ==
           PurchaseRqData::Status::DBRequested);
  if(!db_ok) {
    REACTORID user_reactor =
          find_user_reactor_id(found->second.user_id);
    cashierPurchaseItemResponse(user_reactor,
                found->second.user_request_id, false);
    purchase_item_requests.erase(found);
    return;
  }

  REACTORID gameworld_reactor_id =
          find_gameworld_for_user(found->second.user_id);
  gameworldAddItemRequest(gameworld_reactor_id, request_id,
           found->second.user_id, found->second.item_id);
  found->status =
           PurchaseRqData::Status::GameWorldRequested;
}

void CashierReactor::gameworldAddItemResponse(
     REACTORID peer_reactor, int request_id,
     bool gw_ok) {
  auto found = purchase_item_requests.find(request_id);
  MYASSERT(found != purchase_item_requests.end());
  MYASSERT(found->status ==
           PurchaseRqData::Status::GameWorldRequested);

  REACTORID user_reactor =
           find_user_reactor_id(found->second.user_id);
  cashierPurchaseItemResponse(user_reactor,
                  found->second.user_request_id, gw_ok);
  purchase_item_requests.erase(found);
}
```

“Holy Grail” (10 LoC):

```
bool CashierReactor::purchaseItem(
            int item_id, int connection_id) {

  int user_id = get_user_id(connection_id);

  bool db_ok = REENTRY
       dbPurchaseItem(db_reactor_id,
                      user_id, item_id);

  if(!db_ok)

    return false;

  REACTORID gameworld_reactor_id =
         find_gameworld_for_user(user_id);
  bool gameworld_ok = REENTRY
                      gameworldAddItem(
                      gameworld_reactor_id,
                      user_id, item_id);

  return gameworld_ok;

}
```

“Item Purchase” Example - Take 3. OO Callbacks.

Take 3:

```
class DbPurchaseItemCallbackA
: public DbPurchaseItemCallback {
  shared_ptr<CashierPurchaseItemReply> reply_handle;
  int user_id;
  int item_id;

  public:
  DbPurchaseItemCallbackA(Reactor* r,
     shared_ptr<CashierPurchaseItemReply>& reply_handle_,
     int user_id_, int item_id_)
  : DbPurchaseItemCallback(r), reply_handle(reply_handle_),
    user_id(user_id_), item_id(item_id_) {
  }

  void react(bool db_ok) override;
};

class GameworldAddItemCallbackA
  : public GameworldAddItemCallback {
  shared_ptr<CashierPurchaseItemReply> reply_handle;
  int user_id;
  int item_id;

  public:
  GameworldAddItemCallbackA(Reactor* r,
     shared_ptr<CashierPurchaseItemReply>& reply_handle_,
     int user_id_, int item_id_)
  : GameworldAddItemCallback(r),
reply_handle(reply_handle_),
    user_id(user_id_), item_id(item_id_) {
  }

  void react(bool gw_ok) override;
};

void CashierReactor::cashierPurchaseItem(
  shared_ptr<CashierPurchaseItemReply> reply_handle,
  int item_id) {

  int user_id = get_user_id(reply_handle);
  auto cb = new DbPurchaseItemCallbackA(
               this, reply_handle,
               user_id, item_id);
  dbPurchaseItem(cb, db_reactor_id,
               user_id, item_id);
}

void DbPurchaseItemCallbackA::react(bool db_ok) {
  if(!db_ok) {
    reply_handle->reply(false);
    return;
  }
  REACTORID gameworld_reactor_id =
        get_reactor()->find_gameworld_for_user(user_id);
  auto cb = new GameworldAddItemCallbackA(
               get_reactor(), reply_handle,
               user_id, item_id);
  gameworldAddItem(cb, gameworld_reactor_id,
                  user_id, item_id);
}

void GameworldAddItemCallbackA::react(bool gw_ok) {
  reply_handle->reply(gw_ok);
}
```

“Item Purchase” Example - Take 3. OO Callbacks.

Take 3:

```
class DbPurchaseItemCallbackA
: public DbPurchaseItemCallback {
  shared_ptr<CashierPurchaseItemReply> reply_handle;
  int user_id;
  int item_id;

  public:
  DbPurchaseItemCallbackA(Reactor* r,
     shared_ptr<CashierPurchaseItemReply>& reply_handle_,
     int user_id_, int item_id_)
  : DbPurchaseItemCallback(r), reply_handle(reply_handle_),
    user_id(user_id_), item_id(item_id_) {
  }

  void react(bool db_ok) override;
};
```

```
class GameworldAddItemCallbackA
  : public GameworldAddItemCallback {
  shared_ptr<CashierPurchaseItemReply> reply_handle;
  int user_id;
  int item_id;

  public:
  GameworldAddItemCallbackA(Reactor* r,
     shared_ptr<CashierPurchaseItemReply>& reply_handle_,
     int user_id_, int item_id_)
  : GameworldAddItemCallback(r),
reply_handle(reply_handle_),
    user_id(user_id_), item_id(item_id_) {
  }

  void react(bool gw_ok) override;
};

void CashierReactor::cashierPurchaseItem(
  shared_ptr<CashierPurchaseItemReply> reply_handle,
  int item_id) {

  int user_id = get_user_id(reply_handle);
  auto cb = new DbPurchaseItemCallbackA(
                this, reply_handle,
                user_id, item_id);
  dbPurchaseItem(cb, db_reactor_id,
                user_id, item_id);
}

void DbPurchaseItemCallbackA::react(bool db_ok) {
  if(!db_ok) {
    reply_handle->reply(false);
    return;
  }
  REACTORID gameworld_reactor_id =
        get_reactor()->find_gameworld_for_user(user_id);
  auto cb = new GameworldAddItemCallbackA(
                get_reactor(), reply_handle,
                user_id, item_id);
  gameworldAddItem(cb, gameworld_reactor_id,
                   user_id, item_id);
}

void GameworldAddItemCallbackA::react(bool gw_ok) {
  reply_handle->reply(gw_ok);
}
```

Boilerplate:

```
class DbPurchaseItemCallbackA
: public DbPurchaseItemCallback {
  shared_ptr<CashierPurchaseItemReply>
                                reply_handle;
  int user_id;
  int item_id;

  public:
  DbPurchaseItemCallbackA(Reactor* r,
     shared_ptr<CashierPurchaseItemReply>&
       reply_handle_,
     int user_id_, int item_id_)
  : DbPurchaseItemCallback(r),
    reply_handle(reply_handle_),
    user_id(user_id_), item_id(item_id_) {
  }

  void react(bool db_ok) override;
};
```

“Item Purchase” Example - Take 3. OO Callbacks.

Take 3:

```
class DbPurchaseItemCallbackA
: public DbPurchaseItemCallback {
  shared_ptr<CashierPurchaseItemReply> reply_handle;
  int user_id;
  int item_id;

  public:
  DbPurchaseItemCallbackA(Reactor* r,
     shared_ptr<CashierPurchaseItemReply>& reply_handle_,
     int user_id_, int item_id_)
  : DbPurchaseItemCallback(r), reply_handle(reply_handle_),
    user_id(user_id_), item_id(item_id_) {
  }

  void react(bool db_ok) override;
};

class GameworldAddItemCallbackA
  : public GameworldAddItemCallback {
  shared_ptr<CashierPurchaseItemReply> reply_handle;
  int user_id;
  int item_id;

  public:
  GameworldAddItemCallbackA(Reactor* r,
     shared_ptr<CashierPurchaseItemReply>& reply_handle_,
     int user_id_, int item_id_)
  : GameworldAddItemCallback(r),
reply_handle(reply_handle_),
    user_id(user_id_), item_id(item_id_) {
  }

  void react(bool gw_ok) override;
};

void CashierReactor::cashierPurchaseItem(
  shared_ptr<CashierPurchaseItemReply> reply_handle,
  int item_id) {

  int user_id = get_user_id(reply_handle);
  auto cb = new DbPurchaseItemCallbackA(
                this, reply_handle,
                user_id, item_id);
  dbPurchaseItem(cb, db_reactor_id,
                user_id, item_id);
}

void DbPurchaseItemCallbackA::react(bool db_ok) {
  if(!db_ok) {
    reply_handle->reply(false);
    return;
  }
  REACTORID gameworld_reactor_id =
        get_reactor()->find_gameworld_for_user(user_id);
  auto cb = new GameworldAddItemCallbackA(
                get_reactor(), reply_handle,
                user_id, item_id);
  gameworldAddItem(cb, gameworld_reactor_id,
                   user_id, item_id);
}

void GameworldAddItemCallbackA::react(bool gw_ok) {
  reply_handle->reply(gw_ok);
}
```

Boilerplate:

```
class GameworldAddItemCallbackA
  : public GameworldAddItemCallback {
  shared_ptr<CashierPurchaseItemReply>
    reply_handle;
  int user_id;
  int item_id;

  public:
  GameworldAddItemCallbackA(Reactor* r,
     shared_ptr<CashierPurchaseItemReply>&
       reply_handle_,
     int user_id_, int item_id_)
  : GameworldAddItemCallback(r),
      reply_handle(reply_handle_),
    user_id(user_id_), item_id(item_id_) {
  }

  void react(bool gw_ok) override;
};
```

“Item Purchase” Example - Take 3. OO Callbacks.

Take 3:

```
class DbPurchaseItemCallbackA
: public DbPurchaseItemCallback {
  shared_ptr<CashierPurchaseItemReply> reply_handle;
  int user_id;
  int item_id;

  public:
  DbPurchaseItemCallbackA(Reactor* r,
     shared_ptr<CashierPurchaseItemReply>& reply_handle_,
     int user_id_, int item_id_)
  : DbPurchaseItemCallback(r), reply_handle(reply_handle_),
    user_id(user_id_), item_id(item_id_) {
  }

  void react(bool db_ok) override;
};

class GameworldAddItemCallbackA
  : public GameworldAddItemCallback {
  shared_ptr<CashierPurchaseItemReply> reply_handle;
  int user_id;
  int item_id;

  public:
  GameworldAddItemCallbackA(Reactor* r,
     shared_ptr<CashierPurchaseItemReply>& reply_handle_,
     int user_id_, int item_id_)
  : GameworldAddItemCallback(r),
reply_handle(reply_handle_),
    user_id(user_id_), item_id(item_id_) {
  }

  void react(bool gw_ok) override;
};

void CashierReactor::cashierPurchaseItem(
  shared_ptr<CashierPurchaseItemReply> reply_handle,
  int item_id) {

  int user_id = get_user_id(reply_handle);
  auto cb = new DbPurchaseItemCallbackA(
               this, reply_handle,
               user_id, item_id);
  dbPurchaseItem(cb, db_reactor_id,
               user_id, item_id);
}

void DbPurchaseItemCallbackA::react(bool db_ok) {
  if(!db_ok) {
    reply_handle->reply(false);
    return;
  }
  REACTORID gameworld_reactor_id =
        get_reactor()->find_gameworld_for_user(user_id);
  auto cb = new GameworldAddItemCallbackA(
               get_reactor(), reply_handle,
               user_id, item_id);
  gameworldAddItem(cb, gameworld_reactor_id,
                  user_id, item_id);
}

void GameworldAddItemCallbackA::react(bool gw_ok) {
  reply_handle->reply(gw_ok);
}
```

Somewhat-Meaningful:

```
void CashierReactor::cashierPurchaseItem(
  shared_ptr<CashierPurchaseItemReply>
    reply_handle,
  int item_id) {
  int user_id = get_user_id(reply_handle);
  auto cb = new DbPurchaseItemCallbackA(
               this, reply_handle,
               user_id, item_id);
  dbPurchaseItem(cb, db_reactor_id,
               user_id, item_id);
}
```

“Item Purchase” Example - Take 3. OO Callbacks.

Take 3:

```
class DbPurchaseItemCallbackA
: public DbPurchaseItemCallback {
  shared_ptr<CashierPurchaseItemReply> reply_handle;
  int user_id;
  int item_id;

  public:
  DbPurchaseItemCallbackA(Reactor* r,
     shared_ptr<CashierPurchaseItemReply>& reply_handle_,
     int user_id_, int item_id_)
  : DbPurchaseItemCallback(r), reply_handle(reply_handle_),
    user_id(user_id_), item_id(item_id_) {
  }

  void react(bool db_ok) override;
};

class GameworldAddItemCallbackA
  : public GameworldAddItemCallback {
  shared_ptr<CashierPurchaseItemReply> reply_handle;
  int user_id;
  int item_id;

  public:
  GameworldAddItemCallbackA(Reactor* r,
     shared_ptr<CashierPurchaseItemReply>& reply_handle_,
     int user_id_, int item_id_)
  : GameworldAddItemCallback(r),
reply_handle(reply_handle_),
    user_id(user_id_), item_id(item_id_) {
  }

  void react(bool gw_ok) override;
};

void CashierReactor::cashierPurchaseItem(
  shared_ptr<CashierPurchaseItemReply> reply_handle,
  int item_id) {

  int user_id = get_user_id(reply_handle);
  auto cb = new DbPurchaseItemCallbackA(
               this, reply_handle,
               user_id, item_id);
  dbPurchaseItem(cb, db_reactor_id,
               user_id, item_id);
}

void DbPurchaseItemCallbackA::react(bool db_ok) {
  if(!db_ok) {
    reply_handle->reply(false);
    return;
  }
  REACTORID gameworld_reactor_id =
        get_reactor()->find_gameworld_for_user(user_id);
  auto cb = new GameworldAddItemCallbackA(
               get_reactor(), reply_handle,
               user_id, item_id);
  gameworldAddItem(cb, gameworld_reactor_id,
                  user_id, item_id);
}

void GameworldAddItemCallbackA::react(bool gw_ok) {
  reply_handle->reply(gw_ok);
}
```

Somewhat-Meaningful:

```
void DbPurchaseItemCallbackA::
                          react(bool db_ok) {
  if(!db_ok) {
    reply_handle->reply(false);
    return;
  }
  REACTORID gameworld_reactor_id =
        get_reactor()->
        find_gameworld_for_user(user_id);
  auto cb = new GameworldAddItemCallbackA(
              get_reactor(), reply_handle,
              user_id, item_id);
  gameworldAddItem(cb,
                     gameworld_reactor_id,
                     user_id, item_id);
}
```

“Item Purchase” Example - Take 3. OO Callbacks.

Take 3 (40LoC):

```
class DbPurchaseItemCallbackA
: public DbPurchaseItemCallback {
  shared_ptr<CashierPurchaseItemReply> reply_handle;
  int user_id;
  int item_id;

  public:
  DbPurchaseItemCallbackA(Reactor* r,
     shared_ptr<CashierPurchaseItemReply>& reply_handle_,
     int user_id_, int item_id_)
  : DbPurchaseItemCallback(r), reply_handle(reply_handle_),
    user_id(user_id_), item_id(item_id_) {
  }

  void react(bool db_ok) override;
};

class GameworldAddItemCallbackA
  : public GameworldAddItemCallback {
  shared_ptr<CashierPurchaseItemReply> reply_handle;
  int user_id;
  int item_id;

  public:
  GameworldAddItemCallbackA(Reactor* r,
     shared_ptr<CashierPurchaseItemReply>& reply_handle_,
     int user_id_, int item_id_)
  : GameworldAddItemCallback(r),
reply_handle(reply_handle_),
    user_id(user_id_), item_id(item_id_) {
  }

  void react(bool gw_ok) override;
};

void CashierReactor::cashierPurchaseItem(
  shared_ptr<CashierPurchaseItemReply> reply_handle,
  int item_id) {

  int user_id = get_user_id(reply_handle);
  auto cb = new DbPurchaseItemCallbackA(
               this, reply_handle,
               user_id, item_id);
  dbPurchaseItem(cb, db_reactor_id,
               user_id, item_id);
}

void DbPurchaseItemCallbackA::react(bool db_ok) {
  if(!db_ok) {
    reply_handle->reply(false);
    return;
  }
  REACTORID gameworld_reactor_id =
        get_reactor()->find_gameworld_for_user(user_id);
  auto cb = new GameworldAddItemCallbackA(
               get_reactor(), reply_handle,
               user_id, item_id);
  gameworldAddItem(cb, gameworld_reactor_id,
                  user_id, item_id);
}

void GameworldAddItemCallbackA::react(bool gw_ok) {
  reply_handle->reply(gw_ok);
}
```

“Holy Grail” (10 LoC):

```
bool CashierReactor::purchaseItem(
            int item_id, int connection_id) {

  int user_id = get_user_id(connection_id);



  bool db_ok = REENTRY
       dbPurchaseItem(db_reactor_id,
                      user_id, item_id);


  if(!db_ok)


    return false;

  REACTORID gameworld_reactor_id =
         find_gameworld_for_user(user_id);


  bool gameworld_ok = REENTRY
                      gameworldAddItem(
                      gameworld_reactor_id,
                      user_id, item_id);


  return gameworld_ok;
}
```

Take 4:

```
void
CashierReactor::cashierPurchaseItem(
  shared_ptr<CashierPurchaseItemReply>
   reply_handle,
  int item_id) {
  int user_id =
        get_user_id(reply_handle);
  dbPurchaseItem(db_reactor_id,
    user_id, item_id,
    [=](bool db_ok) {
     if(!db_ok) {
        reply_handle->reply(false);
        return;
      }
      REACTORID gameworld_reactor_id =
      find_gameworld_for_user(user_id);
      gameworldAddItem(
        gameworld_reactor_id,
        user_id, item_id,
        [=](bool gw_ok){
          reply_handle->reply(gw_ok);
        });
    });
}
```

“Item Purchase” Example - Take 4. Lambda Pyramids.

Take 4 (12 LoC):

```
void
CashierReactor::cashierPurchaseItem(
  shared_ptr<CashierPurchaseItemReply>
   reply_handle,
  int item_id) {
  int user_id =
        get_user_id(reply_handle);
  dbPurchaseItem(db_reactor_id,
    user_id, item_id,
    [=](bool db_ok) {
     if(!db_ok) {
        reply_handle->reply(false);
        return;
      }
      REACTORID gameworld_reactor_id =
      find_gameworld_for_user(user_id);
      gameworldAddItem(
        gameworld_reactor_id,
        user_id, item_id,
        [=](bool gw_ok){
          reply_handle->reply(gw_ok);
        });
    });
}
```

“Holy Grail” (10 LoC):

```
bool CashierReactor::purchaseItem(
  int item_id, int connection_id) {



  int user_id =
      get_user_id(connection_id);
  bool db_ok = REENTRY
       dbPurchaseItem(db_reactor_id,
         user_id, item_id);
  if(!db_ok)
    return false;


  REACTORID gameworld_reactor_id =
    find_gameworld_for_user(user_id);
  bool gameworld_ok = REENTRY
    gameworldAddItem(
      gameworld_reactor_id,
      user_id, item_id);
  return gameworld_ok;


}
```

Take 5:

```
void CashierReactor::cashierPurchaseItem(
  shared_ptr<CashierPurchaseItemReply>
    reply_handle,
   int item_id) {
  int user_id =
    get_user_id(reply_handle);
  ReactorFuture<bool> db_ok =
    dbPurchaseItem( this, db_reactor_id,
                    user_id, item_id);
  ReactorFuture<bool> gw_ok(this);
  db_ok.then([=](){
    if(!db_ok.value()) {
      reply_handle->reply(false);
      return;
    }
    REACTORID gameworld_reactor_id =
      find_gameworld_for_user(user_id);
    gw_ok = gameworldAddItem(
            this, gameworld_reactor_id,
            user_id, item_id);
  });

  gw_ok.then([=](){
    reply_handle->reply(gw_ok.value());
  });
}
```

Take 5:

```
void CashierReactor::cashierPurchaseItem(
  shared_ptr<CashierPurchaseItemReply>
    reply_handle,
   int item_id) {
  int user_id =
    get_user_id(reply_handle);
  ReactorFuture<bool> db_ok =
    dbPurchaseItem( this, db_reactor_id,
                    user_id, item_id);
  ReactorFuture<bool> gw_ok(this);
  db_ok.then([=](){
    if(!db_ok.value()) {
      reply_handle->reply(false);
      return;
    }
    REACTORID gameworld_reactor_id =
      find_gameworld_for_user(user_id);
    gw_ok = gameworldAddItem(
            this, gameworld_reactor_id,
            user_id, item_id);
  });

  gw_ok.then([=](){
    reply_handle->reply(gw_ok.value());
  });
}
```

“Item Purchase” Example - Take 5. Futures.

Take 5:

```
void CashierReactor::cashierPurchaseItem(
  shared_ptr<CashierPurchaseItemReply>
    reply_handle,
   int item_id) {
  int user_id =
    get_user_id(reply_handle);
  ReactorFuture<bool> db_ok =
    dbPurchaseItem( this, db_reactor_id,
                    user_id, item_id);
  ReactorFuture<bool> gw_ok(this);
  db_ok.then([=](){
    if(!db_ok.value()) {
      reply_handle->reply(false);
      return;
    }
    REACTORID gameworld_reactor_id =
      find_gameworld_for_user(user_id);
    gw_ok = gameworldAddItem(
            this, gameworld_reactor_id,
            user_id, item_id);
  });

  gw_ok.then([=](){
    reply_handle->reply(gw_ok.value());
  });
}
```

“Holy Grail”:

```
bool CashierReactor::purchaseItem(
  int item_id, int connection_id) {


  int user_id =
      get_user_id(connection_id);

  bool db_ok = REENTRY
       dbPurchaseItem(db_reactor_id,
         user_id, item_id);

  if(!db_ok)
    return false;


  REACTORID gameworld_reactor_id =
    find_gameworld_for_user(user_id);
  bool gameworld_ok = REENTRY
    gameworldAddItem(
      gameworld_reactor_id,
      user_id, item_id);
  return gameworld_ok;


}
```

Take 5:

```
void CashierReactor::cashierPurchaseItem(
  shared_ptr<CashierPurchaseItemReply>
    reply_handle,
   int item_id) {
  int user_id =
    get_user_id(reply_handle);
  ReactorFuture<bool> db_ok =
    dbPurchaseItem( this, db_reactor_id,
                    user_id, item_id);
  ReactorFuture<bool> gw_ok(this);
  db_ok.then([=](){
    if(!db_ok.value()) {
      reply_handle->reply(false);
      return;
    }
    REACTORID gameworld_reactor_id =
      find_gameworld_for_user(user_id);
    gw_ok = gameworldAddItem(
            this, gameworld_reactor_id,
            user_id, item_id);
  });

  gw_ok.then([=](){
    reply_handle->reply(gw_ok.value());
  });
}
```

Unlike std::future<>, more like folly::Future<>

“Item Purchase” Example - Take 6. Code Builder.

Take 6:

```
void CashierReactor::cashierPurchaseItem(
  shared_ptr<CashierPurchaseItemReply>
    reply_handle, int item_id) {
  int user_id = get_user_id(reply_handle);
  RectorFuture<bool> db_ok;
  ReactorFuture<bool> gw_ok;

  CCode code(
    ttry(
      [=](){
        db_ok = dbPurchaseItem(
                db_reactor_id,
                user_id, item_id);
      },
      waitFor(db_ok),
      [=](){
        if(!db_ok.value()) {
          reply_handle.reply(false);
          return eexit();
        }
        REACTORID gameworld_reactor_id =
             find_gameworld_for_user(user_id);
        gw_ok = gameworldAddItem(
                     gameworld_reactor_id,
                     user_id, item_id);
      },
      waitFor(gw_ok),
      [=](){
        reply_handle.reply(gw_ok.value());
      }
    )//ttry
    .ccatch( [=](std::exception& x){
      LogException(x);
    }
  );//CCode
}
```

“Holy Grail”+Exceptions:

```
bool CashierReactor::purchaseItem(
  int item_id, int connection_id) {
    int user_id =
        get_user_id(connection_id);

  try {

    bool db_ok = REENTRY
         dbPurchaseItem(db_reactor_id,
           user_id, item_id);

    if(!db_ok)
      return false;

    REACTORID gameworld_reactor_id =
            find_gameworld_for_user(user_id);
    bool gameworld_ok = REENTRY
      gameworldAddItem(
          gameworld_reactor_id,
          user_id, item_id);

  return gameworld_ok;
  }

  catch( std::exception& x ) {
    LogException(x);
  }

}
```

“Item Purchase” Example - Take 6. Code Builder.

Take 6a:

```
void CashierReactor::cashierPurchaseItem(
  shared_ptr<CashierPurchaseItemReply>
    reply_handle, int item_id) {
  int user_id = get_user_id(reply_handle);
  RectorFuture<bool> db_ok;
  ReactorFuture<bool> gw_ok;

  CCODE
    TTRY
      db_ok = dbPurchaseItem(
                db_reactor_id,
                user_id, item_id);
      WAITFOR(db_ok)
      if(!db_ok.value()) {
          reply_handle.reply(false);
          return eexit();
      }
      REACTORID gameworld_reactor_id =
             find_gameworld_for_user(user_id);
      gw_ok = gameworldAddItem(
                      gameworld_reactor_id,
                      user_id, item_id);
      WAITFOR(gw_ok)
      reply_handle.reply(gw_ok.value());
   ENDTTRY
   CCATCH
      LogException(x);
   ENDCCATCH
  ENDCCODE
}
```

“Holy Grail”+Exceptions:

```
bool CashierReactor::purchaseItem(
  int item_id, int connection_id) {
    int user_id =
        get_user_id(connection_id);



  try {
    bool db_ok = REENTRY
         dbPurchaseItem(db_reactor_id,
           user_id, item_id);

    if(!db_ok)
      return false;


    REACTORID gameworld_reactor_id =
          find_gameworld_for_user(user_id);
    bool gameworld_ok = REENTRY
      gameworldAddItem(
          gameworld_reactor_id,
          user_id, item_id);
     return gameworld_ok;
  }
  catch( std::exception& x ) {
    LogException(x);
  }

}
```

Take 7:

```
void CashierReactor::cashierPurchaseItem(
  shared_ptr<CashierPurchaseItemReply>
    reply_handle, int item_id) {
  int user_id = get_user_id(reply_handle);

  ReactorFuture<bool> db_ok(this);
  ReactorFuture<bool> gw_ok(this);

  try {
    db_ok = dbPurchaseItem(
            db_reactor_id,
            user_id, item_id);
    WAITFOR(db_ok);
    if(!db_ok.value()) {
      reply_handle.reply(false);
      return;
    }
    REACTORID gameworld_reactor_id =
      find_gameworld_for_user(user_id);
    gw_ok = gameworldAddItem(
            gameworld_reactor_id,
            user_id, item_id);
    WAITFOR(gw_ok);
    reply_handle.reply(gw_ok.value());
  }
  catch(std::exception& x) {
      LogException(x);
  }
}
```

“Holy Grail”+Exceptions:

```
bool CashierReactor::purchaseItem(
  int item_id, int connection_id) {

  int user_id =
        get_user_id(connection_id);



  try {
    bool db_ok = REENTRY
         dbPurchaseItem(db_reactor_id,
           user_id, item_id);

    if(!db_ok)
      return false;


    REACTORID gameworld_reactor_id =
      find_gameworld_for_user(user_id);
    bool gameworld_ok = REENTRY
      gameworldAddItem(
          gameworld_reactor_id,
          user_id, item_id);
    return gameworld_ok;
  }
  catch( std::exception& x ) {
    LogException(x);
  }
}
```

Take 7:

```
void CashierReactor::cashierPurchaseItem(
  shared_ptr<CashierPurchaseItemP  >>
    reply_handle, int item_id) {
  int user_id = get_user_id(re  dle);

  ReactorFuture<bool> db_ok(
  ReactorFuture<bool> gw_ok  ,

  try {
    db_ok = dbPurchaseI
            db_reactor
            user_id,  d);
    WAITFOR(db_ok);
    if(!db_ok.valu
      reply_handl  (false);
      return;
    }
    REACTORID  rld_reactor_id =
      find_g  d_for_user(user_id);
    gw_ok =  rldAddItem(
              orld_reactor_id,
              _id, item_id);
    WAI  _ok);
    re  dle.reply(gw_ok.value());
  }
  ca  d::exception& x) {
      Exception(x);
  }
}
```

Severe Problems Serialising

"Holy Grail"+Exceptions:

```
bool CashierReactor::purchaseItem(
  int item_id, int connection_id) {

  int user_id =
        get_user_id(connection_id);



  try {
    bool db_ok = REENTRY
          dbPurchaseItem(db_reactor_id,
            user_id, item_id);

    if(!db_ok)
      return false;


    REACTORID gameworld_reactor_id =
      find_gameworld_for_user(user_id);
    bool gameworld_ok = REENTRY
      gameworldAddItem(
          gameworld_reactor_id,
          user_id, item_id);
    return gameworld_ok;
  }
  catch( std::exception& x ) {
    LogException(x);
  }
}
```

Take 7x (DON'T DO IT):

```
void CashierReactor::cashierPurchaseItem(
  shared_ptr<CashierPurchaseItemReply>
    reply_handle, int item_id) {
  int user_id =
    get_user_id(reply_handle);

  try {
    bool db_ok = dbPurchaseItem(
            db_reactor_id,
            user_id, item_id);
   if(!db_ok.value()) {
      reply_handle.reply(false);
      return;
    }
    REACTORID gameworld_reactor_id =
      find_gameworld_for_user(user_id);
    gw_ok = gameworldAddItem(
            gameworld_reactor_id,
            user_id, item_id);
   reply_handle.reply(gw_ok.value());
  }
  catch(std::exception& x) {
      LogException(x);
  }
}
```

"Holy Grail"+Exceptions:

```
bool CashierReactor::purchaseItem(
  int item_id, int connection_id) {

  int user_id =
        get_user_id(connection_id);



  try {
    bool db_ok = REENTRY
          dbPurchaseItem(db_reactor_id,
            user_id, item_id);

    if(!db_ok)
      return false;


    REACTORID gameworld_reactor_id =
      find_gameworld_for_user(user_id);
    bool gameworld_ok = REENTRY
      gameworldAddItem(
          gameworld_reactor_id,
          user_id, item_id);
    return gameworld_ok;
  }
  catch( std::exception& x ) {
    LogException(x);
  }
}
```

Take 7x (DON'T DO IT):

```
void CashierReactor::cashi                    tem(
  shared_ptr<CashierPurch
    reply_handle, int it
  int user_id =
    get_user_id(reply

  try {
    bool db_ok = d
            db_re
            use
   if(!db_ok.v
      reply_h
      returr
    }
    REACT                    or_id =
      fi                     er(user_id);
    gw                       em(
                             ctor_id,
                             m_id);
                             gw_ok.value());

                             on& x) {
                             x);
}
```

"Hey, how we'll know those points where the state can be modified??"

"Holy Grail"+Exceptions:

```
bool CashierReactor::purchaseItem(
  int item_id, int connection_id) {

  int user_id =
        get_user_id(connection_id);



  try {
    bool db_ok = REENTRY
          dbPurchaseItem(db_reactor_id,
            user_id, item_id);

    if(!db_ok)
      return false;


    REACTORID gameworld_reactor_id =
      find_gameworld_for_user(user_id);
    bool gameworld_ok = REENTRY
      gameworldAddItem(
          gameworld_reactor_id,
          user_id, item_id);
    return gameworld_ok;
  }
  catch( std::exception& x ) {
    LogException(x);
  }
}
```

“Item Purchase” Example - Take 8. co_await.

Take 8:

```
void CashierReactor::cashierPurchaseItem(
  shared_ptr<CashierPurchaseItemReply>
    reply_handle, int item_id) {
  int user_id =
    get_user_id(reply_handle);

 try {
    bool db_ok = co_await dbPurchaseItem(
            db_reactor_id,
            user_id, item_id);
   if(!db_ok.value()) {
      reply_handle.reply(false);
      return;
    }
    REACTORID gameworld_reactor_id =
      find_gameworld_for_user(user_id);
    bool gw_ok = co_await
       gameworldAddItem(
            gameworld_reactor_id,
            user_id, item_id);
   reply_handle.reply(gw_ok.value());
  }
  catch(std::exception& x) {
      LogException(x);
  }
}
```

“Holy Grail”+Exceptions:

```
bool CashierReactor::purchaseItem(
  int item_id, int connection_id) {

  int user_id =
        get_user_id(connection_id);

 try {
    bool db_ok = REENTRY
         dbPurchaseItem(db_reactor_id,
           user_id, item_id);
     if(!db_ok)
      return false;


    REACTORID gameworld_reactor_id =
      find_gameworld_for_user(user_id);
    bool gameworld_ok = REENTRY
      gameworldAddItem(
          gameworld_reactor_id,
          user_id, item_id);
    return gameworld_ok;
  }
  catch( std::exception& x ) {
    LogException(x);
  }
}
```

“Item Purchase” Example - Take 8. co_await.

Take 8:

```
void CashierReactor::cashierPurchaseItem(
  shared_ptr<CashierPurchaseItemReply>
    reply_handle, int item_id) {
  int user_id =
    get_user_id(reply_handle);

 try {
    bool db_ok = co_awa      urchaseItem(
            db_reactor
            user_id,        d);
   if(!db_ok.value(
      reply_handle      ,false);
      return;
    }
    REACTORID      ld_reactor_id =
      find_ga      _for_user(user_id);
    bool gw_      _await
       gamew      dItem(
            ga      world_reactor_id,
            user_id, item_id);
   reply_handle.reply(gw_ok.value());
  }
  catch(std::exception& x) {
      LogException(x);
  }
}
```

Serialisation?

“Holy Grail”+Exceptions:

```
bool CashierReactor::purchaseItem(
  int item_id, int connection_id) {

  int user_id =
        get_user_id(connection_id);

 try {
    bool db_ok = REENTRY
         dbPurchaseItem(db_reactor_id,
           user_id, item_id);
     if(!db_ok)
      return false;


    REACTORID gameworld_reactor_id =
      find_gameworld_for_user(user_id);
    bool gameworld_ok = REENTRY
      gameworldAddItem(
          gameworld_reactor_id,
          user_id, item_id);
    return gameworld_ok;
  }
  catch( std::exception& x ) {
    LogException(x);
  }
}
```

Comparison

	Plain Messages	void RPCs	OO Callbacks	Lambda Pyramid	Futures	Code Builder	Stackful Coroutines	co_await
Take	1	2	3	4	5	6/6a	7/7x	8/8x
Prereq	C++98	C++98	C++98	C++11	C++11	C++11	boost::context, N3985	N4663
Verbosity	+600%	+400%	+300%	+20%	+30%	+50%/+10%	0	0
Readability	Very Poor	Very Poor	Poor	Poor	Acceptable	Acceptable/ Good	Good	Good
Hidden state changes	No	No	No	No	No	No	Nested-Only/ Yes	No
Serialisation	Easy	Easy	Easy	Doable but currently UGLY	Doable but currently UGLY	Doable but currently UGLY	No	MIGHT be doable

Comparison

	Plain Messages	void RPCs	OO Callbacks	Lambda Pyramid	Futures	Code Builder	Stackful Coroutines	co_await
Take	1	2	3	4	5	6/6a	7/7x	8/8x
Prereq	C++98	C++98	C++98	C++11	C++11	C++11	boost::context, N3985	C++2a
Verbosity	+600%	+400%	+300%	+20%	+30%	+50%/+10%	0	0
Readability	Very Poor	Very Poor	Poor	Poor	Acceptable	Acceptable/ Good	Good	Good
Hidden state changes	No	No	No	No	No	No	Nested-Only/ Yes	No
Serialisation	Easy	Easy	Easy	Doable but currently UGLY	Doable but currently UGLY	Doable but currently UGLY	No	MIGHT be doable

Part 2. Current Standard Proposals and Implementation Wishes

Current proposals:

- ***co_await***, *currently billed as "stackless coroutines" (formerly Resumable Functions); current proposal is N4663*
  - *probably the best one for our purposes (though see above re. potential to avoid REENTRY markers, and serialisation)*
- ***stackful coroutines***, *current proposal is N3985.*
  - *not too bad, but REENTRY is absent, and no idea how to implement cross-platform serialisation*
- ***Resumable Expressions*** *(P0114R0).*
  - *difficulties enforcing REENTRY-style markers*
  - *hidden mutex(!) when emulating co_await*
- ***Call/CC*** *(P0534R0):*
  - *IMO too low-level to be used directly at app-level*

Part 2. Current Standard Proposals and Implementation Wishes

Implementation Wishes:

- *we DO need to see those points where state can suddenly change*
  - *in this regard, I am a big fan of Suspend-Out model; please do NOT throw it away on the premises such as those in P0114R0.*
- *we DO need serialisation*
  - *as serialisation is not realistic for now, AT LEAST we need to (a) be sure that await-frames are using ONLY allocator, and (b) able to override default allocator for lambdas/await-frames/…*
  - *as soon as serialisation (via static reflection or whatever-else) is available - we need it for both lambdas and for await-frames*
- *we DO need coroutines to be thread-agnostic*
  - *no mutexes in implementation, PRETTY PLEASE*
    - *mutexes has been seen more than once to cause BAD bugs in WG21-related code*

P0114R0 emulating await:

```
void CashierReactor::cashierPurchaseItem(
  shared_ptr<CashierPurchaseItemReply>
    reply_handle, int item_id) {
  int user_id =
    get_user_id(reply_handle);

 try {
    bool db_ok = await(dbPurchaseItem(
            db_reactor_id,
            user_id, item_id));
   if(!db_ok.value()) {
      reply_handle.reply(false);
      return;
    }
    REACTORID gameworld_reactor_id =
      find_gameworld_for_user(user_id);
    bool gw_ok = await(
       gameworldAddItem(
            gameworld_reactor_id,
            user_id, item_id));
   reply_handle.reply(gw_ok.value());
  }
  catch(std::exception& x) {
      LogException(x);
  }
}
```

"Holy Grail"+Exceptions:

```
bool CashierReactor::purchaseItem(
  int item_id, int connection_id) {


  int user_id =
        get_user_id(connection_id);

  try {
    bool db_ok = REENTRY
          dbPurchaseItem(db_reactor_id,
            user_id, item_id);
```

up to 1M CPU cycles extra cost

<Really Big Ouch! />

```
      find_gameworld_for_user(user_id);
    bool gameworld_ok = REENTRY
      gameworldAddItem(
          gameworld_reactor_id
          user_id, item_i
    return gameworld_ok;
  }
  catch( std::exception& x ) {
    LogException(x);
  }
}
```

From P0114R0:

```
void  run() {
  struct state_saver {
    waiter* prev = active_waiter_;
    ~state_saver() {
       active_waiter_ = prev; }
   }  saver;

   active_waiter_ = this;
   std::lock_guard<std::mutex> lock(mutex_);
   nested_resumption_ = false;
   do_run();
}
```

MAY call f.resume() while mutex is locked

In turn, MAY lead to a deadlock on one single recursive mutex <Even Bigger Ouch! />

Disclaimers:

- *Code in P0114R0 is convoluted enough, so I might have misread it*
- *More importantly, mutex-related problems MIGHT be fixable (or MIGHT be not)*

TL;DR:

- *we (as in "quite a few developers out there, including, but not limited to, gamedevs, financial devs, and HPC devs") DO need a way to handle non-blocking returns*
  - *the whole point of handling non-blocking returns is to allow interaction with the current state.*
    - *as a result - we DO need a way to clearly see when the state has a potential to change (REENTRY marker).*

- *out of all the available and proposed options - none is perfect.*
  - *some options are ugly, some don't have this way-to-see-potential-to-change, and quite a few cause trouble when we're trying to serialise them.*

co_await is our best shot

Slides Are Available at github.com/CppCon/CppCon2017, cppcon2017.sched.com, and ithare.com

Questions?

Questions?

References:

- ‘No Bugs’ Hare, “Development&Deployment of Multiplayer Online Games”, Vol. II, pp. 70-129.
- Kevlin Henney, “Thinking Outside the Synchronisation Quadrant”, ACCU2017
- “Effective Go”, https://golang.org/doc/effective_go.html
- Dmitry Ligoum, Sergey Ignatchenko. Autom.cpp. https://github.com/O-Log-N/Autom.cpp
- N4463, N3985, P0114R0, P0534R0
- Chuanpeng Li, Chen Ding, Kai Shen, “Quantifying The Cost of Context Switch”, Proceedings of the 2007 workshop on Experimental computer science
- “STL Implementations and Thread Safety”, Sergey Ignatchenko, “C++ Report" , July/August 1998, Volume 10, Number 7.