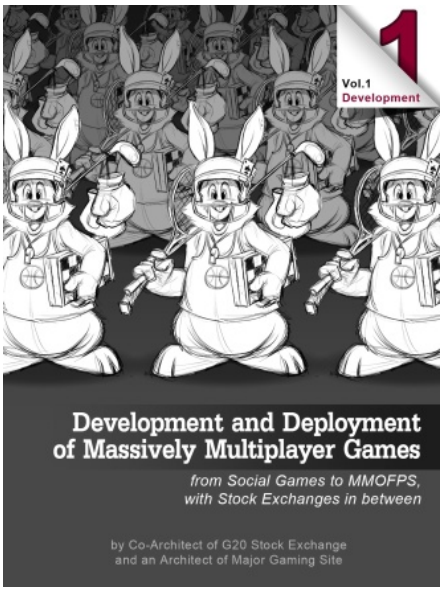
Asynchronous Processing for Finite State Machines/Actors: from plain event processing to Futures (with OO and Lambda Call Pyramids in between)

posted January 11, 2016 by "No Bugs" Hare, translated by Sergey Ignatchenko

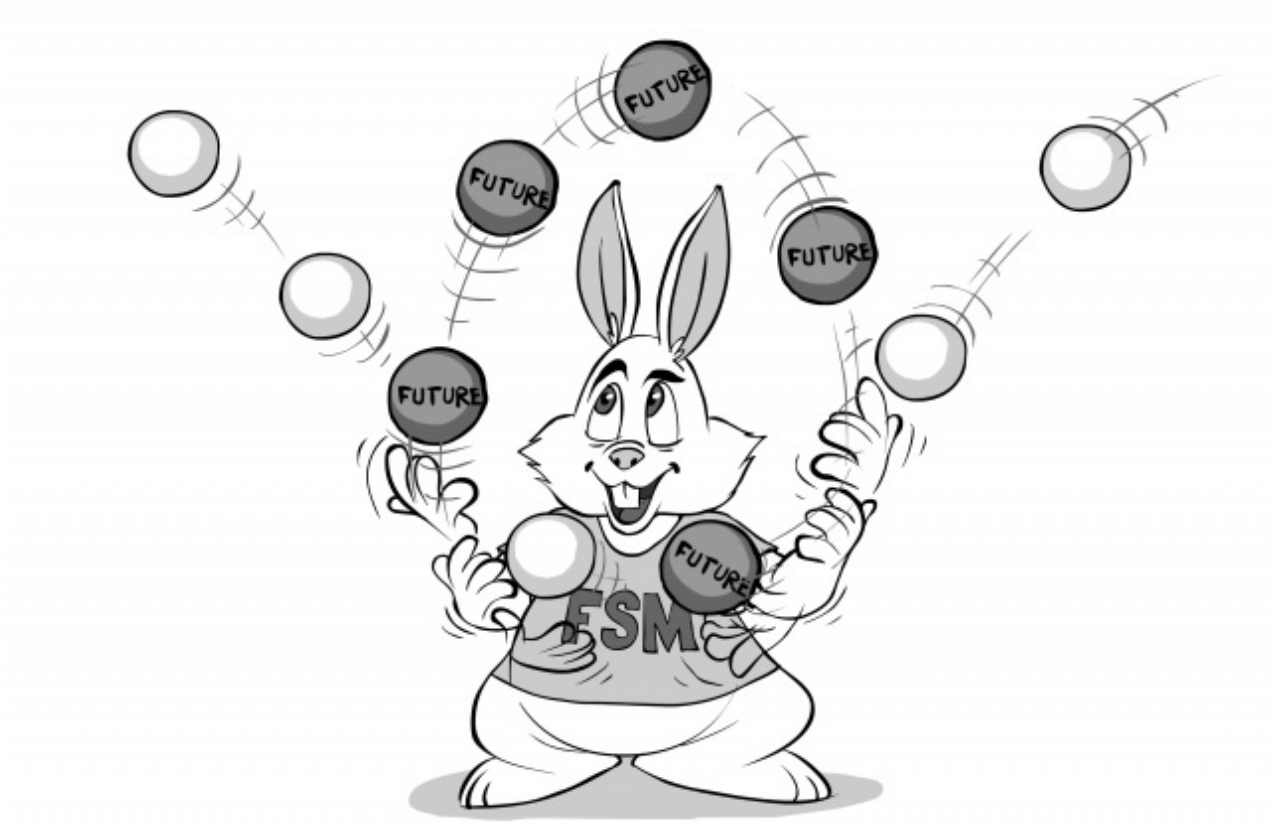


[[*This is Chapter VI(d) from the upcoming book "Development&Deployment of Massively Multiplayer Online Games", which is currently being beta-tested. Beta-testing is intended to improve the quality of the book, and provides free e-copy of the "release" book to those who help with improving; for further details see "Book Beta Testing". All the content published during Beta Testing, is subject to change before the book is published.*

To navigate through the book, you may want to use Development&Deployment of MMOG: Table of Contents.]]

[[I was planning the next part of Chapter VI to be about server-side programming languages, but have found that to speak about them, it would be better to describe a bit more about FSMs and an important part of them – futures and exception-related FSM-specific stuff. My apologies for this change in plans, and I hope that the part about server-side programming languages will be the next one]]

When programming Finite State Machines (FSMs, with Erlang/Akka-style Actors, or more generally – non-blocking event-driven programs, being very close) in a really non-blocking manner, two practical questions arise: "how to deal with communications with the other A in a non-blocking way", and "what to do with timed actions".



For the purposes of this section, we'll use C++ examples; however, leaving aside syntax, most of the reasoning here will also apply to any other modern programming language (with an obvious notion that the part on functional-style implementation will need support for lambdas); one obvious example is JavaScript as it is used in Node.js (more on it below).

Also, for the purpose of our examples, we assume that we have some kind of IDL compiler (more on in in Chapter VII), which takes function definitions and produces C++ stubs for them. The idea behind an IDL is to have all the inter-FSM communications defined in a special Interface Definition Language (see examples below), with an IDL compiler producing stubs (and relevant marshalling/unmarshalling code) for our programming language(s). IDL serves two important purposes: first, it eliminates silly-but-annoying bugs when manual marshalling is done differently by sender and receiver; second, it facilitates cross-language interactions.

IDL

Interface definition language (IDL) is a specification language used to describe a software component's application programming interface (API). IDLs describe an interface in a language-independent way, enabling

Take 1. Naïve Approach: Plain Events (will work, but is Plain Ugly)

Both inter-FSM communication and timed actions can be dealt with without any deviation from FSM/Actor model, via introducing yet another couple of input events. Let's say that we have a non-blocking RPC call from FSM A to a FSM B, which returns a value. RPC call translates into a message coming from

FSM A to FSM B (how it is delivered, is a different story, which will be discussed in Chapter [[TODO]]). FSM B gets this message as an input event, processes it, and sends another message to FSM A. FSM A gets this message as an input event, and performs some actions (which are FSM-specific, so FSM writer needs to specify them).

In a similar manner, whenever we're scheduling a timer, it is just a special timer event which will be delivered by FSM framework (="the code outside of FSM") to FSM more or less around requested time.

> **communication between software components that do not share one language**
>
> — *Wikipedia* —

First, let's consider a very simple example. Let's say our Game World FSM needs to report that our player has gained level, to DB (so that even if our Game World crashes, the player won't lose level, see "Containment of Game World server failures" section above for further discussion). In this case, our IDL may look as follows:

```
1  void dbLevelGained(int user_id, int level);
2    //ALL RPC calls are NON-BLOCKING!!
```

After this IDL is compiled, we may get something like:

```
1  //GENERATED FROM IDL, DO NOT MODIFY!
2  int dbLevelGained_send(FSMID fsm_id, int user_id, int level);
3    //sends a message to fsm_id
4    //returns request id
```

Then, calling code in FSM A may look like this:

```
1  dbLevelGained(db_fsm_id,user_id,level);
```

So far, so simple, with no apparent problems in sight. Now, let's see what happens in a more elaborated "item purchase" example. Let's say that we want to show player the list of items available for purchase (with items for which he has enough money on the account, highlighted), allow her to choose an item, get it through DB (which will deduct item price from player's account and add item to his DB inventory), and add the item to the game world.

Don't worry if you think that the code in Take 1 is ugly. It is. Skip to OO-based and function-based versions if this one affects your sensibilities

To do this, our IDL will look as follows:

```
int dbGetAccountBalance(int user_id);
list<StoreItem> dbGetStoreItems();
void dbBuyItemFromAccount(int user_id, ITEMID item);
  //MUST be a separate call to ensure data integrity without external locking,
  //  see "Containment of Game World server failures" subsection for discussion

int clientSelectItemToBuy(list<StoreItem>,int current_balance);
```

After this IDL is compiled, we may get something like:

```
//GENERATED FROM IDL, DO NOT MODIFY!
#define DB_GET_ACCOUNT_BALANCE 123
#define DB_GET_STORE_ITEMS 124
#define DB_BUY_ITEM_FROM_ACCOUNT 125
#define CLIENT_SELECT_ITEM_TO_BUY 126

int dbGetAccountBalance_send(FSMID fsm_id, int user_id);
  //sends a message, returns request_id
pair<bool,int> dbGetAccountBalance_recv(Event& ev, int request_id);
  //return.first indicates if incoming message matches request_id
int dbGetStoreItems_send(FSMID fsm_id);
pair<bool,list<StoreItem>> dbGetStoreItems_recv(Event& ev, int request_id);
int dbBuyItemFromAccount_send(FSMID fsm_id, int user_id, ITEMID item);
pair<bool,bool> dbBuyItemFromAccount_recv(Event& ev, int request_id);

int clientSelectItemToBuy_send(FSMID fsm_id, const list<StoreItem>& items,
  int current_balance);
pair<bool,int> clientSelectItemToBuy_recv(Event& ev, int request_id);
```

And, our code in FSM A will look like the following (this is where things start getting ugly):

```
//WARNING: SEVERELY UGLY CODE AHEAD!!
void MyFSM::process_event(Event& ev) {
  switch( ev.type ) {
    case SOME_OTHER_EVENT:
      //...
      //decided to make a call
      int request_id = dbGetAccountBalance_send(db_fsm_id,user_id);
      account_balance_requests.push(pair<int,int>(request_id,user_id));
        //account_balance_requests is a member of MyFSM
        //need it to account for multiple users requesting purchases
        //  at the same time
      //...
      break;

    case DB_GET_ACCOUNT_BALANCE:
      for(auto rq:account_balance_requests) {
        auto ok = dbGetAccountBalance_recv(ev, rq.first);
        if(ok.first) {
          int user_id = rq.second;
          int balance = ok.second;
          //got account balance, let's get list of items now
          int request_id2 = dbGetStoreItems_send(db_fsm_id);
          store_items_requests.push(
```

```
        pair<int,pair<int,int>>(request_id2,pair<int,int>(user_id,balance)));
       break;
      }
      MY_ASSERT(false,"Cannot happen");
       //throws an exception, more on MY_ASSERT in Chapter[[TODO]]
     }
     break;

   case DB_GET_STORE_ITEMS:
     for(auto rq:store_item_requests) {
      auto ok = dbGetStoreItems_recv(ev, rq.first);
      if(ok.first) {
       pair<int,int> user_id_and_balance = rq.second;
       list<StoreItem>& items = ok.second;
       //got everything client needs, let's send it to client now
       int request_id3 = clientSelectItemToBuy_send(user_fsm_id,
        items,user_id_and_balance.second);
       client_select_items_to_buy_requests.push(
        pair<int,int>(request_id,user_id));
       break;
      }
      MY_ASSERT(false,"Cannot happen");
       //throws an exception, more on MY_ASSERT in Chapter[[TODO]]
     }
     break;

   case CLIENT_SELECT_ITEM_TO_BUY:
     for(auto rq:store_item_requests) {
      auto ok = clientSelectItemsToBuy_recv(ev, rq.first);
      if(ok.first) {
       int user_id = rq.second;
       ITEMID selected_item = ok.second;
       //got client selection, let's try buying now
       int request_id4 = dbBuyItemFromAccount_send(db_fsm_id,
        user_id,selected_item);
       buy_item_requests.push(pair<int,pair<int,ITEMID>>(
        request_id,pair<int,int>(user_id,selected_item)));
       break;
      }
      MY_ASSERT(false,"Cannot happen");
       //throws an exception, more on MY_ASSERT in Chapter[[TODO]]
     }
     break;

   case DB_BUY_ITEM_FROM_ACCOUNT:
     for(auto rq:store_item_requests) {
      auto ok = dbBuyItemFromAccount_recv(ev, rq.first);
      if(ok.first) {
       pair<int,ITEMID> user_id_and_item = rq.second;
       bool item_ok = ok.second;
       //got DB confirmation, let's modify our game world now
       players[user_id].addItem(user_id_and_item.second);
       //phew
       break;
      }
      MY_ASSERT(false,"Cannot happen");
```

```
79          MY_ASSERT(false, "Cannot happen");
80            //throws an exception, more on MY_ASSERT in Chapter[[TODO]]
81          }
82          break;
83        }
84      }
```

If you feel that this code has been beaten with an ugly stick – that's because it is

Over 60 lines of code with only about 5 being meaningful (and the rest being boilerplate stuff) is pretty bad. Not only it takes a lot of keystrokes to write, but it is even worse to read (what really is going on is completely hidden within those tons of boilerplate code). And it is very error-prone too, making maintenance a nightmare. If such a thing happens once for all your 1e6-LOC game – that's ok, but you will need these things much more than once. Let's see what can we do to improve it.

LOC

Lines of Code is a software metric used to measure the size of a computer program by counting the number of lines in the text of the program's source code

— Wikipedia —

Take 2. OO-Style: Less Error-Prone, but Still Unreadable

In OO-style, we will create a Callback class, will register it with our FSM, and then it will be FSM framework ("the code outside of FSMs") dealing with most of the mechanics within. Rewriting our "item purchase" example int OO-style will change the whole thing drastically. While IDL will be the same, both generated code and calling code will look very differently. For OO-style asynchronous calls, stub code generated from IDL may look as follows:

```
1   //GENERATED FROM IDL, DO NOT MODIFY!
2   void dbGetAccountBalance_send(FSM* fsm, /* new */ Callback* cb,
3     FSMID target_fsm_id, int user_id);
4     //sends a message, calls cb->process_callback() when done
5   int dbGetAccountBalance_parsereply(Event& ev);
6
7   void dbGetStoreItems_send(FSM* fsm, /* new */ Callback* cb, FSMID target_fsm_id);
8   list<StoreItem> dbGetStoreItems_parsereply(Event& ev);
9   void dbBuyItemFromAccount_send(FSM* fsm, /* new */ Callback* cb,
10    FSMID target_fsm_id, int user_id, ITEMID item);
11  bool dbBuyItemFromAccount_parsereply(Event& ev);
12  void clientSelectItemToBuy_send(FSM* fsm, /* new */ Callback* cb,
13    FSMID target_fsm_id, const list<StoreItem>& items, int current_balance);
14  ITEMID clientSelectItemToBuy_parsereply(Event& ev);
```

And our calling code may look as follows:

```
1   //LESS ERROR-PRONE THAN TAKE 1, BUT STILL UNREADABLE
2   //TO BE AVOIDED IF YOUR COMPILER SUPPORTS LAMBDAS
3   class BuyItemFromAccountCallback : public Callback {
```

```
class BuyItemFromAccountCallback : public Callback {
  private:
  MyFSM* fsm;
  int user_id;
  ITEMID item;

  public:
  BuyItemFromAccountCallback(MyFSM* fsm_,int user_id_, ITEMID item_)
  : fsm(fsm_),user_id(user_id_), item(item_)
  {
  }
  void process_callback(Event& ev) override {
    bool ok = dbBuyItemFromAccount_parsereply(ev);
    if(ok)
      fsm->players[user_id].addItem(user_id_and_item.second);
  }
};
class ClientSelectItemToBuyCallback : public Callback {
  private:
  MyFSM* fsm;
  int user_id;

  public:
  ClientSelectItemToBuyCallback(MyFSM* fsm_,int user_id_)
  : fsm(fsm_),user_id(user_id_)
  {
  }
  void process_callback(Event& ev) override {
   ITEMID item = clientSelectItemToBuy_parsereply(ev);
   dbBuyItemFromAccount_send(fsm,
     new BuyItemFromAccountCallback(fsm,user_id,item),
     fsm->getDbFsmId(), user_id, item);
  }
};
class GetStoreItemsCallback : public Callback {
  private:
  MyFSM* fsm;
  int user_id;
  int balance;

  public:
  GetStoreItemsCallback(MyFSM* fsm_,int user_id_, int balance_)
  : fsm(fsm_),user_id(user_id_), balance(balance_)
  {
  }
  void process_callback(Event& ev) override {
    list<StoreItem> items = dbGetStoreItems_parsereply(ev);
    clientSelectItemToBuy_send(fsm,
      new ClientSelectItemToBuyCallback(fsm, user_id),
      fsm->getClientFsmId(user_id), items, balance);
  }
};

class GetAccountBalanceCallback : public Callback {
  private:
  MyFSM* fsm;
  int user_id;
```

```
59      int user_id;
60
61    public:
62    GetAccountBalanceCallback(MyFSM* fsm_,int user_id_)
63    : fsm(fsm_), user_id( user_id_ )
64    {
65    }
66    void process_callback(Event& ev) override {
67      int balance = dbGetAccountBalance_parsereply(ev);
68      dbGetStoreItems_send(fsm,
69        new GetStoreItemsCallback(fsm,user_id, balance), fsm->getDbFsmId() );
70    }
71  };
72
73  void MyFSM::process_event(Event& ev){
74    switch( ev.type ) {
75      case SOME_OTHER_EVENT:
76        //...
77        //decided to make a call
78        dbGetAccountBalance_send(this,
79          new GetAccountBalanceCallback(this, user_id), db_fsm_id, user_id);
80        //...
81         break;
82    }
83  }
```

This one is less error-prone than the code in Take 1, but is still very verbose, and poorly readable. For each meaningful line of code there is still 10+ lines of boilerplate stuff (though it is easier to parse it out while reading, than for Naïve one).

In [Facebook] it is named “callback hell”. Well, I wouldn’t be that categoric (after all, there was life before 2011), but yes – it is indeed very annoying (and poorly manageable). If you don’t have anything better than this – you might need to use this kind of stuff, but if your language supports lambdas, the very same thing can be written in a much more manageable manner.

Take 3. Lambda Continuations to the rescue! Callback Pyramid

As soon as we get lambda functions (i.e. more or less since C++11), the whole thing becomes much easier to write down. First of all, we could simply replace our classes with lambda functions. In this case, code generated from the very same IDL, may look as follows:

```
//GENERATED FROM IDL, DO NOT MODIFY!
void dbGetAccountBalance(FSM* fsm, FSMID target_fsm_id, int user_id,
  std::function<void(int)> cb);
  //sends a message, calls cb when done

void dbGetStoreItems(FSM* fsm, FSMID target_fsm_id,
  std::function<void(const list<StoreItem>&)> cb);
void dbBuyItemFromAccount(FSM* fsm, FSMID target_fsm_id, int user_id, ITEMID item
  std::function<void(book ok)> cb);
void clientSelectItemToBuy(FSM* fsm, FSMID target_fsm_id,
  const list<StoreItem>& items, int current_balance,
  std::function<void(ITEMID item)> cb);
```

And calling code might look as follows:

```
//inside MyFSM::process_event():
//...
//decided to make a call
dbGetAccountBalance(this,db_fsm_id,user_id,
  [=](int balance) {
    //this lambda is a close cousin of
    //  Take2::GetAccountBalanceCallback
    //  You may think of lambda object created at this point,
    //    as of Take2::GetAccountBalanceCallback
    //    automagically created for you
    dbGetStoreItems(this,db_fsm_id,
      [=](const list<StoreItem>& items) {
        //this lambda is a close cousin of
        // Take2::GetStoreItemsCallback
        clientSelectItemToBuy(this,user_fsm_id,items,balance,
            //here, 'this', 'user_fsm_id', and 'balance' are 'captured'
            //  from the code above
          [=](ITEMID item) {
            //this lambda is a close cousin of
            //  Take2::ClientSelectItemToBuyCallback
            dbBuyItemFromAccount(this,db_fsm_id,user_id,item_id,
              [=](bool ok) {
                //this lambda is a close cousin of
                //  Take2::BuyItemFromAccountCallback
                if(ok) {
                  players[user_id].addItem(item_id);
                }
              }
            );
          }
        );
      }
    );
  }
);
```

Compared to our previous attempts, such a “callback pyramid” is indeed a big relief. Instead of previously observed 50+ lines of code for meaningful 5 or so (with meaningful ones scattered around), here we have just about 2 lines of overhead per

each meaningful line (instead of previous 10(!)), and also all our meaningful lines of code are nicely gathered in one place (and in their right order too). Phew. With all my dislike to using lambdas just for the sake of your code being "cool" and functional, this is one case when using lambdas makes very obvious sense (despite the syntax looking quite weird).

In fact, this code is very close to the way Node.js programs handle asynchronous calls. Actually, as it was mentioned in Chapter V [[TODO!: mention it there]] the whole task we're facing with our QnFSMs (which is "event-driven programming with a completely non-blocking API") is almost exactly the same as the one for Node.js, so there is no wonder that the methods we're using, are similar.

On Continuations

"Don't even think of converting *all* of your code to a so-called Continuation-Passing-Style

Those lambdas we're using here, are known as "continuations". In general, "continuation" is a thing, which says what we should do when we reach certain point within our logical flow. To make our FSMs (and Node.js) work – continuations are the only feasible way to do it (in fact, our Take 1 and Take 2 also implemented continuations, albeit in an unusual way).

However, don't even think of converting *all* of your code to a so-called Continuation-Passing-Style[1] (the one with an explicit prohibition for any function to return any value, instead each and every function taking additional function parameter to be called with would-be return value). Full conversion to continuation-passing-style will make your code significantly less readable, and will hit your performance too. Think of our "callback pyramid" code above not as a final proof of lambdas being the-utlimate-solution-to-all-your-problems, but as of a useful pattern, which can be used to simplify coding *in this specific scenario*.

Exceptions

Now, as we got rid of those ugly Take 1 and Take 2 (where any additional complexity would make them absolutely incomprehensible), we can start thinking about adding exceptions to our code. Indeed, we can add exceptions to the "callback pyramid", by adding (to each of RPC stubs and each of the lambdas) another lambda parameter to handle exceptions (corresponding to usual 'catch' statement). Keep in mind that to provide usual try-catch semantics (with topmost-function exception handler catching all the stuff on all the levels), we need to pass this 'catch' lambda downstream:

```
dbGetAccountBalance(this,db_fsm_id,user_id,
 [=](int balance, std::function<void(std::exception&)> catc) {
  dbGetStoreItems(this,db_fsm_id,
   [=](const list<StoreItem>& items, std::function<void(std::exception&)> catc) {
    clientSelectItemToBuy(this,user_fsm_id,items,balance,
     [=](ITEMID item, std::function<void(std::exception&)> catc) {
      dbBuyItemFromAccount(this,db_fsm_id,user_id,item_id,
       [=](bool ok, std::function<void(std::exception&)> catc) {
        if(ok) {
         players[user_id].addItem(item_id);
        }
       }
      ,catc);
     }
    ,catc);
   }
  ,catc);
 },
 [=](std::exception&) {//'catch'
  //do something
 }
);
```

As we can see, while handling exceptions with 'callback pyramid' is possible, it certainly adds to boilerplate code, and also starts to lead us towards the Ugly Land ☹ .

Limitations

For the 'callback pyramid' above, I see two substantial limitations. The first one is that adding exceptions, while possible, adds to code ugliness and impedes readability (see example above).

The second limitation is that with 'callback pyramid' it is not easy to express the concept of "wait for more than one thing to complete" , which leads to unnecessary sequencing, adding to latencies (which may or may not be a problem for your purposes, but still a thing to keep in mind).

On the other hand, as soon as we have lambdas, we can make another attempt to write our asynchronous code, and to obtain the code which is free from these two limitations.

> "with 'callback pyramid' it is not easy to express the concept of 'wait for more than one thing to complete' , which leads to unnecessary sequencing, adding to

---

[1] which is the first thing Google throws at you when you're typing in "node.js continuation"

Take 4. Futures

While lambda-based 'callback pyramid' version is indeed a Big Fat Improvement over our first two takes, let's see if we can improve it further. Here, we will use a concept known as "futures" (our FSMFuture is similar in concept, but different in implementation, from std::future, boost::future, and folly::Future, see "Similarities and Differences" section below for discussion of differences between the these). In our interpretation, "future" is a value which is already requested, but not obtained yet. With such "futures", IDL-generated code for the very same "item purchase" example, may look as follows :

latencies (which may or may not be a problem for your purposes, but still a thing to keep in mind).

```
FSMFuture<int> dbGetAccountBalance(FSM* fsm, FSMID db_fsm_id, int user_id);
FSMFuture<list<StoreItem>> dbGetStoreItems(FSM* fsm, FSMID db_fsm_id);
FSMFuture<void> dbBuyFromAccount(FSM* fsm, FSMID db_fsm_id,
  int user_id, ITEMID item);

FSMFuture<ITEMID> clientSelectItemToBuy(FSM* fsm, FSMID client_fsm_id,
  list<StoreItem>, int current_balance);
```

And the calling code will look along the lines of:

```
//inside MyFSM::process_event():
//...
//decided to make a call
FSMFuture<int> balance = dbGetAccountBalance(this, db_fsm_id ,user_id);
  //sends non-blocking RPC request
FSMFuture<list<StoreItem>> items = dbGetStoreItems(this, db_fsm_id);
  //sends non-blocking RPC request

// all further calls don't normally do anything right away,
//   just declaring future actions
//   to be performed when the results are ready

//declare that we want to wait for both non-blocking RPC calls to complete
FSMFutureBoth<int,list<StoreItem>> balance_and_items(this, balance, items);

//declare what we will do when both balance and items are ready
FSMFuture<ITEMID> clientSelection = balance_and_items.then(
  [=]() {
    return clientSelectItemToBuy(this, user_fsm_id,
      balance_and_items.secondValue(), balance_and_items.firstValue());
  }
).exception(
    //NOTE that when we're attaching exception handler to a future,
    //  FSMFuture implementation can also apply it
    //    to all the futures 'downstream'
    //    unless it is explicitly overridden
  [=]() {
    //handle exception
  }
);

//declare what we will do when
FSMFuture<bool> purchase_ok = clientSelection.then(
  [=]() {
    return dbBuyFromAccount(this, db_fsm_id, user_id, clientSelection.value());
  }
);

purchase_ok.then(
  [=]() {
    players[user_id].addItem(clientSelection.value());
  }
);
```

While being a bit more verbose than lambda-based “call pyramid” version, at least for me personally it is more straightforward and more readable. Also, as a side bonus, it allows to describe scenarios when you need two things to continue your calculations (in our example – results of dbGetAccountBalance() and dbGetStoreItems()) quite easily, and without unnecessary sequencing which was present in all our previous versions. In other words, the future-based version as written above, will issue two first non-blocking RPC requests in parallel, and then will wait for both of them before proceeding further (opposed to all previous versions issuing the same calls sequentially and unnecessary losing on latency). While writing the same parallel logic within the previous takes is possible (except maybe for Take 3), it would result in a

code which is too ugly to deal with and maintain at application level; with futures, it is much more straightforward and obvious.

Similarities and Differences

All the different takes are similar

It should be noted that for our "item purchase" example (and actually any other sequence-of-calls scenario), all our versions are very similar to each other, with most of the differences being about "syntactic sugar". On the other hand, when faced with code from Take 1, and equivalent one from Take 4, I would certainly prefer the latter one 🙂 .

> **"Performance-wise, the differences between different code versions discussed above will be negligible for pretty much any conceivable scenario.**

Performance-wise, the differences between different code versions discussed above will be negligible for pretty much any conceivable scenario. Consistently with Erlang/Akka/Node.js approaches, our unit of processing is always a message/event. Events as such roughly correspond to context switches, and context switches are quite expensive beasts (for x64 – very roughly of the order of 10'000 CPU clocks, that is, if we account for cache reloads, YMMV, batteries not included).[2] So, even if we're using our non-blocking RPCs to off-load some calculations to different threads (and not for inter-server communications, where the costs are obviously higher), the costs of each message/event processing are quite high, and things such as dynamic dispatching or even dynamic allocations won't be large enough to produce any visible performance difference.

Differences from std::future etc.

Traditionally, discussions about asynchronous processing are made in the context of "Off-loading" some calculations into a different thread, doing some things in parallel, and waiting for the result (at the point where it becomes necessary to calculate things further). This becomes particularly obvious when looking at std::future: among other things, it has get() method, which waits until the future result is ready (the same goes for boost::future, and folly::Futures have wait() method which does pretty much the same thing). As our FSMs in QnFSM model are completely non-blocking, we are not allowed to have things such as std::future::get() or folly::Future::wait().

Whenever an std::future (or folly::Future) completes computation, it reports back to original future object via some kind of inter-thread notification. Also for this kind of futures, the code in callbacks/continuations MAY (and usually will) be called from a different thread, which means that callbacks are normally not allowed to interact with the main thread (except for setting a value within the future).

Similarities to Node.js

In contrast, our asynchronous processing is based on the premise that whenever a future is available, it is delivered as a yet another message to FSM::process_event(). It stands for all our four different versions of the code (with the differences, while important practically, being more of syntactic nature). As a consequence, all versions of our code guarantee that all our callbacks (whether lambda or not) will always be called from the same thread,[3] which means that

we are allowed to use FSM object and all it's fields from all our callbacks (lambdas or not) and without any thread synchronization[4]

It allows to handle much more sophisticated scenarios than that of linear calculation/execution. For example, if in our "item purchase" example there is a per-world limit of number of items of certain type, we MAY add the check for number of items which are already present within our world, into processing of clientSelectItemToBuy reply, to guarantee that the limit is not exceeded. If there is only one item left, but there are many clients willing it, all of them will be allowed to go up until to clientSelectItemToBuy, but those who waited too long there, will get an error message at the point after clientSelectItemToBuy returns. All this will happen without any thread synchronization.

"All of this will happen without any thread synchronization.

From this point of view, our futures (as well as the other our Takes on the asynchronous communications) are more similar to Node.js approach (where all the callbacks are essentially made within the same thread, so no thread synchronizations issues arise).

Alternatively, we can see Take 3 and Take 4 as a quite special case of coroutines/fibers. In such interpretation, we can say that there is an implicit coroutine "yield" before each RPC call in a chain, which allows other messages to be processed while we're waiting for the reply. Still, when a reply comes back, we're back in the same context and in the same thread where we were before this implicit "yield" point.

---

[2] context switches being that expensive is one reason why off-loading micro-operations to other threads doesn't work (in other words: don't try to off-load lone "int a+ int b" to a different thread, it won't do any good)

[3] strictly speaking, in some fairly unusual deployment scenarios there can be exceptions to this rule, but no-synchronization needed claim always stands

[4] which is why we have significant simplification for our lambda version compared to the one in [Facebook]

On serializable lambdas in C++

To have all the FSM goodies (like production post-mortem etc.), we need to be able to serialize those captured values within lambdas (this also applies to FSMs). For most of the languages out there, pretty much everything is serializable, including lambda objects, but for C++, serializing lambda captured values is not easy 🙁 .

The best way of doing it which I currently know, is the following:

- write and debug the code written as in the examples above. It won't give you things such as production post-mortem, or full FSM serialization, but they're rarely needed at this point in development (if necessary, you can always go via production route described below, to get them)
- add prefix such as SERIALIZABLELAMBDA before each such lambda; define it to an empty string (alternatively, you may use specially formatted comment, but I prefer empty define as more explicit)
- have your own pre-processor which takes all these SERIALIZABLELAMBDAs and generates code similar to that of in Take 2, with all the generated classes implementing whatever-serialization-you-prefer (and all the generated classes derived from some base *class SerializableLambda* or something). Complexity of this pre-processor will depend on the amount of information you provide in your SERIALIZABLELAMBDA macro:
  - if you write it as SERIALIZABLELAMBDA(int i, string s), specifying all the captured variables with their types once again, then your pre-processor becomes trivial
  - if you want to write it as SERIALIZABLELAMBDA w/o parameters, it is still possible, but deriving those captured parameters and their types can be not too trivial
  - which way to go, is up to you, both will work
- in production mode, run this pre-processor before compiling
- in production mode, make sure that RPC functions don't accept std::function (accepting *class SerializableLambda* instead), so that if you forget to specify SERIALIZABLELAMBDA, your code won't compile (which is better than if it compiles, and fails only in runtime)

TL;DR for Asynchronous Communications in FSMs

- We've discussed in detail asynchronous RPC calls, but handling of timer-related messages can be implemented in a very similar way
- As our FSMs are non-blocking, being asynchronous becomes the law (exactly as for Node.js)
- You will need IDL (and IDL compiler) one way or another (more on it in Chapter [[TODO]])

- Ways of handling asynchronous stuff in FSMs are well-known, but are quite ugly (see Take 1 and Take 2)
- With introduction of lambdas, it became much better and simpler to write and understand (see Take 3 and Take 4)
- Futures can be seen as an improvement over "call pyramid" use of lambdas (which is consistent with findings in [Facebook])
    - in particular, it simplifies handling of "wait-for-multiple-results-before-proceeding" scenarios
    - FSM futures, while having the concept which is similar to std::future and folly:Future, are not identical to them
        - in particular, FSM futures allow interaction with FSM state from callbacks without any thread synchronization
- To get all FSM goodies in C++, you'll need to implement serializing lambdas, see details above

FSMs and Exceptions

One more FSM-related issue which was uncovered until now, is related to subtle relations between FSMs and exceptions. Once again, most of our discussion (except for the part marked "C++-specific") will apply to most programming languages, but examples will be given in C++.

Validate-Calculate-Modify Pattern

One very important practical pattern for FSMs, is Validate-Calculate-Modify. The idea behind is that most of the time, when processing incoming event/message within our FSM, we need to do the following three things:

- **Validate.** check that the incoming event/message is valid
- **Calculate.** calculate changes which need to be made to the state of our FSM
- **Modify.** Apply those calculated changes.

"Effects of any exception happening before Modify stage are trivial: as we didn't modify anything, any exception will

This pattern has quite a few useful applications; however, the most important uses are closely related to exceptions. As long as we don't modify state of our FSM within Validate and Calculate stages, effects of any exception happening before Modify stage are trivial: as we didn't modify anything, any exception will lead merely to ignoring incoming message (without any need to rollback any changes, as there were none; handling of on-stack allocations depends on the programming language and is discussed below), which exactly what is necessary most of the time (and this has some other interesting uses, see "Exception-based Determinism" section below). And Modify stage is usually

trivial enough to avoid vast majority of the exceptions.

Enforcing const-ness for Validate-Calculate-Modify (C++-specific)

lead merely to ignoring incoming message, without any need to rollback any changes, as there were none

To rely on "no-rollback-necessary" exception property within Validate-Calculate-Modify pattern, it is important to enforce immutability of FSM state before Modify stage. And as it was noted in [[GDC2015 – TODO!]], no rule is good if it is not enforced. Fortunately, at least in C++ we can enforce immutability relatively easily (that is, for reasonable and non-malicious developers). But first, let's define our task. We want to be able to enforce const-ness along the following lines:

```
1  void MyFSM::process_event(Event& ev) {
2    ///VALIDATE: 'this' is const
3    //validating code
4
5    //CALCULATE: 'this' is still const
6    //calculating code
7
8    //MODIFY: 'this' is no longer const
9    //modifying code
10 }
```

To make it work this way, for C++ I suggest the following (reasonably dirty) trick:

```
1  void MyFSM::process_event(Event& ev) const {
2    //yes, process_event() is declared const(!)
3
4    //VALIDATE: 'this' is enforced const
5    //validating code
6    //CALCULATE: 'this' is still enforced const
7    //calculate code
8
9    //MODIFY:
10   MyFSM* fsm = modify_stage_fsm();
11     //modify_stage_fsm() returns const_cast<MyFSM*>(this)
12
13   //modifying code
14   //  uses 'fsm' which is non-const
15 }
```

"Depending on

While not 100% neat, it does the trick, and prevents from accidental writing to FSM state before modify_stage_fsm() is called (as compiler will notice modifying const *this* pointer, and will issue an error). Of course, one can call modify_stage_fsm() at the very beginning of the process_event() negating all the protection (or use one of several dozens another ways to bypass const-ness), but we're assuming that you do want to benefit from such a split, and will honestly avoid bypassing protection as long

as it is possible.

Note that depending on your game and FSM framework, post_message() function (the one which posts messages to other FSMs) may be implemented either as as a non-const function (then you'll need to call it only after modify_stage_fsm()), or as a const function (and then it can be called before modify_stage_fsm()). To achieve the latter, your FSM framework need to buffer all the messages which were intended to be sent via post_message() (NOT actually sending them), and to post them after the process_event() function successfully returns (silently dropping them in case of exception).

your game and FSM framework, post_message() function may be implemented either as as a non-const function (then you'll need to call it only after modify_stage_fsm or as a const function (and then it can be called before modify_stage_fsm

Now to the goodies coming out of such separation.

Exceptions before Modification Stage are Safe, including CPU exceptions

There are certain classes of bugs in your code which are very difficult to test, but which do occasionally happen. Some of them are leading to situations-which-should-never-happen (MYASSERTs throwing exception, see Chapter [[TODO]] for further discussion), or even to CPU exceptions (dereferencing NULL pointer and division-by-zero being all-time favourites).

If you're following the Validate-Calculate-Modify pattern, then all such exceptions (that is, if you can convert CPU exception into your-language-exception, see Chapter [[TODO]] for details for C++) become safe, in a sense that offending packet is merely thrown away, and your system is still in a valid state, ready to process the next incoming message. Yes, in extreme cases it may lead certain parts of your system to hang, but in practice most of the time the impact is very limited (it is much better to have a crazy client to hang, than your whole game world to hang, to terminate, or to end up in an inconsistent state).

This resilience to occasional exceptions has been observed to be THAT important in practice, that I think it alone is sufficient to jump through the hoops above, enforcing clean separation along Validate-Calculate-Modify lines.

Exception-based Determinism

One of the ways to achieve determinism which was mentioned in Chapter V with description postponed until later, is exception-based determinism.

Let's consider the following scenario: your FSM MIGHT need some non-determinstic data, but chances for it happening are fairly slim, and requesting it for each call to process_event() would be a waste. One example of such a thing is random data from physical RNG. Instead of resorting to "call interception" (which is not the cleanest method available, and also won't work well if your RNG source is slow or on a different machine), you MAY implement determinism via exceptions. It would work along the following lines:

- RNG_data becomes one of the parameters to process_event(), but is normally empty.
  - Alternatively, you MAY put it alongside with current_time to TLS, see Chapter V for details
- if, by any chance, you find out that you need RNG_data during your CALCULATE stage with RNG_data being empty – you throw a special exception NeedRNGData
  - as your VALIDATE and CALCULATE stages didn't change FSM state, there is nothing to rollback within the state
  - on-stack variable handling will be different for C++ and garbage-collected languages:
    - for C++, as long as you're always using RAII/std::unique_ptr<> for all on-stack resources (which you should for C++ anyway), all such objects will be rolled back automagically without any additional effort from your side
    - for garbage-collected languages, all on-stack objects will be cleaned by garbage collector

RAII

Resource Acquisition Is Initialization is a programming idiom used in several object-oriented languages, most prominently C++, but also D, Ada, Vala, and Rust.

— Wikipedia —

- on receiving such an exception, the framework outside of FSM will obtain RNG_data, and then will call MyFSM::process_event() once again, this time providing non-empty RNG_data
- this time, your code will go along exactly the same lines until you're trying to use RNG_data, but as you already have non-empty RNG_data, you will be able to proceed further this time.

Bingo! You have your determinism in a clean way, without "call interception" (and all because of clean separation between Validation-Calculation-Modification).

FSM Exception Summary

To summarize my main points about FSM and exceptions:

- Validate-Calculate-Modify is a pattern which simplifies life after deployment significantly (while it is not MUST-have, it is very-nice-to-have)
  - if you're following it, enforcing it is a Good Thing(tm)

- Following it will allow you to safely ignore quite a few things-you-forgot-about without crashing (don't overrely on it though, it is not a silver bullet)
- It also allows to achieve determinism without "call interception" via using exception-based Determinism in some practically important cases
- What are you waiting for? Do It! 😉

[[To Be Continued…

This concludes beta Chapter VI(d) from the upcoming book "Development and Deployment of Massively Multiplayer Games (from social games to MMOFPS, with social games in between)". Stay tuned for beta Chapter VI(d), "Modular Architecture: Server-Side. Programming Languages.]]

[–] References

[Facebook] Hans Fugal, "Futures for C++11 at Facebook"

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« *MMOG Server-Side. Eternal Linux-vs-Windows Debate*

MMOG Server-Side. Programming Languages »

Filed Under: Distributed Systems, Network Programming, Programming, System Architecture
Tagged With: asynchronous, finite state machine, game, multi-player