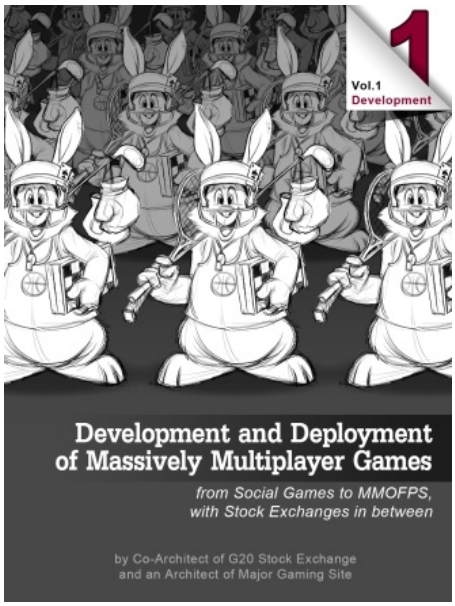
MMOG Server-Side. Programming Languages

posted January 18, 2016 by "No Bugs" Hare, translated by Sergey Ignatchenko

[[*This is Chapter VI(e) 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.*

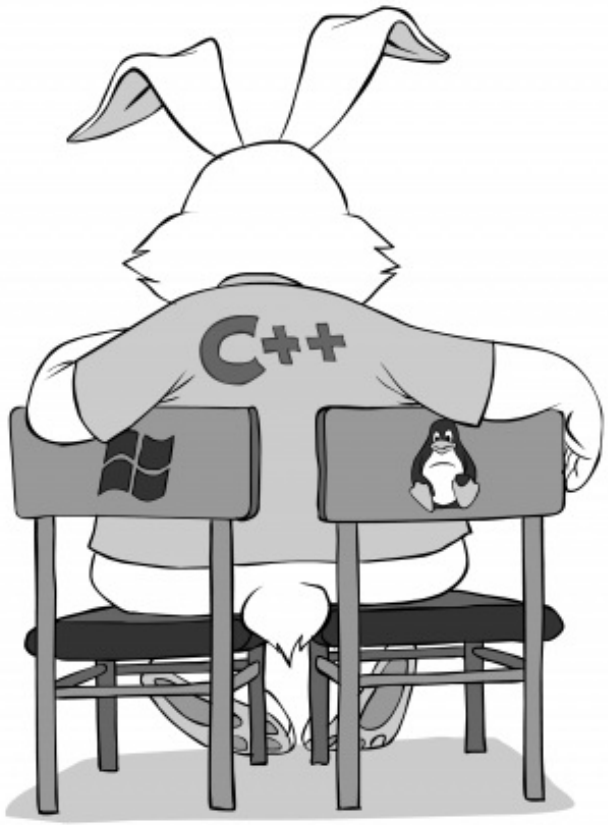
To navigate through the book, you may want to use Development&Deployment of MMOG: Table of Contents.]]



Going Cross-Platform

In one of the previous sections we've discussed choosing a platform for your MMOG servers. However, one of the first things I've noted was that you should certainly consider developing cross-platform code. In fact, this is what I am usually doing (that is, if I can get past management, which is usually supported by a bunch of fellow developers who neither know, nor don't want to learn anything but their-favorite thing). But let's see what going cross-platform means from the programming languages point of view.

Cross-platform C++



Actually, my personal favorite for cross-platform development, is cross-platform C++. To those having any doubts: yes, C++ can be made cross-platform, I've done it myself on numerous occasions. It works even better when you have your code restricted to event-driven side-effect-free processing (a.k.a. deterministic finite-state-machines (FSMs), see Chapter V for details). For our current discussion, one thing is important about FSMs: as soon as your FSM becomes deterministic, it doesn't really have any significant interaction with the system, so it is "pure logic" (a.k.a. "moving bits around", and is pretty much like "pure" functions from functional programming). And "pure logic" is inherently cross-platform (that is, as long as you keep it "pure").

On the other hand, to keep your logic "pure", you'll need to make quite significant effort, and to be extremely vigilant when it comes to platform-specific dependencies (see also relevant discussion in Chapter V). This is especially true for C++.

Note that for some out-of-FSM pieces of code, you MAY want to use platform-specific stuff as an optimization. Usually, it works as follows (*yes, I know it is really old news for all the seasoned C++ cross-platform developers, but believe me or not, there are lots of C++ developers out there who don't know it, especially hardcore zealots of Windows-specific development)*:

"Note that for some out-of-FSM pieces of

- You develop a perfectly cross-platform version, which uses only cross-platform APIs. It doesn't really matter

whether cross-platform API is a part of official C++ standard, more important question is whether it is really implemented across the board. In practice, there are several big sets of APIs which we can safely consider cross-platform:

code, you MAY want to use platform-specific stuff as an optimization.

- C++11 standard (C++14 is still only partially supported across the board), including std:: library
- Most of C Standard Library (see discussion on it's limitations in Chapter [[TODO]])
- boost:: library
- Berkeley sockets (while it is not strictly 100% cross-platform, for practical purposes it is very close)
- Note that POSIX standard stuff (the one which is not a part of C library) is generally NOT cross-platform. Notable example: fork() which is missing under Windows
    - Moreover, some Windows functions which look like their POSIX counterparts and have the same signatures, exhibit different behavior. One notable example includes Microsoft _exec*() family of functions, which has very different semantics from POSIX exec*() .

- You launch it, and it works for a while
- Then, you realize that performance of your cross-platform code can be improved for one specific platform. Just as one example – your cross-platform version implemented inter-thread queues-with-select() (see Chapter V for the rationale behind these queues, which are waiting either for somebody pushing something into the queue, or for data arriving to one of the sockets) via sockets+anonymous-pipe, and you realized that under Windows WaitForMultipleObjects()-based version will work faster.
    - Ok, you're rewriting relevant piece of code (keeping all the external interfaces of this piece intact), and placing it under an ugly (but still working perfectly fine) #ifdef MY_DEFINE_WINDOWS_ONLY (and relevant portion of the cross-platform code under #ifndef MY_DEFINE_WINDOWS_ONLY). Bingo! You have your Windows-specific version running under Windows, and your cross-platform version running everywhere else.

Bottom line: C++ can be made cross-platform. For further details, see Chapter [[TODO]].

Cross-platform Languages

…the purpose of Newspeak was not only to provide a medium of expression for the world-view and mental habits proper to the devotees of IngSoc, but to make all other modes of thought impossible.

Another way to achieve cross-platform code is to use one of the cross-platform languages, such as Java, Python, C#, or Erlang.

From cross-platform point of view, these languages have one significant advantage over cross-platform C++: *most* of their APIs are already cross-platform, so they don't provide you *that much* opportunities to deviate into platform-specific stuff. While going platform-specific is still possible (via JNI/Python ctypes/PInvoke or unmanaged code/...), it is usually more difficult with cross-platform languages.

This "going platform-specific being more difficult" is actually the main advantage of cross-platform languages when going cross-platform

In other words, the problem with C/C++ is that they're providing you more freedom with going platform-specific (and yes, having more freedom is not always a good thing). The way cross-platform languages are doing it, can be seen as an (almost) enforcement of a self-imposed rule that "everything should be cross-platform".

Now let's consider these languages against our "baseline" cross-platform C++.

Pros (compared to C++)

"Almost all cross-platform programming languages I know are garbage-collected

- Almost all cross-platform programming languages I know[1] are garbage-collected.
  - It means less time spent on memory management during development, which in turn means faster time-to-market. On the other hand, I will argue that for an FSM model (especially in gaming context, where memory allocations are often discouraged as too expensive), memory management is rudimentary either way, so the difference will be negligible (that is, provided that you have at least one seasoned C++ developer who knows how the things should be done at lower levels, and provided that you are using std::unique_ptr<>).
  - It means no pointers, and no bugs related to misuse of pointers (and, Ritchie save us, pointer arithmetic). Note that once again, we're in the realm of having too much freedom causing trouble (and once again, it is only a question of self-discipline to avoid using them, as references do just fine 90% of the time, and reference-like use of pointers will fill the rest).
- As noted above, keeping your code cross-platform requires much less efforts

in Java/Python/… than in C++.

- Learning curve. C++ learning curve is steep. It is not too bad if you're staying within limits of the FSM, but reading a book on C++ can easily be overwhelming (especially books which start with discussing interesting-but-not-really-important-and-rarely-used-things such as "how to overload operators" and multiple inheritance).
- Good C++ developers are few and far between, not to mention they're very expensive. For most of the languages above (except for Erlang) finding a good developer is usually significantly easier.

Cons (compared to C++)

When speaking about deficiencies of the cross-platform programming languages, several things come to mind (note that while the list of cons is longer than that of pros, it doesn't mean that cross-platform languages are inherently worse; it is just that these cons are not as well-known as cons, so I'm spending more time elaborating on them):

"Almost all cross-platform programming languages I know are garbage-collected

- Almost all cross-platform programming languages I know[1] are garbage-collected. This means that they tend to suffer from two problems:
  - the first problem is memory bloat (if you have any doubts that such a problem exists – take a look at Eclipse or at OpenHAB). I tend to attribute this apparent bloat to the following. While garbage-collected languages eliminate so-called "syntactic memory leaks" (pieces of memory which *cannot possibly* be used), they cannot possibly eliminate "semantic memory leaks" (pieces of memory which *can* be used, but won't be used, ever) [NoBugs2012]. And those "semantic memory leaks" for garbage-collected languages tend to be worse than for manually memory managed languages such as C++, because of "we don't need to care about memory leaks" mentality, and because garbage collectors are obligated to stay on the absolutely safest side, keeping in memory everything that has a slightest chance to be used (i.e. everything *reachable*). Of course, memory bloat for garbage-collecting languages can be managed (there is nothing difficult in explicitly assigning null to a reference); however, whether after doing it they will still provide that much speedup in development time over C++ – is not obvious to me.
    - On the other hand, it should be noted that for FSM-based development (which usually implies states of rather limited size), the problem of "semantic memory leaks" is usually not too bad (based on the same reasoning why manual

Semantic Garbage

Semantic garbage cannot

memory management is usually not that much of a problem for FSM-based development), and fixing them isn't *too* difficult.

> **be automatically collected in general, and thus cause memory leaks even in garbage-collected languages.**
>
> — *Wikipedia* —

- The second problem is garbage collector's infamous "stop the world" (mis)feature. In short – to perform garbage collection, most of GCs out there need to "stop the world" (i.e. to stop *all the threads(!)* within the same VM) for some time. For most of the applications, it is not a problem (as delays even of a hundred milliseconds are so short that your application won't really notice them). However, if we're speaking about a fast-paced game such as an MMOFPS, these delays are known to cause lots of trouble. Even worse, when you run into such things, it is usually too late to rewrite your whole code, which leads to really ugly workarounds such as "let's not run garbage collector at all for a while" (then, if your game event, such as match, is long enough, you can easily eat all the server RAM and even more). While it doesn't mean that GC languages cannot possibly work with MMOFPS, I'd suggest to be very cautious in this regard, and to research how big "stop-the-world" pauses are for the GC used by your target VM (also note that it is about VM, and not about language, so, say, the same C# may exhibit very different behaviour under CLR and Mono).
    - As a mitigating measure, it is possible to reduce the time of "stopping the world" effect (at the cost of some performance loss); see, for example, "Concurrent Mark-and-Sweep" and "G1" garbage collectors for JVM, and <gcConcurrent>/SustainedLowLatency for CLR; they run a large portion of GC processing without "stopping the world" (so only a small part of GC loop needs to be run in the "stop the world" mode). From what I know, these GCs (at the cost of minor overall performance penalty) bring pauses down to single-ms range even for large heaps, which makes it "good enough" even for MMOFPS; as usual, YMMV, batteries not included. For Mono, there is a supposedly similar GC flag *concurrent-sweep*, though I have no information how small the "stop-the-world" pauses are when Mono GC runs with this flag.
    - As another mitigation technique (which, at least in theory, may also work as a compliment to concurrent collectors), it is possible to reduce "stop the world" time by splitting your system into separate VMs (such as JVM or CLR VM[2]) and each VM will run a separate GC. This tends to help because the smaller your "world" is, the less time garbage collector will need to run, so the less time "stop the world" will take. The technique actually flies extremely good with FSMs (as FSMs, at least our FSMs and Erlang/Akka Actors, are share-nothing, they can be easily put into separate VMs). In the extreme case, you may even end up with running one VM for each of "game world"

FSMs. There is a price of it, however, and it is related to the overhead brought by each of VMs; where the optimum for your game (balancing overhead vs latencies) – you'll need to find out yourself.

- The third GC-related problem is related to asynchronous I/O (in our context – socket I/O). Intensive server-side asynchronous I/O tends to cause problems with GC at least under CLR, as to pass the buffer to an asynchronous Win32 API, it needs to be "pinned" (i.e. cannot be relocated, what reflects pretty badly on CLR's copying GC), and having too many pinned buffers may cause CLR's GC to stall, up to the point of being deadlocked. While there is a workaround for it, via SocketAsyncEventArgs (or you can always go into an unmanaged mode, accessing Win32 APIs directly and losing being cross-platform pretty much as we've discussed it for C++ in [[TODO!]] section above), this is a complication one needs to be aware about in a highly-loaded network-oriented environments. Also I have no idea whether the workaround would work as intended under Mono.

JIT

Just-In-Time (JIT) compilation, also known as dynamic translation, is compilation done during execution of a program – at run time – rather than prior to execution

— Wikipedia —

- Unless your target platform has a JIT compiler for bytecode of your language, you're most likely looking at 10x+ performance penalty
  - Fortunately, all the languages mentioned above do have JIT, with only one unfortunate exception (leaving discussion about Lua/LuaJIT aside until "Scripting Languages" section). Erlang, while working on BEAMJIT, still seems to have it only as a proof-of-concept 🙁 .[3]
- Even when compared with JIT-enabled cross-platform language, C++ performance can be made at least *somewhat* better 99% of the time. On the other hand, 95% of the time you won't bother with such optimizations. Possible exceptions include heavy AI and/or heavy physics simulations (especially if they go well with SSE).

SSE

Streaming SIMD Extensions (SSE) is an SIMD (Single Instruction Multiple Data) instruction set extension to the x86 architecture

— Wikipedia —

---

[1] Rust being the only exception

[2] I know that Microsoft prefers to call it "Execution Engine", but it still looks like a VM, swims like a VM, and even quacks like a VM

[3] As for Python, while CPython as such doesn't have JIT, other Python implementations, such as native PyPy and JVM-based Jython, do have JITs.

Personal Preferences and FSMs

Out of the aforementioned cross-platform programming languages, I am especially fond of Erlang's actors (and it also reportedly has a good record for development of large-scale distributed systems, though an overhead due to apparent lack of JIT is significant). Java and Python are not bad either (within their own applicability limits). I have never been a big fan of C#, in particular because it traditionally has the blurriest line between cross-platform APIs and platform-specific stuff (which is not really surprising as such policy makes perfect business sense for Microsoft), but if you're planning your servers as Windows-only – it will certainly do, and if you're going to go Linux – Mono MIGHT work for your too (though in the latter case it is not that obvious).

On the other hand, I need to note that with some self-discipline, FSMs described in Chapter V (and which are strictly equivalent to Erlang's actors/processes), can be easily implemented in *any* of these languages (and in C++ too).

Scripting Languages

We went through C++ and cross-platform languages, but we're not done yet.

"On the server side (unlike client-side) protection from bot writers is not an issue (as server-side code is never exposed to players)

As it was mentioned in Chapter V, for game development, there is a common practice to use scripting languages for game logic. People writing in scripting languages include, but are not limited to, are game designers. Moreover, on the server side (unlike client-side) protection from bot writers is not an issue (as server-side code is never exposed to players), so it means that scripting languages become more feasible for the server-side. Therefore, it seems to make perfect sense to allow using some kind of scripting language on the server too.

Two most common scripting languages, used in games, are Lua and JavaScript. I won't go into comparison of these two languages, but will just note that both will do their job when it comes to game scripting. Just one thing worth mentioning in this regard is that there is an internal conflict within Lua (between main Lua team and Mike Pall/LuaJIT, who actively dislikes changes in Lua 5.3, so LuaJIT doesn't seem likely to support Lua 5.3+, ever(!)); this kind of internal conflicts can be really devastating for the language in the medium- and long-run, which makes it an argument against Lua ☹ .

The most common concern about allowing scripting on the server side is related to performance. However, with LuaJIT (with limitations mentioned above, and I don't really like them) and V8 JavaScript (which also has it's own JIT), this is much less of a concern than for non-JIT-ted script engines.

On Languages as Such

"From the 50'000-feet point of view, 90% of the differences between modern mainstream programming languages (as they're normally used - or better to say, SHOULD be used - at application-level) are minor or superficial.

I know that I will be hit hard (once again) for not going into a lengthy discussion about pros and cons of different programming languages (those which I mentioned above and those which I failed to mention). However, my strong position is that from the 50'000-feet point of view, 90% of the differences between modern mainstream programming languages (as they're normally used – or better to say, SHOULD be used – at application-level) are minor or superficial.[4] This is also confirmed by Line-to-Line conversion exercise discussed in "Line-to-Line conversions: '1.5 code bases'" section below.

Another observation which helps in this regard, is that there is a tendency for modern programming languages to converge as the time goes. For example, C++11 code is much closer to Python code than C++03, and Java 5+ (with generics) is much closer to C++ than Java 4- (the one without generics). Programming languages borrow certain constructs and practices (usually best ones, but it is not guaranteed) from each other, bringing them closer as the time goes.

Still, there are two things which tend to be quite different between the programming languages. The first and more obvious one, is, of course, the difference between manual and automated memory management. Still, with more-or-less modern C++ (with widespread use of containers and std::unique_ptr<>), the difference is not that drastic.

The second thing which MIGHT be quite different between the languages, is related to support for lambdas (which, as we've discussed in "Take 3. Lambda Continuations to the Rescue! Callback Pyramid" and "Take 4. Futures" subsections above, is important). For example, lambdas in Python [StackOverflow.PythonLambdaLoop] and C# [StackOverflow.C#LambdaLoop] have rather strange peculiarities with regards to lambdas within loop (or maybe it's C++ peculiarity that it behaves as intuitively expected?). However, in most cases, some strict equivalent between the languages still exists.

One further word of caution is related to co-routines. While co-routines/fibers do simplify development (this stands to some extent even when we compare them to futures), they have significant practical drawbacks related to lack of "stack snapshot" (which is necessary to implement quite a few FSM goodies, including realistic production post-mortem, see Chapter V for details [[TODO! add section on coroutines and "stack snapshot" to Chapter V]]). Also, their support is still much less universal than that of lambdas.[5]

[4] it doesn't really stand for Erlang, and I am not 100% sure whether it stands for Lua, as I don't have practical experience with it, but C++/C#/Java/Python/Javascript as-you-use-them-for-application-level-programming are all pretty much the same, saving for relatively limited amount of oddities and peculiarities

[5] For C++, you can use fibers, but IIRC Java as such doesn't support them, and Python 2 which is still used quite a lot, doesn't have coroutines

Which Language is the Best? Or On Horses for Courses

horses for courses

An allusion to the fact that a racehorse performs best on a racecourse to which it is specifically suited.

— Wiktionary —

Right above, we've described quite a few options for server-side programming languages. The Big Question is, as usual, the following: which one to choose?

My two ~~cents~~ points in this regard are the following. First, there is no such thing as "the best language for everything". What we need is a language-best-for-some-specific-task. And here there are quite a few different scenarios, from "just a scripting language for game designers to work with" (where C++ and even Java are pretty much out of question), to "time-critical simulation code", with "something for integration with enterprise web apps" in between. As a very wild guess, you might want to use Lua or JavaScript for the first one, C++ for the second one, and Java/C# for the third one (been there, seen that). Doing everything in one single language, while possible, in many cases will be suboptimal.

My second point in this regard is that with FSMs, it is easy to combine FSMs written in different languages, in any way you want. Personally, I've made such things myself for three languages: C++, Java, and JavaScript. It went along the following lines:

- Originally, the whole thing (both outside-FSMs infrastructure code and intra-FSM code) was written in C++. Great performance, full control, no problems with GC, everybody was really happy, etc. etc. But finding good C++ developers isn't easy 🙁 .
- As a result, at some point, it was decided to make an analytics portal and to develop it in Java.
- As pure DB access wasn't sufficient (as they needed real-time updates, and DB triggers didn't look optimal at all) Java guys asked for a way to get the data from C++ system.

- Ok, here went a line-to-line translation project of outside-FSM infrastructure code into Java (to facilitate writing FSMs in Java), see "Line-to-Line Translations: "1.5 code bases"" section below for further details[6]
- This outside-FSM infrastructure Java code was compatible at message format level with C++ code, which means that from C++ FSM standpoint, Java-based FSM was indistinguishable from a C++-based one, and vice versa.
- So, C++ and Java FSMs could interact easily (after agreeing on interfaces, for more details see Chapter [[TODO]]), without no problems whatsoever. In particular, Java FSMs were able to "subscribe" to the data "published" by C++ FSMs, and get all the updates in real-time (most of the data necessary was already published by C++ FSMs, so Java FSM subscribing to the data they needed, was mostly possible without changing C++ code).

"Here went a line-to-line translation project of outside-FSM infrastructure code into Java (to facilitate writing FSMs in Java)

In a different project (and similar situation), a JavaScript FSM was produced to allow server-side scripting (in addition to existing C++ FSMs). In this case, C++ outside-of-FSM code was re-used, which called process_event() (written in JavaScript) from within. The same approach can be (more or less easily) extended to all the other programming languages of interest, see "Supporting Different Environments" section below for further discussion.

In any case, all the paradigms of our FSMs were transparently maintained for all the FSMs across all the supported languages. This included more or less the following things:

- process_event() as a single access point to our FSM, see Chapter V
  - current_time was passed to process_event() either as an explicit parameter, or via TLS and current_time() call (see Chapter V for details)
- timer actions (in those projects, it was timer messages, but now I suggest same-thread futures instead, see "Take 4. Futures" subsection above)
- communication interfaces (see Chapter [[TODO]]), including:
  - support for non-blocking RPCs (it was OO-style same-thread callbacks, but now I suggest same-thread futures instead, see "Take 4. Futures" subsection above)
  - support for state synchronization interfaces (see Chapter [[TODO]])) with same-thread callbacks
- all the recording/replay goodies described in Chapter V

---

[6] we could try to go JNI route instead, but we preferred pure Java and didn't regret

this decision

Supporting ANY language/compiler/JIT: Is It Worth the Trouble?

The next obvious question on this way is the following:

Are such cross-language things worth the trouble of implementing them?

Well, of course, YMMV, but from my experience the answer is

Absolutely!

In such an FSM-based multi-language development paradigm you're no longer tied to one programming language. You may say "hey, this is what CLI/Mono (as well as non-Java compilers into JVM bytecode) are about!" Right, but with CLI/CLR you're still tied to one type of VM (ok, two if we're Windows-only).

And with FSM-based cross-language approach, we're not restricted to one single VM, or to the availability of specific compilers which compile into that single VM. With cross-language FSMs we can use the very best language/compiler pair for each specific job – whether it is Lua/LuaJIT, or JavaScript/V8, or Python/PyPy, or Java/HotSpotVM, or C++/LLVM (note that none of these popular and very-well performing combinations is possible under CLI/CLR).

"With cross-language FSMs we can use the very best language/compiler pair for each specific job – whether it is Lua/LuaJIT, or JavaScript/V8, or Python/PyPy, or Java/HotSpotVM, or C++/LLVM

I rest my case.

Supporting Different Environments

The next question (assuming that I've managed to sell you the idea of using cross-language FSMs) is "how to implement them?"

From my experience, there are two possible approaches. The first one is to have a C++ outside-of-FSM code, and then to integrate C++ into each of the engines you need. Usually, it is not that difficult. For example, you can have C++ threaded communication code running under JNI and calling your Java MyFSM.process_event() fram there. Or under CLI, it is possible to have unmanaged code doing pretty much the same thing. Or with LuaJIT/V8, it is easy to have C++ app calling appropriate script engine.

The second approach is related to line-to-line translations.

Line-to-Line Translations: "1.5 code bases"

Originally, I've written a nice 1500-word piece about line-to-line translations, but then realized that it doesn't really warrant that many words in the context of this book. Still, as it was promised in Chapter V, here comes a brief overview of this not-that-well-known technique.

Let's assume that you already have a working piece of code in C++, and want to port in into Java (it will work pretty much the same for other languages too, like C++-to-ActionScript for the client side, but let's use C++-to-Java for the purposes of our example).

The aim of line-to-line conversion is not only to port the code, but also to keep roughly 1-to-1 correspondence between the original code and the translated code; as we will discuss below, this correspondence is very important for further maintenance of the port (and code maintenance is the thing which haunts all the multiple code bases in the real world).

The idea behind is the following. When you have sufficiently straightforward and platform-independent code in any modern OO programming language, the essence of the code can be translated to a different OO programming language in a very straightforward manner. For example, if your C++ code is implementing Dijkstra's pathfinding algorithm as follows, [7]:

```
pair<map<const Vertex*,int>, map<const Vertex*,const Vertex*>>
   dijkstra(const Graph& g, const Vertex* source) {
  set<const Vertex*> Q;
  map<const Vertex*,int> dist;
  map<const Vertex*,const Vertex*> prev;

  for(const Vertex* v: g.vertexes()) {
    dist[v] = INT_MAX;
    prev[v] = NULL;
    Q.insert(v);
    }

  dist[ source ] = 0;

  while(Q.size()>0) {
    //find u from Q with minimum dist[u]
    const Vertex* u = NULL;
    int distU = INT_MAX;
    for(const Vertex* it : Q) {
      int distIt = dist[it];
        if(distIt < distU) {
        u = it;
        distU = distIt;
      }
    }
    //u found
    assert(u!=NULL);

    Q.erase(u);

    for(const Vertex* v : g.neighborsOf(u)) {
      int alt = dist[u] + g.length(u,v);
      if(alt < dist[v]) {
        dist[v] = alt;
        prev[v] = u;
      }
    }
  }
  return pair<map<const Vertex*,int>,map<const Vertex*,const Vertex*>>(dist,prev);
}
```

...then, when you need to rewrite this code into, for example, Java, you can simply take your (supposedly working) C++ code, and to write its Java equivalent along the following lines:

```
1  public class Dijkstra {
2    public static Pair<TreeMap<Vertex,Integer>,TreeMap<Vertex,Vertex>>
3       dijkstra(Graph g, Vertex source) {
4     TreeSet<Vertex> Q = new TreeSet<Vertex>();
5     TreeMap<Vertex,Integer> dist = new TreeMap<Vertex,Integer>();
6     TreeMap<Vertex,Vertex> prev = new TreeMap<Vertex,Vertex>();
7
8     for(Vertex v: g.vertexes())  {
9        dist.put(v, new Integer(Integer.MAX_VALUE));
10       prev.put(v, null);
11       Q.add(v);
12    }
13
14    dist.put(source, new Integer(0));
15
16    while(Q.size()>0) {
17      //find u from Q with minimum dist[i]
18      Vertex u = null;
19      int distU = Integer.MAX_VALUE;
20      for(Vertex it: Q) {
21        int distIt = dist.get(it).intValue();
22        if(distIt<distU) {
23           u = it;
24           distU = distIt;
25        }
26      }
27      //u found
28      assert u!=null;//be careful to keep your Java asserts
29                  //  consistent with your C++ asserts;
30                  //see Chapter [[TODO]] for further discussion
31                  //  on asserts in C++
32
33      Q.remove(u);
34
35      for(Vertex v:g.neighborsOf(u)) {
36        int alt = dist.get(u).intValue() + g.length(u,v);
37        if(alt < dist.get(v).intValue()) {
38          dist.put(v,new Integer(alt));
39          prev.put(v,u);
40        }
41      }
42    }
43    return new Pair<TreeMap<Vertex,Integer>,TreeMap<Vertex,Vertex>>(dist,prev);
44  }
45 }
```

[[TODO: place C++ and Java side by side in a book]]

As you can see, ported Java code visually looks very similar to C++ original; moreover, we can easily see the one-to-one correspondence between the lines of C++ code and Java code. When we have such C++ and Java code, we don't really have two separate code bases, as they're too closely related to name them separate. I

prefer to name such "C++ and some-other-language" pairs as "1.5 code bases" (at least it is clearly more than 1 code base, and certainly less than 2).

In practice, it means that for really platform-independent C++ code, in most cases we can produce an equivalent code in a different programming language, but with the same semantics and (hopefully ;-)) producing exactly the same result. Moreover,

it is usually possible to produce an equivalent code which has one-to-one line-to-line correspondence with the original.

This last observation is extremely important in practice, for the purposes of code maintenance. As it is pretty well-known in the industry,[8] having two code bases very frequently leads to major problems because of code maintenance issues. In other words, after having changed one code base, it is often a problem to make a strictly equivalent change in another code base. However, with line-to-line conversion and "1.5 code bases", this maintenance process (while still not being a picnic!) becomes significantly simplified: after we've had our code bases equivalent, and we've made a single change in our first code base, then making an equivalent change becomes a breeze: just look at source control differences for the first code base, and apply an equivalent thing to the second code base. It is important to note that

in this apply-changes-to-second-code-base process, there is usually no need to understand the essence of the change which was made; most of the time, the change can be applied based only on general understanding of inter-language equivalence rules[9]

Here in original 1500-word piece there were examples of modifying the C++ code – getting differences from the source control – applying those differences to Java, but within the scope of this book, it probably would be an overkill, so I've eventually decided to skip it. [[TODO!: write a separate article about it with more examples]]

---

[7] based on [Wiki.Dijkstra]

[8] and I've seen several times myself as competitors successfully started second client with a separate code base, and then second client started to lag behind in development, up to the point of being unplayable, with subsequent abandoning of the second client

[9] in fact, I'm pretty sure that, given restricted dialect of C++, which I am normally using for platform-independent code, it is perfectly possible to build a C++-to-Java (or C++-to-ActionScript) compiler *at source level*; however, parsing C++ (which is not a LALR(1) grammar) is difficult, so I've never had enough time to undertake such an

endeavor

Line-to-Line Translations: Are They Practical?

The code above is an interesting exercise, but of course, it is merely an example, so you may still have questions whether this exercise scales well to larger-scale pieces of code.

As noted above, I was personally involved in an exercise of porting C++ into Java in a Line-to-Line manner. Porting 20'000 lines of code took 2 weeks for the first 80%, and two months for the remaining 20%, and worked happily ever after (the code was changed since the port, but quite rarely). I don't know how it would scale to a million of lines of code, or to a code which is changed twice a day, or to a code which is not as straightforward. Still, if you're out of other options, line-to-line source translations may happen to work for you.

> **"Porting 20'000 lines of code took 2 weeks for the first 80%, and two months for the remaining 20%, and worked happily ever after**

Also note that performance-wise the converted code might be not top-notch one (while concepts and ideas are generally very similar between the languages, subtle performance-related details don't). In other words, with good conversion if your algorithms were O(N) they generally should stay O(N), but you may easily face 20% performance hit (potentially more in extreme and fringe cases) compared to the best possible code in target language.

One further thing to keep in mind in this regard, is that porting from C++ to Java (C#/...) is generally simpler than the other way around. In particular, this is because while removing manual memory management is trivial, adding it can be quite difficult and MAY require intimate knowledge of the program internals (which goes against the idea of purely mechanistic conversion).

Inter-Language Equivalence Testing: FSM Replay Benefits

In quite a few cases, you may need to port a part of your code from one language to another one. It may happen, for example, to optimize the time-critical FSM, or to have "1.5 code bases" in a line-to-line conversion manner as described right above. And with all such conversions, one of the biggest problems is the question "how we can be sure that the code-in-new-language and the code-in-old-language are strictly equivalent?"

Fortunately, for FSMs there is an easy way to test the code equivalence. The procedure goes as follows:

- "record" a big chunk of inputs and outputs for FSM-being-ported (and running old code); "recording" can be done along the lines described in

Chapter V, and may be done even in production.

- "replay" it in lab on the new code. (as described in Chapter V)

- if the results are exactly the same for old code and new code, on a sufficiently large chunk of real-world data, it means very good chances that the code is indeed equivalent (at least within the bounds which are of practical interest). In practice, it has been noticed that for quite a big site, if there is no bug after the first four hours after new code deployment, there won't be any bug in "core logic" at all. Pretty much the same applies to record/replay testing.

    - if there is a non-equivalence, it can be found very quickly by simply running the same "replay" over both languages in debugger line by line, and comparing corresponding variables.

On Code Generators and YACC/Lex (or Bison/Flex)

"As soon as you've got lots of boilerplate code - you MIGHT be able to generate it with your own code generator

One thing which doesn't strictly belong to server-side, but which I need to mention somewhere, is YACC/Lex. As we'll see later, there are quite a few cases where having your own source-code-generator is beneficial. Two most obvious examples include Interface Definition Language (a.k.a. IDL, discussed in Chapter [[TODO]]), and prepared-statements-code-generator (discussed in Chapter [[TODO]]). Other game-specific things (usually not really comping code, but dealing with some declarative statements and converting them to code) might also be helpful (in general, as soon as you've got lots of boilerplate code – you MIGHT be able to generate it with your own code generator). In the (rather extreme) case you may even be able to write your own code generator to support co-routines-with-stack-snaphot.

While these compilers not strictly required (and you're able to write all the code you need by hand), they will speed up your development a lot. Just as one example: if you need a thousand of those prepared statements, writing them by hand in C++ (or Java/pick-your-poison), while possible, is very tedious and error-prone. The same goes for any kind of marshalling/IDL.

Whenever you have your code generator, it always works as follows:

- You have a file in your own language (usually more of declarative nature than of imperative nature).

- Then, you run your code generator over it, obtaining kinda-source code in your programming language.
  - **This generated kinda-source code MUST NEVER EVER be modified manually.** Instead, either source-code-in-your-own-language needs to be modified, or your code generator.
- Then, you compile kinda-source with your usual language compiler (or interpret it, whatever)

In most cases, the best way to implement such compilers is via YACC/Lex (or Bison/Flex, which is pretty much the same thing). As for these generators performance is not important, alternatively to classical YACC/Lex/Bison/Flex you may want to look at [PLY] (Python Yacc/Lex). The idea of all of them is pretty much the same:

"In most cases, the best way to implement such compilers is via YACC/Lex (or Bison/Flex, which is pretty much the same thing).

- you're writing "language grammar" (in .y file, which is more or less reminiscent of BNF forms)
- you're specifying what exactly you want to do with it as you're parsing (within the same .y file)
- you're compiling this .y file and obtaining your C (or Python, in case of PLY) code
- you're running this compiled code over your source file, and (if you've done everything right) are obtaining an abstract syntax tree (AST)
  - as you've got your AST, you can generate any code you need, out of it

For further information on C-language YACC, please refer to the classical tutorial [Niemann]. One further trick I'm using a lot for such code generators, is the following:

- define YYSTYPE as a C++ class (which will be essentially your "AST node"); usually YYSTYPE is int, but nothing prevents you from re-defining it
- define member functions for your YYSTYPE so that you add other AST nodes into current one. Don't be afraid to make deep copies here – you won't notice performance differences anyway.
- use code such as { $$.add($1,$2); } within your .y file (see tutorial mentioned about on the meaning of these magical $$ and $1/$2).
- when the whole hierarchy is processed, you'll get your whole AST at the (logically) topmost rule of your .y file.

Due to lots of copies, this approach is damn inefficient compared to traditional compilers,[10] but for vast majority of our gaming purposes parser performance

won't matter, saving you quite a bit of development time (and as it is run only on your development/build machines, it won't affect performance of your runtime code at all).

---

[10] and was also reported to fail under some YACC implementations when trying to compile hundreds of thousand of lines, but this problem is solvable

[[To Be Continued…

This concludes beta Chapter VI(e) 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 VII, "Modular Architecture: Protocols."]]

[–] References

[NoBugs2012] 'No Bugs' Hare, "Memory Leaks and Memory Leaks"
[StackOverflow.C#LambdaLoop] "Captured variable in a loop in C#", StackOverflow
[StackOverflow.PythonLambdaLoop] "What do (lambda) function closures capture in Python?", StackOverflow
[Wiki.Dijkstra] "Dijkstra's algorithm", Wikipedia
[PLY] David Beazley, http://www.dabeaz.com/ply/
[Niemann] Tom Niemann, "Lex & Yacc Tutorial"

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« *Asynchronous Processing for Finite State Machines/Actors:…*

MMOG. RTT, Input Lag, and How to Mitigate Them »

Filed Under: Distributed Systems, Network Programming, Programming, System Architecture
Tagged With: game, multi-player, programming language, server