XuFly Design: Choosing a Language for a Flynetting Swarm

Arpit Jasapara, XXXXXXXXX

Abstract

XuFly is a company that wants to create a swarm of tiny robots to control the insect population. It needs to know which language would be most suitable for its needs. It needs a language that supports multithreading and parallelism, while allowing for easily switching the hardware and platform. The language needs to also be performant given in mind the storage, memory, and other hardware limitations, but also reliable and relatively error-free. The language should also be easy to use. We compare C++, Java, Python, and Go in all of these categories, and discover that Go and Python would be best suited for XuFly, with slight preference towards Go.

1 Introduction

XuFly plans on creating tiny robotic insects in order to provide flynetting and eventually flyswatting services where traditional forms of insect control such as pesticides cannot be used. In order to do so, they need to have a very small hardware component running machine-learning and cooperation algorithms, while the hardware itself could be switched out easily. Thus, it becomes very important to select a language that will not become a bottleneck in this case and would maximize the performance of these insects on such limited hardware. The languages that we will be looking at today are C++, Go, Java, and Python.

2 Multithreading/Parallelism

An important factor to consider between all the languages is its ability to support multithreading and parallelism. While all languages support parallelism in one way or another, there are obviously better and lighter weight choices. C++ uses pthread library while Java uses the in-built Thread classes to allow for multithreading. Similarly, C++ uses the fork() function to spawn off child processes while Java uses the exec() keyword. Both languages are almost identical in their usage of multithreading and parallelism, but research has shown that C++ has a better overall speedup while Java performs much better with higher thread granularity. Java does have an edge due to built-in support for threads though. However, both of these languages aren't even close to Go and Python's support for parallelism.

Python uses a library called asyncio to allow for asynchronous programming while Go uses goroutines which are similar to child processes for specific functions. C++ and Java's threads and child processes are considerably heavy on memory and considering the fact that most of the hardware has very limited memory, these languages have a serious bottleneck. For example, the Coral SoM has only 1GB of LPDDR4 RAM. On the other hand, Python's asyncio and Go's goroutines take up considerably less memory and are much more lightweight.

Between asyncio and goroutines, the distinction is very slight, but it still does exist. Asyncio is a module for Python, whereas goroutines are part of and in fact core to the Go language. Thus, goroutines have an advantage due to its native nature and the fact that the language was built around its use. Furthermore, goroutines block much less often than Python's asyncio due to its memory sharing being a lot more lax. This may cause issues where tasks need to be more serializable, but in the case of the flynetting swarm, since each individual task seems relatively simple, it is more important to make sure that certain robots are not simply blocked. Lastly, due to native support for goroutines, they execute threads in parallel and automatically handle blocking syscalls in a separate thread, whereas asyncio uses a thread-based event pool and an executor to call for a separate thread.

Clearly, in terms of multithreading and parallelism, Go's goroutines win out with Python's asyncio coming in a close second. Another advantage for Go is that as of Go 1.5, there is multiprocessor support for the language whereas asyncio does not support it. This is a significant difference and again comes from the fact that goroutines are natively handled in Go so multiprocessor support was a natural addition to later versions of the language.

3 Platform Interchangeability/Storage

Since XuFly wants to be able to switch suppliers easily, it is very important to choose a language that takes this idea into consideration. Technically, with any of the 4 languages being considered here, XuFly can switch suppliers. However, some provide more challenges and bottlenecks than others. In order to execute C++ code, a binary that is OS and platform specific must be generated. This means that every time the company switches platforms, the entire code would need to be recompiled and ported over to the robot. This can be very tedious and time-consuming. Additionally, each time a change is made in the code, the entire code would need to be recompiled and distributed across the robots. Since the robots are probably operating on different hardware, this would mean that several different binaries would need to be created for each code change, which becomes very unmanageable.

Java is much better than C++ in this regards since the bytecode generated by its compiler can be run on any OS and platform without specific changes necessary. Thus, only one JVM bytecode would need to be distributed across all the robots regardless of platform. However, this comes with the same problem that C++ has where each code change would need to be recompiled into bytecode and redistributed. On the other hand, if the robots actually receive the code updates and recompile the code themselves, then the matter at hand becomes slightly different. In this case, both C++ and Java become identical since the gcc on each machine would create only the specific binary needed for that robot. However, this act of compilation is very resource-heavy and would take up a significant amount of time and resources so the robot would become ineffective during this compilation period. It would also take up significant amounts of storage since the robot would contain both the source code and the binary on its hard drive.

Python, on the other hand, would be very easy for platform interchangeability. Since Python's source code is dynamically interpreted, there is no compilation process involved so code changes would be very easy to distribute. It would not be tied to any platform whatsoever. Additionally, since Python code takes up much less storage on average than the verbose C++ and Java source code, it would take up less hard drive storage, especially when considering that no additional binaries are needed. However, these advantages come at the cost of runtime performance since Python is translating and executing the code at the same time, whereas the machine code for C++ and Java would take up much less CPU cycles to run. Similarly, error and bug handling can become more problematic since the code would be dynamically updated instead of statically in different releases.

Go is similar to C++ and Java in that it is a compiled language compared to Python's interpreted language. However, Go's compiler is much more lightweight than C++ and Java's compiler in that it does not have any exposed intermediary stages. Additionally, Go's compiler supports building for all platforms and is not OS/platform specific like the ones for C++ and Java. While this adds slight bloat to the compiler, most of it is negligible since most of the compiler is OS independent anyway. Plus, the language itself saves on storage when compared to C++ and Java since it's considerably less verbose. The compilers themselves also would be easier to update and the binaries are easier to distribute. Thus, Go is probably the best in this category out of the compiled languages. Compared to Python, the code runs considerably faster with and has the other benefits of a compiled language, but loses out on the massive dynamic interoperability of an interpreted language. Thus, in terms of platform support and storage, Python wins out with Go being a close second.

4 Performance/Reliability

In terms of performance, the languages are fairly similar since the scale of the program is relatively small. However, Java is reputed as one of the more inefficient languages due to its latent garbage collection and other bloat surrounding the language. With machine learning and AI algorithms, this could cause a potential issue in performance speeds. On the reliability side, Java is very reliable. Since it does not use pointers and hides away most of the memory usage, there is little issue with memory access and other such risky actions. However, it still suffers from race conditions in its parallel approach and requires strict locking to prevent this issue from rising. For error handling, Java uses the throws keyword to ensure that all foreseen errors are appropriately handled in some kind of graceful manner. Thus, Java becomes a very reliable albeit not performant language.

C++ on the other hand can be very performant. It has little bloat since there is no garbage collection and memory management is handled by the user.

However, this brings up its own security and reliability issues with illegal memory accesses that can crash the system, invalid addresses, memory leaks, and other errors that must be carefully checked for by the user. Like Java, it suffers from race conditions even more so since its threads are not even native to the language. C++ has the barebones requirement for an object-oriented language, making it very performant, but very prone to errors and low on the reliability scale.

Python, on the other hand, is very reliable. It not only handles garbage collection and memory management, but it is also dynamically typed, so these kinds of errors do not occur as often. However, unlike Java, it does not force all foreseen errors to be handled appropriately so it leaves that open to the user. Python is more performant than Java as a language, but loses out on the fact that it is a dynamic interpreted language, whereas C++ and Java are compiled languages. Thus, it eats up more CPU cycles and loses out on performance in that regard. However, there are more lightweight versions of Python such as PyPy that remove a lot of the performance issues while still keeping the reliability benefits of using Python. Asyncio helps remove many race conditions since it uses a thread based event pool but explicit locking is still needed for shared resources.

Go is the newest language and as such, combines a lot of the benefits from C++, Java, and Python. Since it is a compiled language, it does not suffer from the Python interpreter performance bottleneck. It also has very sophisticated garbage collection, but still uses pointers for all of its objects to make memory management easier but less performance-heavy. It natively supports parallelism through its goroutines and does the best job of preventing race conditions due to its channel approach of parallelism. While Go prevents the random invalid addresses and memory leaks of C++, it still prevents the bloat of Java by keeping pointers. It is also more performant than Python for the most part. Thus, in terms of performance and reliability, Go seems to be the best balance needed in the modern world where reliability is crucial, with Python a close second and Java right behind. C++ with its lack of reliability loses out big on this category.

5 Ease of Use

C++ is notorious as a hard to use language. From static typing to pointers to memory management to manual garbage collection to unfriendly class and object usage, it is very difficult for any programmer who is unfamiliar with the language to get used to. There are a lot of things to keep in mind and even more things that are not often explained to you when you begin programming. It takes time to master this language and as such is the most difficult programming language to use on this last.

Java is better than C++ but is still relatively difficult to get used to. Most of the memory management stuff is abstracted away so the user only needs to focus on static typing and learning the many keywords in the language. They also need to learn about the many libraries in the language and get used to the difficult syntax. While all of these boundaries are present in C++, their presence in Java makes it the second most difficult language to use on this list.

Python is dynamically typed with the most friendly and easy to read syntax. It abstract away all the memory management issues and does not require a class to get started. It’s as simple as typing in the first line in any Python environment. Out of all the languages, it is by far the easiest to use and the most intuitive to learn.

Go is loosely statically typed. It still requires a type for its variables but can figure out what the type is often by itself with no specific user-mandated keywords. It also abstracts away the memory management stuff and the syntax is marginally easier than Java. However, to use the language to its full capabilities, there’s definitely more to learn, but it’s still the easiest to use after Python on this list.

6 Conclusion

Clearly, based on the many common areas of concern for a language, the two best options are Go and Python. Go narrowly wins on more categories, but Python is still a fair alternative, especially smaller variants such as PyPy. C++ has too many issues for an application of this scale, and Java would not be performant enough especially in this modern age of TensorFlow and machine learning algorithms. In the industry, Python and Go are most often used for these types of applications and the reasoning is clearly laid out above.

References

- https://www.geeksforgeeks.org/c-vs-java-vs-python/
- https://codeburst.io/why-we-switched-from-python-to-go-60c8fd2cb9a9
- https://coral.ai/products/som/
- https://www.tensorflow.org/lite
- https://stackoverflow.com/questions/20993139/how-does-goroutines-behave-on-a-multi-core-processor
- https://stackoverflow.com/questions/44272954/goroutines-vs-asyncio-tasks-thread-pool-for-cpu-bound-calls
- http://kth.diva-portal.org/smash/get/diva2:648395/FULLTEXT01.pdf
- http://net-informations.com/python/iq/interpreted.htm
- https://www.pluralsight.com/blog/software-development/the-go-compiler
- https://codeburst.io/why-golang-is-great-for-portable-apps-94cf1236f481