C768 Technical Communication, Task 1

Zachary J. Allen

Western Governors University

C768 Technical Communication

William Dean

May 21, 2022

About Colombia Corporation **3**
A.1 Products and Services 3
A.2 Organization Size and Number of Locations 3
A.3 Colombia Corporation and Existing Industry 4

Research - Rust **5**
B.1 ‘Rust: A Language for the Next 40 Years’, by Carol Nichols 5
B.2 ‘Type-Driven API Design in Rust’, by Will Crichton 5
B.3 ‘Whoops! I Rewrote it in Rust’, by Brian Martin 6

Rust in the Lab **7**
Executive Summary 8
Technical Debt 9
More Careful 9
Transparent Configuration 10
Memory Safety for Free 11
Compose and Modularize 11
Rust Supports our Talent Acquisition 12
Conclusion 13

Explanation of Diction **14**

References **14**

A. About Colombia Corporation

Colombia Corporation is a startup focused on automating the deployment of modular instrument control software, firmware, and analytical tools. I was recently hired to work under the CTO during early development, so I've included their internal values and description here for reference.

A.1 Products and Services

Colombia is in the process of developing Astoria, an ecosystem of LaC (Laboratory as Code) tools and protocols which will radically accelerate and simplify the deployment of new laboratory experiments, instruments, and data analysis tools, all written in C and C++. Astoria allows researchers to draft, simulate, deploy, and analyze entire experimental processes, all in one place. Given an experiment plan, Astoria can create firmware and circuitry for custom sensors and controls; set up networking, server, and database configurations to track the status and consume the results of experiments; and write, execute, and visualize R or Matlab programs to extract the described experimental results.

A.2 Organization Size and Number of Locations

Colombia Corporation started in 2021 and recently completed seed funding. We're a proud team of 4 individuals working remotely in the United States, with no central office space. Our business processes mainly take place in a commercial license of GSuite, where we manage our business documents, scheduling, video calls, and presentations. Our software development team, once they've been hired, will continue development on our prototype of Astoria, and will operate tightly within Amazon Web Services (AWS), which will also be the backbone of Astoria.

A.3 Colombia Corporation and Existing Industry

We aim to forge a clear trail between laboratory scientists and instrumentation engineers. We expect to disrupt the fractured protocols, black boxes, and walled gardens which permeate scientific instrumentation. Astoria enables high-agility research and development labs to out-compete slower, more traditional laboratories to high-value research products, while avoiding the high cost of troublesome instruments and difficult databases. Astoria does this by moving the plane of interaction up to the researcher's level, simplifying and often eliminating in-house software development or LIMS services.

B. Research - Rust

In order to automatically deploy reliable, performant software, I've started researching whether we need to write large sections of Astoria in Rust. Rust is a systems-level language which values memory safety and zero-cost abstractions.

B.1 '*Rust: A Language for the Next 40 Years*', by Carol Nichols

Nichols presents a summary of the role C has played in enterprise software development, how cultures of unsafety get replaced, and what Rust offers as a long-term C replacement. She primarily argues that memory safety issues are the most important concern of systems-level programmers, transitively quoting Matt Miller (2019): since at least 2006, "~70% of the vulnerabilities addressed through a [Microsoft] security update each year continue to be memory safety issues." Nichols introduces Rust as a solution. Rust's memory management is centered around Ownership and Borrowing, where information about where data is stored in memory is always connected to how that information can be stored, modified, or shared. By enforcing a handful of rules around storage, modification, and sharing, Rust is able to refuse code which introduces unsafe memory management. Astoria is expected to correctly and reliably handle complex laboratory configurations, and this presentation suggests Rust can help prevent a majority of serious errors.

B.2 '*Type-Driven API Design in Rust*', by Will Crichton

Crichton delivers a detailed live demonstration of the Rust trait system, its motivation, and how traits prevent incorrect use. Traits are guarantees about what can and can't be done with a structure of data. For example, the Iterator trait guarantees that a structure of data can be consumed piece by piece in an ordered way; whereas numbers like 11 should not have the Iterator trait, because they don't have pieces. Functions are also defined partially in terms of

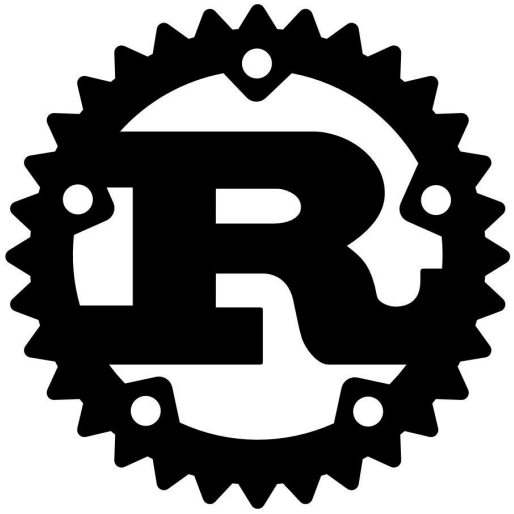
what traits their arguments and results must have, using ‘where clauses’ to indicate that the function should only be permitted where the trait exists. With these simple ideas, Crichton shows that “you can combine these where clauses in really creative ways to help users avoid mistakes[,] to make sure you can only call a method when certain capabilities exist or [...] when your object is in a certain state.” By ensuring that a function is never called on structured data which may have unacceptable (‘out of bounds’) traits, Rust can ensure that a properly made function is hard to misuse. One of the likely challenges when writing Astoria will be defining the protocols, data structures, and expectations of every piece of a laboratory. Rust’s trait system may allow the foundational work of Astoria to establish a set of traits for each role in the lab, so that as the ecosystem grows, new components can be added which interact cleanly with older ones.

B.3 ‘*Whoops! I Rewrote it in Rust*’, by Brian Martin

Martin describes the process he used to rewrite and improve a Twitter cache server in Rust. His initial plan was to replace the server’s networking and event loop with an equally fast Rust implementation, reusing core C components and calling them from Rust. As the project developed and he ran into debugging and tracing tool incompatibility issues, he expanded the scope to a full rewrite in Rust. This gave him the opportunity to implement a better (more options for free) storage memory layout, and the project is nearing feature parity. Currently his implementation has equal or better single core performance than the C implementation. He also discusses the benefits of excellent tooling around Rust, including fuzzers and benchmarking of critical components. Martin’s discussion of the transition costs between C and Rust implementations suggest that writing Astoria in a mixture of Rust and C can be viable, performant, and stable, IF the tooling in use allows that interoperability.

Rust in the Lab

A Language Proposal for Astoria



Zach Allen

May 16, 2022

Colombia Corporation

92343 Fort Clatsop Rd

Astoria, OR 97103

ID 001091806

zacharyja@colombia.com

(509)555-2471

Executive Summary

Writing Astoria in C/C++ is guaranteed to introduce crippling technical debt, which we won't have the resources to pay. This technical debt will take the form of memory and type safety issues across half a dozen platforms and perhaps dozens of interfaces. The cost to purge these issues is essentially infinite; permitting them will make Astoria flakey, inflexible, and obtuse. A strategic compromise requires innovative developers, driving up hiring costs. Rust is the best alternative. Rust guarantees memory and type safety across every platform Astoria visits. Rust is loved by those who use it, and is exciting to those who don't. Colombia Corporation can use Rust as part of our developer culture to attract the talent and inventiveness needed to thrive. Rust is the right choice for Astoria.

[1] *Rust Logo*. Rust Foundation. Accessed via https://www.rust-lang.org/static/images/rust-logo-blk.svg.

Technical Debt

Colombia Corporation has leveraged expectations of Astoria's anticipated MVP, to be written in C and C++. Our investors expect excellent reliability and an easy Laboratory-as-Code (LaC) interface; in return, we have a seed funded budget of $4.4M. However, that budget may also be leveraging technical debt: the growing, long-term penalty for making poor foundational design decisions with high short-term payoff. This technical debt will take the form of memory and type safety issues across half a dozen platforms and perhaps dozens of interfaces. Microsoft found that "~70% of the vulnerabilities addressed through a security update each year continue to be memory safety issues" (Miller & Microsoft, 2019), with similar stories from other software companies. Memory safety issues come from unchecked freedoms in C and C++, languages chosen decades ago for their high short-term payoff in the race to dominate the home PC market. These 70% of vulnerabilities are interest paid on technical debt, borrowed from the foundational languages the industry relies on, diverting value away from shareholders and customers.

More Careful

More vexingly, there seems to be no repayment plan for this technical debt: the Microsoft study found the rate of memory safety vulnerabilities was stable at 70% since at least 2006, suggesting these issues are being added as quickly as they get replaced. When the patch for the 2015 Android Stagefright exploit was first released, replacing the original integer overflow with another one just like it, XDA-Developers.com's Developer Admin Pulser_G2 (2015) wrote that "The best way to prevent these kinds of attacks is either to use a higher level language, which manages memory for you (albeit with less performance), or to be very, very, very, very careful when coding. More careful than the entirety of the Android security team, for sure." The operational, hiring, and agility cost for bringing that much care to Astoria's development would

be crushing, requiring the development and enforcement of exemplary unit tests, protocol fuzzers, fault tolerance systems, interface contracts, standardization and documentation requirements, and methodical code- and change-review systems at every level. Colombia Corporation's developer culture has always expected to maximize best practices; however, the investment required to be more careful that Microsoft's and Android's security teams simply isn't available before Astoria comes to market, and will not be possible after. Therefore, the value gained by making it to market written in C/C++ must be bought with technical debt.

Transparent Configuration

Without this unsurmountable investment, the burden of safety and interoperability falls to our customers, undermining Astoria's central value to researchers: bringing their experiment's infrastructure up to their level[2]. In order to provide a wide, transparent ecosystem of LaC components, Astoria needs to transparently manage the integration of microcontrollers, network protocols, local workstations, databases, cloud services, web-based control panels, reports, dashboards, and analysis pipelines. In conflict with this breadth and depth of component interactions, researchers need to describe their experiments in as minimal and intuitive a fashion as possible, without needing to spend time considering whether Astoria can support their configuration. Market research[3] suggests that designing against integer overflows is not an intuitive task to any of our customers.

---

[2] *A Business Case for Laboratory Infrastructure as Code, rev. 3.6*. Colombia Corporation, Dec. 7 2021. Accessed internally. Fictional Source.

[3] *Astoria Market Research Report: TaPP 2020*. Colombia Corporation, July 3 2020. Accessed Internally. Fictional Source.

Memory Safety for Free

So, how will Colombia Corporation leverage high short-term payoffs, make it to market, and avoid inescapable technical debt? I propose Rust. Rust started development in the 2010's at Mozilla as a C/C++ replacement within Firefox (Rust Foundation, n.d.), reaching 1.0/Stable in 2015. Like C and C++, Rust is primarily a systems programming language, forgoing simplicity and perfect portability to get performance and stability. Rust's design focuses on "build[ing] reliable and efficient software," including proven guarantees that Rust code will not compile with unnoticed memory safety issues. This is achieved through a minimal set of rules about Ownership and Borrowing of data in memory. Instead of the compiler allowing pointers to be freely duplicated, converted to integers, or used after being freed, the Rust compiler tracks the status of what reference to an object owns that object, and what references are borrowing it. This tracked information allows the compiler to reject code that accesses memory with an uncertain or unacceptable owner or borrowing status, without imposing a runtime penalty or garbage collector. This foundational reliability will ensure Astoria's growing codebase will be an asset, not a liability.

Compose and Modularize

Rust also provides foundational support for the expressiveness and intuition of researchers. As an experimental design is created, the underlying structure of how its components integrate can be represented as a programming paradigm, and the enforcer that makes sure they integrate correctly represents a type system. Rust is a multi-paradigm language, including support for object-oriented generics and functional programming. These allow researchers the flexibility to compose and modularize their experimental descriptions with the full support of the language behind them. Rust's type system is robust and brief, avoiding many

of the wrapper/reflection/proxy object types in favor of a system of traits - interfaces which can be declared for objects meeting certain conditions, even across package boundaries (Crichton, 2021). This trait system fully supports generics; however, its inheritance system offers less support than some other languages, like Java. Rust's ecosystem provides support for Rust in the cloud, browser, server, desktop, and microcontroller (Rust Foundation, n.d.), and Rust's traits and paradigms allow for all of these deployments to share a solid, interoperable system of types, protocols, and expectations. This means Astoria's functionality may rest squarely on the paradigms and type system of Rust.

Rust Supports our Talent Acquisition

Beyond the technical benefits of Rust, it can also help ensure the next hiring window captures the talent Colombia Corporation needs. According to the 2021 Stack Overflow Developer Survey (2021), 60% of developers who worked extensively with C dread working with it, with C++ doing a little better at only 51%. In contrast, Rust has won their ranking of "Most Loved Programming Language" since 2016, with 87% of Rust developers saying they love working with it. This high praise has put Rust at the forefront of languages developers want to learn, with 14% reporting they want to learn Rust (Python taking the lead at 19%). Learning Rust is easy, and migrating from C/C++ offers a largely familiar syntax. With extensive community support, a native package manager, and development tools focused on reliability and benchmarking, Colombia Corporation can use Rust as part of our developer culture to attract the talent and inventiveness needed to thrive.

Conclusion

Industry reports indicate that C and C++ introduce serious technical debt in the form of vulnerable and unsafe memory use, and that no scale of investment can eradicate these languages' vulnerabilities. Writing Astoria in C/C++, even with a strategic approach to managing this debt, exceeds the capital available to bring our product to market and remain agile. C and C++ also provide insufficient support for the varied platforms, rich expression, and rigorous validation of experimental infrastructure. In contrast, Rust is a beloved and exciting systems programming language offering guaranteed memory safety, a robust type system, and multiple programming paradigms. Rust eliminates the risk of unnoticed unsafe memory use, deploys to every platform Astoria needs, and has first-class support for the various ways researchers want to express and compose their experimental designs. Rust is an exciting topic among software developers and can act as a draw for talented new hires. Rust is the right choice for Astoria.

Explanation of Diction

Since I'm a new hire at Colombia Corporation, and it's a growing company, it's likely my supervisor is a founder at some level, and will make decisions by collaborating with other founders. Because of this, I mixed my technical jargon based on context and content. I did my best to provide in-text clarifications for every technical term related to memory safety and technical debt, as those make up around 80% of the argument in favor of Rust, so they need to be explained to other founders. In those sections I also relied on an understanding of financial terms and metaphor, to best explain the relationship between language choice and liability, which is something my supervisor has a chance of not being comfortable with, so I gave some basic explanations. This is also the approach I took in the conclusion. For other parts of the document, those which were less central to the other founders, I relied on my supervisor's degree to provide knowledge about basic programming concepts like 'generics' and 'functional programming', in which case I didn't provide in-text clues. Much of the document is in a 'business casual' tone, dictated with precision and medium formality.

References

Crichton, W. (2021, October 1-2). *Type-Driven API Design in Rust*. YouTube. Retrieved May 6, 2022, from https://www.youtube.com/watch?v=bnnacleqg6k

Martin, B. (2021, October 1-2). *Whoops! I Rewrote it in Rust*. YouTube. Retrieved May 6, 2022, from https://www.youtube.com/watch?v=XdMgH3eV6BA

Miller, M., & Microsoft. (2019, February 6-7). *Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape*. YouTube. Retrieved May 10, 2022, from https://www.youtube.com/watch?v=PjbGojjnBZQ

Nichols, C. (2019, April 23-24). *Rust: A Language for the Next 40 Years* [Conference Presentation]. YouTube. Retrieved May 6, 2022, from https://www.youtube.com/watch?v=A3AdN7U24iU

Rust Foundation. (n.d.). *Frequently Asked Questions*. rust-lang.org. Retrieved May 12, 2022, from https://web.archive.org/web/20160609195720/https://www.rust-lang.org/faq.html

Rust Foundation. (n.d.). *Rust Language*. Rust Programming Language. Retrieved May 21, 2022, from http://rust-lang.org

Stack Overflow. (2021, August 2). *Stack Overflow Developer Survey 2021*. Stack Overflow Annual Developer Survey. Retrieved May 21, 2022, from https://insights.stackoverflow.com/survey/2021

XDA-Developers.com & Pulser_G2. (2015, August 19). *A Demonstration of Stagefright-like Mistakes*. XDA-Developers. Retrieved May 15, 2022, from https://web.archive.org/web/20150819162559/https://www.xda-developers.com/a-demonstration-of-stagefright-like-mistakes/