On Utilizing Rust Programming Language for Internet of Things

Tunç Uzlu, Ediz Şaykol
Beykent University, Department of Computer Engineering,
Ayazağa, 34396, İstanbul, Turkey
tuncuzlu9@gmail.com; ediz.saykol@beykent.edu.tr

***Abstract*—Rust, as being a systems programming language, offers memory safety with zero cost and without any runtime penalty like high level languages while providing complete memory safety unlike others like C, C++ or Cyclone. Todays world is in a transition from dumb devices to smart devices that are connected to the Internet all the time. Low cost embedded hardware is a key element for this kind of devices. Software needs to be smaller, lighter and power efficient. How one can operate with such limited hardware while preserving reliability? At the end, high level designs require runtime penalties while low level designs are known for memory unsafety and complicated design paradigms. Rust is higher level than other systems programming languages, has a rich standard library and compile-time abstractions for blazingly fast execution. While being completely available in mobile world, Internet of Things (IoT) devices are to be operated by all known mobile hardware as well. To this end, Rust, pushes limits of systems programming for two different views; first, at the core of hardware, running as daemon and talking to firmware, second, as a mobile controller software talking to mobile operating system. In this study, we summarize some concepts, employed in Rust, in terms of embedded systems development to clarify the appropriateness of using Rust within IoT world.**

I. Introduction

Rust[1] is a systems programming language that is being developed by Mozilla Foundation. Like other similar languages – C, C++, Cyclone or Assembly – Rust is low level, allows raw memory managements, and has a detachable standard library [1]. Unlike comparable alternatives Rust ensures compile time memory safety (even across threads) and offers rich standard library with functional elements. Cyclone is the closest language to Rust and aimed to provide memory safety on top of C language by preserving language structure that means with little to no modification, it is possible to reuse existing codebase. This rationale allowed porting applications from C to Cyclone so much easier [2], but at the end semantics are restricted. Memory safety with high performance is the raison d'etre for Rust language even for a recent bootloader design [3]. This is one of the main reasons that Rust enforces compile time semantics and be a complete memory safe language (even across threads). Rust also prevents access to unallocated, uninitialized, freed memory along with pointer addresses beyond data boundaries; which are seemed as fundamental rules for memory safety [4].

Rust is also the development language of Servo[2], Mozilla Foundations massively parallel web browsing engine, which is unique because of its concurrent process rendering and compositing steps [5]. Rust is open-source and hosted on Github. Nightlies are suitable for testing new features, embedding inline assembly and feature gates (language semantics that are not enabled by default, mostly experimental ones), which may be the key for communicating with helper co-processors in embedded Internet of Things (IoT) hardware.

Here, we intended to compare Rust with several programming languages based on the concepts that we extracted in our literature study. Since the main idea behind using Rust is programming a critical-and-safe low-level task with utilizing high-level programming concepts, designs with miniscule embedded hardware found on smart devices are a typical application for this purpose. The basic aim of our study is to make the Rust programming language useful on collection of IoT devices for both practitioners and technology developers.

The paper starts with presenting comparisons of Rust to some other languages in Section II. Then, in Section III, Rust language basics and the design choices on protocol level that make Rust suitable are mentioned. In Section IV, we discuss security issues with memory safety and the Internet access, and finally, Section V concludes our paper.

II. Comparison with Other Languages

First, we provide a comparison of Rust with several programming languages based on the concepts that we extracted in our literature study. The tabular comparison in Figure 1 indicates that Rust performs at a great balance with zero red/blue cells. Columns contain several architectural software concepts while implicitly covering safety, size, performance and energy consumption. Green cells indicate the best possible choice for the feature. Yellows indicate that, for the attribute, another language probably has a better perspective. Blues show that it may be possible to practice in the specified language, but involves feature gates, modification of the compiler, language extensions or require an extreme skill set.

MC (compiled to machine language) and ZCA (zero cost abstractions) are about the performance. Being able to compile directly to target hardware is the key for high execution speeds. For example, Python has to be compiled into C language

[1]https://www.rust-lang.org/

[2]https://servo.org/

first in order to produce an executable. With the help of modern build tools, it is extremely easy one to make builds for production and testing builds with a common Rust codebase. The safety guarantees with Rust contain compile time checks with runtime additions that are only baked when it is necessary. The result is C-like speeds with high level language semantics. As far as limited resources on IoT hardware is concerned, this actually helps a lot to decrease both space and time complexity.

IL (intermediate language), FSL (functional standard library), GN (generics) and PM (package manager) are about convenience. Rust utilize LLVM compiler infrastructure so that the same back-end optimizations for C and C++ are available also for Rust. The LLVM stack is open-source and has wide platform support. Most IoT applications share common design methodologies. Rust generics allows reusable designs. With rich and high level elements from the standard library, functional elements like higher order functions or lazy iterators are ready to use. Cargo, Rusts package manager, builds and tests Rust applications for multiple targets natively. Strict version tagging of dependent resources, scriptability through rs (filename extension of Rust code files) recipes and document generation shows us Cargo is capable enough for multi-platform oriented purposes. Cargo also compiles tools from other sources, accomplishing to have a rich environment.

IO (interoperable with de-facto systems languages), PA (platform agnosticity), NOS (operable in no-OS environment) and RMM (raw memory management) are about accessibility. Rust provides zero-cost and unsafe foreign function interface, making almost every resource from C language on hand. Cross compiled toolchains target most Linux distributions (other major operating systems are also supported). Rusts lacks garbage collector at core (although there are GC packages – called crates) so does non-deterministic delays. C languages have special standard libraries for extreme low-level purposes. Rust has libcore for such purpose. By sacrificing the actual standard library, one can operate on device drivers or when there is no operating system exists. With unsafe Rust, complex systems magics are easy; volatile memory regions, memory fence, unpadded data structures and so on.

TS (type safety), SMM (safe memory model) and SBT (safety between threads) are about preventing memory corruption and stability. Rust assures bindings on data has a unique owner with immutable borrows or only one mutable borrow. This simple ownership model is the foundation of memory safety in Rust even between threads. One of a kind safe inter-thread communication (with the help of special Marker Traits), read-write or read-only locking of bindings (not the code itself), compile-time static type checking and dynamic ownership/mutability checks with Cell/RefCell/RefMut synchronization primitives at runtime constructs a bucket of credibility. For instance, C++ is also a type-safe language, but implicit conversions between types weakens its type system a lot.

With embedded IoT applications, line between the firmware, kernel and the high level daemons are sometimes blurry. Most devices benefit from Linux kernel and Rust is located at firmware level; hardware specific system code and daemon applications. That may be the operating system counterpart in desktop computers. In complicated scenarios with programmable devices by the end user, Rust can be used in combination with byte-coded high level languages (e.g. Pawn). When user scripts are loaded from a USB mass storage device and interpreted by the ANSI-C abstract machine, Rust interaction is imminent; thanks to zero-cost Rust foreign function interface.

III. Related Rust concepts

Rust memory safety semantics are compile time checks. Run time support when it is not possible to enforce otherwise, e.g. bounds checking or I/O. Rust's nature of being decoupled from operating system core is significant improvement when it is compared with other systems languages. One example with C language is one can not disable signed integer overflow optimizations when IBM XL or Intel C compiler is used [6].

A. Ownership

Rust ownership semantics are similar to Singularity operating systems resource management model. Singularity exchange heap operates on interprocess communication on memory, is not garbage collected, reference counted, divides memory regions into chunks called region, creates a handle struct called allocation for read-only memory sharing [7]. Here idea of having a unique ownership of data is similar to Rust ownership model and read-only sharing is very similar to Rust's immutable borrowing. Rust RC and Weak reference counting primitives are also similar. Besides Rust also allows mutable borrowing (as long as there is one mutable borrower and there is no access of the owner) and multiple ownership via RefCell with dynamic checking (runtime performance penalty).

B. Unsafe Rust

Unsafe is the key for situations where application takes control of Rust safety mechanisms. In such occasion, one may workaround Rust's ownership and borrowing models; aliasing data by having two or more owners is possible. Mutating immutable data and casting to/from raw pointers is also possible. Unsafe Rust is more like designing in C language. Rust foreign function interface is build upon unsafety and making binding modules possible. Such modules are Rust's way to build safe abstractions over foreign function interface, low level memory operations, interactions with hardware and operating system so they include collection of unsafe functions (unsafe fn; entire function is unsafe and are also unsafe to call).

C. Protocols

Extensible Messaging and Presence Protocol (XMPP) and Message Queuing Telemetry Transport (MQTT) are suitable protocols for IoT service designs. Even though XMPP protocol – formerly named as Jabber – is designed as messaging protocol, its lightweight and text-based design makes it suitable for IoT applications. Facebook utilizes MQTT in their Messenger

	MC	IL	FSL	IO	GN	SBT	SMM	PM	TS	PA	NOS	ZCA	RMM
Rust	yes	LLVM	partial	fast FFI	yes	yes	yes	yes	yes	cross compile	Core	yes	Unsafe
C	yes	LLVM	no	yes	no	no	no	no	no	yes	partial	yes	yes
C++	yes	LLVM	no	yes	yes	no	no	no	no	yes	partial	yes	yes
Haskell	yes	type safe	pure	expensive	yes	partial	garbage collector	yes	yes	cross compile	complex	no	complex
Lisp	yes	C	partial	expensive	no	no	garbage collector	varies	no	special builds	complex	no	complex
Python	no	C	no	CFFI	no	no	garbage collector	yes	no	CPython	no	no	no
Go	yes	GCCGO	no	CGO	no	no	garbage collector	3rd party	no	cross compile	no	no	no
Swift	yes	LLVM	no	extension	yes	no	ARC	3rd party	yes	no	no	no	no

Fig. 1. Rust compared to other programming languages. MC=compiled to machine architecture, IL=intermediate language, FSL=functional standard library, IO=interoperable with de-facto systems languages, GN=generics, SBT=safety between threads, SMM=safe memory model, PM=package manager, TS=thread safety, PA=platform agnosticity, NOS=operable in no-OS environment, ZCA=zero cost abstractions, RMM=raw memory management (e.g. volatile or memory fence).

service; MQTT is actually designed for sending telemetry data to space probes [8]. This makes the protocol highly sensitive to bandwidth and energy consumption.

Applying XMPP with Rust applications is fairly possible. Rust language has strong C bindings, called foreign function interface. This makes every C codebase available for the effort. Talking to C routines is very fast as it is emulated through zero-cost (like most Rust abstractions) function calls. This makes catching program errors much easier, as errors are now encapsulated in unsafe blocks. Along with C interface, functional Rust elements makes the language highly suitable for design of XML parsers and socket/event handling.

D. Hands-on Experience

A typical event based approach states that the design requires two Rust channels (Due to Rust's module scopes, ones configuration may have multiple channel types; here we mean mpsc :: sync :: channel type), one for main event loop and another for foreign function interface [1]. As Rust channels are thread-safe, memory safety is assured automatically on main event loop. As the second loop works across foreign library boundaries, it is unsafe by nature. The channel type has handler of a tuple type (tx, rx). Front-end application owns receiving channel end and transfers ownership of the transmitting channel end to the XMPP back-end that later exposes two handles; Context and Plug. Context denotes an abstract interface to operating system IO and XML parser and Plug denotes active connection to host. As XMPP is fully decentralized, there should be a hosting end (or collection of servers as server-to-server communication is also part of the standard. This can be thought as a smart home environment with numerous devices that listens for incoming connections).

When the back-end engine is loaded from a module that is written in another language than Rust, it should provide data pointer parameters in callbacks. This is because second channel of Rust must be passed to the callback routines as an unknown data pointer. When an engine event is triggered, the back-end library passes this pointer back to Rust foreign function wrapper and then this address is casted (this is the major reason for using unsafety here) back to transmitting channel end. This channel data must be hold in heap area inside a Box, but under this circumstances, twice. The reason is first boxing converts the object into a Trait Object which is a pack of multiple pointers (actually a vtable like structure, but may be changed in the future as this is an internal detail) and then second boxing covers the Trait Object yielding a C compatible pointer. Every single event, sharing a mutual channel or not, can be paired with a Rust synchronization mechanism without using a hash table (C++ equivalent is map structure, not an actual hashmap).

In order to send commands to the back-end engine, Rust closures are implemented over traits [9]. There are three closure types; *Fn* (takes &self as argument and can not mutate the state), *FnMut* (takes &mut self as argument so can mutate the state) and the special *FnOnce* (takes self as argument, so the ownership, and can be called only once). Traits are interfaces much like Haskell typeclasses. In order to pass a closure argument to the back-end the closure should be a move. Moving closures take ownership of its elements (content of the closure) so when it is sent into another back-end thread through a Rust channel, its elements are ensured to be safe. The closure must have the ownership to transfer it to the channel later on and the contained data can not be dropped

(Rust equivalent of throwing out of memory via Drop trait) if they've been allocated on stack; moved closure creates a copied stack. With this kind of design, it is not possible to pass Rust closures by value into channel as their size (as they are simply a lambda function with anonymized name after all) is unknown at compile time. They are wrapped into a Box; heap allocated abstraction. This also makes static dispatching effectively possible.

XMPP standard includes a core specification and lots of module-like features, called XEPs. There are complex XEPs that depends on other XEP specifications. This design could be mapped directly to Rust module system. By providing attach and detach routines, it is possible to abstract each XEP and by including modules into another module, it is possible to emulate depended XEPs. Swapping modules at runtime is also possible. Even if this does not benefit from Rusts compile time checks, low memory constraints on embedded hardware may force runtime-loaded modules. Similar to Extensible Firmware Interface concept called dependency expression, each module may have a dependency string which is evaluated for every Rust XEP module and if the result is TRUE, corresponding module is dispatched. For application startup, the core module is hard wired. This recursive operation eliminates need for parent module to have knowledge about subfeatures.

E. Example toolset

IoT devices are integrated with other smart devices. Rust has very rich standard library and a strong package manager, Cargo. Existing Rust codebase is much larger for mobile devices than firmware foundations. For example, with the famous objc crate (Rust packages are called crates) allows Rust to send messages to Objective-C runtime. This infrastructure makes most Objective-C resources to be available to integrate with Rust projects. On daemon back-end, libstrophe, an XMPP library that runs on POSIX threads, may be a great choice. On mobile side, for example for Apple IOS operating system, XMPPFramework may be the choice as it is based on Objective-C and runs on operating systems native threads. The frameworks logging capabilities can be an extended interface of Rusts de-facto logging crate, *log*. An external log viewers can be connected to named pipe from this setup, resulting various toolsets from multiple disciplines utilized in a compatible fashion.

Through Rust foreign function interface, combined with Rust standard library, core library and makes Rust completely safe application designs combined with low level unsafe abstractions over operating system foundations; e.g. Grand Central Dispatch. Linker configuration can be achieved through creating a Cargo config file, called .cargo. This is need for configuring the Rust compiler, *Rustc*, is redundant as compiler configurations are more complicated than package managers. Complex resources, like macOS framework bundles, requires a target triplet in .cargo file; $[target.x86_64 - apple-darwin.pseudo-library]$ and two flags; -l for library name and -L for library path, mimicking GCC.

IV. Security

Current status of SOHO network devices should be noted here. More intelligent SOHO devices used in combination with IoT setups means more effective firewalling and handling of connection states. These devices utilize Linux operating system with similar configurations. Open source is key factor here, as this devices are responsible for firewall-ing against attacks that coming from outbound internet access. Nowadays these devices include Linux kernel version 2.6 because of possible outdated SoC chips, closed-source wireless or xDSL drivers. This branch was released in December 2003. Most popular LTS version of this kernel branch, 2.6.32, was released in December 2009, has lost the maintenance status in March 2016.

As closed source chip drivers tied to a single kernel version, it is not possible to follow future standards with these kind of hardware. It is sometimes possible to make simpler drivers, like xDSL, to work with radical firmwares by extracting and copying closed-source binary driver files (called blobs) into a newer operating system (often the opposite happens; called kernel backports). However this does not provide confidence about safety or reliability of the system. This is the only way for most xDSL or DOCSIS modulating hardware, at least for now. Wireless world has much better open source support especially devices supported by Linux kernel itself.

A. Rust safety

Billion dollar mistake [10], memory pointers that can be null, is just one of the memory problems with current systems designs. Current paradigm with these languages is a very over-powered tool which can be very powerful if only used correctly and highly depends on skill. Lack only one of memory safety, type safety or thread safety is enough to introduce runtime corruption. Memory malformation of application state caused by undefined behaviours of the language may not be detected with tests %99 of time and may yield extreme results, even travel in time [11], as it is undefined, but most of the time; just crashes. Rusts compile time ownership semantic prevents well known behaviours like used-after-freed descriptors or invalidation of iterators as unique owner of some data can only access it and borrow it immutably – with other languages, similar checks are not enforced by the compiler. Manual implementations with APIs are inconsistent and can be forgotten – [12] or borrow it mutably, but then can not access it until borrowing is handed back. Most checks are impossible to implement without compiler support.

Not only on application side, Rust is proven to be a solid foundation for building network infrastructure. Network Function Virtualization, NFV aims to replace network hardware, dedicated to specific networking tasks, with simulating software that runs in VMs (similar to container environments managed by provisioner tools). Virtual machine environment is a must as memory sharing should be prohibited.

This approach not only increases throughput also provides simpler upgradeability and testing with the cost of operating system virtualization overhead. A recent study study shows

safety ideas of Rust along with low level memory fine-tuning makes Network Function designs revisited without consolidating virtual environments [13].

Rust memory safety semantics checks for every possibility of memory unsafety. Unsafety of operations on sequential memory areas, invalidation of iterators negative indexing or overflowing are ensured to be non-existent. With lack of garbage collector and GC delays, results are much more deterministic.

B. IoT Security

IoT devices must comply with security rules such as confidentiality, integrity, availability, authenticity, non-repudiation; actually rules apply for any device today. Cryptography is at the core of security countermeasures. This requirements are adopted by encryption algorithms, hash functions, digital signatures and key exchange algorithms [14]. Because of always-online nature of IoT hardware and human routines are concerned; confidentiality is very serious as most of the subjected devices were not part of any network beforehand.

DARPA started a program in 1999 which predicted that systems can be built to tolerate successful attacks. However it is shown that this practice imposed more complexity and high resource usage [15]. For example; asymmetric cipher keys can be assigned to home gateways by the service provider of an IoT smart home provider [16]. With requirements of security, these powerful devices may be the center of cryptography at home. It is well known that these devices are targets of denial of service or zombie attacks along with other smart devices with network access.

V. Conclusion

In this study, we summarize some concepts, employed in Rust, in terms of embedded systems development to clarify the appropriateness of using Rust within IoT world. Rust, as being the most modern systems programming technology today, is ready to become the main language for IoT device daemons. Rust's ability to run incorporated with existing codebase and development environments is strong.

Compile time abstractions means no run time performance costs either from compile time safety checks or garbage collectors. Rust indeed has some compile time checks, but when it is not possible otherwise. That means lower time and space complexity in ordinary operations; preserving valuable CPU cycles for more complicated algorithms, richer user experience or lower energy consumption with miniscule IoT hardware.

Cross compiling Rust and its rich standard library to embedded word is possible and requires less to zero changes on Rust itself. On the other hand Rust indeed decreases possibility of buffer related attacks, firmware level security is still critical; from lowest level to applications, security is a whole across the system. Today memory unsafety causes serious problems in terms of security and stability. Hence adaptation of Rust is not economical or social, but rather intellectual.

Here, we also provide a comparison of Rust with several programming languages based on the concepts that we extracted n our literature study. A conceptual map is presented to clarify the understanding of the related concepts of Rust and hence, help the Rust programming language be useful on collection of IoT devices for both practitioners and technology developers on their critical-and-safe low-level task with utilizing high-level programming concepts.

References

[1] J. A. Holm, "Mozilla's Rust programming language at critical stage," https://opensource.com/life/14/6/mozillas-rust-programming-language-critical-stage, 2014.

[2] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney, "Region-based memory management in Cyclone," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, 2002, pp. 282–293.

[3] T. Uzlu and E. Saykol, "Utilizing Rust programming language for EFI-based bootloader design," in *2nd International Conference on Recent Trends and Applications in Computer Science and Information Technology (RTA-CSIT'16), CEUR Workshop Proceedings Volume 1746*, 2016, pp. 100–106.

[4] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdoglite: Hardware-accelerated compiler-based pointer checking," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'14)*, 2014, pp. 175–184.

[5] T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen, "Session types for Rust," in *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming (WGP'2015)*, 2015, pp. 13–22.

[6] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Undefined behavior: What happened to my code?" in *Proceedings of the Asia-Pacific Workshop on Systems (APSYS'12)*, 2012, pp. 1–7.

[7] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fhndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill, "An overview of the singularity project," Microsoft Research, Tech. Rep. MSR-TR-2005-135, 2005.

[8] L. Zhang, "Building Facebook Messenger," https://www.facebook.com/notes/facebook-engineering/building-facebook-messenger/10150259350998920, August 2011.

[9] A. Turon, "Abstraction without overhead: traits in Rust," http://blog.rust-lang.org/2015/05/11/traits.html, 2015.

[10] T. Hoare, "The billion dollar mistake," https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare, 2009.

[11] R. Chen, "Undefined behavior can result in time travel (among other things, but time travel is the funkiest)," https://blogs.msdn.microsoft.com/oldnewthing/20140627-00/?p=633, 2014.

[12] A. Hobden and Y. Coady, "Understanding over guesswork," https://hoverbear.github.io/rust-education-paper/paper.pdf, University of Victoria, Department of Computer Science, 2015.

[13] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "Netbricks: Taking the V out of NFV," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 203–216.

[14] A. Kanuparthi, R. Karri, and S. Addepalli, "Hardware and embedded security in the context of internet of things," in *Proceedings of the 2013 ACM workshop on Security, Privacy & Dependability for Cyber Vehicles*, 2013, pp. 61–65.

[15] B. G. P.E. Black, L. Badger and E. Fong, "Dramatically reducing software vulnerabilities," National Institute of Standards and Technology, Tech. Rep. Interagency Report 8151, 2016.

[16] X. Li, R. Lu, X. Liang, X. Shen, J. Chen, and X. Lin, "Smart community: An internet of things application," *IEEE Communications Magazine*, vol. 49, no. 11, pp. 68–75, 2011.