Rust as a platform for IoT

Part of Course ID2201 at KTH (2021)

Yannik Sander <yannik@kth.se>

May 25, 2021

Contents

Introduction

The IoT and Ubiquitous Computing already makes for a large share of software being written today. To prove secure writing such software should be ergonomic while staying performant and efficient, the potential for human error should be minimized.

Rust is a young programming language with a lot of promises, including being a viable platform for IoT of Ubiquitous computing. This essay will introduce Ubiquitous Computing and IoT, then, draw a picture of the current Rust ecosystem relevant for these fields and finally analyze how well this ecosystem can provide for the demands of the field.

Ubiquitous Computing and IoT

Introduction to the character of ubiquitous computing

The notion of Ubiquitous Computing, often and in the following simply referred to as **Ubicomp**, has been established by Mark Weiser in his 1991 paper envisioning "The Computer for the 21st Century"[1]. In a time where computing was visible, immobile and rear compared to today's standards, Weiser spoke of a "disappearance" of technology. Surely, he did not speak about a decrease in computing, but the opposite. The disappearance was about the obvious presence of said technology. Weiser predicted a world where computing is omnipresent – *ubiquitous* – "weaved into everyday life." On the one hand, it was about the mobility of computers, i.e. being able to take computing everywhere. Yet, this doesn't cover it completely! Indeed, it was the unintrusive enhancement of people's life that defined Ubicomp.

In the modern day, Ubicomp has become an important part of human-computer interaction (HCI) implementation and research. Especially being aware of the user's context and acting

upon that is an important aspect of Ubicomp relevant to HCI and carrying numerous opportunities for future advancements in computing [2]. There have been great developments in technology since Weiser formulated the concept of Ubicomp that enable many of his ideas. Poppe et.al. [3] pointed out critical developments in this regard.:

Context Awareness and Pro-Activeness Multitudes of sensors both owned (or even worn) by the user or present in the public domain allow the greater possibility to take user context into account. This includes classifying user's actions, emotions, health and location and allows them to provide services related to that. This might be sports tracking e.g. analytics and recommendations in tennis [4] or granting access to public transport without any interaction, enabled by face detection [5]. Further, context awareness empowers services to be gradually more proactive, likewise reducing the amount of interaction required by a user, although "mixed-initiative" is said to be more appropriate to HCI. Siri [6] and Google Assistant [7] are examples of such context-sensitive, mixed-initiative services. They provide information when they are queried by a user, most notably facilitating non-physical interaction with distributed devices (i.e. through voice commands handled by a supported client device). They might also present information on their own, based on context and need.

Adaptability *Need* is interesting on its own. It might mean external factors such as emergency warnings, but often employs a different concept important to Ubiquitous computing, namely, *adaptability*. Evers et.al.[8] claim that "future computing systems must adjust to the user's situations, habits, and intentions." Proactiveness, as mentioned above, is much more useful, if not only useful, if it supports and anticipates the user's intentions and develops with their behaviour. As such, it's imperative to not only collect user context but also user feedback on the actions anticipated.

Intelligence Tangential to the ability to learn from user feedback is the perceived intelligence of a Ubiquitous system. As humans, we are used to expecting decent levels of intelligence in natural interaction with each other. In effect, to be perceived as a *natural* part of one's environment, rather than being a tool, ubiquitous technology needs to show intelligence too. Adaptivity and pro-activeness, as discussed before, are some aspects of this. Additionally one might ask for additional criteria such as an ability to reflect and anticipate consequences, improve their behaviour and show diverse strategies as well as *natural* social competence.

Speaking of intelligence, one can discern different types of intelligence by the agent that shows it. Things or machines that display intelligence are typically considered robots. User software becomes a (software) agent or softbot. While softbots (for instance the aforementioned personal agents) can already be integrated into the environment and provide non-physical interaction, going a step further one can also separately distinguish smart environments. These are often referred to as implementations of ambient intelligence. We see intelligence embedded into objects in our environment [9] such as appliances, thermostats and similar devices. Research might go even further exploring rooms, that sense the user's presence, can store and prepare different states for different users [10, p. 21] and be augmented virtually [1].

In fact, augmented reality was also envisioned by Weiser [1] as part of a ubiquitously computing future. Several projects are pushing the idea of AR implemented in projects of varying comfort in form of the Google Glasses and Microsoft Holo Lens or smartphone-based solutions like Google Lens.

Summary of the Dimensions of Ubicomp Summarizing the nature of Ubiquitous computing one can distinguish advances on different dimensions. First, implementations of Ubicomp might have varying *distance* to the user. It can be public domain face detection based access control, or intelligent rooms. Closer to the user

there are wearables or smart fabrics. In recent days project like Neuralink [11] picture a future with even closer integration of computing. Second, different grades of *artificial intelligence* are shown. Systems that interact close to the user are supposed to do so naturally, i.e. intelligently. The less direct the interaction the less the requirement for pro-activeness, talking about face detection as an example. Weiser himself coined another dimension, namely *size*:

> "Inch scale, foot scale and yard scale devices."

Computing nowadays can be as big or bigger than smart screens, down to tablet and smartphone/smartwatch size. Yet, even smaller computing is present in our environment. Credit cards, contactless keys, or glucose meters are examples of such smaller and certainly much more transparent areas of computation with medical devices reaching even greater records in smallness.

So far, Ubicomp has been mainly described as a form of HCI, providing ways for users to interact more or less directly with a greater service. Ubicomp itself while popular in academics (listing more than 1.5M search results on Google Scholar[1]) hasn't become nearly as present in the industry and everyday life as a term. Possibly due to its nature being more of a concept without clear and concrete borders as seen above. Instead, the term *Internet of Things* (IoT) experienced a phase of ubiquity in industry and consumer electronics. What is IoT then?

The Internet of Things

While Ubicomp as a concept seems to concentrate more on the connection with humans and the possibilities it offers to them, IoT has a greater impact as a marketing term for connected devices. In a way, it's a more practical term than Ubicomp. Yet, that does not mean it is a more concrete one. Applications of IoT include Home Automatization, Smart Cities, Media Consumption and Transportation to name a few many of which are also part of Ubicomp. The difference to Ubicomp is that IoT describes *actual* systems/networks of devices that work together and communicate, as well as their protocols and standards.

As shown in an article in *Business Horizons* [12] numerous artefacts are part of the IoT. It is the nature of a system that qualifies it as part of *the* IoT. Data is produced at one end by wirelessly connected sensors, send over a network, processed by specific middleware running *in the cloud* and driving IoT based applications.

Only considering consumer applications the market for IoT is enormous. Home automation for example has evolved rapidly over the last years, with multiple applications by competing vendors reaching from lighting overheating to property security and more. The great interest in this market led to a substantial fragmentation of the market on nearly every layer of IoT. In the pursuit of standard multiple communication protocols evolved. Nowadays, Zigbee, Z-Wave, Bluetooth LE and WiFi are the dominant standards to build networks[13]. Yet, device protocols often remain largely incompatible still.

If building on open standards, the entrance to IoT has become quite simple. Hubs provide an interface to compatible connected devices or connect them to cloud-based services. Yet there is little standardization around the interaction between different devices' specially if coming from different vendors.

Looking at industrial contexts, an apparent difference is the predominance of sensors at the bottom of the network becomes apparent. IIoT is characterized by a multitude of wirelessly connected actuators and sensors [14]. Unsurprisingly, IIoT generates a lot of data, which needs to be stored and processes or analyzed[15]. This fact strongly motivates cloud computing or even more immediate processing at the *edge of the network*. [16]. Additionally, the uniqueness of many applications implies that there are even incompatibilities between systems than in the consumer market.

[1] May 4th, 2021

To summarize, IoT consists of three main elements each of which can vary in complexity based on the application as crystallized by Jayavardhana Gubbi in a 2013 paper [17].

1. **Hardware** is the common term for sensors, actuators and communication drivers
2. **Middleware** provides intermediate analytics and data storage
3. **Presentation** conveys the findings to the end-user

One might additionally include **Software**, especially protocols, in the list. In the following, this essay focuses on software related to the first two points.

Rust

Rust is a relatively modern programming language that was first introduced in 2010 by Mozilla as a basis for their experimental browser engine Servo[18] parts of which are now driving the Firefox Browser[19]. Its trifecta of speed, safety and concurrency caught peoples interest early on. Since the beginning rust strived to provide greater safety through an advanced type system. By design, Rust disallows concurrent mutable access to the same data. Instead, it employs the concepts of data *ownership* and *borrowing*. At compile-time, Rust can resolve how long references are used and when they are cleared up. In effect, (modern) Rust does not implement a garbage collector. This and the fact that it is compiled to native code through LLVM put it in the same category as other unmanaged languages such as C/C++ and account for Rust's performance. The first stable version of Rust was released in 2015[20]. Since then public interest grew starkly, due to its promises.

By now, Rust has been voted the "most-loved" language since 2016 by developers on StackOverflows yearly survey [21]. Its today's most convincing features are summarized by Jake Goulding[22] in a blog post from January 2020. Firstly, its versatile and ergonomic type system enables very practical safety measures, for example, replacing `null` pointers for more expressive and safe `Option<T>` types and enforcing the handling of errors through a `Result<Error, T>`. Additionally, these are also examples of Rust's capability of algebraic data types. The aforementioned garbage collection model – or the lack thereof – is as well highly appreciated by users of the language as it decreases the applications memory footprint dramatically. The possibility of *safe* direct memory access has likewise driven Rust to be an aspiring candidate for embedded{@} devices as well as recently becoming an officially supported option for Linux kernel module development[23], not at last because sticking to Rust's compiler enforced rules drastically reduces the possibility of segfaults. Segfaults, typically occurring when accessing invalid memory, are by default prevented by Rust's memory design.

Additionally to the language design, Rust has built a thriving ecosystem. Its standard build tool and package manager Cargo[24] is the pivoting point of this ecosystem. Using cargo one can easily manage dependencies, config feature flags, run tests and much more. It also offers great extensibility through custom commands and built-in integrability with IDEs [25]. Cargo links in and provides tools for publishing libraries on its package library crates.io[26]. These libraries referred to as *crates* are considered to be one of Rust's most important features on their own. Traditional languages such as C/C++ do not have any standard package manager, libraries are typically installed as precompiled binaries that need to be linked at compile time or runtime in case of shared objects. This requires the developer to include header files that are only resolved using a basic preprocessor, install these libraries separately and track/require them using third party tooling with little control over the actual version being used leaving many security issues to be dealt with by the user of the software and OS maintainers. C++ recently added support for modules[27] solving some problems related to header files but remains fragmented in general. Rust got inspired by more modern and ergonomic solutions of more recent languages such as NPM[28].

The Rust Ecosystem for IoT

In the introduction, Rust's ecosystem was outlined. Focusing on IoT one needs to take a deeper look into the accompanying tools and libraries. This essay will introduce key technologies and concepts that enable the development of IoT devices and related edge computing.

When speaking of IoT ARM is by far the leading manufacturer of Chipsets used at the edge of the IoT[29] and embedded devices such as sensors. As such, to be a viable option to cover the IoT space as a language, support for ARM-based processors is imperative! Hence special focus will lie on ARM support in Rust.

Rust on Microcontrollers

Tools

A major component of the ecosystem of a programming language are tools that simplify or automate the development processes. These processes can become highly complex even for rather simple projects. For instance programming, a common microchip, requires a debugger, a connector to the on-chip debugger and the programmer and the build tool to work together. The latter also needs to be configured for the programmed chip. Several tools that have been developed try to shrink the associated learning curve and strive to allow for greater productivity quicker.

Being based on LLVM Rust supports a multitude of platforms [30] including many ARM platforms. Cargo complements this by offering an interface to cross-compile to foreign architectures. Additionally, rustup[31] provides an interface to easily acquire toolchains for these architectures and simplifies keeping track of the fast-paced releases of the Rust language. Combining these tools, cross[32] has been developed by the rust-embedded working group which uses isolated docker containers to minimize the efforts required and possible failures of setting up a development environment by providing a managed prepackaged solution. With this running and testing code for different architectures becomes as easy as

```
$ cross test \
  --target mips64-unknown-linux-gnuabi64
```

Targeting microcontrollers, in particular, the knurling project [33] develops tools that make embedded development more seamless. `probe-run` is a project that integrates downloading binaries to controllers, and running code, as well as connecting debuggers with cargo and can therefore be easily integrated with IDE's. `defmt` significantly reduces resource overhead of logging on microchips and has been found to offer a highly integrated debug process [34].

Abstraction layers

Speaking about abstraction layers one must first understand why they are needed. Programming microcontrollers is flooded with hardware-level interaction, unsurprisingly. While rust is capable of doing these accesses, in many cases *some* of Rusts safety measures need to be disabled. While more is possible in these `unsafe` environments, obviously one strives to reduce the use of `unsafe`. Besides safety, ergonomics and compatibility are more reasons to ask for abstractions. Rust is known for its capabilities to bring these virtues to its users in other areas already due to expressive Generics and its trait system. In the context of embedded programming, this has enabled people to create various levels of abstractions on top of the lowest levels of interaction with the hardware.

Accessing the hardware Peripherals on microcontrollers are configured through so-called memory-mapped registers. Manipulating the state of these registers changes how the external connectors to the chip behave, whether they are inputs or outputs, digital or analogue. Also, internal structures can be controlled this way, e.g. timers can be set and reacted upon. Unfortunately, there is no common interface to these registers not only due to the number of different manufacturers but also different chip design and application.

While configuration and layout differ, it does

not do so undocumented. In fact for long SVD files [35] are being made available by manufacturers describing the chip layout formally. In Rust, this is made use of to create so-called *peripheral access crates* (PAC). Using a `svd2rust` [36] one can *generate* a rust library that implements a **safe interface** to all of the specified registers including context-based functions, such as being able to write or read from pins, or start timers using a method rather than setting bits manually.

Abstracting hardware functions PACs do a great job making raw hardware accessible by Rust in an automated and safe way. Building on top of this, one might perform standard operations such as communicating to peripherals connect to USB, enable timers and so on. While building this functionality from the ground up based on peripheral access, a safer and more portable solution is building on a shared abstraction. Such an abstraction is provided by the `embedded_hal`[37] crate.

The functionality provided by `embedded_hal` fulfils some important requirements:

1. It is independent of any specific chip
2. Does not make restricting assumptions about how it is used on a specific chip
3. Provides low-cost abstractions that are compassable into higher-order abstractions (note that `embedded_hal` is still a *very* low-level abstraction)
4. Stemming from the previous point: Offers sufficient freedom and capabilities to base device-independent drivers upon.

Note, that `embedded_hal` does not implement most of the functionality, but defines interfaces that are eventually implemented for a specific chip or family of devices.

Drivers

Apart from accessing mere hardware, the most important aspect of embedded development is, as in non-embedded scenarios, processing data, and providing functionality. In the context of IoT data is typically produced by the periphery, and communicated over some network channel, it is still the *internet* of things. Enabling this, one finds themselves at a gap. So far, the discussed abstractions merely provide hardware access. Yet, communication, in particular, requires conformity to often complex protocols (i.e. IEEE802.11/WLAN[38]). Implementations for these protocols readily exist in C, less so in Rust, often because the modems are using more niche platforms, to begin with. Instead of reimplementing the existing C implementation for those modems, Rust focuses on offloading this functionality. Offloading means to employ a second chip running a firmware that drives a communication module and exposing the data access through a firmware specific (serial) interface. Drivers have been implemented for all sorts of such devices and often make use of the aforementioned `embedded_hal` to be usable from any host device.

While serial protocols such as USB, RS232 or i2C can be part of a HAL, data protocols like AT[39] are implemented separately. Crates like `atat`[40] transparently offer access to these protocols. Building on that, driver crates for popular networking modems are already available. With IoT in mind, we can find drivers for cellular access[41] or short-range networks[42] that connect to u-blox[43] devices. The drogue IoT project [44] not only brings support for common network standards like WiFi or LoRaWAN but also abstracts these to a transparent network interface, such that from Rust each of these network gateways can be used the same way providing TCP/UDP sockets. Building on this network abstraction the project also implements an MQTT and HTTP client.

Notable Mentions

LoRaWAN[15] is known for its application as the basis for IoT. Especially The Things Network[45] plays a major role in pushing LoRa by providing a shared infrastructure that is energy efficient, yet reliant, open and secure. Incidentally, crates to create clients to this network already exist.

Leaving Microcontrollers

At this point, Rust's support for microcontrollers was comprehensively presented… Although a lot of the IoT is implemented on the smallest of processors, often one has more resources to spare. Devices that could be described as "raspberry pi sized," can run a supported operating system (e.g. GNU/Linux) on a higher architecture, such as `aarch64`. Consequently, they profit from full Rust-Support. These offer more versatile tools and capabilities to connect to complex technologies such as Bluetooth, or processing greater amounts of data, such as camera feeds.

Rust on the Edge

In recent years WebAssembly [46] (WASM) has been growing as an OS-independent platform, meant to run programs in web-browsers at near-native speed in secure sandboxes. Rust as a language has been pushing this development forward, by language support and tooling. Not only have the three biggest freestanding WASM runtimes adopted Rust as their implementing language[49], but also there have evolved standard tools to integrate WASM into your JavaScript Codebase [50] and library support to narrow the gap between the Rust/WASM world and the JavaScript runtime[51].

Companies like *fastly* and *Cloudflare* have developed services that facilitate this platform to offer easy entrance to Edge Computing. Cloudflare Workers [52] offers the infrastructure for reliable functions on the web that can act as an ingress point for IoT devices. Workers run code compiled to WebAssembly which makes them a ready target for Rust.

Rust as a platform

In the previous section, a multitude of applications and capabilities of the Rust Language has been presented. While the ecosystem is large, it is important to also analyze it with an eye on qualitative factors to come up with a convincing conclusion about the usability of Rust as a Platform for IoT today. As rust is still evolving, many things will still improve. This essay aims to summarize the current developments and make an educated guess where Rust is heading.

Today

Rust is known for assessing its performance in many areas publicly in the form of "are we X yet" websites[54]. Unfortunately, for the domain of embedded/IoT, the community has not yet started such a project. Yet, as that form has proven very informative, this essay will adopt a similar approach.

++ Rust has stable and mature support. You can use Rust for this
+ Rust offers some support/development. Think twice.
± Ideas are there but little has evolved from it.
- Close to nothing has been developed. You are on your own, not recommended to use Rust here.

The good

In some regards, rust can already shine, although it might need some polish in some places. Especially the strong features of Rust, tooling and performance, can shine too in the IoT context.

Performance and Ergonomics (++) One of the strongest points to make about Rust is probably its performance. This does not mean solely its runtime performance but also its development process.

Rust is often hailed for the high-level elements that make it look and act like a general-purpose language in many regards. At the same time, it embraces the concept of "Zero-Cost Abstractions" that let it produce highly optimized code without accepting drawbacks on its high-level features.

Yet in areas where every byte counts, with rust one, has to trust on the optimizer to produce sufficiently small binaries. Which becomes harder given how easy it is to add de-

pendencies to a project. Projects exist to help monitor the size of binaries but the main problem remains.

Looking at it in another way, given how high-level rust can be, memory accesses are not as obvious as they are with C. Clearly, this can cause unexpected problems in performance, especially on microcontrollers with limited memory speeds.

Tools (++) Rust is known for its great tooling. This doesn't stop in the world of embedded systems and IoT. For one, Rust is pioneering the world of WebAssembly. Additionally, the entry into Embedded systems is made greatly easier given the rust tooling.

As we saw with the abstractions above, embedded Rust does not exclude the possibility to use cargo and its package management. In fact, it even provides a measure to tell whether a library can run on embedded devices or not, precisely as long as it does not use the standard library that is built on top of specific operating systems. Such crates are marked as `#[no_std]`. It doesn't stop there. We saw projects automatizing the whole process of downloading binaries and running/debugging them through standard cargo invocations that integrate well with IDEs.

Of course, not everything is perfect in this regard area yet. As a lot of Rust's tooling is automatized, as a user one faces a rather high-level view of the process. While this can be desirable, it reduces the account one has to tell what parts are causing errors. Also, due to abstraction over multiple interfaces, the functionality provided might be less than what would theoretically be possible using those tools directly, at which point using Rust might become more of a burden as configuring these tools to fit Rust might not be trivial in every case.

A similar thing can be said about Rust for Edge Computing using WebAssembly. We have seen it as the driving language for modern runtimes and offering rich library support[55]. As such given its tools, it is easy to integrate them into existing platforms or important in the domain of Edge Computing write high performant workers in it.

Frameworks and Libraries (+) The presentation shed light on some of the most influential projects. While on the WASM front Rust has already developed a mature environment. In the space of embedded devices, libraries while bringing support for many devices, do not enjoy the same kinds of maturity and maintenance.

The `embedded-hal` is a great leap toward a unified API on microcontrollers and is already heavily used. Yet, many projects have implemented drivers independently or base on incompatible versions. Drivers are generally added more as an implementation to tick the boxes for a specific use case and therefore do usually not cover the available functionality.

`drogue-wifi` for instance implements a driver for the WiFi breakout board ESP8266. Yet, while the target chip is capable of a whole range of functionality, the driver only implements a limited subset of that such that it fits the drogue project. More extensive support would be desirable to use rust more ergonomically and ease the development.

The bad

As a comparatively young language, especially compared to its contenders in the embedded world, naturally, Rust has a set of drawbacks mainly connected to its development pace and lack of maturity and experience in the industry.

Documentation (+/±) Generally, Rust is known for its great documentation. There is even specialized tooling around it. Rustdoc[56] is the standard tool to generate documentation from rust source code. Docs.rs[57] adds to this hosting documentation for the whole crates.io index of packages. The fact that all documentation is entangled this tightly is a major win to the whole community and aids development dramatically.

Rust has also evolved the mdBook[58] tool. It is used throughout the rust community to as-

Table 1: Summary of Rust's ecosystem today

Ecosystem element	Maturity
Performance and Ergonomics	++
Tools	++
Frameworks and Libraries	+
Documentation (Rust)	+
Documentation (Libraries)	±
Stability	±
Framework Interoperability	±
Architecture Support (not ARM)	-

well. RTIC[59] for example promises to provide a framework that works concurrently by managing interrupts and resources. Yet, it brings in a very different way to set up projects that complicate how to get started with the whole system especially as examples are scarce here too.

The ugly

Lastly, some tasks are so far virtually impossible to achieve with Rust or require a lot of work to be invested by the user.

~~ARM~~ (-) We have seen that on ARM Rust offers great support. Indeed the stm32-rs group, for example, tries to provide embedded-has implementations for all chips manufactured by STM. Other ARM vendors have similar good support.

It is once one asks to write software for other architectures. First, they are limited by support for these architectures of the LLVM backend. The popular IoT platform Arduino for instance runs on AVR based chips which require a fork of LLVM to be programmed for In Rust. Extensa, the architecture employed by the ESP8266/ESP32 chips also requires additional care. These reasons drive the current disinterest in writing software for these chips (as it is not possible or difficult) which in turn affects the motivation to bring support for the platforms to Rust in the first place.

In the future

Today, Rust's ecosystem is not all roses. A lot of things are missing still. Yet, observing the community fundamental steps are being taken. Looking at the future there are a few clear indicators that Rust will become a growing influence for not and ubiquitous computing.

Rust Foundation

More generically, the advent of the Rust Foundation [60] will eventually also benefit the embedded section of the language. With greater structural organization and backing from industry-leading companies such as Google and Amazon, Microsoft, Huawei and Facebook, Rust manifests itself as a credible choice.

Embedded-WG

The embedded working group has been mentioned throughout this essay. They have proven themselves as the originators of remarkable work that has brought rust a long way. Its projects will for sure continue to improve the experience of embedded development in Rust.

Ferrous

Ferrous Systems, the leading force behind the excellent knurling project, is committed to further invest in Rusts embedded future. Ferrous already contributes to the Rust Open Source community in many ways, and with knurling still being a young project, its influence in the embedded world is still to be expanded.

Libraries/Tools

Finally, with more companies and individuals committing to Rust as their language of choice, albeit its current pitfalls will eventually populate the language with more helpful libraries and tools. Especially, once fundamental libraries such as the `embedded-hal` a point of stability, libraries building on these are expected to follow.

Taking into account WebAssembly, we see that

with its adoption in all major browsers and the stabilization of its specification, WASM is there to stay. Rust has been influencing its development a lot until today

Conclusion

Throughout this essay, The domain of Ubiquitous Computing and IoT has been described in detail. Building on that the programming language Rust has been closely examined on its capabilities to fulfil the needs of this field. We have seen which infrastructure drives Rust's support of embedded programming, how WebAssembly enables Rust to play a leading role as a language to implement functions on the Edge. Apart from this descriptive part, we put the available ecosystem into perspective, pointing out its strengths and current weaknesses.

Finally, we can conclude that Rust is perfectly capable of doing specific tasks, in the area of Embedded Computing and more so on higher levels of the Internet of Things, such as lightweight computing on the edge and the implementation of backend services. Yet, it shows that Rust is a fairly recent language. As such parts of its ecosystem, relevant to IoT, are still evolving, lack even some significant foundations and are far from stable. For early adopters and the generally curious Rust still offers the foundations on which one can build their own solutions, albeit without providing the maturity of decades of development. On the bright side, we see several companies like Drogue, Ferrous Systems and other independent groups, doing exactly that. Crucial foundations are in active development and promise a brighter future for Rust.

Condensing this essay into one sentence

> Rust shows the potential to become the IoT platform of choice in the future, providing speed, ergonomics and safety, but does not show the maturity to be readily used as such without thorough consideration.

References

[1] M. Weiser, "The Computer for the 21st Century," p. 8.

[2] A. Dey, J. Mankoff, G. Abowd, and S. Carter, "Distributed mediation of ambiguous context in aware environments," in *Proceedings of the 15th annual ACM symposium on User interface software and technology*, Oct. 2002, pp. 121–130. doi: 10.1145/571985.572003.

[3] R. Poppe, R. Rienks, and B. van Dijk, "Evaluating the Future of HCI: Challenges for the Evaluation of Emerging Applications," in *Artifical Intelligence for Human Computing*, 2007, pp. 234–250. doi: 10.1007/978-3-540-72348-6_12.

[4] M. Sharma, R. Srivastava, A. Anand, D. Prakash, and L. Kaligounder, "Wearable motion sensor based phasic analysis of tennis serve for performance feedback," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Mar. 2017, pp. 5945–5949. doi: 10.1109/ICASSP.2017.7953297.

[5] T. Li, "You will soon be able to pay your subway fare with your face in China," Mar. 13, 2019. https://www.scmp.com/tech/innovation/article/3001306/you-can-soon-pay-your-subway-ride-scanning-your-face-china (accessed May 04, 2021).

[6] "Use Siri on all your Apple devices." https://support.apple.com/en-us/HT204389 (accessed May 04, 2021).

[7] "Google Assistant." https://developers.google.com/assistant (accessed May 04, 2021).

[8] C. Evers, R. Kniewel, K. Geihs, and L. Schmidt, "The user in the loop: Enabling user participation for self-adaptive applications," *Future Generation Computer Systems*, vol. 34, pp. 110–123, May 2014, doi: 10.1016/j.future.2013.12.010.

[9] D. J. Cook, J. C. Augusto, and V. R. Jakkula, "Ambient intelligence: Technologies, applications, and opportunities," *Pervasive and Mobile Computing*, vol. 5, no. 4, pp. 277–298, Aug. 2009, doi: 10.1016/j.pmcj.2009.04.001.

[10] E. Eliasson, "Secure Internet Telephony: Design, Implementation, and Performance Measurements," p. 80.

[11] A. N. Pisarchik, V. A. Maksimenko, and A. E. Hramov, "From Novel Technology to Novel Applications: Comment on 'An Integrated Brain-Machine Interface Platform With Thousands of Channels' by Elon Musk and Neuralink," *Journal of Medical Internet Research*, vol. 21, no. 10, p. e16356, Oct. 2019, doi: 10.2196/16356.

[12] I. Lee and K. Lee, "The Internet of Things (IoT): Applications, investments, and challenges for enterprises," *Business Horizons*, vol. 58, no. 4, pp. 431–440, Jul. 2015, doi: 10.1016/j.bushor.2015.03.008.

[13] S. Elhadi, A. Marzak, N. Sael, and S. Merzouk, "Comparative Study of IoT Protocols," Social Science Research Network, Rochester, NY, SSRN Scholarly Paper ID 3186315, May 2018. doi: 10.2139/ssrn.3186315.

[14] F. Foukalas, P. Pop, F. Theoleyre, C. A. Boano, and C. Buratti, "Dependable Wireless Industrial IoT Networks: Recent Advances and Open Challenges," in *2019 IEEE European Test Symposium (ETS)*, May 2019, pp. 1–10. doi: 10.1109/ETS.2019.8791551.

[15] "What Edge Computing Means for Infrastructure and Operations Leaders." //www.gartner.com/smarterwithgartner/what-edge-computing-means-for-infrastructure-and-operations-leaders/ (accessed May 06, 2021).

[16] "Real-Time Distributed Computing at Network Edges for Large Scale Industrial IoT Networks." https://ieeexplore.ieee.org/abstract/document/8495797?casa_token=CHZD9tpi9_cAAAAA:saG8dFLJX9KMntLMllGoBLn5JZWpOs8cAv0sJu9oPZINmQ3qzV5ecEDQzByUTTupjNtl_MkOfRc (accessed May 04, 2021).

[17] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, Sep. 2013, doi: 10.1016/j.future.2013.01.010.

[18] graydon, "Project Servo." Available: http://venge.net/graydon/talks/intro-talk-2.pdf

[19] "Quantum - MozillaWiki." https://wiki.mozilla.org/Quantum (accessed May 13, 2021).

[20] "Rust/RELEASES.md at master · rust-lang/rust." https://github.com/rust-lang/rust/blob/master/RELEASES.md (accessed May 13, 2021).

[21] "Stack Overflow Developer Survey 2020." https://insights.stackoverflow.com/survey/2020/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2020 (accessed May 03, 2021).

[22] J. Goulding, "What is Rust and why is it so popular?" Jan. 20, 2020. https://stackoverflow.blog/2020/01/20/what-is-rust-and-why-is-it-so-popular/ (accessed May 13, 2021).

[23] "Rust in the Linux kernel." https://security.googleblog.com/2021/04/rust-in-linux-kernel.html (accessed May 04, 2021).

[24] "Introduction - Discovery." https://docs.rust-embedded.org/discovery/ (accessed May 04, 2021).

[25] "External Tools - The Cargo Book." https://doc.rust-lang.org/cargo/reference/external-tools.html (accessed May 13, 2021).

[26] "Crates.io: Rust Package Registry." https://crates.io/ (accessed May 05, 2021).

[27] corob-msft, "Overview of modules in C++." https://docs.microsoft.com/en-us/cpp/cpp/modules-cpp (accessed May 05, 2021).

[28] "Npm." https://www.npmjs.com/ (accessed May 05, 2021).

[29] "Who is the World's Leading IoT Chipmaker?" Sep. 13, 2020. https://www.nanalyze.com/2020/09/worlds-leading-iot-chipmaker/ (accessed May 05, 2021).

[30] "Platform Support - The rustc book." https://doc.rust-lang.org/nightly/rustc/platform-support.html (accessed May 05, 2021).

[31] "Introduction - The rustup book." https://rust-lang.github.io/rustup/ (accessed May 06, 2021).

[32] *Rust-embedded/cross*. Rust Embedded, 2021. Accessed: May 13, 2021. [Online]. Available: https://github.com/rust-embedded/cross

[33] "Knurling-rs." https://knurling.ferrous-systems.com/ (accessed May 13, 2021).

[34] "Using GDB and defmt to debug embedded programs." https://ferrous-systems.com/blog/gdb-and-defmt/ (accessed May 06, 2021).

[35] "SVD Description (*.svd) Format." https://www.keil.com/pack/doc/CMSIS/SVD/html/svd_Format_pg.html (accessed May 06, 2021).

[36] *Rust-embedded/Svd2rust*. Rust Embedded, 2021. Accessed: May 13, 2021. [Online]. Available: https://github.com/rust-embedded/svd2rust

[37] "Embedded_hal - Rust." https://docs.rs/embedded-hal/0.2.5/embedded_hal/index.html (accessed May 06, 2021).

[38] "IEEE Standard for Information technology–Telecommunications and information exchange between systems Local and metropolitan area network–Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 5: Preassociation Discovery," *IEEE Std 802.11aq-2018 (Amendment to IEEE Std 802.11-2016 as amended by IEEE Std 802.11ai-2016, IEEE Std 802.11ah-2016, IEEE Std 802.11aj-2018, and IEEE Std 802.11ak-2018)*, pp. 1–69, Aug. 2018, doi: 10.1109/IEEESTD.2018.8457463.

[39] "Hayes command set," *Wikipedia*. May 04, 2021. Accessed: May 06, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Hayes_command_set&oldid=1021310741

[40] *BlackbirdHQ/atat*. Blackbird, 2021. Accessed: May 13, 2021. [Online]. Available: https://github.com/BlackbirdHQ/atat

[41] *BlackbirdHQ/ublox-cellular-rs*. Blackbird, 2021. Accessed: May 06, 2021. [Online]. Available: https://github.com/BlackbirdHQ/ublox-cellular-rs

[42] *BlackbirdHQ/ublox-short-range-rs*. Blackbird, 2021. Accessed: May 06, 2021. [Online]. Available: https://github.com/BlackbirdHQ/ublox-short-range-rs

[43] "U-blox." https://www.u-blox.com/en (accessed May 06, 2021).

[44] "Drogue IoT." https://github.com/drogue-iot (accessed May 13, 2021).

[45] "The Things Network." https://thethingsnetwork.org/ (accessed May 06, 2021).

[46] "WebAssembly." https://webassembly.org/ (accessed May 10, 2021).

[47] *Wasmerio/wasmer*. Wasmer, 2021. Accessed: May 13, 2021. [Online]. Available: https://github.com/wasmerio/wasmer

[48] *Bytecodealliance/wasmtime*. Bytecode Alliance, 2021. Accessed: May 13, 2021. [Online]. Available: https://github.com/bytecodealliance/wasmtime

[49] *Bytecodealliance/lucet*. Bytecode Alliance, 2021. Accessed: May 13, 2021. [Online]. Available: https://github.com/bytecodealliance/lucet

[50] *Rustwasm/wasm-pack*. Rust and WebAssembly, 2021. Accessed: May 13, 2021. [Online]. Available: https://github.com/rustwasm/wasm-pack

[51] *Rustwasm/wasm-bindgen*. Rust and WebAssembly, 2021. Accessed: May 13, 2021. [Online]. Available: https://github.com/rustwasm/wasm-bindgen

[52] "Cloudflare Workers®." https://workers.cloudflare.com/ (accessed May 10, 2021).

[53] "Areweyet - MozillaWiki." https://wiki.mozilla.org/Areweyet (accessed May 13, 2021).

[54] "Are we web yet? Yes, and it's freaking fast!" https://www.arewewebyet.org/ (accessed May 13, 2021).

[55] "WebAssembly » AWWY?" https://www.arewewebyet.org/topics/webassembly/ (accessed May 10, 2021).

[56] "What is rustdoc? - The rustdoc book." https://doc.rust-lang.org/rustdoc/index.html (accessed May 10, 2021).

[57] "Docs.rs." https://docs.rs/ (accessed May 10, 2021).

[58] "mdBook - mdBook Documentation." https://rust-lang.github.io/mdBook/ (accessed May 10, 2021).

[59] "Preface - Real-Time Interrupt-driven Concurrency." https://rtic.rs/0.5/book/en/ (accessed May 10, 2021).

[60] "Laying the foundation for Rust's future | Rust Blog." https://blog.rust-lang.org/2020/08/18/laying-the-foundation-for-rusts-future.html (accessed May 03, 2021).