Literature Survey of use of non-traditional languages in distributed systems

Thomas de Zeeuw
Vrije Universiteit, Amsterdam
t.m.de.zeeuw@student.vu.nl

February 28th, 2020

Abstract

Distributed systems are more and more common place and are becoming the new norm. Distributed system are inherently complex due to large number of non-deterministic behaviours and number of moving parts, yet more are being build. This literature survey looks at what benefits non-traditional languages (i.e. other than C, C++, Java or Python) can bring to distributed system development.

In this survey I did not only looked at academic papers on the subject, but also looked what languages were used in practice and by who. This included looking at usage of non-traditional languages by large tech companies as well as open source projects.

Keywords — Distributed systems, Distributed system languages, Non-traditional languages, Modern languages.

1 Introduction

More and more systems are becoming distributed nowadays, desktop applications are replaced with web applications and mobile apps that must synchronise with each other just to keep up with the competition. All this requires that, what used to be a single system, is now split over multiple machines and becomes a distributed system.

This survey looks at what advantages non-traditional languages can be bring over widely used languages. Non-traditional language, as used in this paper, means any language other than C, C++, Java and Python (the status quo). To not exclude certain languages, terms like "new", "modern" (would exclude e.g. Erlang), or "non-mainstream" (would exclude e.g. Go) wouldn't be a good fit.

Section 2 discusses how the materials used for this paper were gathered and analysed. Section 3 discusses non-traditional languages in the context of a single machine, followed by the discussion of the non-traditional languages in a distributed setting in section 4. Section 5 looks at the usage of non-traditional languages in practice. Finally section 6 describes directions for future research and section 7 contains the conclusion.

The main research question this survey aims to answer is: *what do non-traditional languages have to offer to distributed system development?* With a sub-question: *what non-traditional languages are used in practice?*

2 Material gathering method

This survey is based on three sources of materials: papers, technical blog posts and a questionnaire send to developers involved in developing distributed systems. The following sections describe the methods used to gather each of the materials.

2.1 Selecting papers

Two methods were used to select papers in this survey: snowballing [1] and searching for additional papers manually.

For the snowballing method the starting papers were provided my supervisor Animesh Trivedi. From the references of the starting papers more papers were selected to consider in this survey. From those papers more pa-

pers were selected from the references, etc. The selected papers were mostly[1] limited to papers from the last five years (between 2014 and 2019) and to the conferences listed in table 1. The papers were limited to these conferences to be ensured of the quality of the papers. For this survey the quality of the papers was not reviewed.

Category	Conference
Computer architecture	ISCA
Data management	SIGMOD
	VLDB
Distributed computing	ICDCS
	PPOPP
	HPDC
	TPDS
High-performance computing	SC
Networking	NSDI
	SIGCOMM
Operating/storage Systems	ASPLOS
	ATC
	FAST
	HotStorage
	OSDI
	SOSP
Programming languages	ICFP
	OOPSLA
	PLDI
	POPL
Other or various categories	EuroSys
	HotCloud
	SIGMETRICS
	SoCC

Table 1: Conferences considered in the paper selection. Note that some conferences could fall in multiple categories.

To not limit the scope of the papers to already known papers, the starting papers in the snowball method and the papers known to those authors, I also searched for additional papers manually. I did this search using a paper search tool called AIP[2]. This tool allows for searching for papers based on keywords and published year, much like other search engines such as Google Scholar, but can also filter based on the conference in which the paper is published. The queries used are shown in table 2. Note that the queries are not specific to the use of non-traditional languages, this is because many papers (published in the distributed systems space) didn't specify the use of a non-traditional language in the title or abstract. As a result the number of papers found using more specific queries (i.e. including keywords focused on the use of non-traditional languages) didn't return the desired number of papers. The selected papers were again limited to papers from the last five years and to the conferences mentioned in table 1, same as for the snowballing method.

Query
distributed system
distributed performance
fault isolation
fault tolerance
actor model
actor system

Table 2: Queries used to search for papers.

2.2 Analysis of selected papers

Using the methods described in section 2.1 resulted in a large amount of papers, not all of which relevant to this survey. For some papers it was trivial to see they didn't fit the survey, others were manually filtered based on title and than abstract. The criteria for the filtering were the relevance to distributed systems and usage of a non-traditional languages. However not a large amount paper fit the cross section of distributed systems and usage of a non-traditional language. Because of this papers that did not concern a distributed system but did make use of a non-traditional language were still considered for this survey.

After filtering each paper was summarised, taking note of usages of non-traditional languages features and user-experiences reports. These notable feature make up the subsections in sections 3 and 4.

2.3 Selecting technical blog posts

Although not peer-reviewed blog posts present a more practical point of view, showing what is used in practice, which brings it own set of challenges that do not always

[1] Some exceptions were made.

[2] Provided by the Distributed Systems Group at the VU, unpublished at the time of writing.

present themselves during tests (in a lab setting). To not search the endless space of the internet for blog posts, the search was limited to posts made by prominent IT companies/departments (in 2019) in the last five years (between 2014 and 2019). Blog posts are only used in section 5, which looks at non-traditional language use in practice.

The blog posts where selected based on technical detail provided and filtered in a similar manner as the papers, as described in section 2.1.

2.4 Questionnaire

A third source of material was a exploratory questionnaire prepared for this survey. The goal of this questionnaire was to get a feeling for what languages where used in practice and what kind of systems were build using them. The questionnaire was available for a month and a half between December 2019 and January of 2020. It was shared with various developers and researchers, working on or doing research into distributed systems. The questions asked can be found in appendix A.

2.5 Language overview

To get the broadest possible scope of non-traditional languages used in distributed systems the search space was not limited to certain languages. However while conducting the survey only a few languages came forward, contrasting to a similar survey conducted in 1989 which showcased 15 languages [2]. Table 3 shows an overview of languages discussed in this paper.

3 Single machine systems

This section presents features or advantages that some non-traditional languages have over traditional languages in a single machine system. These features come from the papers gathered using the methods described in section 2.

3.1 Memory safety

Languages such as C and C++ don't provide any memory safety features, it up to the programmer to ensure allocations and arrays are used correctly, avoiding things like use-after-free [9], double free [10] and buffer overflows [11, 12, 13]. This contributes to the high amount of vulnerabilities in projects such as the Linux kernel [14], in which two-thirds can be attributed to using a unsafe language [15, 16]. Many newer languages provide various kinds of aids to ease the programmers' burden in avoiding the previously mentioned problems. For example a Garbage Collector (GC) alleviates the programmer from having to manually allocate and free memory, or automatic checking of array bounds before accessing memory not part of the array. But for some kind of applications this is seen as an unacceptable overhead. For example video games which aims to render 60 frames per second (FPS) [3] have budget of 16.6 milliseconds for each frame, or 8.3 milliseconds for 120 FPS, if a GC pause takes up more than 10 milliseconds it means that the FPS target will not be met.

However Cutler et al. [17] show that writing a kernel in Go [4], a programming language with a concurrent (or pauseless) GC [18, 19, 20, 21], is possible with manageable overhead. According to there benchmarks the GC uses up to 3% CPU, with the longest pause being 115 microseconds. The longest delay for a single request in a benchmark using NGINX [22] (a HTTP server) was 600 microseconds. So even for the previous example of a video game rendering at 60 or 120 FPS these pause times would be acceptable. Other costs in there kernel account for about 10% CPU. There reference C version is about 5% to 15% faster than the Go implementation.

Lankes et al. [23] also build a kernel, but a special kind called a unikernel [24]. A unikernel doesn't support multiplerunning processes, unlike the traditional kernels, it only support running a single application. This integrates the application and kernel into a single (bootable) binary that avoids the overhead of context switching when making system calls. For building this unikernel they explore the Rust programming language [25], a new language without a GC, but with automatic memory management. Rust also comes with various features commonly found in high-level languages such as memory safety, array bounds checking and first-class functions. They port HermitCore [26] (from C) to Rust, causing a 17% slowdown in the worst case ($\sim$ 1300 cycles), which they consider "acceptable to avoid memory issues."

The two papers discussed show that high-level language features, such as a GC or array bound checking, do bring a runtime overhead, but one that is acceptable for many kinds of high performance systems, something that historically been seen as unacceptable.

[3] That means render the entire screen 60 times every second.

Language	Description	Main benefit	Section(s) discussed
DS2	Domain-specific language and integrated framework for specifying, synthesizing, and reasoning. [3]	Enhances productivity while ensuring correctness of implementation. [3]	4.3.1
Go	Language that makes it easy to build simple, reliable, and efficient software. [4]	Developer productivity.	3.1, 3.3, 5.3
Mace	A C++ language extension that translates a concise but expressive distributed system specification into a C++ implementation. [5]	Handles the implementation details, allowing the programmer to focus on the essential elements, without compromising performance or reliability.	4.3, 4.3.1
MiXT	A domain-specific language for mixed-consistency transactions.	Faster transactions.	4.2
Pony	An actor model and capabilities-secure programming language. [6]	Strong type system and build on the actor model [7, 8].	3.2, 4.3
Rust	General purpose multi-paradigm system programming language.	Strong type system to improve correctness and concurrency safety.	3.1, 3.2, 3.3, 4.1, 4.3, 5.2

Table 3: Overview of languages discussed in this paper.

3.2 Type system

Beyond memory safety some newer languages also provide a more capable type system. Some of these newer type systems are based on *linear types* [27], such as the type systems used in the Rust [25] and Pony [6] programming languages. Linear types underpin Rust's *ownership model* [28] and Pony's *deny properties* [29] (discussed further below and in section 4.3).

Balasubramanian et al. [16] show that Rust uses the idea of linear types for a restricted ownership model to achieve memory safety, ensuring that only a single thread can access memory mutably. In addition to the ownership model Rust has the concept of lifetimes, which tracks how long a stack or heap allocation lives to ensure any references (pointers) to them don't outlive the original allocation, because that would be a use-after-free bug. This allows the compiler, at compile time, to determine when to allocate and free memory without the overhead of a GC. Furthermore it ensures that bugs such as use-after-free or double-free, such as mentioned in section 3.1, are no longer possible. This means that the runtime overhead of the language is limited to array bound checking. But this is often elided by the use of iterators [30, 31]. The authors further argue that "Rust enables faster and more accurate verification", compared to languages like C/C++. One of the problems with languages like C/C++ for verification is aliasing, where more then one pointer can point to the same memory location, in the verification process this increases the state-space making verification less practical. Rust completely eliminates aliasing making verification easier and more efficient.

Boucher et al. also use Rust in [32], they use it to explore a new design for Functions-as-a-service (FaaS), also known as Cloud Functions. Using the FaaS architecture a FaaS provider runs multiple functions, often from different and untrusted tenants, on the same machine. Because the functions come from different tenants they need to be isolated from one another, this is often done using traditional processes (and restricted permissions). One of the reasons why FaaS has become popular is the low cost, as you only pay per function invocation. To keep costs down FaaS providers don't keep all functions in memory as that would cost too much. When a function is not

in memory it needs to be loaded from stable storage and a new process is created for it, this is called a cold-start (when the function is in memory it is called a warm-start). With this the costs are kept low, but the performance for cold-starts is degraded. One key metric for the performance of Faas is *invocation latency*, which is the time between a request coming in and the time the function starts processing the request. Using the cold-start cost optimisation it creates a tension between low latency invocation and low cost. Boucher et al. find in there experiments that using process-based isolation warm-starts take 9μs and cold-starts 2846μs, achieving 300,000 warm-start function invocations per second. But the tail latency is much worse, for 99% the warm-start latency is 27μs and 15976μs for cold-starts. To reduce the invocation latency they propose to run the functions in shared processes with language-based compile-time guarantees for isolation. This language-based isolation achieves 1.2μs warm-start latency (2μs for 99%) and over 5 million invocations per second. Even better is the cold-start latency which is 38μs and 42μs for 99%. Section 4.1 contains more papers using language-based isolation for software fault isolation.

Next we'll look at a paper that proposes changes to the design of (future) language runtime systems, in the following paragraphs we'll see how the type system can play a role in this. In this paper Maas et al. [33] argue that for "the cloud 3.0 Era" runtimes needs to be rethought. They propose seven *tenets* for the design of (future) language runtime systems. The fifth tenet is about managing resource disaggregation: moving memory between pools of memory, e.g. high-bandwidth local memory and larger pools of remote memory. The authors argue that doing this efficiently requires application-level knowledge, e.g. not moving frequently used memory to slower to access remote memory pool. Furthermore having application-level knowledge of the memory layout makes it possible to move data at a finer granularity, e.g. object level granularity compared to page level granularity.

I believe that what Clebsch et al. present in [34] can be used to solve this problem. Clebsh et al. show how the type system of Pony can be used for more efficient garbage collection of objects and actors (from the actor model [7, 8]). Pony's type system uses *deny capabilities* [29] for their ownership model, which the authors use to avoid any form of synchronization between actors for garbage collection. For example if an actor, that is being garbage collected, owns an object (has unique access to it) the garbage collection is simple as simple as freeing it, no scanning of the stack/heap or equivalent operation is required.[4] The same type information could be used for managing different resources, such as the memory pools mentioned in Maas' et al. fifth tenet. For example if an actor is not running it and all objects it owns (known to the type system) can be moved to a different (cheaper) memory pool.

The seventh tenet from Maas et al. argues that GC pauses cause unpredictable performance, which is often worse then a lower throughout. They advocate for using a fully concurrent (or pauseless) GC, such as already seen in Go (section 3.1). Clebsch et al. [35] again make use of the type system in Pony to avoid stop-the-world pauses to make the GC pauseless. At the core of Pony is the actor model in which each actor has its own private heap, actors can't access another actor's heap to due to restrictions in its type system. Using this knowledge the authors are able to garbage collect an actor without synchronising with other actors once the actor is considered dead (i.e. no longer used).

The papers discussed in this section show that a more capable type system can bring more performance, without sacrificing safety. Furthermore a more expressive and restrictive type system has a path to more accurate verification. But newer languages also have better support for concurrent programming, which is discussed in the next section.

3.3 Concurrent programming support

Cutler et al. [17] mention, in there paper in which they build a kernel in Go, that overall high-level languages (HLL) eases concurrent programming: reducing or eliminating certain kinds of bugs. However GC and safety checks are not free as they consume CPU and cause delays (due to GC pauses) as seen in section 3.1. Furthermore language runtimes enforce abstractions and may reduce developers implementation options.

In favour of the Rust programming language Holk et al. [36] argue that Rust's ownership model, specifically owned vs. management memory, make concurrent programming easier. Using the ownership model in Rust it is not possible for more than two objects/threads to have *mutable* access to the same memory, avoiding data races. This ensures at compile time that an entire class of bugs is eliminated.[5] Rust also includes native support for

[4] Scanning is a phase in garbage collection in which is determined whether or not memory is still in use and thus should be freed or not.

[5] Note that this is only true for *safe* Rust, that is not including the *unsafe* superset of Rust (which allows dereferencing raw

concurrent programming in the form of the actor model: supporting channels (a way to send messages) and a task system.

Kulkarni et al. [37] agrees that Rust's ownership model prevents data races. In additional they say that it also prevents segmentation faults, as dereferencing raw pointers is not allowed in safe Rust. Furthermore Rust's concept of lifetimes, previously discussed in section 3.2, ensures that references to objects don't outlive the objects themselves.

This section shows that a more capable type system is beneficial in writing correct concurrent code, in addition to making it possible to achieve better performance as seen in the previous section. Although the features discussed so far are still limited to single machine systems it already shows various features and benefits that non-traditional languages have over traditional languages. Various restrictions enforced by the type system, such as Rust's ownership model or Pony's deny capabilities, allow for more *trust* in the code as the compiler will enforce certain bug are simply not present. This removes a great deal of worrying from the programmer both in writing and reviewing the code. In the next section we'll see what non-traditional languages have to offer distributed system development.

4 Distributed systems

More distributed systems are being build today [38], mobile applications and rich web applications often communicate with a back-end server, which in itself is often a distributed system consisting of application servers, storage servers, etc. However distributed computing has not moved far from its own "assembly": message passing [39].

Furthermore building (correct) distributed systems is hard, Leesatapornwongsa et al. [40] build a taxonomy of distributed concurrency bugs in four widely-deployed data center distributed systems. It contains 104 bugs not found using conventional testing, but discovered in production. It is safe to say that building distributed systems is hard [41].

4.1 Fault isolation

Joe Armstrong wrote in his thesis: "the essential problem that must be solved in making a fault-tolerant software system is [...] that of fault-isolation" [42]. Software Fault Isolation (SFI) enforces process-like boundaries around program modules, without relying on hardware protection. There are various implementations, traditional SFI architectures include isolating components to a private heap [43, 44, 45] or tagging memory [46]. But (almost) all implementation incur some runtime overhead, copying data or validating tags on pointer dereference.

When building a key-value store named Splinter Kulkarni et al. [37] needed fault isolation. With Splinter they aim to reduce the amount of data moved between storage and compute, pushing the compute to the store itself. To that end Splinter supports extensions to allow user-specified code to be executed on the store itself. Because the code comes from untrusted tenants they needed to isolate these extensions. But they found that at microsecond timescale hardware based isolation (context switches) is too expansive. Instead they used Rust's type system to isolate the extensions. There extensions are low-cost: about 20% overhead (compared to the estimated 80% for hardware isolation), and are still capable of using zero-copy I/O.

Panda et al. [47] also use Rust's type system for isolation in Network function virtualization (NFV), a concept that aims to replace hardware middle-boxes in the network with software functions. One of the main requirements here is performance: per-packet latency must be on the order of 10s of μs, with throughput on the order of 10s of Gbps. There solution, Netbricks, provides a programming model and execution environment, which relies on Rust for memory and fault isolation. This allows them to achieve Zero-Copy Software Isolation (ZCSI) and meet the performance requirements.

While exploring what Rust had to offer beyond safety features, Balasubramanian et al. [16] also looked at how Rust's type system could be used for SFI. They argue that Rust allows SFI with negligible (under 1%) overhead, without copying or tagging. They experiment using NetBricks [47] to show that recovery of failed domains, including clean up and creating a new domain, took 4389 cycles on average.

Levy et al. [48] present a fourth paper that uses Rust's type system for isolation, but they use it in the context of building an embedded Operating System (OS). Embedded programming requires the OS to be bundled with the application. This means that all running code must be trusted as any code can take down the entire OS and all applications. Furthermore it is not possible to only shutdown a part of the system that is failing, it is all or nothing. They present the Tock Operating System, which aims to change that. It uses Rust language's type-

pointers, something not allowed in safe Rust).

and module systems for low cost fault isolation, creating a process like abstraction.

This section shows that Rust type system is capable of providing low-cost Software Fault Isolation without sacrificing things like zero-copy I/O. We've seen it being used in three domains: storage, networking and embedded OS, in all of which performance is an important factor. The next section will focus on the performance aspect.

4.2 Scalability and performance

Scalability is an essential attribute of a distributed system as poor scalability can result in poor performance [49]. Some distributed systems have lowered there consistency model to eventual consistency [50] to achieve better scalability, such as Cassandra [51] and MongoDB [52]. The problem with eventual consistency is that is up to application to deal with retrieved values that might not be up-to-date. Furthermore not all data can be stored using eventual consistency, some data simply requires strong consistency, e.g. bank account balance.[6] Milano et al. [53] make a key observation that consistency is not a property of a transaction, but a property of the data. They propose mixing weak and strong consistency in the same transaction based on what data is used, i.e. use strong consistency only for data that requires it. For this purpose they created MiXT, an embedded Domain Specific Language (DSL), for mixing consistency in transactions. There experiments show that it can achieve an 2x throughput improvement if 50% of the data is causally consistent, up to 3x if 80% of the data is causal consistent.

However a fast system is useless if it doesn't return the correct response, hence the next section is about correctness of distributed systems.

4.3 Correctness

Design and implementing of a high performance and *correct* distributed system is challenging. Killian et al. [5] say there are three ways of specifying a system. First, formalisms such as using I/O Automata [54] or the Pi-Calculus [55]. However this ignores low-level details that are important for robust and high performance systems. Second, high-level language which ease programming (automatic memory management, etc.), however these come with a performance overhead.[7] Thus developers often take the third route, ad-hoc application implementation in a low-level language (C, C++, etc.). This sacrifices structure, readability and extensibility for performance.

Killian et al. instead present Mace, a C++ language extension, which enables practical model checking and allows for components to more easily be reused. Mace aims to reduce the complexity by forcing a certain structure on the program: mainly a state machine with fixed transitions between states. The code generated by Mace enforces that only valid state transitions are made and handles various implementation details for the developer. This leaves the developer to create the business logic in the form of upcalls. [8]

In the third option from Killian et al., ad-hoc application implementation, one common problem in concurrent (and thus also distributed) systems is data races. Traditional language provide little to no utilities to guarantee data-race freedom. But various non-traditional language provide stronger type systems that can guarantee data-race freedom, among other things. In section 3 we've already Rust and Pony as example of such type systems. Rust's type system guarantees the absence of data races, buffer overflows, stack overflows, and accesses to uninitialized or deallocated memory [56]. Furthermore its type system is (being) verified [28, 57]. Clebsch et al. show something similar for the Pony programming language called *deny properties* [29]. Like Rust's type system it is based on linear types (as discussed previously in section 3.2) and it eliminates data races completely, ensuring data-race freedom statically.

4.3.1 Verification

The next step in correctness is verification, but as Killian et al. [5] already argued current generation of verification languages ignore essential low-level details that prevents a distributed system to be written in them (and run from the same code base). This means that the system is often verified in one language, e.g. TLA+ [58, 59, 60], mCRL2 [61, 62] or Coq [63], and implemented in another programming language.

It is important that the gap between verification- and programming languages is reduced or closed completely, allowing a single language to be used for verification and

[6] Users shouldn't be able to spend money they don't have, but when operating on an outdated account balance this could be possible.

[7] We have seen in section 3.1 however that this is overhead is significantly lowered in recent years and now acceptable for many kind of high performance applications.

[8] Similar to, but not quite the same as, remote procedure calls (RPC), implemented as functions in Mace.

as a programming language (that is usable).[9] We've already seen on such language: Mace [5]. Mace is a Domain Specific Language (DSL), which supports practical model checking, but is also a practical programming language. Mace itself compiles to C++, so that most existing tools surrounding C++ can be used for Mace. This means that Mace can be used as a programming language, but can also be more easily verified.

Another language that tries to fill this gap is DS2 [3]. In there paper about DS2 Al-Mahfoudh et al. argue that the tools to reason about the correctness of distributed system are lacking. To solve this they present DS2 a DSL for specifying, verifying, and constructing distributed systems. Like Mace it forces a certain structure to the application that helps with verification and DS2 itself handles various low level details.

So far neither Mace or DS2 seem to have taking the industry by storm however. This means that either verification must be improved to be able to verify *programming* languages (such as C/C++) or verification languages must become easier to program in to become "mainstream". However the industry seems to have little interest in the latter option as Mace is over a decade old at this point, but still seems to receive little use. Rust could help in accomplishing the first option as it will be easier to verify as we've already seen in section 3.2.

Although practical verification for general purpose programming languages is not quite here yet, we've seen two DSLs that do support practical verification for distributed systems. Furthermore we've seen that the type systems that Rust and Pony offer eliminate entire classes of hard to catch bugs, such as data races, at compile time. This makes it easier to write correct systems compared to traditional language such as C, C++, knowing that various classes of bugs are simply no longer possible.

4.4 Availability

Beyond being fast and correct a system must also be available to its users to be useful. A distributed system can be available in different ways/modes: fully available, all responses returned are correct and up-to-date[10], or in some kind of degraded mode in which responses might be outdated because the node processing the request is disconnected from its peers or backing store and is returning responses from its (possibly outdated) cache. Put simply availability are the questions: *does the system respond within a acceptable time period?* And *is that response a correct and non-error response?*

An important part in programming a distributed system that is highly available is dealing with error conditions. In Mace [5] (already discussed in section 4.3) dealing with error conditions is taken care of. Killian et al. write that Mace takes take of "tedious" things such as failure detection and handling. Furthermore they argue that overall the structure imposed by Mace greatly simplifies the implementation by allowing the programmer to focus only on the essential elements, without compromising performance or reliability. While experimenting the authors recreated some existing application in Mace. In doing so they found a hand-full of concurrency bugs in the original applications. These bugs don't exists in the Mace port as the generated code (by Mace) takes of those edge cases for the programmer.

4.5 Reusability

A common programming pattern is reusability, being able to reuse code saves time and resources re-implementing (almost) the same code. In single machine programming this is a common practice, almost all language provide some kind of library support that allows common/non-application-specific code to be reused in multiple applications. Some languages take it a step further and support generics, allowing part of the code to be used by any data type, commonly used when building containers (such as vectors, binary trees etc.). However when it comes to distributed system development this is much less common.

Prokopec et al. [39] argue that distributed programming is missing a unified standard library of reusable high-level components. The authors point to the lack of composability of the primitives exposed in current low-level models used in distributed programming as the root cause of this. A primitive is composable if it can be implemented (and used) on its own but also combined into a more complex primitive. The authors mostly focus on message protocols. Most distributed system communicate by exchanging messages, such as done in the actor model [7, 64] (a popular and much used model for distributed system), the specific pattern in which the messages are send is called a message protocol. There are different message protocols [65, 54], but few of them are composable the authors argue. To solve this Prokopec et al. propose a new model, or an extension to the actor model, they call *reactive isolates*. The model uses three basic abstractions *isolates*, *channels* and *event streams*.

[9] A language with good (enough) performance and developer productivity, etc.

[10] What this means differs per system.

The main difference compared to the actor model is the introduction of types, e.g. channels (a handle to send an isolate (actor) a message) are typed. For full details of the model see the paper [39]. Although the proposed model is minimalistic they argue its sufficient to build stronger abstractions, i.e. it is composable.

To summarise this section: developing distributed system must move forward along with single-machine development, moving away from its low level abstractions such as message passing and into reusable message protocols (just to name a direction). Furthermore I believe that the advances made in programming languages since C, C++ and Java do warrant a serious consideration for them when building distributed systems. We've seen that non-traditional language can help in important aspects of distributed system development such as low-cost fault isolation, performance, availability and in writing correct code, even opening a path to practical verification. In the next section we'll look at how non-traditional languages are used in practice.

5 Use in practice

For the look at the use of non-traditional languages in practice, I looked at two languages: Rust [25] and Go [4]. I choose these two languages because (on the surface) these two languages seem some of the most used when it comes to distributed systems (among the newer non-traditional languages)[11]. The languages are also popular and beloved, according to Stack Overflow [66], and are the two must used language in the questionnaire.

5.1 Questionnaire results

As mentioned in section 2.4 I've created a questionnaire for this survey. This was done in an effort to answer the research sub-question: *what non-traditional languages are used in practice?* Because of this the questionnaire is meant to be an exploratory questionnaire, not to prove a hypothesis. Furthermore due to the low number responses (37) we can't generalise the questionnaire results to the distributed system development community at large, however it does give us some insight into the usage of non-traditional languages in practice. Below I present a short summary of the questionnaire results. All responses can be found in appendix B.

[11] This is based on the personal experience of the author and on the amount of material available publicly regarding these languages.

Q1 *What language(s) do you use outside of "traditional" languages (such as C/C++/Java/Python)?* Introductory question to see what languages where being used and providing context to the answers to the other questions. The most popular languages were Go (37.8%) and Rust (27%), followed by Erlang (21.6%) and Scala (18.9%). This was used in deciding to looking into Rust and Go for blog posts.

Q2 *What kind of system have you worked on (using the language(s) in 1)?* Much like the first question this question provided some context to the survey answers. Distributed data processing leads with 64.9%, followed by distributed storage (40.5%) and distributed queueing/messaging and local storage (both 35.1%). Overall the responders worked on 18 different kind of systems.[12] This gives a fairly broad view of the different systems being worked on (for the number of responses).

Q3 *Why was this/these language(s) chosen?* Choosing a non-traditional language brings an extra risk compared to choosing a traditional language, with this question I was looking for the deciding factor(s) that go into choosing a language. The most chosen answer was productivity (73%), followed closely by language features (70.3%, one less vote). Productivity is not something mentioned much in the papers read for this survey, however in practice this does seem important and also came up in the blog posts about Go (section 5.3.1). Language features have come up throughout this paper and its good to see that both practice and academia agree that its an important reason to chose a non-traditional language. The third most common answer was personal interest (54.1%), I see this as something positive as it means that developers *want* to program in a non-traditional language (something not always true for traditional languages). Language ecosystem (45.9%) and general tooling around the language (40.5%) are also important. Interesting to see these being important factors because for most traditional languages the ecosystem and tooling is/are larger and more established compared to non-traditional languages. It seems that the language ecosystem and tools is large and mature enough for practical use for some languages (this also came up as a weak point of some languages in question five).

[12] If no distinction would be made between distributed and local systems this would be 11 different kind of systems.

Q4 *What are the strong points of the language(s)?* Various features and benefits have been mentioned throughout this paper already, this question aims to see if those strong points are also an attractive/important feature in practice and to see if any strong point was missed. Due to the variation in the answer provided it is hard to provide a single conclusion. However what became clear is that each language has its own niche benefits. For example Erlang's and Exlixir's (based on Erlang) isolate processes, Rust's strong static analyser, Go is naturally concurrent and Haskell's type system eliminating runtime errors.

Q5 *What are the weak points of the language(s)?* Most papers and blog posts discussed in this paper emphasise the strong points of a language, but no language is perfect. To provide some counter-weight to all the positives this question looks at the weak points of the language/ecosystem that might not be clear when choosing the language, or when using it for a small(-ish) experiment for a single paper. Two things mentioned in the responses I found interesting: small community and immaturity. The small community is a clear danger when choosing for a non-traditional language. This is true for both an employer, as the pool of (suitable) programmers is smaller compared to traditional languages, and for employees, as the number of employers using a non-traditional language is also smaller. I believe immaturity is also a consequence of the small community, and is reflected in (among other things) in the lack of good tooling.

Q6 *How important is existing tooling when choosing a language?* Most traditional languages come with whole host of tools, open source and commercially available ones. Since most non-traditional languages either have a small community/user-base or aren't around for too long, they often lack in the tools available. This question looks at whether or not existing tooling is an important factor in choosing a language. Question three already showed that tooling is an reason why a non-traditional language was chosen and this question re-enforces that. 32.4% of the responses say existing tooling is very important, another 40.5% say its important for a total of 72.9% saying its important. Somewhat surprising (to me) 24.3% answered neutral (not important and not unimportant) to the question. This to me says that developers are willing to work with language with lesser tools if the language itself provides something more valuable, but that improving tooling is important to them.

Q7 *How large is the team working on the system?* Larger teams have different requirements from there language, tools and ecosystem then smaller teams. For example for a small team working on a small system they might be able to completely understand the entire system, knowing all the details of it. However as the team and system grows it more likely that people only work on a single component of the system, rather than all of it. This means the interface between the components are more important, something a good type- and module system can help with. Furthermore a larger team also requires a larger investment, bringing more risk with it. With this question I wanted to know if non-traditional languages are also used by bigger teams, or only by smaller (experimental) teams. The majority of the responders work in teams smaller then five (34.3%), another 20% in teams smaller than ten. On the other end 17.1% of the responses say there working in a team with more than 50 people. This shows that non-traditional languages are also used by larger teams and larger companies, which is good for the continuation of these languages.

Q8 *On how many machines does the system run?* Much like with question 7, I wanted to see at what scale of machine the responders worked. 11.8% of the responders work at a scale of $\mathcal{O}(10,000)$ machines, another 29.4% at $\mathcal{O}(1,000)$ machines, showing that non-traditional languages are used for very large scale system deployments. However the majority of the responders work a slightly more modest scale: 26.5% at $\mathcal{O}(100)$ and 29.4% at $\mathcal{O}(10)$ machines.

Q9 *Additional information.* This allowed the responders to share anything else they wanted to shared. However no interesting (for this survey) answers were provided.

5.2 Rust

The Rust programming language has been mentioned multiple times already throughout this paper. But not only academia has picked up this language, it's used in practice as well. Rust originated at Mozilla Research [67] where it was made to create Servo [68], a parallel browser engine (thus it is created for developing concurrent programs). Since its creation Servo have been slowly inte-

grated into Mozilla's web browser Firefox [69, 70, 71, 72], known as project Quantum [73].

Outside of Mozilla Rust has gained a large number of users that run the language in production [74]. Dropbox [75] build "Magic Pocket" [76], there custom infrastructure to store Dropbox's data, originally in Go, but moved (parts of it) to Rust "to handle more disks, and larger disks, without increased CPU and Memory costs by being able to directly control memory allocation and garbage collection." [77, 78, 79].

Facebook [80] also build various software project in Rust, such as Mononoke [81]: a Mercurial source control server replacement, parts of HHVM [82]: virtual machine that executes Hack/PHP code, and there cypto currency Libra [83], but have not made public blog posts about why they chose Rust for these projects.

As shown throughout this paper Rust is also used in academia, one more example I wanted to mention is Noria [84, 85], a dynamic, partially-stateful data-flow database.

5.2.1 Tools

The Rust distribution comes with various, easily obtained and open source tools. Such as Cargo [86]; the package (library) manager for the language, along with crates.io [87]; Cargo's main package registry of open source packages, Clippy [88]; a linter [89, 90] for Rust code, and rustfmt [91]; a tool to format the code automatically.

npm [92], the largest software registry in world with upwards of 1.3 billion package downloads per day, chose Rust for there authorisation service [93]. One of the reasons they named was that "Rust has absolutely stunning dependency management" [94].

PingCAP [95], the company behind TiDB [96] (an open source distributed SQL database), chose Rust for its ecosystem (among other reasons) to build TiKV [97] (an open source distributed transactional key-value database) [98]. Emphasising its "excellent package management tool, Cargo" and its many libraries available.

These two cases show that package management is an important feature of a language used in practice.

5.2.2 Safety guarantees

We've already discussed the safety guarantees Rust brings in sections 3.1 and 3.2 and I'll not repeat them here. AWS [99] chose Rust for its safety guarantees, w.r.t. avoiding security vulnerabilities such as overflowing buffers, when building Firecracker [100, 101] the virtualisation technology powering AWS Lambda [102] and AWS Fargate [103].

Microsoft [104] also chose Rust for its safety guarantees to build the Security Daemon for there Azure IoT Edge platform [105, 106].

Cloudflare [107], a CDN/infrastructure provider, used Rust to build a Wireshark-filter parser [108] used in there DDoS protection Specturm [109, 110], and in BoringTun [111, 112], an implementation of the WireGuard [113] protocol.

For these three use cases the clear benefit of Rust's safety guarantees over traditional languages like C or C++ were an important factor. This section shows that various companies, in various industries, are willing to advance there technology stack over the known and stable (C, C++) because of the benefits Rust provides.

5.3 Go

The Go programming language [4], also previously mentioned in this paper, was originally developed by Google [114] and is sponsored by them. It has a large list of users [115], most which chose Go for its performance, such as 500px [116], NTP Pool Project [117] (used as "time server" for a lot of OSes and applications), Slack [118], Stack Overflow [119, 120] and Malwarebytes [121] (anti-malware software).

Another reason is Go's inherent concurrency friendly design, following Hoare's communicating sequential processes (CSP) [122]. From the CSP model comes one of Go's mantras: "do not communicate by sharing memory; instead, share memory by communicating" [123]. This mantra advocates to use Go's built-in channel to communicate across threads rather then sharing memory (protected by lock) used in traditional languages. Its concurrency friendly design is one of the reasons why CloudFlare [124, 125] and Riot Games [126] (maker of the popular League of Legends game) choose Go.

5.3.1 Developer productivity

Go has a small language surface compared to languages like C/C++. The language specification of Go is roughly 75 pages [127],[13] compared to 1605 pages found in the C++17 specification [128]. Having a smaller language makes it easier for new developers to pick it up and makes them quickly productive in Go. Uber says that it takes just a few days for C++, Java or Node.js developer to learn Go [129]. Netflix also says that developers

[13] There is no official document with page numbers, only the website. The 75 page number comes from copying all text into a document and counting the pages of it.

are more productive in Go [130]. Disqus went from initial commit to shipping a back-end in roughly a week's time [131, 132]. For fast moving industries having these kind of quick turn-around of adding new developers and creating entire products can be of great benefit.

5.3.2 Open source projects

Beyond companies that use Go there are also a number of popular, large open source systems written in Go. Here I'll mention three, but there are many more.

Kubernetes [133, 134] is an automated deployment, scaling and management system for containerised applications. It manages anything from 1 to 10,000 nodes and scales accordingly.

Moby [135, 136] (previously Docker [137]), is a runtime for containers. Among other places its for example used by Kubernetes to run applications.

Third, Vitess [138, 139] a database clustering system for horizontal scaling of MySQL, created to scale YouTube's MySQL servers [140].

These projects show that Go is far beyond the stage of concept or experimental language and is used in distributed systems today.

6 Future research

A promising direction of future research is practical model checking using *programming* languages with an advanced and expressive type system. Something Balasubramanian et al. already mentioned in [16] for Rust and progress is being made [141, 28, 57]. The first step would be practical model checking for a single machine system, following by a distributed system.

A second direction is more practical: implementations of reusable distributed components. Throughout this paper various distributed system have been discussed, but none of them make extensive use of standardised components for there implementation, each system has its own implementation of their required components. Most new languages have a better dependency management story and because of it it's easier to use a dependency compared to language like C and C++ which lack a standard dependency manager.

Third, distributed development need to advance its model(s). They way I see it is that distributed system development is currently stuck in the "*loop* phase". We've moved on from assembly jump statements to a *loop* construct: from sending bytes over the network to sending messages using standardised protocols. But we've yet to make the next step: moving from *loop*s to *iterators*, removing off-by-one errors [142] and making the construct far more composable. For distributed system this means making the primitives easier to use and more composable. This should be the next phase in distributed systems development and I think newer, non-traditional language can be of great benefit here.

7 Conclusion & discussion

This papers started with two questions: *what do non-traditional languages have to offer to distributed system development?* and *what non-traditional languages are used in practice?*

To answer the first question sections 3 and 4 show a number of useful language features: memory safety, advanced and expressive type systems, concurrent programming support, low-cost software fault isolation and a path to practical verification, that can help in developing a distributed system. An overview of the main benefits discussed in this paper can be found in table 4.

Section 5 answers the second question: both Rust and Go are used in production settings in some of the largest deployments of distributed systems in the world. Both have proven there worth and can be considered valid alternatives to the likes of C, C++ and Java.

This paper shows that advances in programming languages have been made in the last 55 and 35 years since C and Java were released, and that those advances can greatly benefit distributed development. However there is still room for further improvement, specifically distributed system development needs to move away from its assembly (message passing) into something more high-level, more composable and easier to use.

8 Acknowledgments

Thank you to Animesh Trivedi for supervising this survey, as well to Alexandru Iosup as secondary supervisor. Also thank you to Laurens Versluis for providing the AIP search tool. Thank you to the entire Distributed System group for feedback on and sharing of the questionnaire, and to everybody who took the questionnaire.

Benefit	Example	Section(s) discussed
Performance, without sacrificing safety.	Mace: provides a restricted programming model, ensuring safety, but it compiles to C++ with all performance benefits from it.	4.3.1.
	MiXT: safely mixes weak and strong consistency within the same transaction.	4.2.
	Pony: *deny capabilities* allow for pauseless GC avoiding synchronisation between actors.	3.2.
	Rust: ownership model allow sharing of immutable memory, ensuring it can't be mutated (as the type system won't allow it).	3.2.
Correctness of concurrent code.	DS2: language only allows for correct, verifiable code.	4.3.1.
	Go: provides high-level language features avoiding low-level complexities.	3.3.
	Mace: restrictive programming model avoids low-level problems like data races.	4.3.1.
	Pony: type system ensures absence of data races.	3.2.
	Rust: type system ensures absence of data races.	3.2.
Low-cost software fault isolation (SFI).	Pony: each actor (from the actor model) has its own private heap.	Not this discussed in this paper, but its builtin to the language and runtime.
	Rust: type system ensures single access to mutable data, isolating objects from one another.	4.1.

Table 4: Overview of benefits mentioned in this paper.

References

[1] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In Martin J. Shepperd, Tracy Hall, and Ingunn Myrtveit, editors, *18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14, London, England, United Kingdom, May 13-14, 2014*, pages 38:1–38:10. ACM, 2014.

[2] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Comput. Surv.*, 21(3):261–322, 1989.

[3] Mohammed Al-Mahfoudh, Ganesh Gopalakrishnan, and Ryan Stutsman. Toward rigorous design of domain-specific distributed systems. In *Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering, FormaliSE@ICSE 2016, Austin, Texas, USA, May 15, 2016*, pages 42–48. ACM, 2016.

[4] The go programming language. `https://golang.org`.

[5] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. Mace: language support for building distributed systems. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 179–188. ACM, 2007.

[6] The pony programming language. `https://www.ponylang.io`.

[7] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In Nils J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA,*

USA, August 20-23, 1973, pages 235–245. William Kaufmann, 1973.

[8] Gul Agha and Carl Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In S. N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science, Fifth Conference, New Delhi, India, December 16-18, 1985, Proceedings*, volume 206 of *Lecture Notes in Computer Science*, pages 19–41. Springer, 1985.

[9] Cwe-416: Use after free. `https://cwe.mitre.org/data/definitions/416.html`.

[10] Cwe-415: Double free. `https://cwe.mitre.org/data/definitions/415.html`.

[11] Cwe-120: Buffer copy without checking size of input ('classic buffer overflow'). `https://cwe.mitre.org/data/definitions/120.html`.

[12] Cwe-121: Stack-based buffer overflow. `https://cwe.mitre.org/data/definitions/121.html`.

[13] Cwe-122: Heap-based buffer overflow. `https://cwe.mitre.org/data/definitions/122.html`.

[14] Linux kernel cve security vulnerabilities, versions and detailed reports. `hhttps://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33`.

[15] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In Haibo Chen, Zheng Zhang, Sue Moon, and Yuanyuan Zhou, editors, *APSys '11 Asia Pacific Workshop on Systems, Shanghai, China, July 11-12, 2011*, page 5. ACM, 2011.

[16] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamaric, and Leonid Ryzhyk. System programming in rust: Beyond safety. In Alexandra Fedorova, Andrew Warfield, Ivan Beschastnikh, and Rachit Agarwal, editors, *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017*, pages 156–161. ACM, 2017.

[17] Cody Cutler, M. Frans Kaashoek, and Robert Tappan Morris. The benefits and costs of writing a POSIX kernel in a high-level language. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 89–105. USENIX Association, 2018.

[18] Richard Hudson. Go gc: Prioritizing low latency and simplicity. 2015. `https://blog.golang.org/go15gc`.

[19] Henry G. Baker Jr. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, 1978.

[20] Balaji Iyengar, Gil Tene, Michael Wolf, and Edward F. Gehringer. The collie: a wait-free compacting collector. In Martin T. Vechev and Kathryn S. McKinley, editors, *International Symposium on Memory Management, ISMM '12, Beijing, China, June 15-16, 2012*, pages 85–96. ACM, 2012.

[21] Bill McCloskey, David F Bacon, Perry Cheng, and David Grove. Staccato: A parallel and concurrent real-time compacting garbage collector for multiprocessors. *Report RC24504, IBM*, 2008.

[22] nginx, an http and reverse proxy server,. `https://nginx.org`.

[23] Stefan Lankes, Jens Breitbart, and Simon Pickartz. Exploring rust for unikernel development. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 8–15. ACM, 2019.

[24] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David J. Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: library operating systems for the cloud. In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 461–472. ACM, 2013.

[25] Rust programming language. `https://www.rust-lang.org/g`.

[26] Stefan Lankes, Simon Pickartz, and Jens Breitbart. Hermitcore: A unikernel for extreme scale computing. In Kamil Iskra and Torsten Hoefler, editors, *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers, Kyoto, Japan, June 1, 2016*, pages 4:1–4:8. ACM, 2016.

[27] Philip Wadler. Linear types can change the world! In Manfred Broy, editor, *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, page 561. North-Holland, 1990.

[28] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *PACMPL*, 2(POPL):66:1–66:34, 2018.

[29] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos Varela, editors, *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015*, pages 1–12. ACM, 2015.

[30] Processing a series of items with iterators. `https://doc.rust-lang.org/book/ch13-02-iterators.html`.

[31] std::iter::iterator trait. `https://doc.rust-lang.org/std/iter/trait.Iterator.html`.

[32] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 645–650. USENIX Association, 2018.

[33] Martin Maas, Krste Asanovic, and John Kubiatowicz. Return of the runtimes: Rethinking the language runtime system for the cloud 3.0 era. In Alexandra Fedorova, Andrew Warfield, Ivan Beschastnikh, and Rachit Agarwal, editors, *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017*, pages 138–143. ACM, 2017.

[34] Sylvan Clebsch, Sebastian Blessing, Juliana Franco, and Sophia Drossopoulou. Ownership and reference counting based garbage collection in the actor world, 2015.

[35] Sylvan Clebsch and Sophia Drossopoulou. Fully concurrent garbage collection of actors on many-core machines. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 553–570. ACM, 2013.

[36] Eric Holk, Milinda Pathirage, Arun Chauhan, Andrew Lumsdaine, and Nicholas D. Matsakis. GPU programming in rust: Implementing high-level abstractions in a systems-level language. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, Cambridge, MA, USA, May 20-24, 2013*, pages 315–324. IEEE, 2013.

[37] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 627–643. USENIX Association, 2018.

[38] Christopher S. Meiklejohn. On the design of distributed programming models. *CoRR*, abs/1701.07615, 2017.

[39] Aleksandar Prokopec and Martin Odersky. Isolates, channels, and event streams for composable distributed programming. In Gail C. Murphy and Guy L. Steele Jr., editors, *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 171–182. ACM, 2015.

[40] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In Tom Conte and Yuanyuan Zhou, editors, *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pages 517–530. ACM, 2016.

[41] Denise Yu. Why are distributed systems so hard? Portland, OR, October 2019. USENIX Association.

[42] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2003.

[43] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: software guards for system address spaces. In Brian N. Bershad and Jeffrey C. Mogul, editors, *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 75–88. USENIX Association, 2006.

[44] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The JX operating system. In Carla Schlatter Ellis, editor, *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*, pages 45–58. USENIX, 2002.

[45] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 79–93. IEEE Computer Society, 2009.

[46] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software fault isolation with API integrity and multi-principal modules. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 115–128. ACM, 2011.

[47] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. Netbricks: Taking the V out of NFV. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 203–216. USENIX Association, 2016.

[48] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 234–251. ACM, 2017.

[49] Andre B. Bondi. Characteristics of scalability and their impact on performance. In *Second International Workshop on Software and Performance, WOSP 2000, Ottawa, Canada, September 17-20, 2000*, pages 195–203. ACM, 2000.

[50] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[51] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010.

[52] Peter Membrey, Eelco Plugge, and DUPTim Hawkins. *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing*. Apress, 2011.

[53] Matthew Milano and Andrew C. Myers. Mixt: a language for mixing consistency in geodistributed transactions. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 226–241. ACM, 2018.

[54] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[55] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[56] Nicholas D. Matsakis and Felix S. Klock II. The rust language. In Michael Feldman and S. Tucker Taft, editors, *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, pages 103–104. ACM, 2014.

[57] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: an aliasing model for rust. *PACMPL*, 4(POPL):41:1–41:32, 2020.

[58] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[59] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla$^+$ specifications. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 1999.

[60] Tla+. `https://lamport.azurewebsites.net/tla/tla.html`.

[61] The formal specification language mcrl2. In Ed Brinksma, David Harel, Angelika Mader,

Perdita Stevens, and Roel J. Wieringa, editors, *Methods for Modelling Software Systems (MMOSS), 27.08. - 01.09.2006*, volume 06351 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[62] mcrl2 formal specification language. https://www.mcrl2.org/web/user_manual/index.html.

[63] The coq proof assistant. https://coq.inria.fr.

[64] Gul A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press, 1990.

[65] Rachid Guerraoui and Luís E. T. Rodrigues. *Introduction to reliable distributed programming*. Springer, 2006.

[66] Developer survey results 2019. https://insights.stackoverflow.com/survey/2019.

[67] Mozilla research. https://research.mozilla.org.

[68] Servo, the parallel browser engine project. https://servo.org.

[69] Firefox. https://www.mozilla.org/firefox.

[70] Dave Herman. Shipping rust in firefox. 2019. https://hacks.mozilla.org/2016/07/shipping-rust-in-firefox.

[71] Put your trust in rust – shipping now in firefox. 2017. https://blog.mozilla.org/firefox/put-trust-rust-shipping-now-firefox.

[72] David Bryant. A quantum leap for the web. 2016. https://medium.com/mozilla-tech/a-quantum-leap-for-the-web-a3b7174b3c12.

[73] Quantum. 2017. https://wiki.mozilla.org/Quantum.

[74] Rust production users. https://www.rust-lang.org/production/users.

[75] Dropbox. https://www.dropbox.com.

[76] Akhil Gupta. Scaling to exabytes and beyond. 2016. https://blogs.dropbox.com/tech/2016/03/magic-pocket-infrastructure.

[77] Magic Pocket and Hardware Engineering Teams. Extending magic pocket innovation with the first petabyte scale smr drive deployment. 2018. https://blogs.dropbox.com/tech/2018/06/extending-magic-pocket-innovation-with-the-first-petabyte-scale-smr-drive-deployment.

[78] Jamie Turner. Go-ing to rust: Optimizing storage at dropbox. 2017. https://qconsf.com/sf2016/sf2016/presentation/going-rust-optimizing-storage-dropbox.html.

[79] Cade Metz. The epic story of dropbox's exodus from the amazon cloud empire. 2016. https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire.

[80] Facebook. https://facebook.com.

[81] Mononoke. https://github.com/facebookexperimental/mononoke.

[82] Hhvm. https://hhvm.com.

[83] Libra. https://github.com/libra/libra.

[84] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Tappan Morris. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 213–231. USENIX Association, 2018.

[85] Noria: dynamic, partially-stateful data-flow database. https://github.com/mit-pdos/noria.

[86] Cargo - the rust package manager. https://github.com/rust-lang/cargo.

[87] crates.io - the rust package registry. https://crates.io.

[88] Clippy - a bunch of lints to catch common mistakes and improve your rust code. https://github.com/rust-lang/rust-clippy.

[89] S. C. Johnson. Lint, a c program checker. In *COMP. SCI. TECH. REP*, pages 78–1273, 1978.

[90] lint (software). https://en.wikipedia.org/wiki/Lint_(software).

[91] rustfmt - format rust code. https://github.com/rust-lang/rustfmt.

[92] npm node.js package manager. https://www.npmjs.com.

[93] Sergio De Simone. Npm adopted rust to remove performance bottlenecks. 2019. https://www.infoq.com/news/2019/03/rust-npm-performance.

[94] Community makes rust an easy choice for npm. 2019. https://www.rust-lang.org/static/pdfs/Rust-npm-Whitepaper.pdf.

[95] Pingcap. https://pingcap.com/en.

[96] Tidb repository. https://github.com/pingcap/tidb.

[97] Tikv repository. https://github.com/tikv/tikv.

[98] Why did we choose rust over golang or c/c++ to develop tikv? 2017. https://pingcap.com/blog/2017-09-26-whyrust.

[99] Aws. https://aws.amazon.com.

[100] Jeff Barr. Firecracker – lightweight virtualization for serverless computing. 2018. https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/.

[101] Firecracker. https://github.com/firecracker-microvm/firecracker.

[102] Aws lambda. https://aws.amazon.com/lambda.

[103] Aws fargate. https://aws.amazon.com/fargate.

[104] Microsoft. https://www.microsoft.com.

[105] Ryan Levick. Why rust for safe systems programming. 2019. https://msrc-blog.microsoft.com/2019/07/22/why-rust-for-safe-systems-programming.

[106] Raj Vengalil. Building the azure iot edge security daemon in rust. 2019. https://msrc-blog.microsoft.com/2019/09/30/building-the-azure-iot-edge-security-daemon-in-rust.

[107] Cloudflare. https://www.cloudflare.com.

[108] wirefilter: An execution engine for wireshark-like filters. https://github.com/cloudflare/wirefilter.

[109] Cloudflare spectrum: Ddos protection for tcp and udp services. https://www.cloudflare.com/products/cloudflare-spectrum.

[110] Ingvar Stepanyan and Andrew Galloni. Building fast interpreters in rust. 2019. https://blog.cloudflare.com/building-fast-interpreters-in-rust.

[111] Boringtun: Userspace wireguard implementation. https://github.com/cloudflare/boringtun.

[112] Vlad Krasnov. Boringtun, a userspace wireguard implementation in rust. 2019. https://blog.cloudflare.com/boringtun-userspace-wireguard-rust.

[113] Wireguard. https://www.wireguard.com.

[114] Google. https://google.com.

[115] Gousers - companies currently using go throughout the world. 2019. https://github.com/golang/go/wiki/GoUsers.

[116] Paul Liu. How 500px serves up over 500tb of high res photos. 2015. https://developers.500px.com/how-500px-serves-up-over-500tb-of-high-res-photos-fa81a376.35tz2wtg2.

[117] Dns server in go - big ntp pool upgrade. 2012. https://news.ntppool.org/2012/10/new-dns-server.

[118] Saroj Yadav, Matthew Smillie, Mike Demmer, and Tyler Johnson. Scaling slack's job queue. 2017. https://slack.engineering/scaling-slacks-job-queue-687222e9d100?gi=6f8eabdb3848.

[119] David Fullerton. Announcing bosun, our new open source monitoring & alerting system. 2014. https://stackoverflow.blog/2014/11/11/announcing-bosun-our-new-open-source-monitoring-alerting-s

[120] Bosun - time series alerting framework. https://github.com/bosun-monitor/bosun.

[121] Marcio Castilho. Handling 1 million requests per minute with go. 2015. http://marcio.io/2015/07/handling-1-million-requests-per-minute-with-golang.

[122] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[123] Andrew Gerrand. Share memory by communicating. 2010. https://blog.golang.org/share-memory-by-communicating.

[124] John Graham-Cumming. Go at cloudflare. 2012. https://blog.cloudflare.com/go-at-cloudflare.

[125] Sean Gallagher. Cloudflare blows hole in laws of web physics with go and railgun. 2013. https://arstechnica.com/information-technology/2013/02/cloudflare-blows-hole-in-laws-of-web-physics-with-go-and-r

[126] Jennie Lees. Managing and scaling real time data pipelines using go. 2016. https://www.youtube.com/watch?v=2s5l9ErNL3s.

[127] The go programming language specification. 2019. https://golang.org/ref/spec.

[128] The c++17 official standard. 2017. https://www.iso.org/standard/68564.html.

[129] Kai Wei. How we built uber engineering's highest query per second service using go. 2016. https://eng.uber.com/go-geofence.

[130] Scott Mansfield, Vu Tuan Nguyen, Sridhar Enugula, and Shashi Madappa. Application data caching using ssds. 2016. https://netflixtechblog.com/application-data-caching-using-ssds-5bf25df851ef.

[131] Matt Robenolt. Trying out this go thing... 2013. https://blog.disqus.com/trying-out-this-go-thing.

[132] Update on disqus: It's still about realtime, but go demolishes python. 2014. http://highscalability.com/blog/2014/5/7/update-on-disqus-its-still-about-realtime-but-go-demolishes.html.

[133] Kubernetes - production-grade container orchestration. https://kubernetes.io.

[134] Kubernetes repository. https://github.com/kubernetes/kubernetes.

[135] Moby. https://mobyproject.org.

[136] Moby repository. https://github.com/moby/moby.

[137] Docker. https://www.docker.com.

[138] Vitess. https://vitess.io.

[139] Vitess repository. https://github.com/vitessio/vitess.

[140] Joab Jackson. Youtube scales mysql with go code. 2012. https://www.computerworld.com/article/2493815/youtube-scales-mysql-with-go-code.html.

[141] Aaron Weiss, Daniel Patterson, and Amal Ahmed. Rust distilled: An expressive tower of languages. *CoRR*, abs/1806.02693, 2018.

[142] Cwe-193: Off-by-one error. https://cwe.mitre.org/data/definitions/193.html.

A Questionnaire

The questions from the questionnaire, (previously) found at `https://docs.google.com/forms/d/e/1FAIpQLScz9xY2WuUc4xciQoP8b8XgDEvh2qcYV8OCyw4VSk30fYILjA/viewform`, are reproduced below.

What language(s) do you use outside of "traditional" languages (such as C/C++/Java/Python)?

- ☐ Chapel
- ☐ Clojure
- ☐ D
- ☐ Elixir
- ☐ Erlang
- ☐ Go
- ☐ Haskell
- ☐ Kotlin
- ☐ Nim
- ☐ Rust
- ☐ Scala
- ☐ Swift
- ☐ Typescript
- ☐ Zig
- ☐ Other: ...

What kind of system have you worked on (using the language(s) in 1)?
In the following categories "local" should be seen as local to a single machine, i.e. not distributed.

- ☐ Compiler
- ☐ Distributed compiler
- ☐ Local storage (e.g. SQL databases, graph databases, key-value stores)
- ☐ Distributed storage
- ☐ Local data processing
- ☐ Distributed data processing
- ☐ Local scheduling
- ☐ Distributed scheduling (e.g. Kubernetes)
- ☐ Local deployment
- ☐ Distributed deployment (e.g. Kubernetes)
- ☐ Operating System
- ☐ Distributed Operating System
- ☐ Local queueing/messaging
- ☐ Distributed queueing/messaging (e.g. NATS)
- ☐ Local ledger/blockchain (e.g. Parity Ethereum)
- ☐ Distributed ledger/blockchain
- ☐ Other: ...

Why was this/these language(s) chosen?

- ☐ Language features
- ☐ General tooling around the language
- ☐ Tooling for distributed system for the language
- ☐ Fault tolerance
- ☐ Isolation
- ☐ Language ecosystem
- ☐ Productivity
- ☐ Personal interest
- ☐ Other:

What are the strong points of the language(s)?

Open answer

What are the weak points of the language(s)?

Open answer

How important is existing tooling when choosing a language?

	1	2	3	4	5	
Not important	◯	◯	◯	◯	◯	Very important

How large is the team working on the system?

- ◯ < 5
- ◯ < 10
- ◯ < 20
- ◯ < 50
- ◯ > 50

On how many machines does the system run?

- ◯ O(1)
- ◯ O(10)
- ◯ O(100)
- ◯ O(1,000)
- ◯ O(10,000)

Additional information
Anything else you would like to share.

Open answer

B Questionnaire results

The results of the questionnaire, (previously) found at `https://docs.google.com/forms/d/1iGmcLh6K_7a4BnHWF501jChYOwOkIBa5EFhnuXJpfFA/viewanalytics`, are reproduced below. In total the questionnaire received 37 responses.

Figure 1: Responses to question 1. What language(s) do you use outside of "traditional" languages (such as C/C++/Java/Python)?

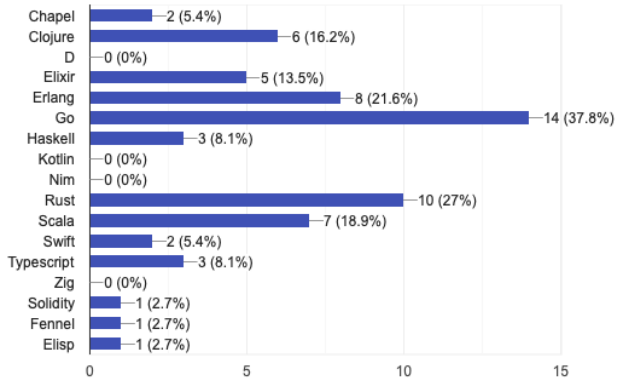


Figure 2: Responses to question 2. What kind of system have you worked on (using the language(s) in 1)?

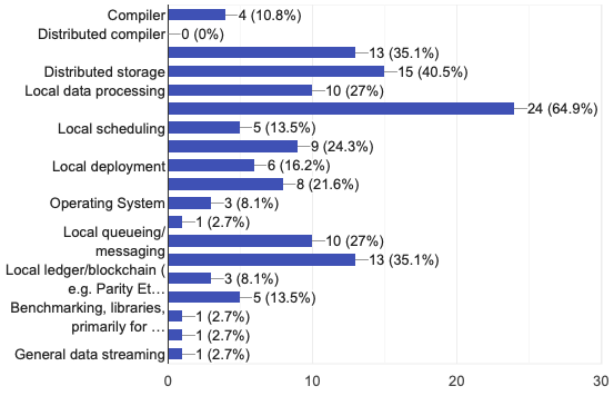
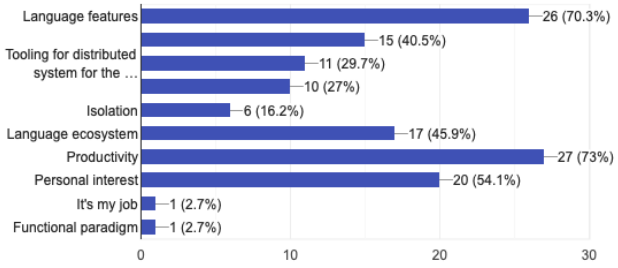


Figure 3: Responses to question 3. Why was this/these language(s) chosen?



Responses to question 4. What are the strong points of the language(s)?

- Parallelism and locality are first-class concepts in the language; productivity features
- For Haskell, eliminating runtime errors and lightweight concurrency with reasonable primitives. For Rust, low level control with a great deal of safety.
- Java/C/C++ are actually the worst options. All the other languages while more niche offer certain technical benefits.
- - very easy to learn and become productive - good production tooling: profilers, tracers
- Isolated process (actors) as first class citizens of the language. Distribution and scalability.
- High performance (due to LLVM optimization, etc.), strong static analyzer (e.g. borrow checking to guarantee correctness in preventing memory leaks, use-after-free, etc.)
- elixir ftw
- Mature VM, low-level details like memory handling already handled by language, mature concurrency primitives and libraries, FP support
- Syntax, type-safety, performance, low memory usage
- Expressiveness
- Functional programming makes concurrency much easier; a good repl makes debugging very nice.
- Extremely powerful debugging & tracing tools, strong dynamic typing, the ease of parallelising various problems' solutions
- Pervasive focus on isolation and fault tolerance makes it easy to build resilient distributed systems

- Haskell is pure and elisp is consistent to work with.
- Memory safety, convenient libraries for productive programming
- Strict typing with great tools to make this less burdensome. Great language design and syntactical sugar.
- go and grpc work seamlessly together/abstractions fit very well obvious, but important: naturally concurrent very easy to test on das due to static compilation
- Tooling is massively useful, asynchronous primitives make many things much easier
- Performance, Memory Safety
- Relatively simple, good stdlib, decent ecosystem of other libs
- Consise, very data-driven, functional, maps very well to distributed data processing.
- Documentation for the languages I mentioned is good. Easy to use and quite intuitive. I generally use languages when required by a course or required by the place where I work. So I do not try out a language based on just interest.

Responses to question 5. What are the weak points of the language(s)?

- Small community (so far)
- Haskell doesn't have very good editor tooling, still. Both are not well known.
- None. Elixir for example is strictly superior to C++ for writing high reliability, fault-tolerant web services.
- - not a systems programming language; poor control over scheduling and memory allocation - various concurrency primitives don't compose: channels, condition variables, context.Context cancelation
- Low performance in cpu intensive computations.
- Steep learning curve to make use of advanced features
- elixir wtf
- The opposite of above points
- Tooling
- Dynamic typing in clojure
- Erlang and Clojure both have a strong disregard for usability; stack traces are ugly and obscure important information.
- Really basic UI capabilities (it would be nice sometimes to create a simple visualisation easily), no JIT
- Small community, lack of high performance computing without resorting to native code
- Writing concurrent data structures with shared state is tricky (rightly so); writing pointer-based datastructures is more challenging than in C/C++
- Lack of "advanced" features. In Swift particularly, missing coroutine support in the exposed language.
- we had data race bugs -> languages like rust can statically prevent. especially important for systems level code like this go does make concurrency very easy, but doesn't help at all with making concurrent code correct -> dangerous IMHO simplistic type system leads to repetitive code: no "findIndex" on array, etc. -> another potential source for bugs Go doesn't (or didn't) have a (versioning) package manager. Making sure everybody has the same packages installed at the same versions was a pain and is a bane to reproducibility in the scientific setting
- No generics or unions is pretty annoying, and the package management system kinda sucks
- Learning Curve
- versioning & dependencies, perf & scale regressions, lack of important constructs like const pointers
- Community is rather small, its development is tied to a single commercial organization (Cognitect), performance is not as good as java.
- It is not easy to integrate these with a lot of existing applications. Most support comes only with traditional languages. Choosing 'other' languages has to be done carefully and according to specific use case.

Figure 4: Responses to question 6. How important is existing tooling when choosing a language?

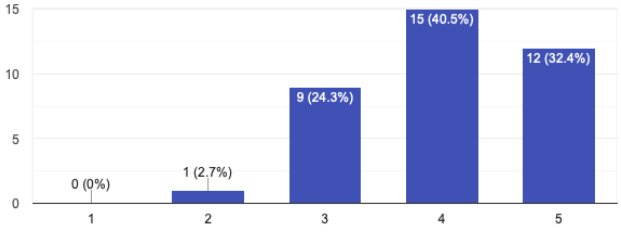


Figure 5: Responses to question 7. How large is the team working on the system?

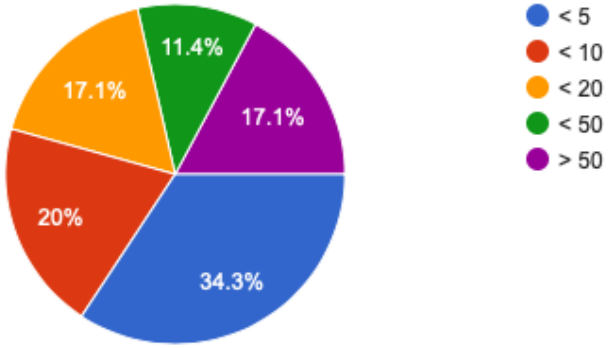
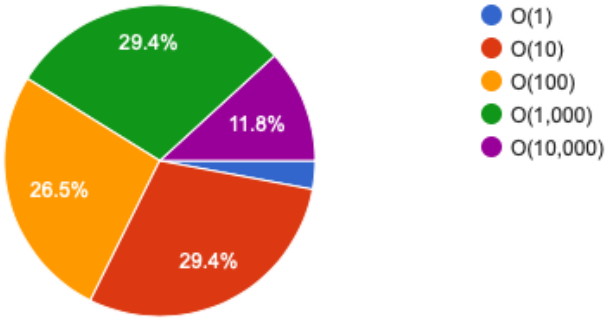


Figure 6: Responses to question 8. On how many machines does the system run?



Responses to question 9. Additional information

- I'm understanding "machines" in the previous question to mean "distinct compute nodes on a cluster or cloud". If that's not the correct interpretation, I might need to answer differently.
- The system in question is CockroachDB.
- The last question is not very accurate. In my case the system can be run on 2 machines or 13-15 depending on the load it needs to support. Usually the cluster of MongooseIM nodes is aroud 3-4 nodes. The biggest installation I worked with consisted of 12 Erlang clusters with 12 Erlang nodes each.
- Most of my distributed system work is already handled by Tendermint so our job is easy