Comparing Languages for Engineering Server Software: Erlang, Go, and Scala with Akka

Ivan Valkov, Natalia Chechina, and Phil Trinder
School of Computing Science, University of Glasgow G12 8RZ, United Kingdom
Iv.V.Valkov@gmail.com, {Natalia.Chechina, Phil.Trinder}@glasgow.ac.uk

Abstract

Servers are a key element of current IT infrastructures, and must often deal with large numbers of concurrent requests. The programming language used to construct the server has an important role in engineering efficient server software, and must support massive concurrency on multicore machines with low communication and synchronisation overheads.

This paper investigates 12 highly concurrent programming languages suitable for engineering servers, and analyses three representative languages in detail: Erlang, Go, and Scala with Akka. We have designed three server benchmarks that analyse key performance characteristics of the languages. The benchmark results suggest that where minimising message latency is crucial, Go and Erlang are best; that Scala with Akka is capable of supporting the largest number of dormant processes; that for servers that frequently spawn processes Erlang and Go minimise creation time; and that for constantly communicating processes Go provides the best throughput.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures

Keywords Server applications, programming languages, Erlang, Go, Scala, Akka

1. Introduction

Modern web service applications like social networks, online games, and chat applications get increasingly large loads to handle. Millions of users can be simultaneously interacting with a server. When the traffic grows significantly, new hardware may be added (increasing operating costs), or the system may be unable to meet the demand and must be redesigned. To minimise the hardware requirements and hence operating costs the server programming language must effectively utilize hardware resources.

Many early concurrency models were very low level, e.g. C with PThreads, and challenged the developer with issues like deadlock, livelock and race conditions. To minimise development time and reduce software development costs many servers are now engineered in languages that combine high level computational models, e.g. functional or object-oriented, with high-level coordination models, e.g. actors as in Erlang [2] or a process algebra as in Go [6]. Indeed, the success of some server-based companies is even attributed to their use of specific languages. As examples WhatsApp's success is attributed to their use of Erlang [27]; the video streaming leader Twitch uses Go to serve millions of users a day [13]. Research contributions of this paper include the following:

- A survey of programming language characteristics relevant for servers (Section 2.1).
- The design and implementation of three benchmarks to analyse the multicore server capabilities of Erlang, Go, and Scala with Akka (Section 3).
- An evaluation of Erlang, Go, and Scala with Akka for key server capabilities, i.e. process communication latency, spawn time, maximum number of supported processes, and process communication throughput (Section 4).

SAC 2018, SAC 2018: Symposium on Applied Computing, April 9–13, 2018. Pau, France.

ACM 978-1-4503-5191-1/18/04…$15.00.
http://dx.doi.org/10.1145/10.1145/3167132.3167144

2. Server Languages

2.1 Language Characteristics

Table 1 analyses the features of languages commonly used to implement web services, like web servers or instant messaging servers. The table covers computation, coordination, compilation, and popularity. While assembling the table we did not aim to include every possible server language in it, but rather to cover representative languages that reflect important characteristics of server languages. The languages in the table are presented in accordance with their computation model.

Computation models are well known. Procedural languages like C, Go, and Rust encourage programming with data structures, and functions. Object oriented languages like Java and C# encapsulate data structures with the methods that operate on them to facilitate composition and re-use. Functional languages like Clojure, Elixir, Erlang, and Haskell primarily use pure functions and immutable data structures. Supporting multiple models, or paradigms, has recently become popular, and some server languages follow this trend, e.g. F# is considered both object oriented and functional.

Typing. The majority of modern languages are strongly typed, and hence report compile or runtime errors rather than failing in unexpected ways. Of the languages considered only C is weakly typed. Four of the languages have significant dynamic typing: Erlang, Elixir, Scala with Akka (Scala/Akka), and Clojure. The remaining languages are primarily statically typed, enabling early error detection [20] and faster runtimes as there are no dynamic type checks.

Computation abstraction. Due to the complexity, size of web applications, and pressure to deliver services to the market quickly, low level languages are rarely used for engineering servers. Hence the only low level language we consider is C.

Table 1. Server Language Characteristics

	Computation			Coordination			Compilation	Popularity	
Language	**Model**	**Typing**	**Abstraction**	**Model**	**Abstraction**	**Determinism**	**Runtime environment**	**Tiobe Apr 2017**	**RedMonk Jan 2017**
C + OpenMP	Procedural	Weak Static	Low	Annotations	High	No	Native	2	9
C + PThreads	Procedural	Weak Static	Low	Explicit	Low	No	Native	2	9
Go	**Procedural**	**Strong Static**	**High**	**CSP**	**High**	**No**	**Native**	**19**	**15**
Rust	Procedural	Strong Static	High	Explicit	High	No	Native	45	∼40
C#	Object Oriented	Strong Static	High	Explicit	High	No	Native	4	5
Java	Object Oriented	Strong Static	High	Explicit	High	No	JVM	1	2
F#	Functional Obj. Orien.	Strong Static	High	Explicit	High	No	Native	29	∼40
Clojure	Functional	Strong Dynamic	High	STM	High	No	JVM	50	20
Elixir	Functional	Strong Dynamic	High	Actors	High	No	Erlang VM	50+	∼40
Erlang	**Functional**	**Strong Dynamic**	**High**	**Actors**	**High**	**No**	**Erlang VM**	**42**	**26**
Scala + AKKA	**Functional**	**Strong Static**	**High**	**Actors**	**High**	**No**	**JVM**	**32**	**14**
Haskell	Pure Functional	Strong Static	High	Eval Strat	High	Yes	Native/GHCi	40	16

Coordination model. Exploiting the capabilities of multicore architectures with massive concurrency is a key design goal for server software engineering. The coordination model of a language strongly influences the performance achieved. Different models make escaping common pitfalls like deadlock and livelock easier. *Explicit* concurrency is used in languages like Java, Rust, and C with PThreads. Here the programmer works with threads and uses mutexes to protect shared resources. This approach is relatively low level and can lead to problems like race conditions, livelock, or deadlock. *Actor* concurrency [12] is used in languages like Erlang, Elixir, and the Akka framework. The unit of computation is the actor, and actors communicate by passing messages. Moreover, actors are completely isolated from each other and do not share memory. This provides a higher level of abstraction than the explicit model and makes creating complex systems easier. Go, on the other hand, bases its concurrency model on the Communicating Sequential Processes (CSP) process algebra [14]. The main building blocks in the language are goroutines (that resemble lightweight threads) and channels for communication between the goroutines.

Coordination abstraction. As for computation abstraction, low level coordination abstraction makes building web applications difficult. So, C with PThreads is the only programming language with low level coordination abstraction considered here.

Coordination determinism. As a pure functional language, only Haskell provides deterministic parallelism, i.e. guarantees that the parallel program computes the same value as a sequential counterpart.

Compilation. The type of compilation – either directly to machine code or to some intermediate language for a Virtual Machine (VM) – impacts performance. Moreover VMs impose restrictions, e.g. on the number of threads that can be maintained.

Popularity is a measure of how widely the selected languages are used, and we consider two popularity rankings. The Tiobe Index [33] measures the popularity of programming languages based on the number of search engine results for queries containing their name. The RedMonk [26] measures programming traction on GitHub and Stack Overflow.

2.2 Selected Languages

For detailed investigation we select three representative languages: Erlang (Section 2.2.1), Scala/Akka (Section 2.2.2), and Go (Section 2.2.3). All three languages have high level computation and coordination models with excellent support for concurrency and are used in industry for engineering scalable and high performance servers.

The languages selected enable us to compare the impact of different language characteristics on server performance. We investigate different concurrency models as Erlang and Scala use actors while Go uses Communicating Sequential Processes (CSP). We investigate the impact of the runtime environment as Scala uses the Java Virtual Machine (JVM), Erlang the Erlang VM and Go compiles to native code. We investigate different typing regimes as Erlang and Akka are dynamically typed, while Go is statically typed.

2.2.1 Erlang

Erlang is a functional programming language originally developed at the Ericsson Computer Science Laboratory in 1986 by Joe Armstrong [2] which was then released as open source in 1998. It was designed to meet requirements of distributed, massively concurrent, and fault tolerant systems. The language has strong, dynamic typing, and includes garbage collection. It compiles to byte code that runs on the Erlang virtual machine.

Concurrency Model. From the very beginning Erlang has been designed to be highly concurrent, and this is one of the primary strong points of the language. Erlang adopts the actor model, where the primitive computation unit is the actor, and actors communicate with each other via message passing. Compared with the explicit model, where shared mutable state has to be protected with locks and mutexes, Erlang's approach is less susceptible to the common pitfalls of the prior – race conditions and deadlock, for example. This is achieved by asynchronous massage exchanges between actors,

without the need to block until their massage has been received. Moreover, in Erlang isolated processes do not share memory and variables are immutable, i.e once a value is assigned, it cannot be changed.

Fault tolerance is another important part of Erlang's design. The language follows the "let-it-crash" principle, which is a non-defensive way of programming. It puts the burden of error handling to the Erlang VM, where actors can supervise other actors and take actions in the event of a failure. This is achieved be means of the mentioned earlier avoidance of shared memory and a well structured process supervision model.

Use Cases. Originally developed for use in telecommunication systems, Erlang's key characteristics, including run-time safety, concurrency, and distribution, are highly relevant in today's server environment. Erlang is considered to be behind WhatsApp's success [27]. Erlang's fault tolerance and performance are some of the reasons Bet365, a leader in online betting services, uses the language [3]. Erlang is also the language of choice in the multiplayer video games Call of Duty [7] and League of Legends [25], and the NoSQL database CouchDB [1].

2.2.2 Scala and Akka

Scala is a programming language designed in 2003 by Martin Odersky at EPFL and is now an open source project [22]. It combines object-oriented and functional features, i.e. every value is an object and every operation is a method call, while the language features many functional tendencies like currying, first-class functions, immutability, and lazy evaluation.

Scala has strong, static typing. However, when the Akka framework [32] is used the messages sent between actors are type checked during runtime, therefore resembling dynamic typing. Due to the fact that Scala compiles to byte code that runs on the Java Virtual Machine, Java and Scala classes can be freely mixed. This is a huge advantage for the language because it gets exposed to a large community.

Concurrency Model. In this project Scala is used with the Akka framework. Akka is a toolkit for building highly concurrent, distributed, and resilient applications on the JVM based on the actor model. Therefore, it can be expected that the features Akka provides are similar to those of Erlang. Actors are still the primitive units of computation and they communicate through messages. Similarly to Erlang, Akka does not support guaranteed delivery of these messages. Moreover, the fault tolerance in Akka uses the same principles of supervisor hierarchy and "let-it-crash" approach to failures like Erlang. However, there are some differences between the two technologies. Since Akka runs on top of the JVM, enforcing true data separation between actors would hurt performance a lot (basically checking every single message) [17]. Therefore, by passing reference of mutable data structures, it is possible to have shared data between actors in Akka. Doing this, however, is considered a bad practice, but is a good example of some of the limitations the JVM puts on a toolkit like Akka [18].

Use Cases. Scala with Akka support the development of scalable high-load services. Companies like William Hill use Scala and Akka to build their multi million user betting applications; some financial companies like UniCredit and Barclays use them as well [18]. The reasons these companies choose Scala and Akka include the need for performance, scalability, and fault tolerance. Moreover, the familiarity with the JVM and the community behind it are also noted as important.

2.2.3 Go

Go is an open source programming language initially designed at Google in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson [6]. It was designed with the idea of combining expressiveness like a dynamically typed interpreted language and the safety of statically typed compiled language. Even though it has a strong, static type system, it is designed to be simpler than comparable typed languages. Go inherently supports garbage collection and concurrency. The language is compiled to machine code. The way Go supports object oriented design is through interfaces. However, the designers of the language deliberately omitted some concepts like inheritance or generics. The reasoning behind this is to both reduce complexity in the type system and run-time, and to keep the design of the language simple.

Concurrency Model. Good concurrency support in Go has been considered since the earliest stages of its design [10]. The language bases its concurrency model on Hoare's Communicating Sequential Processes [14]. The main building blocks in the language's concurrency are goroutines, which resemble lightweight threads, and channels, which are used for communication between the goroutines. The idea of *"do not communicate by sharing memory; instead, share memory by communicating"* [9] summarizes Go's approach to concurrency. Instead of explicitly using locks to mediate access to shared data, Go encourages the use of channels to pass references to data between goroutines. This way it is ensured that at any given time only one goroutine has access to the data.

Use Cases. Born out of frustration with several software systems in Google that were suffering from high complexity, Go has now been used by many different companies outside its birthplace. Web services are one of the big use cases of the language. The video streaming leader Twitch mainly uses Go for many of their busiest systems including the streaming and chat services [13]. The deployment tool Docker is also almost entirely written in Go [23]. Netflix is another example of a big company that has adopted Go [19].

3. Benchmarks for Server Languages

3.1 Server Architecture Patterns

To evaluate server languages realistically we must understand common server architectural patterns. These include listener/worker, leader/followers, workpool, and forking server [11, 29]. Figure 1 shows a simplified architecture of a server containing entities similar to most patterns. The *Clients* send simultaneous requests to the server. A *Dispatcher* handles the requests, forwarding them to workers. A *Worker* processes requests and returns the results to the clients.

Server architecture patterns share a number of similarities: a worker processes a single request at a time; at least one message is sent between processes in the server for every request (*Dispatcher* to *Worker*). Therefore, *to maintain low latency and high throughput a server language must provide fast communication between processes and support a sufficiently large number of worker processes.*

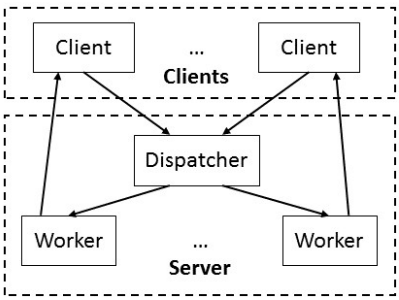


Figure 1. A Typical Web Application Architecture

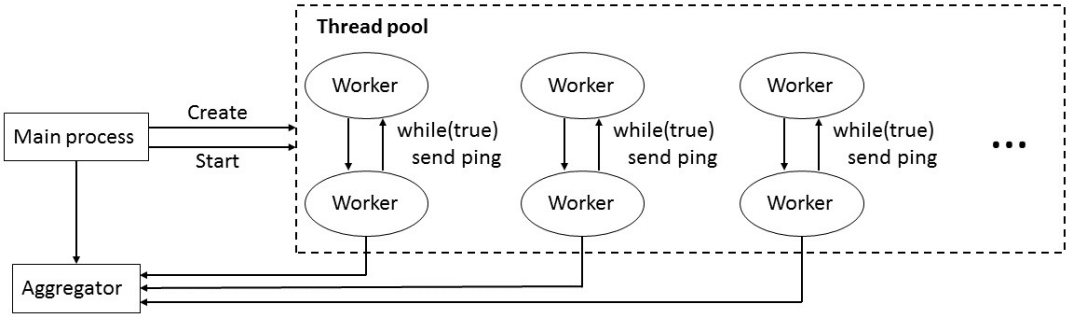


Figure 2. Design of the Concurrent Process Throughput Benchmark

3.2 Programming Language Benchmarks

Many studies compare programming languages using sets of benchmarks, e.g. [16] analyses performance, scalability, ease of implementation, and maintainability of Erlang, Go, and F# using matrix multiplication and finding prime numbers. In [34] authors also use matrix multiplication to benchmark concurrency of Go and Java. The reason for the wide use of matrix multiplication is due to the operations of the algorithm being independent from each other. This means that matrix multiplication is easy to parallelize making it appealing for benchmarking concurrent technologies.

Other algorithms used to compare programming language performance include graph traversal, MapReduce, and spectral methods to compare Erlang and C++ [21] and a parallel version of the optimal binary search to compare Go and C++ [30].

The benchmarks above compare the performance of programming languages for completing some computational task. While computation is a factor for modern server applications, the primary performance bottlenecks centre around concurrency, i.e. the sheer number of requests they need to handle [15]. Therefore, benchmarks measuring language support of massive concurrency in terms of communication cost and the number of processes that can be spawned are most relevant for server applications.

Such concurrency oriented benchmark suites are less common that computation oriented benchmarks. One such set of benchmarks is the Intel MPI Benchmarks [31] that evaluate the efficiency of the most important functionalities of the Message Passing Interface (MPI) library, as well as the performance of a set of processors running algorithms concurrently. In [5] MPI Benchmarks' implementations in Java, Erlang and Scala are used to compare the languages in the context of Web 2.0 applications. The limitation of using these 'microbenchmarks' is the fact that they focus only on one aspect of a server at a time. An alternative approach is to use a generic server to measure multiple aspects; for example, the `genstress` BenchErl benchmark [28] analyses latency, throughput, and the number of concurrent connections.

3.3 Server Benchmark Designs and Implementations

We present the design and motivation of three benchmarks designed to evaluate the key performance aspects of server languages for the study in Section 4.

Process Communication Latency. Although simplified, Figure 1 enables us to visualise the impact of significant loads on a server. When clients flood the server with requests, the *Dispatcher* must be able to quickly distribute the workload to *Workers*. Hence fast inter-processes communication is vital. The microbenchmark design is based on Intel's MPI PingPing benchmark [31], that measures message latency in languages. In the microbenchmark *ProcessA* sends a message of arbitrary size to *ProcessB*; and *ProcessB* asynchronously sends the same message back to *ProcessA*. The time to send and receive the message back is collected after an arbitrary number of repetitions.

Process Creation Time & Maximum Processes Supported. As depicted in Figure 1, many servers spawn worker processes to deal with client requests, and hence fast process creation time and the ability to support high numbers of concurrent processes are crucial. Moreover, many server processes are idle, awaiting client interaction. To test these properties this microbenchmark spawns N dormant processes, measures the spawn time, increases N and repeats. Dormant processes are inactive once spawned.

Concurrent Process Throughput. The two previous microbenchmarks examine key server characteristics: communication and process spawning. This more realistic benchmark measures the throughput of a system consisting of a number of worker-client process pairs constantly exchanging messages. Various examples of such application can be seen in industry, e.g. online games, video chats, and streaming applications [8]. In online games, for example, clients constantly send information about their positions, status, and actions of players, and then the server responds with data about the environment and any events happening around the players. Usually for each client connection there is a process on the server responsible for it, requiring high throughput.

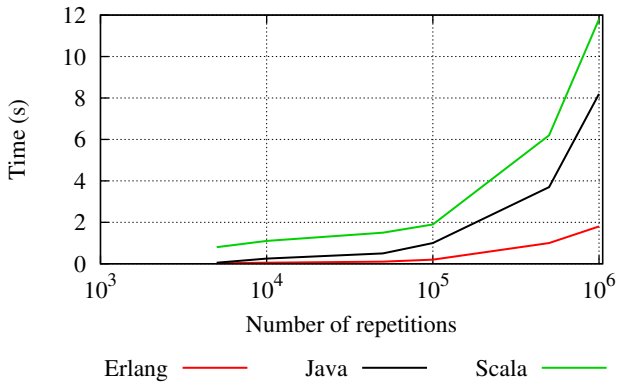
The benchmark design is presented in Figure 2. The *Thread pool* contains initialized *Worker* processes that wait for a start message from the *Main process*. The *Aggregator* collects the number of messages passed in the system over time. Each *Worker*-*Worker* pair periodically reports to the *Aggregator* the time taken to exchange a certain number of messages. This number of messages is chosen to be small enough to provide accurate information, but large enough to ensure that the system does not spend much of time on the aggregation. Experiments with different values showed that 10,000 messages work the best.

4. Evaluation

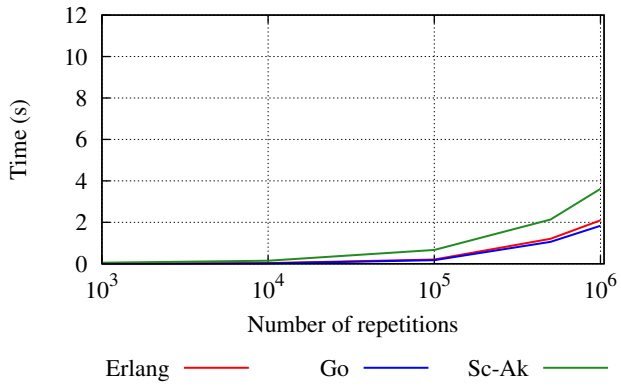
This section evaluates the performance of Go, Erlang, and Scala/Akka on multicore machines using the benchmarks from Section 3.3. In the experiments we use the following versions of the languages: Erlang/OTP 19.0, Go 1.7.1, Scala 2.12.0 with Akka 2.4.12. The benchmarks are measured on two machines:

- (mini-server) Intel Core i5-3230Mv2, 2.60GHz, 8GB RAM, Windows 10 (Sections 4.1 and 4.2);
- (medium-server) Intel Xeon E5-2640v2 16 Cores, 2GHz, 64GB RAM, 4GB RAM/core, Scientific Linux 6 (Section 4.3).

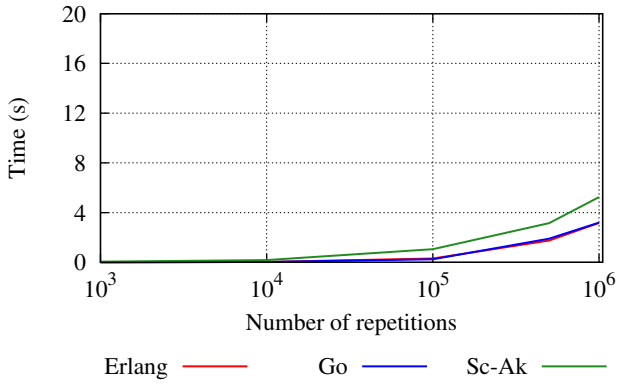
Unless otherwise stated the message size is selected to be 500 bytes, which is close to real life use cases [8]. All experiments are repeated seven times and the median values are reported. The experiment data, additional figures and analysis are at `https://github.com/bbstk/server-languages-benchmarks`.



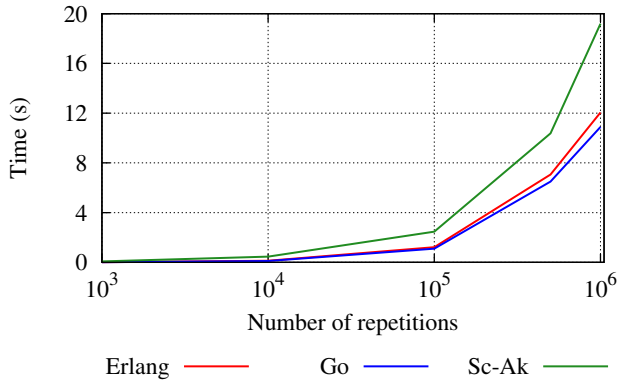
(a) 5kB Message Size [5]



(b) 5kB Message Size

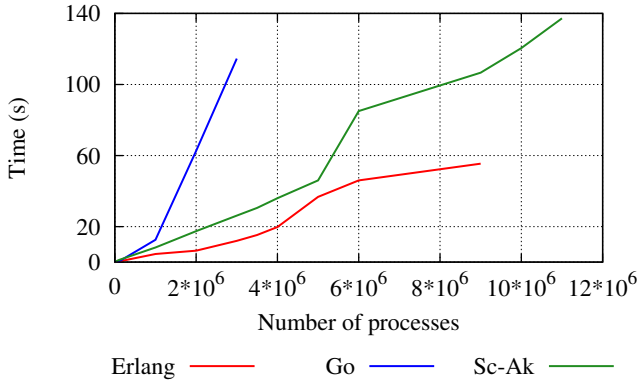


(c) 10kB Message Size

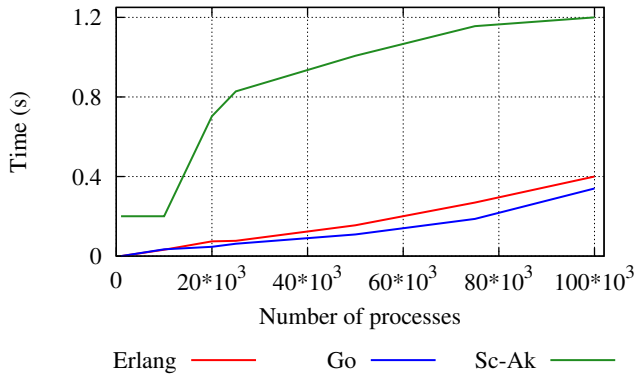


(d) 50kB Message Size

Figure 3. PingPing Benchmark



(a) Max Number of Supported Processes

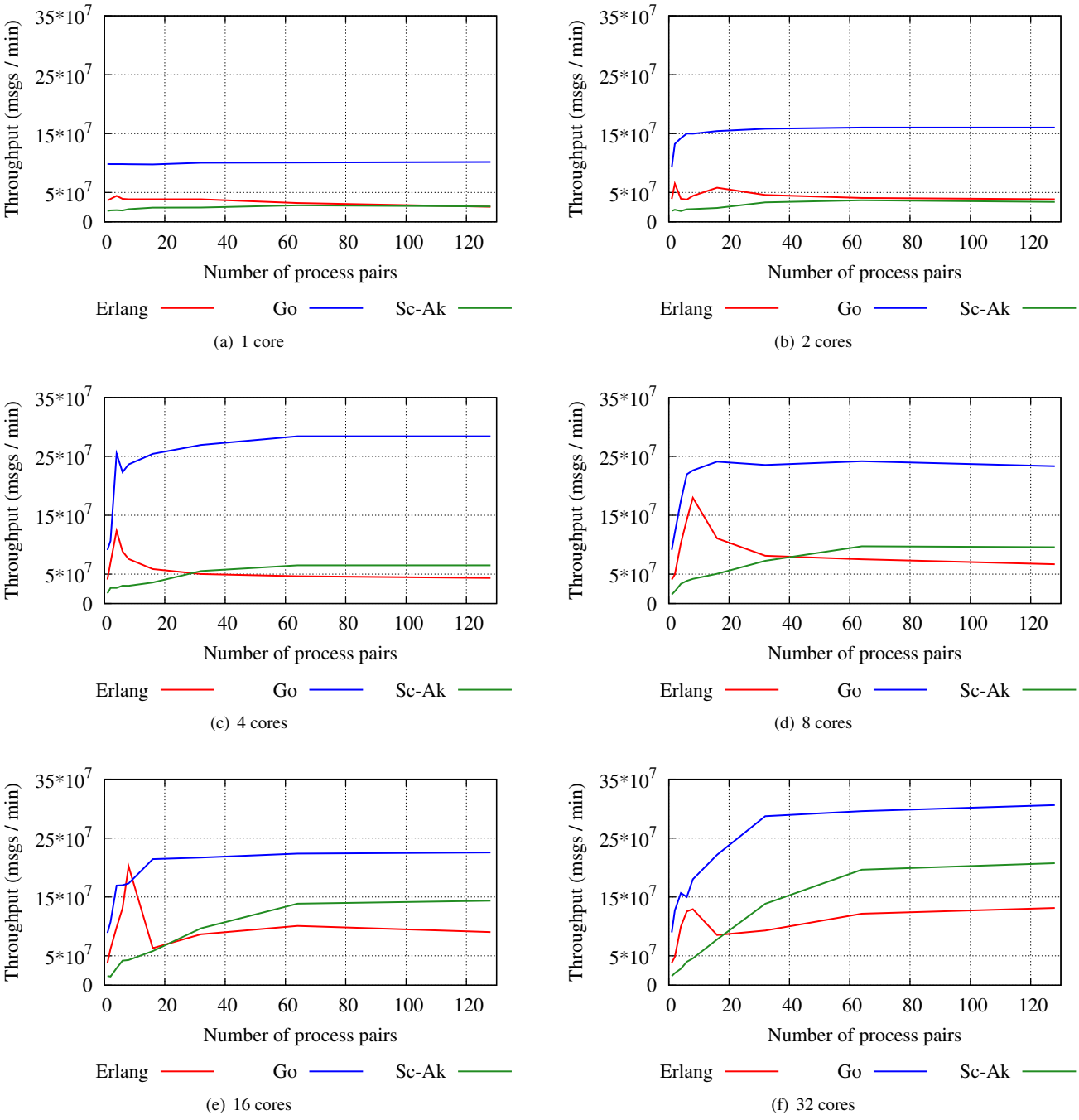


(b) Up to 100,000 Processes

Figure 4. Spawn Time

4.1 Process Communication Latency

We replicate the methodology from [5]: for each of the languages we run the benchmark with three message sizes: 5kB, 10kB, and 50 kB. The time for each message size is recorded after 1,000, 10,000, 100,000, 500,000, and 1,000,000 repetitions of pairs of processes exchanging messages.



(a) 1 core (b) 2 cores (c) 4 cores (d) 8 cores (e) 16 cores (f) 32 cores

Figure 5. Concurrent Process Throughput

The primary observations from Figures 3(b), 3(c) and 3(d) are that Scala/Akka has higher communication latency than Erlang and Go, and the latter have very similar latencies, at all message sizes.

The results in Figure 3(b) concur with Erlang performance results in [5] and reproduced in Figure 3(a). Figures 3(a) and 3(b) show that *Akka significantly reduces communication latency in Scala* (3 times). We believe this is due to the event-based execution model in Akka being more efficient than the combination of thread-based and event-based execution in Scala [24].

4.2 Process Creation Time & Maximum Processes Supported

Recall that this microbenchmark spawns a defined number of dormant processes and returns the time taken to perform the task. Figure 4(a) shows that Scala/Akka can support the largest number of processes: ∼11M, which we believe is due to more efficient memory management in the JVM than in Erlang and Go. Erlang has the fastest spawn time, e.g. it takes Erlang 12s to spawn 3M processes, while Scala/Akka and Go need 26s and 114s respectively.

We attribute this to processes (actors) being central to Erlang and Scala/Akka, while goroutines are less central to Go.

In the second part of the experiment we analyse spawn time of up to 100,000 processes – a more realistic number of processes. Specifically we measure spawn time for 1,000, 10,000, 20,000, 25,000, 50,000, 75,000, and 100,000 processes. Figure 4(b) shows that Erlang and Go scale similarly with steady increases in spawn time. In Scala/Akka not only is spawn time greater than for Erlang and Go, it scales unpredictably with a sudden increase from 10,000 to 20,000 processes. This could be due to a number of reasons, e.g. the JVM garbage collection, or Akka actors implementation, or some internal Akka data structures.

We conclude that *Scala/Akka enables the maximum number of dormant processes; Erlang minimises spawn time; and Erlang and Go process creation time scales smoothly to 100,000 processes*.

4.3 Concurrent Process Throughput

Recall that this benchmark spawns pairs of communicating processes and measures their throughput. The experiments are run on the Beowulf cluster (Section 4) with 16 physical (32 virtual) cores. We run the benchmark for 60s with different numbers of paired processes (1, 2, 4, 6, 8, 16, 32, 64, and 128 pairs) varying the number of cores. The measurements reported are for cold starts, i.e. we restart the Erlang and Scala VMs for each measurement.

Figure 5 compares the throughput of Erlang, Go, and Scala/Akka varying the number of cores. Go consistently demonstrates higher throughput (2–5 times) than Erlang and Scala/Akka. This is probably because Go passes messages over statically typed channels, and does not require the pattern matching on messages used in Erlang and Akka. While Erlang outperforms Scala/Akka at small scales (up to 8 cores and 20 process pairs), the situation is reversed at larger scales.

An analysis of the impact of the number of cores on the throughput in each language individually shows that Erlang and Go reach their peak when the number of process pairs in the system is equal to the number of cores. In contrast Scala/Akka does not exhibit a fast start, its throughput increases slowly until approximately 62 process pairs regardless of the number of cores. Other observations that we may investigate in the future include the following: (a) after the peak Erlang demonstrates significant drops in performance; (b) Go does not scale beyond 4 cores, i.e. on 32 cores Go has approximately the same throughput as on 4 cores. We conclude that *Go provides highest throughput for servers with constant communication*.

5. Conclusion

We have investigated programming language characteristics that support the engineering of multicore web servers. Crucially these languages must be able to support massive concurrency on multicore machines with low communication and synchronisation overheads. We have analysed 12 languages considering computation, coordination, compilation, and popularity, and selected three representative high-level languages for detailed analysis: Erlang, Go, and Scala/Akka (Section 2). We have designed three server benchmarks that analyse key performance characteristics of the languages, i.e. inter-process communication latency, process creation time, the maximum number of supported processes, and throughput (Section 3).

A summary of the recommendations based on this small set of benchmarks is as follows. For a server where minimising message latency is crucial, Go and Erlang are the best choice (Figure 3). Interestingly Akka significantly reduces communication latency in Scala (Figures 3(a) and 3(b)). Scala/Akka are capable of maintaining the largest number of dormant processes ($\sim$11M processes in Figure 4(a)), while Erlang performs the best when processes are short lived and the goal is to ensure minimal spawn time, e.g. Erlang takes 58s to spawn 9M processes (Section 4.2). In server applications where up to 100,000 processes are frequently spawned, Erlang and Go minimise process creation time and scale smoothly (Figure 4(b)). Experiments with communicating pairs of processes show that Go provides the highest throughput independent of the number of cores and the number of process pairs (Figure 5).

Comparing the performance of complete case study servers implemented in each language would significantly reinforce these results, and one possibility is an Instant Messaging (IM) benchmark [4]. It would also be interesting to study the performance overheads of providing fault tolerance and of recovering from faults, another key server capability. Finally we could compare the performance of server languages on distributed memory architectures, e.g. a cluster of multicores.

References

[1] Apache Software Foundation. Apache CouchDB. `http://couchdb.apache.org/`, 2017. [Online: December 13, 2017].

[2] J. Armstrong. *Programming Erlang: software for a concurrent world*. Pragmatic Bookshelf, 2007.

[3] Bet365. Tech blog. `http://bet365techblog.com/welcome-to-our-research-and-development-team`, 2013. [Online: December 13, 2017].

[4] N. Chechina, M. Moro Hernandez, and P. Trinder. A scalable reliable instant messenger using the SD Erlang libraries. In *Erlang'15*, pages 33–41. ACM, 2016.

[5] J. M. da Silva Jr, R. D. Lins, and L. M. dos Santos. Comparing the performance of Java, Erlang and Scala in web 2.0 applications. *IADIS International Journal on WWW/Internet*, 10(2):121–137, 2013.

[6] A. A. A. Donovan and B. W. Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.

[7] M. Dowse. Erlang and first-person shooters. `http://www.erlang-factory.com/upload/presentations/395/ErlangandFirst-PersonShooters.pdf`, 2011. [Online: December 13, 2017].

[8] S. Fiedler, M. Wallner, and M. Weber. A communication architecture for massive multiplayer games. In *NetGames'02*, pages 14–22. ACM, 2002.

[9] Google. The Go blog: Share memory by communicating. `https://blog.golang.org/share-memory-by-communicating`, 2010. [Online: December 13, 2017].

[10] R. Grieseme. The evolution of Go. `https://talks.golang.org/2015/gophercon-goevolution.slide`, 2015. [Online: December 13, 2017].

[11] B. Gröne and P. Tabeling. A system of patterns for concurrent request processing servers. In *VIKINGPLOP'03*, pages 61–93, 2003.

[12] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI'73*, pages 235–245. Morgan Kaufmann, 1973.

[13] R. Hiltner. Go's march to low-latency GC. `https://blog.twitch.tv/gos-march-to-low-latency-gc-a6fa96f06eb7`, 2016. [Online: December 13, 2017].

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

[15] J. C. Hu, I. Pyarali, and D. C. Schmidt. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. In *GLOBECOM'97*, volume 3, pages 1924–1931. IEEE, 1997.

[16] A. Järleberg and K. Nilsson. *Go, F# and Erlang*. Bachelor's thesis, KTH, School of Computer Science and Communication, 2012.

[17] Lightbend Inc. Akka and the Java memory model. `http://doc.akka.io/docs/akka/2.4.4/general/jmm.html`, 2015. [Online: December 13, 2017].

[18] Lightbend Inc. Case studies & stories. `http://www.lightbend.com/resources/case-studies-and-stories`, 2015. [Online: December 13, 2017].

[19] S. Mansfield, V. T. Nguyen, S. Enugula, and S. Madappa. Application data caching using SSDs. http://techblog.netflix.com/2016/05/application-data-caching-using-ssds.html, 2016. [Online: December 13, 2017].

[20] E. Meijer and P. Drayton. Static typing where possible, dynamic typing when needed. In *Workshop on Revival of Dynamic Languages*, 2005.

[21] S. Mikael. *Evaluation and Comparison of Programming Frameworks for Shared Memory Multicore Systems*. MSc thesis, Linköping University, 2014.

[22] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical report, Ecole Polytechnique Federale de Lausanne, 2004.

[23] J. Petazzoni. Docker + Golang = <3. https://blog.docker.com/2016/09/docker-golang/, 20116. [Online: December 13, 2017].

[24] D. Pollak, J. Zaugg, P. Haller, and V. Klang. All actors in scala compared. http://doc.akka.io/docs/misc/Comparison_between_4_actor_frameworks.pdf, 2010. [Online: December 13, 2017].

[25] M. Ptaszek. Riot games engineering. chat service architecture: Servers. https://engineering.riotgames.com/news/chat-service-architecture-servers, 2015. [Online: December 13, 2017].

[26] RedMonk. The RedMonk programming language rankings: March 2017. https://redmonk.com/sogrady/2017/03/17/language-rankings-1-17/, 2017. [Online: December 13, 2017].

[27] R. Reed. Scaling to millions of simultaneous connections. http://www.erlang-factory.com/upload/presentations/558/efsf2012-whatsapp-scaling.pdf, 2012. [Online: December 13, 2017].

[28] RELEASE Project. Benchmark suite for Erlang/OTP. http://release.softlab.ntua.gr/bencherl/benchmarks.html, 2014. [Online: December 13, 2017].

[29] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, volume 2. John Wiley & Sons, 2013.

[30] D. Serfass and P. Tang. Comparing parallel performance of Go and C++ TBB on a direct acyclic task graph using a dynamic programming problem. In *Southeast Regional Conference*, pages 268–273. ACM, 2012.

[31] G. Slavova. Intel MPI benchmarks. http://software.intel.com/en-us/articles/intel-mpi-benchmarks/, 2013. [Online: December 13, 2017].

[32] M. Thurau. Akka framework. Technical report, University of Lübeck, 2012.

[33] TIOBE software BV. Tiobe index. http://www.tiobe.com/tiobe-index/, 2017. [Online: December 13, 2017].

[34] N. Togashi and V. Klyuev. Concurrency in Go and Java: Performance analysis. In *ICIST'14*, pages 213–216, April 2014.