A Reliability Benchmark for Actor-Based Server Languages

Aidan Randtoul
aidan@randtoul.co.uk
School of Computing Science, University of Glasgow
Glasgow, United Kingdom

Phil Trinder
Phil.Trinder@Glasgow.ac.uk
School of Computing Science, University of Glasgow
Glasgow, United Kingdom

Abstract

Servers are a key element of current IT infrastructures, and must often deal with large numbers of concurrent requests. Reliability is crucial as any disruption is extremely costly. Some important reliable servers are implemented in actor languages/libraries that provide process isolation and supervision. Reliability benchmarks model fault scenarios to measure the reliability characteristics of systems.

The paper presents the design and implementation of a new reliability benchmark for actor-based server languages: Supervised Communicating Processes (SCP). SCP extends an existing server concurrency benchmark by supervising server actors/processes. We outline Erlang and Scala/Akka SCP implementations, and an associated fault injector.

We compare the reliability characteristics of Erlang and Scala/Akka for server-style computations using SCP in the following four main experiments. (1) Progressive permanent failures, where a percentage of server processes fail permanently. (2) Recovery from different percentages (0% .. 20%) of failures occurring uniformly, randomly, or in bursts, and with a range of supervisor/supervisee ratios. (3) Comparing how the Erlang and Scala/Akka SCPs handle burst, random and uniform failure patterns. (4) Comparing how Erlang and Scala/Akka handle server actor/process faults with different fault patterns and failure rates.

CCS Concepts: • **Computer systems organization → Reliability**; • **Software and its engineering → Distributed programming languages**.

Keywords: Benchmark, Server, Reliability, Fault Tolerance, Distributed System, Erlang

Erlang '22, September 11, 2022, Ljubljana, Slovenia
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9435-2/22/09…$15.00
https://doi.org/10.1145/3546186.3549928

ACM Reference Format:
Aidan Randtoul and Phil Trinder. 2022. A Reliability Benchmark for Actor-Based Server Languages. In *Proceedings of the 21st ACM SIGPLAN International Workshop on Erlang (Erlang '22), September 11, 2022, Ljubljana, Slovenia.* ACM, New York, NY, USA, 12 pages.
https://doi.org/10.1145/3546186.3549928

1 Introduction

Servers are a key technology supporting activities like social networks, online games, chat applications, and many others. Many servers must often deal with large numbers of concurrent requests, and have low overheads when tolerating failures. The programming language used to construct the server has an important role in engineering efficient and reliable server software. So what language technologies should be selected to engineer a new server?

Programming languages are commonly compared using sets of benchmarks. For example, the famous Computer Language Benchmarks Game compares languages for completing computational tasks. For servers, however, concurrency and reliability are more important than compute speed. Comparing concurrent performance is harder, and benchmark suites are less widely used, some include the Barcelona OpenMP Benchmarks [10] and BenchErl [3] for Erlang.

While the Savina benchmarks are designed to compare actor languages/libraries, including both concurrency and parallelism [13], we are not aware of any benchmark that attempts to compare the reliability of actor languages/libraries. Measuring reliability is hard [5], for example, there are multiple failure modes to consider like random failures or bursts of failures. We do so in the restricted context of high throughput servers with supervised server actors/processes.

The paper makes the following research contributions.

1. **The design and implementation of a new language neutral benchmark for reliable actor-based server languages/libraries: Supervised Communicating Processes (SCP).** We believe that SCP is the first benchmark for comparing the reliability of actor-based languages/libraries. SCP extends an existing server benchmark previously used to compare Erlang, Go and Scala/Akka for high throughput servers [34]. It does so by adding a supervision tree over the stateless server actors. We outline Erlang and Scala/ Akka SCP implementations, and an associated deterministic

fault injector that supports four failure patterns: burst, uniform, random and progressive (Section 3).

2. **A systematic comparative analysis of the reliability characteristics of Erlang and Scala/Akka for server-style computations using SCP.** We describe and analyse the performance of the Erlang and Scala/Akka SCP implementations in the following four main experiments. Under test the SCP implementations execute with the sole use of a 16-core server.
(1) Progressive permanent failures, where a percentage of server processes fail permanently. For example, we find that the throughput of the Erlang and Scala/Akka SCPs drops by approximately 5% for every 5% of process pairs killed, 10% for every 10% of process pairs killed, and in general approximately X% for every X% of process pairs killed. In both Erlang and Scala/Akka the drop in throughput deviates from the expected drop on occasion, and the deviations are greater for Scala/Akka (Section 4.1).
(2) Recovery from different percentages (0% .. 20%) of failures occurring uniformly, randomly, or in bursts, and with a range of supervisor/supervisee ratios. The Erlang SCP achieves good throughput with ratios of 1:1 and 1:128 process pairs per supervisor, and Scala/Akka with a 1:1 ratio. For example, if 2.5% of process pairs fail per second Erlang SCP throughput is 5005, 5031 and 5032 Messages/s for burst, uniform and random failure modes; and the throughputs for the Scala/Akka SCP are 4615, 4596 and 4593 Messages/s (Section 4.2).
(3) We compare how the Erlang and Scala/Akka SCPs handle burst, random and uniform failure patterns. For example, we find that with a supervisor/supervisee ratio of 1:64 throughput falls at a similar rate for all patterns, and by around 12% at a 20% failure rate in both Erlang and Scala/Akka. However, with a 1:1 supervisor/supervisee ratio both Erlang and Scala/Akka SCPs tolerate high burst failures better with throughput falling by just 4% at a 20% failure rate (Section 4.3).
(4) Comparing Erlang and Scala/Akka SCP performance demonstrates that the Erlang SCP has a 10% higher throughput in the absence of failures (5085 vs ~4680 Messages/s). Normalising for this difference allows us to directly compare the impact of different fault patterns and failure rates on the Erlang and Scala/Akka SCPs (Section 4.4).

2 Related Work

2.1 Servers & Server Technologies

2.1.1 Importance of Servers. In the last 25 years the size of the internet is increased dramatically and it is estimated that there are now ~4.9 billion users worldwide [17]. This has been driven by the rise of online retail, streaming, and social media sites such as Amazon, Netflix and Facebook. The throughput of servers has been required to grow to meet this burgeoning demand, e.g. WhatsApp reported that their servers had processed 64 billion messages in a single day or 2.6 Billion messages per hour in 2014 [6]. Efficient and effective server implementations are essential to service such high request rates. Servers come in many forms including Web-Servers, Cloud Servers, Real-Time Systems, Blockchain, and many more. Choosing appropriate server hardware and software is crucial, and a key element are the language technologies used to implement the server.

2.1.2 Server Languages & Technologies. There are a range of languages commonly used to implement servers with an array of different characteristics. Table 1 outlines the computation, coordination, reliability, and popularity characteristics of a range of common server languages, extending a table in [34]. The Computation Model is the language paradigm, e.g. Object-Oriented (OO). At a high abstraction level the programmer specifies less information compared with a low abstraction level. Coordination model is the paradigm used to create, communicate and synchronise units of computation like threads or actors. Strong typing is a key mechanism for improving reliability and may be enforced statically or dynamically. The languages support a variety of reliability mechanisms like exceptions, panics and actor supervision. We consider support for distribution to be a reliability characteristic as it enables a server language to tolerate the failure of a host.

Here we propose a reliability benchmark for actor languages, and of the actor languages we choose to study Erlang and Scala/Akka for the following reasons. Both languages have high level computation models; both implement green threads crucial for server performance on a single host; both support distributed execution across multiple hosts which is essential for engineering scalable servers.

2.1.3 Erlang. Erlang is designed for engineering reliable and scalable distributed systems that may need to 'run forever' and/or meet soft real-time constraints [2]. Its computational model is functional and executes on the BEAM Virtual Machine. Erlang's coordination model is actors (called processes). Reliability in Erlang is multi-layered: as in all actor languages each process has a private state, preventing a failed or failing process from corrupting the state of other processes. Erlang avoids type errors by enforcing strong typing, albeit dynamically. Connected VMs (nodes) check liveness with heartbeats and can be monitored from outside Erlang, e.g. by an operating system process. However, the most important way to achieve reliability is *supervision*, which allows a process to monitor the status of a child process and react to any failure, for example by spawning a substitute process to replace a failed process. Supervised processes can in turn supervise other processes, leading to a supervision tree [23]. The rapid engineering of reliable systems is facilitated by the OTP libraries, and these encode supervision behaviour.

Table 1. Server Language Characteristics

	Computation		Coordination		Reliability			Popularity
Language	**Model**	**Abstr. Level**	**Model**	**Abstr. Level**	**Typing**	**Primary Reliability Mechanisms**	**Supports Distribution**	**RedMonk 2022**
Java	Object Oriented	Mid	Explicit	Mid	Strong Static	Exceptions	Yes (Akka)	3
Rust	Procedural	Mid	Explicit	Mid	Strong Static	Panic, Result	Yes (constellation)	19
Elixir	Functional	High	Actors	High	Strong Dynamic	Supervision	Yes	~31
Go	Procedural	Mid	CSP	Mid	Strong Static	Panic	No	16
Scala/Akka	OO & Functional	High	Actors	High	Strong Static	Exceptions, Supervision, Futures	Yes	14
Erlang	Functional	High	Actors	High	Strong Dynamic	Supervision	Yes	34

2.1.4 Scala/Akka.

Scala/Akka. Scala is a modern multi-paradigm language [16], and combined with the Akka libraries is commonly used to engineer servers. Scala provides multiple computation paradigms, supporting both object-oriented and functional paradigms. It executes on Java Virtual Machines (JVMs). For reliability, Scala provides a sophisticated strong static type system that provides features such as algebraic data types, anonymous and higher-order types.

Akka is an open-source toolkit and run-time used for developing reliable concurrent and distributed applications on the JVM [14]. While Akka implements multiple concurrency models, a key element is Erlang-inspired actor-based concurrency. Akka Classic dynamically type-checks messages between actors. More recently Typed Akka [15] statically types messages using Scala typing. The Scala SCP implementation uses Typed Akka.

2.2 Server Benchmarking

Servers are routinely benchmarked to evaluate performance using a range of Benchmark Management Systems (BMSs) for different kinds of servers. For example Wrk [36] and Apachebench [1] are popular BMS for web servers. BMS generate loads on the server and can be parameterised to test different aspects, e.g. different pages in a web server. BMS are commonly concurrent in order to generate high loads. Server benchmarking composes the System under Benchmarking (SUB) with the BMS. The BMS is typically executed on a different host(s) from the SUB hosts to avoid competition for computation, memory, and other resources.

CPT Benchmark. The Concurrent Process Throughput (CPT) benchmark has been proposed to evaluate the scalability of server languages and was used to compare Erlang, Go and Scala/Akka [34]. CPT spawns an array of actor (process) pairs that communicate with each other continuously and records the total number of messages sent every 2 seconds, as depicted in figure 1. Experiments showed that Go had the highest throughput, benefiting from low latency channels and the high number of goroutines that could be spawned. We extend the CPT benchmark in the next section to explore the impact of failures on server throughput.

2.3 Server Reliability

Having a server crash can cause a company significant economic disruption and reduce customer confidence [26]. The ability of a system to deliver a high quality of service when failures are experienced is known as its **dependability** [20]. Two key terms used to describe dependability are reliability and availability. There are many strategies for ensuring a system is fault-tolerant and reliable. A key strategy is failure-oblivious computing which involves allowing the running of the system to continue in the event of failures e.g. try-catch blocks. A key implementation of this strategy is **supervision** which is where a process is 'supervised' by another process which will monitor for faults and step in to resolve them if they occur. Another strategy is recovery-shepherding which is when errors are caught and dealt with before they corrupt the execution flow e.g. Panics in Go [11]. Another widely used strategy is redundancy which is where multiple instances of a process are made and swapped out for each other if one fails. A key implementation of this strategy is process isolation which is where processes within a system are designed to be stateless and independent of each other in order to make swapping in new instances quick and easy. In order to engineer a system to be reliable you must know the kinds of failures you may encounter and so many studies have been completed on failures [19, 24].

Failure Types. It is well established from studies that hardware failures are common in large-scale systems as the sheer number of components in use can make even small failure rates in components, problematic [24, 28, 30, 35]. In many

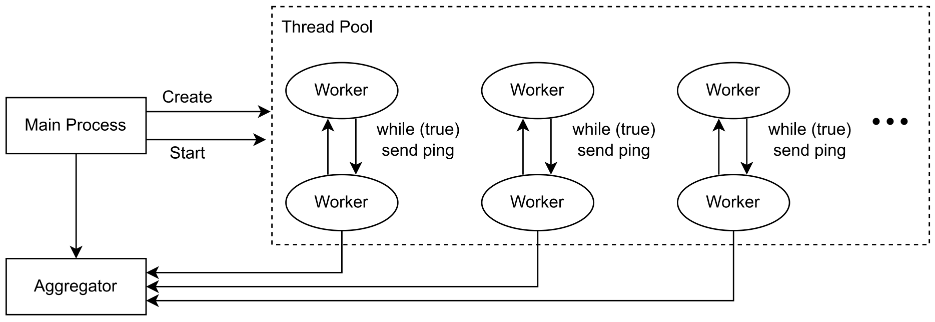


Figure 1. Concurrent Process Throughput (CPT) Benchmark Architecture [34]

cases, these failures have also not occurred uniformly however they often fall into one of three categories: infant mortality, random failures and wear-out failures [18]. Failures within network & communication hardware can also create issues, especially in servers where communication between systems is vital [12]. The second most common type of failure is software failures [30]. The field of distributed computing further complicates bug finding as it introduces what is known as non-deterministic bugs [21]. These bugs are very common in concurrent systems and can be particularly troublesome as they often result in incorrect outputs and even fatal errors [29]. Server systems must also consider a failure type known as overload which is where a server is overloaded with requests and suffers a degradation of performance [31, 33].

2.4 Reliability Benchmarking

Reliability or dependability benchmarking is notoriously resource and time-intensive [5]. Hence SCP is designed to be lightweight to have a low runtime to allow repeated execution in a short space of time.

A key parameter is the fault load which determines the types of failures that occur, how many failures occur, and when the failures occur. There have been many proposed methods for designing fault loads that are lightweight, representative and portable [8, 9]. Fault loads may be realistic, i.e. based on some real-world data, or manufactured.

Failure Patterns. The fault loads used in this study are designed to represent real-world scenarios and are derived from the analysis of the most common failure types described in section 2.3. Four different fault loads are used in this study and they are: **burst**, **random**, **uniform**, and **progressive**. In the first three patterns the failed server processes are recovered, and so induce only a temporary reduction in server throughput during recovery. In the last pattern, progressive, the failed processes are not recovered, and server throughput drops permanently.

The **uniform** failure pattern represents scenarios where failures are spread evenly across system components and over time. This pattern is seen, for example, in web servers where requests periodically fail. One study found the median failure rate was ~1.5% [25]. In SCP the uniform pattern is implemented by terminating single actor/process pairs at a specified frequency.

The **burst** failure pattern represents scenarios where a sequence of failures occur close in time. This pattern is seen, for example, in the hardware and network failures in data centers, as discussed in Section 2.3. Both hardware and network failures will terminate a large number of server processes almost simultaneously [12]. In SCP the burst pattern is implemented by terminating 25 actors/processes pairs at a specified frequency. Terminating 25 actors/process pairs in each burst is based on [29] however the bursts have been increased to reduce the duration of the benchmark.

The **random** failure pattern is more realistic, representing the common scenario that combines uniform and burst failures. This pattern is seen, for example, in web servers where in addition to uniform request failures, 70% of servers experience burst failures with a failure rate of at least 5-10% [25]. In SCP the random pattern is implemented by combining uniform failures and burst failures, and each burst terminates up to 10 process pairs.

The **progressive** failure pattern represents scenarios where a burst of failures are not recoverable. This pattern is seen, for example, in servers with failing storage, either hard drives [35] or RAID controllers [27]. In SCP the progressive pattern is implemented similarly to the burst pattern, except that a specific number of server process pairs can be terminated, and the terminated pairs are not restarted.

3 SCP: A Benchmark for Comparing Reliable Actor-Based Server Languages

We propose Supervised Communicating Processes (SCP) as a benchmark for comparing server languages and frameworks that provide fault tolerance using supervised actors. Rather than a random Chaos Monkey [4, 7], SCP provides a deterministic fault injector that makes it possible to explore the impacts on throughput of different fault rates and patterns.

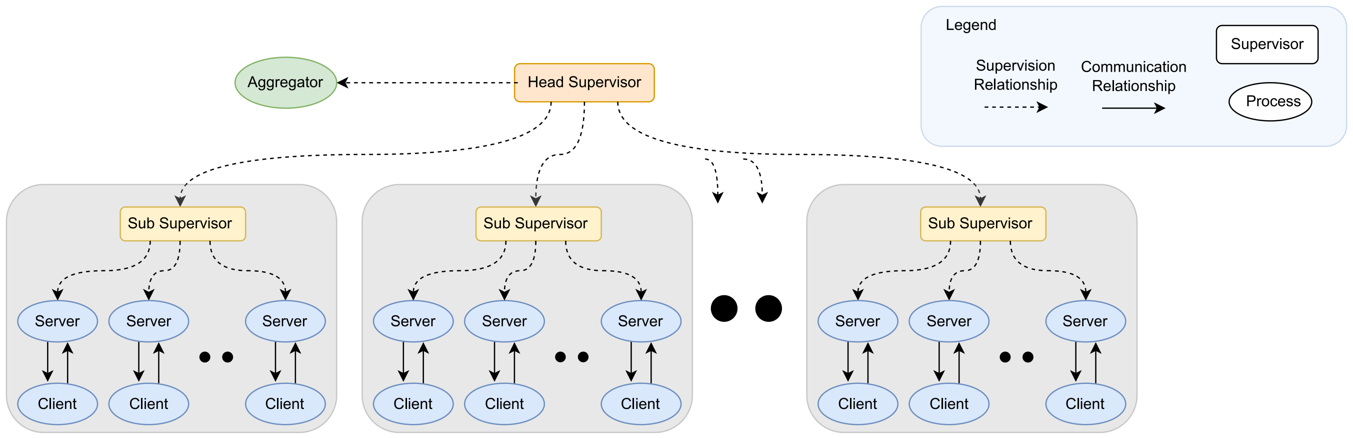


Figure 2. Supervised Communicating Processes (SCP) Benchmark SUB Architecture Diagram

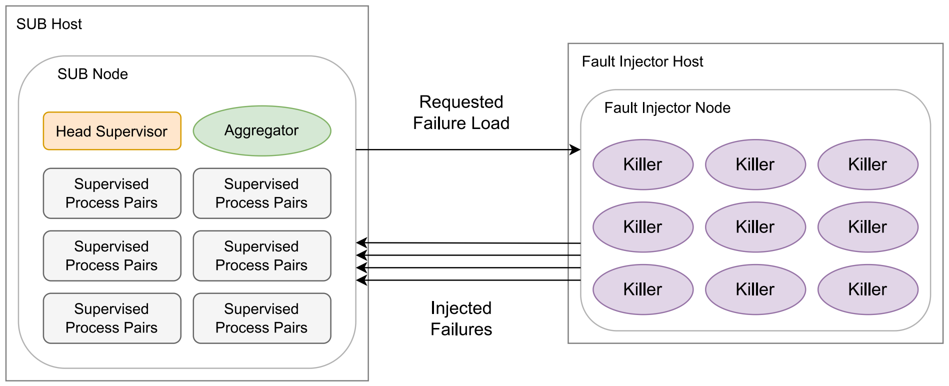


Figure 3. SCP Benchmark in Operation

3.1 SCP Design

SCP benchmark is designed as a simple and effective way to analyse the reliability of actor-based server languages. The key reliability strategies exploited are process isolation and actor supervision, as explored in sections 4.1 & 4.2 respectively. SCP is designed to be implemented idiomatically in each language.

SCP extends the CPT benchmark with a supervision tree that recovers from the failure of "server actors", i.e. the pairs of communicating actors. In addition, a communication delay is added between each message exchange between the pairs. The delay represents a typical server processing time [22] and prevents the server from hitting a throughput ceiling, as in [34]. Throughput is also measured slightly differently, being measured every second with the average computed over a 60s execution. The simplicity of CPT means that the server rapidly reaches a steady state.

SCP has a two-level supervision tree as depicted in Figure 2. The 'head' supervisor supervises the aggregator process along with an array of 'sub' supervisors. Each 'sub' supervisor spawns and supervises a set of actor/process pairs. The ratio of supervisors to supervisees is customisable and is a metric explored in the experiments.

3.2 SCP SUB Implementations

The Erlang and Scala/Akka SCP SUB implementations are available at https://github.com/AidanRandtoul/SCP.

3.2.1 Erlang.

The Erlang SCP launches the fault injector on one node before launching the SCP SUB on a separate node. When starting the SUB the user specifies the desired parameters for the benchmarking e.g. supervisor to supervisee ratio, and the fault injection rate. The Erlang SCP SUB has the following three main components.

Process Specification functions define the behaviour for the aggregator, each process pair, along with spawning functions for each. In each pair, there is a "main worker" process that initiates communication and logs each completed communication with the aggregator. Server-side processing is simulated with a call to `timer:sleep(200)` that suspends the process for 200ms[1]. The aggregator counts the number of messages it has received and at 1s intervals records the throughput in messages/s.

Benchmark Control coordinates the benchmark execution and spawns all SCP components. Once the SUB and the Fault Injector are started a message is sent to the fault injector with the requested fault load and a list of process-ids of the supervisors. When benchmarking is complete the injector is sent a message to stop, the results are logged, and SCP SUB is terminated.

Process Supervision specifies the SCP supervisors using OTP. In SCP the child processes are temporary (never restarted) in the Progressive Permanent Failures experiments, and transient (only restart if crashed but not if stopped) in the Failures with Recovery experiments. The restart strategy, intensity and period are set by creating a map which represents the `sup_flags` and contains the desired values for each. All other values in the `sup_flags` and `child_specs` are left default.

3.2.2 Scala/Akka.

The Scala/Akka SCP is implemented in an object-oriented style using classes and methods. To launch you must first run the fault injector executable on the fault injector host, and then run the SUB executable on the SUB host. The Scala/Akka SCP SUB has the following main components.

The **SUB** defines a set of classes and a process to connect to the fault injector. The type system requires that the types of all potential messages are defined. For example, the main worker actor can receive 3 message types: **StartMessage**, **PongMessage** from the other worker, and **Stop** from the fault injector. The aggregator behaviour is as in the Erlang SCP. As `Thread.sleep()` is thread blocking in Scala the 200ms message delay is implemented using the Akka scheduler that will send a message after a given delay. An infelicity is that scheduled messages are sent even if the process that requested it has crashed. These are handled with a `PostStop` clause that runs when an actor crashes/stops and cancels any scheduled messages. A second fail-safe tracks message timestamps and actors disregard messages sent before they were created i.e. messages from an actor that is now dead.

Supervision in Scala/Akka is implemented very simply by enclosing the `context.spawn` function in a `behaviours.supervise` call. Using the `.onFailure` parameter, the type of failure and required action are specified. In the Scala/Akka SCP, the type of failure encountered is an Arithmetic Exception and either a restart or stop is required. The available options for behaviour are to: Stop the process, Restart the process, Escalate the failure or Ignore the failure. Using the `.onFailure` clause settings such as restart intensity and period can also be set however as with the Erlang SCP most settings were left as default.

[1]Using `sleep()` inflates throughput as the sleeping process is descheduled and, unlike a real server process, makes no demands on the host cores. However, the key metrics for SCP are how fault recovery impacts throughput, rather than absolute throughput achieved.

3.3 Fault Injector Design

To evaluate reliability, faults are induced into the SUB as outlined in section 3.1, and for SCP this is done with a deterministic parametric fault injector. To remove the performance impact of the fault injector, it runs on a separate machine/host. The fault injector spawns a set of 'killer' processes which at a set interval and in a set pattern kill processes in the SUB, as depicted in Figure 3. The fault injector implements burst, uniform, random and progressive failure patterns as outlined in section 4.2. The burst pattern kills a set of actors/processes in quick succession. The uniform pattern kills actors/processes uniformly over time with a set interval between kills. The random pattern combines both burst and uniform patterns and is the most realistic. The progressive pattern kills a set of actor/processes at 5s intervals.

3.4 Fault Injector Implementations

The Erlang and Scala/Akka fault injectors are available at https://github.com/AidanRandtoul/SCP.

3.4.1 Erlang.

The Erlang fault injector spawns an array of 'killer' processes which kill processes at a set frequency and pattern depending on the specified failure pattern. Each killer is given a copy of the supervisor list from the benchmark control with a unique ordering of the list to spread the killers and by extension the kills over all the supervisors in the system. The killers are spawned evenly over the interval between kills so that increasing the failure load results in an increased frequency of kills. Two helper functions are used by the killers. The **burstKill** function kills a given number of processes simultaneously. The **randKill** function kills a given number of processes with a delay between kills to implement a uniform failure pattern. The code snippet below illustrates the Erlang **killProcess** function.

```
killProcess(Sup) ->
    PList = supervisor:which_children(
        Sup),
    {_, Target, _, _} = lists:nth(rand:
        uniform(length(PList)), PList),
    Status = rpc:call(SubNode, erlang,
        is_process_alive, [Target]),

    if
        Status ->
            exit(Target, kill);
        true ->
            killProcess(Sup)
    end.
```

3.4.2 Scala/Akka.

The Scala/Akka Fault Injector implementation is similar to the Erlang injector. The Akka Scheduler provides the correct intervals between kills as Scala lacks a non-thread blocking sleep function. The process of killing a process pair is also slightly different due to the way that ActorRefs (the equivalent of pid in Erlang) work. When a supervisor restarts an Actor, it simply swaps out the old instance of the actor with a new one, maintaining the existing ActorRef. This means that a new list of ActorRefs is not needed after each kill as in the Erlang Injector. The code snippet below illustrates the Scala/Akka **killProcess** function used by the injector.

```
private def killProcess(): Unit = {
    var target = Random.nextInt(supLen)
    serverList(start)(target) ! Server.
        Stop(0)
    start += 1
    if (start => nSups) {start -= nSups}
}
```

3.5 Platform and Methodology

The experiments are all executed on two nodes of the University of Glasgow GPG Cluster. The hardware of each of these nodes is: a pair of Intel Xeon E5-2640v2 8 core (16 thread) processors, 64GB RAM, and Scientific Linux 6. All results were conducted on cold starts i.e. the Erlang VM and Scala VM (JVM) were restarted after each run.

Table 2. SCP Benchmark Variables

Experimental Parameters	Values
Supervisor to Supervisee Ratio	1:1-1:1024
Fault Injection Rate	0-20% of messages/s
Failure Pattern	Burst, Uniform, Random, Progressive
Failure Recovery	True or False
Burst Size (Burst Pattern)	25 process pairs
Burst Size (Random Pattern)	1-10 process pairs

4 Some SCP Reliability Experiments

4.1 Progressive Permanent Failures

The Progressive Permanent Failures (PPF) experiments investigate the effectiveness of actor/process isolation in Erlang and in Scala/Akka. That is, how does server throughput change as server actors/processes are killed? The fault injector repeatedly kills a set of SCP SUB server actors/processes until there are no processes left to kill, and throughput reaches 0 messages/s. The killing bursts occur at 5s intervals to reduce the benchmarking time, so failures occur significantly faster than for most real servers. The consistent killing behaviour makes the results more understandable and easier to analyse, and we use more realistic failure patterns in later experiments.

There are 1000 process/actor pairs and the failure rate is set at 10%, 5% and 2.5% of processes corresponding to 100, 50 and 25 process pairs failing in each burst respectively. Figure 4 Plots throughput against time for the Erlang and Scala/Akka SCPs with 5% failure bursts.

Figure 4 shows that the throughput of the Erlang and Scala/Akka SCPs drops by approximately 5% for every 5% of process pairs killed. In Erlang the drop in throughput deviates from the expected drop at 40s and 90s. The drop at 40s is significant with throughput dropping 185 M/s below the expected value. We speculate that the Erlang deviations are due to some background process, perhaps garbage collection. In Scala/Akka the deviations from expected throughput occur at higher loads, like 5-10s and 15-20s. The deviations reduce in magnitude and frequency as the load reduces. In addition to background processes, the Scala/Akka deviations may also be due to message latency fluctuating significantly at high loads [32].

The throughput graphs showing the killing of 10% of process pairs, and of 2.5% of process pairs are available in the GitHub repository. They show a similar pattern, e.g. a 10% drop in throughput for every 10% of process pairs killed. The deviations from the expected fall in throughput are again present and are less and more prominent respectively.

Variability is measured in 10 executions of these experiments, and the error bars in Figure 4 plot the range of values recorded when variance is > 5%. The Erlang SCP exhibits little variation, with a maximum variation of 7.12% when 2.5% of process pairs fail. The Scala/Akka SCP exhibits significantly more variation with a maximum variation of 26.2% when 10% of process pairs fail. This is again likely due to the fluctuating message latency of Scala/Akka [32]. The Erlang SCP has less variability with large bursts, whereas the Scala/Akka SCP has less variability with smaller bursts.

As throughput for both Erlang and Scala/Akka SCPs falls only in proportion to the percentage of processes that fail we **conclude** that both Erlang and Scala/Akka provide effective process isolation.

4.2 Burst, Random & Uniform Failures with Recovery

In these experiments the Erlang and Scala/Akka SCP SUBs recover after the failure of server actors/processes, exploiting their supervision structures. We consider three common failure patterns: burst, uniform and random from Section 2.4. We vary the number of supervisors and hence the ratio of supervisors to supervisees. To facilitate uniform partitioning the SCP SUBs have 1024 actor/process pairs and between 1-1024 supervisors. This allows us to explore the impact of the supervisor/supervisee ratio in each language, and identify an appropriate range of ratios.

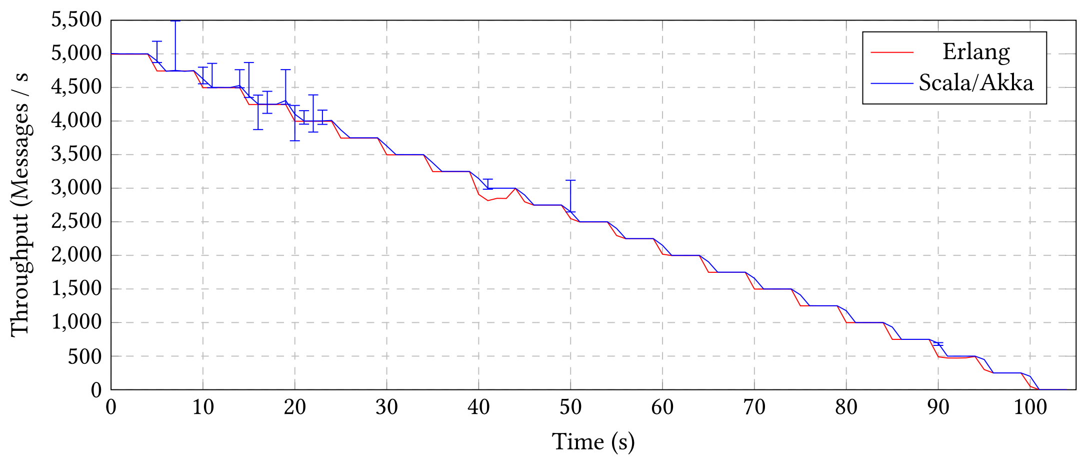
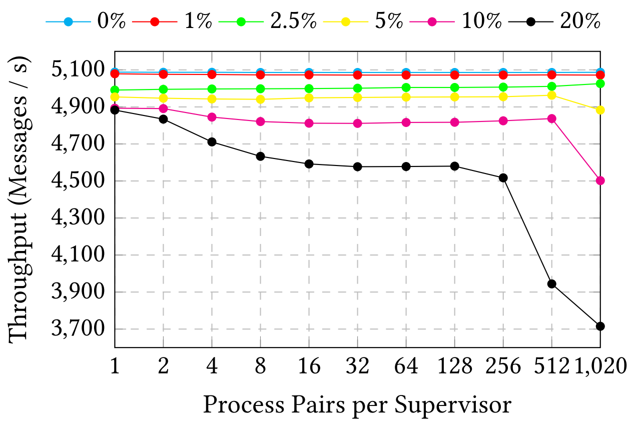
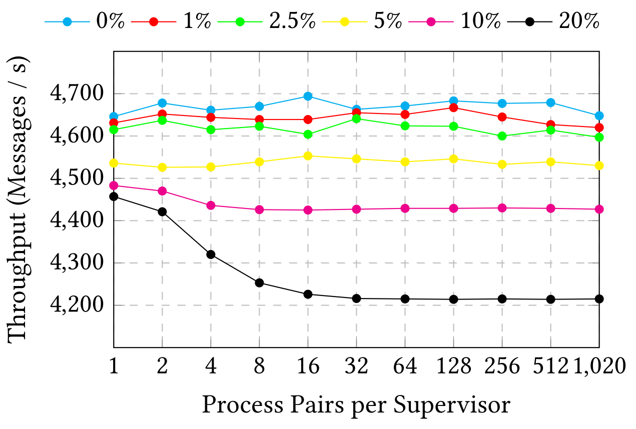


Figure 4. Progressive Permanent Failure of 5% Process Pairs every 5s

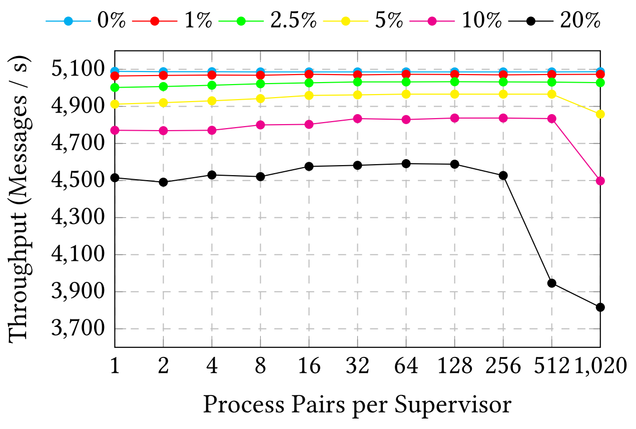


(a) Erlang

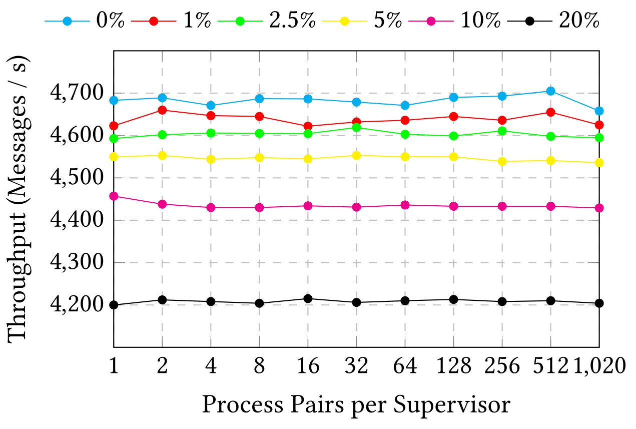


(b) Scala/Akka

Figure 5. Burst Failure Pattern



(a) Erlang



(b) Scala/Akka

Figure 6. Random Failure Pattern

Figure 5a plots the throughput of the Erlang SCP SUB with different supervisor/supervisee ratios, and with different percentages of server actor/processes killed in bursts. In the experiment, the SCP and the fault injector are run for 60s. The failure rates discussed are in relation to the messages sent between process pairs and so a failure frequency of 1% means that 1% of the message sequences experience a failure every second. With failure rates below 5% throughput is maintained well compared with the no failures baseline: it falls by no more than 4% and is lowest at 4883 M/s with 1024 process pairs per supervisor (PPS). However with higher failure rates and a higher number of PPS throughput falls more significantly, e.g. by 10% to ~4580 M/s with 20% failures with up to 128 PPS. At high failure rates having a large supervisor/supervisee ratio significantly reduces throughput, e.g. with 20% failures and a single supervisor it falls to just 3715 M/s, a reduction of ~27%. Conversely at failure rates of 10% or more throughput is better maintained by having a supervisor for every server process.

Figure 5b shows similar patterns for the Scala/Akka SCP SUB tolerating *burst* failures. Throughput without failures is more variable, and lower than in Erlang at 4650-4680 vs 5085 Messages/s. With failure rates below 5% throughput is maintained well compared with the no failures baseline: it falls by less than 4% to ~4540 M/s. At higher failure rates (10% or more) Scala requires more supervisors than Erlang to maintain throughput, i.e. no more than two PPS.

Figures 6a and 6b show the throughputs with server actor/processes killed at *random*. The Erlang SCP SUB maintains throughput well even at high failure rates providing there are sufficient supervisors. The Scala/Akka SCP SUB throughput falls steadily with increasing failure rates and is less sensitive to the number of supervisors. This may be due to the role of the Akka Dispatcher in organising the restarting of actors and assigning the tasks to processor threads. The throughput graphs for *uniform* failures again show similar patterns, following the random pattern results in Erlang and the burst pattern results in Scala/Akka. They are available for viewing in the GitHub Repo.

4.3 Failure Pattern Comparison

These experiments explore how the Erlang and Scala/Akka SCP SUBs handle the different failure patterns: burst, random and uniform. The previous section suggested that supervisor/supervisee ratios of 1:1 and 1:64 are sane choices, and are selected here.

Figure 7a plots an Erlang SCP SUB throughput curve for each failure pattern against failure percentage with a supervisor/supervisee ratio of 1:64. Throughput falls at a similar rate for all patterns, and by around 12% at a 20% failure rate. Conversely, Figure 7b reveals that with a 1:1 supervisor/supervisee ratio the Erlang SCP SUB tolerates high burst failures better than uniform or random failures, with throughput falling by just 4% at a 20% failure rate.

With a supervisor/supervisee ratio of 1:64 Figure 8a shows that the Scala/Akka SCP SUB throughput also falls at a similar rate for all patterns, and also by around 11%. However, Figure 8b reveals that with a 1:1 supervisor/supervisee ratio the Scala/Akka SCP SUB tolerates high uniform and burst failures better than random failures. That is throughput for uniform and burst failures falls by around 4% compared with 11% for random failures and at a 20% failure rate.

4.4 Comparing Erlang & Scala/Akka for Server Faults

These experiments compare Erlang and Scala/Akka for handling server actor/process faults using SCP SUB. We do so by selecting sane supervisor/supervisor ratios (1:1 and 1:64) as determined in Section 4.2. To account for the different throughputs of the Erlang and Scala/Akka SCP SUBs we compute the percentage of the throughput in the absence of failure achieved by the SUBs under different failure patterns and percentages.

Figure 9a plots the reduction in throughput in the SUBs for different percentages of burst failures. Erlang and Scala/Akka have very similar behaviours, and throughput falls less with a 1:1 supervisor/supervisee ratio. These results corroborate the results in Section 4.3 suggesting that the additional supervisors allow both SCPs to handle bursts of failures faster.

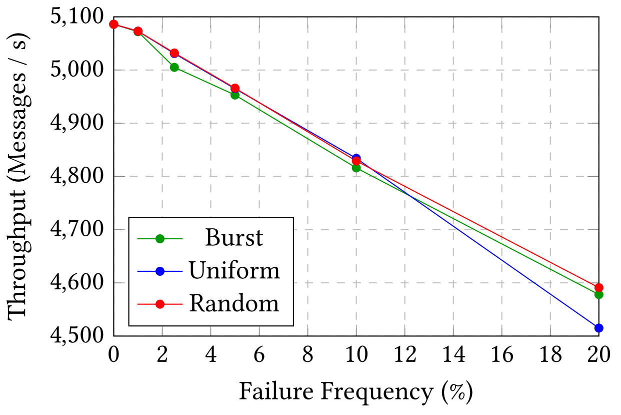
Figure 9c plots the reduction in throughput in the SUBs for different percentages of random failures, and the curves for the Erlang and Scala/Akka SCP SUBs are very similar. Figure 9b plots the reduction in throughput in the SUBs for different percentages of uniform failures. While the curves for the Erlang and Scala/Akka SCP SUBs are similar with a 1:64 supervisor/supervisee ratio, the Scala/Akka throughput falls far less than the Erlang throughput with a 1:1 supervisor/supervisee ratio.

5 Conclusion

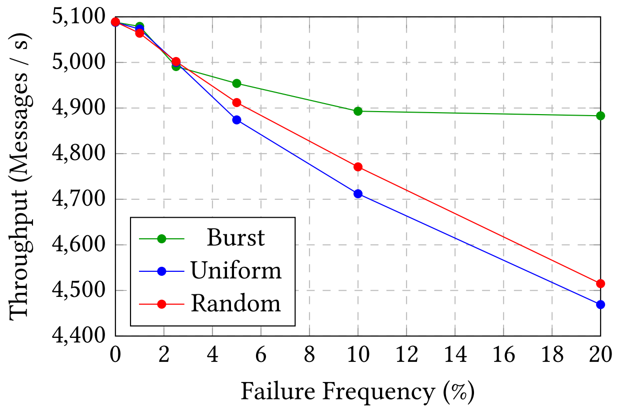
This paper specifies a new reliability benchmark for reliable server languages (SCP in Figure 2). We believe this to be the first such reliability benchmark for actor server languages/libraries. We outline Erlang and Scala/Akka SCP implementations, and an associated fault injector (Section 3).

The Erlang and Scala SCPs behave broadly as expected, and the **take-home messages** are as follows.

(1) When a percentage of server processes fail permanently the throughput of the Erlang and Scala/Akka SCPs drops in proportion to the reduction in server processes, e.g. Figure 4. In both Erlang and Scala/Akka the drop in throughput deviates from the expected drop on occasion. In Scala/Akka the deviations are greater and more common at high throughputs (Section 4.1).

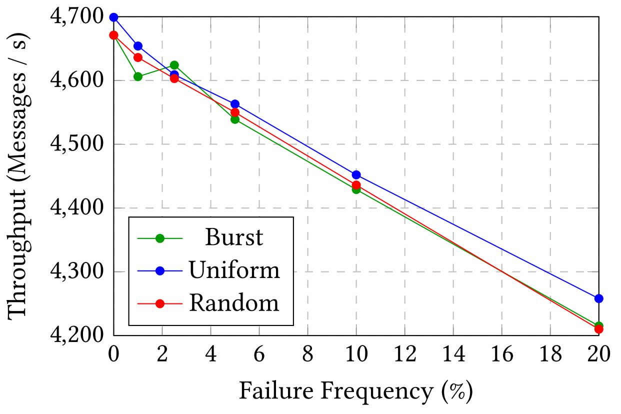


(a) 64 Process Pairs per Supervisor

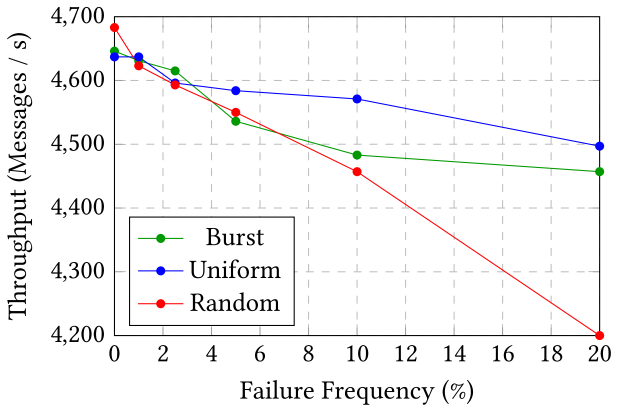


(b) 1 Process Pair per Supervisor

Figure 7. Erlang Failure Pattern Comparison



(a) 64 Process Pairs per Supervisor



(b) 1 Process Pair per Supervisor

Figure 8. Scala/Akka Failure Pattern Comparison

The remaining observations are for systems where failed actors/processes are recovered by their supervisor.

(2) Selecting a suitable supervisor/supervisee ratio is critical for tolerating higher failure rates. Where the Erlang SCP tolerates failures well up to a 1:128 ratio, the Scala/Akka SCP often requires a 1:1 ratio, e.g. Figures 5a and 5b (Section 4.2).

(3) While failures reduce SCP throughput, the reductions are small, even at high failure rates. Higher failure rates cause greater drops in throughput, although with a suitable supervisor/supervisee ratio the throughput drop is less than 12% even at high (20%) failure rates, e.g. Figures 5a and 6a (Section 4.2).

(4) The Erlang and Scala/Akka SCPs survive the failure of processes in different patterns: uniform, burst and random. As before reductions in throughput are small given a suitable supervisor/supervisee ratio, e.g. Figures 6a and 6b (Section 4.3).

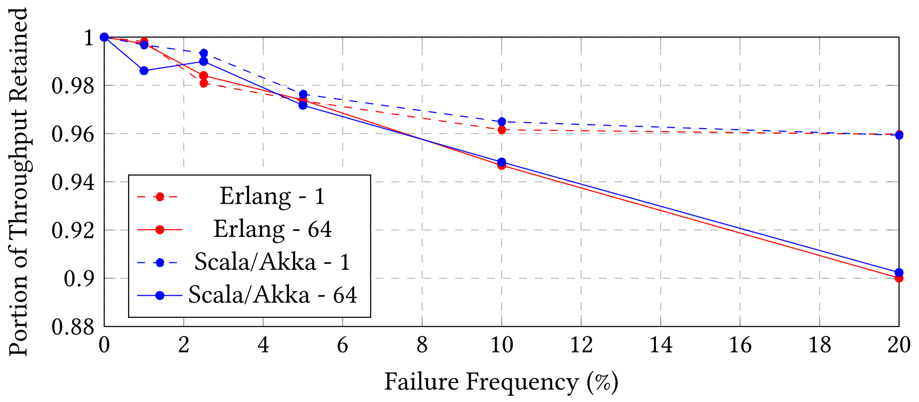
(5) At high failure rates throughput drops least with a 1:1 supervisor/supervisee ratio (Figures 7b and 7a). The Erlang SCP tolerates burst failures better than uniform or random (Figure 7b). The Scala/Akka SCP tolerates burst and uniform failures better than random (Figure 8b) (Section 4.3).

(6) In the absence of failures the Erlang SCP has a 10% higher throughput (5085 vs ~4680 Messages/s) than the Scala/Akka SSCP. Normalising for this difference allows direct comparison the Erlang and Scala/Akka SCPs. Their performance with 1:1 and 1:64 supervisor/supervisee ratios, and burst and random failures, are very similar (Figures 9a and 9c). The one divergence is that with a 1:1 ratio and random failures the Scala/Akka SCP throughput falls far less than the Erlang SCP throughput (Figure 9b). This divergence warrants further investigation (Section 4.4).

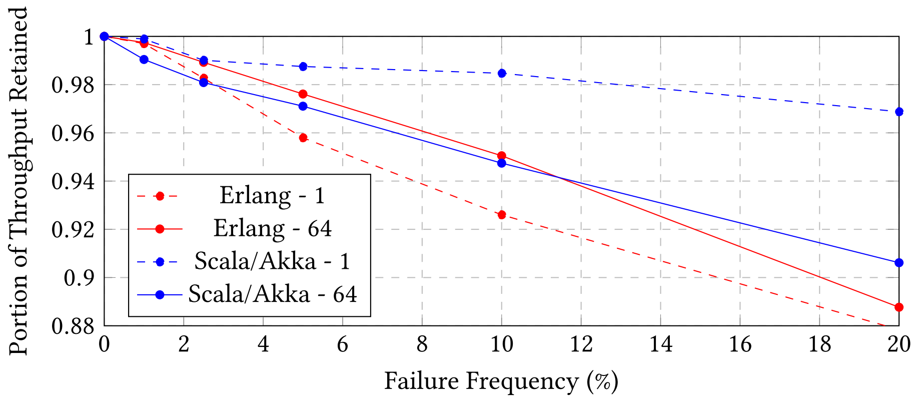
A **limitation** of SCP is that the server actors are stateless, and most real server actors are stateful, e.g. in a chat server they record the chat participants. During recovery the state is recovered from some store, either in-memory or persistent. However, designing a benchmark to compare the reliability of supervised stateful server actors seems very challenging:

different actor languages & frameworks offer different stores, and often support a range of stores, e.g. Erlang provides ETS, DETS, Mnesia and more.

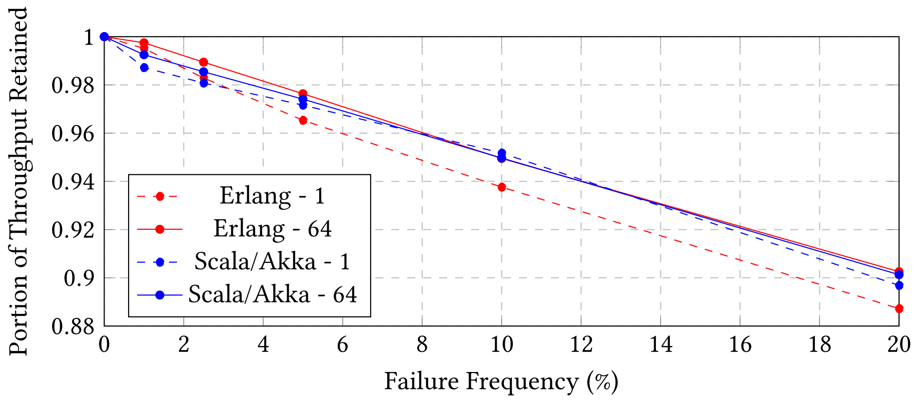
Further Work could use SCP to compare other actor server languages like Elixir and Java/Akka. SCP could also be used to explore different recovery strategies in actor languages. SCP could be elaborated in various ways, e.g. the fault injector could be elaborated to allow the exploration of different types of failures. In addition SCP currently explores the failure of fairly small sets of actors within a VM, and this could be broadened to consider the failure and recovery of large numbers of actors when an entire VM fails and must be restarted.



(a) Burst Failure Comparison



(b) Uniform Failure Comparison



(c) Random Failure Comparison

Figure 9. Erlang vs Scala/Akka for Server Faults

References

[1] Apache. 2022. *Apache HTTP Server Benchmarking Tool.* https://httpd.apache.org/docs/2.4/programs/ab.html

[2] Joe Armstrong. 2007. A History of Erlang *(HOPL III)*. Association for Computing Machinery, New York, NY, USA, 6–1–6–26. https://doi.org/10.1145/1238844.1238850

[3] Stavros Aronis, Nikolaos Papaspyrou, Katerina Roukounaki, Konstantinos Sagonas, Yiannis Tsiouris, and Ioannis E. Venetis. 2012. A Scalability Benchmark Suite for Erlang/OTP *(Erlang '12)*. New York, NY, USA, 33–42. https://doi.org/10.1145/2364489.2364495

[4] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. 2016. Chaos Engineering. *IEEE Software* 33, 3 (2016), 35–41. https://doi.org/10.1109/MS.2016.60

[5] Aaron B. Brown, Jonathan Traupman, Pete Broadwell, and David A. Patterson. 2002. Practical Issues in Dependability Benchmarking. In *EASY '02*. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.66.6202

[6] Ryan Bushey. 2014. WhatsApp Delivered A Mind-Melting 64 Billion Messages In One Day. https://www.businessinsider.com/whatsapp-64-billion-messages-24-hours-2014-4?r=US&IR=T

[7] Michael Alan Chang, Bredan Tschaen, Theophilus Benson, and Laurent Vanbever. 2015. Chaos Monkey: Increasing SDN Reliability through Systematic Network Destruction. In *SIGCOMM'15*. New York, NY, USA, 371–372. https://doi.org/10.1145/2785956.2790038

[8] Pedro Costa, João Gabriel Silva, and Henrique Madeira. 2009. Dependability Benchmarking Using Software Faults: How to Create Practical and Representative Faultloads. In *PRDC'09*. IEEE Computer Society, 289–294. https://doi.org/10.1109/PRDC.2009.52

[9] João Durães and Henrique Madeira. 2004. Generic Faultloads Based on Software Faults for Dependability Benchmarking. In *DSN'04*. IEEE Computer Society, 285–294. https://doi.org/10.1109/DSN.2004.1311898

[10] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. 2009. Barcelona openmp tasks suite. In *ICPP'09*. IEEE, 124–131. https://doi.org/doi:10.1109/ICPP.2009.64

[11] Andrew Gerrand. 2010. Defer, Panic, and Recover. https://go.dev/blog/defer-panic-and-recover

[12] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications *(SIGCOMM '11)*. Association for Computing Machinery, New York, NY, USA, 350–361. https://doi.org/10.1145/2018436.2018477

[13] Shams M. Imam and Vivek Sarkar. 2014. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries *(AGERE '14)*. New York, NY, USA, 67–80. https://doi.org/10.1145/2687357.2687368

[14] Lightbend Inc. 2021. *Akka Classic Documentation.* https://akka.io/docs/ [Online: October 25, 2021].

[15] Lightbend Inc. 2021. *Akka Typed Documentation.* https://doc.akka.io/docs/akka/2.5/typed/index.html [Online: October 25, 2021].

[16] Lightbend Inc. 2021. *Scala Documentation.* https://docs.scala-lang.org/ [Online: October 25, 2021].

[17] Joseph Johnson. 2021. Internet usage worldwide. https://www.statista.com/topics/1145/internet-usage-worldwide/

[18] G.A. Klutke, P.C. Kiessler, and M.A. Wortman. 2003. A critical look at the bathtub curve. *IEEE Transactions on Reliability* 52, 1 (2003), 125–129. https://doi.org/10.1109/TR.2002.804492

[19] George Kola, Tevfik Kosar, and Miron Livny. 2005. Faults in Large Distributed Systems and What We Can Do About Them, In Euro-Par 2005. *Lecture Notes in Computer Science* 3648, 442–453. https://doi.org/10.1007/11549468_51

[20] Jean-Claude Laprie. 1985. Dependable computing and fault-tolerance. *Digest of Papers FTCS-15* 10, 2 (1985), 124. https://doi.org/10.1109/FTCSH.1995.532603

[21] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems *(ASPLOS '16)*. 517–530. https://doi.org/10.1145/2872362.2872374

[22] Littledata. 2022. What is the Average Server Response Time? https://www.littledata.io/average/server-response-time [Online: January 2, 2022].

[23] Vamsi Mokari. 2020. Erlang "Let it Crash" Approach to Building Reliable Services. https://medium.com/@vamsimokari/erlang-let-it-crash-philosophy-53486d2a6da

[24] David Oppenheimer, Archana Ganapathi, and David A. Patterson. 2003. Why Do Internet Services Fail, and What Can Be Done about It? *(USITS'03)*. USA, 1. https://dl.acm.org/doi/10.5555/1251460.1251461

[25] Venkata N. Padmanabhan, Sriram Ramabhadran, Sharad Agarwal, and Jitendra Padhye. 2006. A Study of End-to-End Web Access Failures *(CoNEXT '06)*. New York, NY, USA, Article 15, 13 pages. https://doi.org/10.1145/1368436.1368457

[26] David A. Patterson. 2002. A Simple Way to Estimate the Cost of Downtime *(LISA '02)*. USA, 185–188. https://dl.acm.org/doi/10.5555/1050517.1050538

[27] David A. Patterson, Garth Gibson, and Randy H. Katz. 1988. A Case for Redundant Arrays of Inexpensive Disks (RAID) *(SIGMOD '88)*. New York, NY, USA, 109–116. https://doi.org/10.1145/50202.50214

[28] Ramendra K. Sahoo, Anand Sivasubramaniam, Mark S. Squillante, and Yanyong Zhang. 2004. Failure Data Analysis of a Large-Scale Heterogeneous Server Environment. In *DSN'04*. IEEE Computer Society, 772. https://doi.org/10.1109/DSN.2004.1311948

[29] Swarup Kumar Sahoo, John Criswell, and Vikram S. Adve. 2010. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *ICSE'10*. ACM, 485–494. https://doi.org/10.1145/1806799.1806870

[30] Bianca Schroeder and Garth A. Gibson. 2010. A Large-Scale Study of Failures in High-Performance Computing Systems. *IEEE Transactions on Dependable and Secure Computing* 7, 4 (2010), 337–350. https://doi.org/10.1109/TDSC.2009.4

[31] Bianca Schroeder and Mor Harchol-Balter. 2006. Web Servers under Overload: How Scheduling Can Help. *ACM Trans. Internet Technol.* 6, 1 (2 2006), 20–52. https://doi.org/10.1145/1125274.1125276

[32] Jan-Philipp Stauffert, Florian Niebling, and Marc Erich Latoschik. 2016. Reducing application-stage latencies of interprocess communication techniques for real-time interactive systems. In *2016 IEEE Virtual Reality (VR)*. 287–288. https://doi.org/10.1109/VR.2016.7504766

[33] Nesime Tatbul and Stanley B. Zdonik. 2006. Dealing with Overload in Distributed Stream Processing Systems. In *ICDE'06*. IEEE Computer Society, 24. https://doi.org/10.1109/ICDEW.2006.45

[34] Ivan Valkov, Natalia Chechina, and Phil Trinder. 2018. Comparing languages for engineering server software: erlang, go, and scala with akka. In *SAC'18*. ACM, 218–225. https://doi.org/10.1145/3167132.3167144

[35] Guosai Wang, Lifei Zhang, and Wei Xu. 2017. What Can We Learn from Four Years of Data Center Hardware Failures?. In *DSN'17*. IEEE Computer Society, 25–36. https://doi.org/10.1109/DSN.2017.26

[36] Wg. 2021. A Modern HTTP Bencharking Tool. https://github.com/wg/wrk