# Investigating the Scalability Limits of Distributed Erlang

Amir Ghaffari, Natalia Chechina, Phil Trinder
Department of Computer Science,
School of Mathematical and Computer Sciences,
Heriot-Watt University,
Edinburgh, EH14 4AS, UK

June 2013

## 1 Introduction

The trend toward horizontally scalable architectures [1](e.g. clusters, grids, and clouds) will continue because they offer more available and scalable hardware platform in a cost-effective way. These scalable infrastructure typically consist of loosely connected commodity servers in which node or connection failures are common.To take full advantage of horizontal scaling, the need for reliable scalable programming paradigms is obvious.

In the RELEASE project [2]we aim to scale distributed Erlang for building reliable software on massively parallel and distributed machines.

Recently distributed Erlang has become a popular platform to develop large-scale distributed applications, e.g. *Facebook chat backend*, *T-Mobile advanced call control services*, *Ericsson AXD301 ATM switch* [3], and *Riak DBMS* [4]. This popularity is due to a combination of factors, including data immutability, share-nothing concurrency, asynchronous message passing based on actor model, and process's location transparency [5].

But this doesn't mean that there is no bottleneck in distributed Erlang. Experience with distributed Erlang shows that some distributed applications (e.g. Riak [6]) do not scale up as expected.

To investigate the scalability limits of distributed Erlang we have designed and implemented a benchmarking suite for distributed Erlang. *DEbench* ("Distribute Erlang bench") is an Erlang application which measures the throughput and latency of distributed Erlang commands on a cluster of Erlang nodes.

# 2 How Does the Benchmark Work?

## 2.1 Platform

The benchmark is carried out on the Kalkyl cluster [7]. The Kalkyl cluster consists of 348 nodes with 2784 64-bit processor cores which are connected via 4:1 oversubscribed DDR Infiniband fabric. Nodes have 24GB RAM memory and 250 GB hard disk. The Kalkyl cluster is running Scientific Linux 6.0 , a Red Hat Enterprise Linux. Each node comprises Intel quad-core Xeon 5520 2.26 GHz processors with 8MB cache. In this report, to avoid confusion with Erlang nodes, we use "host" to refer to the Kalkyl nodes (physical machines).

## 2.2 How Hosts and Nodes are Organized?

The same as ordinary distributed Erlang systems, this benchmark consists of a number of Erlang Virtual Machines (Erlang VMs) communicating with each other. The benchmark is run on a cluster of hosts and there can be multiple Erlang VMs on each host but each Erlang VM runs only one instance of *DEbench* application. For example, Figure 1 depicts a cluster with 2 hosts and 2 Erlang nodes (Erlang VMs) per each host.

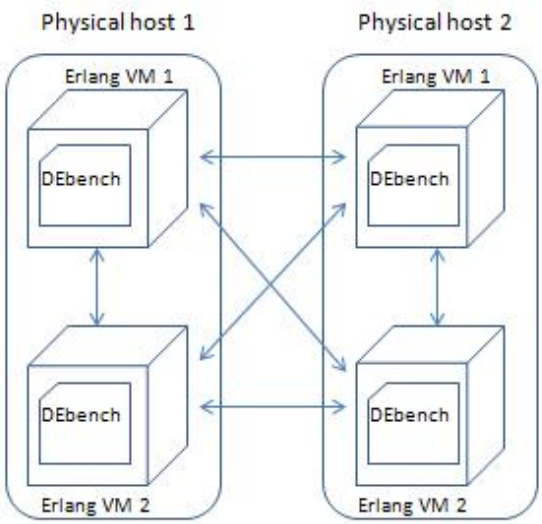


Figure 1: An example to show how 2 hosts, 2 Erlang VMs per each host, and one *DEbench* per each VM are organized

During the benchmark, all Erlang nodes are fully connected. For example as shown in Figure 1, each Erlang node is connected to all the other nodes.

This is because of transitive connections in a distributed Erlang system [8]. By default when node A connects to node B, and node B has a connection to node C, then node A automatically will connect to node C.

### 2.3 The Design and Implementation of DEbench

To evaluate the scalability of distributed Erlang, we need to measure how adding more Erlang nodes to a cluster of Erlang nodes will increase the throughput. By *throughput* we means the total number of successful distributed Erlang commands per second. To avoid reinventing the wheel, we search for benchmark suites which written in Erlang. Finally, we find that Basho Bench [9] is very close to your requirements, i.e. measuring the throughput and the latency of distributed Erlang commands. Since Basho Bench has a modular structure, we could use that modules in the development of *DEbench* and that makes our progress much faster. Here, we don't want to discuss the design and implementation of the common parts of *DEbench* and *Basho Bench* because you can find about them at the *Basho Bench* documentation [9]. Thus, we just describe that *DEbench*'s features which we design and implement to benchmark the scalability of distributed Erlang.

To benchmark distributed Erlang on thousands of Erlang nodes, we need to design an scalable approach. Thus, we design our benchmark in a P2P way to allow *DEbench*'s worker processes to operate without central synchronisation, which makes *DEbench* a suitable tool for large-scale distributed environments (e.g. clusters, clouds, grids). To exploit all the resources in the cluster, we run one *DEbench* on each Erlang VM, e.g. to benchmark a 50-node Erlang cluster, we run one instance of *DEbench* on each node which means we run 50 instance of *DEbench* in total.

Figure 2 represents the internal design of *DEbench*. *DEbench* first runs 10 worker processes in parallel. Each worker randomly selects an Erlang node and a distributed Erlang command from the config file. If the selected command is *spawn*, worker will run it on the selected Erlang node, otherwise the command will be run locally. *DEbench* measures and saves the latency and the throughput of commands in appropriate CSV files. In order to avoid contention for accessing the CSV files, they are stored on the local disk drive per each node.

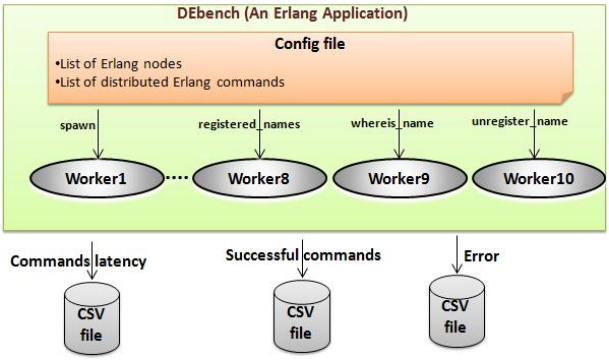


Figure 2: *DEbench* design

In this benchmark we want to benchmark the most commonly used distributed Erlang commands. Commands which we think are commonly used in a typical distributed Erlang system are:

- *spawn(Node, Fun)*: Creates a process at a remote node.
- *register_name(Name, Pid)*: Globally associates the name Name with a pid.
- *whereis_name(Name)*: Returns the pid with the globally registered name Name.
- *unregister_name(Name)*: Removes the globally registered name Name from the network of Erlang nodes.

We know that these commands are not used at the same rate in a typical distributed Erlang system. For instance, *spawn* is the most commonly used among them and *unregister_name* is used less in comparison to others. Thus, we use them in the benchmark with different rate to make it closer to a real life Erlang application.

If *register_name(Name, Pid)* be called more than once for the same pid (process identifier), an exception will be thrown. To address this problem we define an internal state for *DEbench* . Figure 3 shows three states that *DEbench* can have. *whereis_name(Name)* and *unregister_name(Name)* are run just if they come after *register_name(Name, Pid)*, otherwise they will be ignored. Also after running *register_name(Name, Pid)*, both *whereis_name(Name)* and *unregister_name(Name)* will be run respectively. To avoid name clashes, we use a timestamp function to generate a globally unique name.

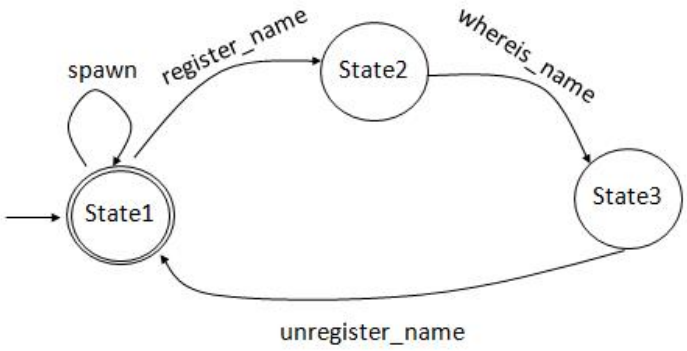


Figure 3: *DEbench* internal states

# 3 Benchmarking Results

In this section we employ *DEbench* to investigate the scalability limits of distributed Erlang. In the scalability benchmark, we measure the throughput for different size of Erlang clusters and observe how adding more Erlang nodes to a cluster affect its throughput. We also measure the latency of distributed Erlang commands (i.e. *spawn*, *register_name*, *whereis_name*,*unregister_name*) individually to find which of them could be a bottleneck for distributed Erlang systems.

## 3.1 Scalability

We perform the benchmark on 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100 Erlang nodes and measure the throughput by counting the successful operation per each experiment. There is one Erlang VM on each host and as always one *DEbench* on each VM. After the end of each experiment, we gather all the CSV files from the all nodes, and then aggregate them to find the total throughput, e.g. for benchmarking a 70-node cluster, 70 *DEbench*s are run simultaneously and consequently they produce 70 CSV files which need to be aggregated to find the throughput of the 70-node cluster. In this benchmark, the rate of using *spawn* command is 100 times more than other commands, i.e. per each one hundred of *spawn* command, we run one command of *register_name*, *whereis_name*,and *unregister_name* (0.5% global update). We perform the benchmark three times and Figure 4 represents the median of three executions. We see from Figure 4 that distributed Erlang for 0.5% global update, doesn't scale beyond ≈50 nodes. As Figure 4 shows when number of Erlang nodes exceed 50, the throughput starts to decrease.

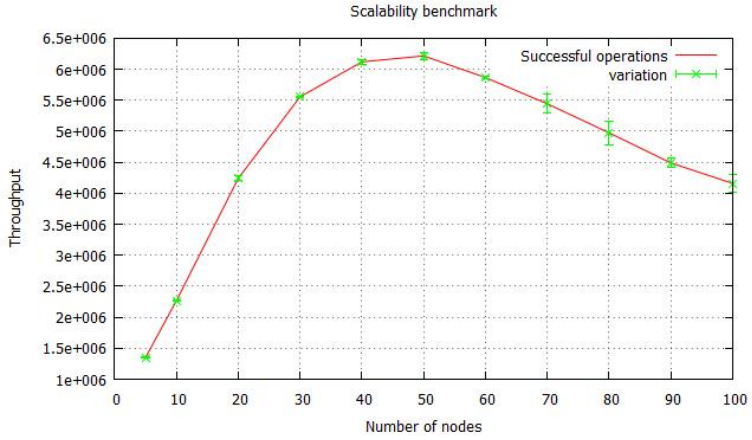


Figure 4: Scalability of Distributed Erlang with 1% global update

### Frequency of Global Operation

To ensure that global operation limits the scalability of distributed Erlang, in Figure 5 we compare the scalability of distributed Erlang for different rates of global operation. We see from the figure, as we use more global operation, scalability become more limited, e.g. maximum throughput for 1% global operation is 30 nodes (red curve) and for 0.33% global operation is 70 nodes (blue curve).

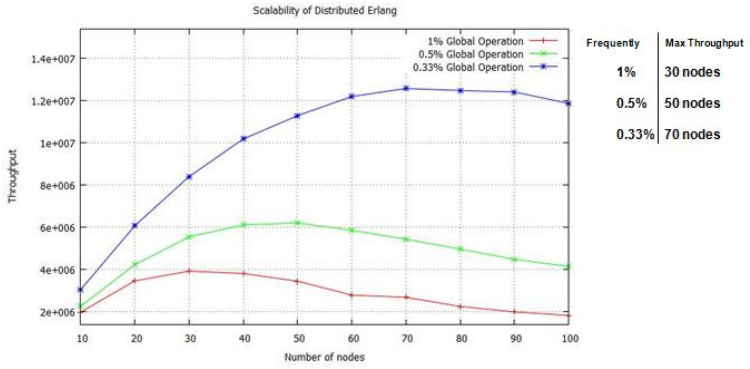


Figure 5: Frequency of Global Operation

## 3.2 What is the Bottleneck?

For finding the reason of limits in the scalability of distributed Erlang, we measure the latency of the commands (i.e. *spawn*, *register_name*, *whereis_name*, *unregister_name*) individually. Latency is the time that a command takes

to be completed. Figure 6 represents the latency of *register_name* and *unregister_name* commands.

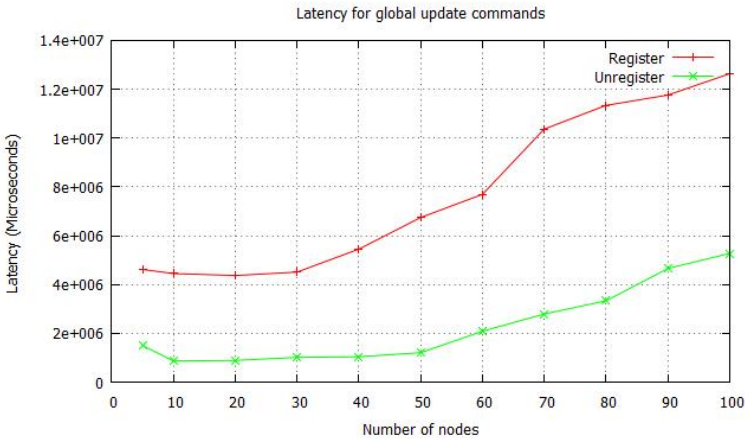


Figure 6: Latency of *register_name* and *unregister_name* commands

Commands *register_name* and *unregister_name* respectively register and unregister names globally. As shown in Figure 7 names are stored in global name tables on every node [10] and these tables are strongly consistent which means an update is considered complete whenever all nodes acknowledge it. Thus, as Figure 6 reveals, global update commands could become a bottleneck for distributed Erlang very soon. The latency of atomic operations like registering and unregistering increase rapidly for larger cluster since there is a lock mechanism for updating the replicated information.

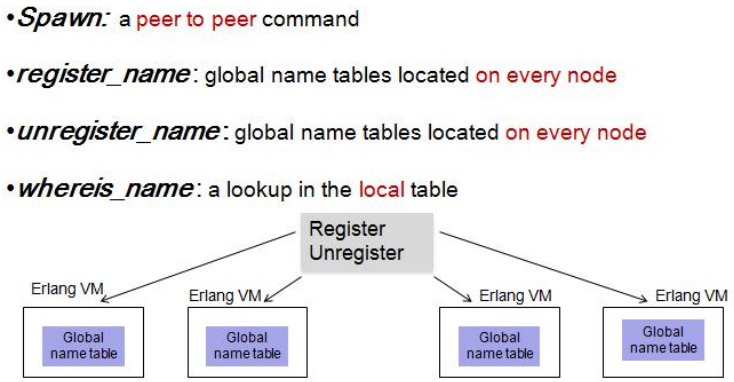


Figure 7: Local, P2P, and Global commands

As shown in Figure 7, *spawn* is a peer to peer command. In spawn, a process is run on a remote node and then a *received* is used to get an

acknowledge from that remote node. *whereis_name* command translates a name to a pid (process identifier) and it is fast because it is always done locally since global name tables exist on every node (Figure 7) [10].

Figures 8 and 9 represent the latency of *spawn* and *whereis_name* respectively. As we see from these diagrams, the latency of both (i.e. *spawn* and *whereis_name*) decrease when number of nodes increases. This is because when the size of cluster grows, *DEbench* spend more time in idle mode (sleep mode) for getting feedback from global update commands (i.e. *register_name* and *unregister_name*). Thus, when *DEbench* is more idle, the Erlang VM which hosts that *DEbench*, has more resources (e.g. CPU, RAM, etc) and consequently can respond more quickly to the other commands ( *spawn* and *whereis_name*)(Figure 10).

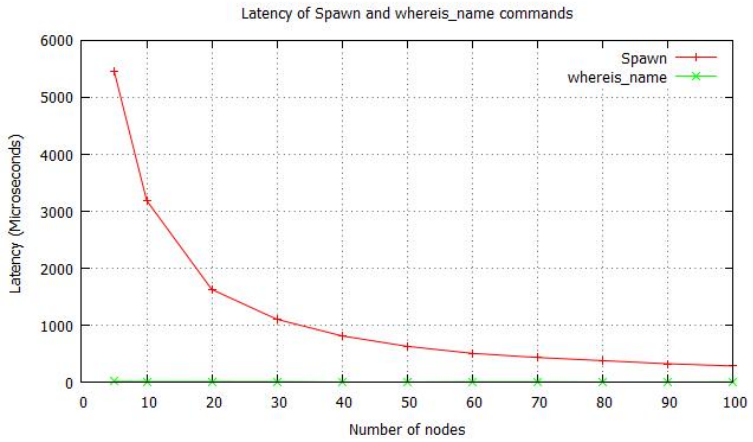


Figure 8: Latency of *spawn* command

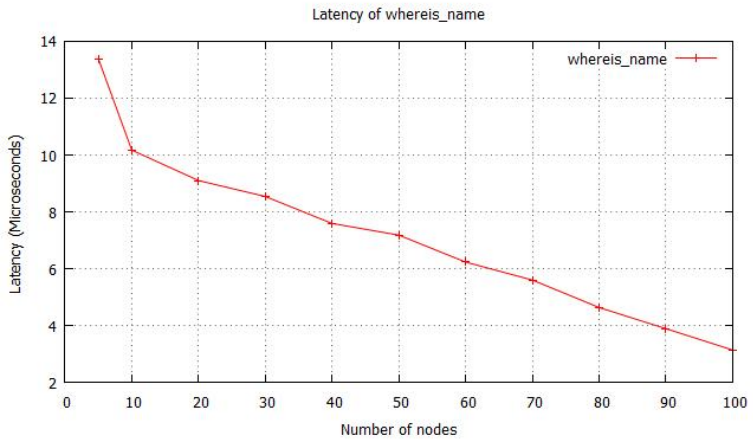


Figure 9: Latency of *whereis_name* command

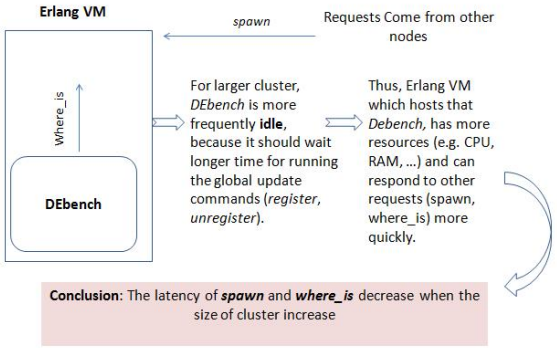


Figure 10: Why the latency of *spawn* and *whereis_name* decrease when cluster size grows

### 3.3 Benchmarking the Connectivity Threshold of Distributed Erlang

Global name tables are not the only threat for the scalability of distributed Erlang. Transitive connection between Erlang nodes could become a bottleneck for large-scale distributed Erlang applications [2]. In this section we employ *DEbench* for benchmarking the connectivity threshold of distributed Erlang. To benchmark the connectivity of distributed Erlang, we don't use commands which are related to global information. In the scalability benchmark (Section 3.1), we saw that global registration commands are non-scalable commands, thus, in this section we just use *spawn*.

Figure 11 shows that Erlang nodes can not communicate with each other when number of nodes exceed ≈1600 and number of failure among *spawn* commands increase dramatically.

We think this limits is due to number of open connection among Erlang nodes. Number of TCP connections in a transitive model growth rapidly by this formula: $\frac{N*\ (N\text{-}1)}{2}$ which N is number of Erlang nodes, e.g. for N=1600 Erlang nodes, the number of TCP connections among nodes is $\frac{1600*\ (1599)}{2}$=1279200

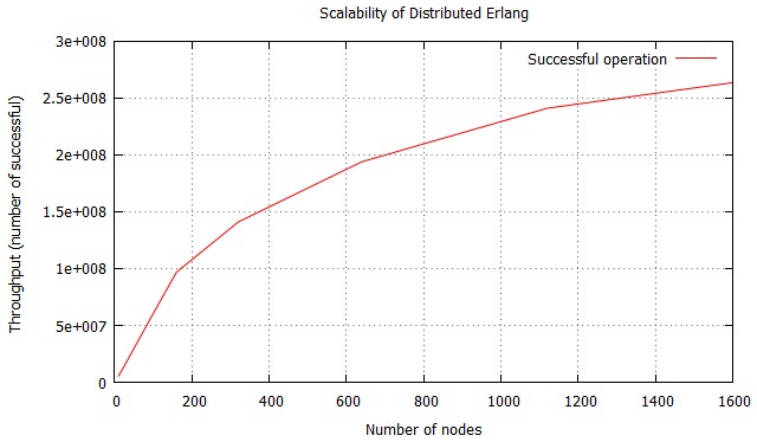


Figure 11: Connectivity Benchmark

# 4 Conclusion and Future Work

We investigate the scalability limits of distributed Erlang from two perspectives. First we benchmark commonly used Erlang commands (i.e. *spawn*, *register_name*, *whereis_name*,*unregister_name*) and we find that updating the global names table is a bottleneck for distributed Erlang.

We also benchmark the connectivity of distributed Erlang and the result from Section 3.3 reveals that distributed Erlang connectivity doesn't scale beyond ≈1600 nodes.

We are working in the RELEASE project on designing and implementing Scalable Distributed Erlang (SD Erlang) to improve the scalability of distributed Erlang by limiting transitive connection and global update to smaller group of Erlang nodes [11].

# References

[1] Naidila Sadashiv and S. M Dilip Kumar. Cluster, Grid and Cloud Computing: A Detailed Comparison. *The 6th International Conference on Computer Science & Education*, 2011.

[2] Olivier Boudeville, Francesco Cesarini, Natalia Chechina, Kenneth Lundin, Nikolaos Papaspyrou, Konstantinos Sagonas, Simon Thompson, Phil Trinder, and Ulf Wiger. RELEASE: A High-level Paradigm for Reliable Large-scale Server Software. *In proceedings of the Symposium on Trends in Functional Programming*, 2012.

[3] Ericsson. Who uses Erlang for product development, 2012. URL `http://www.erlang.org/faq/introduction.html#id49610`.

[4] Basho Basho Wiki. Concepts, 2012. URL http://wiki.basho.com/Concepts.html.

[5] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 1 edition edition, 2007. ISBN 193435600X.

[6] Amir Ghaffari, Natalia Chechina, and Phil Trinder. Scalable Persistent Storage for Erlang: Theory and Practice. 2013.

[7] SNIC-UPPMAX. The Kalkyl Cluster, 2012. URL http://www.uppmax.uu.se/the-kalkyl-cluster.

[8] Distributed Erlang Ericsson AB. Distributed Erlang, 2013. URL http://www.erlang.org/doc/reference_manual/distributed.html.

[9] Basho Technologies. Benchmarking, 2012. URL http://wiki.basho.com/Benchmarking.html.

[10] Ericsson AB EricssonABGlobal. Global Name Registration Facility, 2013. URL http://www.erlang.org/doc/man/global.html.

[11] Natalia Chechina, Phil Trinder, Amir Ghaffari, Rickard Green, Kenneth Lundin, and Robert Virding. The Design of Scalable Distributed Erlang. *Symposium on Implementation and Application of Functional Languages*, 2012.