ICT-287510
RELEASE
A High-Level Paradigm for Reliable Large-Scale Server Software
A Specific Targeted Research Project (STReP)

D6.7 (WP6): Scalability and Reliability for a Popular Actor Framework

Due date of deliverable: 31st December 2014
Actual submission date: 21st March 2015

Start date of project: 1st October 2011 Duration: 36 months

Lead contractor: The University of Glasgow Revision: 0.2

Purpose: To investigate the feasibility and limitations of adding SD Erlang scalability / reliability constructs, and design patterns, to a popular Actor framework for a dominant language.

Results: The main results of this deliverable are as follows.

- We have implemented the Orbit benchmark in Cloud Haskell and showed that it has similar scalability constraints as distributed Erlang.
- With a view to discovering the wider applicability of our results, we explore the application of SD Erlang techniques to Scala and Akka. Their explicit placement of actors and management of connections means that the s_group construct cannot be directly applied.
- We outline some approaches applying SD Erlang features, i.e. s_groups and semi-explicit process placement, directly in Cloud Haskell, indirectly in Scala and Akka.

Conclusion: We analyse the architecture of distributed applications in Cloud Haskell, Scala, and Akka. We conclude that applying SD Erlang approaches to reliable scalability could be directly applied to Cloud Haskell, but only indirectly applied to Scala or Akka, i.e. by introducing a library that will collect and update information about attributes of connected nodes.

Project funded under the European Community Framework 7 Programme (2011-14)		
Dissemination Level		
PU	Public	⋇
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential only for members of the consortium (including the Commission Services)	

Scalability and Reliability for a Popular Actor Framework

Contents

Executive Summary

This deliverable investigates the feasibility of introducing SD Erlang-like constructs for reliable scalability into other popular actor languages, libraries, and frameworks taking Cloud Haskell, Scala, and Akka as specific examples. All three are gaining popularity, and are inspired by the Erlang distributed actor model. The results for Cloud Haskell (Section 2) show that it exhibits scalability limitations very similar to distributed Erlang largely to the close similarity to Erlang. We conjecture that providing SD Erlang features will benefit Cloud Haskell the same way they benefit distributed Erlang, and will be an almost direct mapping from SD Erlang. Scala and Akka borrow from Erlang ideas of process interaction within the nodes but do not follow distributed Erlang inter-node interaction and full connectivity (Section 3). Therefore, application of SD Erlang ideas is not as direct as in Cloud Haskell. However, the scalability issues the SD Erlang tackles are not unique to distributed Erlang, and are applicable to other languages. That is introducing techniques to avoid maintaining a large number of connections as the number of nodes grows (e.g. s_groups in SD Erlang), and providing mechanisms to enable programmability and portability of large scale applications (e.g. semi-explicit placement in SD Erlang).

1 Introduction

The objectives of Task 6.7 are to provide a "report investigating the feasibility and limitations of adding SD Erlang scalability / reliability constructs, and design patterns to a popular Actor framework for a dominant language". The lead participant is the University of Glasgow.

The Erlang distributed actor programming model is widely acknowledged as very effective, and it has influenced and inspired a number of languages and frameworks, including Scala [16], Akka [27], Cloud Haskell [13], and Go [15]. Some of the most popular applications implemented in Erlang are as follows: mobile messaging application WhatsApp [19], the backend of the Facebook Messenger, the distributed fault-tolerant database Riak [3], and the open source message broker software RabbitMQ [22].

In the RELEASE project we have designed and implemented Scalable Distributed Erlang (SD Erlang) [6, 5, 4, 23, 24] – a modest conservative extension of distributed Erlang whose scalable computation model consists of two aspects: s_groups and semi-explicit placement. The reasons we have introduced s_groups are to reduce the number of connections a node maintains, and reduce the size of namespaces. A namespace is a set of names replicated on a group of nodes and treated as global in that group. The semi-explicit and architecture aware process placement mechanism was introduced to enable performance portability, a crucial mechanism in the presence of fast-evolving architectures.

In this deliverable we discuss possibilities and limitations of implementing SD Erlang ideas in three popular Actor languages and frameworks: Cloud Haskell (Section 2), Scala, and Akka (Section 3).

Partner Contributions to D6.7. The Glasgow team coordinated the deliverable and investigated opportunities to apply SD Erlang scalability principles to Scala and Akka programming languages. The ICCS team analysed Orbit performance in Cloud Haskell and distributed Erlang, and outlined features of SD Erlang that can benefit scalability of Cloud Haskell. The University of Kent and ESL teams contributed to identifying applications written in Erlang, and approaches used in other languages like Scala/Akka to scale distributed applications.

2 Scaling Cloud Haskell

Cloud Haskell [7] is a library that implements distributed concurrency in the Haskell functional programming language [17]. The model of reliable distributed computation is inspired by Erlang and provides a message passing communication model based on lightweight processes. The purpose is to make it easier to write programs for clusters of machines. The Cloud Haskell platform consists of a generic network transport API, libraries for sending static closures to remote nodes, a rich API for distributed programming, and a set of libraries modelled after Erlang/OTP. It currently provides two generic network transport back-ends, for TCP and in-memory messaging, as well as several other back-ends including a transport for Windows Azure.

2.1 A Closer Look at Cloud Haskell's implementation

Lightweight processes in Cloud Haskell are implemented as computations in the `Process` monad (which uses GHC's lightweight threads, in the most common Haskell implementation). Roughly, Cloud Haskell applications use the following modules:

- `Control.Distributed.Process` for setting up nodes, processes, etc. Applications employ a specific Cloud Haskell backend to initialize the transport layer by setting up the appropriate topology.

  This module provides the machinery for starting nodes, spawning processes, sending and receiving messages, monitoring and linking processes and nodes, as well as the implementation of a (missing in Erlang) typed channel mechanism. Processes are `forkIO` threads. The `Process` monad is an instance of `MonadIO`, which additionally keeps track of the state associated with a process, primarily its message queue. Data that is exchanged in messages between processes must implement the type class `Serializable`.

- `Network.Transport.*`, providing a network abstraction layer which is supplemented with various implementations, among which `Network.Transport.TCP`.

  In Cloud Haskell networks, nodes are modelled by `EndPoints` (heavyweight stateful objects), identified by `EndPointAddresses` (serializable, so they can be send and received as parts of messages). Communication between nodes is modelled by `Connections`, which are designed to be unidirectional and lightweight. Incoming messages for *all connections* to an `EndPoint` are collected via a shared receive queue.

One of the dedicated processes that Cloud Haskell maintains in each node of an application is the Discovery process [12, §4.2.4]. This process is used for inter-node communication: before an application can communicate with a remote node, it must first know that it exists. Function `getPeers` returns a structure containing information about all known nodes; this information is obtained by combining:

- Static peer discovery: Examines the nodes currently running on a fixed list of hosts (provided statically when the application starts). It accomplishes this by by sending a message to the node registration server of each host which then returns the nodes.
- Dynamic peer discovery: Finds nodes running on hosts that are not mentioned in the applications configuration. It accomplishes this by sending a UDP broadcast message to all hosts on the local network. Any Cloud Haskell nodes running on those hosts will respond by sending a message to the originating process. (This can be disabled for security reasons.)

The primary mechanisms in Cloud Haskell for message passing and name registering in a distributed application are the following. They are provided by the Cloud Haskell module `Control.Distributed.Process` [9] and, in the case of global registration, by the module `Control.Distributed.Process.Global` [10]. Both are heavily inspired by similar mechanisms implemented in Erlang/OTP and, in particular, the `global` module.

- Registration of named processes, implemented in `Internal.Primitives`:
  - `register :: String -> ProcessId -> Process ()`

    Registers a process with the local registry (asynchronous).
  - `registerRemoteAsync :: NodeId -> String -> ProcessId -> Process ()`

    Registers a process with a remote registry (asynchronous). This requires the identifier of the remote node.
- Sending messages to named processes, implemented in `Internal.Primitives`:
  - `nsend :: Serializable a => String -> a -> Process ()`

    Sends a message to a named process in the local registry (asynchronous).
  - `nsendRemote :: Serializable a => NodeId -> String -> a -> Process ()`

    Sends a message to a named process in a remote registry (asynchronous).

  Both these functions wrap calls to the following low-level function, which is implemented in `Internal.Messaging`:

  -
    ```
    sendCtrlMsg :: Maybe NodeId   -- Nothing for the local node
                -> ProcessSignal -- Message to send
                -> Process ()
    ```

    Sends a control message to a process, local or remote.

- Global name registration, implemented in `Global.Server`. Lookups are always local and therefore fast (each node maintains an up-to-date list of all registered names). On the other hand, registration requires acquiring a cluster lock and broadcasting; therefore, it should not be done too often.

  –
  ```
  globalRegister :: TagPool -> String -> ProcessId
                 -> ResolutionMethod -> Process Bool
  ```

  The global equivalent of `register`, which provides a name registry for the entire cluster of nodes. Registered names are synchronously shared with all nodes in the cluster, and given to new nodes as they enter the cluster. If the given name is already registered, `False` is returned. The `ResolutionMethod` parameter determines how conflicts are handled when two clusters merge, if they have different values for the same name.

  – `globalWhereis :: TagPool -> String -> Process (Maybe ProcessId)`

  Retrieves the process identifier of a registered global name.

- Extended process registry implemented in `Platform.Service.Registry` [11]. This module provides an extended process registry, offering slightly altered semantics to the built-in register and unregister primitives. Using this service, it is possible to monitor a name registry and be informed whenever changes take place. This registry is a single process, parametrised by the types of key and property value it can manage. It is possible to start multiple registries and inter-connect them with one another, via registration. Such a mechanism is not provided in the basic libraries of Erlang/OTP.

In the rest of this section we compare the scalability behaviour of a distributed application, written in Erlang, to the behaviour of the same application written in Cloud Haskell. Experimenting with a distributed application in Cloud Haskell was crucial for understanding the Cloud Haskell platform and studying its implementation. Furthermore, by comparing the behaviours of the two versions of the same application, we intended to verify that the two platforms (distributed Erlang and Cloud Haskell) exhibit similar scalability properties when it comes to distributed applications of several nodes. This, combined with the apparent similarity in the implementation of the two platforms, would provide some confidence that a solution that works well in the case of distributed Erlang could also work well in the case of Cloud Haskell.

2.2 Orbit in Distributed Erlang and Cloud Haskell

To compare Cloud Haskell with distributed Erlang in practice, we reimplemented the distributed Erlang Orbit benchmark from [23] in Cloud Haskell.[1] Orbit is one of the benchmarks that we have repeatedly used for the needs of this project. It is a distributed implementation of Orbit, which in turn is a symbolic computing kernel and a generalization of a transitive closure computation. Orbit is described in detail in a previous deliverable (D3.4, §3.1) and elsewhere.[2]

The implementation in Cloud Haskell was based on the corresponding implementation in Erlang,[3] and we tried to make this translation as direct as possible. Table 1 shows the lines of code per source file of each implementation. Notice that the Haskell sources are approximately 50% longer; this is partly explained by the fact that we added type signatures for all top-level functions (as is customary in Haskell and considered good practice), whereas on the other hand the Erlang sources do not contain `-spec` annotations. Notice also that, in the translation, we opted to merge two source files (`master.erl` and `worker.erl`), as they contained mutual dependencies that would be difficult to maintain in Haskell.

[1]Available from `https://github.com/release-project/cloud-orbit`.

[2]More detailed documentation is available from `https://github.com/release-project/benchmarks`, under the directory `Orbit/`.

[3]Available from the project's main repository `https://github.com/release-project/RELEASE`, under the directory `Research/Benchmarks/orbit-int/`.

Table 1: Implementation of Orbit in distributed Erlang (left) and in Cloud Haskell (right).

Source file	**Lines**
bench.erl	142
credit.erl	64
master.erl	215
sequential.erl	108
table.erl	110
worker.erl	268
Total	907

Source file	**Lines**
Bench.hs	174
Credit.hs	78
MasterWorker.hs	524
Sequential.hs	107
Table.hs	140
Tests.hs	196
Utils.hs	206
Total	1,427

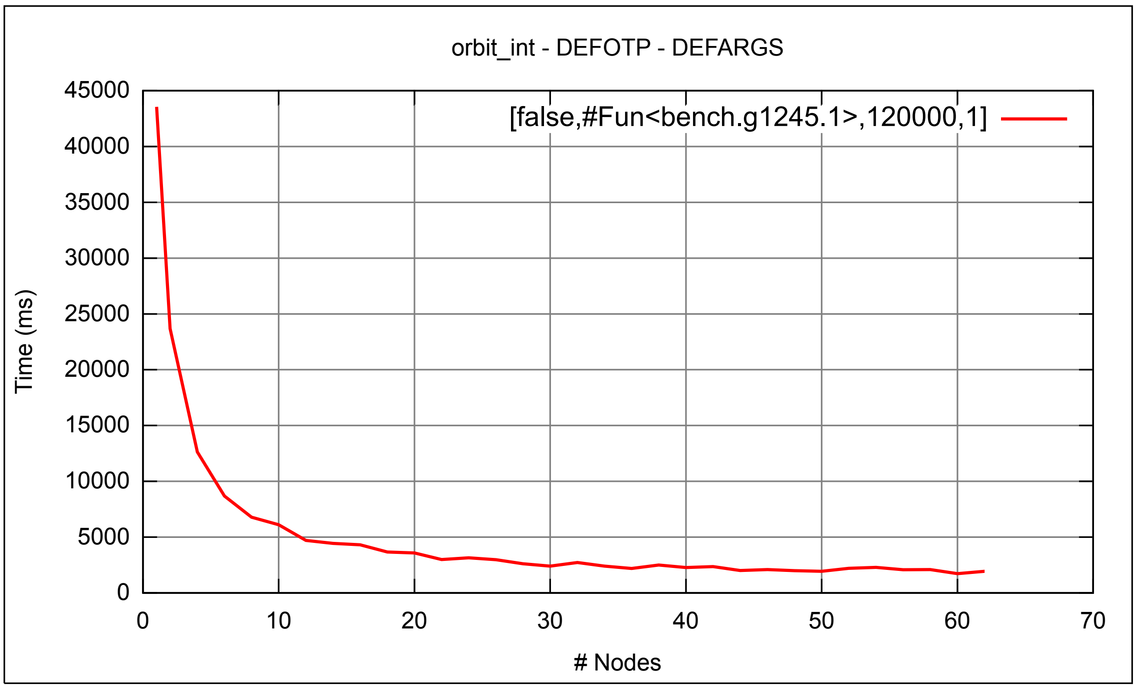
Two things that deserve special mention concerning the Haskell implementation, are the following. First, we decided to use Haskell's purely functional (and lazy) arrays (`Data.Array`) instead of impure alternatives, such as mutable arrays in the IO monad (`Data.Array.IO`) or the ST monad (`Data.Array.ST`), which may give better performance. The reason behind this option is that we wanted to be as close as possible to the Erlang implementation, which uses Erlang's functional arrays (implemented, however, in a completely different way). Second, we had to be careful with Haskell's lazy evaluation. The first version of our Haskell implementation had a very large memory footprint, because the memory occupied by arrays could not be freed (array elements were computed lazily and "earlier" arrays could not be freed, as they were necessary to compute elements of "newer" arrays that had not been needed yet). This was fixed by adding a couple of `seq` operators, forcing the strict evaluation of array elements.

The experiment consisted of running the two implementations of Orbit with two different sizes of input data, coded "120k" and "300k" in what follows (generator `gg1245` was used, parametrised by this size of the space). We used a set of configurations with an increasing number of distributed nodes. The configuration that is used as the basis for the comparison (i.e., against which speedup is calculated) consists of a single node where all calculation takes place. The remaining configurations used one master node and several worker nodes; calculation is performed in the worker nodes, whereas the master serves as the coordinator. When counting the nodes of each configuration, we only count the nodes where calculation takes place; we take $n = 1$ for the base configuration and $n > 1$ for a configuration with n slave nodes and one master node.

Table 2 shows the results obtained from our experiment. As it was not our intention to measure the effects of network communication in this experiment, all nodes were started on the same physical machine. We used a server with four AMD Opteron 6276 (2.3 GHz, 16 cores, 16M L2/16M L3 Cache), giving a total of 64 physical cores, and 128GB of RAM, running Linux 3.10.5-amd64. Execution time is measured in seconds and the speedup is relative to the sequential version of Orbit (all calculation performed in the master node). For the version of Orbit implemented in distributed Erlang, we used the Bencherl infrastructure [1],[4] that has been developed for the needs of this project (detailed documentation can be found in deliverable D2.1, §5). For the version implemented in Cloud Haskell, we used custom scripts that are included in the benchmark's source code. Figures 1 through 4 present diagrams of execution time and speedup for both implementation languages and both input sizes.

The first thing to notice from the obtained results is that the Cloud Haskell implementation is significantly slower than its Erlang counterpart. Execution times for Haskell are on the average an order of magnitude higher than the corresponding ones in Erlang; they range from 4-5 times higher (for the small input size and 16-18 nodes) to 18-20 times higher (for the large input size and the sequential version, or the one using two nodes). We attribute this to the bad performance of Haskell's purely functional arrays, in combination with the language's laziness. The use of functional

[4]Available from `http://release.softlab.ntua.gr/bencherl/` and also on GitHub: `https://github.com:softlab-ntua/bencherl`.



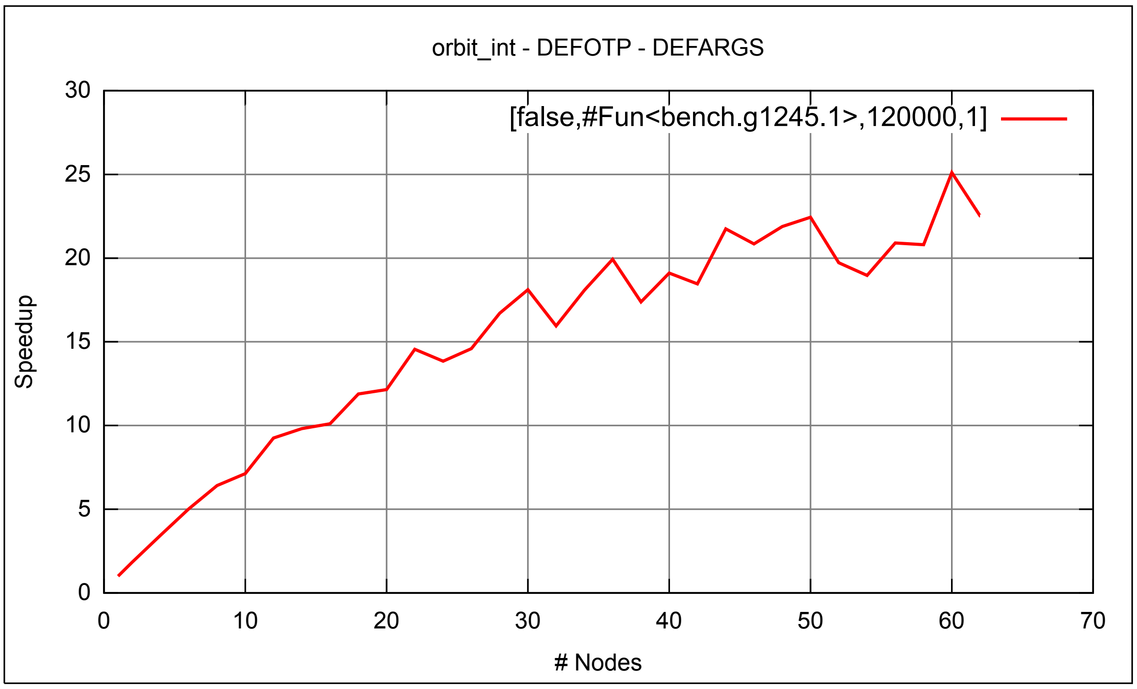
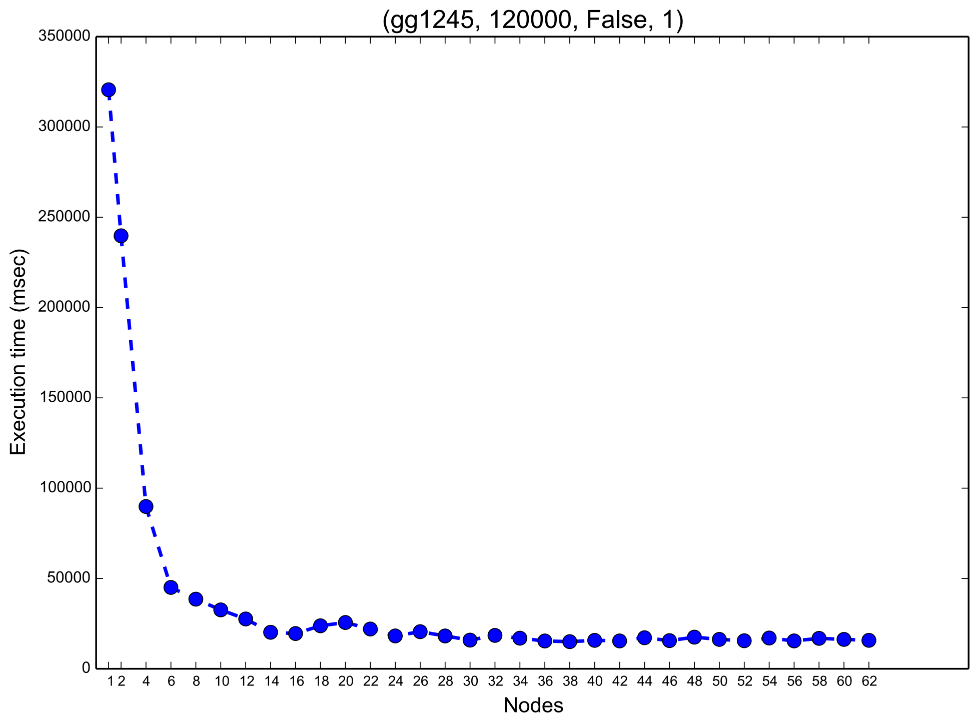


Figure 1: Orbit in Erlang, size = 120K, execution time and speedup.



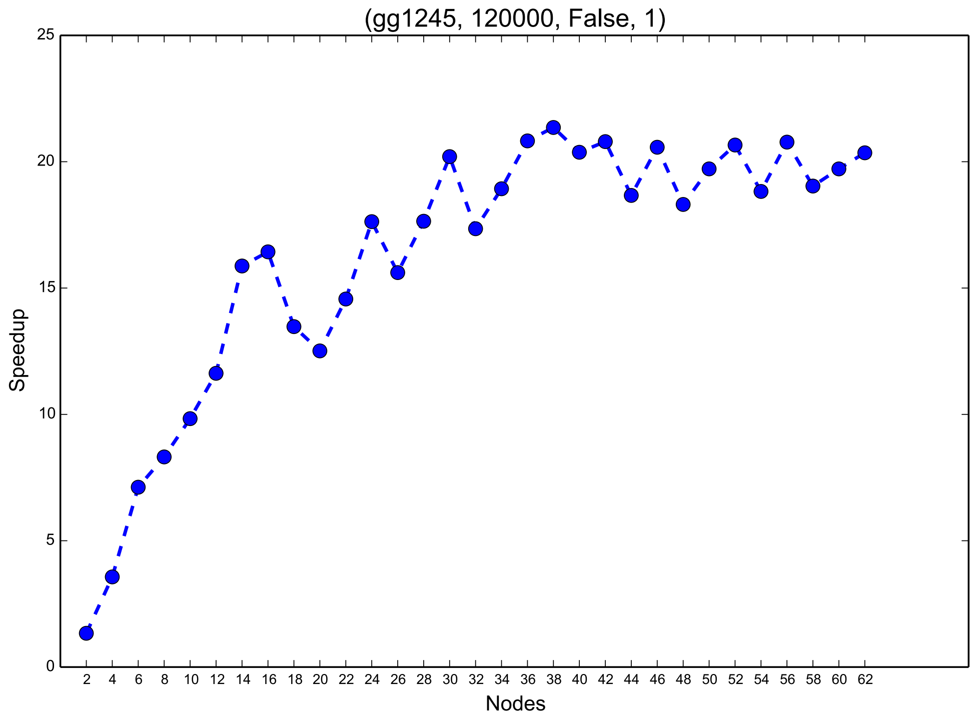
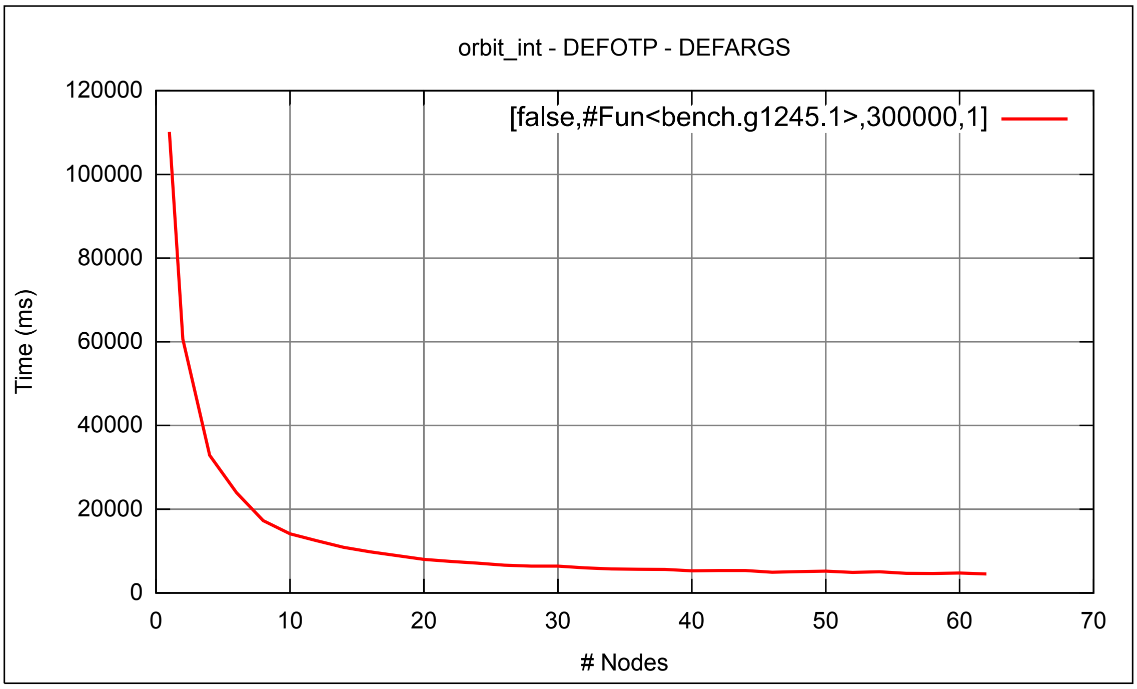


Figure 2: Orbit in Cloud Haskell, size = 120K, execution time and speedup.



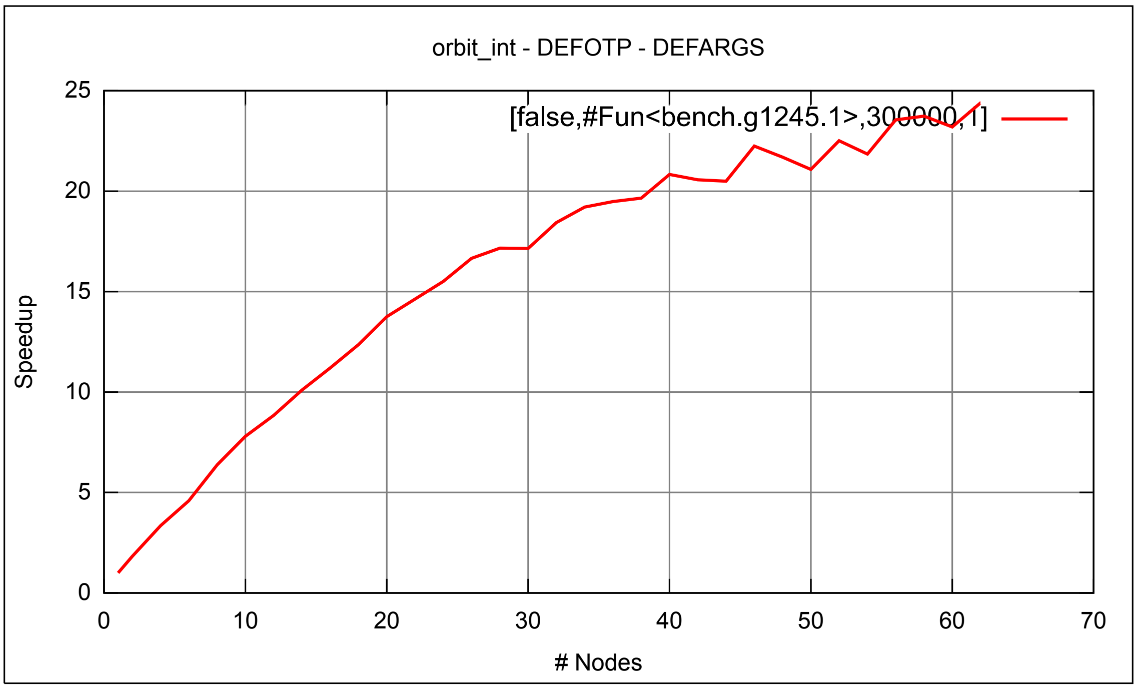
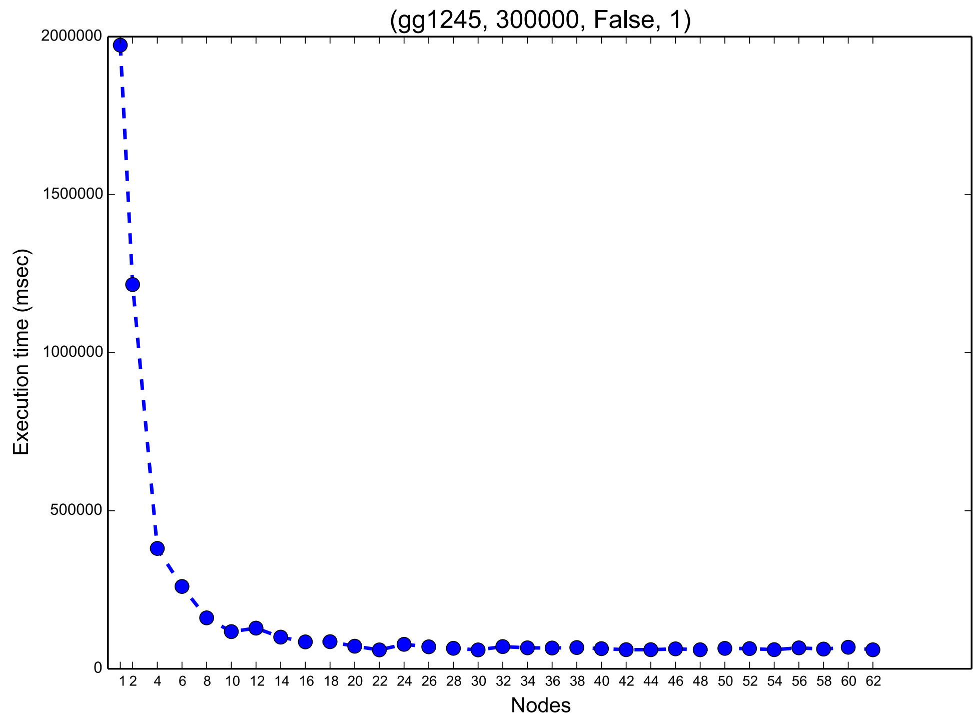


Figure 3: Orbit in Erlang, size = 300K, execution time and speedup.



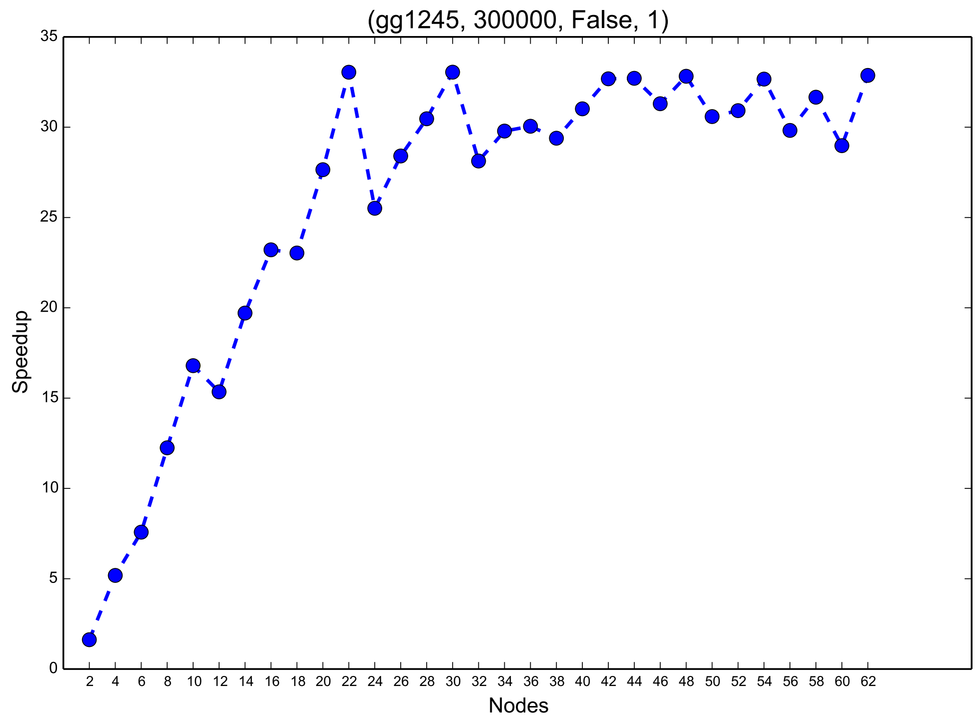


Figure 4: Orbit in Cloud Haskell, size = 300K, execution time and speedup.

Table 2: Orbit in Erlang and Cloud Haskell, execution time (in seconds) and speedup (relative to the sequential version running on one node).

	size = 120k				size = 300k			
	Erlang		Cloud Haskell		Erlang		Cloud Haskell	
nodes	time	speedup	time	speedup	time	speedup	time	speedup
1	43.56	1.00	320.59	1.00	110.16	1.00	1,973.11	1.00
2	23.68	1.84	239.74	1.34	60.61	1.82	1,215.62	1.62
4	12.63	3.45	89.84	3.57	32.91	3.35	380.68	5.18
6	8.68	5.02	45.04	7.12	23.99	4.59	260.36	7.58
8	6.79	6.41	38.55	8.32	17.28	6.37	161.09	12.25
10	6.11	7.13	32.61	9.83	14.13	7.80	117.47	16.80
12	4.71	9.26	27.58	11.63	12.47	8.83	128.57	15.35
14	4.44	9.81	20.20	15.87	10.90	10.10	100.11	19.71
16	4.31	10.11	19.51	16.43	9.83	11.20	85.01	23.21
18	3.66	11.89	23.79	13.47	8.91	12.37	85.66	23.03
20	3.59	12.14	25.62	12.51	8.01	13.75	71.37	27.65
22	2.99	14.56	22.01	14.57	7.53	14.63	59.71	33.04
24	3.15	13.84	18.19	17.63	7.10	15.51	77.34	25.51
26	2.99	14.59	20.54	15.61	6.61	16.66	69.46	28.41
28	2.61	16.71	18.17	17.64	6.42	17.16	64.76	30.47
30	2.41	18.11	15.87	20.20	6.42	17.15	59.71	33.05
32	2.73	15.95	18.48	17.35	5.98	18.43	70.15	28.13
34	2.41	18.09	16.94	18.93	5.73	19.21	66.24	29.79
36	2.19	19.93	15.40	20.82	5.65	19.49	65.66	30.05
38	2.51	17.38	15.01	21.35	5.60	19.66	67.13	29.39
40	2.28	19.11	15.74	20.37	5.29	20.83	63.61	31.02
42	2.36	18.46	15.42	20.80	5.36	20.57	60.38	32.68
44	2.00	21.75	17.18	18.67	5.37	20.50	60.33	32.71
46	2.09	20.85	15.58	20.57	4.95	22.25	63.04	31.30
48	1.99	21.89	17.51	18.31	5.08	21.69	60.12	32.82
50	1.94	22.44	16.26	19.72	5.23	21.08	64.50	30.59
52	2.21	19.73	15.52	20.66	4.89	22.51	63.82	30.92
54	2.30	18.96	17.03	18.82	5.04	21.85	60.41	32.66
56	2.08	20.90	15.43	20.78	4.68	23.55	66.16	29.82
58	2.09	20.80	16.84	19.04	4.64	23.74	62.32	31.66
60	1.73	25.11	16.26	19.72	4.75	23.20	68.09	28.98
62	1.93	22.52	15.75	20.35	4.52	24.39	60.03	32.87

arrays in Haskell places a particularly heavy burden on the garbage collector.

The second thing to notice is that the Cloud Haskell implementation exhibits super-linear speedup, e.g., 7.12 with 6 nodes and 15.87 with 14 nodes for the small input size, and 12.25 for 8 nodes, 16.80 for 10 nodes, and 33.04 for 22 nodes, for the large input size. We attribute this to the fact that a large percentage of execution time (56% and 57% in the small and large input sizes, respectively, for execution on a single node, or 180 and 1,141 seconds, respectively) is spent during garbage collections. For the two input sizes that we used, the total amount of data allocated in the heap (not at the same time, obviously) is 143GB and 892GB, respectively, out of which the garbage collector had to copy 220MB and 636MB, respectively. When calculations are distributed over a larger number of nodes, memory allocation reduces and the garbage collector takes a significantly smaller amount of execution time. Moreover, when fewer data are allocated and used, the benefits of the cache memory are higher.

In terms of scalability, the two implementations show roughly the same behaviour, if we disregard the problems with memory management and garbage collection that are far more apparent in the case of Cloud Haskell. The Erlang implementation shows a constantly increasing speedup, which approaches 20-25 around 64 nodes and shows a tendency to stabilize at this range of values. On the other hand, the Cloud Haskell implementation shows a speedup that quickly approaches 20 and 30 (for the two input sizes, respectively) around 32 nodes, but has a tendency to remain constant or to slightly decrease for larger numbers of nodes.

2.3 Summary

The above experiments suggest that Cloud Haskell applications exhibit similar scalability limitations as distributed Erlang ones. This is also supported by some recent discussions concerning the limitations of Cloud Haskell when it comes to applications spanning large distributed networks [2]. Like in distributed Erlang it is problematic to maintain all to all connections in a large network, as it places considerable strain on system resources, especially in the case of connection-oriented transport layer protocols, such as TCP.

Solutions proposed by the Cloud Haskell community follow two main approaches:

- Handle connections more efficiently by having the transport layer on each node manage the connections on each of its endpoints. Heuristics could be applied, including, e.g., closing connections to remote endpoints that haven't recently been used, closing old connections in favour of new ones, etc. It should be mentioned here that connection management in Cloud Haskell is not as automatic as in Erlang/OTP: failing connections must be re-established explicitly by the application's code.

- Create independent federated (fully connected) clusters of nodes. Such clusters could then be connected to each other by a small group of members, which would act as intermediaries.

However, for the Cloud Haskell development team, these problems and their solutions are of a quite low priority ("way off the radar, now" [2], in December 2012). Of also low priority is massive scalability support, in comparison to the implementation of better connection management (considered of medium priority).

We believe that using our experience in scaling distributed Erlang, and introducing s_groups and semi-explicit placement could significantly improve scalability of Cloud Haskell applications while allowing programmers using familiar coding techniques.

3 Scaling Scala & Akka

3.1 Scala

Scala is a statically typed programming language that combines features of both object-oriented and functional programming languages [20]. It is a Java-like language implemented as a library that compiles to Java Virtual Machine (JVM) and runs at comparable speed [21]. Scala provides sophisticated static type system [18]. This was a design decision to enable painless access of Java libraries from Scala programs and vise versa. Two of the most famous software products that use Scala are online social networking service Twitter [14] and business-oriented social networking service LinkedIn [25]. Twitter wrote most of its backend in Scala. The choice of the language was defined by the following factors: support of long living processes, encapsulation (rather than performance and scalability when comparing with Ruby), and a large number of developers who can code Java. LinkedIn uses Scala to implement its social graph service.

3.2 Akka

Akka is an event-driven middleware framework to build reliable distributed applications [27]. Akka is implemented in Scala. Fault tolerance in Akka is implemented using similar to Erlang 'let it crash' philosophy and supervisor hierarchies [26]. To support reliability Akka uses dynamic types. An actor can only have one supervisor which is the parent supervisor but similarly to Erlang actors can monitor each other. Due to the possibility of creating (spawning) an actor on a different Java VM, two mechanisms are available to access an actor: logical and physical. A logical path follows parental supervision links toward the root, whereas, a physical actor path starts at the root of the system at which the actual actor object resides. Like Erlang Akka does not support guaranteed delivery of messages.

3.3 Scaling

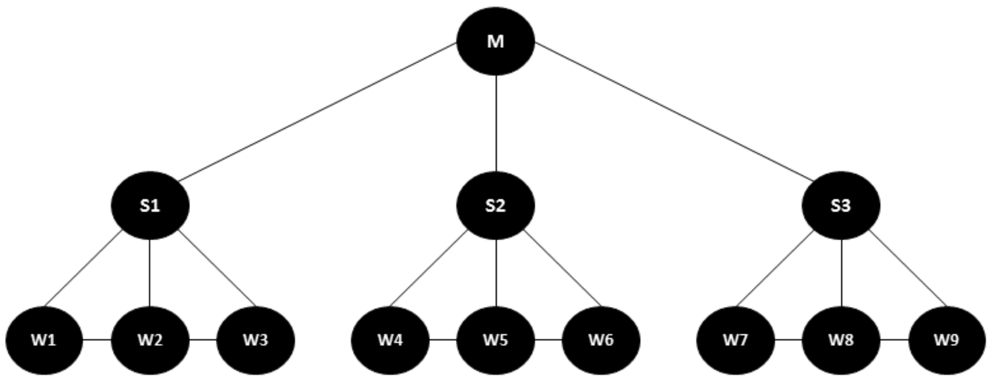
From available documentation on scalable applications implemented in Scala and Akka it appears that those applications are data intensive and the scalability is achieved by using Hadoop-like approaches. That is nodes have master/worker relationship, and the system structure consist of two layers: Hadoop Distributed File System (HDFS) and Map Reduce [8]. The focus of HDFS is to store data in a large system and then easily access this data, whereas the focus of Map Reduce is to break a large job into small tasks, distribute the tasks between worker nodes, and execute them in parallel.

Ideas from SD Erlang. Unlike Erlang VMs in distributed Erlang, JVMs in Scala and Akka applications are not typically transitively interconnected. So, there is no need to constrain transitive connectivity in these languages. However, it is common for these applications to scale horizontally. In this case a master node becomes a single point of failure because it is connected to all worker nodes. To decrease the load on the master node and increase the number of worker nodes we propose to arrange worker nodes into groups, for example, according to a tree structure. That is the master node distributes tasks to submasters nodes, and then submaster nodes distribute the tasks to worker nodes. This hierarchical structure will reduce the number of connections not only in Hadoop-like applications but also in applications where worker nodes communicate with each other. Depending on the application other structures like chain and partial mesh can be also incorporated (Figure 5). To improve efficiency of decision making policies, nodes that belong to multiple groups can serve as gateway nodes that propagate state information about nodes from one group to another.

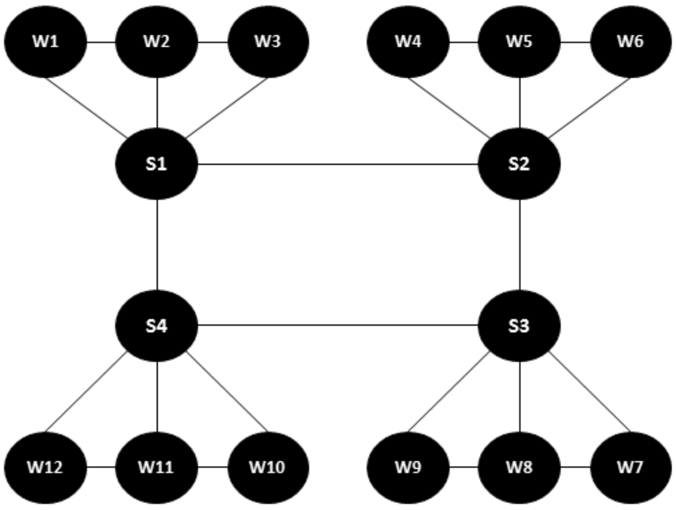
In addition, we believe that semi-explicit placement including node attributes and distance metrics from will significantly improve and simplify portability of applications written in Akka. That is instead of defining a particular node to where a process should be spawned, a programmer will only identify properties of the target node, such as available hardware, or load, or a distance from the parental node, and the process will be spawned to one of the nodes that satisfies given restrictions. A detailed discussion on introducing semi-explicit placement in SD Erlang is presented in deliverable D3.5: SD Erlang Performance Portability Principles [24].

4 Implications and Future Work

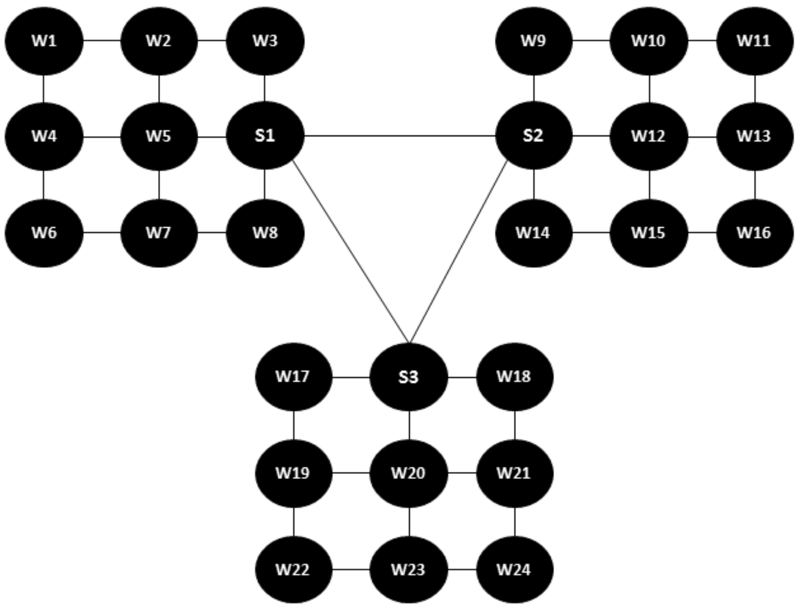
In this deliverable we analyse possibilities of implementing SD Erlang features in other actor languages to increase scalability of distributed applications. For that we have implemented Orbit benchmark in distributed Erlang and Cloud Haskell and compared the performance (Section 2). The results show that Cloud Haskell exhibits very similar scalability limitations as distributed Erlang. From this and from the fact that Cloud Haskell was inspired by distributed Erlang and has similar features, we conclude that introducing s_groups and semi-explicit placement from SD



(a) Tree



(b) Chain



(c) Partial Mesh

Figure 5: Strategies for Grouping Nodes

Erlang will significantly improve scalable reliability and portability of Cloud Haskell applications while retaining familiar programming idioms. We have also analysed scaling in Scala and Akka, and identified the use for SD Erlang-like node grouping and semi-explicit placement.

Change Log

Version	Date	Comments
0.1	11/03/2015	First Version Submitted to Internal Reviewers
0.2	21/03/2015	Revised version based on comments from K. Sagonas submitted to the Commission Services

References

[1] S. Aronis, N. Papaspyrou, K. Roukounaki, K. Sagonas, Y. Tsiouris, and I. E. Venetis. A scalability benchmark suite for Erlang/OTP. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang*, pages 33–42. ACM, 2012.

[2] Cloud Haskell: Support for large distributed networks. https://cloud-haskell.atlassian.net/browse/CH-4. Accessed 5/2/2015.

[3] Basho. Riak, 2015. http://basho.com/.

[4] N. Chechina, H. Li, S. Thompson, and P. Trinder. Scalable SD Erlang reliability model. Technical Report TR-2014-004, The University of Glasgow, December 2014.

[5] N. Chechina, H. Li, P. Trinder, and A. Ghaffari. Scalable SD Erlang computation model. Technical Report TR-2014-003, The University of Glasgow, December 2014.

[6] N. Chechina, P. Trinder, A. Ghaffari, R. Green, K. Lundin, and R. Virding. Scalable reliable SD Erlang design. Technical Report TR-2014-002, The University of Glasgow, December 2014.

[7] Cloud Haskell. http://haskell-distributed.github.io/. Accessed 5/2/2015.

[8] R. de Fatima Pereira, W. Akio Goya, K. Langona, N. Mimura Gonzalez, T.C. Melo de Brito Carvalho, J.-E. Mangs, and A. Sefidcon. Exploiting Hadoop topology in virtualized environments. In *Services (SERVICES), 2014 IEEE World Congress on*, pages 301–308, June 2014.

[9] Cloud Haskell: Package distributed-process. https://github.com/haskell-distributed/distributed-process/. Accessed 5/2/2015.

[10] Cloud Haskell: Package distributed-process-global. https://github.com/haskell-distributed/distributed-process-global/. Accessed 5/2/2015.

[11] Cloud Haskell: Package distributed-process-platform. https://github.com/haskell-distributed/distributed-process-platform/. Accessed 5/2/2015.

[12] J. Eppstein. *Functional Programming for the Data Centre*. PhD thesis, University of Cambridge, Computer Laboratory, June 2011. Available from http://research.microsoft.com/en-us/um/people/simonpj/papers/parallel/epstein-thesis.pdf.

[13] J. Epstein, A. P. Black, and S. Peyton-Jones. Towards Haskell in the Cloud. *SIGPLAN Not.*, 46(12):118–129, 2011.

[14] K. Finley. Twitter engineer talks about the company's migration from Ruby to Scala and Java, July 2011. [Available 2015].

[15] Google. The Go programming language, 2015. https://golang.org/.

[16] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.

[17] The Haskell programming language. `http://www.haskell.org/`. Accessed 5/2/2015.

[18] V. Layka and D. Pollak. Scala type system. In *Beginning Scala*, pages 133–151. Springer, 2015.

[19] A. O'Connell. Inside erlang, the rare programming language behind WhatsApp's success, 2014. http://www.fastcolabs.com/3026758/inside-erlang-the-rare-programming-language-behind-whatsapps-success.

[20] 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 IC/2004/64, Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland, 2004.

[21] M. Odersky and T. Rompf. Unifying functional and object-oriented programming with Scala. *Commun. ACM*, 57(4):76–86, 2014.

[22] Pivotal Software. Erlang AMQP client library, 2015. https://www.rabbitmq.com/erlang-client-user-guide.html.

[23] RELEASE Project. Deliverable D3.4: Scalable Reliable OTP Library Release, September 2014.

[24] RELEASE Project. Deliverable D3.5: SD Erlang Performance Portability Principles, March 2015.

[25] Scala. Scala at LinkedIn. [Available 2015].

[26] Typesafe Inc. *Akka Documentation: Release 2.1 - Snapshot*, July 2012. http://www.akka.io/docs/akka/snapshot/.

[27] Typesafe Inc. Akka: Event-driven middleware for Java and Scala, 2012. www.typesafe.com/technology/akka.