Evaluating Scalable Distributed Erlang for Scalability and Reliability

Natalia Chechina, Kenneth MacKenzie, Simon Thompson, Phil Trinder, Olivier Boudeville, Viktória Fördős, Csaba Hoch, Amir Ghaffari, and Mario Moro Hernandez

Abstract—Large scale servers with hundreds of hosts and tens of thousands of cores are becoming common. To exploit these platforms software must be both scalable and reliable, and distributed actor languages like Erlang are a proven technology in this area. While distributed Erlang conceptually supports the engineering of large scale reliable systems, in practice it has some scalability limits that force developers to depart from the standard language mechanisms at scale. In earlier work we have explored these scalability limitations, and addressed them by providing a Scalable Distributed (SD) Erlang library that partitions the network of Erlang Virtual Machines (VMs) into scalable groups (s_groups). This paper presents the first systematic evaluation of SD Erlang s_groups and associated tools, and how they can be used. We present a comprehensive evaluation of the scalability and reliability of SD Erlang using three typical benchmarks and a case study. We demonstrate that s_groups improve the scalability of reliable and unreliable Erlang applications on up to 256 hosts (6,144 cores). We show that SD Erlang preserves the class-leading distributed Erlang reliability model, but scales far better than the standard model. We present a novel, systematic, and tool-supported approach for refactoring distributed Erlang applications into SD Erlang. We outline the new and improved monitoring, debugging and deployment tools for large scale SD Erlang applications. We demonstrate the scaling characteristics of key tools on systems comprising up to 10 K Erlang VMs.

Index Terms—Scalability, reliability, actors, Erlang

1 Introduction

Changes in hardware manufacturing technologies are driving systems to include ever more cores. Servers comprising hundreds of commodity hosts with tens of thousands of cores in total are becoming commonplace. Experience with high performance and data centre computing shows that reliability is critical at these scales, e.g., host failures alone account for around one failure per hour on commodity servers with approximately 10^5 cores [1].

Distributed actor languages and frameworks like Erlang [2], [3] or Scala/Akka [4] are proven technologies for reliable scalable computing. The key innovation in actor languages and frameworks is to isolate state: that is, actors do not share state with each other but rather exchange information using asynchronous messages.

In Erlang actors are called processes, and their isolated state means that they may fail with minimal disruption to concurrent processes. Moreover one process may *supervise* other processes, detecting failures and taking remedial action, e.g., restarting the failed process. *Distributed Erlang* deploys processes across multiple Erlang Virtual Machines (VMs or nodes) potentially on different hosts. In distributed Erlang fault tolerance is provided by global process registration, where the name of a server process is registered, and if the process fails a new server process can be spawned and re-associated with the name. This allows systems to adopt a "let it crash" philosophy, where a process is written to deal with the common error-free case, and failure is handled by the supervising process.

While distributed Erlang conceptually supports the engineering of scalable reliable systems, in practice it has some scalability limits that force developers to depart from the standard language mechanisms when programming at scale [5]. Scalability is limited by two main factors. First, maintaining a fully connected mesh of Erlang nodes means that a system with n nodes must maintain $O(n^2)$ active TCP/IP connections and this induces significant network traffic above 40 nodes. Second, maintaining a global process namespace incurs high synchronisation and communication costs (Section 3).

In prior work we have addressed these scalability issues by providing a Scalable Distributed (SD) Erlang library that partitions the network of Erlang nodes into scalability groups (s_groups). An s_group reduces the number of connections a node maintains by supporting full mesh connections only to other nodes within the s_group, and pairwise connections to nodes outside the s_group. S_groups reduce the amount of global information, as process names can be registered only in the nodes of the s_group, rather than

- *N. Chechina, K. MacKenzie, P. Trinder, A. Ghaffari, and M. Moro Hernandez are with the University of Glasgow, Glasgow G12 8QQ, United Kingdom. E-mail: {Natalia.Chechina, Phil.Trinder, Amir.Ghaffari}@glasgow.ac.uk, kwxm@inf.ed.ac.uk, kakotamix@hotmail.com.*
- *S. Thompson is with the University of Kent, Canterbury CT2 7NZ, United Kingdom. E-mail: s.j.thompson@kent.ac.uk.*
- *O. Boudeville is with EDF R&D, Clamart 92141, France. E-mail: olivier.boudeville@edf.fr.*
- *V. Fördős and C. Hoch are with Erlang Solutions AB, Budapest 1093, Hungary. E-mail: {viktoria.fordos, csaba.hoch}@erlang-solutions.com.*

Manuscript received 25 Apr. 2016; revised 16 Dec. 2016; accepted 3 Jan. 2017. Date of publication 17 Jan. 2017; date of current version 14 July 2017. Recommended for acceptance by A. Benoit.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2654246

globally. We discuss the motivation for, design of, and implications of s_groups further in Section 3.

This paper presents the first comprehensive evaluation of SD Erlang s_groups and associated tool support, together with guidance about how those constructs and tools can best be used. We argue that s_groups preserve the distributed Erlang approach to reliability while improving scalability. We start by outlining the Orbit, ACO and IM benchmarks and the substantial (approx. 150 K lines of code) Simulation of Discrete Systems of All Scales (Sim-Diasca) case study used to evaluate SD Erlang and demonstrate how the tools are used (Section 4), before presenting the primary research contributions as follows.

(1) We present a systematic and comprehensive evaluation of the scalability and reliability of SD Erlang (Section 5). We measure three benchmarks and the case study on several platforms, but the primary platform is a cluster with up to 256 hosts and 6144 cores.

The benchmarks evaluate different aspects of SD Erlang: Orbit evaluates the scalability impact of transitive network connections, ACO evaluates the scalability impacts of both transitive connections and maintaining a global namespace for reliability, and IM targets reliability. The experiments cover three application-specific measures of scalability: speedup for Orbit, weak scaling for ACO, and throughput for IM.

Crucially we show that s_groups improve the scalability of both reliable and unreliable distributed Erlang applications (Section 5.2); and use Chaos Monkey [6] to show that SD Erlang preserves Erlang's leading reliability model (Section 5.4). While some scalability and reliability results for the ACO and Orbit benchmarks have been reported in a paper that outlines all of the results of the RELEASE project [7], this paper focuses on SD Erlang and describes 9 additional experiments with it, provides a more detailed analysis, and additional evidence from the IM benchmark and the Sim-Diasca case study to support the conclusions.

(2) We present guidance for the construction of SD Erlang systems, through a set of questions that identify key design decisions; these support construction of SD Erlang systems from scratch as well as for refactoring distributed Erlang applications into SD Erlang. This approach is built a suite of new or improved tools for monitoring, debugging, deploying and refactoring SD Erlang applications (Section 6).

(3) We demonstrate the capability of the tools, for example showing that WombatOAM is capable of deploying and monitoring substantial (e.g., 10 K Erlang VMs) distributed Erlang and SD Erlang applications with negligible overheads (Section 6.8).

2 Related Work

2.1 Scalable Reliable Programming Models

There is a plethora of shared memory concurrent programming models like PThreads or Java threads, and some models, like OpenMP [8], are simple and high level. However, synchronisation costs mean that these models generally do not scale well, often struggling to exploit even 100 cores. Moreover, reliability mechanisms are greatly hampered by the shared state: for example, a lock becomes permanently unavailable if the thread holding it fails.

The High Performance Computing (HPC) community build large-scale (10^6 core) distributed memory systems using the *de facto* standard MPI communication libraries [9]. Increasingly these are hybrid applications that combine MPI with OpenMP. Unfortunately MPI is not suitable for producing general purpose concurrent software as it is too low level with explicit, synchronous message passing. Moreover the most widely used MPI implementations offer no fault recovery: if any part of the computation fails, the entire computation fails. Currently, the issue is addressed by using what is hoped to be highly reliable computational and networking hardware, but there is intense research interest in introducing reliability into HPC applications [10].

Server farms use commodity computational and networking hardware, and often scale to around 10^5 cores, where host failures are routine. They typically perform rather constrained computations, e.g., big data analytics, using reliable frameworks like Google MapReduce [11] or Hadoop [12]. The idempotent nature of the analytical queries makes it relatively easy for the frameworks to provide implicit reliability: queries are monitored and failed queries are simply re-run. In contrast, actor languages like Erlang are used to engineer reliable general purpose computation, often recovering failed stateful computations.

2.2 Actor Languages

The actor model of concurrency consists of independent processes communicating by means of messages sent asynchronously between processes. A process can send a message to any other process for which it has the address (in Erlang the "process identifier" or pid), and the remote process may reside on a different host. The notion of actors originated in artificial intelligence [13], and has been used widely as a general metaphor for concurrency, as well as being incorporated into a number of niche programming languages in the 1970s and 80s. More recently it has come back to prominence through the appearance not only of multicore chips but also larger-scale distributed programming in data centres and the cloud.

With built-in concurrency and data isolation, actors are a natural paradigm for engineering reliable scalable general-purpose systems [14]. The model has two main concepts: *actors*, which are the unit of computation, and *messages*, which are the unit of communication. Each actor has an *address-book* that contains the addresses of all the other actors it is aware of. These addresses can be locations in memory, or direct physical attachments, or network addresses. In a pure actor language messages are the only way for actors to communicate.

After receiving a message an actor can do the following: (i) send messages to another actor in its address-book, (ii) create new actors, or (iii) designate a behaviour to handle the next message it receives. The model does not impose any restrictions in the order in which these actions must be taken. Similarly, two messages sent concurrently can be received in any order. These features enable actor based systems to support indeterminacy and quasi-commutativity, while providing locality, modularity, reliability and scalability [14].

Erlang [2], [3] is widely used to develop reliable and scalable production systems, initially with its developer Ericsson and then more widely through open source adoption. There are now actor frameworks for many other languages; these include Akka for C#, F# and Scala [15], CAF [16] for C++, Pykka [17], Cloud Haskell [18], PARLEY [19] for Python and Termite Scheme [20], and each of these is currently under active use and development. Moreover, the recently defined Rust language [21] has a version of the actor model built in, albeit in an imperative context.

2.3 Reliability in Distributed Erlang Systems

Erlang was designed to solve a particular set of problems, namely those in building telecommunications systems, where systems need to be scalable to accommodate hundreds of thousands of calls concurrently, in soft real-time. These systems need to be highly-available and reliable: i.e., to be robust in the case of failure, which can come from software or hardware faults. Given the unavoidability of the latter, Erlang also adopts the "let it crash" philosophy in error situations, and uses the supervision mechanism, discussed shortly, to handle all kinds of faults.

Scaling in Erlang is provided in two different ways. It is possible to scale *within a single node* by means of the multicore Erlang VM which exploits the concurrency provided by the multiple cores. It is also possible to scale *across multiple hosts* using multiple distributed Erlang nodes. In this paper we only address the latter.

Reliability in Erlang is multi-faceted. As in all actor languages each process has private state, preventing a failed or failing process from corrupting the state of other processes. Messages enable stateful interaction, and contain a deep copy of the value to be shared, with no references (e.g., pointers) to the senders' internal state. Moreover, Erlang avoids type errors by enforcing strong typing, albeit dynamically [2]. Connected 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*. In a multi-node system the tree may span multiple nodes.

A global namespace maintained on every node maps *process names* to pids to provide reliable distributed service registration, and this is what we mean when we talk about a *reliable* system: it is one in which a named process in a distributed system can be restarted without requiring the client processes also to be restarted (because the name can still be used for communication).

To see global registration in action, consider spawning a server process on an explicitly identified node (`some_node`) and then globally registering it using `some_server` name

```
RemotePid = spawn(some_node, fun () ->
                  some_module:some_fun() end),
global:register_name(some_server,RemotePid).
```

Clients of the server process can send messages to the registered name, e.g.,

```
global:whereis_name(some_server) ! ok.
```

If the server fails the supervisor can spawn a replacement server process with a new pid and register it with the same name (`some_server`). Thereafter client messages addressed to the `some_server` name will be delivered to the new server process. We discuss the scalability limitations of maintaining a global namespace further in Section 3.1.

3 Scaling Distributed Erlang

In this section we first discuss the scaling limitations of distributed Erlang (Section 3.1) and currently adopted ad hoc solutions (Section 3.2). Then we provide an overview of the SD Erlang approach to scaling distributed systems called s_groups (Section 3.3).

3.1 Scalability Issues

The scalability limitations of distributed Erlang are mainly due to the two mechanisms that support fault tolerance, namely transitive connections and global processes registration [5], [7]. Global name registration depends on the transitivity; therefore, when the latter is disabled with the `-connect_all false` flag, global name registration is not available either.

Transitive connectivity connects all normal (not hidden) nodes in the system. This happens "under the hood" and the information about live and lost connections is kept up-to-date. As a result the system can avoid sending messages to, or expecting messages from, disconnected nodes and automatically adjust to the changed number of nodes. Therefore, apart from fault tolerance, transitivity also provides elasticity, i.e., seamless growth or contraction of the number of nodes in the system. However, full connectivity means that the total number of connections in the system is $n(n-1)/2$, and every node supports $(n-1)$ connections. In addition every connection requires a separate TCP/IP port, and node monitoring is achieved by periodically sending heartbeat messages. In small systems maintaining a fully connected network is not an issue, but when the number of nodes grows a fully connected network requires significant resources becoming a burden that worsens the performance.

Global process registration enables one to associate a process with a name and replicate this information on all transitively connected nodes creating a common namespace. However, as the number of nodes or the failure rate of registered processes grow, global name registration has a significant impact on network scalability [5].

3.2 Ad Hoc Approaches

A straightforward approach to eliminating scalability limitations caused by transitive connections and global process registration is to disable transitive connectivity; currently, this is the main approach used in industrial applications [22]. However, when adapting this approach, either applications lose the fault-tolerance that comes with transitivity and shared namespace, or developers introduce their own libraries which provide features resembling transitivity and shared namespace but restricted to a particular connectivity mechanism. These libraries usually have very limited reusability due to the mechanisms being specialised for a particular application.

Another approach is to use Erlang/OTP global_groups that partition a set of nodes. Nodes within the global_group have transitive connections, and non-transitive connections with nodes outside of the global_group. Each global_group has its own namespace. The drawback is that the approach is limited to the cases when the network can be partitioned. As a result although global_groups are available in Erlang/OTP they are not widely used.

3.3 SD Erlang and S_groups

Scalable Distributed Erlang (SD Erlang) is a conservative extension of distributed Erlang. It was introduced to provide a reusable solution that overcomes scalability limitations posed by both transitive connectivity, global namespace, and a lack of resource awareness, while preserving fault tolerance mechanisms of distributed Erlang. This was achieved by introducing two new libraries: (1) `attribute` that provides semi-explicit process placement [23], and `s_group` that partitions the node connection graph into s_groups [5]. SD Erlang has been available with several releases of Erlang/OTP, and is likely to remain available in the medium term as the Erlang/OTP group at Ericsson indicate no near future plans to change the mechanisms that the `s_group` libraries rely on.

To reduce the number of connections and the size of the namespace, nodes are grouped into s_groups. Nodes in s_groups have transitive connections with nodes from the same s_group , non-transitive connections with other nodes, and each s_group has its own namespace. The s_groups do not partition the set of nodes, i.e., a node can belong to many s_groups facilitating the construction of connection topologies appropriate for different application needs. For example, nodes can be assembled into hierarchical s_groups, where communication between nodes from different s_groups occurs only via gateway nodes. To provide fault-tolerance s_groups may have two or more gateway nodes.

S_group functionality is supported by 15 functions, eight of which manipulate s_groups, including dynamic creation of new s_groups (`s_group:new_s_group/1,2`) and getting information about known s_groups (`s_group:s_groups/0`), and the remainder manipulate names registered in the groups, like registering a name (`s_group:register_name/2`) and getting information about all names registered in a particular s_group (`s_group:registered_names/2`). More details can be found in [5].

For example, the following function creates an s_group called `some_s_group` that consists of three nodes:

```
s_group:new_s_group(some_s_group,
      [some_node, some_node_1, some_node_n]).
```

To register a name, we provide both a pid and also the name of the s_group in which we want to register that name;

```
s_group:register_name(some_s_group,
                        some_server, RemotePid).
```

The s_group name is also required when sending a message to a process using its name:

```
s_group:whereis_name(some_s_group,
                               some_server) ! ok.
```

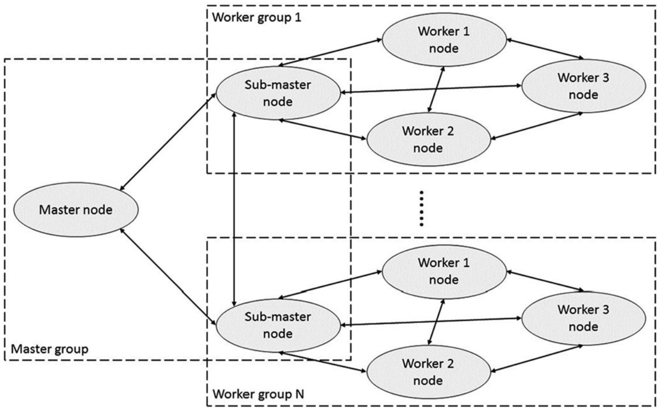


Fig. 1. Communication model in SD Erlang orbit.

4 Benchmarks and a Case Study

To evaluate the performance of SD Erlang we use three benchmarks, named Orbit (Section 4.1), Ant Colony Optimisation (ACO, Section 4.2), and Instant Messenger (IM, Section 4.3), and a case study (Sim-Diasca, Section 4.4). The benchmark code is open source at https://github.com/release-project/benchmarks.

Each benchmark corresponds to a typical class of Erlang applications as follows. Orbit employs a Distributed Hash Table (DHT) of the type used in replicated form in No-SQL DBMS like Riak [24], implemented in Erlang. ACO is a large search, similar to the Erlang Sim-Diasca simulation framework [25]. IM is a simplified version of a very typical internet-scale Erlang application, namely a chat service like WhatsApp [26].

4.1 Orbit

Orbit is a generalization of a transitive closure computation, and is a common pattern in algebraic computations [27]. To compute the *Orbit* of an element x_0 in a given space $[0..X]$, a number of *generator functions* $g_1, g_2, \ldots, g_n$ are applied to x_0, obtaining new elements $x_1, \ldots, x_n \in [0..X]$. The generator functions are applied to the new elements until no new element is generated.

The computation is initiated on the master node. Then processes are spawned to worker nodes to explore the space and populate the DHT maintained by the worker nodes. In the distributed Erlang version of Orbit the master node is directly connected to the worker nodes, so it is enough to hash x_n once to determine the target node that keeps the corresponding part of the DHT. In the SD Erlang version of Orbit the worker nodes are partitioned (Fig. 1) and the master node functionality is shared with the submaster nodes which also perform the routing between the s_groups; therefore, to define the target node, x_n is hashed twice-the first hash defines the s_group, and the second hash defines the worker node in that s_group.

4.2 Ant Colony Optimisation

Ant Colony Optimisation [28] is a metaheuristic which is used for solving combinatorial optimisation problems. Our implementation is specialised to solve a scheduling problem called the Single Machine Total Weighted Tardiness Problem [29]. In the basic single-colony ACO algorithm, a

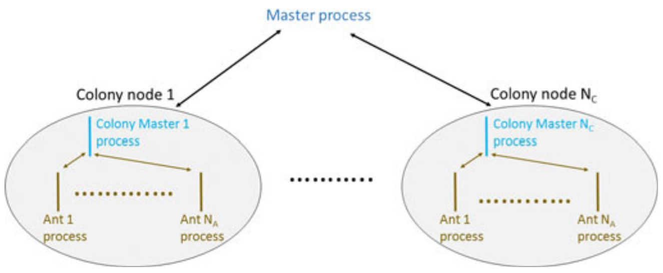


Fig. 2. Two-level distributed ACO.

number of agents called *ants* independently construct candidate solutions guided by problem-specific heuristics with occasional random deviations influenced by a structure called the *pheromone matrix* which contains information about choices of paths through the solution space which have previously led to good solutions. After all of the ants have produced solutions, the best solution is selected and used to update the pheromone matrix. A new generation of ants is then created which constructs new solutions guided by the improved pheromone matrix, and the process is repeated until some halting criterion is satisfied: in our implementation, the criterion is that some fixed number of generations have been completed. The algorithm is naturally parallelisable, with one process for each ant in the colony. Increasing the amount of parallelism (i.e., the number of ants) does not lead to any speedup, but does lead to an improvement in the *quality* of the solution.

In the distributed setting, one can have several colonies (in our implementation, one colony per Erlang node) which occasionally share pheromone information. In addition to increasing the number of ants exploring the solution space, distribution also gives the possibility of having colonies with different parameters: for example, some colonies might have more randomness in their search, making it easier to escape from locally-optimal solutions which are not globally optimal.

We have implemented four variations on the multi-colony ACO algorithm. In each of these, the individual colonies perform some number of *local iterations* (i.e., generations of ants) and then report their best solutions; the globally-best solution is then selected and is reported to the colonies, which use it to update their pheromone matrices. This process is repeated for some number of *global iterations*. Our four versions are as follows.

Two-Level ACO (TL-ACO). There is a single master node which collects the best solutions from the individual colonies and distributes the overall best solution back to the colonies. Fig. 2 depicts the process and node placements of the *TL-ACO* in a cluster with N_C nodes. The master process spawns N_C colony processes on available nodes. In the next step, each colony process spawns N_A ant processes on the local node. In the figure, the objects and their corresponding captions have the same colour. As the arrows show, communications between the master process and colonies are bidirectional.

Multi-Level ACO (ML-ACO). In TL-ACO, the master node receives messages from *all* of the colonies, and thus could become a bottleneck. ML-ACO addresses this by having a tree of submasters (Fig. 3), with each node in the bottom level collecting results from a small number of colonies. These are then fed up through the tree, with nodes at higher

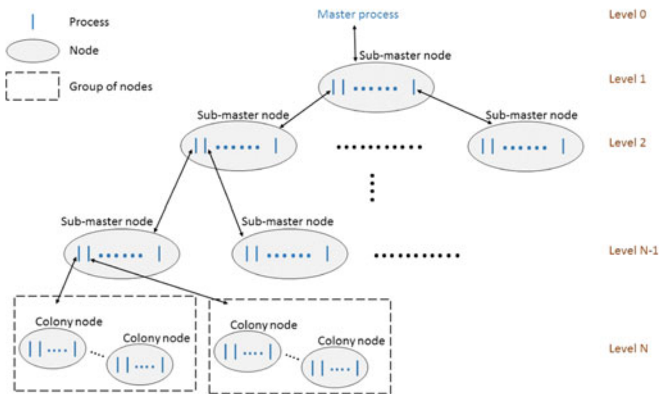


Fig. 3. Multi level distributed ACO.

levels selecting the best solutions from among a number of their children.

Globally Reliable ACO (GR-ACO). This adds fault-tolerance using functions from Erlang's `global` module. In ML-ACO, if a single colony fails to report back the whole system will wait indefinitely. GR-ACO adds supervision so that faulty colonies can be detected and restarted, allowing the system to continue its execution.

Scalable Reliable ACO (SR-ACO). This also adds fault-tolerance, but using SD Erlang's s_groups instead of the `global` methods. In addition, in SR-ACO nodes are only connected to the nodes in their own s_group.

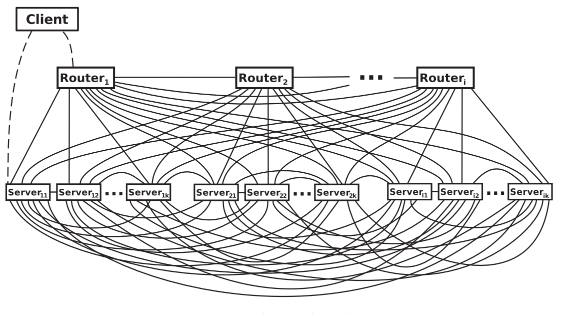
4.3 Instant Messenger

An Instant Messenger is a server application where clients exchange messages via servers like WhatsApp [26]. The IM requirements include both non-functional aspects such as *scalability*, *reliability*, *network topology* or *security* and functional aspects on how the service must be delivered and how the different entities interact with each other [30], [31]. In general, an IM server application provides two services: presence and instant messaging. The first allows the clients to be aware of their contacts' status, whereas the second enables the exchange of messages with other clients. Although we have implemented the IM benchmark only to evaluate scalability of a typical distributed Erlang application, the server part meets all of the requirements above apart from security and encryption [32]. We have implemented the following two versions.

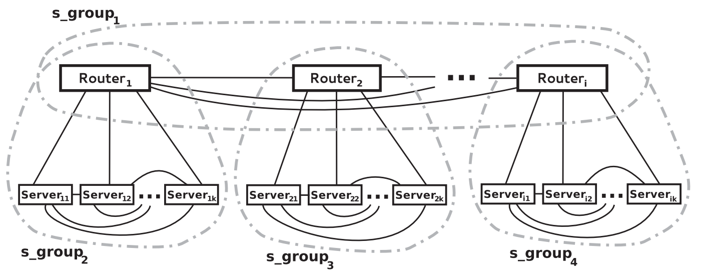
Reliable Distributed IM (RD-IM) is implemented in distributed Erlang where reliability is supported by supervision, global name registration of all processes that maintain databases, and replication of these databases. Therefore, when a database process fails it is restarted by a corresponding supervising process (Fig. 5). The table is then populated from a copy kept on a different node. In the meantime the data can be obtained from a replica. After the recovery process finishes the table can be accessed using the same globally registered name.

Reliable Scalable Distributed IM (RSD-IM). Is implemented in SD Erlang with the same reliability mechanism as in RD-IM. The only difference is that in RSD-IM the names are registered in corresponding s_groups rather than globally.

Fig. 4 compares the node connection in RD-IM and RSD-IM. Connections to the client nodes in both versions are hidden and non-transitive to facilitate comparison of the



(a) Distributed Erlang



(b) SD Erlang

Fig. 4. Connections in IM Server.

distributed and SD Erlang servers. Unlike the previous two case studies the IM benchmark was specifically designed to evaluate SD Erlang fault tolerance in comparison with the distributed Erlang one. For that, we have implemented the `rhesus` module to terminate various processes, and performed minimal refactoring such that the two versions even have identical supervision trees (Fig. 5).

The rhesus module (named after the rhesus macaque) is a modification of Chaos Monkey [6] that we designed specifically to accommodate requirements of the IM application. For example, the Erlang/OTP version of Chaos Monkey [33] does not differentiate between the types of process which means that all processes have equal chances to be terminated. Since the most frequent processes in the IM are `client monitors` and `chat sessions` the rest of the processes have very low chances of termination, which is not suitable for the current study. To overcome that we have introduced weighted termination probability for all types of IM processes and user-defined termination time.

The traffic for client communication is provided by traffic generators that simulate real-life conversations and associated parameters, such as time to type messages and lengths of conversations [34]. A traffic generator randomly picks two clients A and B, and sends client A a request to start a conversation with client B. Then client A generates a random string and sends the message to client B. When client B receives the message, it generates a reply message (again a random string), simulates the typing time, and sends the message to client A. The number of messages in the conversation is random, and defined at the beginning of the conversation.

4.4 Sim-Diasca

Simulation of Discrete Systems of All Scales is a discrete simulation engine developed by EDF R&D in Erlang. Sim-Diasca has been available as free software since 2010 under the GNU LGPL licence [25]. Its objectives are to evaluate correctly the models involved in a simulation, and preserve key properties like causality, total reproducibility and some kind of "ergodicity", i.e., a fair exploration of the possible outcomes of the simulation.

Using the requested simulation frequency, Sim-Diasca splits the simulated time into a series of time steps, automatically skipping those that can be omitted, and re-ordering the inter-model messages so that properties like reproducibility are met. Causality resolution requires time steps to be further divided into as many logical moments (called *diascas*) as needed. During a given diasca, all model instances that must be scheduled are evaluated concurrently. However, this massive parallelism can only occur between two (lightweight) distributed synchronisations.

The *City* simulation example (approx. 10 K lines of Erlang code) has been designed to provide an open, shareable, tractable yet representative use case of Sim-Diasca for benchmarking purposes. As Sim-Diasca is a simulation engine we need to define a simulation instance to create a benchmark. The simulation attempts to represent a few traits of a city, such as a waste management and the weather system.

The City example is potentially arbitrarily scalable in terms of both duration and size: there are bounds neither to the duration in virtual time during which the target city can be evaluated, nor to its size. Therefore, the City example can be used to benchmark arbitrarily long and large simulations, reflecting the typical issues that many real-world simulations exhibit, such as sequential phases and new bottlenecks as the scale increases.

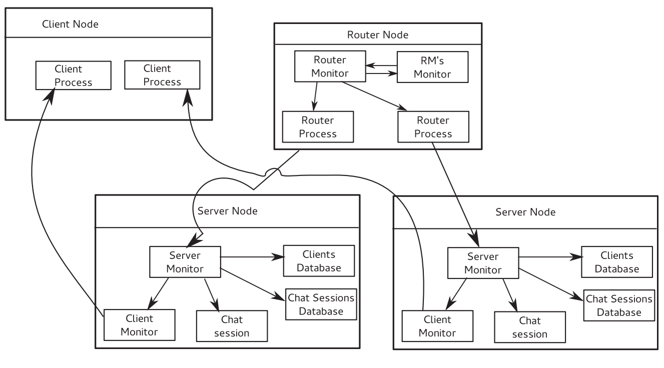


Fig. 5. IM Supervision Hierarchy. The Arrows Indicate Supervision.

5 Performance Evaluation

In this section we first provide a brief overview of the clusters we used in the experiments (Section 5.1), and then address the following research questions by measuring the performance of the benchmarks discussed in Section 4. *RQ1* Does SD Erlang scale better than distributed Erlang for applications with no reliability requirement (Section 5.2), and if so why (Section 5.3)? *RQ2* Can SD Erlang preserve the Erlang supervision-based reliability model despite partitioning the set of nodes (Section 5.4)? *RQ3* Does SD Erlang scale better than distributed Erlang for reliable applications (Section 5.2), and if so why (Section 5.3)?

The benchmarks evaluate different aspects of s_groups: Orbit evaluates the scalability impacts of network connections, ACO evaluates the impact of both network

TABLE 1
Cluster Specifications

Name	Hosts	Cores per host	Max hosts (cores) avail.	Hardware	RAM per host, GB	Inter-connect.
GPG	20	16	20 (320)	Intel Xeon E5-2640v2 8C, 2 GHz	64	10 GB Ethernet
Kalkyl	384	8	176 (1,408)	Intel Xeon 5520v2 4C, 2.26 GHz	24–72	InfiniBand 20 Gb/s
TinTin	160	16	140 (2,240)	AMD Opteron 6220v2 Bulldozer 8C, 3.0 GHz	64–128	2:1 over-subscribed QDR InfiniBand
Athos	776	24	256 (6,144)	Intel Xeon E5-2697v2 12C, 2.7 GHz	64	InfiniBand FDR14

connections and the global namespace required for reliability, and IM targets reliability.

Moreover the experiments cover three measures of scalability. As Orbit does a fixed size computation, the scaling measure is relative speedup (or strong scaling), i.e., speedup relative to execution time on a single core. As the work in ACO increases with the compute resources, weak scaling is the appropriate measure. As IM is a messaging system, scalability is measured as maximum throughput (messages per minute).

The experiments are conducted using Erlang/OTP 17.4 and its SD Erlang modification. Complete descriptions of all experiments on the benchmarks and case study are available in [35], [36].

5.1 Cluster Specifications

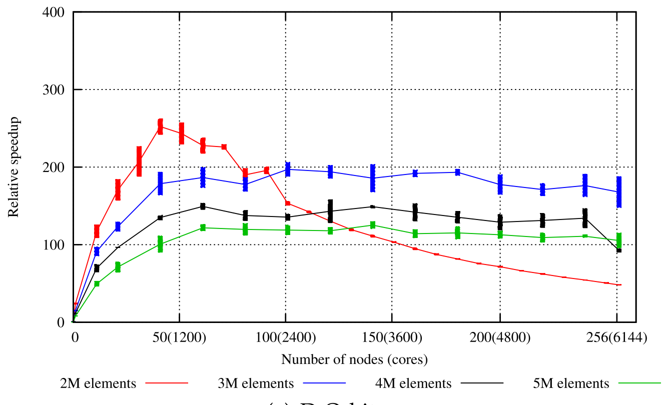
The primary experiments presented in this paper were conducted on the following two clusters: Athos (EDF, France) and GPG (Glasgow University, UK). Additional experiments on Kalkyl and TinTin clusters (Uppsala University, Sweden) presented elsewhere [35] confirm the results presented here. The configuration of the clusters is provided in Table 1.

5.2 Scalability

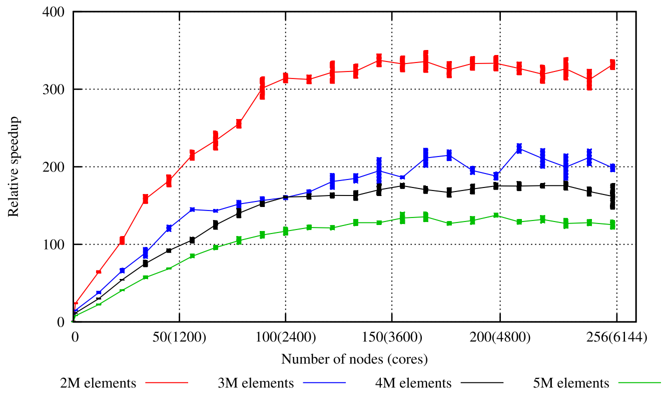
The scalability evaluation is conducted on the Athos cluster (Table 1). To run the experiments we had simultaneous access to up to 256 nodes (6,144 cores) for up to 8 hours at a time.

5.2.1 *Orbit*

In the implementation of Orbit no global operations were used which means the main difference between distributed Erlang (D-Orbit) and SD Erlang (SD-Orbit) versions of Orbit is due to the number of connections maintained by nodes. Assume the total number of nodes is N. Then D-Orbit has one master node and $N-1$ worker nodes; therefore, every node maintains $N-1$ connections. SD-Orbit also has 1 master node but in addition it has S submaster nodes. If we assume that all s_groups have the same number of worker nodes (Fig. 1), then the number of worker nodes per s_group is $k=(N-1)/S-1$, where N and S are such that $k \in \mathbb{N}_{>0}$. Therefore, the number of connections maintained by the nodes is as follows: the master node maintains S



(a) D-Orbit



(b) SD-Orbit

Fig. 6. Orbit relative speedups in SDErl-17.4.

connections, every submaster node maintains $(N-1)/S+S-1$ connections, and every worker node maintains $(N-1)/S-1$ connections.

Fig. 6 shows D-Orbit and SD-Orbit speedup depending on the number of orbit elements, which vary between $2 \cdot 10^6$ and $5 \cdot 10^6$. The speedup is a ratio between execution time on one node with one core and the execution time on the given number of nodes and cores. The execution time does not include the time required to start the nodes but only the time taken by the Orbit calculation. For each of the experiments we plot standard deviation. The speedup results show that as we increase the number of nodes the performance of D-Orbit first grows but then starts degrading (Fig. 6a). This trend is not observed in the corresponding SD-Orbit experiments (Fig. 6b). In addition when we increase the number of orbit elements beyond $5 \cdot 10^6$, D-Orbit fails due to the fact that some VMs exceed the available RAM of 64 GB. However, we did not experience this problem when running SD-Orbit experiments even with $60 \cdot 10^6$ orbit elements.

5.2.2 *ACO*

The ACO community commonly evaluates the quality of ACO implementations using a set of benchmarks whose optimal solutions are known, and then runs a program on them for some fixed number of iterations and observes how close the program's solutions are to the optimal ones. We apply the strategy used in [37], [38] to our two-level ACO.

We ran the TL-ACO application on 25 hard SMTWTP instances of size 100 described in [39, 3.2], gradually increasing the number of colonies, each on a different node, from

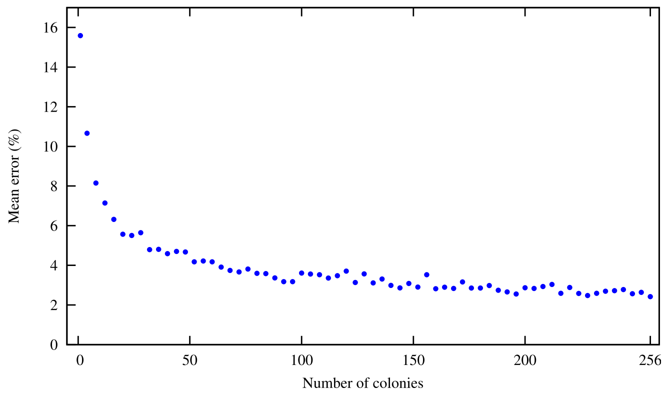


Fig. 7. ACO mean error.

1 to 256. Fig. 7 shows the mean difference in cost between our solutions and the optimal solutions: it is clear that increasing the number of colonies increases the quality of solutions, although the trend is not strictly downwards because the random nature of the ACO algorithm means that repeated runs with the same input may produce different solutions.

Weak scaling is the appropriate performance measure for ACO where the amount of work increases with the compute resources to gain improved solutions. Fig. 8 shows the weak scaling of ACO, plotting mean execution time against the number of colonies, and hence hosts. We see an upward trend due to increasing amounts of communication and the increasing time required to compare incoming results. This is typical of the scaling graphs for the ACO application.

Fig. 9 compares the scalability of the TL-ACO, multi-level ACO, globally reliable ACO and scalable reliable ACO versions, executing each version with 1, 10, 20, ..., 250 compute nodes; for each number of nodes we recorded the execution time for seven runs, and plotted the mean times for these runs. There is some variation in execution time, but this is typically only about 2-3 percent around the mean, so we have reduced clutter in the plots by omitting it. The execution times here were measured by the ACO program itself, using Erlang's `timer:tc` function, and they omit some overhead for argument processing at the start of execution.

We see that ML-ACO performs slightly better than TL-ACO and the performance of GR-ACO is significantly worse than both of these. The performance of SR-ACO is considerably better than all the other versions.

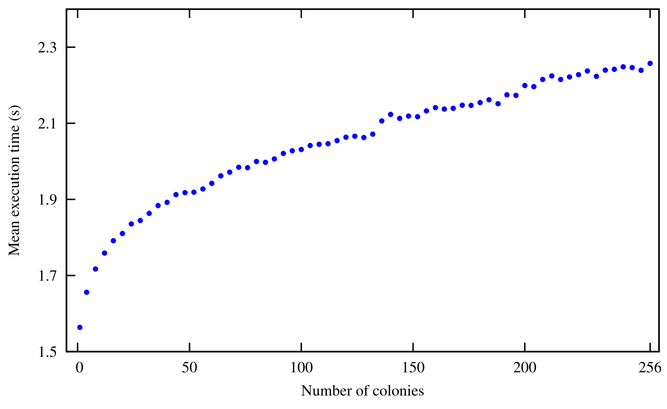


Fig. 8. ACO weak scaling to 256 hosts, 6,144 cores.

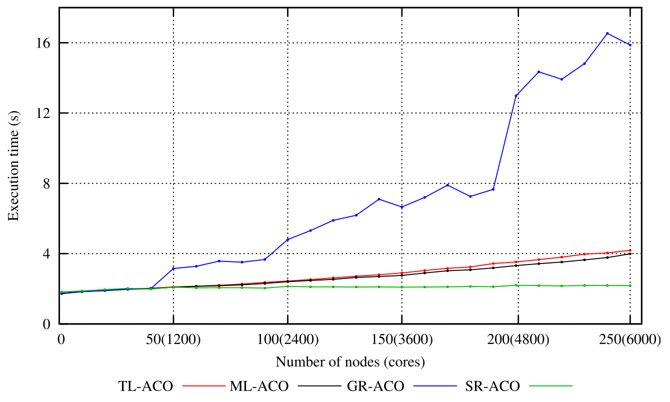


Fig. 9. Weak scaling of ACO versions.

These results are as we would expect. GR-ACO uses global name registration, which is known to limit scalability. TL-ACO uses a single master node which collects messages from all of the worker nodes, and this can cause a bottleneck. ML-ACO eliminates this bottleneck by introducing a hierarchy of submasters to collect results. Both TL-ACO, ML-ACO, and GR-ACO use Erlang's default distribution mechanism, where every node is connected to every other one even if there is no communication between the nodes. In SR-ACO we use SD-Erlang's s_groups to reduce the number of connections and the namespace, and we attribute SR-ACO's superior performance to this fact.

5.3 Network Traffic

To investigate the reason for the improved scalability we measure the impact of s_groups on network traffic. We measure the number of sent and received packets on the GPG cluster for three versions of ACO: ML-ACO, GR-ACO, and SR-ACO. Fig. 10 shows the total number of received packets. The highest traffic (the blue line) belongs to GR-ACO and the lowest traffic belongs to SR-ACO (green line). This shows that SD Erlang significantly reduces the network traffic between Erlang nodes. Even with the s_group name registration SR-ACO has less network traffic than ML-ACO, which has no global name registration. This difference becomes more significant as the number of nodes grows. For example, on 145 nodes, in SR-ACO there were 500,000 packets received whereas in ML-ACO and GR-ACO the number of received packets is two and three times larger respectively.

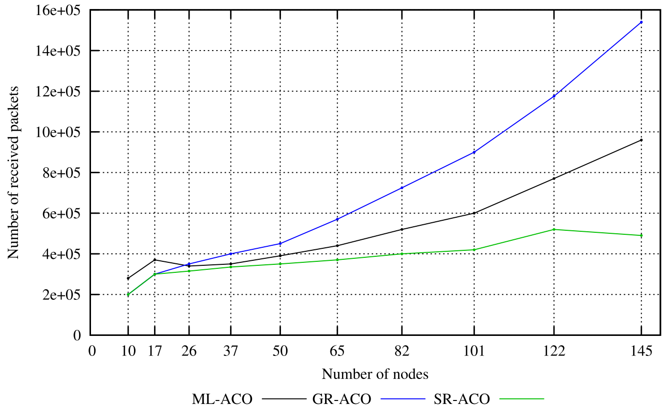


Fig. 10. Number of received packets in ML-ACO, GR-ACO, and SR-ACO (GPG cluster).

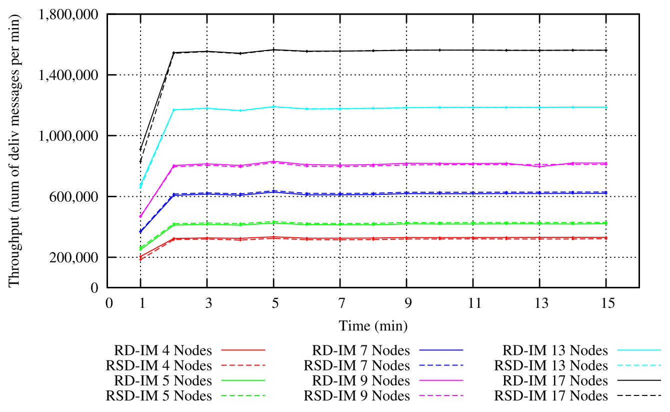


Fig. 11. RD-IM and RSD-IM throughput without failures (GPG cluster).

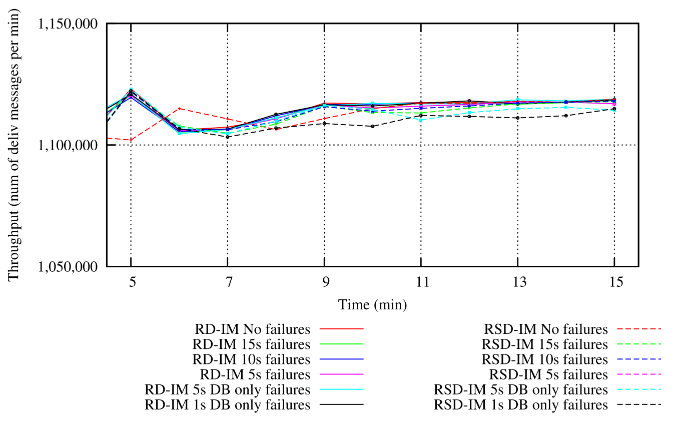


Fig. 12. RD-IM and RSD-IM throughput with and without failures (13 GPG cluster nodes).

5.4 Reliability

In this section we evaluate an impact of failures on performance of distributed Erlang and SD Erlang applications using the IM (Section 5.4.1) and ACO (Section 5.4.2) benchmarks on the GPG cluster.

5.4.1 IM

To analyse an impact of failures in SD Erlang applications we first analyse whether there is a difference in throughput between the reliable versions of IM implemented in distributed Erlang (RD-IM) and SD Erlang (RSD-IM) when no failures occur. For that we vary the number of server nodes (3, 4, 6, 8, 12, 16) while maintaining just a single router node. Since RSD-IM has only one s_group, this setup results in identical architectures for both IM versions. The throughput measures the number of delivered messages per minute. The throughput results presented in Fig. 11 show that RD-IM and RSD-IM scale identically.

We then investigate the impact of failures and their rate on the performance of the RD-IM and RSD-IM applications. In the experiments we use two router nodes and 12 server nodes, making 14 nodes in total. In case of RSD-IM this results in three s_groups: one *router s_group* that consists of only two router nodes, and two *server s_groups* that consist of one router and six server nodes each. We first run experiments with no failures, then we terminate random processes, gradually reducing the rate from 15 and 5 seconds; finally we randomly terminate only globally registered database processes reducing the rate from 5 seconds to 1 second. The processes start failing five minutes into the benchmark execution once the applications are stable, i.e., failures occur only between minutes 5 and 15. The throughput results in Fig. 12 show that the IM fault tolerance is robust and the introduced failure rate has no impact on either of the of the IM versions in the given scale (number of nodes).

5.4.2 ACO

To evaluate an impact of failures in the ACO benchmark we ran Chaos Monkey against GR-ACO and SR-ACO. The fault tolerance here is mainly supported by globally registering master and submaster processes. Recall that in SR-ACO the processes are global only in their s_groups whereas in GR-ACO the processes are global to the whole system. The Chaos Monkey processes ran on every Erlang node (i.e., master, submasters, and colony nodes), periodically killing random Erlang processes. As in the IM experiments, the failures had no measurable impact on the runtime of either GR-ACO or SR-ACO. From the IM and ACO results we conclude that SD Erlang preserves the distributed Erlang reliability model.

6 Tools and System Development

How are scalable SD Erlang systems developed? This section outlines a general development strategy, from inception, though refactoring, performance tuning and operation. Here is not the place to rehearse the Erlang philosophy of concurrency-oriented programming, and the way that fault tolerance, robustness and distribution are integrated into the language and the OTP middleware layer: for more about this see [40], [41]. Instead, we look here at the twin issues of how to build systems in SD Erlang, and how to tune and operate those systems once built.

Fig. 13 gives a schematic view of an SD Erlang system, and the tools that are used in building and tuning it. Each node is an Erlang runtime, running on a multicore host machine; these nodes are grouped into (overlapping, generally) s-groups, each of which forms a fully interconnected network. In this section we first look at how tools support development of SD Erlang programs, and then cover tools for performance tuning and operation. Some of the tools are independent of SD Erlang, others were enhanced by us to support SD Erlang, and the rest were developed from scratch to support it.

6.1 Development Strategy

In designing an SD Erlang system the most important decision we need to make is *how to group nodes together:* nodes in a single s_group are all connected to each other, but

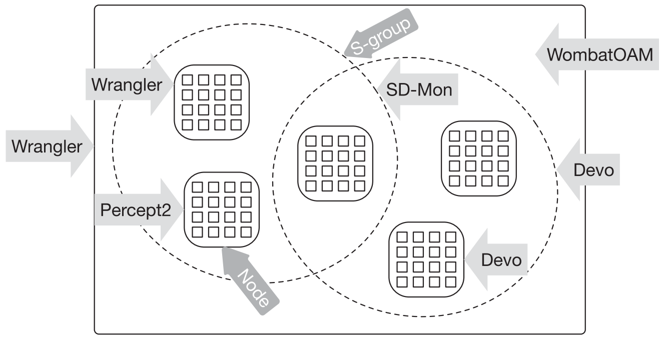


Fig. 13. Tool usage in development and performance optimisation.

connections between nodes can be added in an *ad hoc* way, and overlapping groups can also provide an implicit "routing" capability. This leads to three specific questions:

How should s_groups be structured? Depending on the reason the nodes are grouped-reducing the number of connections, or reducing the namespace, or both-s_groups can be freely structured as a tree, ring, or some other topology. One might also wish to have some free nodes that belong to no s_group (free nodes follow distributed Erlang rules), or to replicate nodes to improve fault-tolerance.

How do nodes from different s_groups communicate? S_groups do not impose any restrictions on the nodes in terms of establishing new connections. Therefore any node can communicate directly with any other node. However, to keep the number of connections minimal the communication between nodes from different s_groups can be routed via gateway nodes, i.e., nodes that belong to more than one s_group. S_groups do not provide any automatic routing mechanism, but we discuss in Section 6.2.2 how a particular generic mechanism can be identified and introduced.

How can one avoid the single point of failure of root/master nodes? To avoid overloading root (or master) nodes in hierarchically structured s_groups, it is advisable to introduce submaster nodes and processes that replicate some of these nodes' responsibilities.

A common development technique is refactoring, and Wrangler [42] is a mature refactoring tool for Erlang. Some existing features are very useful here and it has also been extended to support refactoring distributed Erlang programs into SD Erlang.

6.2 Refactoring with Wrangler

The process of refactoring distributed Erlang applications into SD Erlang applications is very much application specific. However, we identify and support two mechanisms: replacing global_groups with s_groups, and introducing and using a generic communication pattern.

6.2.1 From global_groups to s_groups

SD Erlang extends distributed Erlang by extending Erlang's original communication mechanism and replacing the `global_group` library with a new `s_group` library. As a result, Erlang programs using `global_group` will have to be refactored to use `s_group`. This kind of API migration problem is common as software evolves. To support such changes we extended Wrangler to migrate client code from using an old API to using a new one, with group migration, explained above, as a special case.

Our approach works this way: when an API function's interface is changed, the author of this API function implements an *adaptor function*, defining calls to the old API in terms of the new. From this definition we automatically generate the refactoring, that transforms the client code to use the new API. This refactoring can be supplied by the API writer to clients on library upgrade, allowing users to upgrade their code automatically.

As a design principle, we try to limit the scope of changes as much as possible, so that only the places where the 'old' API function is called are affected, and the remaining part of the code is unaffected. One could argue that the migration can be done by *unfolding* the function applications of the old API function using the adaptor function once it is defined; however, the code produced by this approach would be a far cry from what a user would have written. Instead, we aim to produce code that meets users' expectation. More details about Wrangler's support for API migration are reported in [43], which also presents a more complex API migration for a regular expression library.

6.2.2 Introducing a Generic Communication Pattern

In the first SD-Orbit implementation, described in Section 5.2.1, there is strong coupling of s_group manipulation and the application logic, making it difficult to separate the specific (application) from the generic (groups and communications). Using a general refactoring tool to support operations such as function renaming, function extraction and moving functions between modules, it was possible to separate out the generic portion, a *reusable s_group pattern*.

This reusable pattern provides (i) functions for setting up the s_group structure according to the pattern specified, (ii) functions for spawning gateway processes which are in charge of relaying messages from one s_group to another, and (iii) s_group-specific `send` and `spawn` functions.

With this generic component in place, it was possible to revisit the original D-Orbit code and transform it into SD Erlang with a simple set of refactorings, which set up the group structure and modify a number of functions: for instance, message send of the form `Pid!{vertex,X,Slot,K}` is transformed to a call to `central_grouping:s_group_send/2` with arguments `Pid` and `{vertex,X,Slot,K}`. These refactorings can themselves be automated using Wrangler's extension API.

The advantage extracting a generic component like this is that it allows developers more easily to use s_groups, and also to evolve the architecture of their systems more easily. More details of this process are explained in Deliverable 5.3 of the Release project [44].

6.3 Performance Methodology

To tune performance we are able to use conventional tools, such as unix *top* and *htop* (Section 6.4), as well as others that are specific to Erlang. These include Percept2 (Section 6.7) which can be used to optimise concurrent performance of systems on a single (multicore) node, SD-Mon and Devo (Sections 6.5 and 6.6), which are used to monitor and visualise the performance of a multinode system, and WombatOAM operations and maintenance framework (Section 6.8), that can be used to monitor, tune and adapt live systems.

Which Nodes Should be Grouped Together? Nodes in the same s_group maintain all-to-all connections and a common namespace. So we might want to put nodes in the same group because of, e.g., communication distances or frequency of communication between the nodes, or common node attribute, such as available hardware [23]. To assist this decision, it is possible to use Devo, which shows nodes' affinity (Section 6.6), and Percept2, which shows the communication between nodes (Section 6.7).

How Should the Size of s_groups be Determined? The size of an s_group depends on the intensity of the inter- and intra-s_group communication, and the number of global to s_group operations. The larger these parameters the smaller

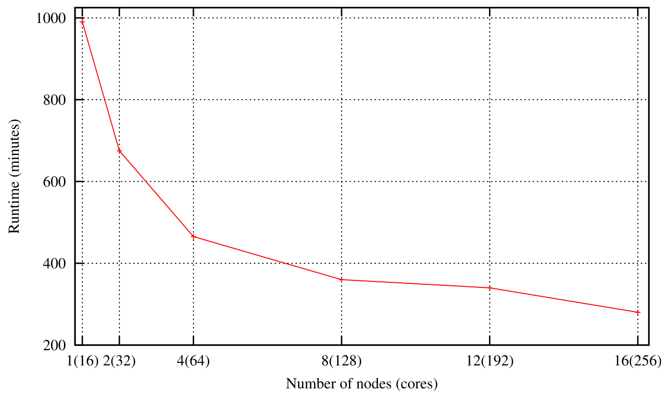


Fig. 14. Runtime. Sim-Diasca city example (GPG cluster).

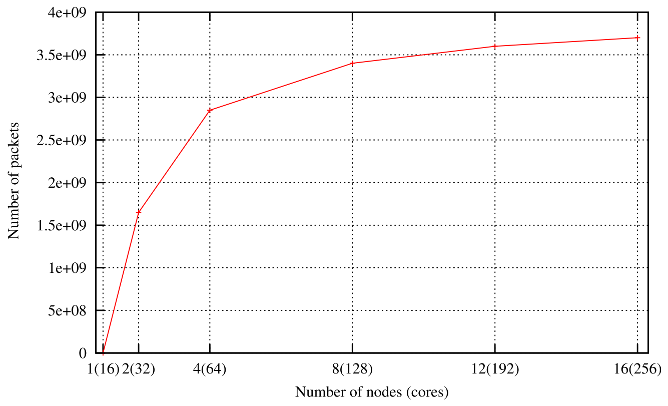


Fig. 15. Network traffic. Sim-Diasca city example (GPG cluster).

the number of nodes in the s_group should be. To analyse intra- and inter-s_group communications and determine these parameters the Devo (Section 6.6) and SD-mon (Section 6.5) tools can be used.

6.4 Existing System Tools

Here we discuss conventional tools using the example of the City instance of the Sim-Diasca case study. We run the `newsmall` scale of the City example that has two phases: *initialisation* and *execution*, but we exclude the former from our measurements. We employ standard Linux tools such as `top` and `netstat` to analyse core, memory, and network usage, and perform measurements on the GPG and Athos clusters described in Table 1.

To analyse scalability we compare the runtime of the Sim-Diasca City example at different GPG cluster sizes: 1, 2, 4, 8, 12, and 16 nodes, with 16 cores per node. Fig. 14 reports the runtime of the Sim-Diasca City example on up to 16 nodes (256 cores). The results show that the case study takes around 1,000 minutes on a single node, and below 300 minutes on 16 nodes. While the runtime of the Sim-Diasca instance continues to fall up to 16 nodes, the available resources are not utilised efficiently: we get a reasonable speedup of 1.5 on two nodes (32 cores), but it is only 2.2 on four nodes (64 cores), and degrades to a maximum of 3.45 on 16 nodes (256 cores).

The Linux `top` command is used to investigate core and memory usage. The maximum core and memory usage are 69 and 14 percent (8.96 GB out of 64 GB) respectively for a single 16-core node. The memory usage on a single host may become a bottleneck when running larger Sim-Diasca instances. As expected in a distributed system, both core and memory usage decrease as the number of nodes grow. Fig. 15 shows the network traffic, i.e., the number of sent and received packets between nodes in the cluster during the case study execution. The network traffic increases as the cluster size grows, while the number of sent and received packets are almost the same.

The results above also illustrate the common practice of tuning an application on a small cluster before moving to a larger and more expensive cluster. The range of tools available on small clusters is often greater than can be used through the batch queue interfaces on large clusters. Despite the issues revealed even at this modest scale we have, for completeness, repeated the experiments on the large Athos cluster and obtained similar results [35].

6.5 SD-Mon

The SD-Mon tool provides the scalable infrastructure to support off- and online monitoring of SD Erlang systems through a "shadow network" of nodes designed to collect, analyse and transmit monitoring data from active nodes and s_groups. SD-Mon can be used to understand the correct allocation of nodes in groups. At an initial stage, just by looking at the inter-node message flow, it can drive the initial grouping itself. After that it can be used to trim the network architecture and to monitor the system, revealing anomalies like intergroup messages bypassing the gateway nodes, i.e., nodes that belong to two or more s_groups.

The tracing to be performed is specified within a configuration file. An agent is started by a master SD-Mon node for each s_group and for each free node, as shown in Fig. 16. Configured tracing is applied on every monitored node, and traces are stored in a binary format in the agent file system.

SD-Mon is dynamic: each network change in the s_group structure is cached by agents and notified to the master, which is the only one having a global network view. It takes care to restructure the shadow network accordingly: if a new s_group is created then a new agent is started; if an s_group is deleted then the related agent is stopped, the tracing files are gathered from its host, and new agents are started for its nodes not controlled by any other agent. This is of particular value for *deployed systems* running in production mode; online monitoring allows devops tuning and anticipation of changed future deployment configurations.

6.6 Devo

Devo is an online visualisation tool for SD Erlang programs.[1] There are two visualization options available to a user: "Low" and "High" level visualizations. The low level visualization shows process migrations and the run queue lengths of a single Erlang node, whereas the high level visualization shows nodes in s_groups using D3's force-directed graph functionality (Fig. 17).[2]

The s_groups to which a node belongs are indicated by the colour(s) of the graph node representing the Erlang node, and the edges connect nodes within the same group. When a single node is in more than one group the node's colour is split between the appropriate colours. When the nodes

1. Devo is available online from github.com/RefactoringTools/devo
2. Neatly replicating the system architecture in Fig. 1.

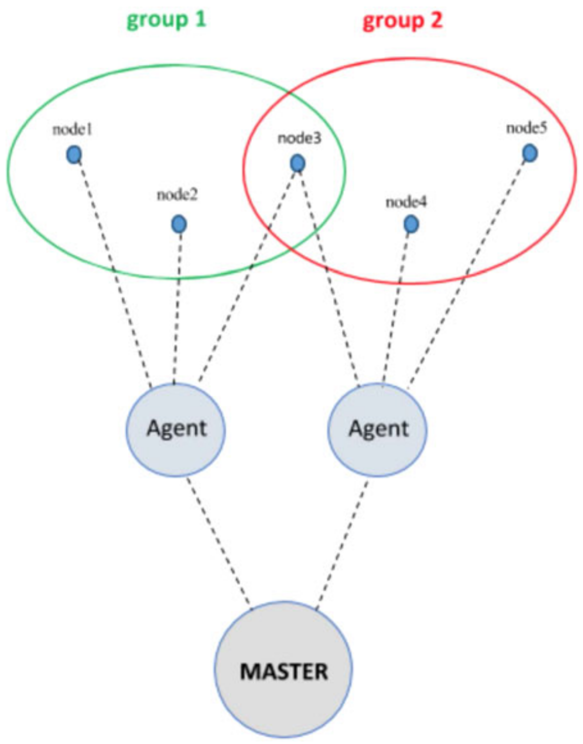


Fig. 16. Monitoring for (2 s_group) SD-Orbit within SD-Mon.

communicate with each other the edges change colour to indicate the level of communication between those two nodes relative the amount of communication between other nodes.

6.7 Percept2

Percept2 enhances the Percept tool from the standard Erlang distribution, which gives an off-line visualisation of the processes making up a concurrent Erlang system on a single node. Percept2 adds the functionality to support multicore and distributed systems, and also refactors the tool to be more scalable [45]. It also improves on the existing functionality in Percept, by, for example, letting users control the (huge) amount of data that is collected by the tool, through allowing them to profile particular aspects of a system execution, or allowing more selectivity in function profiling within processes.

Percept2 also enhances the kind of information presented to users. In particular, Percept2 distinguishes between process states of *running* and *runnable*, i.e., the process is ready to run, but another process is currently running. Runnable, but not running, processes present an opportunity for further exploitation of concurrency. Other enhancements include an improved dynamic function callgraph and a graph displaying process communications.

Percept2 particularly allows the tracing of s_group activities in a distributed system. The trace events collected, together with the initial s_group configuration if there is any, can be used to generate an off-line replay of the s_group structure evolution, as well as the online visualisation of the current s_group structure, of an Erlang system.

The insight gained from examining running and runnable processes in Percept2 underpins the parallelization refactorings in Wrangler [46] that improve scalability by getting the full performance of each multicore node.

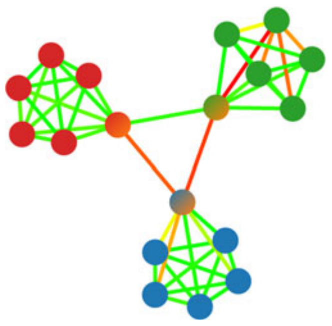


Fig. 17. S_groups in SD-Orbit using a 3D force-directed graph (Devo).

6.8 WombatOAM

WombatOAM is a tool that provides a scalable infrastructure for the deployment of thousands of Erlang nodes. Its broker layer creates, manages, and dynamically scales heterogeneous clusters. For fault tolerance WombatOAM also provides optional monitoring of deployed nodes, periodically checking whether nodes are alive, and restarting failed ones if needed. The deployment phase of WombatOAM comprises these steps: (1) registering providers; (2) uploading the application; (3) defining node families, i.e., nodes in the same node family have identical initial behaviour and run the same Erlang release; and (4) node deployment.

Deployment time is the period between arrival of a deployment request and a confirmation from WombatOAM that all nodes have been deployed. The deployment time depends on various factors, such as the number of nodes (N_C), usage of monitoring service (on or off), and the number of nodes deployed on the same host (fair or unfair). Fair deployment means that the number of nodes on each host is less than, or equal to, the number of cores on that host. Due to the fact that on the Athos cluster we have an access to 256 physical hosts with 24 cores each (Table 1), in the experiments we run up to 6,144 nodes using fair deployment, and up to 10,000 nodes using unfair deployment.

When running an unfair deployment of up to 10k Erlang nodes, the monitoring is off and every CPU core is shared by three Erlang nodes. The results show that deployment time changes linearly with the best fit equal to $(0.0124506 \cdot N + 83.4547)$, and 10,000 Erlang nodes are deployed in less than 4 minutes (approx. 47 nodes per second).

To analyse the impact of monitoring on the deployment time we measure the time of fair deployment (one node per CPU core) against two monitoring states: enabled (on) and disabled (off). From the performance results in Fig. 18 we conclude that at the target scale WombatOAM monitoring has no impact on the deployment time.

Finally, we analysed the impact of using WombatOAM on a running system, the ACO benchmark, comparing its runtime in two scenarios: monitoring enabled (on) and disabled (off). Fig. 19 shows that the WombatOAM monitoring overhead is not intrusive, at 1.33 percent maximum.

7 Conclusion & Future Work

Conclusion. In prior work we have investigated the scalability limits of distributed Erlang for engineering reliable systems, identifying network connectivity and the maintenance of global recovery information as the key bottlenecks. To address these issues we have developed a

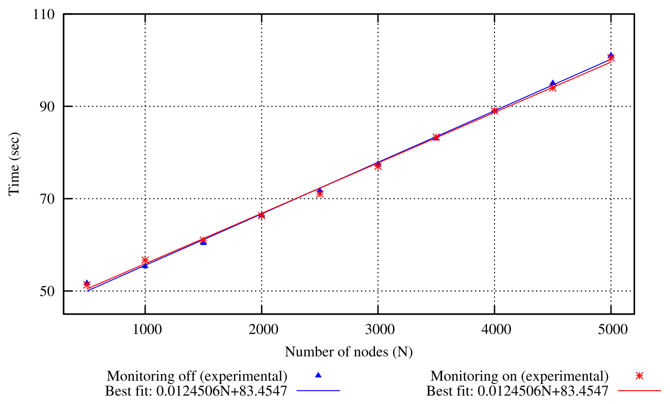


Fig. 18. Impact of WombatOAM's monitoring on deployment time.

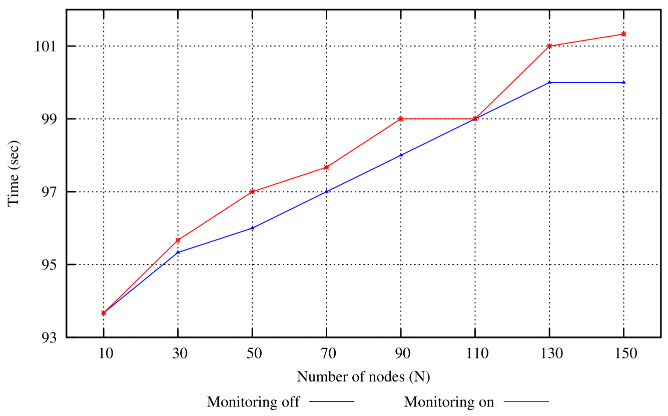


Fig. 19. Impact of WombatOAM's monitoring.

Scalable Distributed Erlang library that partitions the network of Erlang nodes into scalable groups (s_groups) to minimise both network connectivity and global recovery data (Section 3).

This paper presents a systematic evaluation of SD Erlang for improving the scaling of reliable applications using the Orbit, Ant Colony Optimisation, Instant Messenger) archetypal benchmarks and the Sim-Diasca case study outlined in Section 4. We report measurements on several platforms, but the primary platform is a cluster with up to 256 hosts and 6,144 cores.

The benchmarks evaluate different aspects of SD Erlang: Orbit evaluates the scalability impact of transitive network connections, ACO evaluates the scalability impacts of both transitive connections and the shared global namespace required for reliability, and IM targets reliability. The experiments cover three application-specific measures of scalability: speedup for Orbit, weak scaling for ACO, and throughput for IM.

We investigate three performance research questions (Section 5), obtaining the following results that are consistent with other experiments [5], [7].

RQ1: For unreliable applications we show that SD Erlang applications scale better than distributed Erlang applications. Specifically SD-Orbit scales better than D-Orbit, and that SR-ACO scales better than ML-ACO. The SD Erlang applications have far less network traffic. We conclude that, even when global recovery data is not maintained, partitioning the fully-connected network into s_groups reduces network traffic and improves performance on systems with more than 40 hosts on the Athos cluster (Sections 5.2 and 5.3).

RQ2: ACO has relatively simple reliability mechanisms, essentially a supervision tree with a single supervised process type. Reliability in IM is far more elaborate and realistic with multiple types of process supervised, and the potential for Chat_Session or Client databases to fail, and hence more elaborate recovery mechanisms. Chaos Monkey experiments with both ACO and IM show that both are reliable, and hence we conclude that *SD Erlang preserves the distributed Erlang reliability model* (Section 5.4).

RQ3: For reliable applications we show that SD Erlang scales better than distributed Erlang. Comparing the weak scaling of the reliable GR-ACO with the unreliable ML-ACO shows that maintaining global recovery data, i.e., a process name space, induces a huge amount of network traffic and dramatically limits scalability above 40 hosts. Comparing GR-ACO and SR-ACO weak scaling shows that scalability can be recovered by partitioning the nodes into appropriately-sized s_groups, and hence maintaining the recovery data only within a relatively small group of nodes (Section 5.4.2).

We present a new systematic and tool-based *approach for refactoring distributed Erlang applications into SD Erlang*. The approach presents a set of design questions, and builds on a suite of new or improved tools for monitoring, debugging, deploying and refactoring SD Erlang applications (Section 6).

We demonstrate the *capability of the tools*, for example showing that WombatOAM is capable of deploying and monitoring substantial (e.g., 10K Erlang VM) distributed Erlang and SD Erlang applications with negligible overheads (Section 6.8).

Future Work. The SD Erlang libraries would benefit from enhancements: e.g., to automatically route messages between s_groups. It would be interesting to extend our preliminary evidence that suggests that some SD Erlang technologies and methodologies could improve the scalability of other actor languages. For example, the Akka framework for Scala could benefit from semi-explicit placement, and Cloud Haskell from partitioning the network [47]. In the medium term we plan to integrate SD Erlang with other technologies to create a generic framework for building performant large scale servers.

Acknowledgments

We thank our RELEASE project colleagues for technical insights. This work has been supported by the EU grant 'RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software' (287510), and by the UK's EPSRC grant 'Adaptive JIT-based Parallelism (AJITPar)' (EP/L000687/1).

References

[1] L. A. Barroso, J. Clidaras, and U. Hölzle, *The Datacenter as a Computer*, 2nd ed. San Rafael, CA, USA: Morgan and Claypool, 2013.
[2] J. Armstrong, *Programming Erlang: Software for a Concurrent World*, 2nd ed. Raleigh, NC, USA: Pragmatic Bookshelf, 2013.
[3] F. Cesarini and S. Thompson, *Erlang Programming*, 1st ed. Sebastopol, CA, USA: O'Reilly Media, 2009.
[4] M. Odersky, et al., "The Scala programming language," 2012. [Online]. Available: http://www.scala-lang.org/
[5] N. Chechina, H. Li, A. Ghaffari, S. Thompson, and P. Trinder, "Improving the network scalability of Erlang," *J. Parallel Distrib. Comput.*, vol. 90/91, pp. 22–34, 2016.
[6] A. Tseitlin, "The antifragile organization," *Commun. ACM*, vol. 56, no. 8, pp. 40–44, 2013.

[7] P. Trinder, et al., "Scaling reliably: Improving the scalability of the Erlang distributed actor platform," *ACM Trans. Program. Lang. Syst.*, 2016, submitted for publication.
[8] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*. San Mateo, CA, USA: Morgan Kaufmann, 2001.
[9] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman, *MPI: The Complete Reference*. Cambridge, MA, USA: MIT Press, 1995.
[10] A. Gainaru and F. Cappello, "Errors and faults," in *Fault-Tolerance Techniques for High-Performance Computing*. Cham, Switzerland: Springer, 2015, pp. 89–144.
[11] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[12] T. White, *Hadoop: The Definitive Guide*. Sunnyvale, CA, USA: Yahoo! Press, 2010.
[13] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence," in *Proc. 3rd Int. Joint Conf. Artif. Intell.*, 1973, pp. 235–245.
[14] C. Hewitt, "Actor model for discretionary, adaptive concurrency," *CoRR*, 2010.
[15] P. Haller and F. Sommers, *Actors in Scala*. Mountain View, CA, USA: Artima Inc., 2012.
[16] "CAF: C++ actor framework," 2016. [Online]. Available: actor-framework.org/
[17] S. M. Jodal, et al., "Pykka," 2016. [Online]. Available: pykka.readthedocs.org/
[18] J. Epstein, A. P. Black, and S. Peyton-Jones , "Towards Haskell in the cloud," *SIGPLAN Notices*, vol. 46, no. 12, pp. 118–129, 2011.
[19] J. Lee, et al., "Python actor runtime library," 2010. [Online]. Available: http://osl.cs.uiuc.edu/parley/
[20] G. Germain, "Concurrency oriented programming in termite scheme," in *Proc. ACM SIGPLAN Workshop Erlang*, 2006, pp. 20–20.
[21] Rust, 2016. [Online]. Available: https://www.rust-lang.org/
[22] SpilGames, "Spapi-router: A partially-connected Erlang clustering," 2014. [Online]. Available: https://github.com/spilgames/spapi-router
[23] K. MacKenzie, N. Chechina, and P. Trinder, "Performance portability through semi-explicit placement in distributed Erlang," in *Proc. ACM SIGPLAN Workshop Erlang*, 2015, pp. 27–38.
[24] Basho, "Riak," 2014. [Online]. Available: http://basho.com/riak/
[25] EDF, "The Sim-Diasca Simulation Engine," 2010. [Online]. Available: http://www.sim-diasca.com
[26] WhatsApp, 2015. [Online]. Available: https://www.whatsapp.com/
[27] F. Lubeck and M. Neunhoffer, "Enumerating large orbits and direct condensation," *Exp. Math.*, vol. 10, no. 2, pp. 197–205, 2001.
[28] M. Dorigo and T. Stützle, *Ant Colony Optimization*. Scituate, MA, USA: Bradford Company, 2004.
[29] R. McNaughton , "Scheduling with deadlines and loss functions," *Manage. Sci.*, vol. 6, no. 1, pp. 1–12, 1959.
[30] M. Day, J. Rosenberg, and H. Sugano, "A model for presence and instant messaging," IETF, Fremont, CA, USA, Tech. Rep. RFC2778, 2000.
[31] S. Aggarwal, J. Vincent, G. Mohr, and M. Day, "Instant messaging/presence protocol requirements," IETF, Fremont, CA, USA, Tech. Rep. RFC2779, 2000.
[32] M. M. Hernandez, N. Chechina, and P. Trinder, "A reliable instant messenger in Erlang: Design and evaluation," Glasgow Univ., Glasgow, U.K., Tech. Rep. TR-2015-002, 2015.
[33] D. Luna, "Chaos monkey," 2016. [Online]. Available: https://github.com/dLuna/chaos_monkey
[34] Z. Xiao, L. Guo, and J. Tracey, "Understanding instant messaging traffic characteristics," in *Proc. 27th Int. Conf. Distrib. Comput. Syst.*, 2007, pp. 51–51.
[35] RELEASE D6.2, "Scalability case studies: Scalable Sim-Diasca for the blue gene," 2015. [Online]. Available: http://www.release-project.eu/documents/D6.2.pdf
[36] N. Chechina, M. Moro Hernandez , and P. Trinder, "A scalable reliable instant messenger using the SD Erlang libraries," in *Proc. ACM SIGPLAN Workshop Erlang*, 2016, pp. 33–41.
[37] M. den Besten, T. Stützle, and M. Dorigo, "Ant colony optimization for the total weighted tardiness problem," in *Proc. 6th Int. Conf. Parallel Problem Solving Nature*, 2000, pp. 611–620.
[38] D. Merkle and M. Middendorf, "An ant algorithm with a new pheromone evaluation rule for total tardiness problems," in *Proc. Real-World Appl. Evol. Comput. EvoIASP EvoSCONDI EvoTel EvoSTIM EvoROB EvoFlight*, 2000, pp. 287–296.
[39] M. J. Geiger, "New instances for the single machine total weighted tardiness problem," Helmut-Schmidt-Universität, Hamburg, Hamburg, Germany, Tech. Rep. 10-03-01, 2010.
[40] F. Cesarini and S. Vinoski, *Designing for Scalability with Erlang/OTP*, 1st ed. Sebastopol, CA, USA: O'Reilly Media, 2016.
[41] J. Armstrong, "Making reliable distributed systems in the presence of software errors," Ph.D. dissertation, Dept. Microelectron. Inf. Technol., KTH, Stockholm, Sweden, 2003.
[42] Wrangler, 2016. [Online]. Available: https://www.cs.kent.ac.uk/projects/wrangler
[43] H. Li and S. Thompson, "Automated API migration in a user-extensible refactoring tool for Erlang programs," in *Proc. 27th IEEE/ACM Int. Conf. Automat. Softw. Eng.*, 2012.
[44] RELEASE D5.3, "Systematic testing and debugging tools," 2015. [Online]. Available: http://www.release-project.eu/documents/D5.3.pdf
[45] H. Li and S. Thompson, "Multicore profiling for Erlang programs using Percept2," in *Proc. 12th ACM SIGPLAN Workshop Erlang*, 2013, pp. 33–42.
[46] H. Li and S. Thompson, "Safe concurrency introduction through slicing," in *Proc. Workshop Partial Eval. Program Manipulation*, 2015, pp. 103–113.
[47] RELEASE D6.7, "Scalability and Reliability for a Popular Actor Framework," 2015. [Online]. Available: http://www.release-project.eu/documents/D6.7.pdf

Natalia Chechina received the PhD degree from Heriot-Watt University and is a research fellow with the University of Glasgow. Her main research interests include distributed and parallel computing, scaling Erlang programming language, robotics, mathematical, and theoretical analysis.

Kenneth MacKenzie received the BSc degree in mathematics, the MSc degree in theoretical computer science, and the PhD degree in mathematics. He is a research fellow with the University of St Andrews. He is interested in programming language design and implementation.

Simon Thompson is professor of logic and computation in the School of Computing, University of Kent. His research interests include computational logic, functional programming, testing, and diagrammatic reasoning. He is the author of standard texts on Haskell, Erlang, Miranda and constructive type theory.

Phil Trinder is a professor of computing science with the University of Glasgow. For more than 20 years, he has researched the design, implementation, and evaluation of high-level distributed, and parallel programming models.

Olivier Boudeville is a research engineer in the SINETICS Department, EDF R&D, France. His interests include the simulation of complex systems, parallel and distributed architectures, and functional programming. He created the Sim-Diasca simulation engine.

Viktória Fördős has 8 years of software development experience at an insurance broker company and as a researcher with ELTE-Soft belonging to the Hungarian University of Science (ELTE) and at Erlang Solutions Hungary.

Csaba Hoch is a senior Erlang developer with Erlang Solutions, where he has been working on WombatOAM. He spent several years with Ericsson, where he participated in the development of NETSim, one of the largest Erlang programs ever written.

Amir Ghaffari received the PhD degree in computing science from Glasgow University. He is a senior software developer with Fujitsu Consulting, Canada, Inc.

Mario Moro Hernandez received the BA (Hons.) degree in psychology and the BSc (Hons.) degree in computing sciences.