A Scalable Reliable Instant Messenger Using the SD Erlang Libraries

Natalia Chechina, Mario Moro Hernandez, and Phil Trinder
School of Computing Science, University of Glasgow, United Kingdom
{Natalia.Chechina, Phil.Trinder@glasgow.ac.uk}@glasgow.ac.uk

Abstract

Erlang has world leading reliability capabilities, but while it scales extremely well within a single node, distributed Erlang has some scalability issues. The Scalable Distributed (SD) Erlang libraries have been designed to address the scalability limitations while preserving the reliability model, and shown to deliver significant performance benefits above 40 hosts using some relatively simple benchmarks.

This paper compares the reliability and scalability of SD Erlang and distributed Erlang using an Instant Messaging (IM) server benchmark that is a far more typical Erlang application; a relatively large and sophisticated benchmark; has throughput as the key performance metric; and uses non-trivial reliability mechanisms. We provide a careful reliability evaluation using chaos monkey.

The key performance results consider scenarios with and without failures on up to 17 server hosts (272 cores). We show that SD Erlang adds no performance overhead when all nodes are grouped in a single s_group. However, either adding redundant router nodes in distributed Erlang applications, or dividing a set of nodes into small s_groups in SD Erlang applications, have small negative impact. Both the distributed Erlang and SD Erlang IM tolerate failures and, up to the failure rates measured, the failures have no impact on throughput. The IM implementations show that SD Erlang preserves the distributed Erlang reliability properties and mechanisms.

Categories and Subject Descriptors C.4 [*Computer Systems Organization*]: Performance of Systems—Fault tolerance; D.1.3 [*Software*]: Programming Techniques—Concurrent Programming; D.3.2 [*Software*]: Programming Languages—Erlang

Keywords Erlang, distributed computation, fault tolerance, reliability.

1. Introduction

Erlang provides world leading reliability and scales extremely well within a single node, e.g. enabling 10^6 concurrent processes on a single host with 8GB RAM. The highly parallel architectures engendered by the manycore revolution are, however, revealing some scalability issues in large distributed Erlang systems. These issues often limit the scalability to between 40 and 80 hosts, depending on the application and, crucially the reliability mechanisms used [9].

The Scalable Distributed (SD) Erlang libraries have been designed to address the scalability limitations while preserving the reliability model [4, 9].

To date SD Erlang has been shown to deliver significant performance benefits above 40 hosts using some relatively simple benchmarks [5]. The Ant Colony Optimisation (ACO) is a parallel simulation benchmark, and the core of the Orbit benchmark is a non-replicated DHT, similar to NoSQL DBMS like Riak [2]. Both distributed Erlang and SD Erlang versions of ACO and Orbit are open source[1]. These benchmarks

- are relatively small, e.g. 100s of lines of code;
- are somewhat atypical of Erlang applications: Orbit as a DHT-based algebraic computation, and ACO as a simulation;
- have very simple reliability mechanisms: only ACO uses supervision and global name registration.

This paper is novel in presenting an evaluation of the SD Erlang libraries in comparison to distributed Erlang for an Instant Messaging server that

- is a far more typical Erlang application, i.e. a reliable server.
- is a larger and more sophisticated benchmark: 6K lines of code, using ETS tables, realistic chat behaviours, etc. [12] (Section 3).
- uses non-trivial reliability mechanisms, including supervision trees, redundancy, transitive connectivity, and global name registration (Section 3).
- provides a careful reliability evaluation using chaos monkey [10] (Section 6.4).
- has throughput as the key performance metric (Section 6). This is, again, more typical for distributed Erlang applications. In contrast Orbit uses strong scaling, and ACO weak scaling.

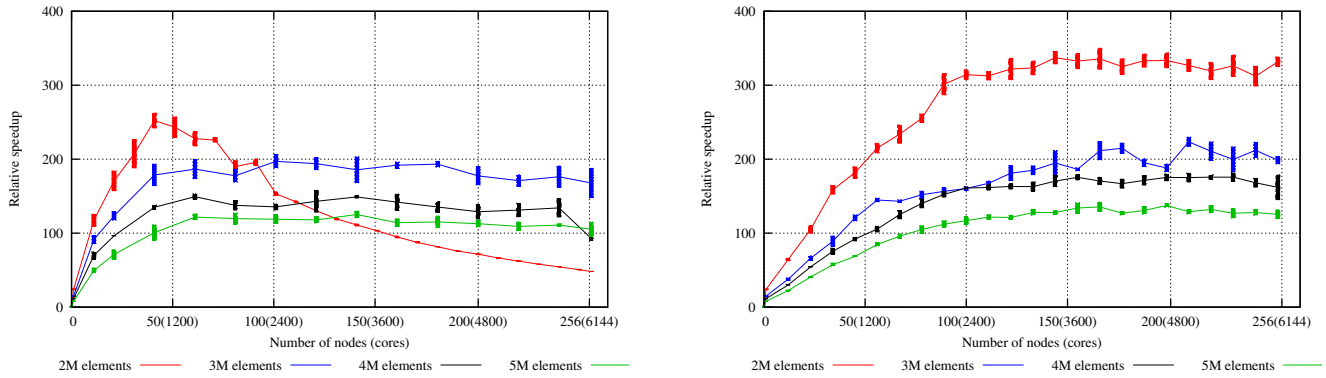
2. Background

Coming from the telecoms sector where systems need to be scalable to accommodate hundreds of thousands of calls concurrently, in soft real-time, a key feature of Erlang is its reliability. The key mechanisms – supervision trees, redundancy, transitive connectivity, and global name registration – led by the "let it crash" philosophy enable the engineering of highly-available and reliable applications [3].

2.1 Scaling and Reliability in Distributed Erlang

Reliability in Erlang is multi-faceted. Each process has a private state, preventing a failed or failing process from corrupting the

[1] `https://github.com/release-project/benchmarks`



(a) Distributed Erlang

(b) SD Erlang

Figure 1. Orbit Relative Speedup [5]

state of other processes. Messages enable stateful interaction, and contain a deep copy of the value to be shared, with no references to the senders' internal state. Moreover, Erlang avoids type errors by enforcing strong typing, albeit dynamically [1].

Connected nodes check liveness with heartbeats, and can be monitored from outside Erlang, e.g. by an operating system process. Enabling 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.

The high cost of maintaining a fully connected network was confirmed in experiments with Orbit [5], a generalization of a transitive closure computation [6]. The Orbit benchmark has no global operations, and hence its scalability depends only on the number of connections maintained. Figure 1 reports relative speedups for Orbit on a cluster with up to 256 hosts (6144 cores). The distributed Erlang results in Figure 1(a) show that performance deteriorates as the number of nodes grows beyond 60. This is particularly visible in the experiments where the number of orbit elements is less than 3M. In addition, the benchmark fails when we attempt to increase the number of orbit elements beyond 5M.

The most important way to achieve reliability is process *supervision*, which allows processes to monitor each other 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.

The namespace is, however, a global data structure, and updates must be propagated to every node. Hence as the number of nodes or the failure rate of registered processes grow, global name registration has a significant impact on network scalability. Figure 2 shows the throughput limit reached for a distributed Erlang peer-to-peer system at 40 nodes and hosts with just 0.01% of global operations [4].

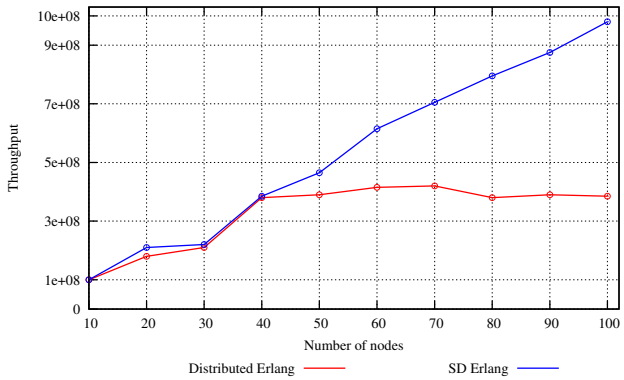


Figure 2. Impact of Global Operations on Network Scalability of Distributed Erlang and SD Erlang [4]

In summary transitive node connectivity, and global name registration for reliability, significantly limit the scalability of distributed Erlang.

2.2 Scalable Distributed Erlang

Scalable Distributed Erlang (SD Erlang) was designed to preserve reliability of distributed Erlang while enabling scalability by partitioning the node connection graph into s_groups [4], and by introducing semi-explicit process placement [8].

To reduce the number of connections and the size of the namespace, nodes are grouped into s_groups. Similarly to Erlang/OTP global_groups[2], 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. However, unlike global_groups the s_groups do not partition the set of nodes, i.e. a node can belong to an unlimited number of s_groups which makes it possible to create different connection topologies 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 ensure fault-tolerance s_groups may have two or more gateway nodes; this will ensure that the s_group nodes remain connected to the rest of the system even when one of the gateway nodes fails.

The functionality of gateway nodes can be combined with other types of nodes. For example, in the SD Erlang version of the IM application, gateway functionality is combined with routing. So, to be consistent with distributed Erlang version of the IM we call these nodes router nodes (Section 3.1).

S_group functionality is supported by 15 functions, 8 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`). 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 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.
```

More details regarding s_group functions can be found in [4].

The impact of s_groups on scalability due to the reduced number of connections alone can be observed in the Orbit benchmark. The relative speedups in Figure 1(b) show that the SD Erlang version of Orbit performs better than its distributed Erlang counterpart (Figure 1(a)) on large number of nodes (beyond 150), and as the number of nodes grows its performance remains stable even when the number of orbit elements is either less than 3M (red line in Figure 1(b)) or at least as big as 60M (this line is not included in the figure).

Systematic experiments with the Orbit and ACO benchmarks on up to 256 nodes (6144 cores) consistently show the scalability limitations of distributed Erlang, and that SD Erlang improves scalability of both reliable and unreliable applications [4, 5, 9].

2.3 Instant Messaging Servers

With an increase in popularity of mobile phones, social networks, and on-line games instant messaging applications also gain more and more popularity [11]. Originating from telecoms Erlang is a natural language of choice for Instant Messengers (IM). A vivid proof for that is the widely used WhatsApp application, which in February 2016 reached 1Bn monthly users[3]. The statistics available from March 2014[4] about WhatsApp is really impressive. At the time it had 465M monthly users delivering 19Bn messages in and 40Bn messages out per day. This volume was supported by only 550 servers, where each server had approximately 1M connections. That is an Erlang system running on more than 11K cores. However, like many other large scale distributed Erlang applications, WhatsApp does not use default Erlang distribution, but rather an ad hoc approach – introducing its own libraries which provide features resembling transitivity and shared namespace, but restricted to a particular connectivity mechanism.

The reason we decided to use an IM benchmark to analyse the SD Erlang reliability is because we needed a typical Erlang application that would be relatively small to allow quick changes and refactoring while providing insights of the reliability properties of distributed Erlang and SD Erlang. We do not intend to simulate or imitate any particular application, rather some generic instant messaging application that has some level of agreement with real life IMs in terms of general functionality, number of messages and users. The benchmark follows client-server pattern, where the server side supports the IM functionality, and the client side provides the traffic generation.

From modern IM applications like WeChat, WhatsApp, and Slack we expect a lot of functionality. For example, a support of different types of messages including texts, videos, and voice recordings, tracking and sharing GPS location, integration of third party applications, and groups of people contributing to the same chat. In our benchmark we focus on the server side while the client side has only supportive role. The server side ensures that messages are passed between the right logged-in clients independently of their location. It also ensures message delivery despite of failure of nodes or connections, and information about users. The client side generates traffic and informs server nodes where the clients are logged-in. As long as a client knows the name of a target user, it does not care where it is and how to deliver messages – this is server's responsibility. The conversations are limited to two users who exchange text messages. Both these limitations are not fundamental and can be modified if needed.

The benchmark is open source, and is available from the following public repository: `https://github.com/release-project/benchmarks/tree/master/IM`.

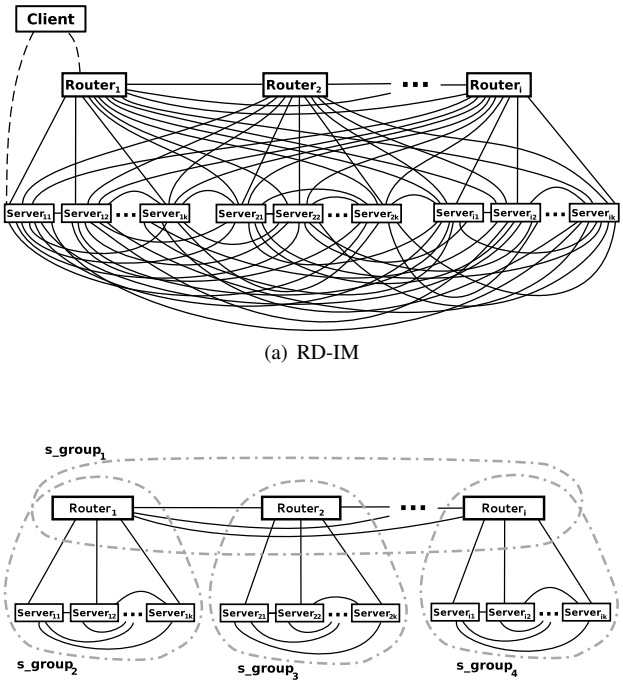
3. Design and Implementation

In this section we discuss design and implementation of the IM benchmark that consists of two parts: client side and server side. The server side has two versions implemented in distributed Erlang (RD-IM) and SD Erlang (RSD-IM). The RD-IM uses default Erlang distribution model, so all server side node are interconnected (Figure 3(a)), whereas RSD-IM utilizes s_groups that have transitive connections only between nodes of the same s_group (Figure 3(b)). The two versions are very similar with only small differences which

[2] `http://erlang.org/doc/man/global_group.html`

[3] `https://blog.whatsapp.com/616/One-billion`

[4] `http://highscalability.com/blog/2014/3/31/how-whatsapp-grew-to-nearly-500-million-users-11000-cores-an.html`



(a) RD-IM

(b) RSD-IM

Figure 3. Node Connections

are due to introducing s_groups in the RSD-IM. Therefore, we first discuss server side (Section 3.1) and client side (Section 3.2) implementations which are common for both RD-IM and RSD-IM versions, and then outline features specific to RSD-IM (Section 3.3).

3.1 Server Side

On the server side we distinguish two types of nodes: routers and servers.

Router nodes act as an interface between client and server nodes. Router nodes have three types of processes that have the following functionality (Figure 4 – links indicate supervision and monitoring dependencies).

- *Router* processes forward client requests to different server nodes deployed in the system. That is when a client logs in, a *router* process identifies the server node to which the client is going to be assigned using hash value of the client's id, and then forwards the client request to the corresponding *server_monitor* process. Every *router* process supervises one *server_monitor* process.
- *Router_supervisor* and *Router_supervisor's_monitor* processes are supervisors used to ensure fault tolerance of processes on the router nodes. *Router_supervisor* processes monitor *router* processes, i.e. if a *router* process terminates unexpectedly, its *router_supervisor* restarts it. *Router_supervisor* and *Router_supervisor's_monitor* processes also monitor each other, so that if one of the processes fails they can re-start each other and provide essential data for recovery.

There can be multiple router processes, but only one *router_supervisor* process and one *Router_supervisor's_monitor* process per router node.

Server nodes are the core element of the IM, that provide message exchange between clients. This is achieved by the following three types of processes (Figure 4).

- *Server_monitor* processes ensure fault tolerance of processes that reside on the corresponding server nodes. In addition these processes handle clients' log-in requests by spawning *client_monitor* processes if the corresponding clients are not logged-in already. It also spawns *chat_session* processes when new conversations are started. Every server node has only one *server_monitor* process.
- *Chat_session* processes enable communication between clients by forwarding messages, and sending confirmations of message delivery to the senders. The number of *chat_session* processes depends on the number of live conversations.
- *Client_monitor* processes keeps track of client processes while the clients are logged-in to the system. The number of *client_monitor* processes depends on the number of logged-in clients. To ensure that clients are not logged-in multiple times, the server nodes on which these processes reside depend on the hash of the client ids.

Server nodes also contain two distributed databases to store the information about the logged-in clients (*Clients_DB*) and the running chat sessions (*Chat_Sessions_DB*). The databases are implemented using ETS tables; while fault tolerance is provided by replicating these databases on neighbouring server nodes.

Number of Connections. The RD-IM (Figure 3(a)) and RSD-IM (Figure 3(b)) have identical total number of the server side nodes(N_T):

$$N_T = N_R + N_S, \tag{1}$$

where N_R is the number of router nodes, and N_S is the number of server nodes. A straightforward way to scale the systems is adding server nodes. However, routers have a limited capacity; therefore, to avoid deteriorating the performance, the router nodes should be also periodically increased.

In the *RD-IM* all nodes are interconnected (Figure 3(a)). Therefore, the total number of node connections N_{C1} is the following:

$$N_{C1} = \frac{(N_R + N_S) \cdot (N_R + N_S - 1)}{2}. \tag{2}$$

From (1) and (2) it can be observed that a linear scaling of the number of nodes in distributed Erlang systems leads to an exponential growth of the number of node connections.

In the *RSD-IM* nodes are only interconnected in the s_groups (Figure 3(b)). Assume that there are N_R s_groups (one router node per an s_group) and all s_groups have the same number of

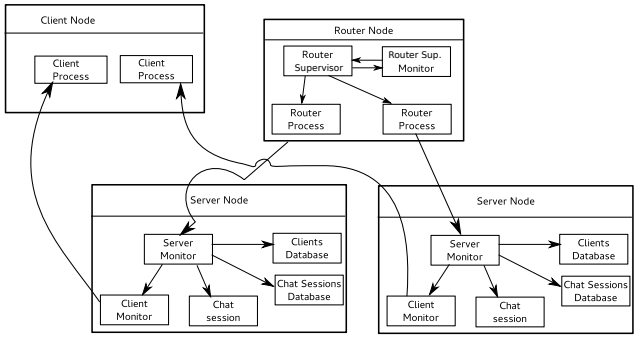


Figure 4. IM Nodes and Processes, and Their Dependencies

nodes (N_{S1}):

$$N_{S1} = \frac{N_S}{N_R} + 1. \quad (3)$$

Then the total number of connections N_{C2} is the following:

$$N_{C2} = N_{CR} + N_R \cdot N_{CS1}, \quad (4)$$

where N_{CR} is the number of connections in the router s_group:

$$N_{CR} = \frac{N_R \cdot (N_R - 1)}{2}, \quad (5)$$

and N_{CS1} is the number of connections in a server s_group:

$$N_{CS1} = \frac{N_{S1} \cdot (N_{S1} - 1)}{2} = \frac{N_S \cdot (N_S + N_R)}{2 \cdot N_R^2}. \quad (6)$$

Replacing (5) and (6) in (4), the total number of connections in the RSD-IM is as following:

$$N_{C2} = \frac{N_R^2 \cdot (N_R - 1) + N_S \cdot (N_S + N_R)}{2 \cdot N_R}. \quad (7)$$

Therefore, the difference between the number of maintained connections in RD-IM and RSD-IM is the following:

$$N_{C1} - N_{C2} = \frac{N_S \cdot (N_R - 1) \cdot (N_S + 2N_R)}{2 \cdot N_R}. \quad (8)$$

The result in (8) is always non-negative, and the only configuration that results in the identical number of connections in both RD-IM and RSD-IM is when the latter has one s_group.

3.2 Client Side

The client side is identical for both RD-IM and RSD-IM. All its nodes are hidden, and therefore their connections are non-transitive. On the client side we have two types of nodes: client and traffic generator.

Client nodes host client processes. These can be of the following two types.

- A *normal client* is a command-line process that sends messages and prints the received messages to the standard output of the terminal.
- A *'doped' client* is a process that based on the *normal client* but uses traffic generation logic to send messages to stress the IM architecture. The *'doped' client* processes operate as following. A *'doped' client* receives a message from a traffic generator process that triggers a conversation with another *'doped' client* logged in the system (the pid of the target process is also provided by the traffic generator process). Then it generates a random string that imitates a line in a conversation and sends the message to the receiver client after a random period of time that ranges between 1 and 20 seconds – this simulates the time spent to type that message. Then it waits a reply to repeat the process. The number of messages the *'doped' clients* exchange is random, and is specified at the beginning of the conversation. When the conversation finished, the *'doped' client* notifies corresponding *chat_session* process, which in turn terminates as if two *normal client* has finished their chat session.

Client nodes also have *router_DB* databases which are containers for the pids of the deployed router processes. From the perspective of the "real life" application, this is an auxiliary process. However, it is essential for benchmarking as it provides the client processes with the pids of the routers enabling clients to connect to the server side of the application.

Traffic Generator nodes contain traffic generator processes that trigger conversations, i.e. select clients and the number of messages in the conversation. Each of the processes control up to ten conversations simultaneously. The traffic generator is parametrised to represent realistic chat behaviours as identified in [12].

3.3 RSD-IM Specific Processes

Additional RSD-IM processes are due to differences in handling namespace in distributed Erlang and SD Erlang. That is, to provide fault tolerance of *client_DB* and *chat_DB* their processes are registered globally. If a database fails, global registration enables easily locate required database to complete the operation, e.g. storing or accessing data.

In the RD-IM the names are replicated on all nodes, i.e.

```
global:register_name(process_db_i,PidI).
```

While in the RSD-IM the processes are replicated only on the nodes of the corresponding server s_groups, i.e.

```
s_group:register_name(s_group_n,process_db_i,PidI).
```

In the current s_group implementation, s_group names are not accessible outside their group. Therefore, to access replicas in the RSD-IM *relay* processes were added. These processes multicast messages to all *router_processes* which in turn have an access to the corresponding server s_group registered names, because router nodes apart from the router s_group, also belong to corresponding server s_groups (Figure 3(b)).

```
s_group:whereis_name(s_group_n, process_db) ! Msg.
```

When a *router_process* finds the target process, it forwards the message. If the *router_process* does not have informatio about the process, the message is discarded.

4. Introducing Reliability

Following Erlang's *"let it crash"* philosophy we start by outlining the application's behaviour when no failures occur (Section 4.1), and then discuss possible failures and approaches to handle them (Section 4.2). In both scenarios we consider RD-IM, and the RSD-IM functionality is very similar, the only difference being that in reaching the target databases in RSD-IM the processes may need to send requests via *relay* processes if the required databases are in different s_groups.

4.1 No Failures

The IM functionality under normal conditions, i.e. when no failures occur, is quite simple, and can be summaries in the following three activities: client logging-in, client logging-out, and a chat session. Each activity has a sequence of steps to accomplish it. As an example, let us consider RD-IM chat session presented in Figure 5.

Here, *Client 1* sends a `start_chat_session` message to a *router* process that forwards the request to the *server_supervisor*. The *server_supervisor* spawns a *chat_session* process which first queries the *Chat_Sessions_DB* to ensure that a similar process for the same clients does not already exist. In a 'no failure' scenario the *Chat_Sessions_DB* confirms that a similar process does not exist, and the *chat_session* process, informs the involved clients about a successful start of the chat session. After that the clients exchange messages. To finish a chat session, one of the clients sends a `finish_chat_session` message to the *chat_session* process which removes its information from the *Chat_Sessions_DB*, and notifies the clients. Finally, *chat_session* process sends an exit message to the server supervisor, and terminates.

4.2 Handling Exceptions

Fault tolerance in the RD-IM and RSD-IM applications is supported by handling the following exceptions.

- A client attempts to log in to the system when it is logged in already.
- A client attempts to establish a chat session with another client with whom it already has an opened chat session.
- A *chat_session* process fails before the session is finished.
- A *router_process* fails.
- A *server_supervisor* process fails.
- A *Chat_Session_DB* or *Client_DB* fail.

As an example, let us consider a failure of a *chat_session* process before the session is finished (Figure 6). In this case the failure is handled by the corresponding *server_supervisor*, which after receiving the `{error, Reason}` system error message sends a query to the *Chat_Session_DB* to remove the record of the failed session. It then spawns a new *chat_session* process, updates the record in the *Chat_Session_DB*, and notifies the clients of the new *chat_session* pid.

5. Evaluation Framework

To evaluate the RD-IM and RSD-IM performance and run the experiments presented in Section 6, we introduce two additional modules: `rhesus` (Section 5.1) and `logger` (Section 5.2).

5.1 Reliability: Rhesus

To analyse fault tolerance of the IM benchmark we introduce the `rhesus` module named after rhesus macaque that follows the chaos monkey [10] principle. The `rhesus` module is written in Erlang, and is specifically designed to test the IM benchmark. In particular, it has the following features.

- Random and user-defined termination time for processes ranging from a second to an hour.
- Weighted termination probability of processes.

The original chaos monkey [10] was developed at Netflix, and is used to terminate virtual machines. There are also alternative versions written in, and for, Erlang like [7]. The reason we introduce the `rhesus` module instead of utilising existing modules is due to the uniform termination probability used in existing approaches. That is, existing approaches do not distinguish between different types of processes, whereas in the IM benchmark the number of *client_monitor* and *chat_session* is several orders of magnitude larger than the other types of processes. Therefore, it would require unnecessary long experiments to analyse recovery time caused by, for example, *server_supervisor* processes. In addition, to pick

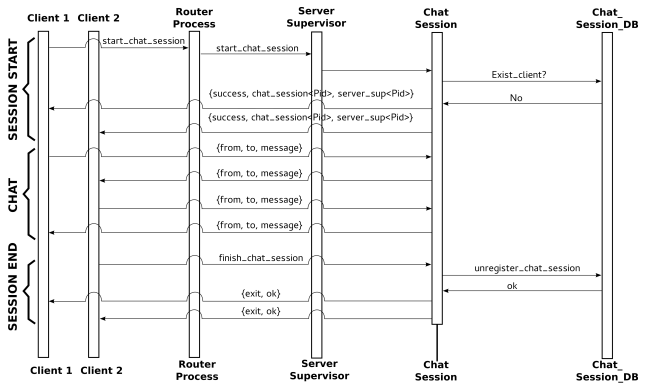


Figure 5. Chat Session Sequence Diagram

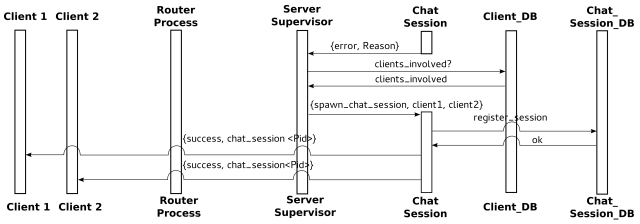


Figure 6. RD-IM *Chat_session* Recovery from a Failure

processes for termination the `rhesus` module exploits existing IM data structures kept on all nodes making the implementation and the analysis fairly straightforward. In the experiments presented in Section 6.4 `rhesus` module terminates only IM processes; however, other types of processes can be also included if needed.

The functionality of the `rhesus` module is as follows. The *rhesus* processes that run on all nodes initiate the termination by calling the `chaos_on/1` function. The function enables to set a number of parameters, including the time when terminations start (e.g. 5min after the experiment starts), the types of processes to terminate (random or specific types only), and the termination rate.

5.2 Measurements: Logger

The `logger` module provides a measurements-gathering facility that is used to analyse the IM performance in terms of latency and throughput. The module provides a set of *recorder* functions that are spawned at the client nodes whenever data-collection is required. At launch time, these processes create a comma separated value (`.csv`) file in which latency and throughput data is written in real-time. After the data collection session has finished, the processes close the file and if there are no more sessions left they terminate.

Recorders can be customised, allowing to specify such parameters as the name and path of the output files, the number of series to be recorded, and the length of the data collection sessions. For the throughput measurements, a latency threshold can be set; this can be used to determine the quality of the service defined as a percentage of messages delivered below the threshold.

6. Evaluation

This section describes the results of the IM tests. In particular, we analyse an impact of various aspects of s_groups on the IM performance in the absence of failures (Sections 6.1–6.3) and then an impact of failures (Section 6.4).

Configuration. All the measurements have been conducted on the GPG Beowulf cluster[5] at Glasgow University. In the experiments we use up to 17 hosts out of 20 (the remaining 3 hosts are used for general purpose). The cluster specification is as follows.

- 16 cores (2×Intel Xeon E5-2640 2 GHz) per host.
- 64 GB RAM (4 GB RAM per core) per host.
- 300 GB local disk.
- 10 Gb Ethernet interconnect.

The software configuration is as follows:

- Scientific Linux 6 (Carbon) 64-bit.
- SD Erlang/OTP based on Erlang/OTP 17.4[6].

[5] `http://www.dcs.gla.ac.uk/research/gpg/cluster.htm`

[6] `https://github.com/release-project/otp/tree/17.4-rebased`

Each server side node is deployed on a separate host, whereas client nodes and traffic generator nodes share hosts. Each client process is engaged in one conversation at a time. The throughput measures the number of delivered messages per minute. We plot the maximum throughput that the IM can handle, where further increase of the number of conversations leads to nodes running out of memory and the IM failure. We run all experiments for 15min, repeating every experiment 5 times.

6.1 Impact of the SD Erlang S_groups (No Failures)

In this experiment we analyse an impact of SD Erlang s_groups when no failure occurs. 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 set-up results in identical architectures for both IM versions where s_group operations in the RSD-IM are identical to the global operations in the RD-IM.

From the throughput results presented in Figure 7 we can make a few observations. Firstly, *at small scale when no failures occur the RD-IM and RSD-IM perform identically*. It also shows that it takes the system approximately *three minutes to reach a stable state* using the current traffic generator.

To analyse the throughput in the stable state against the number of nodes we replot the data to exhibit the relation in Figure 8. The results show that this relation is linear and *the server nodes have an effective capacity of handling approximately* 10^6 *messages per minute*. The curve-fit model follows $94,663.9 \cdot N_{servers} + 47,145.3$ dependency.

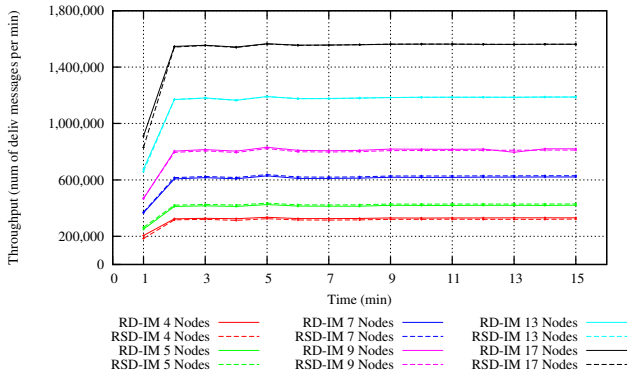


Figure 7. Throughput Without Failures (1 Router)

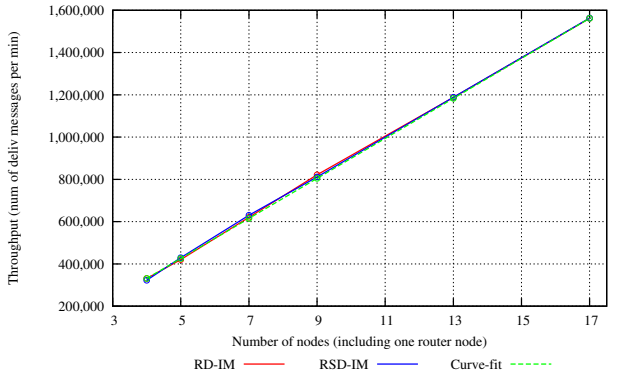


Figure 8. Throughput in a Stable State (1 Router)

Another conclusion that we can make here is that *in the absence of failures RD-IM does not hit its scalability limit at 17 nodes*.

6.2 Impact of the Size of S_groups (No Failures)

In this experiment we analyse an impact of the size of s_groups on the IM performance. For that we again increase the number of servers (6, 8, 12), but this time we fix the number of routers to two. In case of RSD-IM this results in two s_groups where depending on the total number of servers each s_group has either 3, 4, or 6 server nodes. Again we analyse the performance when no failures occur.

The throughput results in Figures 9 and 10 show that RSD-IM has a slightly higher throughput (0.6%–2.7%) in comparison with RD-IM, but both of them are lower than the curve-fit model identified in Figure 8. From this we conclude that *at this scale and set-up RD-IM and RSD-IM keep perform identically*, and *for a small scale of up to 14 nodes when no failures occur an increase of the number of routers does not improve the throughput*.

6.3 Impact of the Number of S_groups (No Failures)

In this experiment we analyse an impact of the number of s_groups on the RSD-IM performance. For that we keep the number of server nodes constant (equal to 12) while varying the number of router nodes, and hence the number of s_groups in the RSD-IM (1, 2, 3, 4).

The throughput results in Figure 11 show that one router (one s_group) configuration has the highest throughput and both DR-IM and RSD-IM perform identically. With the two router (two s_group)

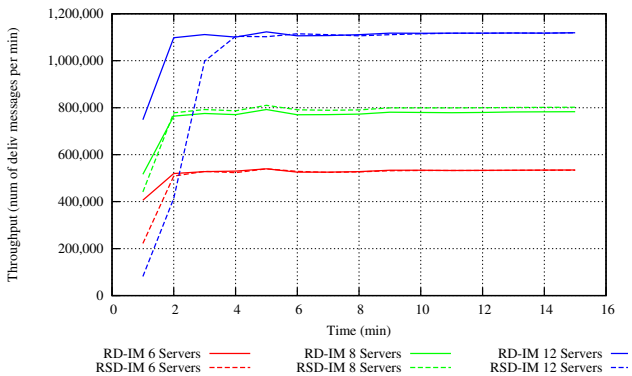


Figure 9. Throughput Without Failures (2 Routers)

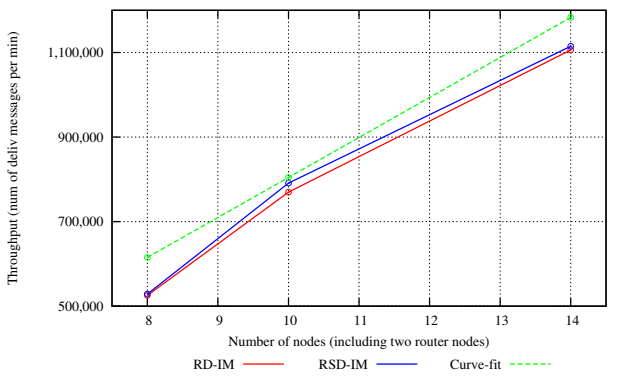


Figure 10. Throughput in a Stable State (2 Routers)

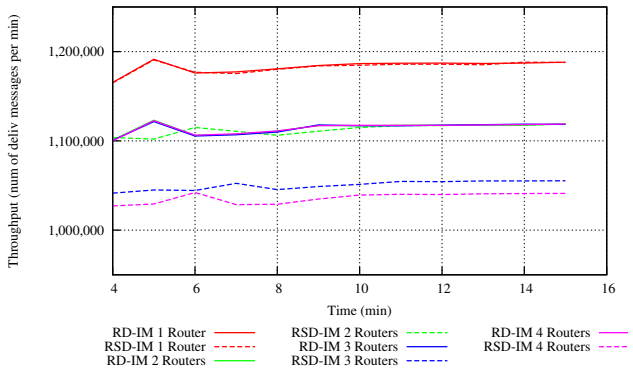


Figure 11. Impact of the Number of Routers (12 servers)

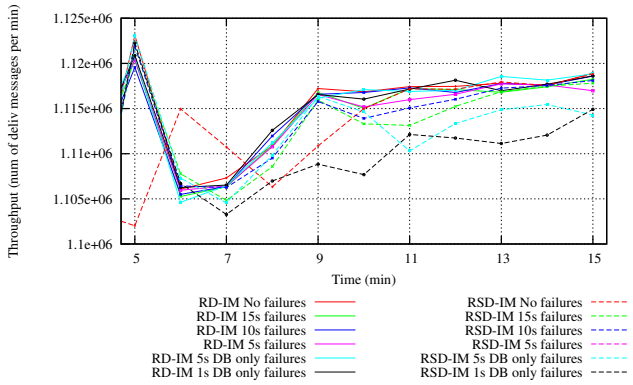


Figure 12. Throughput With and Without Failures

configuration DR-IM and RSD-IM again perform identically; however, their throughput is 6% lower than the one router (one s_group) experiment. Further increase of the number of routers up to four does not seem to have an impact on the RD-IM performance, while the RSD-IM keeps deteriorating by additional 6% (three routers) and then 1% (four routers). This confirms SD Erlang results from other benchmarks [5] in that *dividing a set of nodes into too small s_groups deteriorates the performance*, rather than improves it.

6.4 Impact of Failures

In this experiment we investigate the impact of failures and their rate on the performance of the RD-IM and RSD-IM applications. In the experiments we use 2 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 Figure 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). This demonstrates that SD Erlang can be used to build reliable applications in the same manner as distributed Erlang. Moreover, the use of *s_groups* does not impose any specific restriction to the default reliability mechanisms with which Erlang/OTP is shipped. Recall that the RSD-IM is just a refactored version of the RD-IM, which introduces the minimum necessary changes required to use the *s_groups* functionality. Apart from this, no further changes were made – especially regarding the supervision trees and fault-tolerance mechanisms.

7. Discussion

We have presented an evaluation of the SD Erlang libraries in comparison to distributed Erlang for an Instant Messaging server that is a larger and more sophisticated benchmark; is a far more typical Erlang application; has non-trivial reliability mechanisms; and uses throughput as the key performance metric.

The key performance results consider scenarios with and without failures on up to 17 nodes (272 cores). We have demonstrated that SD Erlang adds no performance penalties when all nodes are grouped in a single s_group (Figure 8). However using either redundant router nodes in RD-IM or dividing a set of nodes into small s_groups in RSD-IM have small negative impacts, i.e. when using four routers RD-IM and RSD-IM throughput decreases by 6% and 13% respectively (Figures 10 and 11). Both RD-IM and RSD-IM tolerate failures, and up to the failure rates measured, the failures have no impact on throughput. SD Erlang preserves the distributed Erlang reliability properties and mechanisms, and requires no additional mechanisms (Figure 12).

Future Work. On the available cluster with up to 17 server hosts the RD-IM has not hit its scalability limit. To analyse RD-IM and RSD-IM performance at scale we plan to conduct additional experiments using either a larger cluster or a cloud. We may also need to update traffic generator to enable the volume of traffic required at the target scale, and to take into account the ever growing sizes, types, and frequency of messages.

Acknowledgements

We would like to thank our RELEASE project colleagues for technical insights. This work has been supported by the European Union grant RII3-CT-2005-026133 'SCIEnce: Symbolic Computing Infrastructure in Europe', IST-2011-287510 'RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software', and by the UK's Engineering and Physical Sciences Research Council grant EP/G055181/1 'HPC-GAP: High Performance Computational Algebra and Discrete Mathematics'.

References

[1] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2nd edition, 2013.

[2] Basho. Riak, 2014. http://basho.com/riak/.

[3] F. Cesarini and S. Vinoski. *Designing for Scalability with Erlang/OTP*. O'Reilly, 2016.

[4] N. Chechina, H. Li, A. Ghaffari, S. Thompson, and P. Trinder. Improving the network scalability of Erlang. *JPDC*, 90-91:22–34, 2016.

[5] N. Chechina, K. MacKenzie, S. Thompson, P. Trinder, O. Boudeville, V. Fördős, A. Ghaffari, C. Hoch, and M. M. Hernandez. Evaluating scalable distributed Erlang for scalability and reliability. *(Submitted to) Journal of IEEE Transactions on Parallel and Distributed Systems*, 2016.

[6] F. Lubeck and M. Neunhoffer. Enumerating large orbits and direct condensation. *Experimental Mathematics*, 10(2):197–205, 2001.

[7] D. Luna. Chaos monkey, Available 2016. https://github.com/dLuna/chaos_monkey.

[8] K. MacKenzie, N. Chechina, and P. Trinder. Performance portability through semi-explicit placement in distributed Erlang. In *Erlang'15*, pages 27–38, New York, NY, USA, 2015. ACM.

[9] P. Trinder, N. Chechina, N. Papaspyrou, K. Sagonas, S. Thompson, et al. Scaling reliably: Improving the scalability of the Erlang distributed actor platform. *(Submitted to) ACM Trans. Program. Lang. Syst.*, 2016.

[10] A. Tseitlin. The antifragile organization. *Commun. ACM*, 56(8):40–44, 2013.

[11] N. Verite. Welcome to the third generation of instant messaging!, 2016. https://www.erlang-solutions.com/blog/welcome-to-the-third-generation-of-instant-messaging-part-1-2.html.

[12] Z. Xiao, L. Guo, and J. Tracey. Understanding instant messaging traffic characteristics. In *ICDCS'07*, pages 51–51. IEEE, 2007.