Scalable Reliable SD Erlang Design

Natalia Chechina, Phil Trinder,
Amir Ghaffari, Rickard Green,
Kenneth Lundin, and Robert Virding

School of Computing Science
The University of Glasgow
Glasgow G12 8QQ
UK

Technical Report TR-2014-002

December 23, 2014

Abstract

This technical report presents the design of Scalable Distributed (SD) Erlang: a set of language-level changes that aims to enable Distributed Erlang to scale for server applications on commodity hardware with at most 10^5 cores. We cover a number of aspects, specifically anticipated architecture, anticipated failures, scalable data structures, and scalable computation. Other two components that guided us in the design of SD Erlang are design principles and typical Erlang applications. The design principles summarise the type of modifications we aim to allow Erlang scalability. Erlang exemplars help us to identify the main Erlang scalability issues and hypothetically validate the SD Erlang design.

We start by giving an overview of hardware architectures to get an idea of the type of typical hardware architectures we may expect in the next 4-5 years. We expect it to be a NUMA architecture where cores are grouped in modules of 32-64 cores, and each host has 4-6 such modules. A server will consist of ≈ 100 hosts, and a cloud will consist of 2-5 servers (Chapter 3). Then we analyse failures that may happen during SD Erlang program execution. The anticipated failures have an impact on the design decisions we take for SD Erlang (Chapter 4).

To scale a language must provide scalable in-memory data structures, scalable persistent data structures, and a scalable computation model. We analyse ETS tables as the main Erlang in-memory data structures. The results show that ETS tables may have scalability problems, however, it is too soon to draw any conclusions as there were no yet real experiments with SD Erlang. As ETS tables are a part of Erlang Virtual Machine (VM) they are out of scope of the current research, and will be dealt by other partners of the RELEASE project (Section 6.1). For the persistent data structures we analyse Mnesia, CoachDB, Casandra, and Riak. The analysis of the database properties shows that such DataBase Management Systems (DBMSs) as Riak and Casandra will be able to provide essential scalability (Section 6.2).

SD Erlang's scalable computation model has two parts. The first part answers the question 'How to scale a network of Erlang nodes?'. Currently, Distributed Erlang has a fully connected network of nodes with transitive connections. We propose to change that and add scalable groups (s_groups). In s_groups nodes have transitive connections with nodes of the same s_group and non-transitive connections with other nodes. The idea of s_groups is similar to Distributed Erlang hidden global groups. S_groups differ from global groups in that nodes can belong to a multiple number of s_groups or do not belong to an s_group at all, and information about nodes is not global, i.e. in s_groups nodes collect information about nodes of the same

s_group but they do not share this information with nodes of other s_groups (Section 5.1).

The second part of the SD Erlang design answers the question 'How to manage a scaled number of Erlang nodes?'. As the number of Erlang nodes scale it is hard for a programmer to know the exact position of all nodes. Therefore we propose a semi-explicit placement. Currently, when a process is spawned the target node must be identified explicitly. In the semi-explicit placement the target node will be chosen by a decision making process using restrictions specified by the programmer, e.g. s_groups, or types of s_groups, or communicating distances on which the target node is located from the initial node (Section 5.2).

Finally, we provide an overview of typical Erlang applications and summarise their requirements to the SD Erlang (Chapter 7). We also discuss Erlang VM implications and outline our future work (Chapter 8).

Contents

Chapter 1

Introduction

The objectives of this technical report are to design a reliable Scalable Distributed (SD) Erlang. Promising ideas are to minimise connections between Erlang nodes and to provide abstract process placement control to maintain locality (affinity) and to distribute work.

The document presents the SD Erlang design and justification of the design decisions. The design is informed by the scalability requirements of the set of distributed exemplars in Chapter 7. For Erlang to scale we have considered the following key aspects.

1. *Hardware architecture trends.* To identify programmer requirements to the Erlang language we need to understand the target hardware architectures that we may expect in the next 4-5 years (Chapter 3).

2. *Anticipated failures.* Erlang has a very good reliability model, to preserve it we need to be aware of failures that may occur during program execution (Chapter 4).

3. *Scalable computation.* To understand Erlang scalability limitations we have analysed typical Erlang applications (Chapter 7). The results showed that the main scalability issue is transitive connections, e.g. a Riak users are already looking for ways to increase the number of Erlang nodes beyond a hundred. There is also a need of semi-explicit placement. Such applications as Sim-Diasca build their own tools to support semi-explicit placement. We believe that rather making developers to build their own tools it is better to build one that can be reusable (Chapter 5).

4. *Scalable in-memory and persistent data structures.* To build scalable applications programmers require support from in-memory and persistent data structures. The data structures need to be able to scale together with the application. Therefore, we have analysed in-memory and persistent data structures that can be used in Erlang applications (Chapter 6).

We start with design principles in Chapter 2. An overview of hardware architecture trends is presented in Chapter 3, and possible failures are covered in Chapter 4. A scalable computation model and scalable data structures are discussed in Chapters 5 and 6 respectively. Exemplars are discussed in Chapter 7. We conclude with discussion on the Virtual Machine (VM) and OTP implication, and future work package plans in Chapter 8.

Chapter 2

Design Principles

The following principles guide the design of SD Erlang.

1. General:

    - *Preserving the Erlang philosophy and programming idioms.*
    - *Minimal language changes*, i.e. minimizing the number of new constructs but rather reusing of existing constructs.
    - *Working at Erlang level* rather than VM level as far as possible.

2. Reliable Scalability:

    - *Avoiding global sharing*, e.g. global names, bottlenecks, and using groups instead of fully connected networks.
    - *Introducing an abstract notion of communication architecture*, e.g. locality/affinity and sending disjoint work to remote hosts.
    - *Avoiding explicit prescription*, e.g. replacing spawning on named node with spawning on group of nodes, and automating load management.
    - *Keeping the Erlang reliability model* unchanged as far as possible, i.e. linking, monitoring, supervision.

Chapter 3

Architecture Trends

Currently energy consumption, cooling, and memory size/bandwidth are considered to be the main factors that shape trends in computer architectures.

- As the number of cores goes up the memory bandwidth goes down, and the larger number of cores share the same memory the larger memory is required. DRAM-based main memory systems are about to reach the power and cost limit. Currently, the main two candidates to replace DRAM are Flash memory and Phase Change Memory (PCM). Both types are much slower than DRAM, i.e. 2^{11} and 2^{17} processor cycles respectively for a 4GHz processor in comparison with 2^9 processor cycles of DRAM, but the new technologies provide a higher density in comparison with DRAM [QSR09].

- Energy consumption and cooling are the main constrains for the high core density. Moreover, the cooling is also a limitation of the silicon technology scaling [War11]. To save energy and maximize compute performance supercomputers exploit small and simple cores apposed to large cores. Many architectures, especially HPC architectures, exploit GPUs [BC11]. GPUs accelerate regular floating point matrix/vector operations. The high throughput servers that RELEASE targets do not match this pattern of computation, and GPUs are not exploited in the server architectures we target. The air cooling might be replaced by one of the following technologies: 2D and 3D micro-channel cooling [HSG+10], phase-change cooling [MJN03], spot cooling [WZJ+04], or thermal-electric couple cooling [SSA+06].

From the above we anticipate the following typical server hardware architecture. A host will contain ∼4–6 SMP core modules where each module will have ∼32–64 cores. Analysis of the Top 500 supercomputers that always lead the computer industry illuminating the next 4–5 year computer architecture trends allows us to assume that ∼100 hosts will be grouped in a cluster, and $\sim$1–5 clusters will form a cloud. Therefore, the trends are towards a NUMA architecture. The architecture is presented in Figure 3.1.

Assumptions. Each host may have a single or a multiple number of IP addresses. A host may have a multiple number of Operating System (OS) instances. An OS instance may have a multiple number of Erlang nodes, and be executed on a multiple number of cores.

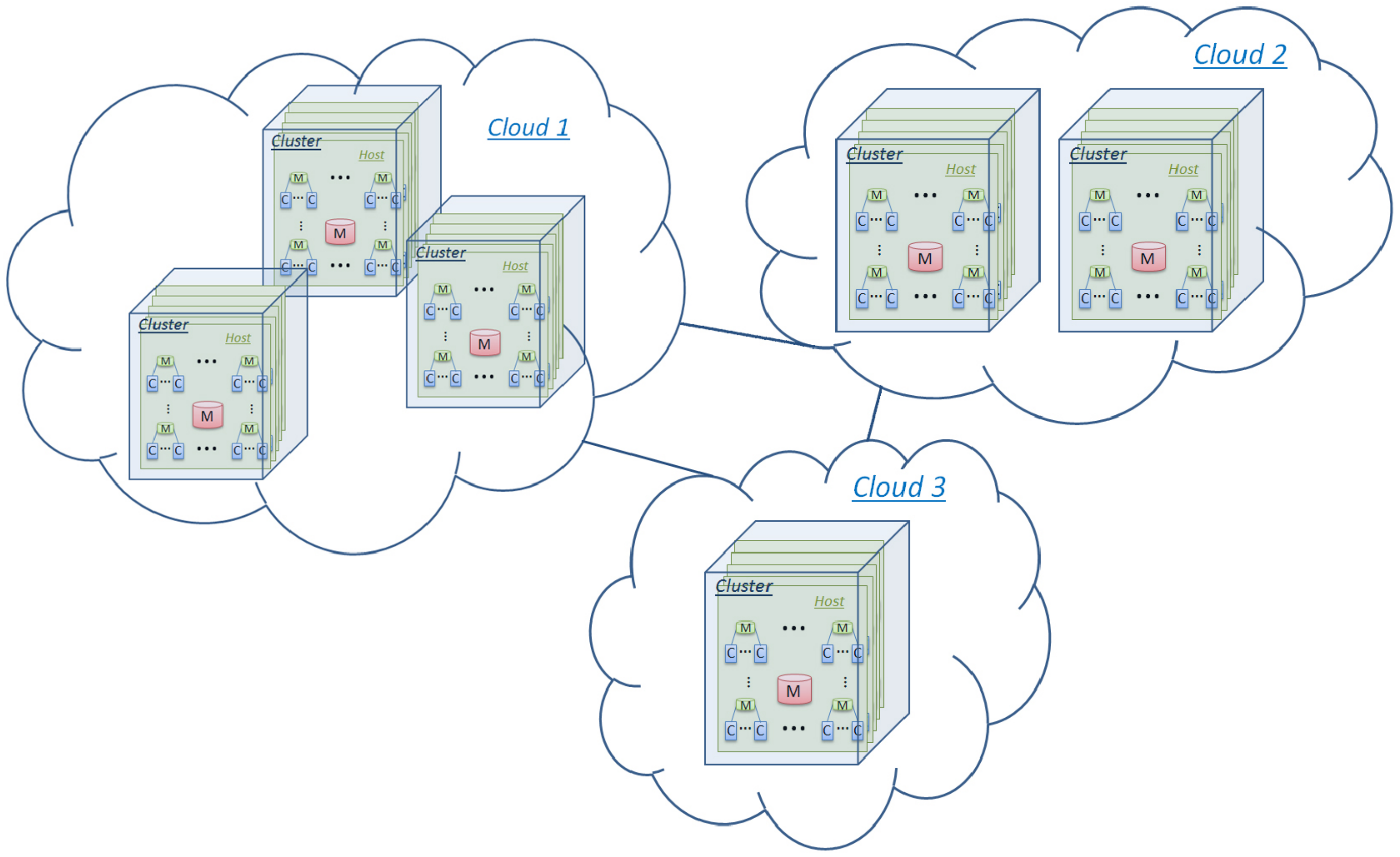

Figure 3.1: A Typical Server Architecture

Chapter 4

Anticipated Failures

To build a scalable and fault tolerant system it is important to identify types of failures and their impact on Erlang performance. Recall that Erlang approach to failures is '*Let it crash and another process will correct the error*' [Arm10].

Erlang nodes and processes may fail for various reasons. Table 4.1 presents a list of failures that may occur (columns *Type* and *Subtype*), and the sides responsible for handling the failures. Column *MTTF/BF* indicates the Mean Time To Failures/ Before Failures. Column *Implication* discusses aftereffects of the failures. Columns *Handling* and *Responsibility* provide information how the failures are dealt with and which side is responsible for the recovery. Here, by 'Erlang' and 'SD Erlang' we mean that the language should provide means to detect failures and recover from them. Column *Present* indicates whether a handling mechanism is already implemented in Erlang. In general we identify four types of failures:

- *Hardware* failures include core, communication link, hard disk, and RAM failures. When a hard disk or a RAM stops working and requires to be replaced, or a core fails we assume that we lose the whole host together with all Erlang nodes on that host. In this case the node failures must be discovered on remote hosts. When a minor hardware failure occurs it is handled by either an OS and/or a DBMS. When a node or a group of nodes become isolated from other nodes of the same group due to communication link failures Erlang provides means to treat the disconnected nodes as failed.

- By *supporting software* we mean software that is not necessarily written in Erlang but is strongly connected to a successful Erlang program operating, e.g. OS and DBMS. In case of a major OS failure the whole host fails. Thus, the node failure must be discovered on remote hosts. In case we consider to use Riak DBMS with Bitcask backend we also need to consider the following. Bitcask requires keydir directory that stores Bitcask key information to be kept in a RAM. However, as the number of cores grows the number of processes and the size of hard disks will grow as well. This in turn may lead to two possible failures: 1) a process bottleneck in attempt to access keydirs in RAM, and 2) incompatibility of RAM and keydir sizes. Further discussion on scalable persistent data structures is presented in Section 6.2. We assume that the remaining failures are resolved on the level of supporting software without disruption of Erlang programs.

- By *Erlang components* we mean the following: a group of nodes, a node, a group of processes, a process, and ETS tables. Erlang component failures are caused by internal component errors. To accelerate the recovery process the component failures should be discovered locally. Thus, it is preferably that a process failure is picked up by another process of the same group or the same node, and a node failure is picked by another node of the same group or on the same host. A discussion on ETS tables is provided in Section 6.1.

- In *other failures* we include failures which are out of scope for the current project due to a very small probability and a very large scale of the events. These are cases, for example, when a whole server or a cloud fails.

For the current project the main interest is in the failures handled by Erlang and Erlang users. Table 4.1 shows that Erlang already provides the means to support the majority of the failures. However, as we introduce new properties to SD Erlang, such as a notion of *scalable groups of nodes* the design of these schemes will depend on the architecture and group functionality that we introduce. ETS tables may also require some modifications to support consistency in the environment of higher magnitude of cores, nodes, and processes.

No.	Type	Subtype	MTTF/BF	Implication	Handling	Responsib.	Present
1	Hardware	Core	11 years [WBPB11], 7 years [SABR05]	The whole host fails together with all nodes placed on that host	Discovery of failed nodes and their recovery on remote hosts on remote hosts	Erlang	Yes
		Communication link		(minor) a hardware object is still connected to other components via remaining links	OS should deals with the issue	OS	Yes
				(major) a node or a group of nodes become isolated from other nodes of the same group	The isolated nodes are treated as failed	Erlang	Yes
		Hard disk		(minor) a disk continues to work and responds to the subsequent requests	OS and DBMS should deal with this issue	OS, DBMS	Yes, Yes
			1.2 Mhours [SG07]	(major) a disk needs to be replaced, as a result the whole host fails together with all nodes placed on that host	DBMS should ensure data duplication on remote hosts. Failed nodes should be discovered on remote hosts.	DBMS, Erlang	Yes, Yes
		RAM		(minor) a RAM continues to work properly and responds to the subsequent requests	OS should deal with the issue	OS	Yes
				not enough memory – a software element (process, node, DBMS, OS) that attempts to access additional memory will terminate	If a process terminates the failure should be detected at the same node and the process restored	Erlang	Yes
					If a node terminates the failure should be detected at the same host or at a remote host, and the node restored	Erlang	Yes
					If a part of DBMS effected the DBMS should be able to restore data kept in the RAM	DBMS	Yes
					If the OS terminates the same actions should be taken as in occurrence of a major OS failure	DBMS, Erlang	Yes, Yes
				(major) a RAM needs to be replaced, as a result the whole host fails together with all nodes placed on that host	DBMS should ensure data duplication on remote hosts. Failed nodes should be discovered on remote hosts.	DBMS, Erlang	Yes, Yes

Table 4.1: Types of Failures and Failure Handling

No.	Type	Subtype	MTTF/BF	Implication	Handling	Responsib.	Present
2	Supporting Software	OS		(minor) Some elements (processes, nodes, DBMS) may fail	If a process fails, the failure should be detected at the same node and the process restored	Erlang	Yes
					If a node fails the failures should be detected at the same host or at a remote host, and the node restored	Erlang	Yes
					If a part of DBMS fails the DBMS should be able to detect the failure and heal itself	DBMS	Yes
				(major) the whole host fails together with all nodes placed on that host	Quick discovery of the failed nodes on remote hosts and their recovery on remote hosts	Erlang	Yes
		DBMS		(minor) Data corrupted or a part of a DBMS is not available	DBMS should be able to detect and restore itself	DBMS	Yes
				(major) RAM is not large enough to keep all Riak-Bitcask keydirs	We need either propose modification to the current model or choose another DBMS	DBMS	No
3	Erlang Components	node or group of nodes		A node or a group of nodes fails together with all corresponding processes	The failure should be detected and restored either on the same hosts or on remote hosts	SD Erlang	Yes (partially)
		Group of Processes		The failure assumes that the corresponding nodes continue their execution. Processes of different nodes can belong the same group.	The failure should be detected by other processes and restored on the same nodes.	Erlang	Yes
		Process		The failure assumes that the corresponding node continues its execution	The failure should be detected by other processes of the same node or the same group, and restored on the same node	Erlang	Yes
		ETS		The more cores and processes share the same ETS the higher the probability of an ETS failure.	ETS tables require more consistency (public ETS tables) or/and means to cope with possible bottlenecks (private & protected ETS tables)	SD Erlang	No
4	Other Failures	Server, Cloud		There is a very small probability that the whole server or a cloud may fail	The problem is out of the project scope	-	-

Table 4.2: Types of Failures and Failure Handling

Chapter 5

Scalable Computation

In this section we discuss how distributed Erlang can be extended to SD Erlang to effectively operate when the number of hosts, cores, nodes, and processes are scaled. We start by introducing scalable groups (s_groups) in Section 5.1 and semi-explicit placement in Section 5.2. Then we discuss high level abstractions in Section 5.3. Finally, SD Erlang security issues in LANs and WANs are covered in Section 5.4.

5.1 Network Scalability

To allow scalability of networks of nodes the existing scheme of transitive connection sharing should be changed as it is not feasible for a node to maintain connections to tens of thousands of nodes, i.e. the larger the network of Erlang nodes the more 'expensive' it becomes on each node to keep up-to-date replications of global names and global states, and periodic checking of connected nodes. Instead we propose to use partial connections where nodes would have transitive connections within their scalable group (s_group) and non-transitive connections with nodes of other s_groups. Collections of s_groups may be presented by one of the following schemes:

- *hierarchical (or recursive)* scheme allows subgroups but does not allow direct communication between nodes from different s_groups and different levels. The scheme presupposes that all communications happen via gateway nodes unless the nodes are directly connected ang belong to the same s_group.

- *overlapping* scheme seems the one that is both the most flexible and the best fitting with the Erlang philosophy, i.e. *any node can directly connect to any other node.* Another advantage of the overlapping scheme is that it can also be used to construct both partition and hierarchical structures.

- *partition* scheme is not flexible, and is a particular case of the overlapping scheme when nodes are allowed to belong to only one s_group.

Below we discuss overlapping s_groups and essential functions to support them.

5.1.1 Overlapping s_groups

The s_grouping implies that a node has transitive connections with nodes of the same s_group, and non-transitive connections with other nodes. In the overlap s_grouping the nodes can belong to more than one s_group. This will allow the following properties.

- A node can have diverse roles towards other nodes, e.g. a node may be an executing node in one s_group and a gateway in another s_group.
- A node can have diverse connectivity, i.e. some nodes may have more connections than then others. In case a node belongs to a local s_group and to a cloud s_group depending on the amount of process work the node may spawn processes to a local node or to a node in a cloud.
- Nodes can be dynamically added and removed from an s_group, e.g. when certain conditions are met nodes are registered as s_group members and processes can be spawned those nodes.
- A network may have *only transitive* connections when all nodes belong to a single s_group or *only non-transitive* connections when each node belongs to a separate s_group.
- A network of nodes may be structured in a hierarchical, partition, or overlapping manner. Examples are discussed in Section 7.

Nodes with no asserted s_group membership will belong to a notional group *G0* that will allow backward compatibility with Distributed Erlang. We believe that for compatibility reasons group *G0* should not be an s_group and thus should follow Distributed Erlang rules. Therefore, s_groups are not compulsory but rather a tool a programmer may use to allow network of nodes scalability. By the backward compatibility we mean that when nodes run the same VM version they may use or not use s_groups and still be able to communicate with each other. If a node belongs to an s_group it follows s_group rules, i.e. has transitive connections with nodes of the same s_group and non-transitive connections with other nodes. If a node does not belong to an s_group it follows Distributed Erlang rules, i.e. has transitive connections with nodes that also do not belong to any s_group, and non-transitive connections with other nodes.

When a node leaves group $G0$ or an s_group the node connections can be either kept or lost.

- *Losing connections.* As soon as a node becomes a member of at least one s_group it looses a membership and all connections with nodes of group *G0*, and establishes new connections in its s_group. The same way when a node leaves an s_group it looses connections with nodes of that s_group. In case the node does not belong to any other s_group it becomes a member of group *G0* and establishes new connections with the group nodes.
- *Keeping connections.* When a node becomes a member of at least one s_group it does not loose connections with nodes of group *G0*, rather from transitive the connections become direct. The same way when a node leaves an s_group all connections remain but become direct.

We propose to implement the keeping connection alternative.

Evidently, global names and global locks cannot be kept in SD Erlang. Therefore, we propose to transform them into s_group names and s_group locks. To avoid a collision of s_group names with the same name $Name$ on node A that belongs to s_groups $G1$ and $G2$ we propose the s_group names to consist of two parts: a name and an s_group name. Then node A will treat names as $Name@G1$ and $Name@G2$.

In distributed Erlang hidden global groups have visible (transitive) connections with the nodes of the same global group, and hidden (non-transitive) connections with nodes of other global groups. The idea of s_groups is *similar* to the idea of hidden global_groups in the following: 1) each s_group has its own name space; 2) transitive connections are only with nodes of the same s_group. The *differences* with hidden global_groups are in that 1) a node can belong to an unlimited number of s_groups, and 2) information about s_groups and nodes is not globally collected and shared.

5.1.2 Types of s_groups

To allow programmers flexibility and provide an assistance in grouping nodes we propose s_groups to be of different types, i.e. when an s_group is created a programmer may specify parameters against which a new s_group member candidate can be checked. If a new node satisfies an s_group restrictions then the node becomes the s_group member, otherwise the membership is refused. The following parameters can be taken into account: communication distance, security, available code, specific hardware and software requirements. We may consider the following options for the proposed types of s_groups.

1. Security:
   (a) only an s_group leader node may add nodes to the s_group.
   (b) member nodes may belong to only one s_group and cannot initiate an s_group themselves.

Type of Information	**Description**
Node information	node name and a list of s_group names to which the node belongs to
s_group information	s_group name and a list of nodes that belong to the s_group
s_group list information	a list of s_groups

Table 5.1: Types of Global Information

   (c) s_group members can only be connected to the nodes of the same s_group.

2. Locality:

   (a) only a node within a particular communication distance can be a member of an s_group (distance can be calculated from the s_group leader node or from all currently participating nodes).

3. Special hardware:

   (a) whether the location of the new member of the s_group satisfies a particular hardware configuration.

4. Special software:

   (a) whether a particular software or code is available on the new node.

The information about specific resources can be collected by introducing node self awareness, i.e. a node is aware of its execution environment and publishes this information to other nodes. A programmer may also introduce his/her own s_group types on the basis of some personal preferences. Communication distance can be measured in a number of ways, i.e. from a particular node such as a leader s_group node or a node that performs the connection, or from every node of the s_group. The design chosen for SD Erlang implements only custom s_group types.

Table 5.1 presents information that we are interested to maintain to exploit s_group advantages in SD Erlang: node information, s_group information, and s_group list information. Obviously for scalability reasons nodes cannot maintain node and s_group information about every other node and s_group. Therefore, one the following approaches can be implemented.

- *P2P distribution.* Each node maintains global s_group list information and node information about nodes of the same s_groups. The disadvantage of this option is that although information is not renewed often it is required to be replicated on all 10^5 nodes. Another possible problem can be in that all nodes maintain global information, i.e. there may be a surge of global s_group information update in the beginning when new s_groups are formed and the information is replicated between the nodes. However, after the initial stabilising there should not be many updates between the nodes.

- *Gateway-based distribution.* Each node maintains node information about nodes of the same s_groups, and s_group leader nodes also maintain s_group list information. The advantage of this method is that there is no globally shared information. The difficulty with the option is that it requires replication mechanisms for the gateway nodes to prevent s_group isolation in case some gateway nodes fail. It also requires an additional effort from the programmers to ensure that gateway nodes are sufficiently connected to each other to support s_group list information distribution and to avoid network partition.

We propose to implement an optional *P2P distribution* approach, i.e. initially the programmer specifies whether global information is collected or is not collected for the network of nodes. Despite the side effects an implementation of the *P2P distribution* approach is relatively easy, the s_group information after stabilising is not changed often, and no effort from a programmer is required to ensure the information distribution.

Communication between nodes that belong to different s_groups may be implemented in the following ways.

- A node establishes connection with the target node. In this case nodes may have a large number of connections with nodes from outside of their s_groups. To eliminate the problem we propose to introduce short lived connections.

- A node passes messages via s_group leader nodes. In this case we keep the number of connections unchanged but this is against the Erlang philosophy that any node may connect to any other node. This also implies an additional load on the s_group leader nodes as all intergroup communication will pass through them.

The design chosen for SD Erlang is as follows. A node can establish a direct connection with any other node and introduce short lived connections. The number of nodes per host will be unrestricted, and we do not consider any language constructs to provide programmers any level of control over cores, i.e. the lowest level a programmer may control in terms where a process will be spawned is a node.

5.1.3 s_group Functions

We propose the following functions to support s_group employment. These functions may be changed during the development. The final impelemntation will be decided during actual SD Erlang code writing and will depend on which functions programmers find useful.

1. Creating a new s_group, e.g.
   `new_s_group(S_GroupName, [Node]) -> ok | {error, ErrorMsg}`

2. Deleting an s_group, e.g.
   `del_s_group(S_GroupName) -> ok | {error, ErrorMsg}`

3. Adding new nodes to an existing s_group, e.g.
   `add_node_s_group(S_GroupName, [Node]) -> ok | {error, ErrorMsg}`

4. Removing nodes from an existing s_group, e.g.
   `remove_node_s_group(S_GroupName, [Node]) -> ok | {error, ErrorMsg}`

5. Monitoring all nodes of an s_group, e.g.
   `monitor_s_group(S_GroupName) -> ok | {error, ErrorMsg}`

6. Sending a message to all nodes of an s_group, e.g.
   `send_s_group(S_GroupName, Msg) -> Pid | {badarg, Msg} | {error, ErrorMsg}`

7. Listing nodes of a particular s_group, e.g.
   `s_group_nodes(S_GroupName) -> [Node] | {error, ErrorMsg}`

8. Listing s_groups that a particular node belongs to, e.g.
   `node_s_group_info(Node) -> [S_GroupName]`

9. Connect to all nodes of a particular s_group, e.g.
   `connect_s_group(S_GroupName) -> [boolean() | ignored]`

10. Disconnect from all nodes of a particular s_group, e.g.
    `disconnect_s_group(S_GroupName) -> boolean() | ignored`

Using the above functions merging and splitting of s_groups may be also implemented.

Example: Assume we start six nodes A, B, C, D, E, F, and initially the nodes belong to no s_group. Therefore, all these nodes belong to notional group $G0$ (Figure 5.1(a)). First, on node A we create a new s_group $G1$ that consists of nodes A, B, and C, i.e. `new_s_group(G1, [A, B, C])`. Note that a node belongs to group $G0$ only when this node does not belong to any s_group. When nodes A, B, and C become members of an s_group they may still keep connections with nodes D, E, F but now connections with these nodes are non-transitive. If connections between nodes of s_group $G1$ and group $G0$ are time limited then the non-transitive connections will be lost over some time (Figure 5.1(b)). Then on node C we create s_group $G2$

that consists of nodes C, D, and E. Nodes D, and E that now have non-transitive connections with node F may disconnect from the node using function `disconnect_s_group(G0)`. Figure 5.1(c) shows that node C does not share information about nodes A and B with nodes D and E. Similarly, when nodes B and E establish a connection they do not share connection information with each other (Figure 5.1(d)).

5.1.4 Summary

To enable scalability of network of nodes we have proposed the following modifications.

- Grouping nodes in s_groups where s_groups can be of different types, and nodes can belong to many s_groups.
- Transitive connections between nodes of the same s_group and non-transitive connections with all other nodes. Direct non-transitive connections are optionally short lived, e.g. time limited.
- Optional global s_group list information.
- Replacing global names with s_group names where each s_group names consists of two parts: a name and an s_group name, e.g. $Name@Group$.

5.2 Semi-explicit Placement

For some problems, like matrix manipulations, optimal performance can be obtained on a specific architecture by explicitly placing threads within the architecture. However, many problems do not exhibit this regularity. Moreover, explicit placement prevents performance portability: the program must be rewritten for a new architecture, a crucial deficiency in the presence of fast-evolving architectures. We propose a dynamic semi-explicit and architecture aware process placement mechanism. The mechanism does not support the migration of processes between Erlang nodes. The semi-explicit placement is influenced by Sim-Diasca (Section 7.1) process placement and architecture aware models [ATL]. In Sim-Diasca a computing load is induced by a simulation and needs to be balanced over a set of nodes. By default, model instances employed by Erlang processes are dispatched to computing nodes using a round-robin policy. The policy proved to be sufficient for most basic uses, i.e. a large number of instances allows an even distribution over nodes. However, due to bandwidth and latency for some specifically coupled groups of instances it is preferable for a message exchange to occur inside the same VM rather than between distant nodes. In this case, a developer may specify a placement hint when requesting a creation of instances. Usually, placement hints are atoms. The placement guarantees that all model instances created with the same placement hint are placed on the same node. This allows the following: 1) to co-allocate groups of model instances that are known to be tightly coupled, 2) to preserve an overall balancing, and 3) to avoid model level knowledge of the computing architecture.

To limit the communication costs for small computations, or to preserve data locality, [ATL] proposes to introduce communication levels and specify the maximum distance in the communication hierarchy that the computation may be located. So sending a process to level 0 means the computation may not leave the core, level 1 means a process may be located within the shared memory node, level 2 means that process may be located to another node in a Beowulf cluster, and level 3 means that a process may be located freely to any core in the machine.

For the SD Erlang we propose that a process could be spawned either to an s_group, to s_groups of a particular type, or to nodes on a given distance. From a range of nodes the target node can be picked either randomly or on the basis of load information. Thus, a programmer will have the following options to spawn a process.

1. An explicit placement, i.e. a programmer will specify a particular node.
2. A semi-explicit placement where either an s_group, s_group type, or distance are defined.
   (a) When an s_group is defined the process will be spawned to a node of a particular s_group.
   (b) When an s_group type is defined the process will be spawned to a node of an s_group of a particular type.
   (c) When a distance is defined the process will be spawned to a connected node within a certain distance.

Initially, we do not plan to implement the case when both s_group/s_group type and a distance are defined.

An s_group leader node by default is the node that creates a corresponding s_group. The s_group leader nodes can be changed. The leader nodes have the following responsibilities.

1. *Representing an s_group in the s_group list.* As nodes cannot collect information about every other node in the network all nodes will know information about s_groups, i.e. s_group names. But to be able to reach an s_group the information about s_group leader nodes will be also provided.
2. *Responding to distant s_group spawn requests.* When an s_group leader node receives a spawning request from a distant node it will respond with the list of s_group members that satisfy the request conditions.

Communication Distances. We distinguish relative and communication distances. A small *relative distance* implies that nodes are in the same s_group, and a small *communication distance* implies that nodes are on the same host. Thus, nodes that have a small relative distance (i.e. are in the same s_group) may have a large communication distance (e.g. are placed in different clouds), and vice versa.

We have discussed relative distances in Section 5.1. The communication distances can be expressed using *communication levels*, i.e. numbers from 0 to the adopted number of levels. We propose to distinguish levels from one another on the basis of *communication time* where communication levels are defined by latencies of some small messages, i.e. the messages will be sent between the nodes and depending on the message latencies it will be decided on which communication levels the nodes are situated from each other. The method will allow compatibility of different systems because independently of a network architecture communication levels will stay the same.

A difficulty may arise in communicating the idea of distances to programmers to help them to decide to which levels they would like to spawn their processes. However, if carefully measured, calculated, and documented, this method seems to be the most promising due to its portability. In the future if smaller communication time will be introduced in comparison to the initial level 0 either levels -1, -2,... may be introduced or the table may be shifted to the right allowing values of level 0 to be smaller.

Another issue to consider is whether the levels should be introduced uniformly and built-in in the load manager or programmers should be allowed to set their own communication levels.

- In case a level calculation is *built-in* then programs can be easily transferred from one hardware architecture to another, i.e. be independent from the underlying hardware. But at the same time predefined communication levels will restrict programmers to the levels that we think are right independently of the particular architectures they are working on.

- On the other hand if each programmer decides to set his/her *own communication levels* programs built in SD Erlang become non-portable, as every time the program moves from one architecture to another the levels should be adjusted. Even more headache will be caused by a system that consists of a few parts where each part was built on a different architecture from the merging one.

We propose to implement communication distances using communication levels where levels are calculated on the basis of communication time. We also propose to provide calculated level measurements and document them in an intuitive and easily understandable way but to allow users to change these parameters if needed notifying about problems the changes may cause.

node@host	**S_Groups**	**Level**	**Host Load**	**Timestamp**
A	G1	1	32 processes per node	
B	G1	1	12 processes per node	
D	G2	2	14 processes per node	
E	G2	3	27 processes per node	

Table 5.2: Load Information

Load Management. When a node is picked on the basis of load an important design decision is the interaction between two main load management components, i.e. information collection component and decision making component. The components can be either merged together and implemented as one element or implemented independently from each other. We propose to implement information collection and decision making as one element, i.e. a load server. Its responsibility will be collecting information from the connected nodes and deciding where a process can be spawned to when a corresponding request arrives. It seems that *one load server per node* is the right number for SD Erlang. In this case the decisions are made within the node (apposed to one load server per s_group, a host, and a group of hosts) and load information redundancy level is not too high (apposed to one per group of processes and a multiple number of load servers per node). A high level representation of a node is shown in Figure 5.2 where *P1, P2,..., PN*, and *Load Manager* are processes.

In Table 5.2 we present information that may be collect in the load manager of node *C* (Figure 5.3). Column *S_Groups* indicates common s_groups between the current node and *node@host* node, column*Level* indicates communication distance, column *Host Load* keeps information about the load of the host on which *node@host* node is located, and column *Timestamp* shows the time of the last update.

5.2.1 `choose_node`

We propose to introduce a new function `choose_node/1` that will return a node ID where a process should be spawned. For example, a programmer can indicate the following parameters: s_groups, s_group types, and minimum/maximum/ideal communication distances.

Node	**Collects information about nodes**
A	B, C
B	A, C, E
C	A, B, D, E
D	C, E
E	B, C, D

Table 5.3: Information Collection (Figure 5.3)

The function can be written in SD Erlang as follows.

```
choose_node(Restrictions) -> node()
Restrictions = [Restriction]
Restriction = {s_group_name, S_GroupName}
| {s_group_type, S_GroupType}
| {min_dist, MinDist :: integer() >= 0}
| {max_dist, MaxDist :: integer() >= 0}
| {ideal_dist, IdealDist :: integer() >= 0}
```

We deliberatly introduce `Restrictions` as a list of tuples. This is done to allow the list of restrictions to be extended in the future. A process spawning may look as follows:

```
   start() ->
        TargetNode = choose_node([{s_group, S_Group}, {ideal_dist,
IdealDist}]),
        spawn(TargetNode, fun() -> loop() end).
```

5.2.2 Summary

To enable semi-explicit placement and load management we propose the following constructs.

- Function `choose_node(Restrictions) -> node()` where the choice of a node can be restricted by a number of parameters, such as s_groups, s_group types, and communication distances.
- The nodes may be picked randomly or on the basis of load.
- Communication levels on the basis of communication time.

Assumptions. We assume when a process is spawned using semi-explicit placement it is a programmer responsibility to ensure that the prospective target node has the required code. If the code is missing an error is returned.

5.3 Scalable Distribution Abstractions

Erlang follows a functional programming idiom of having a few primitives and building powerful abstractions over them. Examples of such abstractions are algorithm skeletons [Col89] that abstract common patterns of parallelism, and behaviour abstractions [Arm03] that abstract common patterns of distribution.

We plan to develop SD Erlang behaviour abstractions over primitives presented in Sections 5.1 and 5.2. For example, s_group supervision and master/slave behaviours. In the s_group supervision abstraction a leaf may be represented by an s_group, and an s_group may supervise another s_group or a number of s_groups. The supervision behaviour may also include such abstractions as taking over a failed s_group or monitoring s_group leaders and replacing them if needed. In the s_group master/slave behaviour a master s_group may dispatch work to an s_group or a number of s_group slaves.

As we develop case studies we shall seek scalable distribution abstractions. The work has a strong connection with the ParaPhrase project which is developing algorithmic skeletons for Erlang.

5.4 Security

The proposal explicitly does not address security issues. However, applications such as Moebius (Section 7.4) and Riak (Section 7.5) even in a small scale need to spread nodes over a wide area network (WAN) where connections cannot be supported by the Erlang VM due to security issues. The scale of the security problems can be divided into a LAN scale and a WAN scale. In a LAN the target application is behind a firewall; therefore, security can be handled by cookies or on the TCP level by opening dedicated ports. Over specific connections data can be sent encrypted. In a WAN the parts of an application are behind different firewalls.

Currently, Riak and Moebius provide their own security mechanisms. Riak's RESTful is a REpresentational State Transfer web service. The RESTful parameters are free-format text and are carried via HTTP POST request. RESTful web services normally do not include service-level security mechanisms but rely on standard defences like firewall to filter critical HTTP requests. [For09] discusses web-security architecture model and REST arguing that caching mechanisms may cause security problems. To overcome the problem the authors introduces a design of hierarchical content protection keys.

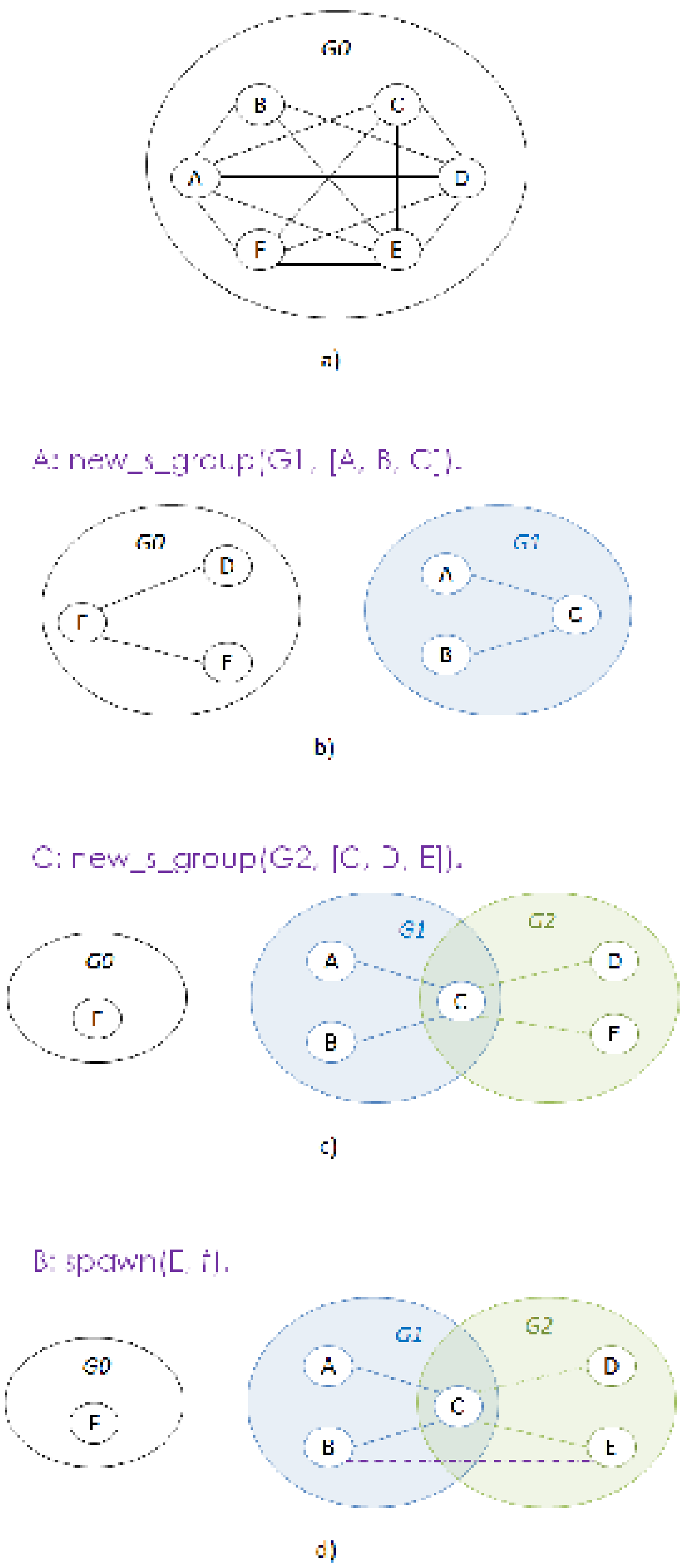
G0
A
B
C
D
E
F
a)
A: new_s_group(G1, [A, B, C]).
G0
G1
b)
C: new_s_group(G2, [C, D, E]).
G0
G1
G2
c)
B: spawn(E, f).
G0
G1
G2
d)

Figure 5.1: Connections in s_groups

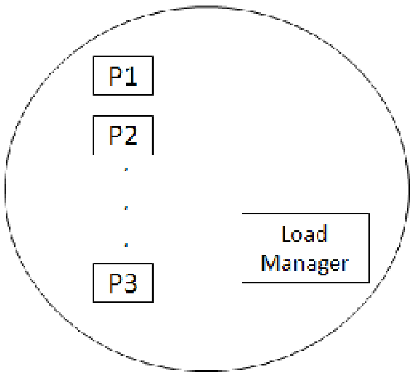

Figure 5.2: Load Manager and Processes within a Node

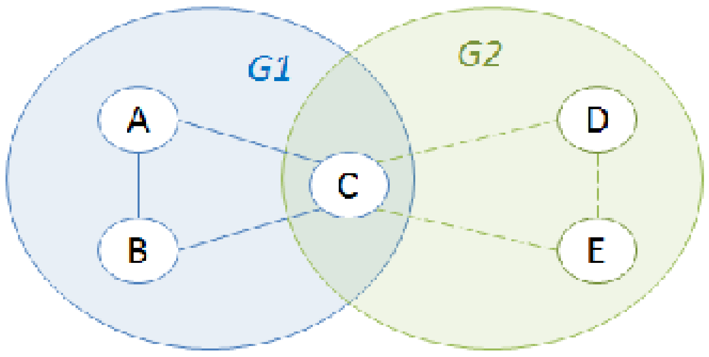

Figure 5.3: Node Connections Using s_groups

Chapter 6

Scalable Data Structures

In this section we discuss scalable data structures. Scalable in-memory data structures are covered in Sections 6.1 and scalable persistent data structures are covered in Section 6.2.

6.1 Scalable In-memory Data Structures

For scalability the primary Erlang in-memory data structure is ETS tables.

6.1.1 ETS

Erlang Term Storage (ETS) is a data structure to associate keys with values, and is a collection of Erlang tuples, i.e. tuples are inserted and extracted from an ETS table based on the key. An ETS is memory resident and provides large key-value lookup tables. Data stored in the tables is transient. ETS tables are implemented in the underline runtime system as a BIF inside the Erlang VM, and are not garbage collected.

ETS tables can be either ordered or unordered. Access time for tables with ordered and unordered keys are $O(logN)$ and $O(1)$ respectively where N is the number of objects stored in the table. An unordered ETS table is implemented using a linear hash bucket array. Internally, ordered sets are represented by balanced binary trees, and the remaining types are represented by hash tables. An ETS may also have such data structures as Judy, AVL balanced binary trees, B-trees, resizable linear hash tables [Fri03]. Currently only resizable linear hash tables and AVL balanced binary trees are supported.

ETS tables are stored in a separate storage area not associated with normal Erlang process memory. The size of ETS tables depends on the size of a RAM. An ETS table is owned by the process that has created it and is deleted when a process terminates. The process can transfer the table ownership to another local process. The table can have the following read/write access [CS10]:

- *private*: only the owner can read and write the table.
- *public*: any process can read and write the table. In this case it is the user responsibility to ensure that table reading and writing are performed consistently.
- *protected*: any process can read the table but only the owner can write it.

"ETS tables provide very limited support for concurrent updates" [CT09]. That is writing a new element may cause a rehash of entire table; when a number of processes write or delete elements concurrently from the table the following outcomes are possible: a runtime error, `bad arg` error, or undefined behaviour, i.e. any element may be returned.

Another scalability obstacle can be a BIF operation `match` that is executed atomically. The operation extracts elements from the table by pattern matching. On a large table `match` operation can stop other processes from executing until the whole table has been traversed.

The only guarantees provided by ETS tables are as follows:

- All table updates are atomic and isolated. An atomic update implies that updating of a single object either succeeds or fails with no effect, and an isolated update implies that intermediate results are not available to other processes.
- Function `safe_fixtable/2` allows to call `first/1` and `next/2` and traverse the table without errors visiting each object only once.

An ETS table can have `write_concurrency` (`read_concurrency`) options being set. The options give exclusive access to the operation that performs a corresponding update of the table blocking other concurrent access to the same table until writing (reading) is finished. The options are especially useful on runtime systems with SMP support when large concurrent write (read) bursts are common. However, if a few reading operations interleave with a few writing operations the options should be used carefully as the cost of the switching operation may exceed `write_concurrency` (`read_concurrency`) gain.

A process or a number of processes can fix an ETS table calling `safe_fixtable`. This will guarantee that a sequence of first/1 and next/2 will traverse the table without errors and all objects will be visited exactly once even if objects are added to, or deleted from, the table simultaneously. The table remains fixed until all processes have released it or terminated. If a process fixes a table and never releases it the memory by the deleted object will never be free.

6.1.2 Summary

As the number of SMP cores increases the number of Erlang nodes and processes also increase. This can lead to either a bottleneck if the table is private/protected or undefined outcomes if the table is public. ETS tables are implemented in the VM, and hence any scalability issues will be addressed by other partners of the RELEASE project.

6.2 Scalable Persistent Data Structures

This section covers scalable persistent data structures. Firstly, we propose general principles of a scalable persistent storage (Section 6.2.1). These principles are defined based on the target architecture (Figure 3.1). Then we presents a survey of scalable DataBase Management Systems (DBMSs), and discuss their technologies and suitability for the target scale (Section 6.2.2). Finally, we summarize the findings (Section 6.2.3).

6.2.1 General Principles

Since the target architecture is loosely coupled, failures in such large-scale system are not an exception. Therefore, we aim to define features and principles of a highly available and scalable database.

- *Fragmenting data* across distributed nodes with respect to the following aspects:

    1. Decentralized model: Data is fragmented among nodes without any central coordination. Decentralized approaches show a better throughput by spreading the loads over a large number of servers and increases availability by removing a single point of failure.
    2. Load balancing: Fragmenting data evenly among the nodes. A desirable load balancing mechanism should take load balancing off the shoulders of developers.
    3. Location transparency: In large scale systems placing fragments in the most suitable location is very difficult to manage. Thus, the preferable method of fragment placement should be carried out systematically and automatically.
    4. Scalability: A node departure or arrival should only affect the node immediate neighbours whereas other nodes remain unaffected.

- *Replicating data* across distributed nodes with respect to following aspects:

   1. Decentralized model: Data is replicated among nodes without using a concept of a master. A P2P model is desirable because each node is able to coordinate the replication.
   2. Location transparency: The placement of replicas should be handled systematically and automatically.
   3. Asynchronous replication: Consistency is sacrificed to achieve more availability.

- *Partition tolerance*: System continues to operate despite loss of connection between some nodes. The CAP theorem [HJK$^+$07] states a database cannot simultaneously guarantee the consistency, availability, and partition-tolerance. We anticipate the target architecture to be loosely coupled; therefore, partition failures are highly expected. By putting stress on availability we must sacrifice strong consistency to achieve partition-tolerance and availability.

6.2.2 Initial Evaluation

In this section we discuss the most popular data storage systems in Erlang community.

Mnesia is a DBMS written in Erlang for industrial telecommunications applications [AB12]. A schema is a definition of all tables in a database. A Mnesia database can reside in RAM. In this case stored information is not persistent and tables need to be created again after restarting the system. A memory-only database which is kept in RAM doesn't need schema. Mnesia data model consists of tables of records and attributes of each record can store arbitrary Erlang terms.

Mnesia provides Atomicity, Consistency, Isolation and Durability (ACID) transaction which means either all operations in a transaction are applied to all nodes successfully, or in case of a failure the operations do not have any effect on the nodes. In addition Mnesia guarantees that transactions which manipulate the same data records do not interfere with each other. To read and write from/to a table through transaction Mnesia sets and releases locks automatically.

Listing 6.1: sample code which shows explicit placement of replicas

```
mnesia:create_table(student, [{disc_copies, [node1@mydomain,
    node2@mydomain, node3@mydomain]},{type, set}, {attributes,[
    id,fname,lname,age]},{index,[fname]}]).
```

Fault-tolerance is provided in Mnesia by replicating tables on different Erlang nodes [AB12]. To create a new table a programmer should specify the name of nodes, i.e. the placement of replicas should be mentioned explicitly. This can be a very difficult to manage task for a programmer in a large-scale architecture.

Generally, to read a record only one replica of that record is locked usually a local one, but to write a record all replicas of that record are locked. Table replication brings two advantages [AB12]. First, in terms of fault tolerance, the table is still available if one of the replicas fails. The second advantage is an improvement in performance because all the nodes which have a table replica are able to read data from that table without accessing the network. The replication can improve a performance during data reading because network operations are considerably slower than local operations. However, replication can become a disadvantage when frequent writing to the table is required. Since every write operation must update all replicas it may be a time consuming task. To alleviate an overhead of transaction processing Mnesia offers dirty operations that manipulate tables without transaction overhead. But in this situation we lose atomicity and isolation properties of operations. Fortunately, in dirty operation we have a certain level of consistency, i.e. each individual read and write operation on a single replica is atomic.

Mnesia also has a limitation on the size of tables. The Mnesia design emphasises a memory-resident database by offering `ram_copies` and `disc_copies`, because `disc_only_copies` tables are slow. Thus, keeping big tables reduces the size of available memory. In addition, since Dets tables use 32 bit integers for file offsets the largest possible Mnesia table is 2Gb.

To cope with large tables, Mnesia introduces a concept of table fragmentation. A large table can be split into several smaller fragments. Mnesia uses a hash function to compute a hash value of a record key. Then, the hash value is used to determine the name of the table fragment. In the fragmentation the placement of fragments must be mentioned explicitly.

Listing 6.2: sample code which shows explicit placement of fragments

```
mnesia:change_table_frag(SampleTable, {add_frag, List_of_Nodes
    }).
```

In summary, there are some limitations in Mnesia for large-scale systems, such as:

- an explicit placement of replicas,
- an explicit placement of fragments,
- a limitation in the size of tables, and
- a lack of support for eventual consistency.

CouchDB (Cluster Of Unreliable Commodity Hardware) is a schema-free document-oriented database written in Erlang [Len09]. Data in CouchDB is organised in a form of a document. Schema-less means that each document can be made up of an arbitrary number of fields. A single CouchDB node employs a B-tree storage engine that allows search, insertion, and deletion handled in logarithmic time. Instead of traditional locking mechanisms for concurrent updates CouchDB uses Multi-Version Concurrency Control (MVCC) to manage concurrent access to the database. MVCC makes it possible to run at full speed all the time even when a large number of clients use the system concurrently. View creations and aggregation reports are implemented by joining documents using a map/reduce technique.

A data fragmenting over nodes is handled by *Lounge* [ALS10]. Lounge is a proxy-based partitioning/clustering framework for CouchDB. Lounge applies a hash function on the document's ID to identify a shard where the document is saved. A hash function is a consistent hash which balances the storage loads evenly across the partitions. Lounge does not exist on all CouchDB nodes. In fact, Lounge is a web proxy that distributes HTTP requests among CouchDB nodes. Thus, to remove single points of failure we need to run multiple instances of the Lounge.

A CouchDBs replication system synchronizes all the copies of the same database by sending the last changes to all the other replicas. Replication is a unidirectional process, i.e. the changed documents are copied from one replica to the others but the reverse process is not automatic. The replicas placement should be handled explicitly.

Listing 6.3: sample code which shows explicit placement of replicas

```
POST /_replicate HTTP/1.1
{"source":"http://localhost/database","target":"http://example.
    org/database", continuous":true}
```

In Listing 6.3 `"continuous":true` means that CouchDB will not stop the replication and automatically send any new changes of the source to the target by listening to the CouchDBs `_changes` API. A conflict situation occurs when a document has different information on different replicas. CouchDB does not attempt to merge the conflicting revision automatically and resolving the conflict should be handled by the application. CouchDB has eventual consistency, i.e. document changes are periodically copied between replicas. A synchronization between nodes within a cluster is possible by means of an automatic conflict detection mechanism.

In summary, CouchDB has some limitations which could be a bottleneck for a large-scale systems, such as:

- an explicit placement of replicas,
- an explicit placement of fragments, and

- a multi-server model for coordinating fragmentation and replication.

Riak is a NoSQL, open source, distributed key/value data store primarily written in Erlang [Tec12]. Riak is highly scalable database suitable for large-scale distributed environments, such as a cloud. Riak is fault-tolerant due to its master-less structure offering a no single point of failure design. Riak is commercially proven and is being used by large companies and organisations such as Mozilla, Ask.com, AOL, DotCloud , GitHub. In Riak data is organized into buckets, keys, and values. A key/value pair is stored in a bucket, and values can be identified by their unique keys. Data fragmenting is handled by means of a consistent hashing technique. The consistent hash distributes data over nodes dynamically adapting as nodes join and leave the system. The fragment placement is implicit and a programmer does not need to specify nodes explicitly inside the code.

Availability is provided by using a replication and a hand-off technique. By default each data bucket is replicated to 3 different nodes. Number of replica, N, is a tunable parameter and can be set per each bucket. Other tunable parameters are read quorum, R and write quorum, W. A quorum is a number of replicas that must respond to a read or write request before it is considered successful. In a hand-off technique when a node fails temporarily due to node failure or network partitioning neighbouring nodes take over the failed node's duties. When the failed node comes back up, Merkle tree is used to determine records that need to be updated. Each node has its own Merkle tree for the hosted keys. The Merkle trees reduce the data needed to be transferred to check inconsistencies among replicas. Riak provides eventual consistency, i.e. an update is propagated to all replicas asynchronously. However, under certain conditions, such as node failure or network partitions, updates may not reach to the all replicas. Riak employs vector clocks to handle such inconsistencies by reconciling the older version and the divergent version.

A default Riak backend storage is Bitcask. Although Bitcask provides a low latency, an easy backup, a restore, and is robust in the face of crashes but it has one notable limitation – Bitcast keeps all keys in RAM and thereby has some limitations as to how many values can be stored per node. For this reason, Riak users apply other storage engines to store billions of records per node. LevelDB is a fast key-value storage library written at Google, and has no Bitcask RAM limitations. LevelDB provides an ordered mapping from keys to values whereas Bitcask is a hash table. LevelDB supports atomic batch of updates. Batch of update may also be used to speed up large updates by placing them into the same batch. There is one file system directory per each LevelDB database where all database content is stored. A database may only be opened by one process at a time by acquiring a lock from the operating system. Adjacent keys are located in the same block which improves the performance. A block is a unit of transfer to and from persistent storage. Each block is individually compressed before being written to persistent storage. It is possible to force checksum verification of all data that is read from the file system. Eleveldb is an Erlang wrapper for LevelDB that is included in Riak, so there is no need to separate installation. LevelDB's read access can be slower in comparison with Bitcask because LevelDB tables are organized into a sequence of levels. Each level stores approximately ten times as much data as the level before it. For example if 10% of the database fits in memory, one seek is needed to reach the last level. But if 1% fits in memory, LevelDB will need two seeks. So using Riak with LevelDB as storage engine can provide a suitable data store for large data. Riak has the following features:

- an implicit placement of replicas,
- an implicit placement of fragments,
- Bitcask has limitations in size of tables but LevelDB has no such limitation,
- eventual consistency, and
- no single point of failure.

We continue Riak discussion in Section 7.5.

Other distributed DBMSs The Apache Cassandra is a high scalable database written in Java recommended for commodity hardware or cloud infrastructures [Fou12]. Cassandra is in use at Twitter, Cisco, OpenX, Digg, CloudKick, and other companies that have large data sets. Cassandra offers an automatic, master-less and asynchronous mechanism for replication. Cassandra has a decentralized structure where all nodes in a cluster are identical. Thus, there are no single points of failure and no network bottlenecks. Cassandra provides a ColumnFamily-based data model richer than typical key/value systems. Since both Riak and Cassandra are inspired by Amazon's Dynamo paper [HJK$^+$07], their methods for load-balancing, replication, and fragmentation are roughly the same. So, we shall nor repeat the details here. Erlang applications employ the Thrift API to use Cassandra. Cassandra has such futures as:

- an implicit placement of replicas,
- an implicit placement of fragments,
- a columnFamily-based data model,
- eventual consistency, and
- no single point of failure.

6.2.3 Conclusion

In the following paragraphs we are going to be taking a look at a brief summary of each evaluation and finally conclude the suitability of each system for the target architecture.

Mnesia: A fragment placement should be carried out explicitly in the code. Mnesia is not scalable due to lack of a mechanism to handle load balancing when a new node joins the system or when a node goes down. Mnesia provides strong consistency.

CouchDB: Fragmentation and replication are performed by a multi-server model. The placement of replicas and fragments should be carried out explicitly in the source code. CouchDB offers a good level of availability through a dynamic load balancing and asynchronous replication.

Riak: A fragment placement and replicas are automatic. Riak has a completely decentralized architecture and has no single point of failure. Riak offers availability through a dynamic load-balancing, systematic and asynchronous replication. Riak provides a tunable consistency, and uses Bitcask as a default backstore. Due to Bitcask memory limitations Riak community uses LevelDB storage to deal with a large amount of data.

We conclude that scalable persistent storage for SD Erlang can be provided by a distributed DBMS with fragmentation, replication and eventual consistency such as Riak or Cassandra.

Chapter 7

Exemplars

In this section we consider the challenges to implement five exemplars in SD Erlang: Sim-Diasca (Section 7.1), Orbit (Section 7.2), Mandelbrot set (Section 7.3), Moebius (Section 7.4), and Riak (Section 7.5).

7.1 Sim-Diasca

Sim-Diasca (SIMulation of DIscrete systems of All SCAles) is a distributed engine for large scale discrete simulations implemented in Erlang. The engine is able to handle more than one million relatively complex model instances using a hundred of cores. Currently the maximum number of Erlang nodes that was used is 32 on 256 cores.

Sim-Diasca has its own process placement (load balancer) and performance tracker applications. In the *process placement* application the decisions are made on the basis of round robin mechanism or a hint if the last is specified. The placement hint is an Erlang term (e.g. atom) that allows to create actors on a particular node. The hints are used to improve performance, e.g. frequently communicating processes are placed on the same node to reduce their communication costs. The *performance tracker* is used to monitor the system. In particular the performance tracker traces the following: 1) memory consumptions on all nodes, 2) the number of Erlang processes and instances overall and on each node, 3) the total and per-class instance count. The performance tracker is recommended to run on a user node to simplify the analysis in case the simulation crashes.

Scalable Sim-Diasca can be built on the basis of locality (Figure 7.1), i.e. the nodes are grouped in s_groups depending on how far the nodes are located from each other. Therefore, a scalable distributed Erlang Sim-Diasca may require the following.

1. The s_groups are anticipated to be relatively static, i.e. once created they most probably will stay unchanged with occasional node joining and leaving due to the node or communication failures.

2. The s_groups may be created on the basis of communication locality.

3. A programmer may need to introduce new s_group types.

4. The number of s_groups most probably will be much less than the number of nodes.

5. A semi-explicit placement will help to place frequent communicating processes close to each other.

6. Global s_group list information may be needed to connect distantly located nodes.

7. Short lived connections may help to prevent node attempts to interconnect communicating nodes in a fully connected network.

7.2 Orbit

Orbit aims to calculate subset Q_1 from set Q that is a set of natural numbers from 0 to $N < \infty$. The program has one supervisor process, a distributed hash table, and initially one working process. The hash table is initially empty and is distributed between all participating nodes. The supervisor process is responsible for spawning the first worker process and for the program termination after all possible numbers are explored.

The Orbit algorithm is as follows. A supervisor picks a random number from 0 to N and spawns the first working process. The target node where the process is spawned to is defined by the hash of the value. The worker process arrives to the target node and checks whether the value is already in the table. If the value is in the table the process returns to the supervisor, provides statistics, and terminates. Otherwise, the value is added to the table, and the worker process using a predefined function generates a new list of values. Then for each value a new worker process is spawned and the procedure is repeated.

The target node of a new worker process is predefined by its value. Therefore, the nodes can be grouped in s_groups in a hierarchical manner according to the part of the distributed table they store. Figure 7.1 presents an example of such grouping. To sent a process from node A to node N node A may establish a direct connection with node N using global s_group list information and information about node N from node F. A scalable distributed Erlang Orbit may require the following.

1. The s_groups may be created on the basis of hash table partition.

2. The s_groups are anticipated to be relatively static.

3. The number of s_groups probably will be much less than the number of nodes.

4. Global s_group list information may be required to locate nodes and establish direct connections with them.

5. Short lived connections may help to prevent node attempts to interconnect communicating nodes in a fully connected network.

7.3 Mandelbrot Set

The exemplar presents escape time algorithm – the simplest representation of the Mandelbrot set. In the current implementation a user initially defines the size of an image (in pixels) and the number of processes that perform the same computations. Thus, the program can be scaled in two dimensions: 1) scaling the size of images; 2) scaling the number of processes that perform the computation. Assume that we start the program on node A. Then a process may be spawned to a node in s_group $G1$, e.g. node B. If a process does not escape before a certain number of iterations then it can be respawned to a node in either s_group $G2$ or $G3$ (after arriving to the target node the computation will start from the beginning). S_groups $G1$ and $G2$ may be in the same cloud and can be grouped depending on the types of instances.

We may want to create an s_group on the basis of locality, so we include nodes we trust but these nodes may be nearby (the same LAN) and far away (in a cloud). Assume that we have one node (master node) that generates processes (work). After generation the processes are immediately spawned to the nearby nodes (levels 1-3) to unload the master node. On these nodes processes are executed for some predefined time and if the processes do not terminate during that time they are sent to remote nodes (levels 4-6). When the computation is finished the processes return results to the master node. A scalable distributed Erlang Mandelbrot set may require the following.

1. The s_groups are anticipated to be relatively static.

2. The s_groups may be created on the basis of communication locality.

3. The number of s_groups probably will be much less than the number of nodes.

4. A semi-explicit placement will help to place frequent communicating processes close to each other.

7.4 Moebius

Moebius is a continuous integration system recently developed by Erlang Solutions. Moebius aims to provide users an automated access to various cloud providers such as Amazon EC2. The system has two types of nodes:

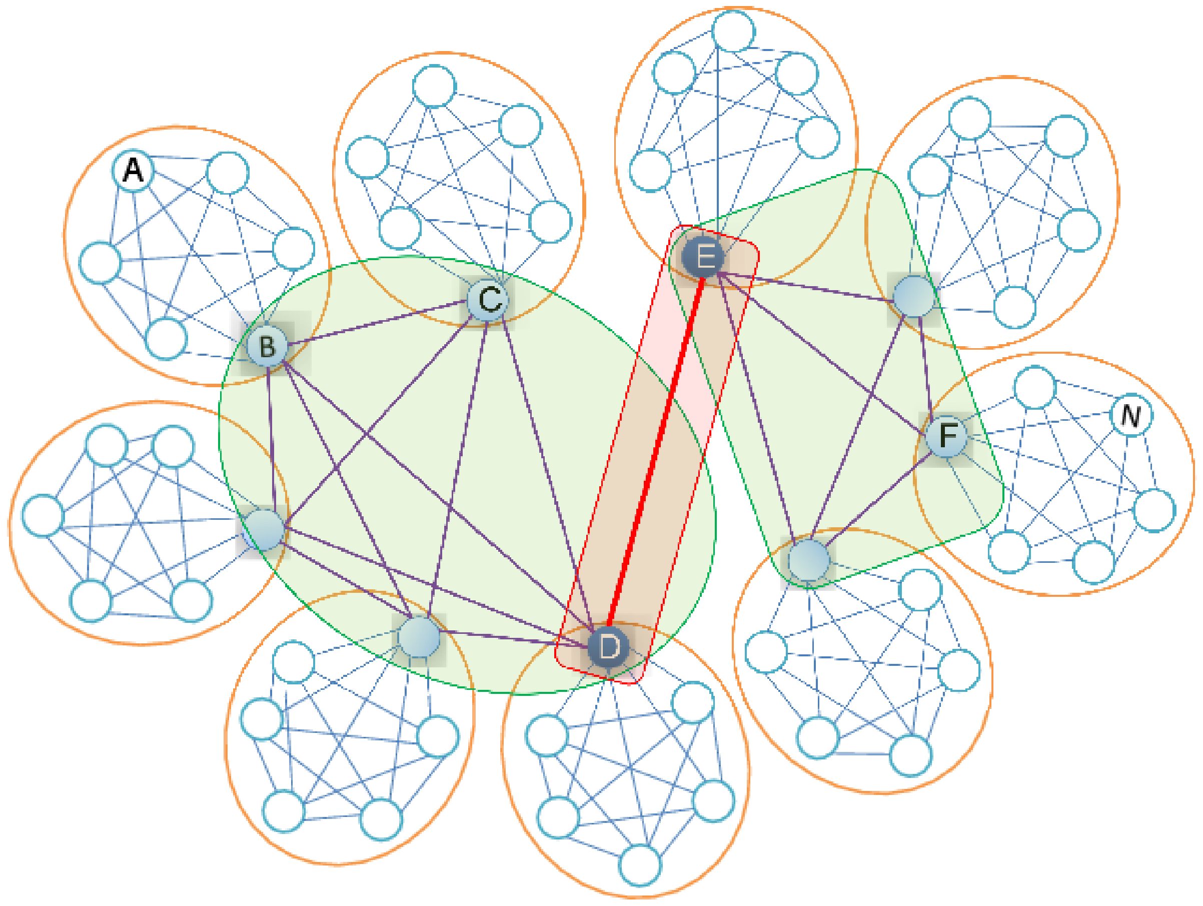

Figure 7.1: Orbit Node Grouping

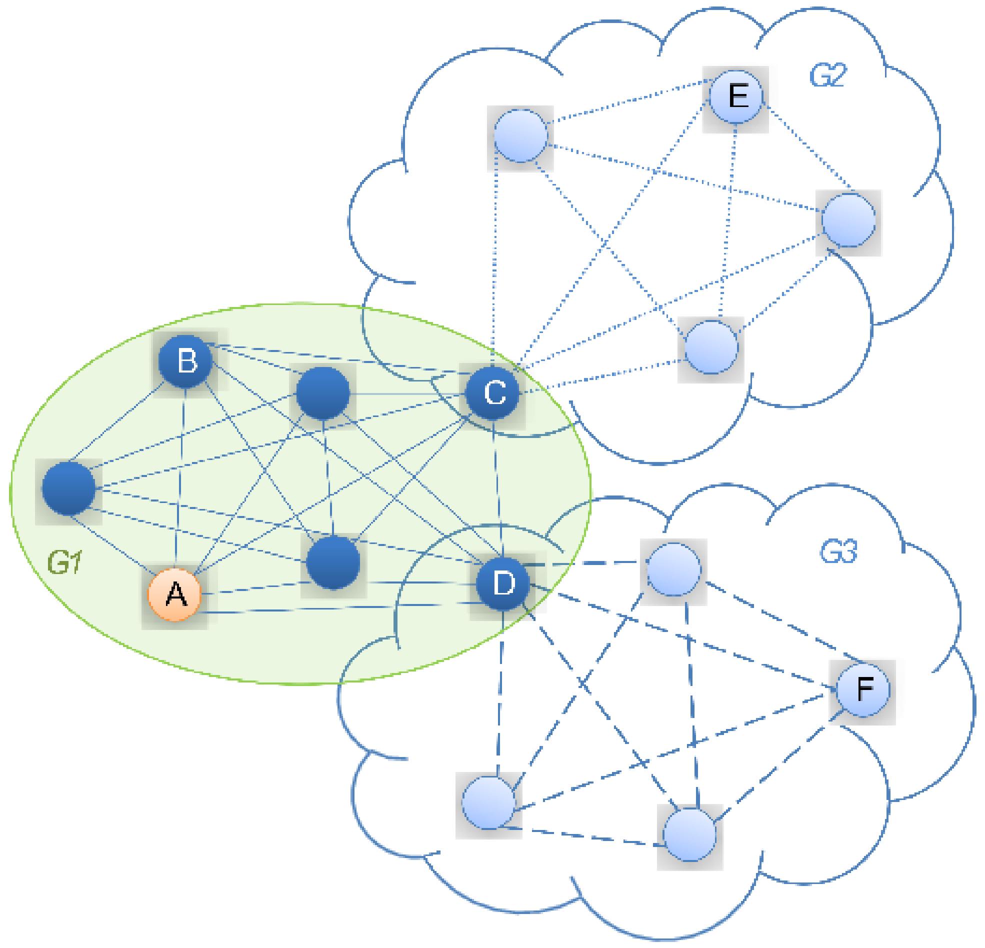

Figure 7.2: Mandelbrot Set Node Grouping

master node and moebius agents. The master node collects global information and makes decisions to start and stop nodes. The moebius agents are located on the utilised nodes and periodically send state information to the master node. Currently, moebius agents are only connected to the master node via HTTP but in the future there are plans to move Moebius to SD Erlang and build a hierarchical master structure.

A top level Moebius algorithm is as follows. A user will be asked to indicate the requirements, e.g. hardware configuration, software configuration, and a description on how the initial master node should start the remaining nodes in the cloud (Figure 7.2). Thus, a hierarchical organization of nodes can be easily set from top to bottom. Additional s_groups from nodes of different levels can also be formed if needed. A scalable distributed Erlang Moebius may require the following.

1. The s_groups may be grouped on the basis of different factor such as communication locality, security, and availability of a particular hardware or software. Therefore, custom s_groups types are required.

2. Nodes and s_groups will dynamically appear and disappear depending on the user current requirements.

3. Moebius master nodes most probably will be organised in a hierarchical manner, so nodes will not need to directly communicate with each other and global s_group list information is not required.

4. The number of s_groups most probably will be much less than the number of nodes.

7.5 Riak

Riak is an open-source distributed database written in Erlang and C with a little JavaScript. Riak is greatly influenced by the CAP (i.e. Consistency, Availability and Partition) Theorem of E. Brewer [FB99] and Dynamo Storage system [HJK$^+$07]. The system is focused on Availability and Partition, and has eventual Consistency. Riak supports multiple back ends via an API that can be 'plugged in' as needed. Currently, Riak supports the following back ends: Bitcask, Dets, ETS, Erlang balanced trees (gb_trees), writing directly to the file system, and Innostore [Tec12]. Bitcask is the default Riak backend. Bitcask is a local key/value store that serves as low latency, high throughput storage back end [Tec12].

To ensure correct data replication Riak recommends to have one node per a physical server (host) in a cluster. Thus, the number of hosts coincides with the number of nodes. A Riak cluster has a 160-bit integer space that is divided into equal sizes, i.e. each node is responsible for $\frac{1}{Num\ of\ physical\ nodes}$ of a ring. The number of virtual nodes (vnodes) on a node is determined as $\frac{Num\ of\ partitions}{Num\ of\ nodes}$. Data inside Riak are organised by means of buckets and keys that are also called Riak objects [Tec12]. To disseminate bucket information the system uses a gossip protocol [JHB01]. All nodes are equal and each of them is fully capable to serve any client request [Tec12]. Riak uses consistent hashing to evenly distribute data across a cluster; this allows a node to find the nodes that store required information.

Riak controls the number of data replicas N by using 'hinted handoff' technique, i.e. neighbours of a failed node in a cluster perform the work. A client can set W value that is the number of nodes that return success message before update is considered to be complete. To detect causal ordering and conflicts all updates to an object are tracked by a vector clock mechanism [Fid88] based on Lamport timestamp algorithm [Lam84]. In Lamport timestamp algorithm when node B receives a message from node A it increases its counter, C_B, by one in case $C_B > C_A$, otherwise $C_B = C_A + 1$. However, C_i bery quickly becomes very large, that therefore [Fid88] proposed to use time stamps instead, i.e. when node B receives a message from node A it increases its timer, $T_B + \Delta t$, in case $C_B > C_A$, otherwise $C_B = C_A + \Delta t$. The system has two types of conflict resolving: last modified and human-assisted (or automated) action [Tec12].

The fastest way to fetch data out of Riak is by using data bucket and key values. Another way to retrieve data is my means of link walking. Riak does not use schema, indexes are set on an object-by-object basis. Indexing is real time and atomic, i.e. indexes are defined at the time when an object is written. All indexing occurs on the partition where the object lives; thus, objects and indexes stay synchronised [Tec12]. Riak supports horizontal scalability (i.e. adding more nodes to the system), and vertical scalability (i.e. adding resources to a single node). The horizontal scalability is supported by means of automatic recalculation of partition of each node when nodes are added and removed. Vertical scalability is supported by means of back end storage engine Bitcask [SS10].

Currently successful Riak applications utilise up to a hundred nodes, but there is a requirement from customers to increase the number of nodes beyond a hundred of nodes. From a private communication with Justin Sheehy a CTO at Basho Technologies we found that this is not feasible due to Distributed Erlang transitive connections. One possible way of scaling Riak with SD Erlang is as follows. Each node is a leader of an s_group that includes nodes from the preference list. When the leader node terminates or becomes off-line a number of options are possible to handle the issue. The group may either terminate, or temporarily coincide with the s_group of the neighbour node, or the leader responsibilities can be handled to another node of the s_group. A scalable distributed Erlang Riak may require the following.

1. The number of s_groups will coincide or exceed the number of nodes,

i.e. Riak nodes cannot maintain global s_group list information.

2. The s_groups will constantly appear, disappear, and change leader nodes.

3. Riak nodes need to be able to send messages to other nodes that they do not have direct connections. This can be handled by Tapestry like overlay routing infrastructure [ZHS+04].

4. Riak will need short lived connections to avoid failures due to constant node attempts to maintain all to all connections.

7.6 Summary

Table 7.1 provides a summary of the exemplar requirements for scalable implementations. Thus, s_groups may be static, i.e. once created nodes rarely leave and join their s_groups or dynamic, i.e. nodes and s_groups are constantly created and deleted from the network. S_groups may be formed on the basis of locality (Sim-Diasca and Mandelbrot set), Hash table (Orbit), Preference list (Riak), or programmers' and users' preferences (Moebius). Some scalable exemplars also require custom s_group types, global s_group list information, short lived connections, and semi-explicit placement. Such applications like Riak may have the number of s_groups compatible with the number of nodes.

No.	**Property**	**Sim-Diasca**	**Orbit**	**Mandelbrot set**	**Moebius**	**Riak**
				s_groups		
1	**Static/ Dynamic**	Static	Static	Static	Dynamic	Dynamic
2	**Grouping**	Locality	Hash table	Locality	Multiple	Preference list
3	**Custom types**	Y	N	N	Y	N
4	**Global s_group list information**	Y	Y	N	N	N
				General		
5	**Number of nodes and s_groups**	$N_g << N_n$	$N_g << N_n$	$N_g << N_n$	$N_g << N_n$	$N_g >= N_n$
6	**Short lived connections**	Y	Y	N	N	Y
7	**Semi-explicit placement**	Y	N	Y	N	N

Table 7.1: Exemplar Summary

Chapter 8

Implications and Future

8.1 Conclusion

This technical report presents the design of Scalable Distributed (SD) Erlang: a set of language-level changes that aims to enable Distributed Erlang to scale for server applications on commodity hardware with at most 10^5 cores. It outlines the anticipated server architectures and anticipated failures. The core elements of the design are to provide scalable in-memory data structures, scalable persistent data structures, and a scalable computation model. The scalable computation model has two main parts: scaling networks of Erlang nodes and managing process placement on large numbers of nodes. To tackle the first issue we have introduced s_groups that have transitive connections with nodes of the same s_group and non-transitive connections with nodes of other s_groups. To resolve the second issue we have introduced semi-explicit placement and `choose_node/1` function. Unlike explicit placement a programmer may spawn a process to a node from a range of nodes that satisfy predefined parameters, such as s_group, s_group type, or communication distance. Next we plan to implement the SD Erlang design.

Bibliography

[AB12] Ericsson AB. Erlang Reference Manual, (last available) 2012. http://www.erlang.org/doc/.

[ALS10] J. C. Anderson, J. Lehnardt, and N. Slater. *CouchDB: The Definitive Guide*. O'Reilly Media, 1st edition, 2010.

[Arm03] J. Armstrong. *Making Reliable Distributed Systems in the Presence of Sofware Errors*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, 2003.

[Arm10] J. Armstrong. Erlang. *Commun. ACM*, 53:68–75, 2010.

[ATL] M. Aswad, P. Trinder, and H.-W. Loidl. Architecture aware parallel programming in Glasgow parallel Haskel (GpH),.

[BC11] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, 2011.

[Col89] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[CS10] M. Christakis and K. Sagonas. Static detection of race conditions in Erlang. In Manuel Carro and Ricardo Peña, editors, *Practical Aspects of Declarative Languages*, volume 5937 of *Lecture Notes in Computer Science*, pages 119–133. Springer Berlin / Heidelberg, 2010.

[CT09] F. Cesarini and S. Thompson. *Erlang Programming: A Concurrent Approach to Software Development*. O'Reilly Media Inc., 1st edition, 2009.

[FB99] A. Fox and E.A. Brewer. Harvest, yield, and scalable tolerant systems. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 174–178. IEEE Computer Society, 1999.

[Fid88] C. J. Fidge. Timestamp in message passing systems that preserves partial ordering. In *Proceedings of the 11th Australian Computing Conference*, pages 56–66, 1988.

[For09] D. Forsberg. RESTful security. Technical report, Nokia Research Centre, Helsinki, Finland, 2009.

[Fou12] The Apache Software Foundation. Apache Cassandra, (last available) 2012. http://cassandra.apache.org/.

[Fri03] S. L. Fritchie. A study of Erlang ETS table implementations and performance. In *Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, ERLANG '03, pages 43–55, New York, NY, USA, 2003. ACM.

[HJK$^+$07] D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SIGOPS'07: Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 205–220, New York, NY, USA, 2007. ACM Press.

[HSG$^+$10] W. Huang, M.R. Stan, S. Gurumurthi, R.J. Ribando, and K. Skadron. Interaction of scaling trends in processor architecture and cooling. In *SEMI-THERM '10: Proceedings of the 26th Annual IEEE Symposium on Semiconductor Thermal Measurement and Management*, pages 198–204. IEEE Computer Society, 2010.

[JHB01] K. Jenkins, K. Hopkinson, and K. Birman. A gossip protocol for subgroup multicast. In *International Conference on Distributed Computing Systems*, pages 25–30. IEEE Computer Society, April 2001.

[Lam84] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst.*, 6:254–280, April 1984.

[Len09] J. Lennon. Exploring CouchDB, 2009. http://www.ibm.com/developerworks/opensource/library/os-couchdb/index.html.

[MJN03] S. Murthy, Y. Joshi, and W. Nakayama. Orientation independent two-phase heat spreaders for space constrained applications. *Microelectronics Journal*, 34(12):1187–1193, 2003.

[QSR09] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. *SIGARCH Comput. Archit. News*, 37:24–33, June 2009.

[SABR05] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. Lifetime reliability: Toward an architectural solution. *IEEE Micro*, 25:70–80, May 2005.

[SG07] B. Schroeder and G. A. Gibson. Disk failures in the real world: What soes an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, Berkeley, CA, USA, 2007. USENIX Association.

[SS10] J. Sheehy and D. Smith. A log-structured hash table for fast key/value data. Technical report, Basho Technologies, Inc., April 2010.

[SSA$^+$06] G.J. Snyder, M. Soto, R. Alley, D. Koester, and B. Conner. Hot spot cooling using embedded thermoelectric coolers. In *Proceedings of the Twenty-Second Annual IEEE Semiconductor Thermal Measurement and Management Symposium*, pages 135–143. IEEE Computer Society, March 2006.

[Tec12] Basho Technologies. Basho documentation, (last available) 2012. http://wiki.basho.com.

[War11] J. Warnock. Circuit design challenges at the 14nm technology node. In *DAC '11: Proceedings of the 48th Design Automation Conference*, pages 464–467, New York, NY, USA, 2011. ACM.

[WBPB11] B. Wojciechowski, K.S. Berezowski, P. Patronik, and J. Biernat. Fast and accurate thermal simulation and modelling of workloads of many-core processors. In *THERMINIC '11: Proceedings of the 17th International Workshop on Thermal Investigations of ICs and Systems*, pages 1–6. IEEE Computer Society, 2011.

[WZJ$^+$04] E.N. Wang, L. Zhang, L. Jiang, J. Koo, J.G. Maveety, E.A. Sanchez, K.E. Goodson, and T.W. Kenny. Micromachined jets for liquid impingement cooling of VLSI chips. *Microelectromechanical Systems, Journal of*, 13(5):833–842, 2004.

[ZHS$^+$04] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *Selected Areas in Communications, IEEE Journal on*, 22(1):41–53, 2004.