THE SCALABILITY OF RELIABLE COMPUTATION IN ERLANG

AMIR GHAFFARI

SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor of Philosophy

SCHOOL OF COMPUTING SCIENCE

COLLEGE OF SCIENCE AND ENGINEERING
UNIVERSITY OF GLASGOW

OCTOBER 2015

Abstract

With the advent of many-core architectures, scalability is a key property for programming languages. Actor-based frameworks like Erlang are fundamentally scalable, but in practice they have some scalability limitations.

The RELEASE project aims to scale the Erlang's radical concurrency-oriented programming paradigm to build reliable general-purpose software, such as server-based systems, on emergent commodity architectures with 10^4 cores. The RELEASE consortium works to scale Erlang at the virtual machine, language level, infrastructure levels, and to supply profiling and refactoring tools.

This research contributes to the RELEASE project at the language level. Firstly, we study the provision of scalable persistent storage options for Erlang. We articulate the requirements for scalable and available persistent storage, and evaluate four popular Erlang DBMSs against these requirements. We investigate the scalability limits of the Riak NoSQL DBMS using Basho Bench up to 100 nodes on the Kalkyl cluster and establish for the first time scientifically the scalability limit of Riak as 60 nodes, thereby confirming developer folklore.

We design and implement *DE-Bench*, a scalable fault-tolerant peer-to-peer benchmarking tool that measures the throughput and latency of distributed Erlang commands on a cluster of Erlang nodes. We employ DE-Bench to investigate the scalability limits of distributed Erlang on up to 150 nodes and 1200 cores. Our results demonstrate that the frequency of global commands limits the scalability of distributed Erlang. We also show that distributed Erlang scales linearly up to 150 nodes and 1200 cores with relatively heavy data and computation loads when no global commands are used.

As part of the RELEASE project, the Glasgow University team has developed Scalable Distributed Erlang (SD Erlang) to address the scalability limits of distributed Erlang. We evaluate SD Erlang by designing and implementing the first ever demonstrators for SD Erlang, i.e. DE-Bench, *Orbit* and *Ant Colony Optimisation*(ACO). We employ DE-Bench to evaluate the performance and scalability of group operations in SD-Erlang up to 100 nodes. Our results show that the alternatives SD-Erlang offers for global commands (i.e. group commands) scale linearly up to 100 nodes. We also develop and evaluate an SD-Erlang implementation of *Orbit*, a symbolic computing kernel and a generalization of a transitive closure computation. Our evaluation results show that SD Erlang Orbit outperforms the distributed Erlang Orbit on 160 nodes and 1280 cores. Moreover, we develop a reliable distributed version of ACO and show that the reliability of ACO limits its scalability in traditional distributed Erlang. We use SD-Erlang to improve the scalability of the reliable ACO by eliminating global commands and avoiding full mesh connectivity between nodes. We show that SD Erlang reduces the network traffic between nodes in an Erlang cluster effectively.

Acknowledgements

First, I would like to express my deepest gratitude to my supervisors, Professor Phil Trinder and Professor Sven-Bodo Scholz, for their continuous support, wise guidance and insightful comments.

In particular, I am grateful to Phil for including me in the RELEASE project and providing me this great opportunity to carry out my research in a wonderful research group with a combination of academic and industrial research viewpoints. I thank my RELEASE project colleagues for their constructive comments.

I must thank the European Union Seventh Framework Programme (FP7/2007-2013) for funding my research by grant IST-2011-287510 RELEASE.

Many of the research computations were performed on resources provided by SNIC through Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX) under Project p2012172.

I sincerely thank my parents for tremendous support and encouragement. Words cannot express my eternal gratitude for everything you have done.

Above all, I would like to express special thanks to my wife Shiva for her love, constant support, and accompanying me on this adventure. Most loving thanks to my daughter, Diana, who was born during my PhD and brought love, hope, and inspiration to me.

Declaration

I declare that the work, except where explicit reference is made to the contribution of others, is entirely my own work and it has not been previously submitted for any other degree or qualification in any university.

Amir Ghaffari

Glasgow, October 2015

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Context

The trend toward scalable architectures such as clusters, grids, and clouds will continue because they combine scalability and affordability. These scalable infrastructures typically consist of loosely-connected commodity servers in which node and network failures are common. To exploit such architectures, we need reliable scalable programming paradigms.

Recently Erlang has become a popular platform to develop large-scale distributed applications, e.g. *Facebook chat backend*, *T-Mobile advanced call control services*, *Whatsapp messenger for smartphones*, and *Riak DBMS* [3]. This popularity is due to a combination of factors, including data immutability, share-nothing concurrency, asynchronous message passing based on the actor model, location transparency, and fault tolerance.

However, in practice the scalability of Erlang is constrained by aspects of the language and virtual machine [4]. For instance, our measurement shows that Riak 1.1.1 does not scale beyond 60 nodes because of a single overloaded supervisor process [5].

This research is conducted as part of the RELEASE project that aims to scale the Erlang's radical concurrency-oriented programming paradigm to build reliable general-purpose software, such as server-based systems, on emergent commodity architectures with 10^4 cores [4]. The RELEASE consortium works to scale Erlang at the virtual machine, language level, infrastructure levels, and to supply profiling and refactoring tools.

This thesis contributes to the RELEASE Project at the language level with focus on persistent storage for Erlang and the scalability of distributed Erlang. It investigates the provision of scalable persistent storage for Erlang and introduces a scalable parameterized benchmarking tool for distributed Erlang. It also evaluates the new Scalable Distributed Erlang (SD Erlang) by developing scalable reliable benchmark applications.

1.2 Contributions

This thesis is undertaken in the context of the RELEASE project and makes the following research contributions:

Scalable Persistent Storage for Erlang: Theory and Practice. We enumerate the requirements for scalable and available persistent storage, and evaluate four popular Erlang DBMSs against these requirements. This analysis shows that Mnesia and CouchDB do not provide suitable persistent storage at our target scale, but Dynamo-style NoSQL DBMSs such as Cassandra and Riak potentially do.

We investigate the scalability limits of the Riak NoSQL DBMS in practice up to 100 nodes. We establish for the first time scientifically the scalability limit of Riak as 60 nodes on the Kalkyl cluster. We show that resources like memory, disk, and network do not limit the scalability of Riak. By instrumenting Erlang/OTP and Riak libraries we identify a specific Riak functionality that limits scalability. We outline how later releases of Riak are refactored to eliminate the scalability bottlenecks. We conclude that Dynamo-style NoSQL DBMSs provide scalable and available persistent storage for Erlang in general, and for our RELEASE target architecture in particular [5, 6, 7, 8] (Chapter 3).

Investigating the scalability Limits of Distributed Erlang. We have designed and implemented *DE-Bench*, a scalable fault-tolerant peer-to-peer benchmarking tool for distributed Erlang. We employ DE-Bench to investigate the scalability limits of distributed Erlang up to 150 nodes and 1200 cores. Our benchmarking results demonstrate that the frequency of global commands limits the scalability of distributed Erlang. We show that distributed Erlang scales linearly up to 150 nodes with relatively heavy data and computation loads when no global commands are used. Measuring the latency of commonly-used distributed Erlang commands reveals that the latency of *rpc* calls rises as cluster size grows. Our results also show that server processes like *gen_server* and *gen_fsm* have low latency and good scalability [8, 9, 10, 2] (Chapter 4).

Evaluating the new Scalable Distributed Erlang (SD Erlang). Due to the default connectivity model of distributed Erlang, most of the distributed Erlang applications have mesh connectivity where all nodes are fully connected to each other. We employ SD-Erlang to develop the first ever SD Erlang demonstrators, i.e. DE-Bench, *Orbit* and *Ant Colony Optimisation* (ACO). We employ DE-Bench to evaluate group commands in SD Erlang and compare them with global commands in traditional distributed Erlang up to 100 nodes. We show that group commands scale linearly up to 100 nodes, whereas global commands fail to scale beyond 40 nodes with the same frequency.

We develop a new distributed version of Orbit (*SD Orbit*) to reduce the connections between

nodes. In SD Orbit, Erlang nodes are divided into a number of groups in which nodes belonging to the same group communicate directly whereas communications between nodes from different groups are routed through gateways. We demonstrate that this approach enhances the performance and scalability of SD Orbit on 160 nodes and 1280 cores.

We also develop a reliable scalable version of Ant Colony Optimisation (ACO), a heuristic search technique that aims to arrange the sequence of jobs in such a way as to minimise the total weighted tardiness. To evaluate the reliability of ACO, we employ Chaos Monkey [11]. We show how reliability limits the scalability of ACO in traditional distributed Erlang. We alleviate the cost of reliability by employing techniques that SD Erlang provides to reduce the network connectivity between Erlang nodes and control locality by using group commands instead of global commands. Investigating the network performance shows that SD Erlang reduces the network traffic in a cluster of Erlang nodes efficiently [8, 10, 2](Chapter 5).

1.3 Authorship

Much of the works is joint with the RELEASE team, and my primary contribution to the publications is as follow:

I am the lead author of *Scalable Persistent Storage for Erlang: Theory and Practice* [5, 7], and the sole author of *Investigating the scalability Limits of Distributed Erlang* [9]. I am also the co-author of *Improving Network Scalability of Erlang* [8].

My primary contributions to deliverables D3.1 (*Scalable Reliable SD Erlang Design*) are sections 7 and 8 [6], and to D3.2 (*Scalable SD Erlang Computation Model*) is section 5 [10], and to D3.4 (*Scalable Reliable OTP Library Release*) are sections 3 and 4 [2].

1.4 Organization

Chapter 2 discusses relevant background material and related approaches. We begin by describing the existing and upcoming trends in parallel and distributed hardware architectures. We also introduce the RELEASE target platform. We next discuss parallel programming models and their scalability and suitability for the target architectures. The main challenges in distributed systems are also discussed. Finally, we explore parallel and distributed programming in Erlang.

Chapter 3 defines general principles for scalable persistent stores. It evaluates the scalability of four popular Erlang DBMSs against these principles. The scalability, availability, and elasticity of the Riak NoSQL database are investigated up to 100 nodes practically.

Chapter 4 investigates the scalability limits of distributed Erlang using DE-Bench. It presents the design and implementation of DE-Bench, a scalable fault-tolerant peer-to-peer benchmarking tool for distributed Erlang. Our measurement results are presented up to 150 nodes and 1200 cores.

Chapter 5 evaluates the scalability of improved Distributed Erlang (SD Erlang) by developing the first ever benchmarking applications for SD Erlang, i.e. DE-Bench, *Orbit* and *Ant Colony Optimisation*(ACO). A comparison of distributed Erlang and SD Erlang up 160 nodes and 1280 cores is presented.

Chapter 6 provides a summary, and discusses limitations and future work.

Chapter 2

Literature Review

This chapter first studies different parallel and distributed computing architectures. We describe our target platform based on existing and upcoming trends in hardware architectures (Section 2.1).

We study a variety of parallel programming models and investigate their scalability and suitability for the target architectures in Section 2.2. We explore the distributed computation models and their main challenges in Section 2.3. Section 2.4 focuses on parallel and distributed programming in Erlang.

2.1 Parallel and Distributed Computing Architectures

Parallel and distributed programming models strongly depend on the architecture of the execution platform. Thus, considering the general structure of existing and upcoming hardware platforms can give us a better view of future trends and techniques in parallel and distributed software development. This section studies the scalability of different hardware architectures and their suitability for reliable scalable software. There are various classifications of parallel architectures, e.g. classification based on memory architecture, interconnections between processes, and Flynn's classification based on instructions and data streams [12]. As discussing all the classifications is beyond the scope of this research, we focus on memory architectures in parallel computers. Parallel architectures are categorized into three major categories in terms of memory arrangement and communication: *shared memory*, *distributed memory*, and *hybrid* architectures [13].

2.1.1 Shared Memory Architectures

In these architectures, memory is accessed by a number of processors via an interconnection network, and changes in a memory location are visible to all processors [14]. In this approach, communication between tasks on different processors is performed through reading from and writing to the shared memory. When multiple processors are trying to access the shared memory simultaneously, synchronization between threads is needed to avoid race conditions that can cause unpredictable behaviors in the system. This is an efficient way of communicating between small amount of processes as it is fast and there is no need for data partitioning and replication. However, for a larger number of processes (a few dozen processors), performance degradation might happen due to access contention to the share memory and the central bus which provides access to the shared memory. Caching techniques can be used to alleviate the contention problem by providing a faster access to the local cache in comparison with the global memory. However, having multiple copies of data might lead to a coherence problem. There are cache coherence protocols to maintain the consistency among the copies [14]. Multi-core processors are an example for shared memory systems in which a common address space is accessed by all cores [13].

Shared memory systems are categorised into two major groups based on their interconnection network: Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA) [14].

In UMA memory locations are accessible by all processors in the same access time [13]. As shown in Figure 2.1, each processor accesses the shared memory through a shared bus. As the number of processes increases, access demand for the common bus grows, and thus the shared bus might become overloaded and a bottleneck for the scalability. The maximum number of processors that can be used in UMA model lies between 32 and 64 [13]. Symmetric Multiprocessors (SMPs) are one of the variant of UMA which are commonly used in new desktop and laptop machines such as Core Duo, Core 2 Duo, and Opteron.

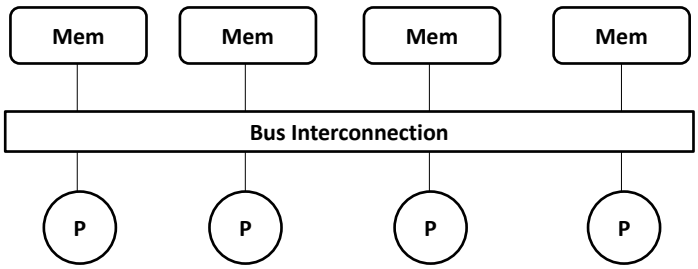


Figure 2.1: Uniform Memory Access

To overcome the scalability problem of UMA, NUMA is an alternative approach that can be used in larger computer systems [13]. In the NUMA model, processors have different access-time to the shared memory depends on their distance to the memory location (Figure 2.2).

The access time of a processor to its own local memory is faster than an access to a data value in the memory of another processor. Firstly, a process looks for data in its local memory address before seeking the data in a remote memory located on the other processors. This approach improves the scalability and performance as all data accesses don't have to go through the interconnection bus, and thus leads to reduction of contention on the bus.

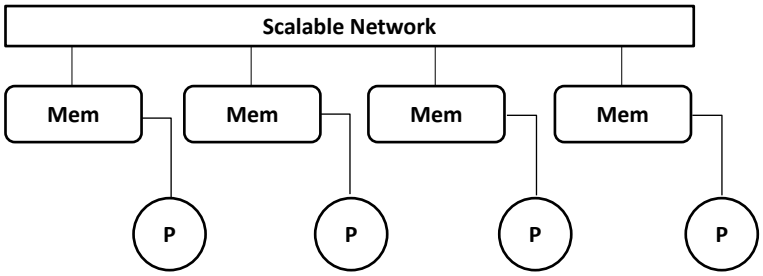


Figure 2.2: Non-Uniform Memory Access

2.1.2 Distributed Memory Architectures

In distributed memory architectures, processors are connected through bus networks [15]. Each processor has its own memory space and there is no global address space in the system. Message passing is used to access a remote memory space. This approach scales better than shared memory because it eliminates the need for synchronization and maintaining global cache coherency. However, it provides non-uniform memory access times because access to data residing on a remote memory takes longer than access to a local memory.

2.1.3 Hybrid Systems

Two decades ago, many people thought that computer performance could be improved by creating faster and more efficient processors. However, in the last decade this trend has moved away from expensive supercomputers to networks of computers. The idea of utilizing the computational capability that is available in a network has gained large interest from the high-performance computing community [16].

Hybrid systems embrace both shared and distributed memory architectures. Clusters of multi-core processors is the most common form of hybrid architecture with a hierarchical interconnection network. This trend imposes a large impact on software development to benefit from the performance increase provided by such hybrid architectures. The most widely used examples of hybrid systems are Clusters, Grids and Clouds.

2.1.3.1 Clusters

A cluster is generally defined as a collection of inter-connected and loosely coupled stand-alone computers (nodes) that work collectively as an integrated computing resource [16]. This technique is used to achieve more computing power by connecting a set of independent computers though high-speed networks, such as gigabit Ethernet, SCI, Myrinet and Infiniband [17]. An individual computer in a cluster can be a single or multiprocessor system but a cluster appears as a single unit to users. The emergence of multi-core architectures have had an important impact on conventional clusters. Multi-core processors have already been widely deployed in clusters which are called *multi-core clusters* [18].

Clusters are an attractive alternative to traditional supercomputers due to their scalability and affordability. They also offer high availability as they continue to provide service in case of node and network failures. If a node fails, the other nodes can take over the failed node roles. The complexities of the underlying system can be hidden from the user through middlewares to create an illusion of a single system instead of all the architecture [19].

Scalability in cluster systems is not only limited to interconnecting the nodes within a cluster. Another aspect of scalability can be achieve through connecting small clusters together to make larger ones with a hierarchical interconnection network. Moreover, heterogeneity is another key feature of clusters because they have potential to accommodate heterogeneous computing nodes and communication links.

2.1.3.2 Grids

The ubiquity of the Internet with the emergence of multi-core commodity and high-speed networks make it possible to share and aggregate geographically distributed heterogeneous computing resources from different organizations and individuals for solving large-scale problems.

A Grid is a large scale, geographically distributed computing system across multiple organizations and administrative domains [20]. Resource management in the Grid environment is a challenging task because usually a Grid is owned by different organizations with different access and cost policies [21]. Resource management also should address issues like heterogeneity, resource allocation and re-allocation, transparent scalability, security, and fault-tolerance and availability. Middlewares and toolkits like Globus, Legion, and Condor have been developed to address some of these issues by providing a software infrastructure to reduce the complexity of the overall system and enables users and applications to view heterogeneous distributed resources as a single system [22, 23, 24].

2.1.3.3 Clouds

The idea of delivering computing service in a way that traditional utilities such as water, electricity, gas, and telephony are delivered is becoming more and more popular [25]. In this model, users consume the provided service based on their requirements without needing to know where and how the service are provided. Consumers of these services can access them whenever required, and then pay the service providers accordingly.

Cloud is a parallel and distributed architecture that consist of a collection of inter-connected computing resources such as computer servers, storage, applications, and services that are dynamically provisioned and released [25]. The term Cloud denotes the ability of accessing the infrastructures and applications from all over the world on demand. This makes it possible to develop software as a service for millions of consumers, rather than locally run on individual computers.

Reliability in Cloud systems is provided by coordinating many machines in a distributed environment and supporting machine fails by provision of another machine to recover it. Another feature that we require for large-scale architectures is auto-scaling that allows us to scale the capacity up or down automatically according to the run-time conditions. In comparison with cluster, Cloud delivers various types of resources mainly over the Internet. Moreover, the level of heterogeneity in Cloud systems is much more extensive in terms of networks infrastructure, computer hardware and the development teams involved.

Cloud computing uses virtualization to separate a single physical machine into one or more virtual machines. Virtualization reduces the IT cost because parts of a server or service can also be leased to other users. Moreover, virtualization improves availability and flexibility as virtual machines are portable and can be moved from one physical server to another one in the same data center, or even in another data center without experiencing downtime.

2.1.3.4 Comparison of Cluster, Grid, and Cloud Computing

This section distinguishes cluster, Grid and Cloud computing systems by comparing their features and properties. Clusters are usually located in single geographic location with a single administrative domain whereas, Grid systems are geographically distributed across multiple administrative domains and locations. Resource management in the Grid systems is more complex than clusters because the Grid is owned by different organizations with different policies, and so middlewares in the Grid environment should consider more sophisticated solutions to address challenges like heterogeneity, resource allocation, scalability, security, and availability [17].

In addition to the characteristics of both clusters and Grids, Cloud has its own special attributes and capabilities such as elasticity, virtualization, reliability and fault tolerance in

case of resource failures, autonomic energy aware resource management to reduce operational costs for providers [25].

2.1.3.5 the RELEASE Target Architecture

The following description is closely based on RELEASE deliverable D3.1 [6].

It is challenging to predict computer architecture trends. However, to address the scalability challenges in a language we must predict typical commodity hardware in the next 5 years.

As we discussed in previous sections, due to the scalability limits of UMA, the NUMA model is increasingly popular in high performance computing. Moreover, considerable advances in network technologies and commodity-off-the-shelf (COTS) hardware are making clusters a low cost solution for high performance supercomputing environment. Thus, the target architecture for the RELEASE project is a mixture of these technologies. In our typical server architecture, as show in Figure 2.3, each host contains ∼4–6 core modules, where each module has ∼32–64 cores. Around a hundred hosts will be grouped in each cluster, and a cloud contains ∼1–5 clusters.

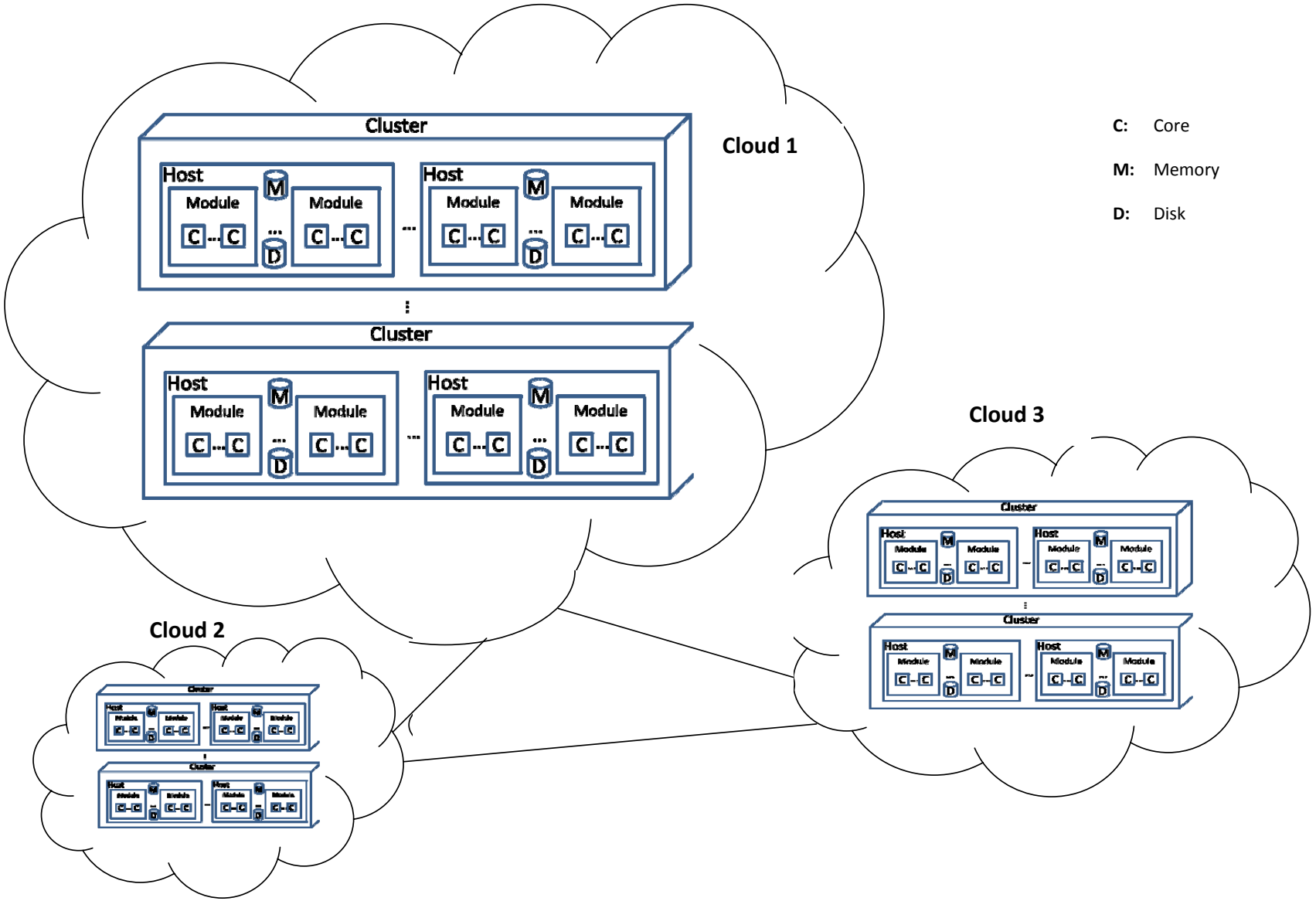


Figure 2.3: RELEASE's Target Architecture

2.2 Parallel Programming Models

After taking a brief look at the hardware architectures, we focus on software aspects including the programming languages and runtime libraries. The complexity of developing parallel applications is due to some factors [26]. Developing a parallel application is dependent on adequate software tools and environments. Integrated tool environments such as debuggers, performance monitors and analyzers for parallel application development have not advanced to the same degree as sequential programming. Moreover, there are some challenges specific to parallel programming such as non-determinism, communication, synchronization, data partitioning, load-balancing, deadlocks, and race conditions.

Parallel programming models provide an abstraction over underlying computing architectures by separating the model from its actual implementation and any specific machine type. They increase programmability and portability of parallel software by providing certain operations to the programming level, and hiding the implementation for each of these operations.

This section explores some of the most common programming models that are being used in parallel computing.

2.2.1 Shared Memory Programming Models

In shared-memory models, communication is usually done through shared variables implicitly. Non-determinism is a common phenomenon in multi-threading environments which is not always a problem and sometimes it is desirable. A parallel system is non-deterministic if it produces different results for identical input when a group of threads do a computation collaboratively in parallel. Race conditions are the most common form of non-determinism especially in shared-memory programs that might be disastrous and result in program errors. Race condition occurs when several threads have asynchronous access to a shared memory and try to change it. In this condition the result is nondeterministic because the order of access to the share data is variable depending on the thread scheduler [27].

To solve the race condition problem only one thread in each moment should be able to access the shared data. The restricted code that should be mutually exclusive accessible is called critical section. The most commonly used mechanisms for insuring mutual exclusion are lock-based methods such as semaphore or monitor.

The advantages of the shared memory model are that the programming is simple because we can write and read to an address pointer easily which is compatible with standard programming languages and systems. Moreover, in small scale architectures, communication between processors is very fast because memory access and inter-processor communication

has low latency. Shared memory also offers a unified address space in which all data can be accessed directly.

However, there are some limitations in using shared memory models. Shared-memory architectures are not designed to scale beyond a few tens of processors because of limitation in the memory bandwidth. As the number of parallel processes increases, memory bandwidth which is shared between the cores will be a bottleneck for scalability. Moreover, in high scale systems when thousands of threads try to get an exclusive lock on a shared memory, memory contention becomes a significant bottleneck. Contention for locks causes parallel threads to execute serially even though there are several threads ready to run and a sufficient number of cores available [27].

2.2.1.1 OpenMP

It is an important model and language extension for shared-memory parallel programming. OpenMP is an industrial standard API developed through the join of top computer manufacturers and software developers such as IBM, Oracle and Intel [28]. OpenMP is supported on the most widely used programming languages such as C and FORTRAN. It is portable across different hardware platforms because any compiler that supports OpenMP can compile parallel applications written in OpenMP. OpenMP programs can be easily converted from sequential into a parallel program because it is a directive-based parallel language.

However, there are some limitations that prevent OpenMP from being used in large-scale systems [28]. Since OpenMP is designed for using in shared-memory systems, it does not function efficiently on distributed-memory machines and its parallel scalability is limited to the number of cores on a single machine. Also there is no explicit control over thread creation, termination, and their priority. It also inherited shared memory problems such as race condition and deadlock, and so developers need to make their code thread-safe.

2.2.2 Distributed Memory Programming Models

In contrast to the share memory approach in which a single memory space used by all processes, in this model each process has its own private memory. Moreover, remote data and services are accessible by communicating with other remote processes across a network such as message-passing and remote invocation. This model offers scalability because no resource is shared among different processes and consequently there is no mutual exclusion. This offers a good solution for parallel application on distributed architectures like computer clusters in which scalability is a key requirement [13].

On the other hand, there are some drawbacks in this model. Distributed memory programming model is more difficult than non-distributed models because the programmer must

explicitly control the data distribution. Moreover, all parallelism is explicit that means the developer is responsible for correctly identifying and implementing parallelism. Communication between processes at small-scale with a few number of processes also can be more expensive than the shared memory model due to the larger latency of message passing in comparison with memory access.

2.2.2.1 Message Passing Models

Message-passing models are the dominant programming models for scalable programs in parallel and distributed computation. Processes have their own local memory (shared nothing) and communicate by sending and receiving messages. The message passing models fit well on distributed memory architecture such as cluster of individual workstations which are connected by a network. The two most popular message passing programming interfaces are Parallel Virtual Machine (PVM) and Message Passing Interface (MPI).

Parallel Virtual Machine (PVM) uses the message-passing model to exploit a collection of distributed heterogeneous computers as a single virtual machine [29]. All message routing, data conversion, and task scheduling across a network are done transparently in PVM. The unit of parallelism in PVM is a task and an application in PVM is a collection of tasks that communicate across a network explicitly by sending and receiving data structures. Tasks are identified by an integer task identifier (TID) which is unique across the entire virtual machine. In addition to basic message passing, a variety of global operations, including broadcast, global sum, and barrier synchronization are also supported in PVM. The PVM system currently supports C, C++, and Fortran languages. PVM is a relatively low-level message-passing system and programmer has to explicitly implement the details of parallelization. PVM is not also a standard and has been supplanted by MPI standard for message passing, although it is still often used [29].

Message Passing Interface (MPI) is a vendor independent, efficient, flexible industry-standard API specification designed for writing message-passing parallel programs on distributed memory architectures. There are numerous implementations of the MPI standards such as MPICH, MVAPICH, Open MPI, GridMPI, and many others [30]. MPICH library is one of the first implementations of MPI-1 (MPI-1 is the first MPI standard). MPI-2 standard that was released in 1996 defined advanced features such as dynamic process management, remote memory access operations and parallel I/O. MPI is the dominant programming model for highly scalable programs in computational science. It offers an efficient communication in a heterogeneous environment by providing virtual communication channels that are able

to communicate across different architectures [13]. MPI has eased the burden for the developer by reusing the code and providing an environment for sending and receiving packets of data as a message between processes. The most basic form of communication in MPI is point-to-point. In this kind of communication two processes participate, one executes a send operation and the another one executes a corresponding receive operation. MPI also supports collective communication operations in which a subset of the processes of a parallel program are involved. Examples of collective operations are broadcast, reduction, gather, scatter, multi-broadcast, multi-accumulation.

However, there are potential pitfalls of using MPI. Message-passing systems are low-level and most of parallelizing tasks such as communication, synchronization and data fragmentation between processes should be handled by application developers. The lack of support for active messages is another drawback in MPI. An active message is a lightweight messaging protocol for reducing communications latency by providing a higher-level communication abstraction [31]. Moreover, MPI does not support a mechanism for fault tolerance. An MPI program could stop working whenever a single computing node fails. A mechanism is demanded that provides a transparent fault-tolerant environment on clusters by avoiding single point of failure [32]. Also, low-level approach such as MPI does not support garbage collection, and so the programmer has to cope with freeing allocated space. Although in small-scale systems the manual memory management may show better performance in comparison with automatic techniques (such as garbage collection), but in large-scale systems, explicit manual memory management can be a major source of errors and makes code more complicated.

2.2.2.2 Actor Model

The traditional way of concurrency is using threads which leads to a series of hard to debug problems such as race conditions, deadlock and starvation. The actor model is a mathematical model which takes a different approach for concurrency to avoid the problems caused by threading and locking mechanisms [33]. The actor model is inherently concurrent because each actor is an autonomous entity that can operate concurrently and asynchronously. Actor can take three types of actions: sending or receiving messages to/from other actors, creating new actors and update its local state for designating the future behaviour. Actors have no mutable shared state and consequently avoid mutual exclusion and related problems which are common in shared memory approaches. In the actor model, the only way that an actor can affect on another actor's state is through sending a message. Each actor has a mailbox and messages that exchange between actors can be buffered in the mailbox. All communications are asynchronous which means sender does not wait for a message to be received and can immediately continue its execution. Programming languages based on actor model offer

higher level models for distributing the work load and communication between processes in compare with the low level approaches such as MPI. Several languages implement the actor model that the most common ones are Erlang and Scala [34, 35].

Scala stands for Scalable Language, is a general-purpose programming language, which effectively integrates functional and object-oriented concepts [35]. Scala compiles down to Java byte-code and can be run efficiently on the Java Virtual Machine (JVM). Scala programs can work perfectly with Java libraries and Scala code can be integrated into Java projects to take advantage of the power of Scala without losing the infrastructure code in Java. Moreover, Scala's pattern matching provides a powerful and flexible declarative syntax for expressing complex business logic [35]. Scala's immutable data structures are important for designing stable applications in a concurrent programming paradigm. Immutable data avoids the common failures which are encountered in mutable state in parallel programs such as unpredictable behaviour due to race condition. The actor model in Scala offers a different mechanism for dealing with concurrency and multi-threading. Actors in Scala are implemented as a library, and there is no support at the language or compiler level.

But there are some downsides in Scala. Scala supports two kinds of actors: thread-based and event-based [36]. Each thread-based actor is a JVM thread and hence relatively expensive to create, run and maintain. Threads in JVM use a considerable amount of memory that may lead to memory exhaustion with large amounts of actors. Moreover, thread switching is a costly operation. This means that for running millions of actors we should avoid thread-based approach, and so the only remaining option is event-based actors. In an event-based approach, there is not a single thread per actor and an actor is captured as a closure that is executed by a thread [36]. When an actor needs to wait for a receiving message(*react*) the container thread is free for a different actor. Thus, event-based actors are more light-weight and are suitable for very large numbers of concurrent actors. There is a pitfall in this approach, however. If all actors in a program are event-based, then all of them are executed on the same thread and thus, we lose parallelism. So adjustment is needed to gain the best result and it has an extra burden on developers. Moreover, due to the mix of object-oriented programming with actor model, the internal state of an actor object might be modified through its public methods. This can cause many regression bugs like unintended state changes in one part of a concurrent program due to intended changes in another part.

Erlang ("**ER**icsson **LANG**uage", or alternatively, a reference to Danish mathematician and engineer Agner Krarup Erlang) is a strict parallel functional language and the most commercial functional language that was designed by Ericsson for developing distributed, real-time, fault tolerant telecommunication systems. Erlang uses the actor model, and each

actor is a separate process. Actors (processes) share no information with each other and exchange of data is done only through message passing [34]. Erlang processes are lightweight which means the creation and destruction of processes in the Erlang runtime system are very fast with a small memory footprint. Processes in Erlang are not operating system threads and they are managed by an internal scheduler in the runtime system, and hence switching between processes is very fast . This feature makes it possible that a typical Erlang system can switch between hundreds of thousands processes on a modern desktop computer. Erlang offers a good mechanism for developing fault-tolerant systems. Erlang processes can be linked together to build hierarchical structures where some processes are supervising other processes. When a process terminates, an exit signal is emitted to all linked processes. When a supervisor gets an error, it can make a decision, for example, the supervisor can restart the process or ignore it. Due to the reputation of Erlang in building fault-tolerant systems and its highly scalable computational model, the RELEASE project, and hence this research focus on Erlang/OTP to build a high-level paradigm for a reliable general-purpose software [4]. Section(2.4) studies the scalability and reliability of Erlang more in details.

2.3 Distributed Computing

The terms ”parallel computing” and ”distributed computing” have considerable overlap in goals, characteristics and challenges. Indeed, the emergence of new hardware technologies in high performance computing (HPC) indicates that computing systems become more hierarchical and heterogeneous. Today, commodity off-the-shelf processors can be connected to build powerful parallel machines to achieve a very high performance. The HPC market-place employs advanced technologies in computer networks to achieving higher performance on large systems of multi-core processors. For example, a cluster of multiprocessor consists of a set of multi-core processors which are connected to each other through fast network connections. Distributed computing plays an essential role to increase performance benefits from these computing infrastructures. In this section, we study distributed programming techniques and concepts and discuss their suitability and usability for large-scale computing architectures as we target in the RELEASE project.

2.3.1 Distributed Computing Challenges

Since the target architecture (section 2.1.3.5) could be distributed within some geographical areas, we need to explore more the distributed systems challenges which need to be overcome. This section explores the challenges such as heterogeneity, scalability, transparency that we should expect to face during the design and implementation phases of scalable software, and then we discuss how they should be addressed in a general form.

- Heterogeneity is an inevitable consequence of the trend towards integration of heterogeneous computing resources to achieve an inexpensive, reliable, powerful computational capabilities. Heterogeneity can be related to several factors such as different sorts of network, computer hardware, operating systems or programming languages [37]. If a system design is adaptable to characteristics of a heterogeneous environment, heterogeneity can be advantageous to improve the scalability and reliability of a distributed system. The flexibility of heterogeneous distributed systems can also be desirable for users because they can get service from different computer platforms, network infrastructures, and operating systems.

  The challenge is finding techniques to handle and exploit various levels of heterogeneity in distributed systems. This is possible through a high level abstraction to mask the heterogeneity of the underlying resources such as platforms, operating systems, and programming languages. Examples for these kinds of abstraction layers are Common Object Request Broker (CORBA) and Java Remote Method Invocation (RMI) for programming languages, virtual machines for operating systems, and Internet protocols for networks [37]. Furthermore, middlewares are a layer of software that mask the heterogeneity of the underlying infrastructure to provide a convenient programming model for the development of distributed systems [37].

- Scalability is one of the key purposes of the distributed systems and the ability of a system to exploit available resources to continuously evolve in order to support a growing amount of load [38]. There are two different ways of scaling a distributed system, i.e. *vertical* and *horizontal* scalability. Vertical scalability obtained by adding more resources to a single node in a system. For example increasing computing power of a node by adding more CPU cores or memory. Horizontal scalability achieved by adding more nodes to a system. These systems generally consist of a set of loosely interconnected nodes; each node running its own instance of operating system. Recently, the most successful Web-based enterprises (e.g., Google, Amazon and Yahoo) show their interest in clustering and horizontal scalability because this approach offers more scalability and availability.

  There are two common ways to measure the scalability of a system, i.e. *strong* and *weak* scalability. The strong scalability analyses how the speedup (work units completed per unit time) grows as the number of resources increases in a system (Figures 2.4). Ideally, the speedup grows linearly, and this shows that the system does not suffer from synchronisation overhead between distributed nodes. The weak scalability analyses how the solution time varies when the number of resources increases for a fixed problem size per resource. Ideally, the time remains constant which indicates the ability of the system to perfectly exploit the available resources to solve the problem

(Figure 2.5).

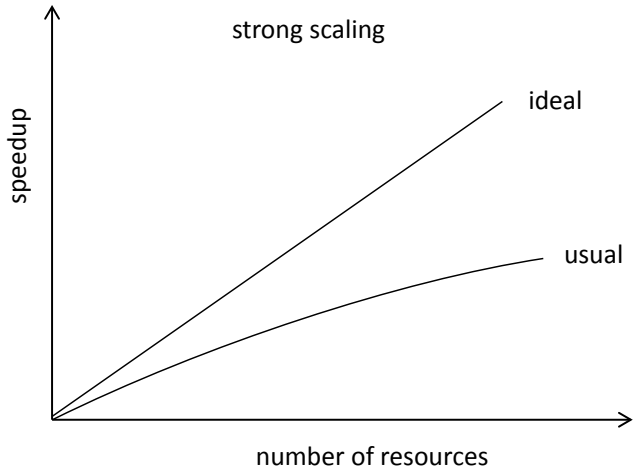


Figure 2.4: Strong Scaling

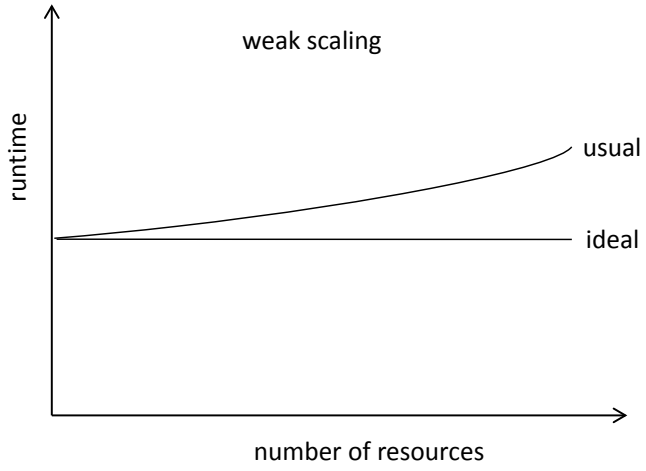


Figure 2.5: Weak Scaling

A proper load balancing mechanism avoids a situation in which a single resource becomes overloaded and consequently a bottleneck for scalability. In decentralized architectures such as peer-to-peer (P2P) models, loads are evenly spread on all the participating nodes that leads to a better scalability.

Moreover, there are some other factors such as consistency and availability that affect the scalability in a system, e.g. the CAP Theorem states, high scalability, high availability, and strong consistency cannot be achieved all together at same time in a distributed system [39].

- Failure Handling
  A distributed system consists of a number of components that are connected together

by network links. Distributed systems are more prone to failures as a fault may occur in any of the components in the system or in the network that connects them. On the other hand, failures in distributed systems are partial that means the system is able to continue to work even in the presence of failure [37]. This feature makes distributed systems potentially highly available and fault-tolerant because the system can continue to work even when some parts of the system are facing failures. In a distributed fault-tolerant system, a failure may result in a degradation of services but it should not lead to an unavailability of the service.

For building reliable distributed systems, techniques like redundancy, fault avoidance, fault detection, and fault tolerance can be used [40]. In the redundancy technique, firstly, information about the failed components and the required components for recovery are obtained, and then the failed redundant components are replaced. Fault avoidance aims to reduce the probability of failure by employing techniques such as testing, debugging, verification. It also can be achieved by reducing the complexity of systems through employing higher abstraction levels to have a design with fewer functions, modules, interactions, and thus reducing failure rates. Fault detection is the first stage in a fault-tolerant system that aims to identify the failed components in the system. There are some detection techniques for distributed systems such as watchdog timer, timeouts, consistency checking, and gossip protocols. Fault tolerance techniques use redundancy to mask the faulty components by using the redundant components. Fault-tolerant languages provide a programming level mechanism for failure recovery. Erlang is reputed for its fault tolerance features that are discussed in section 2.4.

- Transparency
  A collection of independent components should appear to the system users as a single rather than as a collection of cooperating components. There are some kinds of transparency in distributed systems such as location transparency, failure transparency, and replication transparency [37]. Location transparency means local and remote components can be accessed in the same way. Location transparency is important for us because the RELEASE's target architecture (Section 2.1.3.5) is distributed at different geographical locations. Location transparency eases the access to such a large-scale architecture for users without the need of knowing the physical location of the resources. Failure transparency allows users and software applications continues to work despite the failure of hardware or software components. Replication transparency duplicates multiple instances of resources to increase reliability and performance invisibly.

2.3.2 Architectural Models

This section tries to describe a distributed system in terms of placement of computational elements and the model of communication among them. Architectural models simplify and alleviate the design complexity of distributed systems by providing a general abstract model for all problems in a particular domain [37]. There are some basic and common architectural models that are used in the development of distributed systems. In the following sections, we study the two most popular models, i.e. *client-server* and *peer-to-peer* models. We describe the key differences between these models such as their scalability and availability.

2.3.2.1 Client-Server

This is the most common and the most important architectural model in distributed systems [37]. Figure 2.6 illustrates a simple structure in which clients interact with individual server. Clients send their request to the service providers (servers) and servers respond to the requests from clients. In this model clients need to know about servers, but servers don't need to know about clients. As shown in the figure, a server may be a client of another server. For example, a web server can be client of a database server.

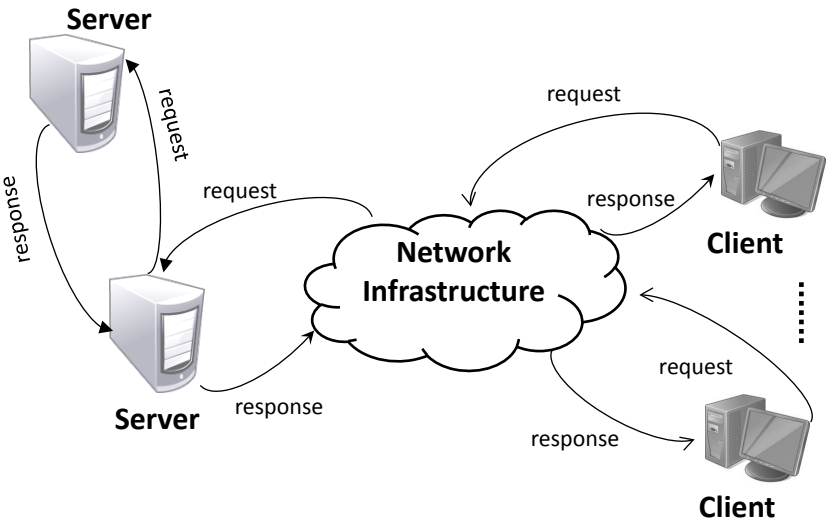


Figure 2.6: Clients Server Interaction

2.3.2.2 Peer to Peer

Peer to peer (P2P) systems are decentralized systems where computations can be done by any of the nodes in a network [37]. P2P allows autonomous computers to communicate directly with each other rather than through a central server (Figure 2.7). This model can take

advantage of all potential computational power of a network members. Moreover, P2P systems provide a better support for maintaining freshness of content. In peer-to-peer systems the network and computing resources owned by the users of a service can also be used to scale that service with the number of users.

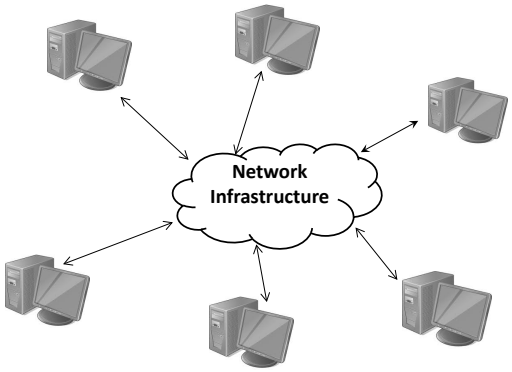


Figure 2.7: Peer-to-Peer Interaction

Comparison of the two models shows that:

- Client-server is a centralized resource that can suffer when a vital server fails. However, the P2P model improves availability by eliminating the single point of failure.
- P2P is naturally scalable and can accommodate new node arrivals easily.
- In the client-server model, a server can become overloaded with clients' requests, but in the P2P approach, loads are evenly distributed on all nodes.
- The client-server approach is prevalent in the Internet services such as HTTP, FTP, and web services, and offers a relatively simple approach for sharing data and other resources.
- Peer-to-peer systems are typically less secure in compare with client-server model because in P2P systems, the security is handled by individual computers, while in the client-server approach it is controlled and supervised centrally.

The aim of the RELEASE project is designing a high level paradigm to gain scalability for server applications. To achieve the desired scalability, the proposed architecture should control the cost of communication between the nodes by reducing the amount of communication and the cost of each individual interaction. As discussed in the previous section, decentralized architectures play a key role to reduce the server and network traffic by spreading the cost (i.e. computational workload and network traffic) over the available nodes. Since our focus is on availability and scalability, we concentrate on decentralized approaches in Chapters 3, 4, 5.

2.3.3 Communication Paradigms

Communication is one of the most important part of a distributed system. Autonomous entities in a distributed environment communicate in two general ways that are explained in following sections [41]. We discuss the scalability and availability of each of the approaches.

2.3.3.1 Direct Communication

In this type of communication, senders explicitly establish a one-way or two-way connection to the receivers and point-to-point messages are exchanged between them [41]. Two-way communications can be synchronous or asynchronous. In a synchronous communication, senders wait until the arrival of a response but in asynchronous communications, senders don't wait for response. In direct communication, senders know about the identity of receivers and usually both sides exist actively at the ends of communication line. Request-reply, remote procedure calls and remote method invocation are some examples of direct communication [37]. In this model, a sender should be aware of joining, leaving ,and relocating the system components. In large-scale systems with hundreds of thousands of entities, keeping the address and availability status of all entities becomes extremely difficult for developers and expensive for service providers.

2.3.3.2 Indirect Communication

In indirect communication, entities in a distributed system communicate through an intermediary with no direct coupling between the sender and the receiver(s) [37]. The degree of decoupling between senders and receivers is high and senders and receivers don't know about each other. Moreover, generally both sides of communication don't need to exist at the same time (time decoupling). Publish-subscribe, group communication (one-to-many communication), message queues and distributed shared memory (on a no share physical memory) are some examples of indirect communications [37]. Indirect communication is suitable for distributed environments where change is anticipated such as mobile environment where nodes can move around and leave or join frequently. Indirect communication is also used for discrete optimization problems such as ant colony optimization (ACO) in which a group of mobile biological organisms like ants communicate with others indirectly for common goals [42]. Scalability and availability are supported in this model because a sender doesn't need to know the identity of the receiver(s) and it can adapt efficiently as nodes join and leave the system. Indirect communication patterns also offer simple programming abstractions for implementing large-scale distributed applications through decoupling of communicating entities [41].

2.4 The Erlang Programming Language

This section reviews features of the languages that are used in the next chapters. We focus on parallel and distributed concurrency in the language and its support for fault-tolerance.

2.4.1 Overview

Erlang is a general purpose, concurrency-oriented functional programming language suited for building massively fault-tolerant parallel and distributed systems. Erlang employs the actor model to support scalable shared-nothing concurrency based on asynchronous message passing [34]. Open Telecommunications Platform(OTP) is a set of Erlang libraries and design principles to ease and quicken the development and maintenance of Erlang applications. Erlang code is compiled to byte-code and runs in a virtual machine (VM). Thus, Erlang applications are portable and can be run on different platforms and operating systems. Erlang processes are lightweight with small memory footprint as they are created and scheduled in the VM, independent of the underlying operating system. Due to aforementioned features, Erlang has become a popular platform to develop large-scale distributed applications, e.g. *Facebook chat backend*, *T-Mobile advanced call control services*, *WhatsApp*, and *Riak DBMS* [3, 43].

2.4.2 Why Erlang?

This section briefly describes some features that make Erlang an attractive programming language in comparison with dominant object-oriented programming languages.

- Functional Languages and Parallel Systems

  Functional languages have good potential to support parallelism because they are stateless and data structures are immutable [44]. By contrast, traditional languages have features such as shared memory, global state and locking mechanisms to avoid race conditions and deadlocks that make parallelism a burden for developers. For these reasons, changing existing sequential code into parallel code in traditional imperative languages is difficult as we need to modify large portions of the code to use threads explicitly [45]. However, in functional languages with adding a small number of high-level parallel coordination constructs, we are able to handle parallelism. Consequently, programs written in a functional language take less time to write and they are shorter and easier to read compared with their imperative equivalents [44]. Also, the programmer is free from specifying low-level error-prone coordination by providing an

abstraction over details such as process placement, communication, and synchronization issues.

- Concurrent Processes

  Concurrency in Erlang is integrated in the programming language (virtual machine) and not the operating system [34]. Creating and scheduling processes is handled by the Erlang runtime system. Erlang processes are lightweight that means creation and context switching between Erlang processes is much cheaper than operating-system thread level. Erlang concurrency is fast and scalable due to its support for lightweight processes. Processes in an Erlang system are scheduled based on the time-sharing technique in which the CPU time is roughly divided into a number of time slices and each slice is assigned to a runnable process. The default priority for all newly created processes is normal but priority of a process can be set to three different levels, i.e *low*, *normal*, and *high*. Each process has a unique identifier which is used by other processes for communication.

- Message Passing

  Erlang is a distributed concurrent system in where each process has a private memory with no possibility of shared memory and consequently no need for locking a memory space to update. Communication among processes is handled by asynchronous message passing, where message can be any kind of data structures. Each process has a queue of messages (mailbox) and new received messages are stored at the end of the queue. Messages are retrieved from the process' mailbox by using pattern matching mechanism. In a distributed Erlang system, each Erlang runtime system is called *node*. Erlang nodes can be located on the same machine or spread on several machines connected by TCP/IP protocol. In a distributed Erlang system, message passing between nodes is transparent and unique process identifiers are used for communication between processes regardless whether they are located on the same node or different nodes. Due to the location transparency, Erlang programs can be easily ported from a single computer to a network of computers.

- Garbage Collection

  Memory management in Erlang is automated and hidden from the user. Erlang supports a mechanism for freeing the space occupied by processes when they are no longer needed. Garbage collection can detect automatically when a memory space is no longer accessible, and makes it available for other processes. Each Erlang process has its own heap, and so garbage collection in Erlang is per process [46]. When garbage collection occurs for a process, the process needs to be stopped. This stop time is short (usually shorter than the time slice of a process) because process heap is ex-

pected to be small. In large scale systems, garbage collection shows good performance because larger systems tend to have more processes rather than larger processes.

- Error Handling

  Erlang is well known for its fault tolerance features. The exception handling in Erlang provides several mechanisms for detection and containment of the failures. Error detection is provided through an exception construct, and failure recovery is supported by bidirectional links. When one of the linked processes fails, the other processes, which are linked to it, will be notified through an *exit* signal.

  Open Telecom Platform (OTP) supports a behaviour module for building an hierarchical process structure called a supervision tree [47]. This structure is a tree of processes in which parent processes supervise their children for keeping them alive by restarting them when they fail. There are two kinds of processes in the supervision tree: *supervisors* and *workers*. A supervisor process is notified when one of its children fails. A child process can either be a supervisor or a worker process. A supervisor can start, stop or monitor its child processes.

2.4.3 Reliable Parallelism and Distribution in Erlang

This section discusses Erlang's supports for reliable distribution and parallelism. We study the techniques that the language employs to provide a reliable, scalable computational model for distributed environments. Firstly, concurrency and parallelism in the language is discussed (Section 2.4.3.1). Distributed computing, communication and security are explained in Section 2.4.3.2. Persistent and non-persistent large data structures are discussed in Section 2.4.3.3. Fault-tolerance and error handling are studied in Section 2.4.3.4. Finally, OTP behaviours for server processes and supervision tree are illustrated (Section 2.4.3.5).

2.4.3.1 Concurrency Oriented Programing

A program is concurrent if it may have more than one underway task at the same time [48]. A concurrent system is parallel if more than one task can be physically active at once. It is possible to have concurrent tasks on a single core using time-shared preemptive multitasking techniques. However, parallelism is not possible on single core as more than one active task cannot be physically run on single core. Erlang is a Concurrency-Oriented Programming (COP) language [34]. Erlang concurrency is based on the actor model, a mathematical theory for concurrent computation introduced in 1973 [49]. An actor is a concurrent computational entity that communicates with other actors asynchronously using message passing. Actors

are reactive, i.e. they only respond to received messages. An actor may perform three basic actions once receiving a message:

- create a finite number of new actors
- sending a finite number of messages to other actors
- determine how to respond to the next message by changing its own state.

There is no concurrency inside actors, thus an actor can only process one message at a time.

The actor model is fundamentally scalable because of its inherent concurrency of computation, dynamic creation of actors, and asynchronous message passing with no restriction on message reception order. However, the scalability of the actor implementation depends on ability of the language to implement lightweight processes. Erlang processes are lightweight because their creation, context switching, and message passing are managed by the Erlang virtual machine [47]. Isolating the processes from operating system threads makes concurrency-related operations independent of the underlying operating system and also very efficient and highly scalable.

The actor implementation in Erlang

Actors in Erlang are implemented as processes. There are three primitives for concurrent programming in Erlang [34]:

- *Pid = spawn(Fun)*: creates a new concurrent process to evaluate function *Fun*. *spawn* returns a process identifier (Pid). Pid can be used to send a message to the process.
- *Pid!M*: sends message *M* to the process with identifier *Pid* asynchronously. The message is saved in the mailbox of the process and can be retrieved and removed from the mailbox later.
- *receive*: receives a message that has been sent to the process. The receiver process uses pattern matching to retrieve and remove messages from its mailbox. If none of the messages in the mailbox matches, then the process is suspended and will be rescheduled when it receives a new message.

In addition to Pid, a process also can be referred by name. The function *register(Name, Pid)* registers the process *Pid* with the name *Name*. The function *whereis(Name)* returns the Pid associated with *Name*.

2.4.3.2 Distributed Programing

Distributed programs can be run on networked computers (nodes) and coordinate their activities by message passing [34]. Here we discuss some of the advantages that distributed applications have over stand-alone applications:

- Performance: Distributed applications improve the performance as different parts of the application can be run in parallel on different nodes. The application needs to divide the problem into smaller problems which are distributed over available nodes, and finally, the small results are collected and reassembled to form the solution. However, to achieve a better performance from a distributed application in compare with a centralized one, a proper design to minimize network traffic and a fine-grained distribution is essential.

- Scalability: A network of computing nodes can easily grow by adding new nodes to the network to increase the computational power. If a distributed application is designed properly, it should be able to exploit the available resources to improve its throughput as the problem size increases.

- Heterogeneity: Large networks such as Grids consist of different hardware and operating systems. Distributed applications are able to exploit such heterogeneous infrastructures by hiding distribution details and facilitate the efficient utilization of the systems by employing proper techniques such as transparent load balancing.

- Reliability: In a centralized environment, if the machine crashes, then there is no way for the program to recover and continue to work. However, in a distributed environment failures are partial and the system is able to function in case of component failure. If a distributed application is designed properly, it should continue to function if some parts of the system fail due to unexpected circumstances such as network outage, a hardware crash, or software faults.

Distribution in Erlang

A distributed Erlang program runs on a cluster of Erlang nodes, in which a node is an Erlang runtime system. In distributed Erlang, all nodes must have a name that is used to refer them remotely. A node has either a short or a full name by using the command line flag *-sname (short names)* or *-name (long names)* respectively [50]. Erlang nodes can be running on different computers, but there can also be multiple Erlang nodes with different names on the same computer. Erlang Port Mapper Daemon (*epmd*) translates node names into appropriate IP addresses [51]. *epmd* is part of the Erlang runtime system and is started automatically

when a node is launched on a machine. One *epmd* exists per machine, even when several Erlang nodes run on a single machine.

By default connections in distributed Erlang are transitive. For example, if node *A* connects to node *B*, and node *B* is connected to node *C*, then node *A* will connect to node *C* automatically. We can turn off this feature by using the command line flag *-connect_all false* [50]. It is also possible to turn off the automatic transitive connectivity for some specific nodes explicitly by using the concept of *hidden nodes*. Hidden nodes do not connect to the other nodes automatically and connections must be set up explicitly.

Distributed Erlang uses TCP/IP as the communication protocol by default that allows a node to connect to any other nodes with any operating system [50]. However, one can implement ones own carrier protocol for distributed Erlang. For example, distributed Erlang can use the Secure Socket Layer (SSL) over TCP/IP to get additional verification and security [52].

For security reasons, a node can communicate with those other nodes in a cluster that have the same cookie. Each node has a single cookie that can be set at its startup time with the command-line argument *-setcookie* or later by evaluating *erlang:set_cookie*. If no value is set for the cookie, the Erlang runtime system uses the value stored in the file *.erlang.cookie* located in the home directory of the user. If the file does not exist, it will be created automatically with a randomly generated value. Cookie is not a secure method as it is a small value that is shared between all nodes. An Erlang node can perform any operation on any other connected nodes in a cluster. Thus, connected nodes should be in a highly trusted environment such as the same LAN behind a firewall. For an untrusted environment, socket-based distribution can be used. This approach uses TCP/IP sockets for distribution, but its programming model is less powerful than the standard distributed Erlang [34].

Here we discuss the main distributed primitives in Erlang:

- *Pid = spawn(Node, Fun)*: creates a new concurrent process to evaluate function *Fun* on the node *Node*. *spawn* returns the identifier of the new generated process (Pid). The only difference between the local and distributed *spawn* is the first argument which specifies the node name.

- *Pid!M*: sends the message *M* to the process with identifier *Pid* asynchronously. There is no difference between sending a message to a local or a remote process. Erlang supports *process location transparency* or *communication transparency* that makes the conversion of a non-distributed application to a distributed one much easier. This also makes it possible to run a distributed application on a single node.

- *receive*: receives a message that has been sent to the process. This also behaves transparently as there is no difference between receiving a message from a local or a remote node.

- *node()*: returns the name of the local node.
- *nodes()*: returns a list of all the nodes in the cluster that this node is connected to.
- *is_alive()*: returns true if the local node can connect to other nodes.

In addition to the built-in functions (BIF), we can use some libraries for distribution. The libraries are written using the distribution BIFs, but they hide details from the programmer by providing higher level of abstractions. *rpc* and *global* are two modules to help developers to work with distribution.

rpc, which stands for Remote Procedure Call, provides a number of services to facilitate executing commands on remote nodes and collecting the results [53]. One of the most useful functions in the module *rpc* is function *call(Node, Module, Fun, Args)* that evaluates the function *Fun* on the node *Node* synchronously and returns the result.

The module *global* is handled by the global name server process which exists on every node and starts automatically when a node is started [54]. It provides functions to globally register and unregister names and monitors the registered Pids. The registered names are stored in replica global name tables on every node. The function *register_name(Name, Pid)* associates the name *Name* with the process Pid globally. The function *whereis_name(Name)* returns the Pid of the globally registered name *Name*.

2.4.3.3 Large Data Structures in Erlang

Storing and retrieving large amounts of data is required in many real-world applications. Erlang lists are suitable for small amount of data, but for large number of elements, access time becomes slow (linear access times) [47]. To lookup an element in a list, 50 percent of the elements on average should be checked to find an existing item. For non-existing items, all the elements need to be checked. Erlang provides two kinds of data structure to store very large quantities of data, i.e. *Erlang term storage* (ETS) and *Disk Erlang Term Storage* (DETS).

Erlang Term Storage (ETS)

An ETS table can store large number of Erlang terms in an Erlang runtime system with constant access time (except *ordered_set* that provides logarithmic access time) [55]. ETS is a mutable in-memory key-value store to provide a mechanism for sharing data between processes. Records in an ETS table are tuples and one of the tuple elements (by default, the first) acts as the lookup key. For example, in a tuple for students, like {*StudentID, Name, Family, Email*}, the element *StudentID* can act as the lookup key.

ETS tables are not implemented in Erlang (but are implemented in C) to provide a better performance than ordinary Erlang objects [34]. ETS tables are not garbage collected to avoid incurring garbage collection penalties for extremely large ETS tables. An ETS table is deallocated either when its owner process terminates or by using *ets:delete* function explicitly.

There are four basic operations on ETS tables:

- create a new table with *ets:new* and returns the table identifier used as reference to the table. The process that creates a table becomes the *owner* of the table.
- insert a tuple or several tuples into a table with *insert(TableId, X)*, where *X* is a tuple or a list of tuples.
- look up tuple(s) in a table for a key. *lookup(TableId, Key)* returns a list of tuples that match *Key*.
- dispose of a table by *ets:delete(TableId)*.

There are four different types of ETS: *set*, *ordered_set*, *bag* and *duplicate_bag*. Choosing a proper type of table is important for performance. Keys in a *set* or *ordered_set* table are unique, i.e. only one object associated with each key. In an *ordered_set*, the tuples are sorted based on the key. In *bag* or *duplicate_bag*, there can be more than one tuple with the same key. Two tuples in a *bag* cannot be identical. However, in a *duplicate_bag*, it is allowed to have multiple copies of a tuple.

There are three access rights for an ETS table, i.e. *private*, *protected*, and *public*. Only the owner process can access a private table. A protected table can be read by any process that knows the table identifier, but only the owner process can write to the table. A public table can be accessed and updated by any process that knows the table identifier.

ETS tables are implemented by hash tables, except the *ordered_set* type which is implemented by balanced binary trees [47]. In this approach, the position of a tuple is determined by a hash function that maps the key of the tuple to the memory location where the tuple should be stored. The hash table method incurs a small amount of memory but achieves a constant access time. For the ordered sets, the access time is proportional to the logarithm of the number of records in the table.

Disk Based Term Storage (DETS)

Many applications just need to store some terms in a file. DETS is a persistent on-disk tuple storage similar to ETS, with a few differences. DETS is much slower than ETS as disk access

is much slower than main memory access. The maximum size for DETS files is 2 GB, and if a larger table is needed, Mnesia's table fragmentation can be used [56].

DETS supports three types of table, i.e. *set*, *bag* and *duplicate_bag*. The function *dets:open_file(Filena* opens an existing table or creates an empty table if no file exists. All open DETS tables must be closed, explicitly by calling *dets:close* or implicitly when the owner process terminates, before terminating the Erlang runtime system. If a DETS table is not closed properly prior to the termination of the Erlang runtime system, a time-consuming process automatically will be started to repair it the next time it is opened.

2.4.3.4 Fault-tolerance and Error Handling

Error handling mechanisms in Erlang fall into two main categories: sequential and concurrent programs [34]. The sequential part focuses on traditional exception handling like *try...catch* expression and the ways an exception can be raised explicitly in the source code. The concurrent part is about the error handling mechanisms in concurrent programs. It provides techniques for concurrent processes to link and monitor each other.

Error Handling in Sequential Programs

Exceptions are raised either automatically by the runtime system when an errors occurs or explicitly by calling *throw(Exception)*, *exit(Exception)* or *error(Exception)* in the source code [34]. Below is a brief description of BIFs for generating exceptions:

- *exit(Reason)*: Raises an exception to terminate the current process (if the exception is not caught) and will be broadcast to all processes that are linked to the current process.
- *throw(Reason)*: Causes an exception that a caller is expected to catch. We document that our function might throw this exception and the user of this function can handle it by a try...catch expression.
- *error(Reason)*: This kind of exception is not expected to happen. The Erlang runtime system uses this to raise exceptions.

When an exception occurs, we can catch it to let the current thread continue to work. *try...catch* can be used to handle expected or unexpected errors. A general form of *try...catch* expression is shown in Code 1. First *Exprs* is evaluated and if no exception raises, then the result of evaluated *Exprs* is matched with *Pattern1*, *Pattern2*, and so on, until a match happens. If a match is found, then the corresponding body is evaluated and the result is achieved. If an exception happens within *Exprs*, then the catch patterns *ExceptionPattern1*,

Code 1: *try...catch* expression

```
try Exprs of
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
catch
    [Class1:]ExceptionPattern1 [when ExceptionGuardSeq1] ->
        ExceptionBody1;
    ...;
    [ClassN:]ExceptionPatternN [when ExceptionGuardSeqN] ->
        ExceptionBodyN
after
    AfterBody
end
```

and so on, are matched to find out the next expressions to evaluate. The keyword *after* is used for cleaning up purposes and is run immediately after any code in the try or catch sections.

When an exception occurs, the latest stack trace can be obtained by calling *erlang:get_stacktrace()*. The stack trace shows the sequence of calls that has happened before getting to the current function. Tail-recursive function calls will not be in the stack trace [34].

Error Handling in Concurrent Programs

Erlang provides simple but powerful techniques to make concurrent processes able to monitor each other and recover from failures.

Linked Processes & Exit Signal This technique provides a way that two concurrent processes become linked together to monitor and detect any possible failure that might happen in any of those. This happens between two processes if one of the processes called the BIF *link(P)*, where P is the Pid of the other process. As shown in Figure 2.8, a link between two processes is bidirectional that means there is no difference whether process A linked to process B or B to A. The two processes will monitor each other and if one of them dies, then an *exit* signal is sent to the other one. This mechanism works when both processes are located on a single node or on two different nodes. In the other words, links are location transparent and works the same in distributed and local systems.

If a linked process fails due to an uncatched exception, an *exit* signal will be sent to all the processes which are linked to the failed process. Figure 2.9 depicts when process B fails, an exit signal is sent to process A because it is linked to the fail process (process B).

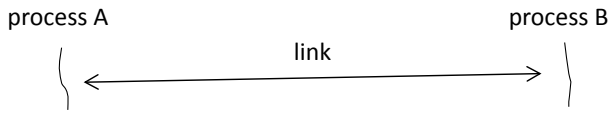


Figure 2.8: Linked Processes

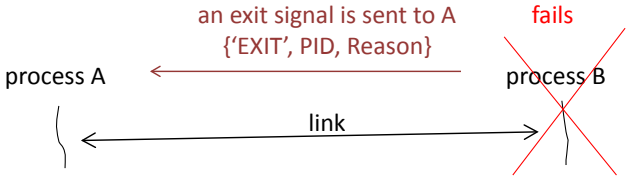


Figure 2.9: Exit Signal

A process that receives an exist signal can either trap the signal to avoid any further propagation or let the signal kill it and propagates new exit signals to all the processes that it is linked to [47]. The exit signal is a tuple with the format $\{$*'EXIT', Pid, Reason*$\}$, containing the atom 'EXIT', the Pid of the failed process, and the Reason of failure.

The exit signals can be trapped by calling the function *process_flag(trap_exit, true)*. When a process calls this function, the exit signal is converted to a messages with the format of $\{$*'EXIT', Pid, Reason*$\}$. A process can retrieve the messages using the receive construct like any other messages. A trapped exit signal does not propagate further and other processes are not affected. Links can be established implicitly by calling *spawn_link(Fun)*. This is equivalent to calling *P=spawn(Fun)* and *link(P)* respectively in an atomic way. It first creates a process and then establishes a link with the newly created process.

Monitors Links are bidirectional whereas monitors are unidirectional. Monitors are created by calling *erlang:monitor(process, Proc)* function, where Proc can be either a process identifier or a registered process name [47]. When a monitored process fails, the message $\{$*'DOWN',Reference,process,PID,Reason*$\}$ is sent to the monitoring process. The message includes a reference to the monitor, the identifier of the failed process, and the reason of failure. A monitor can be removed by calling *erlang:demonitor(Reference)* where *Reference* is the monitor reference.

2.4.3.5 OTP Behaviors

OTP, stands for the Open Telecom Platform, is a set of libraries and powerful tools that are provided with the standard Erlang distribution [34]. OTP behaviors encapsulate common behavioral patterns that are implemented in library modules for simplifying the complexity

of concurrency and error handling. These behaviors solve the generic (nonfunctional) parts of a problem, and the programmer only has to solve the specific (functional) parts by implementing the callback modules [57]. Codes written using behaviours are general, consistent, easy to read and understand, and less error-prone.

There are four standard Erlang/OTP behaviours: *gen_server*, *gen_fsm*, *gen_event*, and *supervision principles* [57].

Generic Servers

Generic servers (*gen_servers*) implement server behaviors of a client-server relation [58]. A *gen_server* consists of interface functions and callback functions. Interface functions are used by the users of a *gen_server* to access its services (callback functions). An interface function calls an appropriate behaviour function from the *gen_server* module from the OTP library. As shown in Figure 2.10, the only layer that has access to the callback functions of a *gen_server* process is OTP *gen_server* module and its behaviour functions.

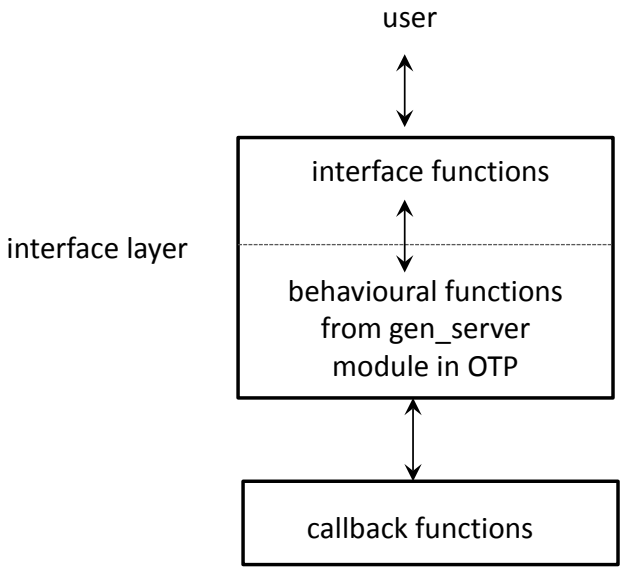


Figure 2.10: Access to a *gen_server* process

The name of interface functions are arbitrary and can be chosen by the programmer, but the name of behaviour functions and callback functions are standard and predefined in OTP. Code 2 shows the behaviour functions and their corresponding callback functions [58]:

The dotted arrows specify which behaviour function calls which callback function. As we see from the code, the function *start_link* creates a *gen_server* process by calling *Module:init/1* for initialisation. The functions *call* and *multi_call* make a synchronous call to the *gen_server* by calling *Module:handle_call/3* to handle the request. The *cast* and *abcast*

Code 2: behaviour and callback functions for gen_server processes

```
gen_server module             Callback module
-----------------             ---------------
gen_server:start_link -----> Module:init/1

gen_server:call
gen_server:multi_call -----> Module:handle_call/3

gen_server:cast
gen_server:abcast     -----> Module:handle_cast/2

-                     -----> Module:handle_info/2

-                     -----> Module:terminate/2

-                     -----> Module:code_change/3
```

functions send an asynchronous request to the *gen_server* and returns immediately. The *Module:handle_cast/2* function is called to handle the request. The callback function *Module:handle_info/2* is called when a timeout occurs or when the *gen_server* receives a message (except synchronous, asynchronous, and the system messages). The callback function *Module:terminate/2* is called when the *gen_server* is about to terminate, and does any necessary cleaning up. The callback function *Module:code_change* is called by the *gen_server* when a release upgrade/downgrade happens.

Generic Finite State Machines

A generic finite state machine process (*gen_fsm*) implements the behaviour of a finite state machine (FSM). In an FSM, when an event (*E*) occurs at the current state (*S1*), an action (*A*) is performed and the current state transits to another state (*S2*), as shown in Figure 2.11.

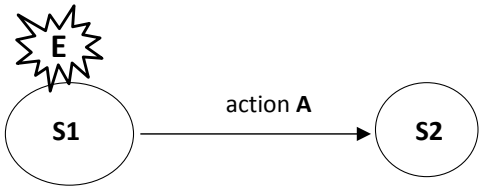


Figure 2.11: Finite State Machine

The same as a *gen_server*, a *gen_fsm* consists of interface functions and callback functions. The behaviour functions are implemented in the *gen_fsm* module from the OTP library [59]. The relationship between the behaviour functions and the callback functions for *gen_fsm*

processes are listed in Code 3 [60].

Code 3: behaviour and callback functions for gen_fsm processes

```
gen_fsm module                      Callback module
--------------                      ---------------
gen_fsm:start_link                  -----> Module:init/1

gen_fsm:send_event                  -----> Module:StateName/2

gen_fsm:send_all_state_event        -----> Module:handle_event/3

gen_fsm:sync_send_event             -----> Module:StateName/3

gen_fsm:sync_send_all_state_event -----> Module:handle_sync_event/4

-                                   -----> Module:handle_info/3

-                                   -----> Module:terminate/3

-                                   -----> Module:code_change/4
```

The behaviour function *gen_fsm:start_link* spawns and links to a new *gen_fsm* process. If the *gen_fsm* process is created successfully, then the *Module:init* callback function will be called for initialization. The *init* function is expected to return {*ok, S1, StateData1*}, where *S1* is the name of the initial state and *StateData1* is the data for the initial state (*S1*). In a *gen_fsm*, the state names are function names, e.g. if there are two states with names *S1*, *S2* in our FSM, then in the code we should have two functions with names *S1* and *S2*.

As we see from Figure 2.11, an event must occur to transit from the current state to another state. To cause an event, the function *gen_fsm:send_event(GenFsmName, E)* is used, where *GenFsmName* is the name of the *gen_fsm* and *E* denotes the event. When the event is received by the *gen_fsm*, function *Module:S1(E, StateData1)* is called which is expected to return the tuple {*next_state, S2, StateData2*}, where *S2* is the name of the next state and *StateData2* is the data for the *S2* state.

Some events can occur at any state of a *gen_fsm*. Instead of sending the message with *gen_fsm:send_event/2* for each state and writing one clause for handling the event for each state function, the message can be sent with *gen_fsm:send_all_state_event/2* and handled with *Module:handle_event/3* as shown in Code 4.

Function *sync_send_event* causes an event and waits until a reply arrives or a timeout occurs. Function *sync_send_all_state_event* can cause an event regardless of the current state. The callback function *Module:handle_info* is called when it receives any other messages rather

Code 4: behaviour and callback functions

```
%% an interface function
wherever_event() ->
    gen_fsm:send_all_state_event(Name_of_gen_fsm, Name_of_event).

...

%% a callback function to handle the Name_of_event event
handle_event(Name_of_Event, _StateName, StateData) ->
...
    {NextState, normal, NewStateData}.
```

than a synchronous or asynchronous event.

Generic Event Handling

A generic event process (*gen_event*) can handle different kinds of events [61]. This model consists of a generic event manager process with an arbitrary number of event handlers which are added and deleted dynamically. When the event manager receives an event, the event will be processed by all its event handlers. In fact, the event manager is a *gen_event* process and event handlers are implemented as a callback module. The relationship between the behaviour functions and the callback functions are listed in Code 5 [61].

As an event manager can have several callback modules which are added and deleted dynamically, *gen_event* is more tolerant of callback module errors than the other behaviours. If one of the callback function fails, the event manager will delete the related event handler, and so will not fail.

Function *start_link* creates an event manager process and returns {*ok,Pid*}, where *Pid* is the *pid* of the event manager. As we see from the Code 5, there is no corresponding callback function for *start_link*. Function *add_handler* adds a new event handler to the event manager. The event manager will call *Module:init/1* to initiate the event handler and its internal state. Function *add_sup_handler* behaves the same as *add_handler* but also supervises the connection between the event handler and the calling process. If the calling process later terminates, the event manager will delete the event handler by calling *Module:terminate*. Function *notify* and *sync_notify* send an event notification to the event manager. The event manager will call *Module:handle_event* for each installed event handler to handle the event. *notify* is asynchronous and *sync_notify* is a synchronous call. The function *call* makes a synchronous call to to a specific event handler by sending a request and waiting until a reply arrives or a timeout occurs. The callback function *Module:handle_call* is called to handle the request. *delete_handler* deletes an event handler from the event manager by calling callback

Code 5: behaviour and callback functions for gen_event processes

```
gen_event module                    Callback module
----------------                    ---------------
gen_event:start_link       ----->   -

gen_event:add_handler
gen_event:add_sup_handler  ----->   Module:init/1

gen_event:notify
gen_event:sync_notify      ----->   Module:handle_event/2

gen_event:call             ----->   Module:handle_call/2

-                          ----->   Module:handle_info/2

gen_event:delete_handler   ----->   Module:terminate/2

gen_event:swap_handler
gen_event:swap_sup_handler ----->   Module1:terminate/2
                                    Module2:init/1

gen_event:which_handlers   ----->   -

gen_event:stop             ----->   Module:terminate/2

-                          ----->   Module:code_change/3
```

function *Module:terminate* to terminate the event handler. Function *swap_handler* replaces an old event handler with a new one. The function *swap_sup_handler* replaces an event handler in the event manager with a new handler but also supervises the connection between the new handler and the calling process. The function *which_handlers* returns a list of all event handlers installed in the event manager. The function *stop* terminates the event manager and all its event handlers by calling *Module:terminate*.

Supervision Principles

OTP supervision is a hierarchical tree structure of processes to provide a generic model for monitoring the processes and keeping them alive. There are two kinds of processes in the tree: *supervisors* and *workers* [62]. A supervisor process is responsible for starting, restarting, monitoring, and stopping its child processes. A child process can either be a supervisor or a worker process. Worker processes can have one of the *gen_event*, *gen_fsm*, or *gen_server* behaviours. Figure 2.12 depicts a supervision tree with two supervisor processes and three worker processes. In the figure, supervisors are denoted as squares and workers are denoted as circles. As we see from the figure, the root process is a supervisor that its children are a supervisor and a worker processes.

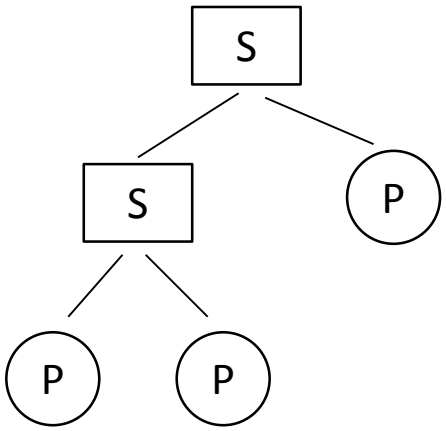


Figure 2.12: A supervision tree with two supervisors and three workers

Specifications of the children of a supervisor process are organized as a list, which is called *child specifications*. A supervisor starts its children in order from left to right according to this list. For termination, a supervisor terminates its children from right to left of the list.

The main aim of supervision tree is that a supervisor keeps its child processes alive by restarting them if they terminate abnormally. There are different strategies for restarting a process:

- *one_for_one*: if a child process terminates, it will be restarted without affecting the other child processes.

- *one_for_all*: if a child process terminates, all its other children will be terminated and restarted again.
- *rest_for_one*: if a child process terminates, all the other child processes that are created after the terminated process (based on the child specifications list), will be terminated and restarted again.

If a newly restarted process terminates repeatedly due to some reasons, then the supervisor gets into a infinite loop. To prevent such a condition, a *maximum restart frequency* is defined to limit the number of restarting a process (*MaxR*) for a period of time (*MaxT* seconds). This means, if more than *MaxR* restarts occur within *MaxT* seconds, then the supervisor terminates itself and all its children.

Specification for a child process is presented as a tuple with the format of {*Id, StartFunc, Restart, Shutdown, Type, Modules*}, where

- *Id* is an unique name that is used internally by the supervisor to refer to the child process.
- *StartFunc* is the function used to start the child process. It can be functions like *supervisor:start_link*, *gen_server:start_link*, *gen_fsm:start_link* or *gen_event:start_link*.
- *Restart* determines how a terminated process should be restarted. It can have three options: *permanent*, *temporary*, and *transient*. Option *permanent* means that the child process is always restarted. *temporary* indicates that the child process is never restarted. A *transient* child process is restarted only if it fails, i.e. terminates abnormally.
- Shutdown specifies how a child process should be terminated. If the *brutal_kill* is chosen, then child processes will be terminated unconditionally using *exit(Pid, kill)*. Alternatively, a timeout can be specified to give more time to the child process before termination. The last option is *infinity* which gives unlimited time to the child process for termination.
- Type specifies whether the child process is a supervisor or a worker.
- *Modules* determines the name of the callback module. For *gen_event* process , *Modules* is dynamic.

The function *start_link* creates a supervisor process by calling the callback function *Module:init*. *start_child* adds a child process to a supervisor dynamically. To terminate a specific process from the supervision tree, function *terminate_child* can be used. To delete and restart

a child process from the supervision tree, the function *delete_child* and *restart_child* can be used respectively. The function *which_children* returns a list of all child specifications and child processes belonging to a supervisor process.

Chapter 3

Scalable Persistent Storage for Erlang

Persistent stores are an important requirement in many software applications, and Erlang applications are no exception. A large percentage of real-world applications need to store, manipulate, and retrieve volumes of data. Database management systems (DBMS) typically manage the storage and retrieval of data persistently, offer query facilities, and control the security and integrity of the data. Relational databases (RDBMSs) have been the most commonly used database technology for decades. However, the growth in the Internet and social media have resulted in an exponential growth in data volumes and the need for wide area distributed storage and access. To underpin enterprise applications with concurrent requests from hundreds of thousands of users, NoSQL DBMSs have become essential in this context, and foremostly for Internet giants such as Amazon, Facebook, Google, and Yahoo! [63, 64, 65, 66].

This chapter investigates the provision of scalable persistent storage for Erlang in theory and practice. We start by exploring techniques and solutions used to handle vast amounts of information efficiently (Section 3.1). We enumerate the principles for scalable and available persistent data storage in Section 3.2. Section 3.3 assesses four popular Erlang DBMSs against these principles, and evaluates their suitability for large-scale architectures. The scalability, availability, and elasticity of the Riak NoSQL DBMS are measured up to 100 hosts practically in Section 3.4. The findings and conclusions of the chapter are discussed in Section 3.5. The work in this chapter is reported in [5, 6, 7].

3.1 Introduction

Many large-scale applications and services require to store their data persistently, and have to handle requests from a large number of users. Legacy relational databases cannot fulfill some of the requirements that are demanded in modern high-performance business applications [67].

- In relational approaches, retrieving and analyzing data is a time-consuming task, which degrades the productivity and performance of the users of business applications, who need quick systems to gain more profits for their organizations. In relational databases, most queries need to perform join operations to reconstruct the entities that business applications need, and this process can be extremely slow and resource intensive for large amount of data and relatively complex queries.

- Programming languages have abstractions of objects and methods whereas relational databases consist of rows and tables and are accessible via query languages such as SQL. To cover the mismatch between the data model of databases and programming languages, developers require to either implement a mapping layer between applications and databases, or embedding SQL statements directly in their code and using a vendor-specific API to communicate with the databases. This affects the productivity of developers as they need to spend a significant amount of time to handle the mismatch between the two sides through developing the mapping layer, rather than focusing on the business logic of the application itself.

- Data models are designed based on business requirements. As business requirements are likely to change over time, a flexible data model with separation from the structure of client applications is desirable. Sometimes the changes might happen too fast that during constructing a data model, the model becomes out of date before completion. Moreover, the relational data model focuses on data as a set of rows and columns, but some kinds of information cannot be fitted easily into this model. As the data model in relational databases is strict and not flexible to accommodate possible changes rapidly, organizations cannot adapt their systems to the evolving requirements accordingly.

- As there is a strong dependency between data models in databases and applications, any change in data requirements will impose a huge impact on every application that uses the database. Sometimes for improving the performance of the system we need to redesign the database to support more volumes of data, which causes the corresponding impacts on the applications that use the database.

- With the advances in a variety of software, hardware, and networking technologies, centralized databases have been distributed into locally or geographically dispersed

sites. In this approach data is partitioned over a set of commodity servers, where each server accommodates a smaller chunk of data. Servers maintain and use metadata to route the request to the appropriate server. Although partitioning makes it possible to handle more data and users, it can degrade the performance of query operations that need to join data from different partitions. Also, keeping consistency between data held in different partitions using transactions can cause significant performance degradation.

- Relational databases guarantee the consistency of data for concurrent users by implementing ACID (Atomic, Consistent, Isolated, and Durable) transactions. In an ACID transaction, updating multiple tables is considered complete only if every update completes successfully and all parts of the transaction succeed. If any part of the transaction fails, then the entire transaction is rolled back and all the updates that already got completed are undone. To implement ACID transactions, most relational databases use lock mechanisms to provide a consistent view of the data for users while the data is changing. During the transaction, all access to the data is blocked until the lock is released. For short-lived transactions, concurrent users may only be delayed for a short while, but for longer transactions, the delay would be noticeable. Lock mechanisms in ACID transactions work well in small-scale systems, but for larger-scale it can become a bottleneck for scalability and performance.

- In relational databases, availability is provided by maintaining copies of the database hosted by separate servers. User requests can be redirected to backup servers either if the primary server fails or due to a heavily loaded primary server. However, keeping both high availability and strong consistency is not possible because in a strong consistency model, updates on data must be applied on all copies to keep the data consistent. Consequently, no update is possible while some nodes are unreachable due to node or network failure, and so the system becomes unavailable for any update.

- Licensing costs for many relational database systems that support scalability and high-availability features are very expensive and a barrier for small to medium sized business that have to manage large amounts of data but only have a limited budget. However, many NoSQL DBMSs are open source, which makes them a very attractive option for such organizations.

3.1.1 NoSQL DBMSs

A number of web-scale companies realized that the relational technology could not provide the scalability, availability, and performance that they required for their large-scale systems with massive amounts of data and large numbers of concurrent users [63, 64, 65, 66]. To

fulfill their requirements, they develop their own non-relational databases, and in some cases, released them as open source. This category of big data storage is generally referred to as *NoSQL*. The term NoSQL conveys the idea of not using SQL to interact with the database but the only feature that is common to them is that they are not relational. However, we can categorize NoSQL DBMSs based on the data model they support: *key/value*, *document*, *column-family*, and *graph* databases [67].

Key/Value Stores

Key/value databases use hashing techniques to store records of data based on each record's unique key. A hashing function is executed on a record's key to determines where the record should be stored, as depicted in Figure 3.1. This technique performs efficiently over a cluster of storage nodes by spreading the load evenly across the available nodes. Key/value stores typically support simple query, insert, and delete operations. To modify a value, the existing data is overwritten with the new value. Key/value stores are schema-less and can store whatever format of data, although the maximum size of values is imposed in some stores. Example key/value databases include Riak, Redis, and Amazon Dynamo DB [68, 69, 63].

Document Stores

Document stores are similar to key/value stores except that a value is a collection of hierarchical value-pairs like XML, JSON, and YAML. Figure 3.2 presents an example of documents with hierarchical value-pairs. As we see from the figure, the field *Order* consists of three other fields, i.e. *ProductID*, *Quantity*, *Cost*. Updating and retrieving multiple fields in a single document are usually performed atomically.

Since the values in a document are transparent and structured, we can index the fields of a document, and queries are not limited only to query by the key. Moreover, the documents in a

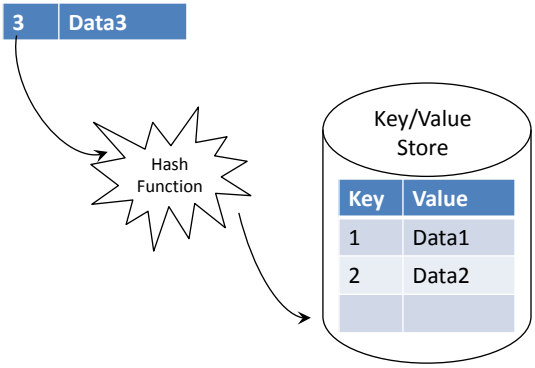


Figure 3.1: Key/value store and hash function

document database can have different structures, and this feature makes the system adaptable to the changes of business requirements. Also, document databases allow modification of the values of specific fields in a document without updating the entire document. Examples of document stores are SimpleDB, CouchDB, and MongoDB [70, 71, 72].

Column-Family Databases

In a column-family database, data is organized into rows and columns, conceptually similar to the relational model. A row is a key-value pair, where the key is mapped to the value that is a set of columns, i.e. *column-family*. A column-family includes a set of columns that are logically related together and more frequently are referred to for query purposes. Figure 3.3 shows how related columns such as personal information (including *name*, *gender*, and *age*) form a column-family and the other related columns (including *degree* and *year*) form another column-family (*qualification*) are organized.

The column-family model aims to optimize queries by reducing the disk activity in a centralized model, and reducing network communication when the table is distributed over a network. In this approach related columns are stored as a column-family (called data locality), so aggregating them to generate required queries is significantly quicker than the traditional row-column model, where all the columns in the table should be traversed regardless of whether they are needed for the query. Examples of column-family databases are HBase, Cassandra, and BigTable [73, 74, 65].

Key	Document
100	Name: Richard Age: 30 City: London Orders: ProductID: 401 Quantity: 3 Cost: 20 ProductID: 530 Quantity: 1 Cost: 40 TotalCost: 100
101	Name: Joe Age: 35 City: Glasgow Orders: ProductID: 200 Quantity: 2 Cost: 50 ProductID: 320 Quantity: 5 Cost: 10 TotalCost: 150

Figure 3.2: An example of document-oriented data model with nested values per key

Graph Databases

A graph database stores entities (nodes) and their relations (edges) to provide an efficient way to perform queries. Each node or edge has a number of properties that provide information about them. A node can have arbitrary number of properties (fields) and edges can have direction which indicates the dependency between nodes. Figure 3.4 presents a simple graph in which nodes denote *employees* and *offices* and edges indicate the relations between these entities. As we see from the figure, for example finding who are managed by *Robert* can be done quickly by following the incoming edges to the *Robert's* node. Graph databases including Neo4j, Graphbase, and InfiniteGraph [75, 76, 77].

3.1.2 The Scalability and Availability of NoSQL DBMSs

Web scale systems employ NoSQL DBMSs to provide the required scalability and availability for their users [64, 63]. In such systems that are distributed across multiple data centers achieving fault tolerance and availability, tunable consistency, and massive scalability with hundreds of terabytes of data is not possible with traditional relational databases [78]. Small

Key	Column Families	
	Personal Info	Qualification
1000	Name=Robert Gender=male Age=31	Degree: Master Year: 2006
1001	Name=Susan Gender=female Age=28	Degree: Bachelor Year: 2005
1002	Name=Albert Gender=male Age=44	Degree: PhD Year: 2003

Figure 3.3: A example of column-family data model

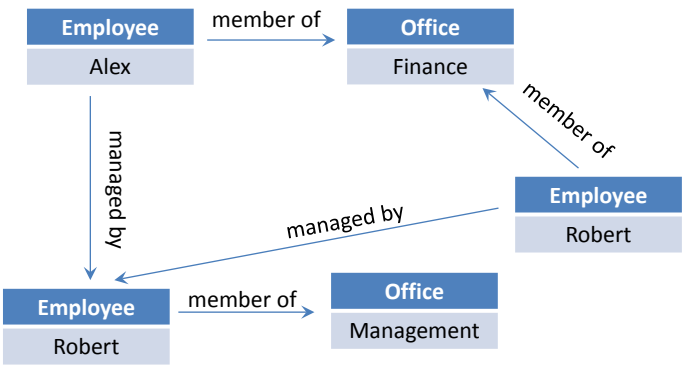


Figure 3.4: An example of graph-oriented data model

to medium-sized companies also have shown their willingness to employ NoSQL technology as using clusters of inexpensive commodity servers has become commonplace.

This section describes the technologies and techniques that NoSQL databases offer to support scalability and availability.

Availability

Availability in many of NoSQL DBMSs is provided through clustering [67]. In this approach data is replicated on several different machines (nodes) in the cluster, and in case of any node failure, requests will be transparently handled by one of the available replicas. Some NoSQL DBMSs support replicating data across multiple data centers. This provides availability even in the event of natural disasters like floods or earthquakes as the copies of data are accessible at different geographical locations.

However, achieving a high level of availability with strict consistency in a distributed environment where network partitions and node failures are common, is not possible [79]. In the strong consistency model, an update operation is considered complete only if all replicas are available and acknowledge the operation. Thus, if one of the replicas becomes unavailable, no update will be possible. NoSQL database solutions are designed with emphasis on high availability and tunable consistency that lets the user decide about the degree of consistency and availability. Many real world applications and Internet-based services such as email, blogging, social networking, search and e-commerce can tolerate relaxed consistency and a certain degree of stale data. Thus, there has to be a trade-off between consistency and availability to provide a highly available service with an acceptable degree of consistency [80]. There are several consistency models in distributed systems [78]:

Strict consistency

This is the most stringent level of consistency in which after an update, all the subsequent reads will always return the updated value. Providing this level of consistency with an acceptable performance is possible on a single machine. However, in a system distributed across multiple machines (nodes) some communications protocol are required to keep all the nodes consistent. If one node fails or becomes unreachable, the system stops to respond.

Causal consistency

This model is a weaker form of consistency in comparison with the strict consistency. It is used in the concurrent environment to guarantee that an operation is executed only if its causal predecessors are executed. In other words, causal writes must be read in sequence, but concurrent operations (i.e. not causally related ones) can be executed

in parallel in different orders. Causality does not necessarily lead to a consistent state eventually, since updating two copies of the same data may be done by two operations with no causal relationship [81]. *Vector clocks* can be used to ensure ordering between causally related events.

Eventual consistency

This is the most widely used consistency model in large-scale distributed systems to achieve high availability. This model guarantees that all the updates to a replica will eventually reach to the other replicas, and so all replicas will gradually become consistent [82]. This improves performance by reducing the latency of operations that update replicas which are distributed over a large network. When a user runs an operation, he or she does not have to wait for acknowledgment from all the replicas, and the user gets reply as soon as a quorum of replicas acknowledge the operation (*write quorum*). In addition to improving the performance, this also improves availability because system continues to work even if some of the replicas do not reply due to network partition or node failure.

However, conflicts between replicas are inevitable in this model because concurrent updates on replicas may cause inconsistency between replicas. Conflicts can be detected by quorum based replication approach in which a value is consistent if a majority of the replicas agree on the value (*read quorum*). Otherwise, the value is considered inconsistent and a conflict resolution mechanism can be used for reconciliation. Another commonly used approach to conflict detection is *vector clocks* [81]. In this approach each node in a cluster uses a counter that shows the number of times it has updated an item. The counters are used during synchronization between nodes to detect any conflict between replicas. A conflict resolution method tracks the causal history of a piece of data generated by techniques such as vector clocks and timestamps to resolve the conflict. The conflicting versions usually are provided to client applications to be solved and NoSQL DBMSs do not by themselves try to resolve them [83].

The number of servers in read or write quorum, in addition to defining the level of consistency, affects the performance and scalability of the database. As a write quorum is larger, more servers must be updated before an update is considered successful. In a read operation also, as the quorum is larger, then more servers should return the same data before the operation completes successfully.

Scalability

Scalability is the main objective of many NoSQL databases. One of the techniques that NoSQL DBMSs employ to improve the scalability is *sharding*, i.e. partitioning data over a

cluster of nodes evenly [67]. Sharding improves the scalability as more nodes can be added to the cluster to handle more concurrent requests. To take advantage of the new nodes, the store can be reconfigured either by the database administrator explicitly or by the database itself automatically. Elastic scalability means that the database can distribute its load on the newly joined nodes seamlessly without reconfiguring the entire cluster [78]. Some NoSQL databases provide auto-sharding, where the database takes on the responsibility of spreading data across servers, without requiring client applications to participate. Auto-sharding makes it possible for servers to join and leave the network without any downtime experienced in client applications. Avoiding splitting related information and keeping them in the same shard improves scalability and performance by reducing network traffic and the latency of requests. In large-scale systems that are distributed at different geographically sites, sharding can decrease the network latency by hosting most likely to access information at closer sites to the users.

3.2 Principles of Scalable Persistent Storage

This section outlines the principles of highly scalable and available persistent storage, i.e. *data fragmentation*, *data replication*, *partition tolerance*, and *query execution strategies*.

Data fragmentation

Data fragmentation techniques divide the data into smaller units of data (fragments) and map the fragments to be stored on different computers in a network. This improves performance by spreading loads across multiple nodes and increases the level of concurrency. Figure 3.5 shows how 20KB data is fragmented and distributed over 10 nodes in a cluster. This improves the performance as the burden is equally shared among 10 nodes instead of one node only. Moreover, this improves the concurrency of the system as 10 times more requests can be handled in parallel by the cluster nodes.

A scalable fragmentation approach should have the following features:

1. Decentralized model: In a decentralised model data is fragmented between nodes without central coordination. The model shows a better throughput in large systems because of reducing the possibility of overloading the coordinator nodes, and also provides a higher availability in comparison with the centralised models due to eliminating single points of failure [63].
2. Automatic load balancing: Data fragmentation implicitly encompasses load balancing by distributing data among available nodes. A desirable load balancing mechanism should take the burden off developers and handle it automatically

at the server side. An example for such decentralized automatic fragmentation technique is consistent hashing [79].

3. Location transparency: Placing fragments at proper locations in a large scale system is a tricky task to undertake. Therefore, the placement should be carried out automatically and systematically without developer interference. Moreover, a reorganisation of database nodes should not impact the programs that access the database [84].

4. Scalability: A node departure and arrival should only affect the node's immediate neighbours, and the other nodes should be unaffected to have a better scalability. In a large-scale system, affecting whole nodes in the system per each join or leave can be costly and limit the scalability. A new node should receive approximately the same amount of data as other available nodes have, and when a node goes down, its load should be evenly distributed between the remaining neighboring nodes [63].

Data Replication

Data replication improves the performance of read-only queries by providing data from the nearest replica. The replication also increases the system availability by removing single points of failure [85]. Figure 3.6 shows record X is replicated over three nodes, and the record is accessible even if two of the three nodes fail or become unreachable due to network partition.

A scalable mechanism for replication should have the following features:

1. Decentralized model: Data should be replicated to nodes without using the concept of master, so each node has full DBMS functionality [63]. A P2P model is desirable because each node is able to coordinate the replication and thus improves overall performance by distributing the loads and network traffic over the available nodes.

2. Location transparency: The placement of replicas should be handled systematically and automatically because managing the location of replicas manually

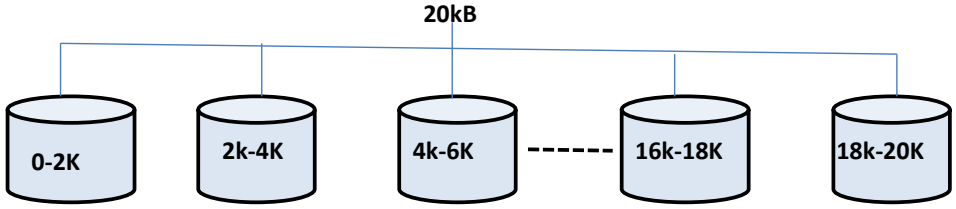


Figure 3.5: 20KB data is fragmented evenly over 10 nodes

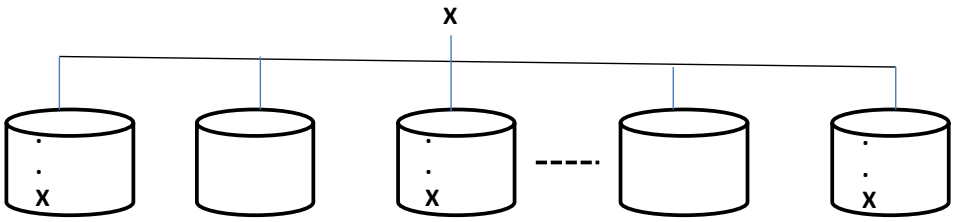


Figure 3.6: Replication of record *X* on 3 nodes

could be a difficult task in large-scale systems with huge number of nodes [84].

3. Asynchronous replication: A `write` command is considered complete as soon as the local storage acknowledges it without necessity to wait for the remote acknowledgments [39]. In large-scale systems with slow communication, such as wide area networks, waiting to receive the confirmation of an update from all replicas is time consuming. Many Internet-scale applications, e.g. email and social network services, may be able to tolerate some inconsistency among the replicas to provide better performance. The approach may violate the consistency of data over time, so *eventual consistency* can be used to address the problem. Eventual consistency is a specific form of weak consistency where updates are propagated throughout the system, and eventually all participants have the last updated value [82]. The Domain Name System (DNS) is the most popular system that employs eventual consistency.

Partition Tolerance

Partition tolerance is essential to cope with node failure and network disruption in loosely coupled large-scale distributed systems. A highly available system should continue to operate despite loss of connections between some nodes. The CAP theorem states a distributed system cannot simultaneously guarantee consistency, availability, and partition tolerance [39].

As depicted in Figure 3.7, network partitions are inevitable in a distributed environment, and so achieving both availability and consistency simultaneously is not possible. To achieve partition tolerance and availability, strong consistency must be sacrificed. Eventual consistency can improve availability by providing weakened consistency.

Query Execution Strategies

Query execution strategies provide mechanisms to retrieve specific information from a database. In addition to providing good performance, a desirable query processing approach hides low-level details including replicas and fragments [85]. A scalable query strategy should have the following features:

1. Local execution: In geographically distributed systems where data is fragmented over nodes, query response times may become very long due to communication latency. Scalable techniques like MapReduce try to reduce the amount of transferring data between the participating nodes in a query execution by *local processing*, i.e. passing the queries to where the data lives rather than transferring data to a client [86]. After the local processing the results of local executions are combined into a single output.
2. Parallelism. In addition to improving the performance, the local processing technique increases parallel execution by dividing the query into sub-queries and spreading them over multiple nodes. A good strategy should exploit the local resources as much as possible to maximize performance.
3. Fault tolerance. Node or network failures are common in large scale databases distributed over cluster of nodes. The query coordinator should tolerate failures by masking the failures and running the query on available replicas.

3.3 Scalability of Erlang NoSQL DBMSs

This section analyses the following four popular NoSQL DBMSs for Erlang systems: Mnesia (Section 3.3.1), CouchDB (Section 3.3.2), Riak (Section 3.3.3), and Cassandra (Section 3.3.4). The databases are evaluated in terms of scalability and availability against the principles outlined in Section 3.2.

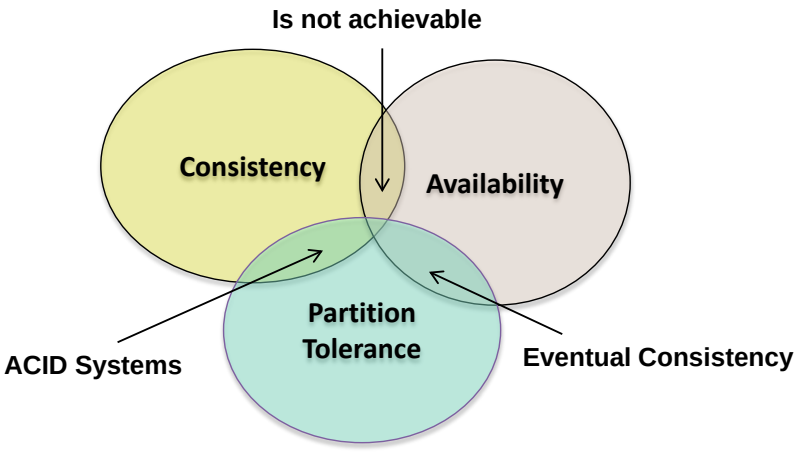


Figure 3.7: CAP Theorem

3.3.1 Mnesia

Mnesia is a distributed DBMS written in Erlang and comes as part of the standard Erlang distribution [87]. The Mnesia data model consists of tables of records. Attributes of each record can store arbitrary Erlang terms. A database schema contains the definition of all tables in the database. An empty schema is created when a database is created and configured, and over time it gets completed. Code 6 shows the creation of a persistent database (on disc) on a number of nodes. A local Mnesia directory is created on every node and these directories cannot be shared between the nodes. For a memory-only database, the schema is created in RAM that will not survive restarts.

Code 6: Creating schema in Mnesia

```
mnesia:create_schema([Nodes]).
```

Mnesia provides ACID (Atomicity, Consistency, Isolation, Durability) transactions, i.e. either all operations in a transaction are applied on all nodes successfully, or, in case of a failure, no node is affected. In addition, Mnesia guarantees that transactions which manipulate the same data records do not interfere with each other. To read from, and write to, a table through a transaction, Mnesia sets and releases locks automatically. Code 7 shows an example of executing a function with name *Fun* in a transaction to guarantee that either all or none of the operations in the function are performed.

all or none of the

Code 7: Transactions in Mnesia

```
mnesia:transaction(Fun).
```

In Mnesia, fault tolerance is provided by replicating tables at different Erlang nodes. Replicas of a table need to be explicitly specified by the client applications. For example, Code 8 shows how the `student` table is replicated on three Erlang VMs (*node1*, *node2*, and *node3*).

Code 8: An explicit placement of replicas in Mnesia

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

We also see from the code that the tuple {*index, fname*} indexes the table by the field *fname*. This makes it possible to look up and manipulate records using the index field.

In general, to read a record only one replica of that record is locked (usually the local one), but to update a record all replicas of that record are locked and must be updated. This can

become a bottleneck for write operations when one of the replica is not reachable due to node or network failures. To address the problem Mnesia offers *dirty operations* that manipulate tables without locking all replicas. However, dirty operations do not provide a mechanism to eventually complete the update on all replicas, and this may lead to data inconsistency.

Another limitation of Mnesia is the size of tables. The limitation is inherited from DETS tables, and since DETS tables use a 32 bit file offset, the largest possible Mnesia table per an Erlang VM is 2 gigabytes. This limitation is just for the *disc_only_copies* storage type, and the other storage types such as *ram_copies* do not have such limitation, as in these model, tables are kept in RAM non-persistently. To cope with large tables, Mnesia introduces a concept of *table fragmentation*. A large table can be split into several smaller fragments on different Erlang nodes. Mnesia employs a hash function to compute the hash value of a record key, and then that value is used to determine the fragment where the record belongs to. The downside of the fragmentation mechanism is that the placement of fragments should be specified explicitly by the user. Code 9 shows an explicit placement of fragments in Mnesia.

Code 9: An explicit placement of fragments in Mnesia

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

Query List Comprehensions (QLCs) is an Erlang query interface to Mnesia [88]. QLCs are similar to ordinary list comprehensions in Erlang programming language, e.g. Code 10 returns the `name` of student with `id=1`.

Code 10: A query example in Mnesia

```
Query = query [S.lname ||
                S <- table(student), S.id == 1] end,
Result = mnesia:transaction(
                 fun() -> mnemosyne:eval(Query)
          end).
```

Suitability for Large Scale

Mnesia is appropriate for telecommunications applications and other Erlang applications which require fast real-time operations, fault tolerance, distribution, and the ability to re-configure the system dynamically at runtime without stopping the system through schema manipulation. However, it has significant limitations for large-scale systems as follows:

- *Explicit placement of fragments and replicas*. Mnesia does not handle replication and fragmentation automatically and developers must explicitly handle these in client applications which is a difficult and error prone task particularly in large-scale systems.

- *Limitation in size of tables*. The largest possible Mnesia table per an Erlang VM is 2 gigabytes. Larger table should be split over several Erlang nodes.

- *Lack of support for eventual consistency and high availability*. Mnesia uses a two-phase locking technique to provide ACID transactions . ACID provides strong consistency and thus reduces availability since write operations cannot be performed while one of the replica is not reachable due to node or network failures.

3.3.2 CouchDB

CouchDB (Cluster Of Unreliable Commodity Hardware) is a schema-free document-oriented database written in Erlang [89]. Data in CouchDB is organised in the 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 it to handle searches, insertions, and deletions in logarithmic time. Instead of traditional locking mechanisms for concurrent updates, CouchDB uses the Multi-Version Concurrency Control (MVCC) to manage concurrent access to the database. With the MVCC a system is able to run at full speed all the time, even when a large number of clients uses the system concurrently. When a document is changed, there will be two versions of that document, i.e. the old and the new versions. In this model, a record can be updated while a read operation is being processed because the update is done on a completely new version of the document, and so, it can proceed without having to wait for the read operation to finish. Figure 3.8 illustrates how the operation *update* starts while the operation *read1* is still being processed. As *read1* started before *update1*, it returns *version1*. However, the operation *read2* returns *version2* as it starts after the completion of *update1*.

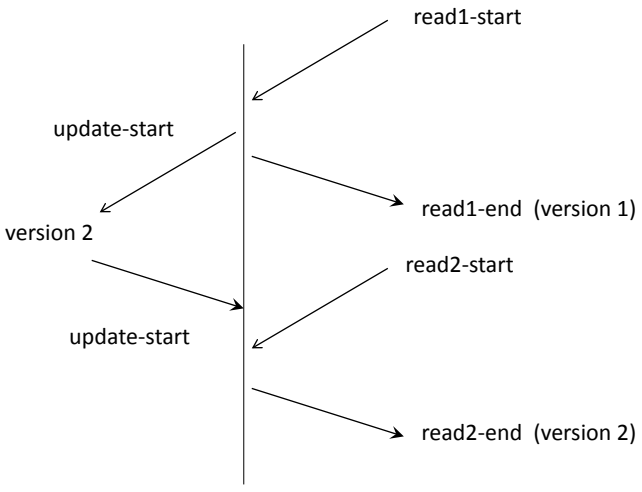


Figure 3.8: Concurrent operations in the MVCC model

Figure 3.9 shows a conflict occurrence in the MVCC approach. In this scenario, *user2* does not know about the change that *user1* has made previously. Thus, CouchDB does not accept the update from *user2* as it likely overwrites the data that he does not know existed. In other words, whoever saves a change to a document first, wins.

In CouchDB, the creation of views and report aggregation is implemented by joining documents using the Map/Reduce technique [71]. A view in CouchDB is a combination of map and reduce functions. Each document in CouchDB has an unique ID per database. It is possible to choose an arbitrary string as the ID, but a Globally Unique IDentifier (GUID) is recommended to ensure two concurrent operations cannot create two different documents with the same ID. Code 11 shows a general format that a map function has to create a view. A map function gets every document in the database as parameter. The built-in *emit(key, value)* function always takes two arguments and creates an entry in the view result. The view result is stored in a B-tree and is created only once when we query a view for the first time. If a document is changed, the map function is called only for that single document to recompute its keys and value.

Code 11: General format of a map function to create a view

```
function(document) {
    var key, value;
    if (document.field1 && document.field2 && ...&& document.fieldN
       key=do_some_calculation1(document.field1, document.field2, .
       value=do_some_calculation2(document.field1, document.field2,
       emit(key, value);
   }
}
```

Data fragmentation over nodes is handled by Lounge, a proxy-based partitioning/clustering framework for CouchDB [71]. To find the shard where a document should be stored, Lounge applies a consistent hash function to the document's ID. Lounge allocates a portion of the

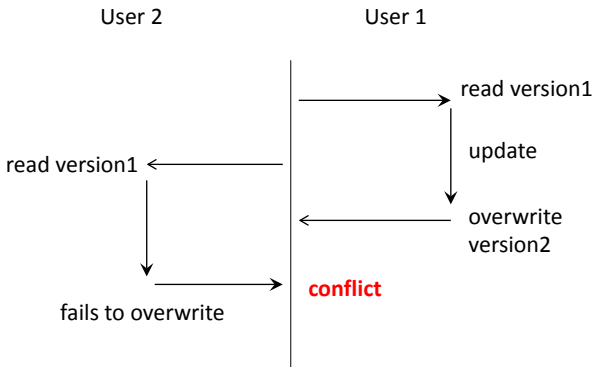


Figure 3.9: Conflict occurrence in MVCC model

hash to each node, and so, we can add as many nodes as needed. Consistent hashing ensures that all nodes get roughly equal load. Lounge does not operate on all CouchDB nodes. In fact, Lounge is a web proxy that distributes HTTP requests to CouchDB nodes. Thus, to remove a single point of failure and to improve the performance, multiple instances of Lounge should be run with a proper load balancing across them, i.e. a multi-server model.

CouchDB replication system synchronizes all copies of the same database by sending the most recent changes to all other replicas. Replication is an unidirectional process, i.e. the changed documents are copied from one replica to another and not automatically vice versa. Currently, replica placements are handled explicitly (Code 12).

Code 12: An Explicit Placement of Replicas in CoachDB

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

In Code 12, ``'continuous'':true` means that CouchDB will not stop the replication and will automatically send new changes of the `source` to the `target` by listening to the CouchDB API changes. The continuous changes API allows the connection will be held open for sending notifications to the target node. CouchDB has eventual consistency, i.e. document changes are periodically copied to replicas. In the synchronization process between two replicas,firstly the two databases are compared to find out which documents on the source differ from the target and then the changed documents, including new documents, changed documents, and deleted documents, are submitted to the target. Documents that already exist on the target with the same vision are not transferred. A conflict occurs when a document has different information on different replicas. In case of a conflict between replicas CouchDB employs an automatic conflict detection mechanism. CouchDB does not attempt to merge the conflicting visions but only attempts to find the latest version of the document. The latest version is the winning version. The other versions are kept in the document's history, and client applications can use them to resolve the conflict in an alternative way that makes sense for the application.

Suitability for Large Scale

CouchDB provides high availability by supporting eventual consistency. CouchDB uses MapReduce to compute the results of a view in parallel efficiently especially when data is partitioned over a cluster of nodes. Instead of traditional locking mechanisms, CouchDB uses Multi-Version Concurrency Control (MVCC) to manage concurrent access to the database.

However, CouchDB has the following limitations for large-scale systems:

- *Explicit placement of replicas*. Code 12 shows that replica placement is handled explicitly in CouchDB which can be a difficult task to undertake in large-scale systems.
- *Multi-server model to coordinate fragmentation and replication*. Assigning specific nodes for coordinating fragmentation and replication can become a bottleneck for scalability. A desirable and scalable model is P2P in which all nodes participate in the process to avoid overloaded nodes and network congestion.
- *Lounge program that handles data partitioning is not a part of every CouchDB node*. Lounge is a proxy-based partitioning/clustering framework that does not operate on all CouchDB nodes.

3.3.3 Riak

Riak is a NoSQL, schemaless, open source, distributed key/value data store primarily written in Erlang [68]. Riak is scalable, available, and fault tolerant database suitable for large-scale distributed environments, such as Clouds and Grids. Riak is highly fault tolerant due to its masterless structure that offers no single point of failure.

Riak is a key/value store that organizes key/value pairs into Buckets. In other words, data is stored and referenced by bucket/key pairs. Buckets allow the same key to exists in multiple buckets, similar to tables in relational databases or folders in file systems. We can also define per-bucket configuration, i.e. all the keys and values within a bucket can have common configurations. For example, all the objects in a bucket can have the same replication factor. There are two ways to access data in Riak, i.e. HTTP based REST API and Protocol Buffers API. Appendix A.1 compares Riak's interfaces on a 20-node cluster practically. For storing an object HTTP methods *PUT* and *POST* are used, and for fetching an object HTTP method *GET* is called. Code 13 shows the command form for retrieving a specific key (*KEY*) from a bucket (*BUCKET*). If the key exists, the response will contain the corresponding value for the key. Also, the command form for storing an object under the specific bucket/key is shown in Code 14.

Code 13: Retrieving a Specific Key from a Bucket

```
GET /buckets/BUCKET/keys/KEY
```

Code 14: Storing an object under the specified bucket/key

```
PUT /buckets/BUCKET/keys/KEY
```

Data fragmentation in Riak is handled implicitly using a consistent hashing technique. The hash space is divided into equally-sized partitions called virtual nodes (vnodes). Physical

servers in the cluster (nodes) accommodate a certain number of vnodes. This approach manages joining and leaving nodes seamlessly. Newly joined nodes host some of the partitions from the existing nodes and the neighbors of a failed node handle its tasks automatically.

For availability, Riak uses replication and hand-off techniques. By default each data bucket in Riak is replicated on three different nodes. However, the number of replicas (N) is a tunable parameter and can be set for every bucket. Other tunable parameters are read quorum (R) and write quorum (W). A quorum is the number of replicas that must respond to a read or write request before it is considered successful. If $R + W > N$ then Riak provides a strong consistency which guarantees that a subsequent accesses returns the previously updated value. When $W + R <= N$ Riak provides a weak consistency and some nodes may keep outdated data.

When communication with a node is lost temporarily due to node failure or network partition, a hand-off technique is used, i.e. neighbouring nodes take over the duties of the failed node. When the failed node comes back a Merkle tree is used to determine the records that need to be updated. Each node has its own Merkle tree for the keys it stores. Merkle trees reduce the amount of data needed to be transferred to check the inconsistencies between replicas.

Riak provides eventual consistency in which an update is propagated to all replicas asynchronously. However, under certain conditions such as node failures and network partition, updates may not reach to all replicas. Moreover, concurrent updates can lead to inconsistent replicas. Riak employs a causality-tracing algorithm called vector clocks. One vector clock is associated with every version of every object. Vector clocks can be used later to determine whether two versions of an object have a causal ordering or causality is violated. Two version *Ver*1 and *Ver*2 are causally ordered if *Ver*1's clock is less than or equal to the *Ver*2's clock, i.e. *Ver*1 is an ancestor of *Ver*2. If there is no causality in either direction between the two versions, a conflict has occurred, and so reconciliation is required to determine the winner between the two versions. Riak offers two ways for resolving a conflict, either the last update can be considered as the winner automatically or both versions return to the client to be resolved by the client itself.

The default Riak backend storage is Bitcask. Although Bitcask provides low latency, easy backup, restore, and is robust in the face of crashes, it has one notable limitation. Bitcast keeps all keys in RAM and therefore can store a limited number of keys per node. For this reason Riak users may use other storage engines to store billions of records per node.

LevelDB is a fast key-value storage library written at Google that has no Bitcask RAM limitation. LevelDB provides an ordered mapping from keys to values whereas Bitcask is a hash table. LevelDB supports atomic batch of updates that may also be used to speed up large updates by placing them into the same batch. A LevelDB database may only be opened by one process at a time. There is one file system directory per each LevelDB database where all

database content is stored. To improve the performance, adjacent keys are located in the same block. A block is a unit of data used to transfer data to and from the persistent storage. The default block size is approximately 8192 bytes. Each block is individually compressed before being written to persistent storage, however, compression can also be disabled. It is possible to force a checksum verification of all data that is read from the file system. Eleveldb is an Erlang wrapper for LevelDB included in Riak, and does not require a 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 search is required to reach the last level. But if 1% fits in memory, LevelDB will require two searches. Appendix A.2 compares Riak's backends on a 20-node cluster practically.

Non-key-based queries in Riak can be done through the MapReduce queries. Both the HTTP API and the Protocol Buffers API support submitting the MapReduce queries. Firstly, a client submits a query to a node. That node will coordinate the MapReduce job. The query has a map function that operates on a list of objects, for example all objects in a bucket. The coordinator node routes the function to be run at appropriate nodes where the objects exist. The coordinator node collects the results and passes them to a reduce function (this step is optional), and finally returns the results to the client.

Suitability for Large Scale

The theoretical analysis shows that Riak meets scalability requirements of a large-scale distributed system. The following summarize the requirements:

- *Implicit placement of fragments and replicas*. Riak employs consistent hashing to partition and replicate data over the cluster automatically. Client applications do not need to handle the placement of replicas and fragments.
- *Bitcask backend has a limitation in size of tables but LevelDB backend has no such limitation, and can be used instead*. Bitcast keeps all keys in RAM and therefore can store a limited number of keys per node. Alternatively, LevelDB has not such limitation and can be used for databases with large number of keys.
- *Eventual consistency that consequently brings a good level of availability*. Riak supports tunable replication properties to determine the number of nodes on which data should be replicated and the number of nodes that are required to respond to a read or write request.
- *No single point of failure as the peer to peer (P2P) model is used*. Every node in a Riak cluster has the same responsibilities and there is no distinguished node or nodes

that take special roles or responsibility.

- *Scalable query execution approach that supports MapReduce*. Non-primary-key-based queries can be very computationally expensive in production clusters that are operating under load. Riak supports the MapReduce queries as a scalable approach to perform diverse operations on data, including querying, filtering, and aggregating information.

3.3.4 Cassandra

There are other distributed DBMSs which are not written in Erlang but Erlang applications can access them using a client library. Apache Cassandra was initially developed at Facebook and is a highly scalable and available database written in Java recommended for commodity hardware and cloud infrastructure [74]. Cassandra's distribution design is based the Amazon Dynamo design and its data model is based on Google's Bigtable. The Cassandra data model consists of a keyspace, column families, keys and columns. The keyspace contains application data, analogous to a database in relational model. For example, the keyspace *system* is used by Cassandra to store metadata about the local node, as well as the cluster. Cassandra Query Language (CQL) can be used to query the Cassandra database. CQLsh tool that ships with Cassandra can run CQL commands from the command line. Code 15 shows the CQL statements to create and select a keyspace.

Code 15: Creating and Selecting a Keyspace

```
create keyspace MyDataBase;
use MyDataBase;
```

The column family, is similar to a table in the relational database model, defines metadata about the columns that contains columns of related data. Code 16 creates a column family and updates the schema with adding three fields to the column family. It also adds an index for *field1*. Code 17 adds data for the key *key1* into the column family.

Code 16: Creating a Column Family

```
create column family MyColumnFamily with comparator = UTF8Type;
update column family MyColumnFamily with
  column_metadata =
  [
    {column_name: field1, validation_class: UTF8Type,
     index_type: KEYS},
    {column_name: field2, validation_class: UTF8Type},
    {column_name: field3, validation_class: UTF8Type}
  ];
```

Code 17: Adding Data for the Key *key1*

```
set MyColumnFamily['key1']['field1'] = 'value1';
set MyColumnFamily['key1']['field2'] = 'value2';
set MyColumnFamily['key1']['field3'] = 'value3';
```

Columns in Cassandra have a timestamp which records the last time the column was updated. This is not generated automatically, and client applications have to provide the timestamp when an update is performed. The timestamp is used for conflict resolution on the server side, and cannot be used in queries. When several updates are made concurrently, the most recent update that has the latest timestamp always wins.

Cassandra has a peer-to-peer distribution model, in which all nodes are structurally identical. The peer-to-peer design improves availability, because a node failure may affect the overall throughput, but it does not interrupt the service. Cassandra employs a gossip protocol for failure detection. Gossip is a peer-to-peer communication protocol used mostly in large scale distributed systems to spread information throughout the system. In Cassandra, the gossip process runs periodically and initializes a gossip session on a randomly chosen node. During a gossip round three messages are exchanged between the initiator node and a receiver node (a friend of the initiator), namely *GossipDigestSynMessage*, *GossipDigestAckMessage*, and *GossipDigestAck2Message* as shown in Figure 3.10. Firstly, the initiator node sends the *GossipDigestSynMessage* to the receiver node and waits to receive an acknowledgment (*GossipDigestAckMessage*). After receiving the acknowledgment, the initiator sends the receiver the *GossipDigestAck2Message* to complete the round of gossip. The gossip messages have a version that is used to identify stale information, so that during a gossip session, older information is overwritten with the most-up-to-date information of a particular node.

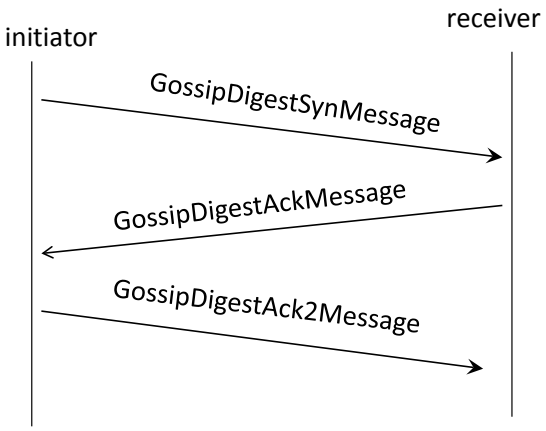


Figure 3.10: Gossip protocol: message exchanges between nodes

Cassandra uses a technique known as hinted handoff for dealing with node failure in a cluster. When a node which should receive a write is unavailable, Cassandra will write a hint to the coordinator node indicating that the write needs to be replayed to the unavailable node. When the coordinator node discovers from gossip that the failed node has recovered, the data will be sent corresponding to the hint to its target. The hinted handoff technique reduces the time required for a temporarily failed node to become consistent again. It also provides extreme write availability when strong consistency is not required.

Data fragmentation in Cassandra is handled by *partitioner* that is configurable through the configuration file or in the API. Cassandra provides three partitioning approaches, i.e. random partitioner (default), the order-preserving partitioner, and collating order-preserving partitioner. One also can create one's own partitioner by implementing the IPartitioner interface and placing it on Cassandra's class-path. The random partitioner is the default partitioner which uses an MD5 hash generator to determine where to place the keys. This approach spreads the keys evenly across the cluster as the distribution is random. In the order-preserving partitioner, rows are stored by key order. As the real-world data typically can not be partitioned evenly, it is likely eventually end up with an unbalanced cluster in which some nodes have lots of data and much less data on other nodes. However, this is a good approach when range queries are needed because keys are stored sequentially. Collating Order-Preserving Partitioner orders keys according to a United States English locale (EN_US). As this approach's usefulness is limited, it is rarely employed.

Reliability and fault tolerance in Cassandra are provided by replicating copies of data on multiple nodes. Cassandra offers an automatic, master-less and asynchronous replication. The number of copies that each piece of data will be stored and distributed across the cluster is referred to as the replication factor. If the replication factor is set to *N*, a written value is replicated on *N* nodes in the cluster. A replication strategy determines the nodes where replicas are placed. There are two replication strategies, i.e. *SimpleStrategy* and *NetworkTopologyStrategy*. The *SimpleStrategy* strategy can be used for a single data center. It places the first replica on a node determined by the partitioner and additional replicas are placed on the next nodes clockwise in the ring with no notion of data centers. The *NetworkTopologyStrategy* strategy is used when data is distributed across multiple data centers. This strategy considers the number of replicas in each data center. As we see from Code 18, the replica placement strategy and the replication factor are defined once a new keyspace is created.

Code 18: Defining the replica placement strategy for a new keyspace

```
CREATE KEYSPACE <ks_name>
    WITH strategy_class = 'NetworkTopologyStrategy'
    AND strategy_options:replication_factor = 4;
```

Large scale queries can be run on a Cassandra cluster by Hadoop MapReduce. Hadoop is an open source framework to process large amounts of data in parallel across clusters of commodity hardware. Hadoop's MapReduce and distributed filesystem (HDFS) subprojects are open source implementations of Google's MapReduce and Google File System (GFS)[86, 90]. There is built-in support for the Hadoop implementation of MapReduce in Cassandra. The *Jobtracker* process coordinates MapReduce jobs by accepting new jobs, breaks them into map and reduce tasks, and assigns those tasks to the *tasktrackers* processes in the cluster. The *Jobtracker* gets the required information about where data is stored from HDFS. Thus, *Jobtracker* is able to assign tasks near to the data they need. There is one *tasktrackers* process running on each Cassandra node which is responsible for running map or reduce tasks from the *Jobtracker*. This is an efficient way to retrieve data because each *tasktrackers* only receives queries for data that the local node is the primary replica.

Erlang applications employ the Thrift API to use Cassandra. There are also some client libraries for common programming languages such as Erlang, Python, Java, recommended to use instead of raw Thrift.

Suitability for Large Scale

Cassandra meets the general principles of scalable persistent storage as listed below:

- *Implicit placement of fragments and replicas*. Cassandra provides fragmentation and replication transparently, and so developers or administrators do not need to handle the placement of fragments and replicas explicitly.
- *ColumnFamily-based data model*. The ColumnFamily model provides an efficient way to query a distributed table by reducing the need for network communication. In this model, related columns are stored as a column-family, and so for generating a query on some specific columns, all the columns in a table are not needed to be traversed, and traversing just those specific columns will suffice.
- *Eventual consistency*. Cassandra offers tunable consistency in which the consistency level can be set on any read or write query. This allows application developers to tune consistency based on their requirements such as response time or data accuracy.
- *No single point of failures due to using a P2P model*. All Cassandra nodes are equal and have the same role and responsibility. This means there is no single point of failure because every node can service any request.
- *Scalable query execution approach by integrating Hadoop MapReduce*. There is built-in support for the Hadoop implementation of MapReduce in Cassandra that provides an efficient way to divide and run queries locally on primary replicas.

3.3.5 Discussion

The theoretical evaluation shows that Mnesia and CouchDB have some scalability limitations, i.e. implicit placement of replicas and fragments, single point of failures due to lack of P2P model (Sections 3.3.1 and 3.3.2). Dynamo-style NoSQL DBMS like Riak and Cassandra do have a potential to provide scalable storage for large distributed architectures as required by the RELEASE project (Sections 3.3.3 and 3.3.4). In the next section (Section 3.4) we investigate Riak scalability and availability in practice.

3.4 Riak Scalability and Availability in Practice

This section investigates the scalability and availability of the Riak NoSql DBMS practically. In the *scalability* benchmark, we measure the throughput of different cluster sizes of Riak nodes. In addition to scalability, we measure the *availability* and *elasticity* of Riak. In the availability benchmark, we examine how a cluster of Riak nodes copes with node failures. The elasticity benchmark investigates the ability of a running cluster of Riak nodes to scale up and down gracefully.

We use the popular Basho Bench benchmarking tool for NoSQL DBMS [91]. Basho Bench is an Erlang application that has a pluggable driver interface and can be extended to serve as a benchmarking tool for data stores.

3.4.1 Experiment Setup

Platform. The benchmarks are conducted on the Kalkyl cluster at Uppsala University [92]. The Kalkyl cluster consists of 348 nodes with 2784 64-bit processor cores connected via 4:1 oversubscribed DDR Infiniband fabric. Each node comprises Intel quad-core Xeon 5520 2.26 GHz processors with 8MB cache, and has 24GB RAM memory and 250 GB hard disk. The Kalkyl cluster runs Scientific Linux 6.0, a Red Hat Enterprise Linux. Riak data is stored on the local hard drive of each node.

Parameters. We use Riak version 1.1.1. The number of partitions, sometimes referred to as virtual nodes or vnodes, is 2048. In general, each Riak node hosts N_1 vnodes, i.e.

$$N_1 = \frac{N_{vnodes\ in\ the\ cluster}}{N_{nodes\ in\ the\ cluster}}$$

Riak documentation recommends 16-64 vnodes per node, e.g. 64-256 vnodes for a 4-node cluster. In the experiments we keep the default setting for replication, i.e. data is replicated

to three nodes on the cluster. The number of replicas that must respond to a read or write request is two, the default value.

3.4.2 How Does the Benchmark Work?

The experiments are conducted on a cluster where each node can be either a *traffic generator* or a *Riak node*. A traffic generator runs a copy of Basho Bench that generates and sends commands to Riak nodes. A Riak node contains a complete and independent copy of the Riak package which is identified by an IP address and a port number. Figure 3.11 shows how traffic generators and Riak nodes are organized inside a cluster. There is one traffic generator for every three Riak nodes in all our experiments.

Each Basho Bench application creates and runs 90 Erlang processes in parallel, i.e. workers in Figure 3.12. We picked this particular number of processes because it seems large enough to keep traffic generators busy. Then, every worker process randomly selects an IP address from a predefined list of Riak nodes. A worker process randomly selects one of the following database operation: `get`, `insert`, or `update`, and submits the corresponding HTTP or Protocol Buffer command to the selected IP address and port number. The default port

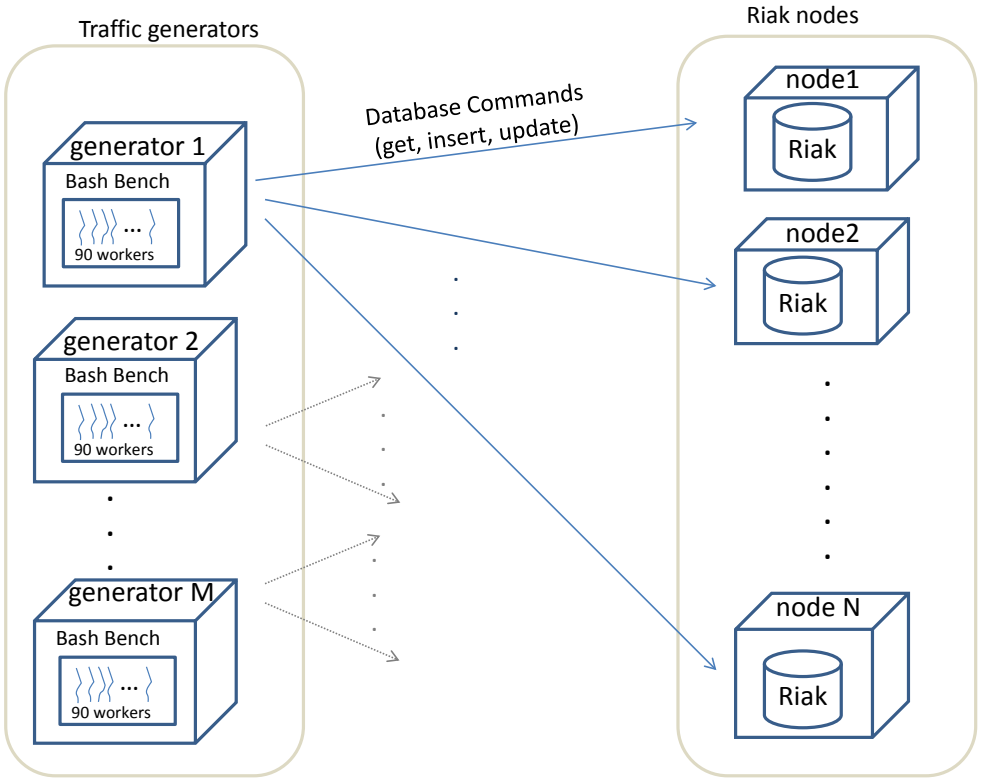


Figure 3.11: Riak Nodes and Traffic Generators

number for the HTTP communication is [8098], and for the protocol buffer the number is [8087]. A list of database operations and corresponding HTTP commands is as follows:

1. `Get` corresponds to the HTTP GET command
2. `Insert` corresponds to the HTTP POST command
3. `Update` corresponds to the HTTP PUT command

3.4.3 Scalability Measurements

We measure how throughput rises as the number of Riak nodes increases. We run our experiments on 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100-node clusters. In the experiments the traffic generators issue as many database commands as the cluster can serve. We repeat every experiment three times and the mean values are presented in all the figures. Figure 3.13 shows the scalability of Riak. As the figure shows, Riak scales linearly up to 60 nodes, but it does not scale beyond 60 nodes. In the figure, the green bars present the variation of different executions. Variation is very small for the clusters with fewer than 60 nodes, but significantly increases when the number of nodes grows beyond that.

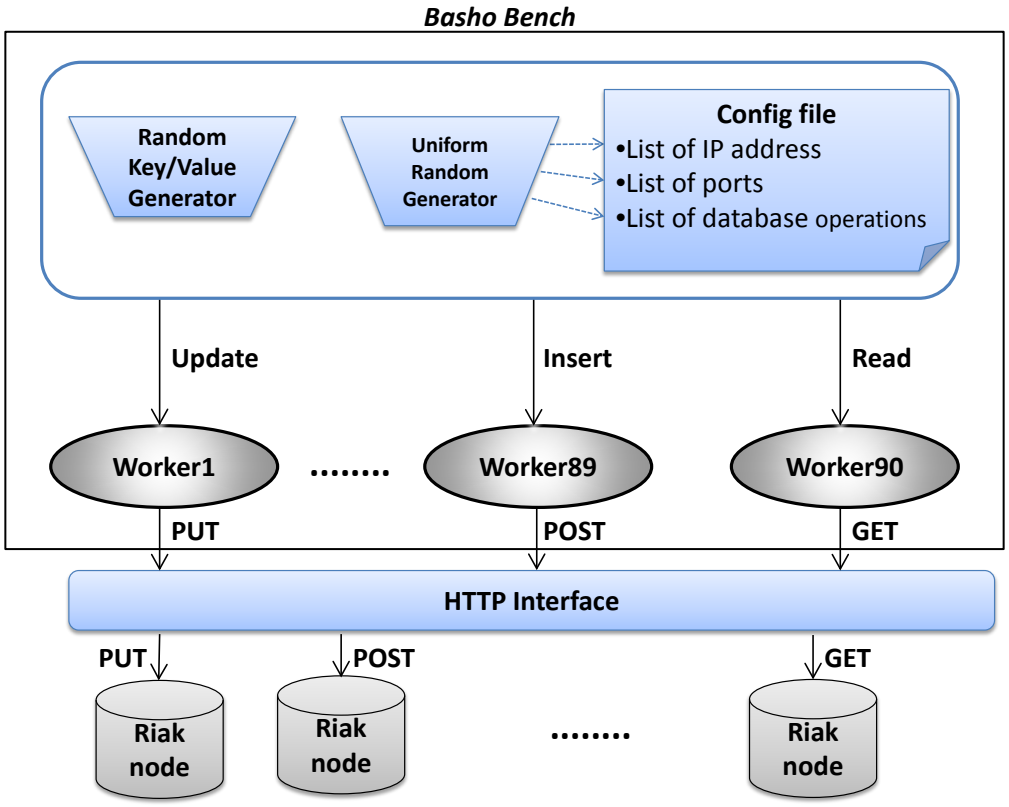


Figure 3.12: Basho Bench's Internal Workflow

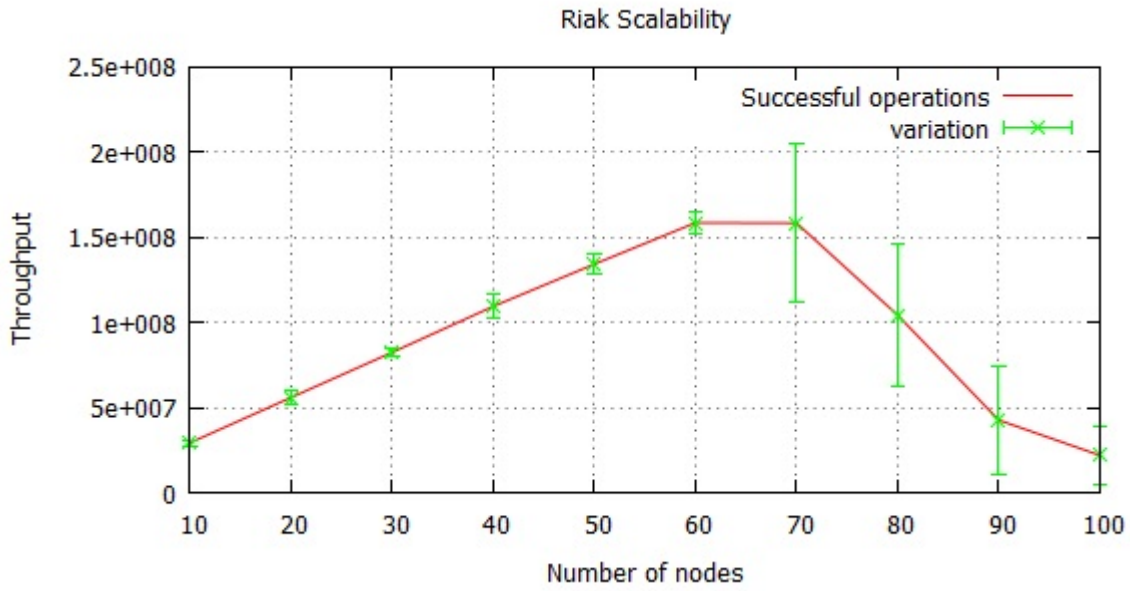


Figure 3.13: The Scalability of Riak NoSQL DBMS

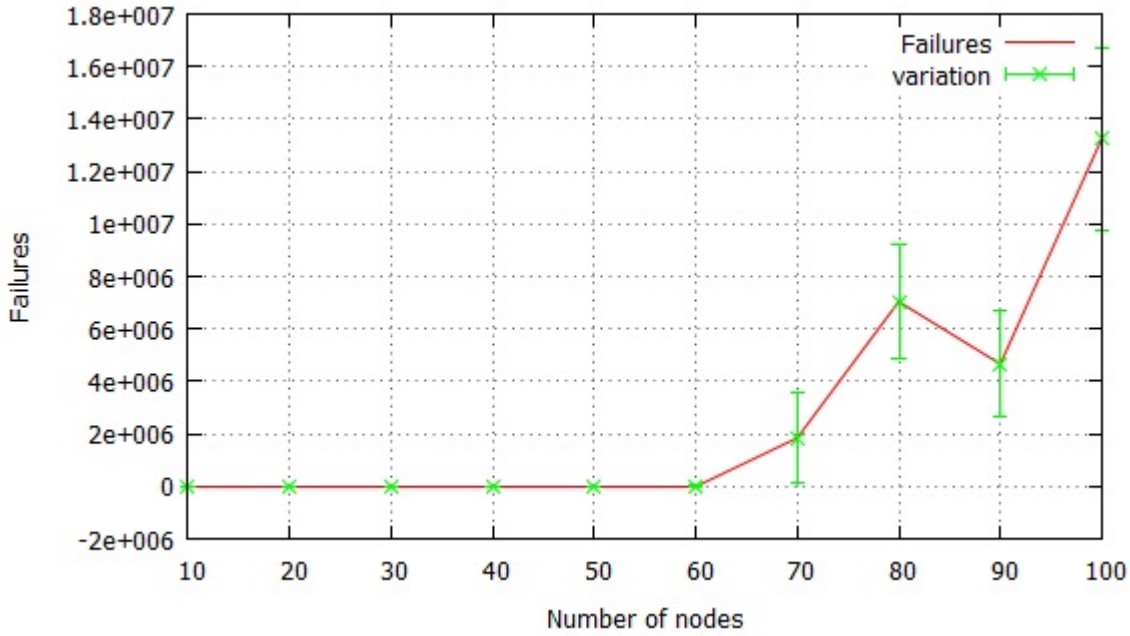


Figure 3.14: Failures in Riak Scalability Benchmark

Figure 3.14 shows the number of failed operations during the benchmark. For clusters with less than 60 nodes, we observe no failure. However, when the cluster size goes beyond 60 nodes, failures emerge. For example there are 2 million failures on a 70-node cluster, i.e. 1.3% of failures to the total number of operations. The log files show that the reason of the failures is timeout. Figure 3.15 shows the mean latency of successful operations. The mean latency of operations for a 60-node cluster is approximately 21 milliseconds which is much less than the timeout (60 seconds).

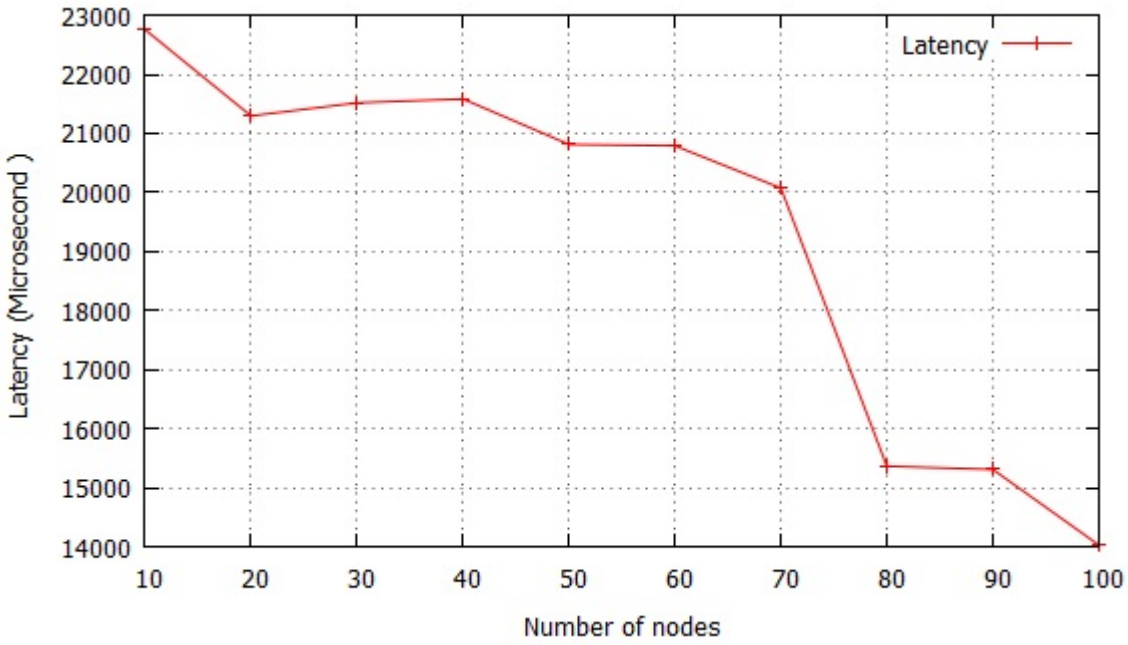


Figure 3.15: Command Latency vs. Number of Nodes

3.4.4 Resource Scalability Bottlenecks

To identify possible bottlenecks of the Riak scalability we measure the usage of random access memory (RAM), disk, cores, and network.

Figure 3.16 shows RAM utilisation on Riak nodes and traffic generators depending on the total number of nodes. The maximum memory usage is 720MB out of 24GB of total RAM memory, i.e. less than 3%. We observe that RAM is not a bottleneck for Riak scalability.

Figure 3.17 shows percentage of time that a disk spends serving requests. The maximum usage of disk is approximately 10% on a 10-node cluster. Disk usage for traffic generators is approximately 0.5%. We see that disk is not a bottleneck for Riak scalability either.

Figure 3.18 shows the core utilisation on 8-core Riak nodes and traffic generators. The maximum core utlisation is approximately 550%, so 5.5 of the 8 cores are used.

When profiling the network we count the number of the following packets: *sent*, *received*, and *retransmitted*. Figure. 3.19 and 3.20 show the results for traffic generators and Riak nodes respectively. As we see from the figures, the growth in the number of sent and received packets is consistent with the rise in the throughput. The number of sent and received packets increases linearly up to 60 nodes and beyond that there is a significant degradation. To check whether there is a TCP incast [93, 94], we count the number of retransmitted packets. In general, TCP incast occurs when a number of storage servers send a huge amount of data to a client, and Ethernet switch is not able to buffer the packets. When TCP incast strikes the number of lost packets increases and consequently causes a growth in the number of

retransmitted packets. However, counting of the retransmitted packets shows that TCP incast has not occurred during the benchmark. The maximum number of retransmitted packets is 200 packets which is negligible.

A comparison of network traffic between traffic generators and Riak nodes shows that Riak nodes produce five times more network traffic than traffic generators. For example on a 10-node cluster traffic generators send 100 million packets, whereas Riak nodes send 500

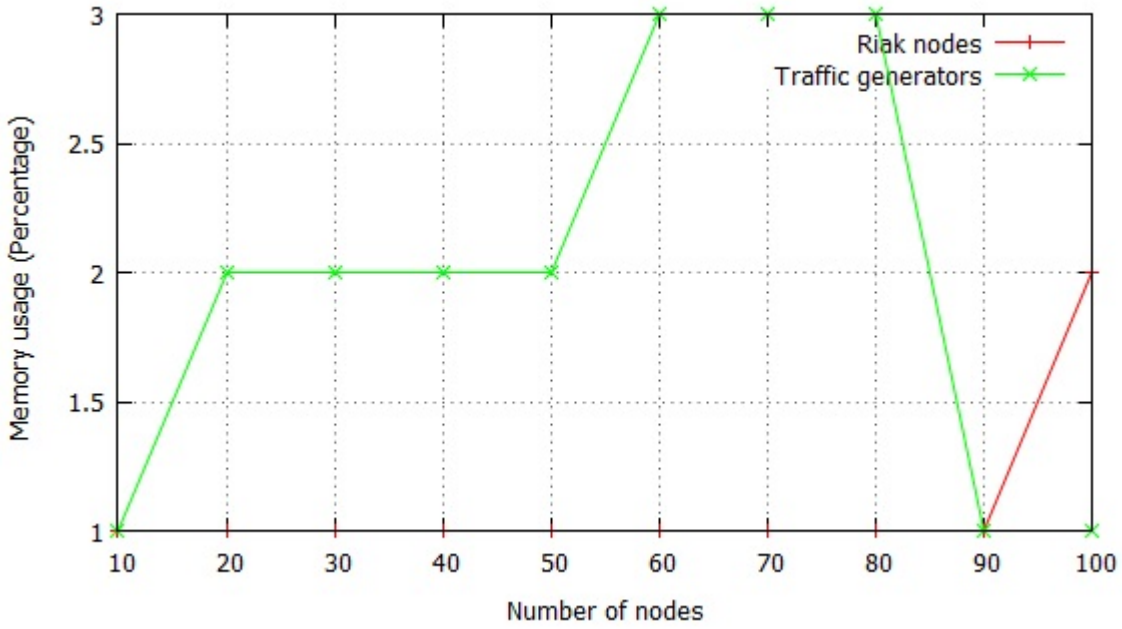


Figure 3.16: RAM usage in Riak Scalability Benchmark

Figure 3.17: Disk usage in Riak Scalability Benchmark

million packets on the cluster of the same size. The reason is due to the fact that to replicate and maintain data Riak nodes in addition to communicating with generators also communicate between each other.

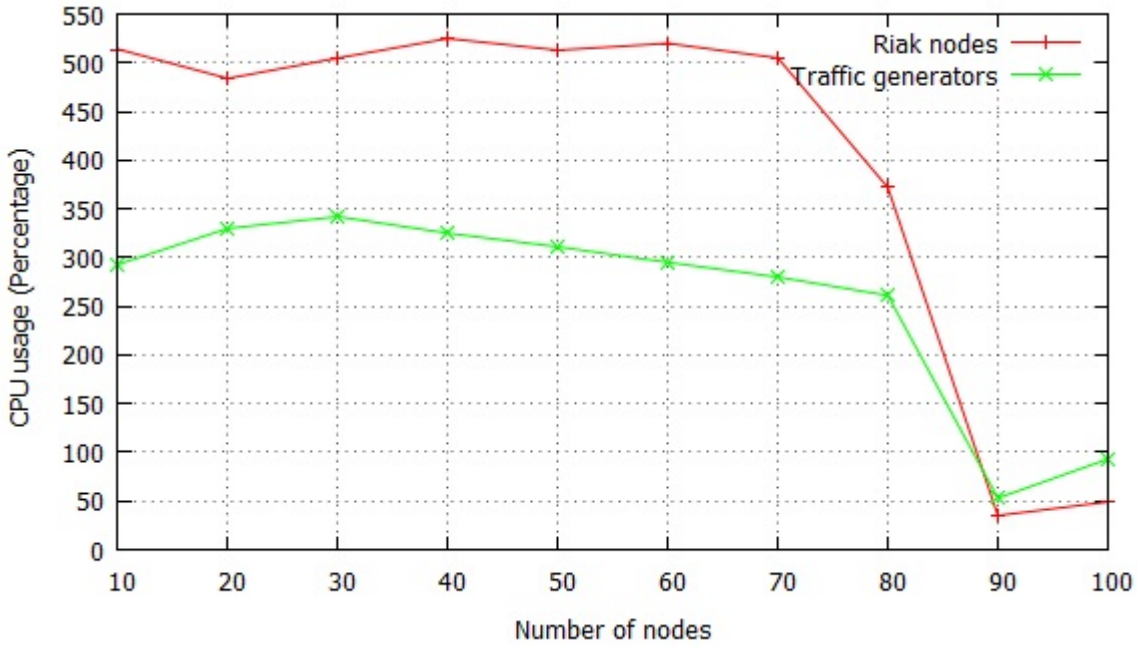


Figure 3.18: Core Usage in Riak Scalability Benchmark

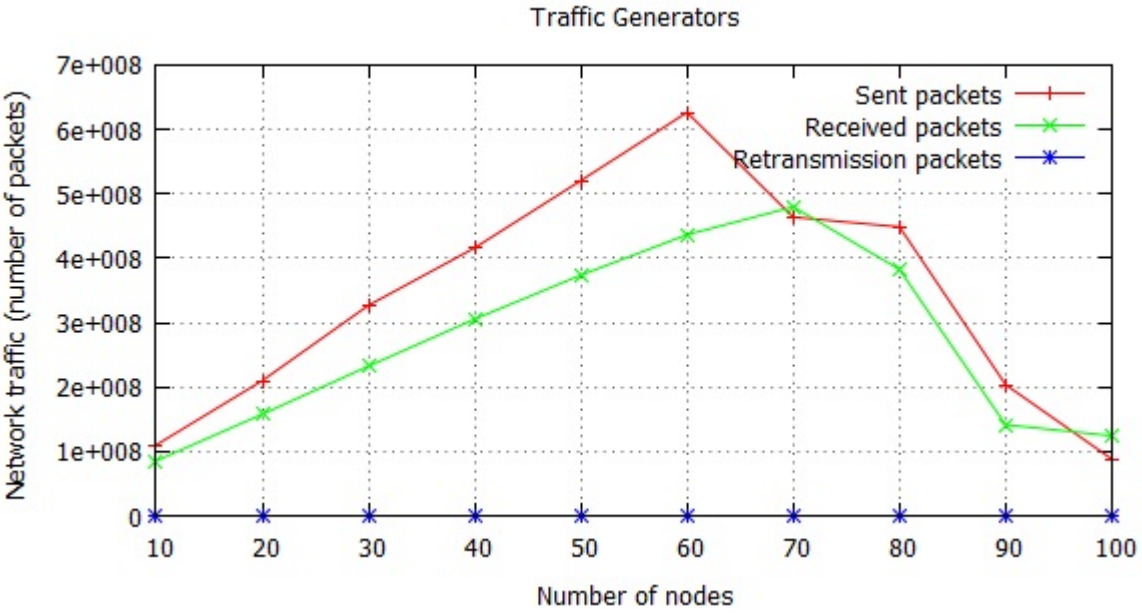


Figure 3.19: Network Traffic of Traffic Generators

3.4.5 Riak Software Scalability

The profiling above reveals that Riak scalability is not bound by resources like memory or disk, so we need to investigate the scalability of Riak software.

In the next chapter and [9], we show that global operations, i.e. operations that engage all Erlang VMs in a cluster, severely limit scalability. While it is not feasible to search the entire Riak codebase for global operations in the form of iterated P2P operations, we investigated two likely sources of global operations.

1. We instrument the global name registration module `global.erl` to identify the number of calls, and the time consumed by each global operation. The result shows that Riak makes no `global.erl` calls.

2. We also instrument the generic server module `gen_server.erl`. Of the 15 most time-consuming operations, only the time of `rpc:call` grows with cluster size. Moreover, of the five Riak *rpc* calls, only `start_put_fsm` function from module `riak_kv_put_fsm_sup` grows with cluster size (Table 3.1). T_{mean} shows the mean time that each function call takes to be completed in microseconds and N_{mean} is the mean number of times that a function is called during 5 minutes benchmarking in 3 executions. Profiling details are available in Appendix A.3.

Independently, Basho [95] have analysed Riak scalability and had identified the `riak_kv_get/put_fsm_sup` issue, together with a scalability issue with statistics reporting. To improve the Riak scalability Basho applied a number of techniques and introduced

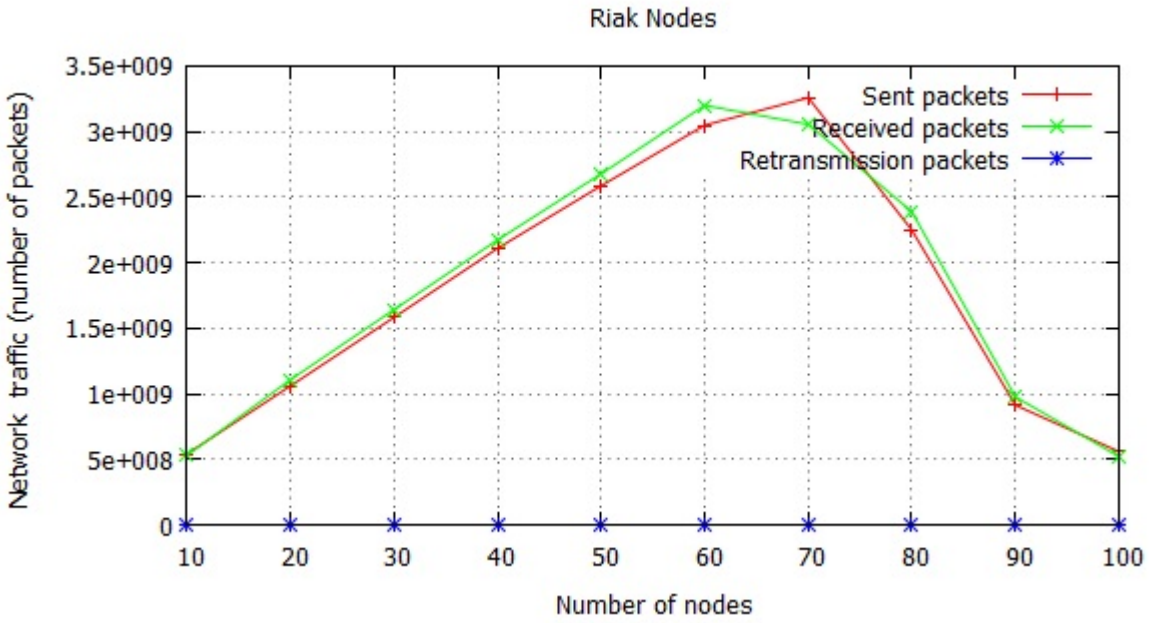


Figure 3.20: Network Traffic of Riak Nodes

Module	Function		Runtime on a cluster of			
			4 nodes	8 nodes	16 nodes	24 nodes
rpc	call	T_{mean}	1953	2228	8547	15180
		N_{mean}	106591	77468	49088	31104
riak_kv_put_fsm_sup	start_put_fsm	T_{mean}	2019	2284	8342	13461
		N_{mean}	104973	82070	48689	33898

Table 3.1: Two the most time-consuming Riak functions

new library `sidejob` [96]. These modifications are available in Riak version 1.3 and upcoming version 1.4. An overview of the modifications is presented below.

In Riak version 1.0.x through 1.2.x creating get/put FSM (Finite State Machine) processes go through two supervisor processes, i.e. `riak_kv_get/put_fsm_sup`. The supervisors are implemented as single-process `gen_servers` and become a bottleneck under heavy load, exhibiting build up in message queue length. In Riak version 1.3 get/put FSM processes are created directly on the external API-handling processes that issue the requests, i.e. `riak_kv_pb_object` (protocol buffers interface) or `riak_kv_wm_object` (REST interface). To track statistics and unexpected process exits without supervisors the get/put FSM processes register themselves asynchronously with a new monitoring process `riak_kv_getput_mon`. Similarly, to avoid another single process bottleneck, Basho replaced *rpc* calls in the put FSM implementation that forwards data to responsible nodes with direct `proc_lib:spawn`. This avoids another single-process bottleneck through the `rex` server used by `rpc:call`. Riak version 1.3 also has some refactoring to clean up unnecessary old code paths and do less work, e.g. the original map/reduce cache mechanism that was replaced by `riak_pipe` was still having cache entries ejected on every update.

In Riak version 1.4 the get/put FSM spawn mechanism was replaced by a new mechanism presented in the sidejob library. The library introduces a parallel mechanism for spawning processes and enforces an upper limit on the number of active get/put FSMs to avoid process table exhaustion when the node is heavily loaded. Exhaustion of the process table has caused a cascade cluster failure in a production deployments. The library was also used to eradicate the bottleneck caused by the statistics reporting process.

3.4.6 Availability and Elasticity

Distributed database systems must maintain availability despite network and node failures. Another important feature of distributed systems is elasticity. Elasticity means an equal and dynamic distribution of the load between nodes when resources (i.e. Riak nodes) are either removed or added to the system [97]. To benchmark Riak availability and elasticity we run seven generators and twenty Riak nodes. During the benchmark the number of generators

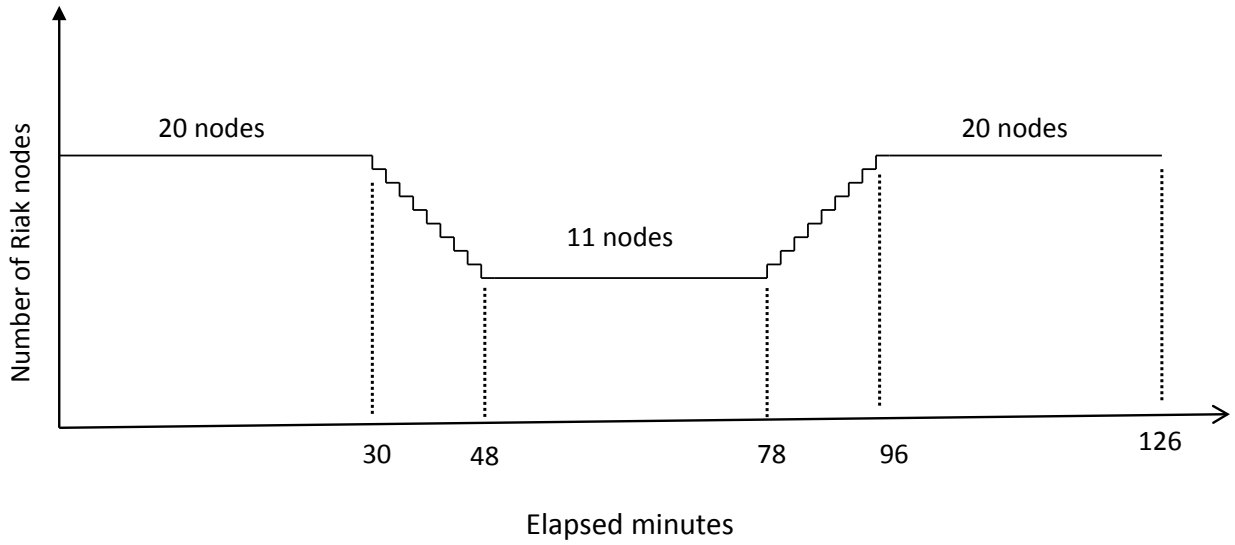


Figure 3.21: Availability and Elasticity Time-line

remains constant (seven) but the number of Riak nodes changes. Figure 3.21 shows that during the first 30 minutes of the benchmark there are 20 Riak nodes. We choose 30 minutes because we want to be sure that the system is in a stable state. After 30 minutes nodes go down every two minutes. In total 9 nodes go down until minute 48, i.e. approximately 50% failures. Between minutes 48 and 78 the system has 11 Riak nodes. After minute 78 nodes come back every two minutes. Thus, after minute 96 all 20 nodes are back. After minute 96 the benchmark runs on 20 Riak nodes for another 30 minutes.

Figure 3.22 shows that when the cluster loses 50% of its Riak nodes (between minutes 30 and 48) Riak throughput decreases and the number of failures grows. However, in the worst case the number of failures is 37 whereas the number of successful operations is 3.41 million. Between minutes 48 and 78 the throughput does not change dramatically, and during and after adding new nodes the throughput grows. Thus, we conclude that Riak has a very good level of availability and elasticity.

3.4.7 Summary

Riak version 1.1.1 scales up to approximately 60 nodes linearly on the Kalkyl cluster (Section 3.4.3). But beyond 60 nodes throughput does not scale and timeout errors emerge (Figure 3.13 and 3.14). To identify the Riak scalability problem we profiled RAM, disk, cores, and network (Section 3.4.4). The results of RAM and disk profiling show that these cannot be a bottleneck for Riak scalability, maximum RAM usage is 3%, and the maximum disc usage is 10%. Maximum core usage on 8 core nodes is 5.5 cores, so cores are available. The

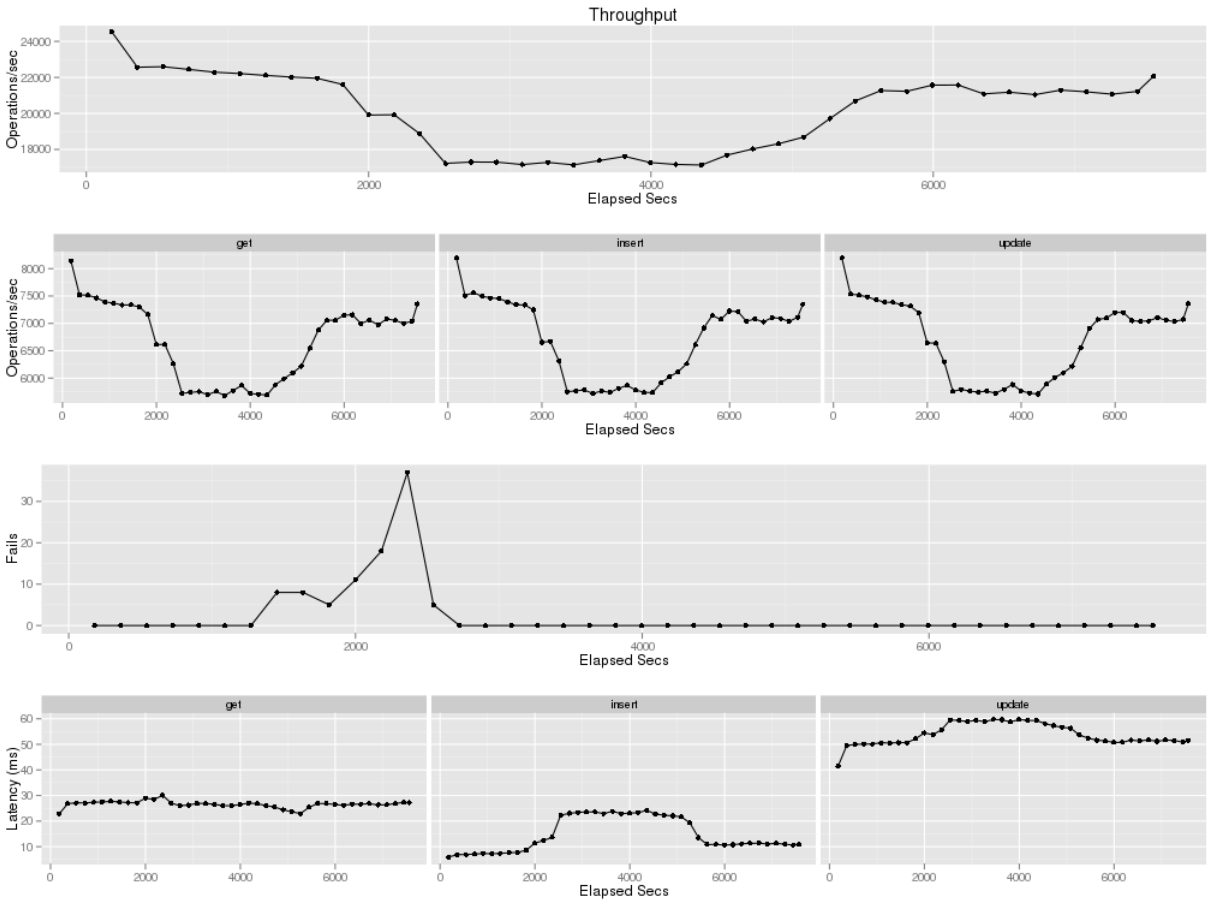


Figure 3.22: Throughput and Latency in the Availability and Elasticity Benchmark

network profiling shows that the number of retransmitted packets is negligible in comparison with the total number of successfully transmitted packets, i.e. 200 packets out of $5 \cdot 10^8$ packets.

Our observations of Riak, together with correspondance with the Basho developers reveal that the scalability limits are due to single supervisor processes such as `riak_kv_get/put_fsm_sup` processes and `rex` process in `rpc.erl` (Section 3.4.5). Riak shows very good availability and elasticity (Section 3.4.6). After losing 9 nodes out of 20 Riak nodes, only 37 failures occurred, whereas the number of successful operations was 3.41 million. When failed nodes come back up the throughput grows gracefully.

3.5 Discussion

We have rehearsed the requirements for scalable and available persistent storage (Section 3.2), and evaluated four popular Erlang DBMS against these requirements. We conclude that Mnesia and CouchDB are not suitable persistent storage at our target scale, but Dynamo-style NoSQL DBMS like Cassandra and Riak have the potential to be (Section 3.3).

We have investigated the current scalability limits of the Riak version 1.1.1 NoSQL DBMS using Basho Bench on 100-node cluster with 800 cores. We establish for the first time scientifically the scalability limit of Riak 1.1.1 as 60 nodes on the Kalkyl cluster, thereby confirming developer folklore.

We show that resources like memory, disk, and network do not limit the scalability of Riak (Section 3.4.5). By instrumenting the `global` and `gen_server` OTP libraries we identify a specific Riak remote procedure call that fails to scale. We outline how later releases of Riak are refactored to eliminate the scalability bottlenecks (Section 3.4).

We conclude that Dynamo-style NoSQL DBMSs provide scalable and available persistent storage for Erlang. The RELEASE target architecture requires scalability onto 100 hosts, and we are confident that Dynamo-like NoSQL DBMSs will provide it. Specifically the Cassandra interface is available and Riak 1.1.1 already provides scalable and available persistent storage on 60 nodes.

Chapter 4

Investigating the Scalability Limits of Distributed Erlang

This chapter introduces DE-Bench, a scalable fault-tolerant peer-to-peer benchmarking tool for distributed Erlang. We employ DE-Bench to investigate the scalability limits of distributed Erlang up to 150 nodes and 1200 cores.

Recall that distributed Erlang is discussed in Section 2.4.3.2. A brief introduction to the chapter is given in Section 4.1. The design, implementation, and deployment of DE-Bench are presented in Section 4.2. The investigation results using DE-Bench at the Kalkyl cluster are discussed in Section 4.3. Finally, a summary of the chapter is given in Section 4.4. The work in this chapter is reported in [8, 9, 10, 2].

4.1 Introduction

One of the main aims of the RELEASE project is improving the scalability of distributed Erlang. To improve the scalability of distributed Erlang, identifying the scalability bottlenecks is a key requirement before proceeding to next steps. For this purpose, we needed a benchmarking tool to perform the necessary measurements on a large-scale architecture with hundreds of nodes and thousands of cores.

Since there was not such a tool to fulfil our requirement, we have designed and implemented *DE-Bench* [9, 98]. DE-Bench, which stands for ”Distributed Erlang benchmark”, is a fault-tolerant parameterized peer-to-peer benchmarking tool that measures the throughput and latency of distributed Erlang commands on a cluster of Erlang nodes. We employ DE-Bench to investigate the scalability limits of distributed Erlang up to 150 nodes and 1200 core. We measure different aspects of distributed Erlang, i.e. global commands, point-to-point

commands, and server processes. We provide the latency of each command individually for different cluster sizes.

4.2 How Does DE-Bench Work?

4.2.1 Platform

The benchmarks are conducted on the Kalkyl cluster at Uppsala University (the platform specification is given in Section 3.4.1) [92]. To avoid confusion with Erlang nodes (Erlang VM), we use the term *host* to refer to the Kalkyl nodes (physical machines). Erlang version R16B has been used in all our experiments.

4.2.2 Hosts and Nodes Organization

The same as an ordinary distributed Erlang application, our benchmark consists of a number of Erlang Virtual Machines (Erlang VMs) communicating with each other over a network. The benchmark is run on a cluster of hosts and there can be multiple Erlang VMs on each host, however, each Erlang VM runs only one instance of DE-Bench. For example, Figure 4.1 depicts a cluster with 2 hosts and 2 Erlang nodes (Erlang VMs) per host. As shown, a node can communicate with all the other nodes in the cluster regardless of whether nodes are located on the same host or not. DE-Bench follows a peer-to-peer model in which all nodes perform the same role independently, and so there is no specific node for coordination or synchronisation. The peer-to-peer design of DE-Bench improves scalability and reliability by eliminating central coordination and single points of failure. The peer-to-peer model and its advantages are discussed in Section 2.3.2.

4.2.3 The Design and Implementation of DE-Bench

To evaluate the scalability of distributed Erlang, we measure how adding more nodes to a cluster of Erlang nodes increases the throughput. By *throughput* we mean the total number of successfully executed distributed Erlang commands per experiment. DE-Bench is based on Basho Bench, an open source benchmarking tool for Riak NoSQL DBMS [91].

One interesting feature of Erlang is its support for failure recovery. Processes in an Erlang application can be organised into a hierarchical structure in which parent processes monitor failures of their children and are responsible for their restart [99]. DE-Bench uses this feature to provide a fault-tolerant service. Figure 4.2 represents the internal workflow of DE-Bench. Initially, a supervisor process runs a number of worker processes in parallel on a

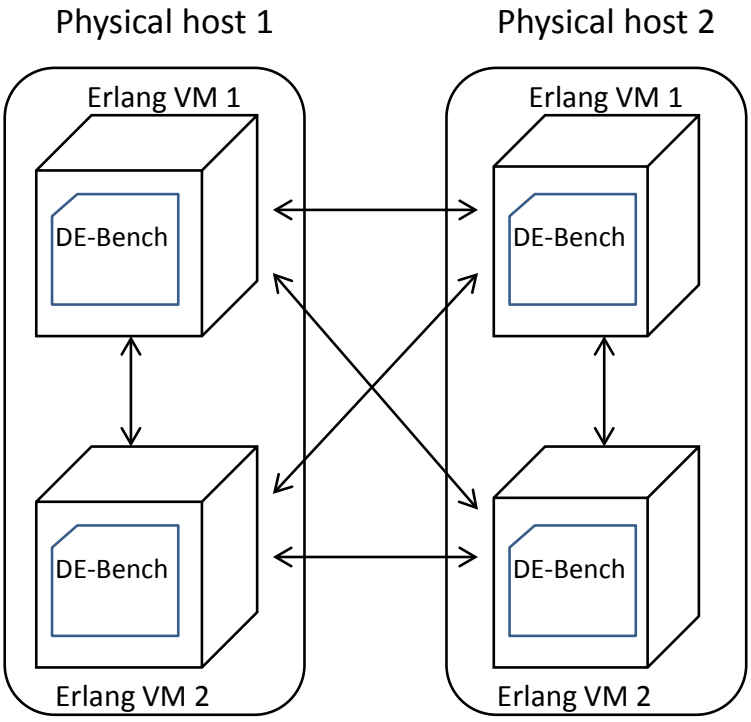


Figure 4.1: 2 hosts and 2 Erlang VMs per host

node. The number of worker processes on each node is definable through a configuration file (Appendix B.1). As each host in the Kalkyl cluster has 8 cores, we run 40 worker processes on each node to exploit available cores. The supervisor process supervises all worker processes and keeps them alive by restarting them in case of failure. A worker process randomly selects an Erlang node and a distributed Erlang command from the configuration file and runs that command on the selected node. There are three kinds of commands in DE-Bench:

- *Point-to-Point*: In point-to-point commands, a function with tunable argument size and computation time is run on a remote node. Figure 4.3 depicts the argument size and computation time for a point-to-point command. As the figure shows, firstly, a function with argument size X bytes is called. Then, a tail recursive function is run on the target node for Y microseconds. Finally, the argument is returned to the source node as result. The pseudo-code for a point-to-point command is shown in Code 19.

  Point-to-point commands include *spawn*, *rpc*, and synchronous calls to server processes, i.e. *gen_server* or *gen_fsm*.

- *Global commands*: When a global command is run, all the nodes in a cluster are involved, and the result is returned only once the command completes successfully on all nodes. Global commands such as *global:register_name* and *global:unregister_name* are defined in the OTP *global* module.

Code 19: The pseudo-code for a point-to-point command in DE-Bench

```
remote_function(Sender, Packet) when is_pid(Sender) ->
    Received_Size=byte_size(Packet),
    Expected_Size=de_bench_config:get(packet_size, ?DEFAULT_SIZE),
    Computation_Time=de_bench_config:get(computation_time, 0),
    Equal=Received_Size=:=Expected_Size,
    case Equal of
        true ->
            de_helper:do_computation(Computation_Time),
            Sender ! {result, Packet};
        false -> Sender ! {no_proper_size, Packet}
    end.
```

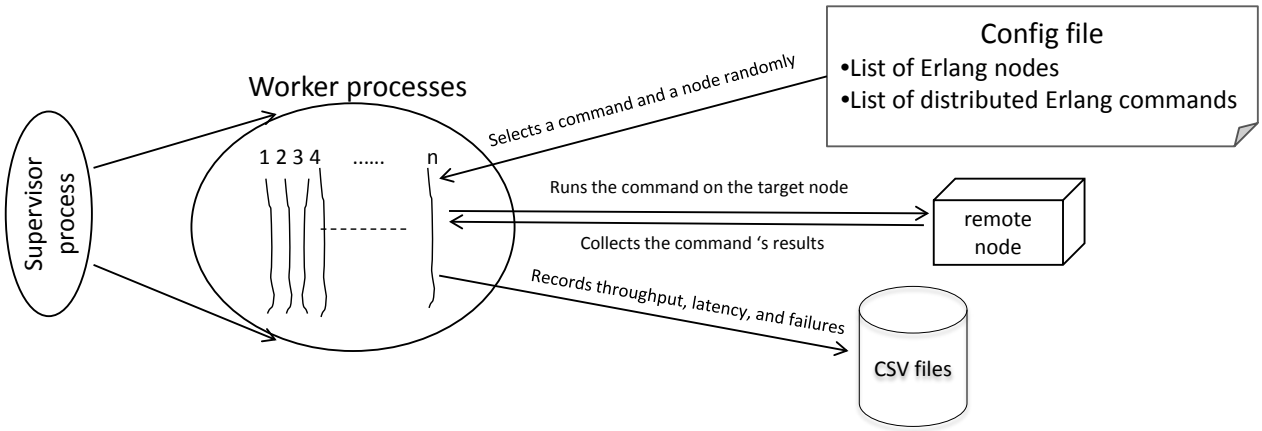


Figure 4.2: DE-Bench's Internal Workflow

- *Local commands*: In local commands such as *register_name*, *unregister_name* and *whereis_name*, only the local node executes the command and there is no need to communicate with other nodes in the cluster. The command *whereis_name* is a look up in the local name table regardless of whether it is from the *global* module or not.

After running a command, the latency and throughput of that command is measured and recorded in appropriate CSV files (a sample CSV file is given in Appendix B.2). The CSV files are stored on the local disk of each node to avoid disk access contention and network communication latency.

DE-Bench is extensible and one can easily add new commands into DE-Bench through the *de_commands.erl* module. At the time of writing this thesis, the following commands are defined and measurable in DE-Bench:

1. point-to-point commands:

   (a) *spawn(Node, Fun)*: a function is called at a remote node with tunable argument

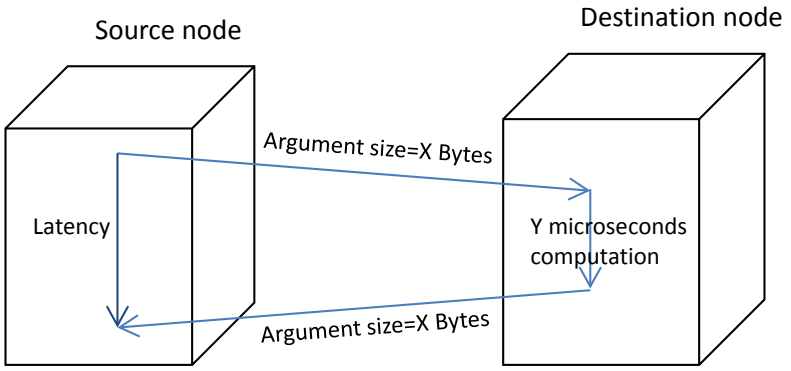


Figure 4.3: Argument size (X) and computation time (Y) in a point-to-point command

size and computation time as depicted in Figure 4.3. Since spawn is an asynchronous call, the elapsed time is recorded after receiving an acknowledgment from the remote node.

(b) rpc*(Node, Fun)*: synchronously calls a function at a remote node with tunable argument size and computation time.

(c) *server process call*: makes a synchronous call to a generic server process (gen_server) or a finite state machine process (gen_fsm) by sending a request and waiting for the reply.

2. Global commands:

   (a) *global:register_name(Name, Pid)*: globally associates a name with a pid. The registered names are stored in name tables on every node in the cluster.

   (b) *global:unregister_name(Name)*: removes a globally registered name from all nodes in the cluster.

3. Local commands:

   (a) *register_name(Name, Pid)*: associates a name with a process identifier (pid).

   (b) *unregister_name(Name)*: removes a registered name, associated with a pid.

   (c) *whereis(Name)*: returns the pid registered with a specific name.

   (d) *global:whereis(Name)*: returns the pid associated with a specific name globally. Although, this command belongs to *global* module, it falls in local commands because it does a lookup in the local name table.

These commands are not used with the same frequency in a typical distributed Erlang application. For instance, point-to-point commands such as *spawn* and *rpc* are the most commonly

used ones and global commands like *register_name* and *unregister_name* are used much less than the others. Thus, to generate more realistic results, we can use each command with a different ratio.

In Erlang, a process identifier (pid) can only be registered once, otherwise an exception is thrown. To prevent the exception, three internal states are defined for a worker process to ensure that after registering a process, all necessary commands like *whereis_name* and *unregister_name* will be executed afterward. Figure 4.4 shows the states that a worker process follows to avoid duplicate registration exception. As shown, point-to-point commands do not change the current state (*state1*). The commands *whereis_name(Name)* and *unregister_name(Name)* are ignored unless they come after *register_name(Name, Pid)*. After running a *register_name(Name, Pid)* command, both *whereis_name(Name)* and *unregister_name(Name)* will be run respectively. To avoid name clashes, a timestamp function is used to generate globally unique names for processes in a cluster.

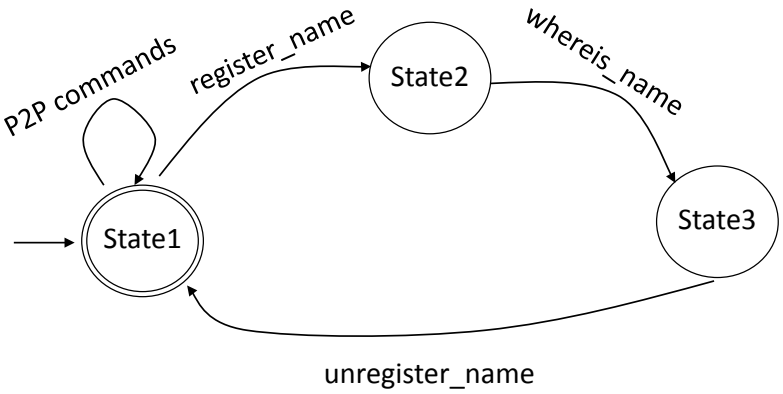


Figure 4.4: Worker Process' Internal States

4.3 Benchmarking Results

In this section, we employ DE-Bench to measure the scalability of distributed Erlang from different perspectives. In the scalability benchmark, we measure the throughput for different sizes of Erlang clusters and observe how adding more Erlang nodes to a cluster affects the throughput.

The benchmark is conducted on 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100-node clusters and measures the throughput by counting successful operations over the duration of an experiment. The duration of an experiment is specified in the configuration file. All the experiments in this section run for 5 minutes. There is one Erlang VM on each host and as always one DE-Bench instance on each VM. After the end of each experiment, the generated CSV files

from all participating nodes are collected and aggregated to find out the total throughput and failures. For example, for benchmarking a 70-node cluster, 70 instances of DE-Bench are run simultaneously and consequently they will generate 70 CSV files which need to be aggregated to find out the total throughput of the 70-node cluster (a sample CSV file is given in Appendix B.2). To provide reliable results, all experiments are run three times and the median values are represented in diagrams.

We will measure following aspects of the scalability of distributed Erlang:

1. **Global Commands**: As illustrated previously in Section 4.2.3, in global commands all nodes in the cluster are involved. This feature of global commands could make them a bottleneck for scalability. To find out the effects of global commands on the scalability of distributed Erlang, we run the measurements with different percentages of global commands.

2. **Data Size**: As shown in Figure 4.3, the argument size of point-to-point commands is tunable. To understand the effect of data size on the scalability and performance of an Erlang cluster, we run the benchmark with different argument sizes.

3. **Computation Time**: As with argument size, the computation time of point-to-point commands is also tunable in DE-Bench (Figure 4.3). We investigate the effect of computation time on both scalability and performance of distributed Erlang.

4. **Data Size & Computation Time**:

   There is a common belief that distributed Erlang does not scale in a large distributed environment with hundreds of nodes and thousands of cores. To assess this belief, we measure how distributed Erlang scales up to 150 nodes and 1200 cores with relatively heavy data and computation loads.

5. **Server Process**: There are two popular types of server process in Erlang/OTP: generic server processes (*gen_server*) and finite state machine processes (*gen_fsm*). This section will inspect the scalability of these server processes, and try to find out whether the server processes are bottlenecks for the scalability of distributed Erlang.

4.3.1 Global Commands

This section investigates the effect of global commands on the scalability of distributed Erlang. We run the benchmark with different frequencies of global commands. The following commands are used in the measurement:

1. point-to-point commands: *spawn* and *rpc* with 10 bytes argument size and 10 microseconds computation time

2. Global commands: *global:register_name* and *global:unregister_name*

3. Local commands: *global:whereis(Name)*

Figure 4.5 shows how frequency of global commands limits the scalability of distributed Erlang. As we see from the diagram, scalability becomes more limited as more global commands are used. For example, when 0.01% of global commands are used (the dark blue curve) , i.e. 1 global command per 10000 point-to-point commands, distributed Erlang doesn't scale beyond ≈60 nodes.

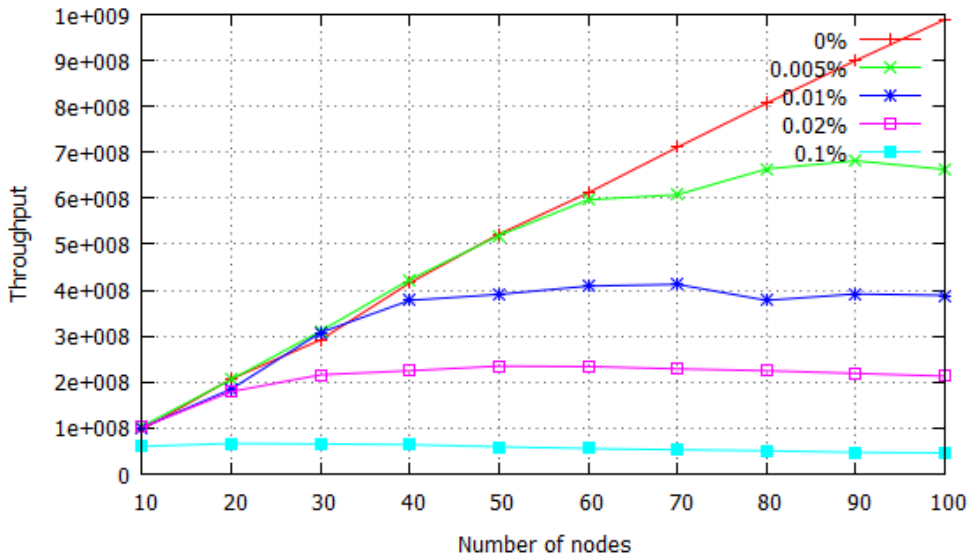


Figure 4.5: Scalability vs. Percentage of Global Commands

Figure 4.6 presents the latency of the commands used in this measurement. The diagram reveals that the latency of both global commands, i.e. *global:register_name* and *global:unregister_name*, increases dramatically when cluster size grows. For example, we see from the figure that a global name registration on a 100-node cluster takes ≈20 seconds which is a considerably long time. The registered names are stored in name tables on every node and these tables are strongly consistent, which means that an update is considered complete only when all nodes have acknowledged [100]. This lock mechanism for updating the replicated information becomes a bottleneck on large clusters. As we see from Figure 4.6, the other commands' latencies (i.e. *spawn*, *rpc*, and *whereis*) are very low and negligible in comparison with the global ones.

4.3.2 Data Size

This section investigates the effect of data size on the scalability of distributed Erlang. As shown in Figure 4.3, point-to-point commands have two configurable parameters, i.e computation time and argument size. In this benchmark, the computation time is constant while the

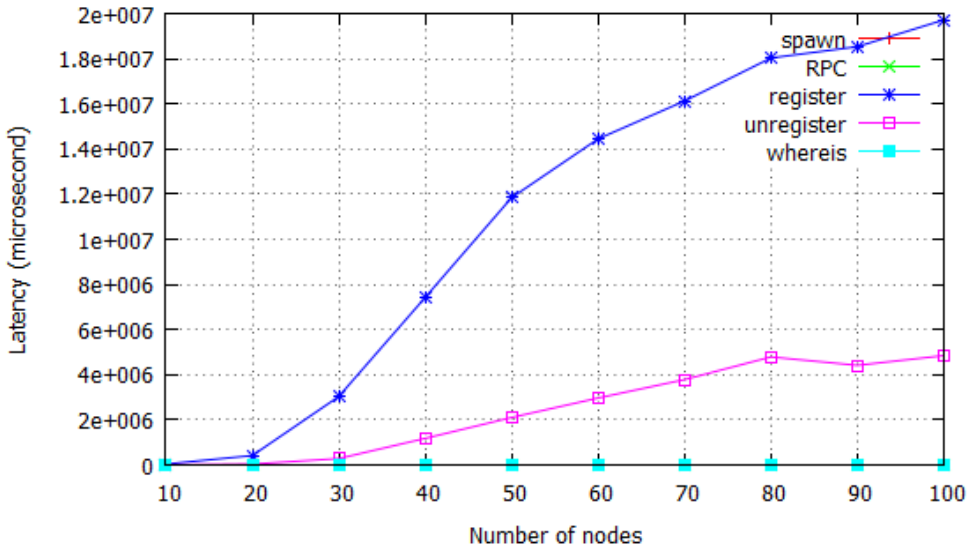


Figure 4.6: Latency of Commands

argument size of point-to-point commands change for different experiments. The following commands are used in this benchmark:

- point-to-point commands, i.e. *spawn* and *rpc*, with 10, 100, 1000, and 10000 bytes argument size and 10 microseconds computation time

Figure 4.7 shows the scalability of distributed Erlang for different data sizes. As the argument size increases, the performance and scalability decrease. For example the best scalability belongs to 10 bytes data size (the red curve) and the worst scalability belongs to 10K bytes data size (the pink curve).

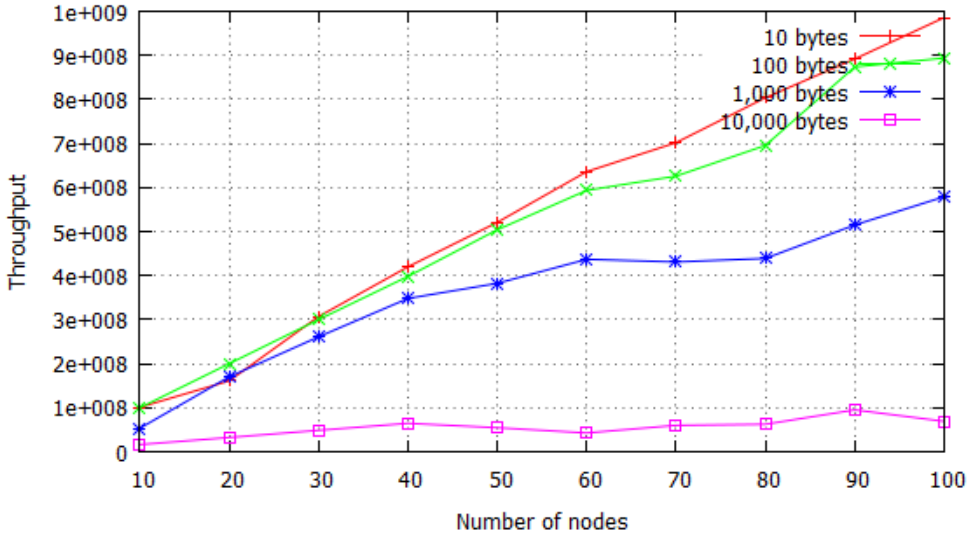


Figure 4.7: Scalability vs. Data Size

4.3.3 Computation Time

This section investigates the effects of computation time on the scalability of distributed Erlang. In this benchmark, the argument size of point-to-point commands is constant while the computation time changes for different experiments. The following commands are used in the benchmark:

- point-to-point commands, i.e. *spawn* and *rpc*, with 10 bytes argument size and 10, 1000, and 1000000 microseconds computation time.

Figure 4.8 presents the scalability of distributed Erlang for different computation times. As we increase the computation time, performance and scalability degrade. The best scalability achieved for 10 microseconds computation time, and 1 second computation time shows the worst scalability. This is expected because for larger computation time, worker processes have to wait longer for their response and consequently spend most of the time idle.

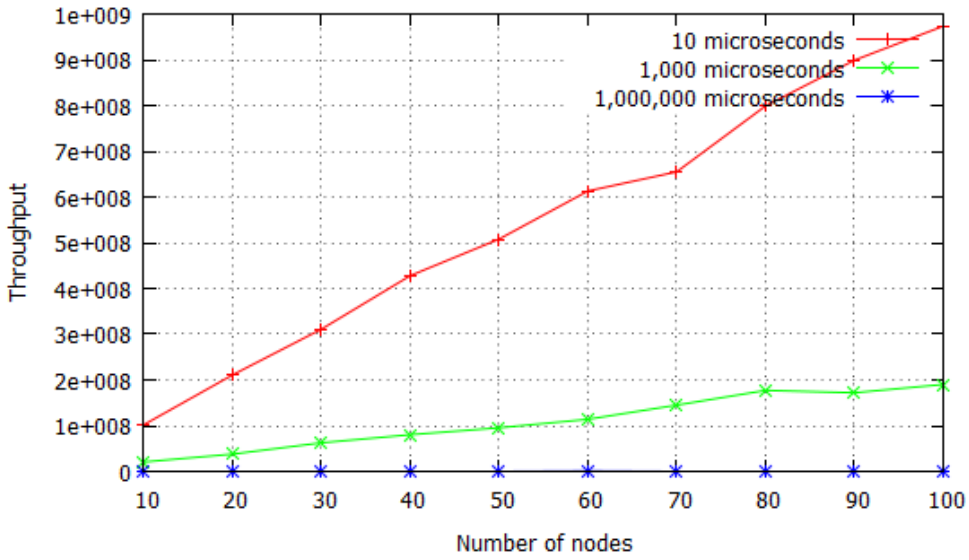


Figure 4.8: Scalability vs. Computation Time

4.3.4 Data Size & Computation Time

Previously, we have seen the individual effects of data size and computation time on the scalability of distributed Erlang (Sections 4.3.2 and 4.3.3). In this section we aim to discover how distributed Erlang scales when both data size and computation time are relatively large.

The following commands are used in the benchmark:

- point-to-point commands, i.e. *spawn* and *rpc*, with 1000 bytes argument size and 1000 microseconds computation time.

We chose 1000 bytes for the argument size because it is a relatively large size for an inter-process communication. Also, running a tail recursive function for 1000 microseconds can be considered as a relatively computation-intensive function. Accessing more than 100 nodes on the Kalkyl cluster is difficult because it's a highly demanded and busy cluster. But we could run this benchmark up to 150 nodes and 1200 cores (8 cores per each node) to see how distributed Erlang scales on that size.

Figure 4.9 shows the result of scaling with a large computation and data size. We see from the figure that distributed Erlang scales linearly up to 150 nodes under relatively heavy data and computation loads when no global commands are used.

However, this doesn't mean that all point-to-point commands have the same scalability and performance. Figure 4.10 depicts the latency of *spawn* and *rpc* commands and it shows that the latency of *spawn* is much less in comparison with *rpc*. In the next section, we discuss why *rpc* doesn't scale well.

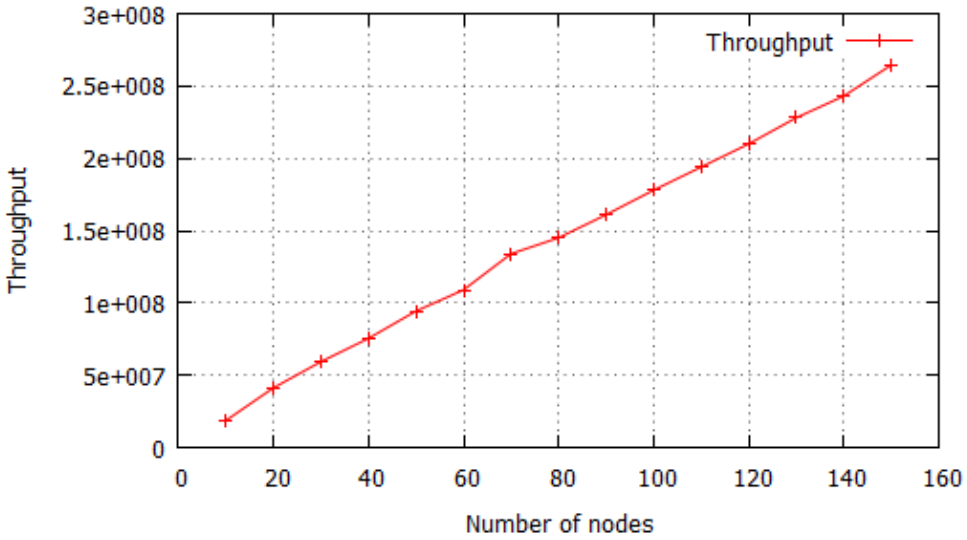


Figure 4.9: Scalability vs. Data Size & Computation Time

4.3.5 Server Process

Our experience with Riak 1.1.1 (Section 3.4) shows how an overloaded server process could limit the scalability of a distributed Erlang application [5]. This section investigates the scalability of two common server processes in Erlang/OTP: generic server processes (*gen_server*) and finite state machine processes (*gen_fsm*). As mentioned in Section 4.2.3, a server process call is a point-to-point command. However, in this benchmark, we use all kinds of point-to-point commands, i.e. server process calls and non-server process calls such as *spawn* and *rpc*. Using all kinds of point-to-point commands makes us able to compare the scalability of

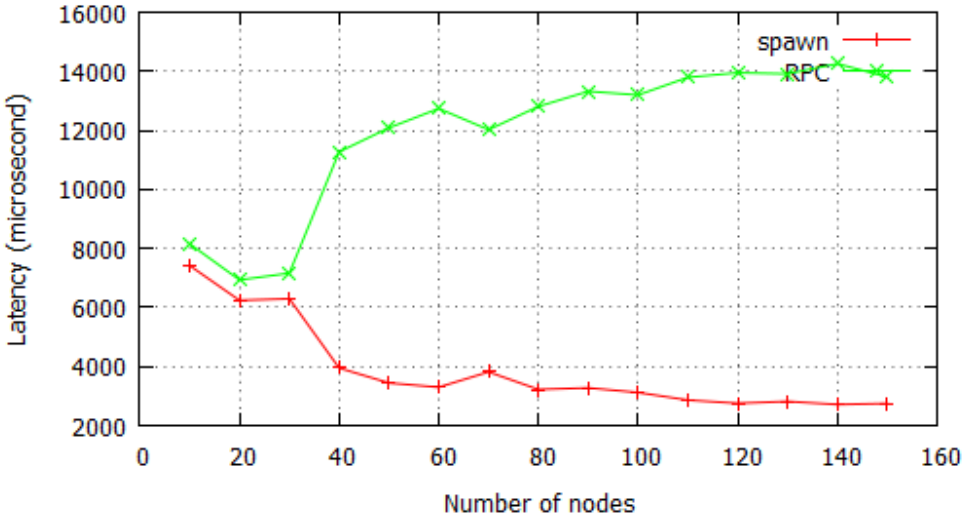


Figure 4.10: Latency of point-to-point commands

server process calls with that of non-server process calls. The following commands are used in the benchmark:

- Non-server process point-to-point commands, i.e. *spawn* and *rpc*, with 10 bytes argument size and 1 microsecond computation time.
- Server process synchronous calls, i.e. *gen_server* and *gen_fsm*, with 10 bytes argument size and 1 microsecond computation time.

Figure 4.11 compares the scalability of distributed Erlang with different percentages of server process calls, i.e. 1% (red line), 50% (green line), and 100% (blue line). For example, when server process call is 1% (the red line), the other 99% of the calls are non-server process, i.e. *spawn* and *rpc*. We see from the figure that as more server calls are used, the scalability improves. For instance, the best scalability is achieved when all calls are server process (the blue line) and the worst scalability occurs when 1% of calls are server process calls (the red line).

We also explore the latency of each command individually to understand which commands' latency increase when the cluster size grows. Figures 4.12 and 4.13 present the latency of commands that we used in the benchmark for 1% and 50% of server process calls. As the figures show, the latency of *rpc* calls rises when cluster size grows. However, the latency of the other commands such as *spawn*, *gen_server*, and *gen_fsm* do not increase as cluster size grows. We see that server processes scale well if they are used properly, i.e. not becoming overloaded as we experienced with Riak [5].

To find out why *rpc*'s latency increases as the cluster size grows, we need to know more about *rpc*. Figure 4.14 shows how an *rpc* call is handled in Erlang/OTP. There is a generic server

process (gen_server) on each Erlang node which is named *rex*. This process is responsible for receiving and handling all *rpc* requests that come to an Erlang node. After handling the request, generated results will be returned to the source node. In addition to user applications, *rpc* is also used by many built-in OTP modules, and so it can be overloaded as a shared service. In contrast with *rpc*, *spawn* is an asynchronous call and the request is handled by a newly-generated process on the target node. This feature makes *spawn* more scalable in comparison with *rpc*.

Alternatively, one also can implement one's own *gen_server* process to handle incoming requests synchronously. This approach reduces the possibility of overloading the process, since one's application is the only client for that server process.

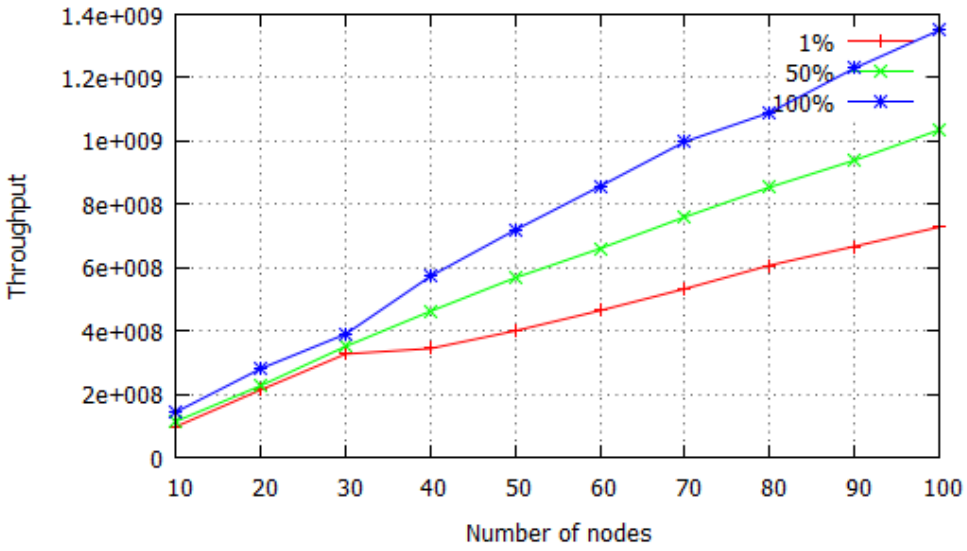


Figure 4.11: Scalability vs. Percentages of Server Process Calls

4.4 Discussion

This chapter has investigated the scalability limits of distributed Erlang by employing DE-Bench. We have presented the design, implementation, and deployment of a scalable peer-to-peer benchmarking tool to measure the throughput and latency of distributed Erlang commands on a cluster of Erlang nodes.

We have demonstrated that global commands are bottlenecks for the scalability of distributed Erlang (Figure 4.5). In the next chapter, we will discuss and evaluate how the new scalable distributed Erlang (SD Erlang) improves this limitation [101].

We have also measured the scalability of distributed Erlang with a relatively large data and computation size. We observed from Figure 4.9 that distributed Erlang scales linearly up

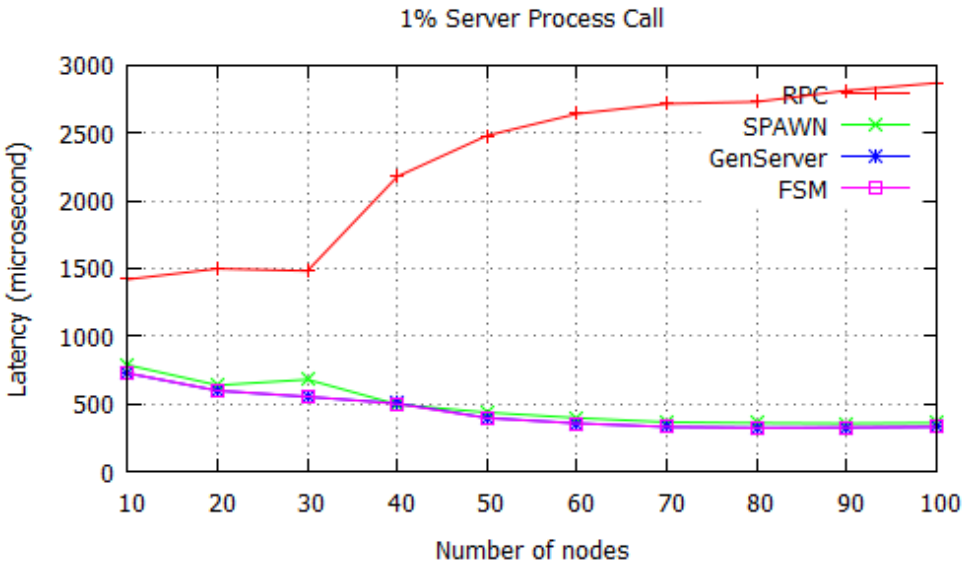


Figure 4.12: Latency of Commands for 1% Server Process Call

to 150 nodes when no global command is used. Our results reveal that the latency of *rpc* calls rises as cluster size grows (Figure 4.10). This shows that *spawn* scales much better than *rpc* and using *spawn* instead of *rpc* for the sake of scalability is advised. Moreover, we have shown that server processes scale well and they have the lowest latency among all point-to-point commands (Figures 4.11, 4.12, and 4.13).

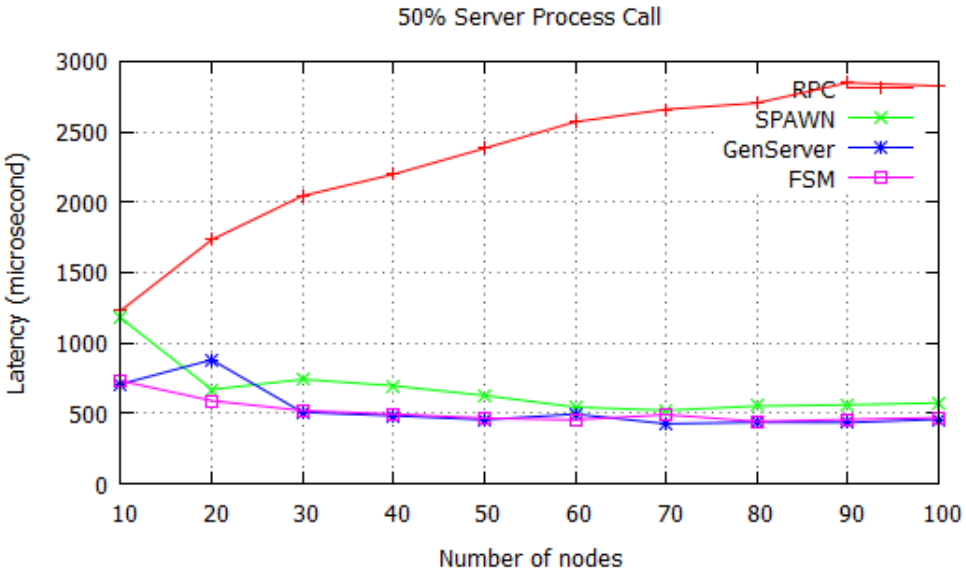


Figure 4.13: Latency of Commands for 50% Server Process Call

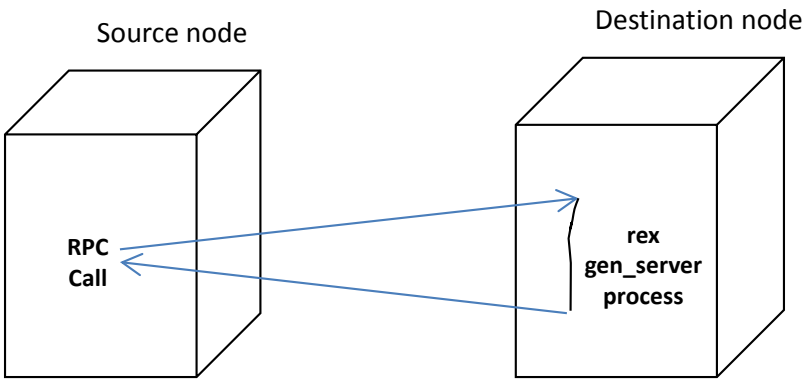


Figure 4.14: *rpc* call in Erlang/OTP

Chapter 5

Evaluating Scalable Distributed Erlang

As part of the RELEASE project, the Glasgow University team has designed and implemented Scalable Distributed Erlang (SD Erlang) to improve the scalability of distributed Erlang [8]. SD Erlang aims to provide mechanisms for controlling locality and reducing connectivity, and to provide performance portability.

This chapter investigates whether SD Erlang can improve the scalability of reliable distributed Erlang systems. We evaluate SD Erlang by developing the first ever demonstrators for SD Erlang, i.e. DE-Bench, *Orbit* and *Ant Colony Optimisation*(ACO).

Section 5.1 gives a brief introduction to the new Scalable Distributed Erlang. Section 5.2 employs DE-Bench to evaluate SD Erlang's group name registration as an alternative to global name registration. An SD Erlang version of the *Orbit* computation is designed, implemented, and evaluated in Section 5.3. We design and implement a reliable version of distributed ACO in Section 5.4. The reliability of ACO is evaluated by employing Chaos Monkey. We use SD Erlang to alleviate the cost of reliability by designing a scalable reliable version of ACO. The impact of SD Erlang on network traffic is also discussed. The work in this chapter is reported in [8, 10, 2].

5.1 Scalable Distributed Erlang

Scalable Distributed Erlang (SD Erlang) offers an alternative connectivity model for distributed Erlang [8]. In this model, nodes are grouped into a number of s_groups in which nodes are fully connected and communicate directly, whereas connectivity between s_groups are non-transitive and communication between nodes belong to different s_groups goes through

gateway nodes. Moreover, SD Erlang provides group name registration as a scalable alternative to global name registration. In this model, there is not a global name space and each s_group has its own namespace which is shared among the group members only.

5.1.1 Connectivity Model in SD Erlang

By default, nodes in a distributed Erlang system are fully connected and each node in a cluster can communicate directly to any other nodes in the cluster. In this model the number of connections between N nodes is $N(N-1)/2$ or $O(N^2)$. For example, there are 28 connections between 8 nodes in a full mesh network as shown in Figure 5.1.

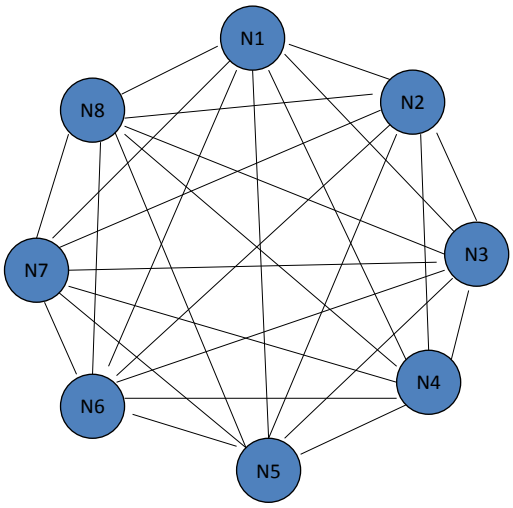


Figure 5.1: Mesh Connectivity Model in Distributed Erlang for 8-Node Cluster

Connections between Erlang nodes are by default transitive which means if node *A* connects to node *B*, and node *B* has a connection to node *C*, then node *A* will connect to node *C* automatically [50]. It is widely believed that the quadratic growth in the number of connections in a full mesh network can limit the scalability of distributed Erlang at large scales.

SD Erlang offers overlapping scalable groups (s_groups) to reduce the number of connections between nodes. Connections between nodes in an s_group are transitive whereas connections between nodes from different s_groups are non-transitive. Figure 5.2 depicts two s_groups communicating through gateways (nodes 2 and 5 are gateways which are denoted by red circles in the figure). In this example, there is one gateway node per s_group, but for larger s_groups there could be several gateways per s_group. S_groups can have common nodes, i.e. a node can belong to several s_groups. For example, Figure 5.3 shows two s_groups communicating through a common gateway node (node 3 denoted by a red circle in the figure).

We have seen from Section 4.3.1 that global name registration limits the scalability of distributed Erlang. SD Erlang provides group name registration as an alternative to global name

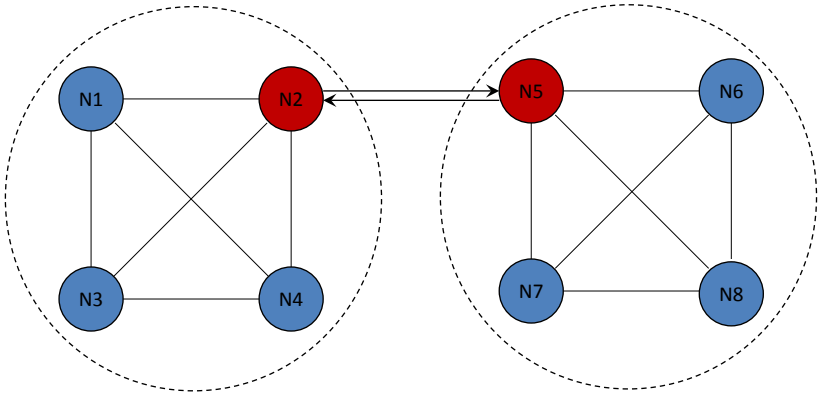


Figure 5.2: Two s_groups Communicating Through Gateways

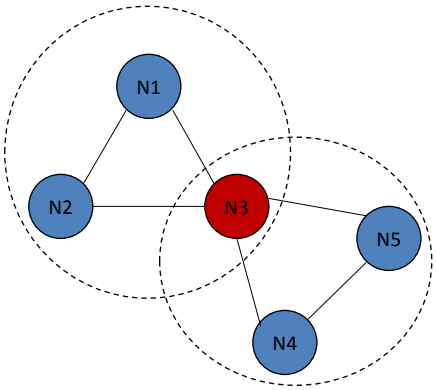


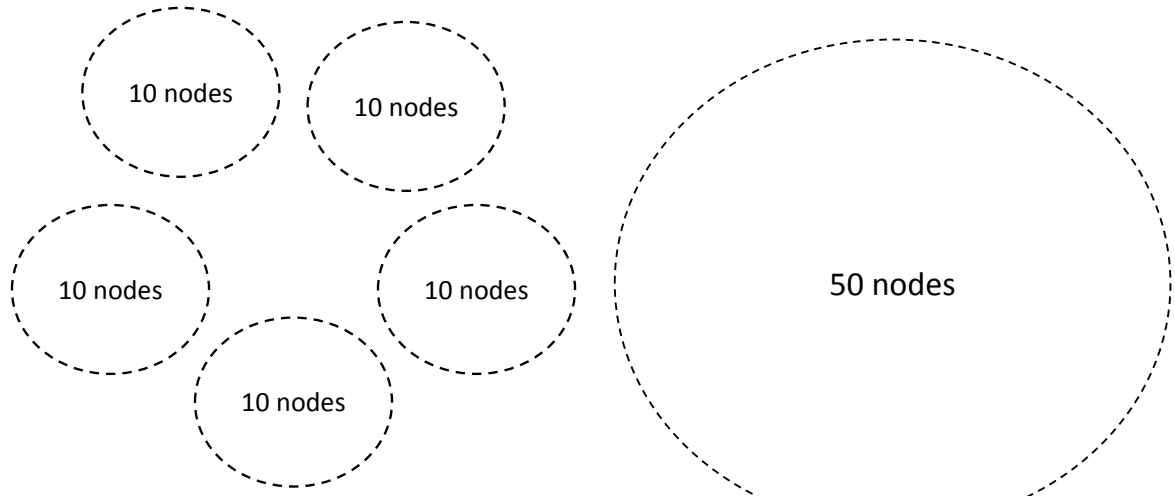
Figure 5.3: Two s_groups Communicating Through a Common Gateway Node

registration. In this approach, each s_group has its own namespace and names are registered within s_groups rather than globally. Group name registration reduces the cost of name registration by limiting the number of engaged nodes within an s_group.

5.2 DE-Bench

Chapter 4 discussed the design and implementation of DE-Bench, a parameterized fault-tolerant benchmarking tool for distributed Erlang. This section employs DE-Bench to measure the scalability of group name registration in SD Erlang and compares the results with the impact of global commands on the scalability of distributed Erlang.

The benchmark was conducted on the Kalkyl cluster with different sizes, i.e 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 nodes (the platform specification is given in Section 3.4.1) [92]. In SD Erlang, the size of s_groups are 10 nodes that means for example we have 1 s_group at a 10-node cluster, 5 s_groups at a 50-node cluster, and 10 s_groups at a 100-node cluster. Figure 5.4 compares a 50-node cluster for SD Erlang and distributed Erlang. As shown in the figure, in SD Erlang the available 50 nodes are grouped in 5 s_groups (Figure 5.4a).



(a) SD Erlang: 50-node Cluster Grouped in 5 s_groups

(b) Distributed Erlang: 50-node cluster

Figure 5.4: SD Erlang 50-node Cluster vs Distributed Erlang 50-node Cluster

However, in distributed Erlang all 50 nodes are fully connected and have a unique namespae (Figure 5.4b).

The following commands are used to benchmark distributed Erlang:

- point-to-point commands, i.e. *spawn* and *rpc*, with 10 microseconds computation and 10 bytes data size
- global commands, i.e. *global:register_name* and *global:unregister_name*
- the local command *global:whereis_name*

The following commands are used to benchmark SD Erlang:

- point-to-point commands, i.e. *spawn* and *rpc*, with 10 microseconds computation and 10 bytes data size
- group commands, i.e. *s_group:register_name* and *s_group:unregister_name*
- the local command *s_group:whereis_name*

As we discussed in Section 4.2.3, the percentage of each command in DE-Bench is tunable. For this measurement, the percentage of global commands is 0.01, i.e. 1 global command runs per 10,000 point-to-point commands. As s_group size is 10 nodes, a registered name is replicated on 10 nodes of a particular s_group, whereas in distributed Erlang a name is replicated on all nodes.

Figure 5.5 compares the scalability of distributed Erlang with SD Erlang. We see from the figure that up to 40 nodes distributed Erlang and SD Erlang perform similarly, and beyond 40 nodes distributed Erlang fails to scale, whereas SD Erlang continues to scale linearly.

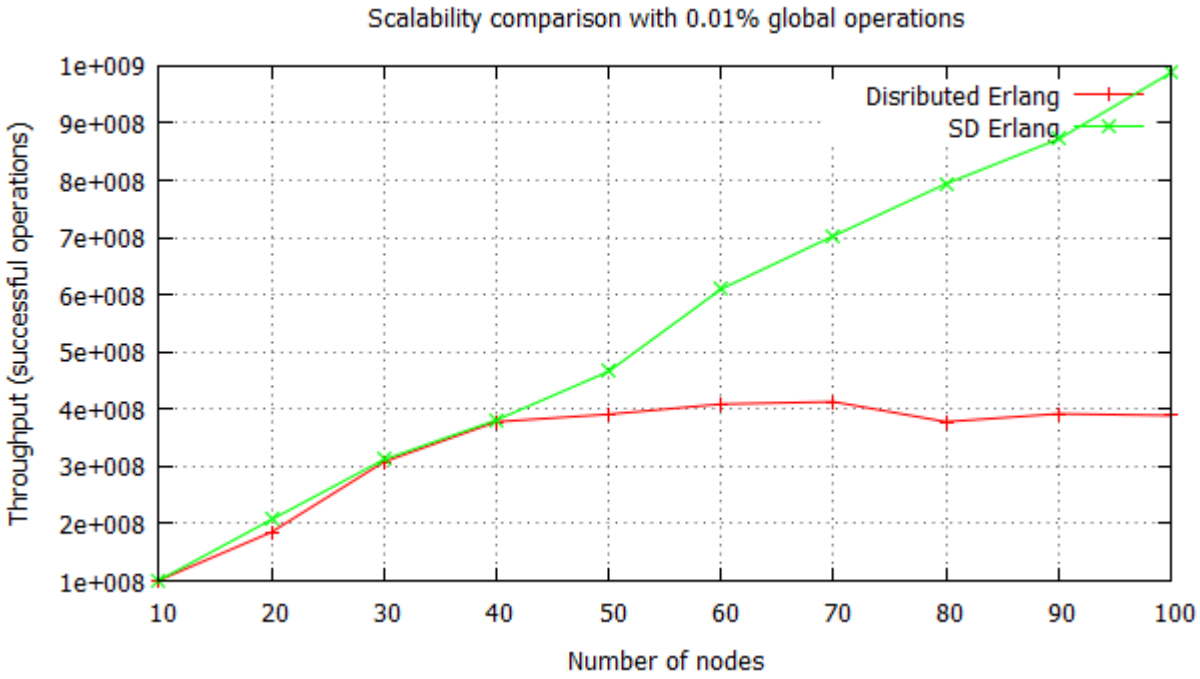


Figure 5.5: Impact of Global Commands on the Scalability of Distribute Erlang and SD Erlang

Figures 5.6 and 5.7 show the benchmark results of the 70-node cluster for distributed Erlang and SD Erlang respectively. There are four rows in the figures: the first row presents the throughput, i.e. the total number of successful commands that run on all 70 nodes. It encompasses all commands including local, point-to-point, and global or group commands. We see from the first row of the figures that the throughput for distributed Erlang is approximately 1,250,000 commands/sec and for SD Erlang the total throughput is 2,400,000 commands/sec. The second row represents the throughput of commands individually. As we defined, the number of global or group commands is 0.01% of all executed commands. The third row shows the number of failures during the benchmark. Each failure means that a command regardless of its type (point-to-point, local, or global) fails to run successfully and raises an exception. As we see from the row, there has been no failed command during the benchmark. The last row shows the latency of each commands individually. We see from the figures that the latency of local and point-to-point commands are negligible in compression with global and group commands. The mean latency of a global name registration on a 70-node cluster is approximately 15 seconds, whereas a group name registration on a 70-node cluster is approximately 20 milliseconds.

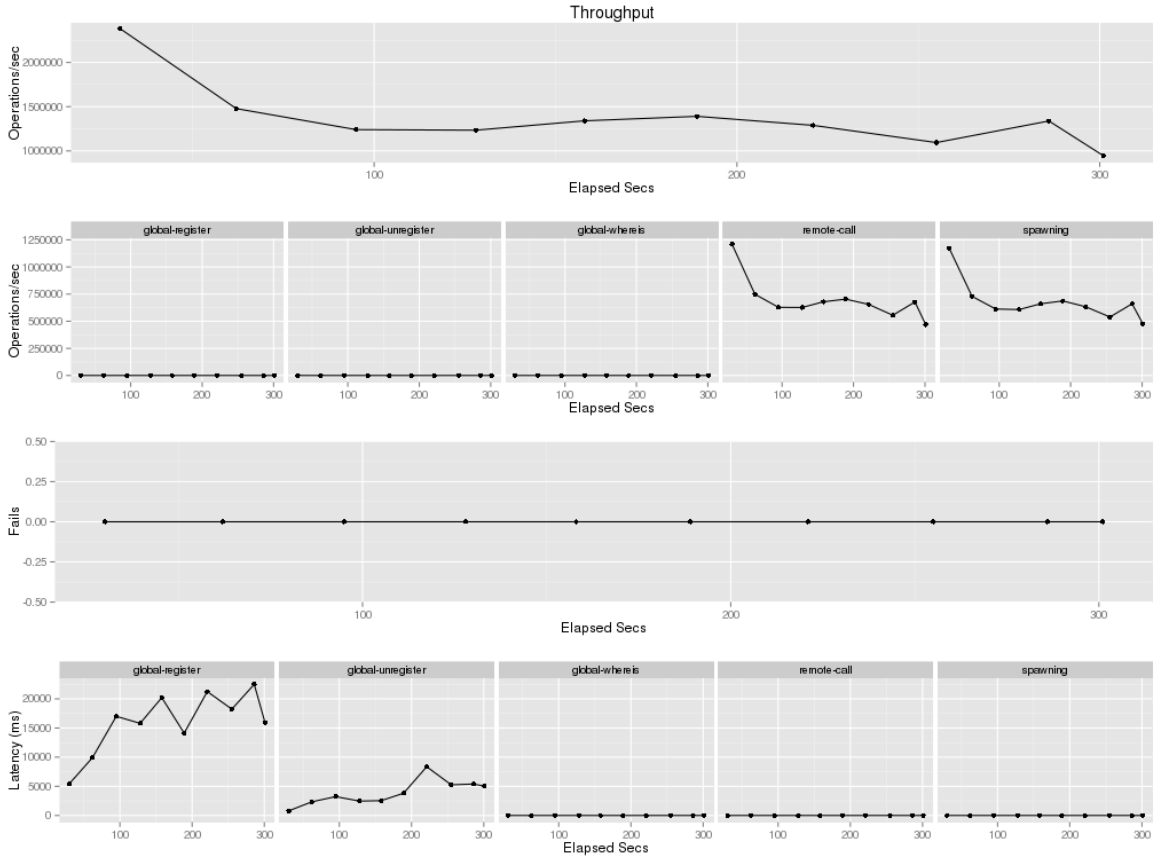


Figure 5.6: Latency of Commands in 70-node Cluster of Distributed Erlang

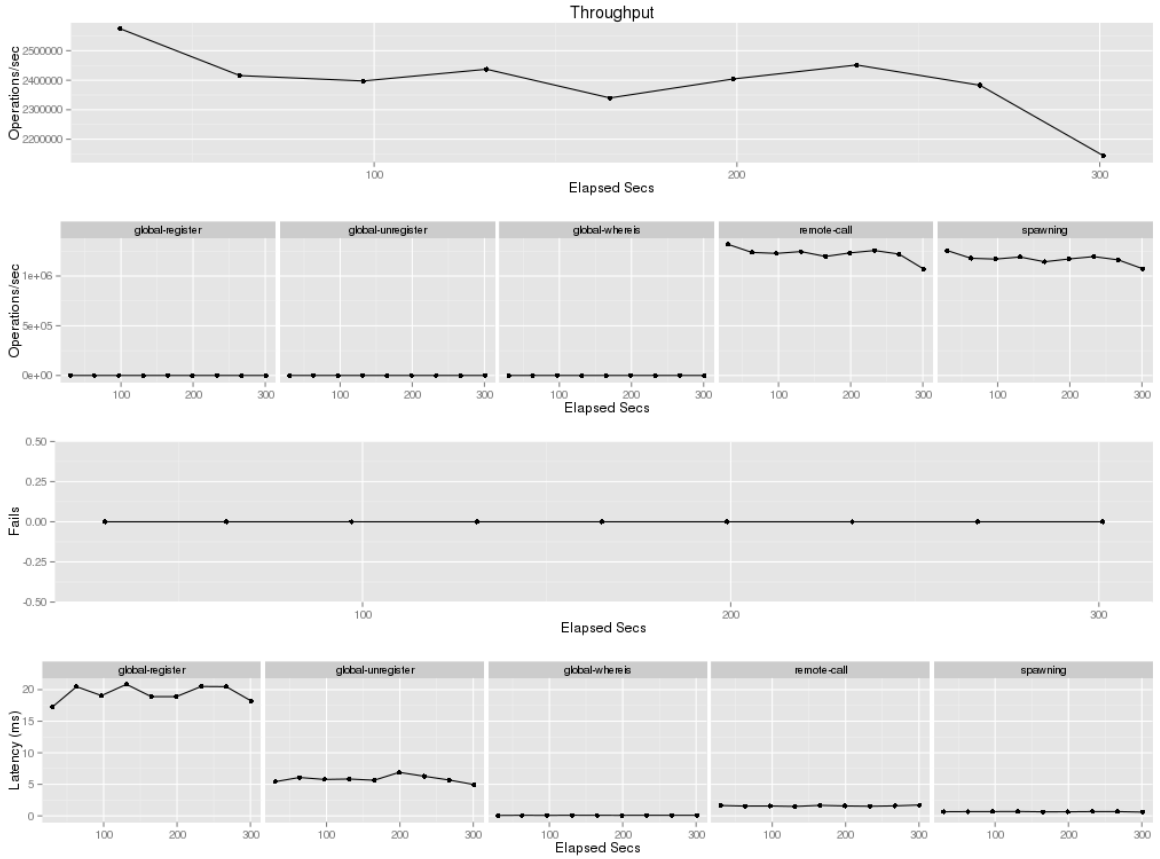


Figure 5.7: Latency of Commands in 70-node Cluster of SD Erlang

5.3 Orbit

Orbit is a symbolic computing kernel and is a generalization of a transitive closure computation [102]. To compute the $Orbit$ for a given space $[0..X]$ a list of generators $g1, g2, ..., gn$ are applied on an initial vertex $x_0 \in [0..X]$. This will create new numbers $(x_1...x_n) \in [0..X]$. Repeatedly, generator functions are applied on each of the new numbers and this continues until no new number is generated.

We implement an SD Erlang version of Orbit based on an existing distributed version developed by Dr Patrick Maier from Glasgow University (Appendix C.1). We evaluate the scalability of both versions to measure the impact of SD Erlang on the scalability of the Orbit computation.

5.3.1 Distributed Erlang Orbit

An Orbit computation is initiated by a master process. The master establishes a P2P network of worker processes on available nodes. Each worker process owns part of a distributed hash table. A hash function is applied on a generated number to find to which worker it belongs.

To detect the termination of Orbit computation, a credit/recovery distributed algorithm is used [103]. Initially the master process has a specific amount of credit. Each active process holds a portion of the credit and when a process becomes passive, i.e. the process becomes inactive for a specific period of time, it sends the credit it holds to active processes. When the master process collects all the credit, it detects that termination has occurred.

The distributed Erlang Orbit has a flat design in which all nodes are fully connected (Figure 5.8). The master process initiates the Orbit computation on all worker nodes and each worker node has connections to the other worker nodes. Worker nodes communicate directly with each other and finally report their calculated results to the master node.

The following features in Orbit makes the benchmark a desirable case study for SD Erlang:

- It uses a Distributed Hash Table (DHT) similar to NoSQL DBMS like Riak that use replicated DHTs [68] (the module *table* from Appendix C.1).
- It uses standard P2P techniques and a credit/recovery distributed termination detection algorithm (the module *credit* from Appendix C.1).
- It is only a few hundred lines and has a good performance and extensibility.

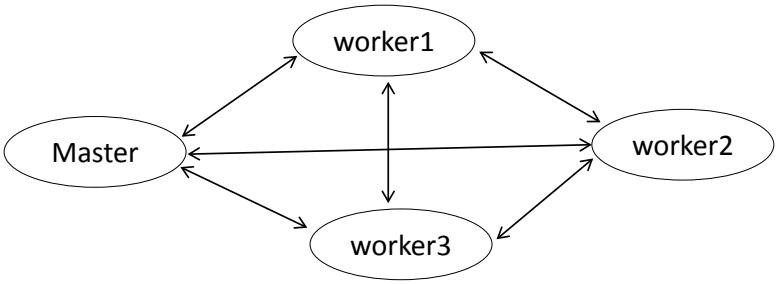


Figure 5.8: Communication Model in Distributed Erlang Orbit

5.3.2 Scalable Distributed Erlang Orbit (SD Orbit)

This section employs SD Erlang to improve the scalability of distributed Erlang Orbit. We introduce an alternative connectivity model to reduce the number of connections between participant nodes in the computation (Appendix C.2).

Communication Model

To reduce the number of connections between Erlang nodes in the Orbit computation, we propose a new design for Orbit in which nodes are grouped into sets of s_groups. In SD-Erlang, s_group nodes have transitive connections with the nodes from the same s_group, and non-transitive connections with other nodes.

There are two kinds of s_groups in this model: *master* and *worker* (Figure 5.9). There is only one *master* s_group that the master node and all sub-master nodes belong to. There can be multiple *worker* s_groups in the system. Each *worker* s_group has only one sub-master node and an arbitrary number of worker nodes.

Communication between nodes inside an s_group is done directly but when a worker node needs to communicate with another worker node outside its own s_group, communication is done through sub-master nodes (the module *sub-master* from Appendix C.2). In this case, the number of messages increases three times, i.e. assume A and C belong to different s_groups, so instead of sending a message from A to C directly (A→C), we would have: A→ $A_{submaster}$ → $C_{submaster}$ →C.

This approach reduces the number of connections between nodes. The number of connections on a worker node is equal to the size of s_group and the number of connections on a sub-master node is equal to the number of worker nodes inside the s_group *plus* the number of all the other sub-master nodes.

In a cluster with N nodes, a distributed Erlang Orbit node has $(N-1)$ TCP connections whereas in SD Orbit with s_group size M, a worker node has $(M-1)$ TCP connections and

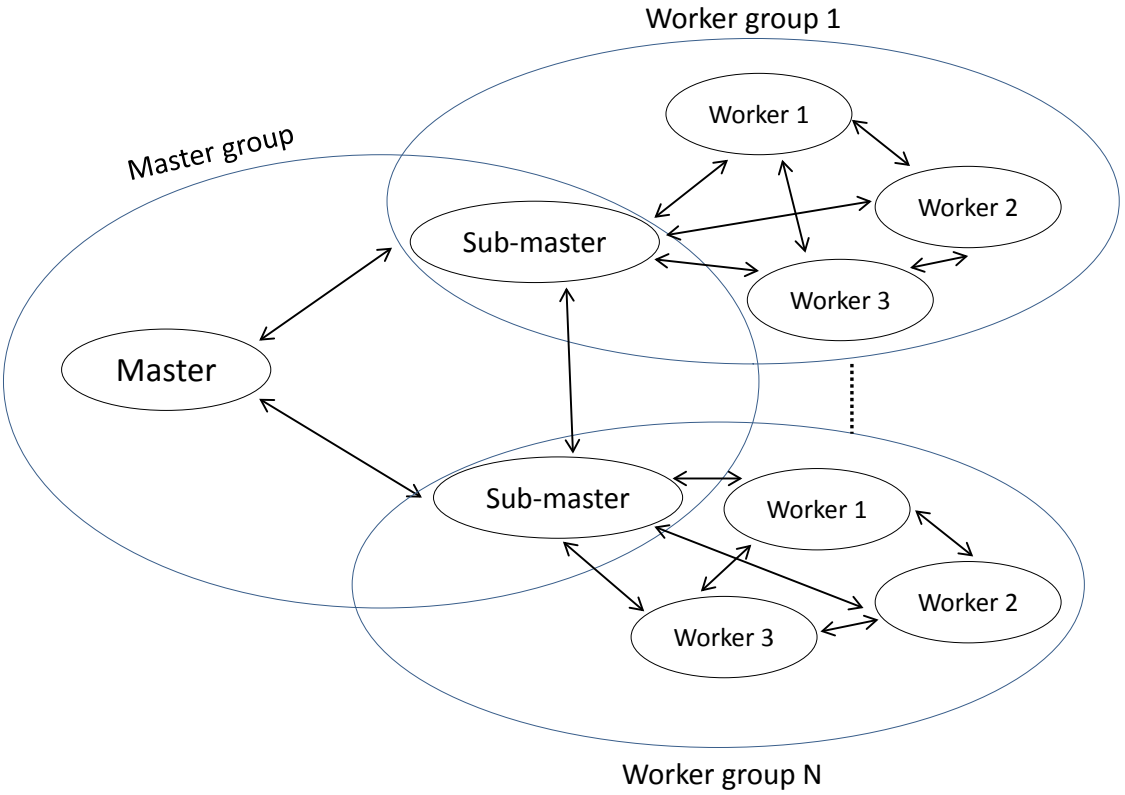


Figure 5.9: Communication Model in SD Erlang Orbit

a sub-master node has $(M-1)+(\frac{N}{M}-1)$ connections.

Computation Model

The Orbit computation is started by the master process on the master node. The master process dynamically and evenly divides nodes into a number of s_groups (the module *grouping* from Appendix C.2). After creating s_groups, the first node of each s_group is chosen as sub-master node. Then, the master process runs two kind of processes, i.e. *submaster* and *gateway*, on each sub-master node. A submaster process is responsible to initiate, collect credits and data, and terminate the worker processes in its own s_group and send the collected data back to the master process. In other words, a submaster process is master's representative in an s_group and behaves like an interface between master and worker processes.

The other kind of process on a sub-master node is *gateway*. The number of gateway processes on a sub-master node can be defined based on the number worker processes on a worker node. When two worker processes from two different s_groups need to communicate, a gateway process from each s_groups participates as follow:

Assume *Process1* from *s_group1* needs to send a message to *process2* in *s_group2*. The name of gateway processes are *gateway1* and *gateway2* in *s_group1* and *s_group2* re-

Group1	1-100
Group2	101-200
Group3	201-300

Table 5.1: Group Hash Partition

Process1	1-25
Process2	26-50
Process3	51-75
Process4	76-100

Table 5.2: Process Table Partition within Group 1

spectively. The message path would be: *Process1*→ *gateway1*→ *gateway2*→ *process2* (as shown in Figure 5.10).

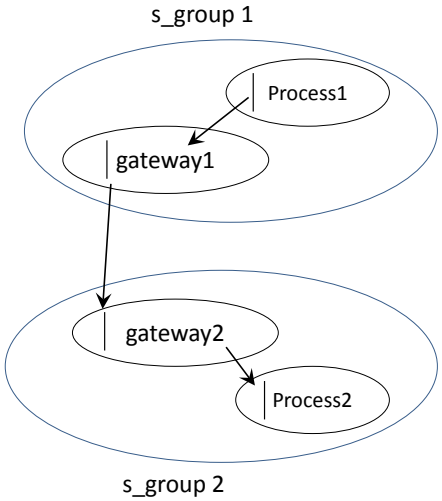


Figure 5.10: Communication Between Two Worker Processes from Different S_group

Double Hashing

In Distributed Erlang Orbit, all processes store a copy of the hash table that specifies which process is responsible for which fragment of the hash table. However, in the SD Erlang design, there are two levels of hash tables. The first level hash table that is created by the master process and stored on all sub-master processes, specifies which group is responsible for which part of the table. The second level hash table is created by sub-masters and stored on worker processes. Each worker process stores a table that specifies which process in the s_group is responsible for which part of the table. For example Table 5.1 shows how range 1-300 is divided among three s_groups and Table 5.2 shows how *Group1* is divided among 4 processes equally.

5.3.3 Scalability Comparison

This section compares the scalability of SD Erlang and Distributed Erlang Orbits on the Kalkyl cluster [92]. SD Orbit run on the Kalkyl cluster with 11, 22, 44, 66, 88, 110, 132,

154, 176 nodes. The size of s_groups is 11 and in each s_group there are 1 submaster node and 10 worker nodes. For example on 11-node cluster we have one s_group in which we have 1 sub-master node and 10 worker nodes. On a 44-node cluster, there are 4 s_groups and in each s_group we have one sub-master node and 10 worker nodes, and so in total there are 40 worker nodes and 4 submasters. There are 40 processes on each worker node, and 60 gateway processes on each sub-master node.

In the distributed Erlang version since there is no sub-master node and all nodes behave as worker nodes, we increase 10 nodes in each experiment. Figures 5.11 and 5.12 compare the runtime and the speedup of SD Erlang and Distributed Erlang Orbits. To gain stable results, we run each experiment 11 times and the median value is represented in the diagrams. Vertical lines (the green and pink colors) in Figures 5.11 and 5.12 represent 95% confidence interval for the 11 samples. We observe from the runtime comparison:

- The SD-Erlang Orbit has greater runtime than the distributed Erlang Orbit on 10 and 20 nodes (80 and 160 cores).
- The SD-Erlang Orbit scales better than the distributed Erlang Orbit, and by 40 nodes (320 cores) appears to outperform it.
- We can be confident that the SD-Erlang Orbit is outperformaing the distributed Erlang Orbit on 160 nodes (1280 cores).

We also see from the relative speedup comparison:

- Distributed Erlang Orbit shows better speedup than SD Erlang Orbit on 10 and 20 nodes (80 and 160 cores).
- The SD-Erlang Orbit speeds up better than the distributed Erlang Orbit beyond 40 nodes (320 cores).
- We can be confident that the SD-Erlang Orbit is speeding up better than the distributed Erlang Orbit on 160 nodes (1280 cores).

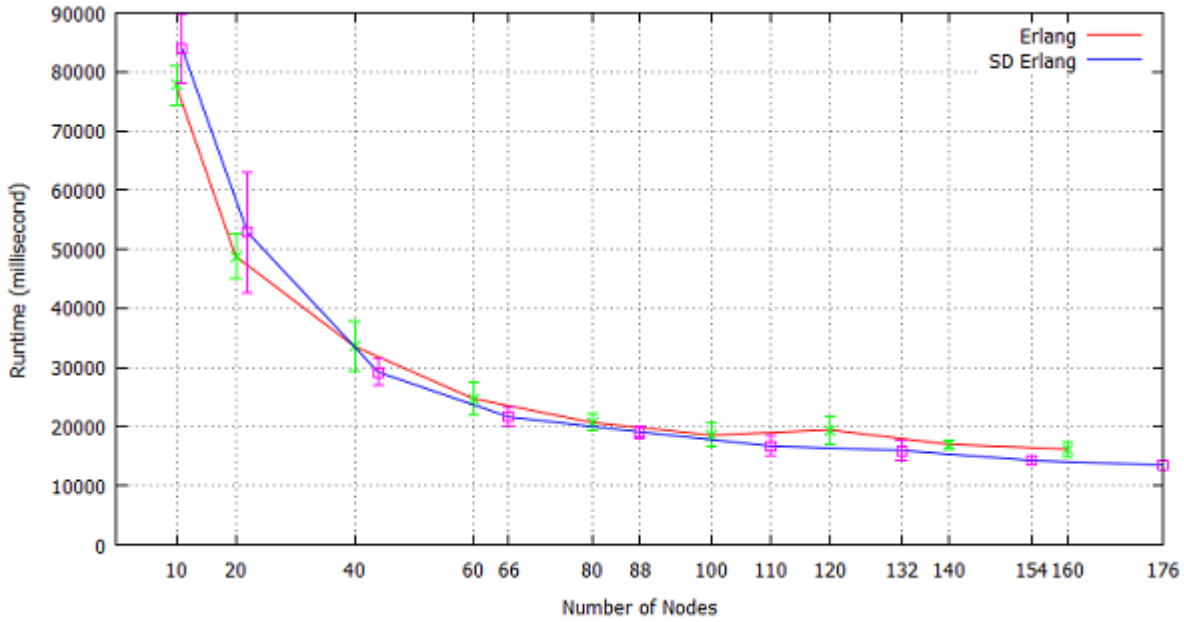


Figure 5.11: Runtime of SD Erlang and Distributed Erlang Orbits

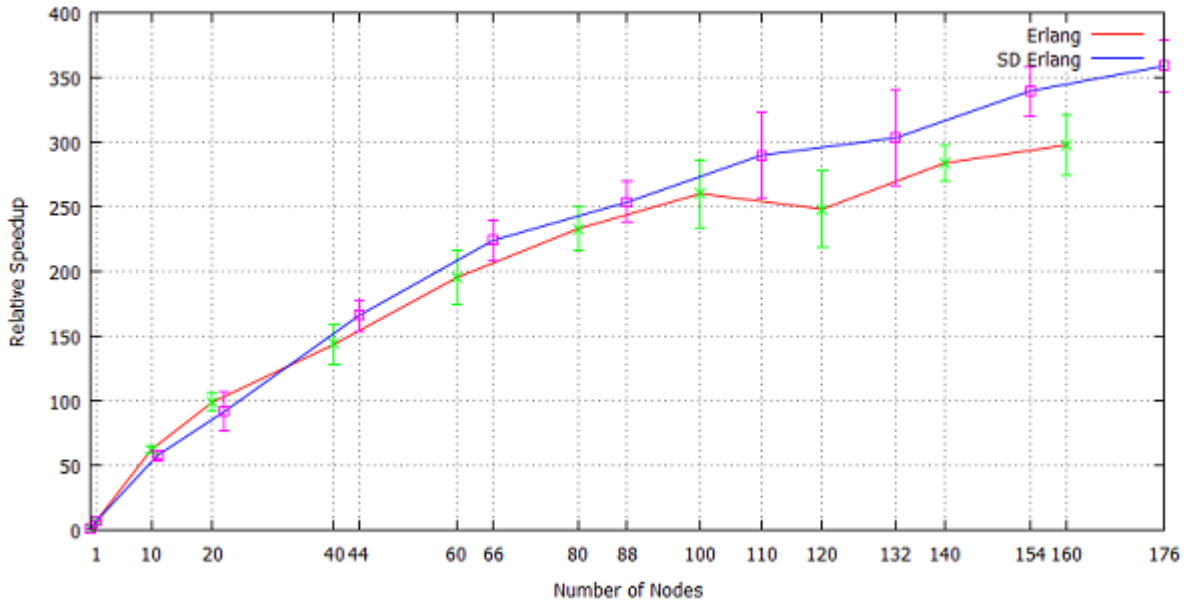


Figure 5.12: Speedup for SD Erlang and Distributed Erlang Orbits

5.4 Ant Colony Optimisation

Ant Colony Optimization (ACO) is a metaheuristic that has proved to be successful in a variety of combinatorial optimization problems such as scheduling problems, routing problems, and allocation problems [104, 105, 106, 107].

The Single Machine Total Weighted Tardiness Problem (SMTWTP) is a well-known scheduling problem which aims to find out the starting times for a given set of jobs on a single processor to minimize the weighted tardiness of the jobs with respect to given due dates [108]. Different heuristic methods have been developed to solve benchmark instances for the SMTWTP successfully [109]. Ant colony optimization is one of the approaches that has been applied for the single machine total tardiness problem [110, 111, 112].

This section improves the scalability and reliability of an existing distributed Erlang version of ACO developed by Dr Kenneth Mackenzie from Glasgow University (Appendix C.3). Section 5.4.1 describes the existing two-level version of ACO. Section 5.4.2 develops a multi-level version of distributed ACO. A comparison of the two-level and multi-level versions are given in Section 5.4.3. A reliable version of ACO is introduced in Section 5.4.4. We employ SD Erlang to improve the scalability of the reliable version in Section 5.4.5. Section 5.4.6 investigates how SD Erlang reduces the network traffic.

5.4.1 Two-Level Distributed ACO (TL-ACO)

The existing distributed ACO developed in Erlang has a two-level design and implementation. Figure 5.13 depicts the process and node placements of the *TL-ACO* in a cluster with N_C nodes. The master process spawns N_C colony processes on available nodes. In the next step, each colony process spawns N_A ant processes on the local node (the modules *colony* and *master* from Appendix C.3). In the figure, all captions and their relevant object have the same color. As the arrows show, communications between the master process and colonies are bidirectional. There are I_M communications between the master process and a colony process. Also, I_A bidirectional communications are done between a colony process and an ant process.

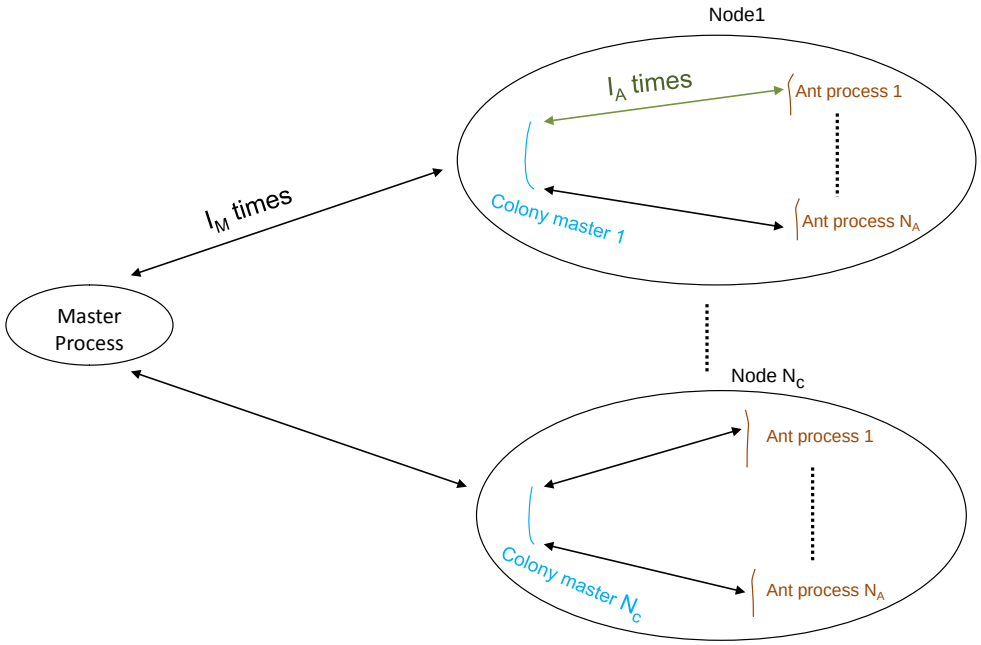


Figure 5.13: Two-Level Distributed ACO

5.4.2 Multi-Level Distributed ACO (ML-ACO)

In *TL-ACO* with a large number of colony nodes the master process, which is responsible for collecting and processing the result from colonies, can become overloaded and a bottleneck for scalability. As a solution to this problem, we propose a Multi-Level design for distributed ACO (*ML-ACO*), in which, in addition to the master node, there can be multiple levels of sub-master nodes to help the master through sharing the burden of collecting and processing the results from colony nodes (Figure 5.14) (the module *ant-submaster* from Appendix C.4). The number of sub-master nodes is adjustable based on the number of colony nodes in a way that each sub-master node handles a reasonable amount of loads.

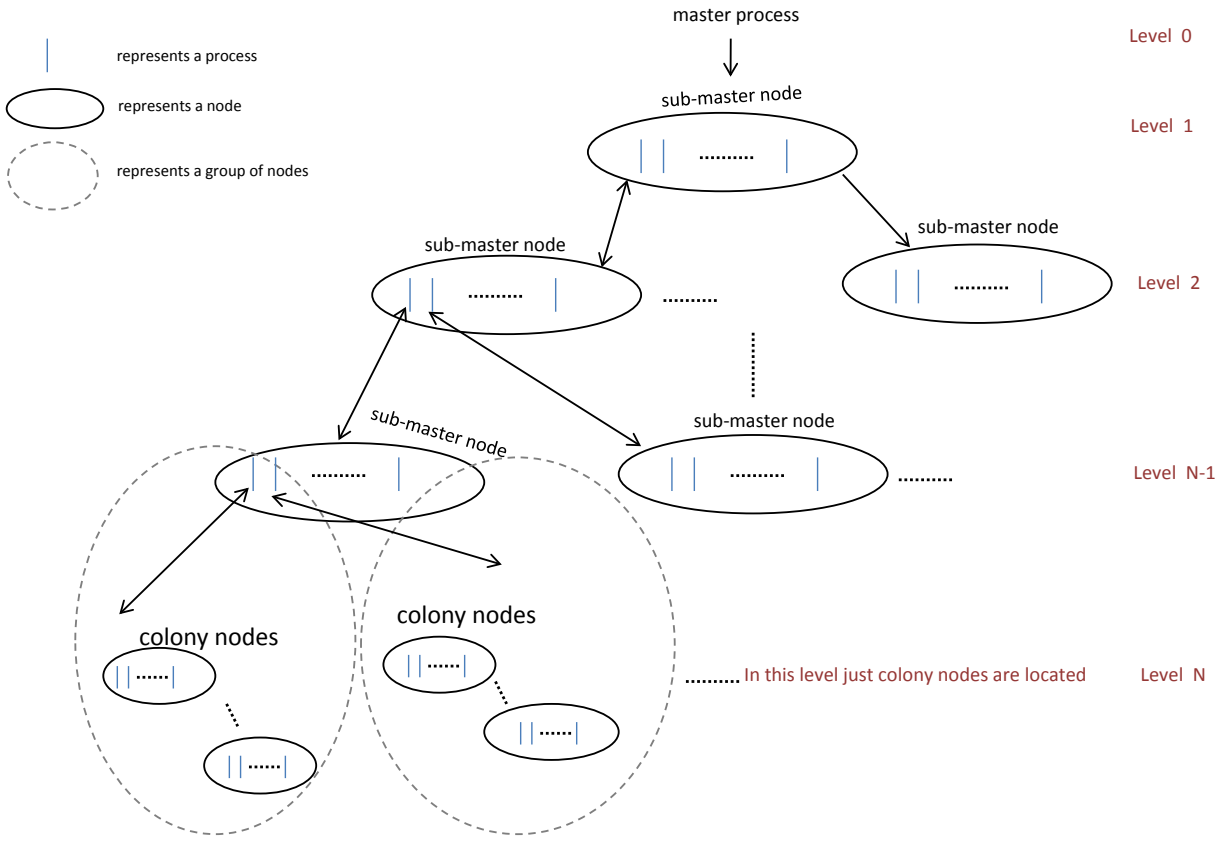


Figure 5.14: Node Placement in Multi Level Distributed ACO

Figure 5.15 depicts the process placement in ML-ACO. If there are P processes per each sub-master node, then the number of processes in level N is P^N and the number of nodes in level N is P^{N-1}. A process in level L creates and monitors P processes on a node at level $L + 1$. However, the last level is an exception because in the last level just colony nodes are located and one colony process per each colony node exists. Thus, a process in level *N-1* (one level prior to the last) is responsible for P node in level N (not P processes) and consequently the number of node in level N (the last level) is P^N.

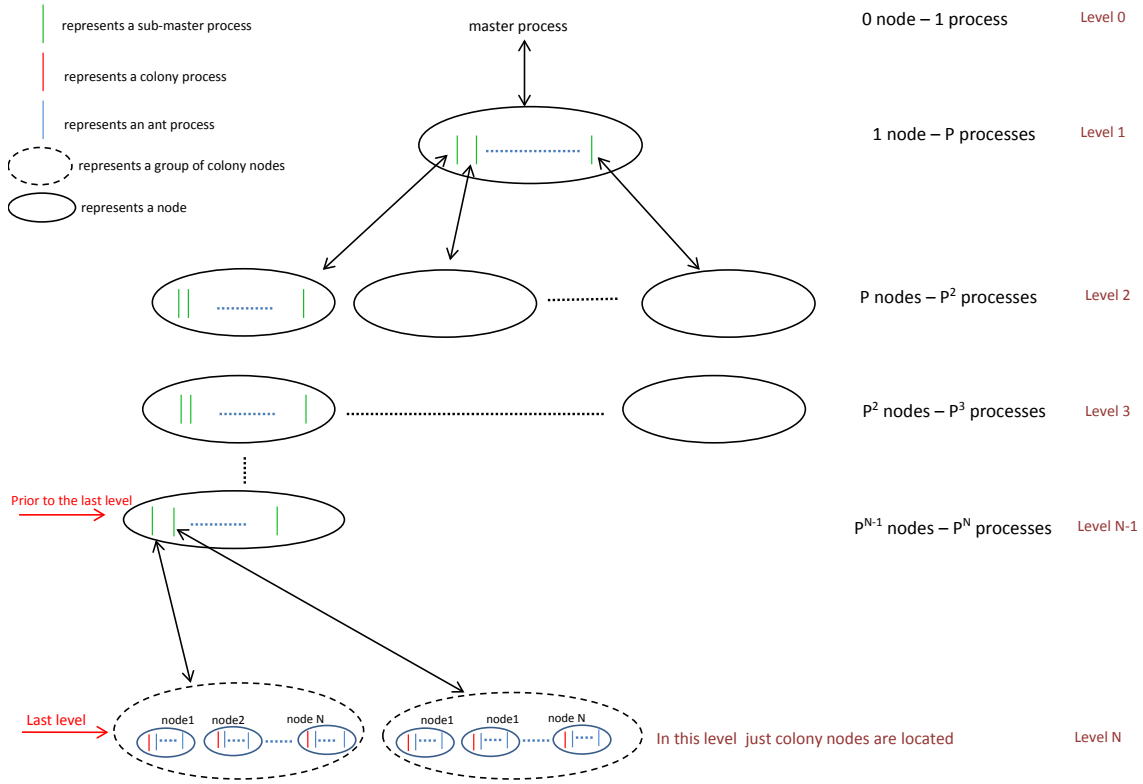


Figure 5.15: Process Placement in Multi Level ACO

To be able to create a multi-level tree of sub-master nodes, we need to formulate the relation between the number of processes, nodes and levels. If the number of processes on each node is P and the number of all available nodes is N, then to find out the number of levels (X) that the tree will have, we need to find the maximum of X in this equation: $1+P+P^2+P^3+...+P^{X-2}+P^X \leq N$. For example, if P=5 and N=150 then the tree will have 3 levels as shown here: $1+5+5^3 \leq 150$. This means we can just use 131 nodes of 150 nodes.

5.4.3 Scalability of ML-ACO versus TL-ACO

To compare the scalability of TL-ACO and ML-ACO, the following parameters are considered in our measurement:

- Input Size: our measurements with different sizes of input show that increasing the input's size just increases the loads on ants processes. For larger input size, ant processes on colony nodes need more time to find their best solution, and it does not affect the master process considerably. In other words, a larger input leads to increase in the size of messages that the master process receives and not the number of messages that it receives.

- Number of Iterations: There are two kinds of iteration in the system: local and global iteration. Local iteration represents the number of times that ant processes communicate with their colony process. Global iteration is the number of times that colonies communicate their best solution with the master process. As the master process handles iterations sequentially, increasing the number of iterations does not make the master process overloaded.

- Number of Colonies: In TL-ACO the main parameter that could lead to an overloaded master process is the number of colony nodes. For a large number of colony nodes, the master process could become overloaded and consequently a bottleneck for scalability. The maximum number of nodes that we could compare both versions is 630 nodes (10 VMS on 63 hosts at Tintin cluster [92]). Our results show that both versions, i.e. TL-ACO and ML-ACO, take roughly the same time to finish. In our measurement, there is 629 colonies on 630-node cluster, and so in TL-ACO version the master process takes 629 messages from its colony nodes in each iteration. The number of received messages (i.e. 629 messages) is not large enough to make the master process overloaded.

Simulation

To investigate the scalability of TL-ACO and ML-ACO we need a large cluster with thousands of nodes and tens of thousands of Erlang VMs. Since we do not have access to such a large cluster, a new version was developed in which each colony node sends more than one message to its parent instead of sending only one message. This approach simulates a large cluster of colony nodes because each colony can play the role of multiple colonies.

Figure 5.16 compares the scalability of TL-ACO and ML-ACO up to 112,500 simulated nodes. The figure presents the middle value of 5 executions and vertical lines show the variation. ML-ACO outperforms TL-ACO beyond 45k simulated nodes (approximately 50K nodes). The measurement is run on GPG cluster with 226 Erlang VMs, 1 VM is dedicated for master and sub-master processes and 225 VMs run colonies [113]. There are 4 ant processes per colony. I_M and I_N (global and local iterations) are 20. Input size is 40, and each colony node sends 100, 200, 300, 400, and 500 messages to its parent. Figures 5.17, 5.18 depict node placements in TL-ACO and ML-ACO respectively. In TL-ACO the master process is singly responsible for collecting results from its children (225 nodes) (Figure 5.17), but in ML-ACO the master process collects the results from 15 sub-master processes and each sub-master process is responsible for 15 colony nodes which are depicted in dotted circles in Figure 5.18.

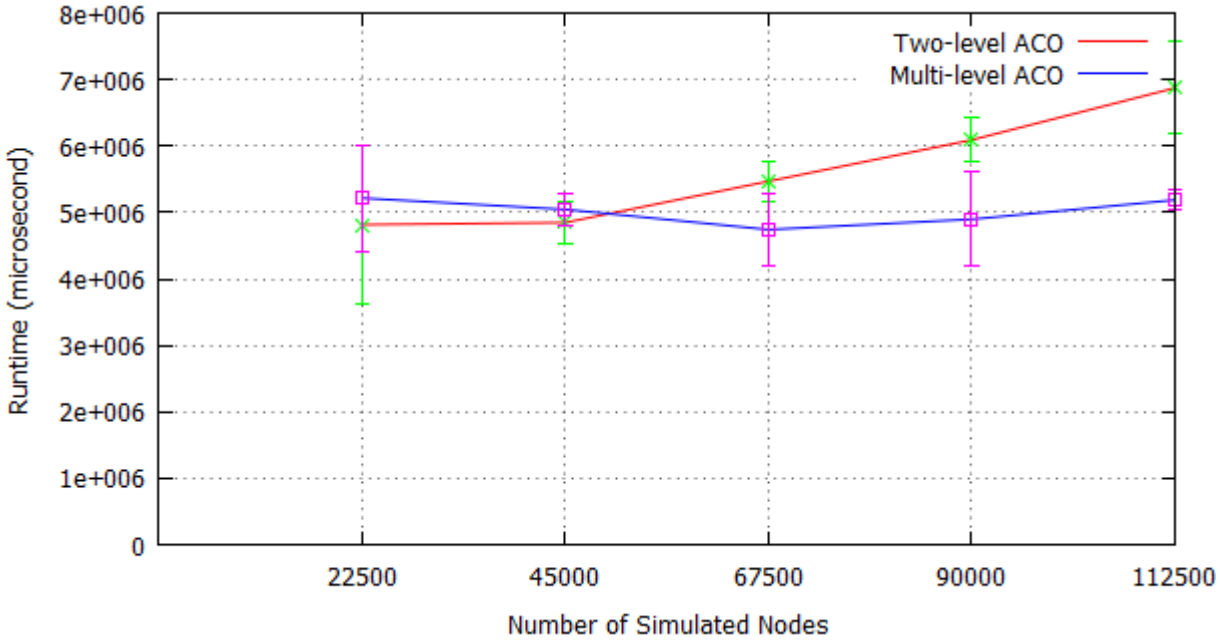


Figure 5.16: Scalability of ML-ACO and TL-ACO

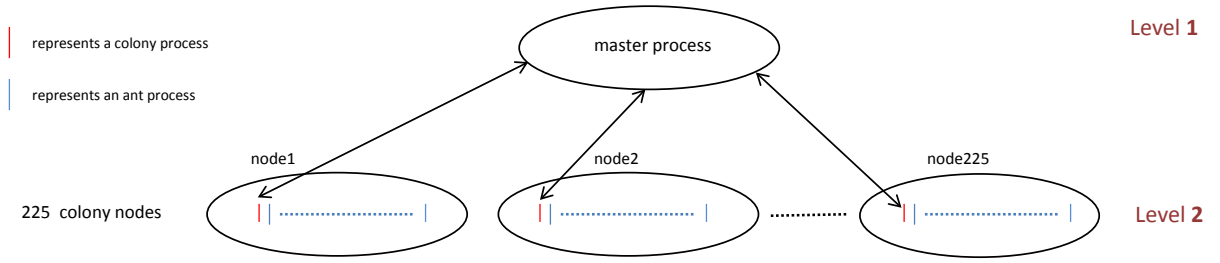


Figure 5.17: TL-ACO with 225 colony nodes

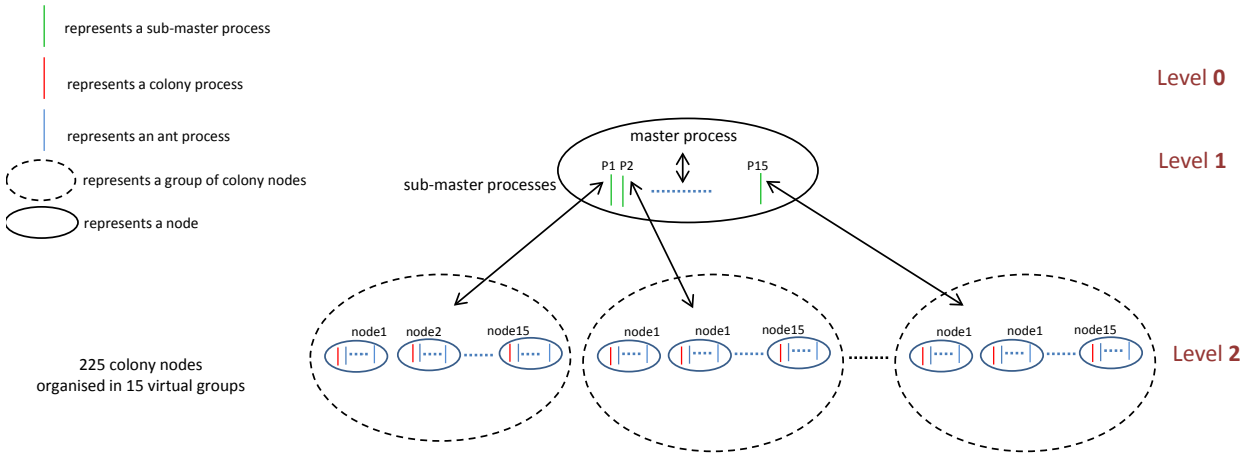


Figure 5.18: ML-ACO with 225 colony nodes

5.4.4 Reliable ACO (R-ACO)

There are four kinds of processes in the ML-ACO version, i.e. Master, Sub-masters, Colonies, and Ants (Figure 5.15). To make the version fault-tolerant, we need to consider the possibility of failure of each kind individually.

Sub-master processes are created recursively to form a multi-level tree structure. The processes on the root node of the tree are created by the master process. Processes on the other levels are created recursively by processes on the prior levels. Thus, the master process will monitor the processes on the root node and each level processes monitor the processes on the next level. The level prior to the last level creates colonies and also collects the results from them. Thus, processes on this level can supervise all the colonies and restart them in case of failure (Appendix C.5).

Each colony creates and also collects the results from its own ant processes. Thus, a colony supervises all its own ant processes.

Ant processes are the last level, and so they do not need to supervise any other processes.

The master process can be created by the *starter process*, a process that starts the computation by creating and supervising the master process. Since the starter process is not involved in computational process over the time, it will not crash and so it can safely monitor the master process. This process can be considered as a user shell process that its duty is just running the master process, and finally printing the result.

In Erlang, if a process dies and restarts again, it will get a new process identifier (Pid). In a fault-tolerant application, other processes should be able to reach that process after its restart. Thus, we need to refer to a process by its name instead of its Pid for reliability purposes. In reliable ACO, the total number of required registered names for an L levels tree with P processes on each node is $P + P^2 + P^3 + ... + P^{L-1} + P^L = \frac{P^{L+1}-P}{P-1}$.

For example, in a tree with 3 levels and 5 sub-master processes on each node, we need 155 global name registrations: $\frac{5^{3+1}-5}{5-1} = 155$ (Figure 5.19).

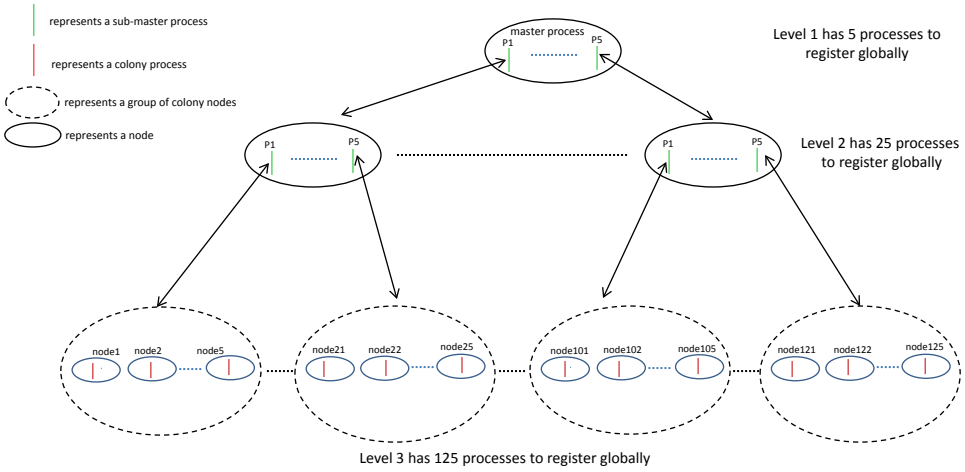


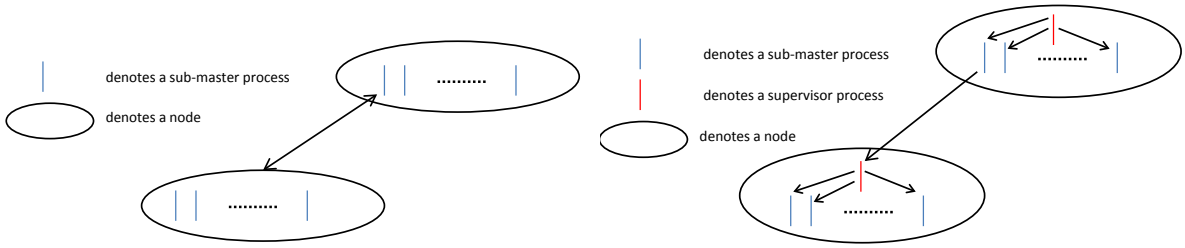
Figure 5.19: Required name registrations in a tree with 3 levels and degree 5

The number of name re-registrations depends on the failure ratio. There will be one name re-registration per each process failure and P re-registrations after a node recovery.

Optimizing Reliable ACO

In the design of reliable ACO given in previous section, a sub-master process supervises all its child processes (Figure 5.20a). For example, if a sub-master process supervises 10 sub-master processes located on a remote node, 10 global name registrations are needed for reliability purposes. To minimize the number of required global name registrations, we can have a local supervisor. As Figure 5.20b shows, in the local supervisor model a sub-master process only supervises a local supervisor process on a remote node, and the local supervisor process is responsible for supervising the sub-master processes located on its locale node. This approach reduces the number of required global name registration form P, where P is the number of sub-master process per node, to 1. Thus, in the optimized reliable ACO, the total number of global name registrations for an L levels tree with P processes on each node is $1 + P^1 + P^2 + ... + P^{L-2} + P^L$.

For example, in a tree with 3 levels and 5 sub-master processes on each node, we need 131 global name registrations: $1 + 5^1 + 5^3 = 131$ (Figure 5.21). This optimization reduces the number of required name registrations for this example from 155 to 131.



(a) Remote Supervisor (b) Local Supervisor

Figure 5.20: Local Supervisor vs. Remote Supervisor

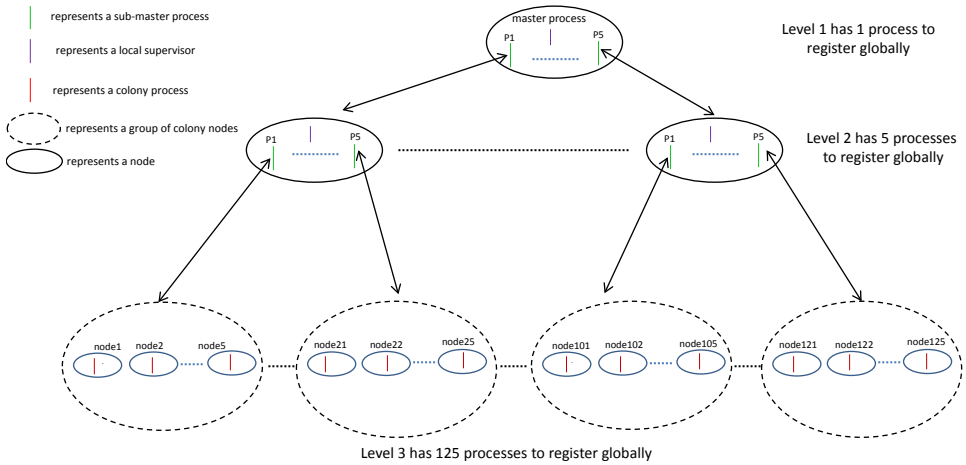


Figure 5.21: Optimized name registrations in a tree with 3 levels and degree 5

Reliability Evaluation

To evaluate the reliability of ACO, we employ Chaos Monkey [114, 11]. Chaos Monkey kills available Erlang processes on a node randomly by sending an exit signal to the processes over its execution time. We ran Chaos Monkey on all computational nodes, i.e. master, sub-masters, and colonies. The results show that the reliable ACO could survive after all failures that Chaos Monkey caused. Victim processes include all kind of processes, i.e. the master, sub-masters, colonies, and ants regardless whether they are initial processes or recovered ones.

Scalability of R-ACO versus ML-ACO

Figure 5.22 compares the scalability of reliable (R-ACO) and unreliable ACO (ML-ACO) up to 145 Erlang nodes. The measurements are run on the GPG cluster with 20 hosts, and the number of VMs per host varies according to the cluster size. 1 VM is dedicated for the master and sub-master processes and the other VMs run colonies. There are 4 ant processes

per colony. I_M and I_N (global and local iterations) are 20 and input size is 100. The scalability of R-ACO is worse than ML-ACO due to global name registration. As the cluster size grows, the runtime of R-ACO version increases with a considerably faster pace in comparison with the ML-ACO. This is because the latency of global name registration increases dramatically as cluster size grows [9]. Moreover, as discussed in Section 5.4.4, more global name registrations are required as the number of colony nodes increases. As a solution to this problem, we develop an SD-Erlang version of reliable ACO in the next section.

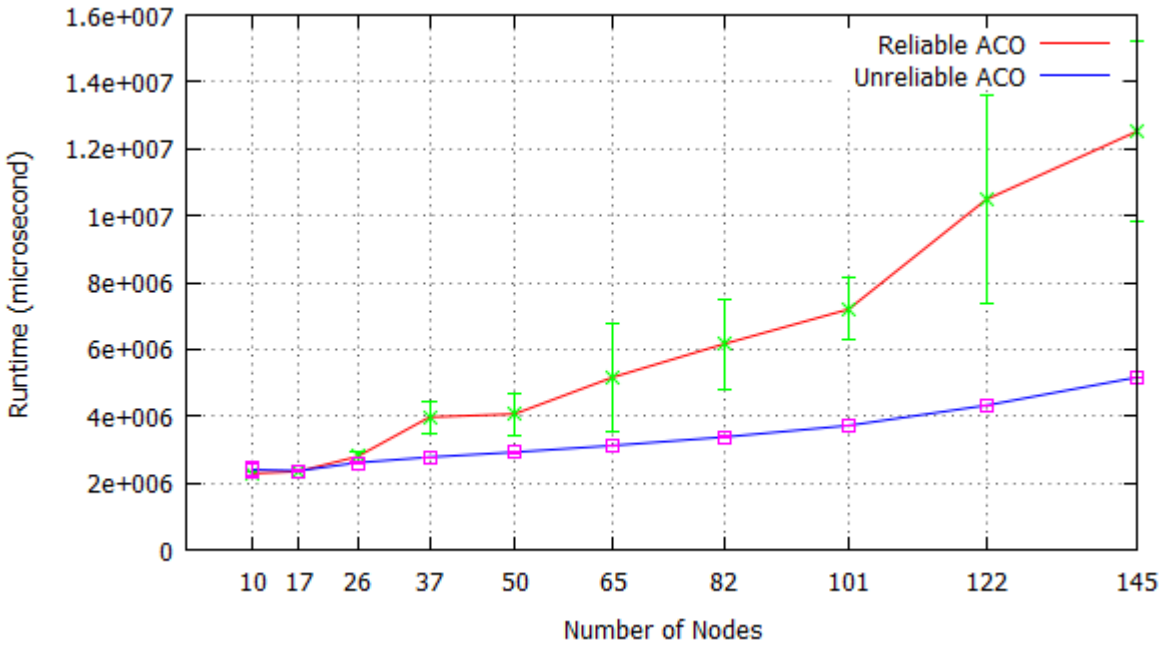


Figure 5.22: Scalability of Reliable vs. Unreliable ACO

5.4.5 Scalable Reliable ACO (SR-ACO)

This section investigates whether SD Erlang can improve the scalability of reliable distributed Erlang systems. We employ SD-Erlang to improve the scalability of reliable ACO by replacing global name registration with group name registration (Appendix C.6). In SR-ACO, global name registrations are replaced with group name registrations. In addition to this, SR-ACO has a non-fully connected model in which nodes are only connected to the nodes in their own group (*s_group*), and no unnecessary connection is made. In SR-ACO, nodes only need to communicate within their own group, and there is no need for a node to be connected to the nodes that belong to other groups. Figure 5.23 depicts s_groups organisation in SR-ACO. The master node belongs to all s_groups, however, colony nodes are grouped into a number of s_groups.

Figure 5.24 compares the weak scalability of scalable reliable (SR-ACO), reliable (R-ACO) and unreliable (ML-ACO) versions. The results show that SR-ACO scales much better than

R-ACO because of replacing global name registrations with s_group name registrations. Surprisingly, we see from the figure that SR-ACO scales even better than the unreliable ACO. Since there is no name registration in unreliable version, this better performance could be because of the reduction in the number of connections between the nodes. This shows that reducing the number of connections between nodes, could lead to a better scalability and performance. Next section investigates the impact of SD Erlang on the network traffic.

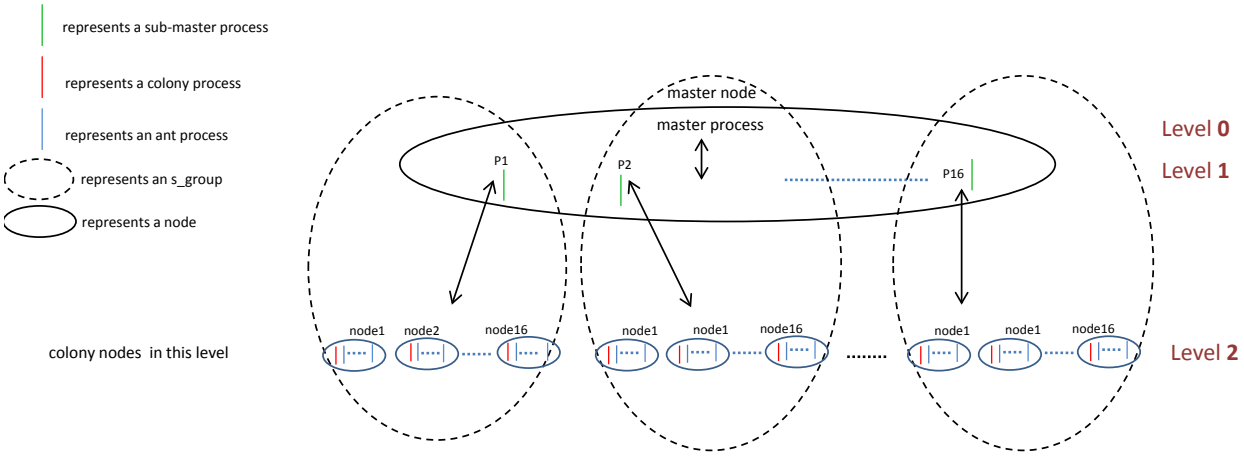


Figure 5.23: S_Groups Organisation in Scalable Reliable ACO (SR-ACO)

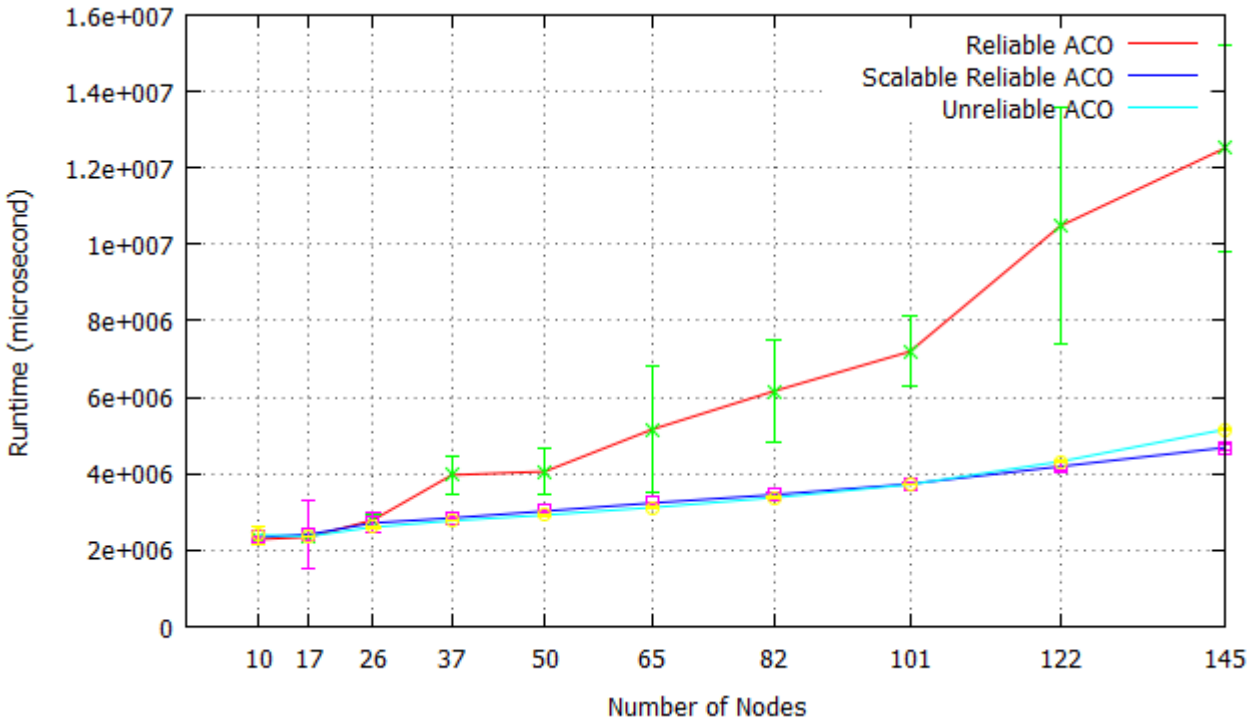


Figure 5.24: Weak Scalability of SR-ACO, R-ACO, and ML-ACO

5.4.6 Network Traffic

To investigate the impact of SD Erlang on the network traffic, we measure the number of sent and received packets for the three versions of ACO, i.e. ML-ACO, R-ACO, and SR-ACO. Figures 5.25 and 5.26 show the total number of sent and received packets for the measurement of Section 5.4.5 (Figure 5.24). The highest traffic (the red line) belongs to the reliable ACO (R-ACO) and the lowest traffic belongs to the scalable reliable ACO (SR-ACO). We observe that SD Erlang reduces the network traffic between the Erlang nodes in a cluster effectively. Even with the group name registration in SR-ACO, SD Erlang reduces the network traffic of SR-ACO more than unreliable ACO (ML-ACO) in which no global or group name registration is used.

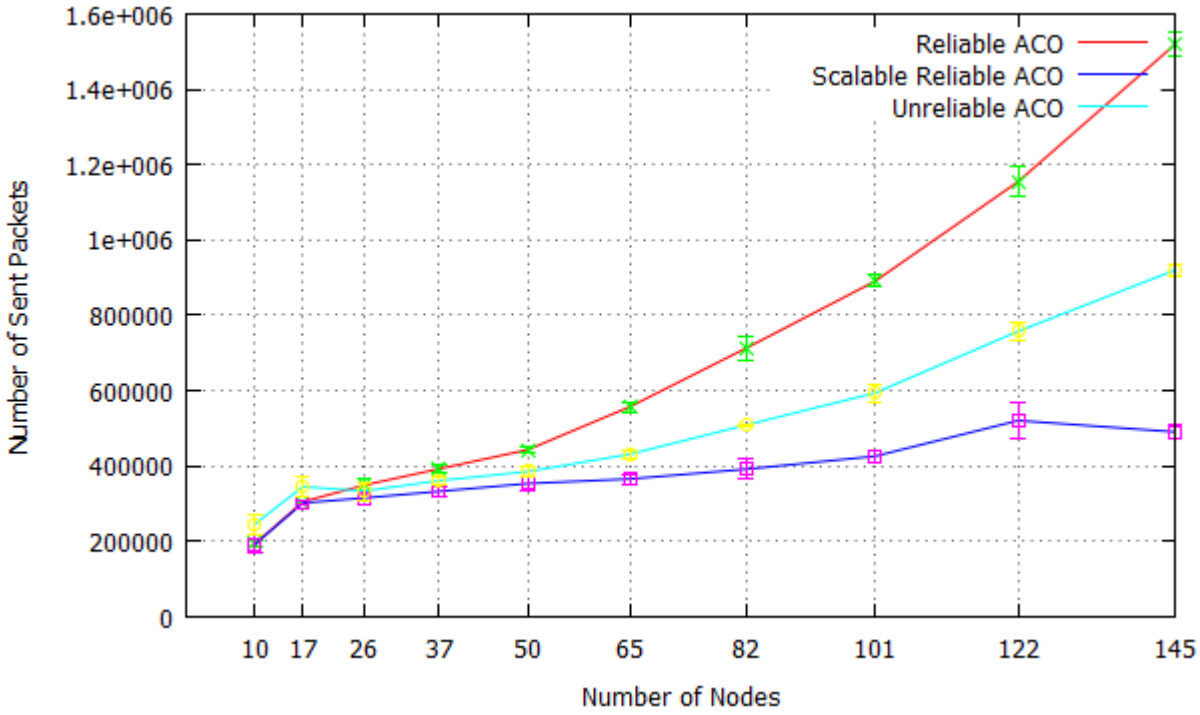


Figure 5.25: Number of Sent Packets in SR-ACO, Reliable ACO, and Unreliable ACO

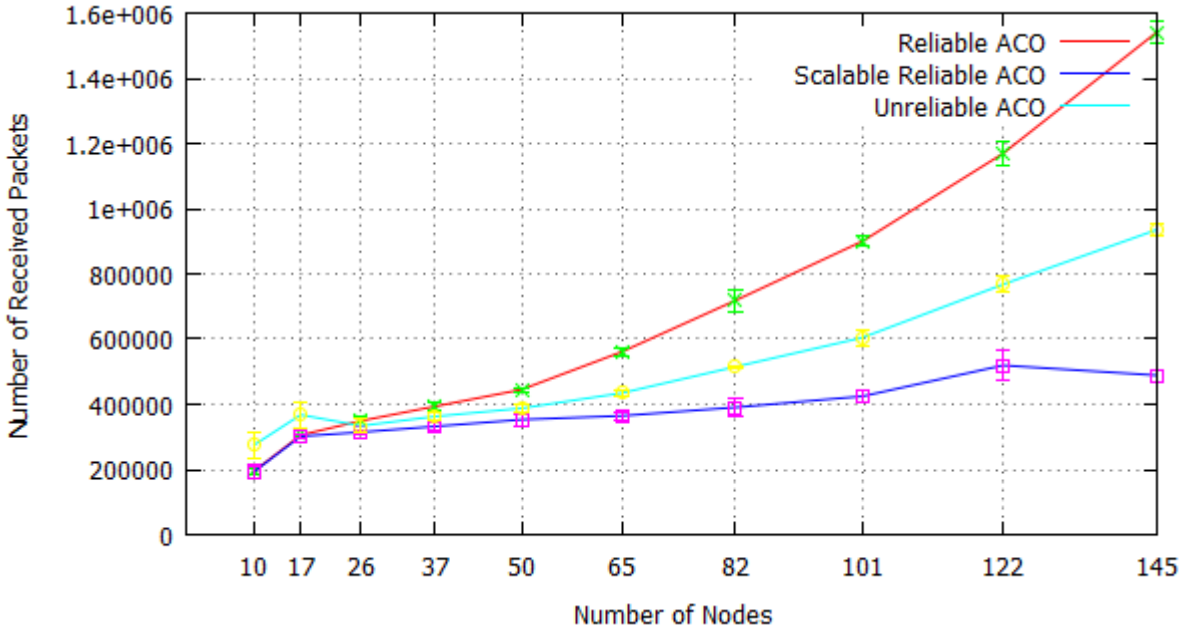


Figure 5.26: Number of Received Packets in SR-ACO, Reliable ACO, and Unreliable ACO

5.5 Discussion

This chapter investigates the potential of the new Scalable Distributed Erlang (SD Erlang) to improve the scalability of reliable distributed Erlang. It does so by develops the first ever SD Erlang benchmarks, i.e. DE-Bench, Orbit, and Ant Colony Optimisation. Our benchmarks cover the two most common models in distributed systems, i.e. P2P and master-worker models.

Orbit and DE-Bench have a P2P design where all nodes perform independently with no central coordination and synchronisation, whereas ACO has a master-worker model in which Colony nodes synchronize their activities with the master and sub-master nodes periodically, as discussed in Sections 5.2 and 5.3.

Section 5.2 has demonstrated that s_group name registration scales linearly and can perform properly as an alternative to global name registration at large scale (Figure 5.5).

Section 5.3 has demonstrated that SD Erlang improves the scalability of Orbit computation by reducing the number of connections between Erlang nodes in a cluster. We have employed a double-hashing technique to direct the communication between peers from different s_groups. We have shown in Figures 5.11 and 5.12 that SD Orbit scales better than distributed Obit on 160 nodes and 1280 cores.

We also developed a reliable version of distributed ACO and evaluated its reliability by employing Chaos Monkey (Section 5.4). We have seen that reliability limits the scalability of

ACO because of the use of global name registration (Figure 5.22). We alleviated the reliability cost by replacing global name registrations with group name registrations (Figure 5.24).

Section 5.4.6 investigates the network traffic by measuring the number of sent and received packets. The results reveal that SD Erlang reduces the network traffic efficiently as SR-ACO has the lowest network traffic even less than unreliable ACO (ML-ACO).

Chapter 6

Conclusion

6.1 Summary

With the advent of powerful commodity servers and high-speed networks, scalable architectures such as clusters, grids, and clouds are becoming popular by networking and aggregating the power of low-cost commodity servers to provide supercomputer performance.

To exploit such architectures where node and network failures are inevitable, reliable scalable programming languages are essential. Actor-based frameworks like Erlang are increasingly popular for developing scalable reliable distributed systems due to a combination of factors, including data immutability, share-nothing concurrency, asynchronous message passing, location transparency, and fault tolerance.

The RELEASE project aims to improve the scalability of Erlang on emergent commodity architectures with 10^4 cores. This thesis research as part of the RELEASE project investigates the scalability and reliability of two aspects of the language, i.e. *Persistent Data Storage* and *Distributed Erlang*.

Chapter 2 presents a critical review of existing and upcoming hardware platforms. We discuss the scalability and affordability of the platforms for massively parallel computing. Moreover, the RELEASE target architecture with consideration to scalability and availability is presented in Section 2.1. Parallel programming models and their suitability and limitations for different hardware platforms are reviewed in Section 2.2. Challenges and benefits of distributed systems at large-scale computing architectures are discussed in Section 2.3. The scalability and reliability of common architectural models and communication paradigms for distributed systems are also given in this section. Erlang light-weight concurrency for development of scalable software and its popularity for fault-tolerance are discussed in Section 2.4. The section also reviews OTP behaviours, location transparency, and security in the language.

Chapter 3 investigates the provision of scalable persistent storage for Erlang in theory and practice. Common techniques and solutions to handle vast amounts of information are studied in Section 3.1. The requirements for scalable and available persistent storage are enumerated (Section 3.2), and four popular Erlang DBMSs are evaluated against these requirements (Section 3.3). Our evaluation shows that Mnesia and CouchDB do not provide suitable persistent storage at our target scale, but Dynamo-style NoSQL DBMSs such as Cassandra and Riak potentially do. The scalability limits of the Riak 1.1.1 NoSQL DBMS is investigated practically up to 100 nodes (Section 3.4). We establish for the first time scientifically the scalability limit of Riak as 60 nodes on the Kalkyl cluster, thereby confirming developer folklore (Figure 3.13). We show that resources like memory, disk, and network do not limit the scalability of Riak (Section 3.4.4). By instrumenting Erlang/OTP and Riak libraries we identify a specific Riak functionality that limits scalability. We outline how later releases of Riak are refactored to eliminate the scalability bottlenecks (Section 3.4.5). The availability and elasticity of Riak is measured in Section 3.4.6. We show that a cluster of Riak nodes continues to work gracefully when 50% of the nodes in the cluster fail. Riak shows a good elasticity and recovers its initial throughput after the failed nodes return back to the cluster (Figure 3.22). We conclude that Dynamo-style NoSQL DBMSs provide scalable and available persistent storage for Erlang in general, and for our RELEASE target architecture in particular (Section 3.4.7).

Chapter 4 presents DE-Bench, a scalable parameterized fault-tolerant benchmarking tool that measures the throughput and latency of distributed Erlang commands on a cluster of Erlang nodes (Section 4.2). We investigate the scalability limits of distributed Erlang up to 150 nodes by employing DE-Bench. Our benchmarking results demonstrate that the frequency of global commands limits the scalability of distributed Erlang (Figure 4.5). We show that distributed Erlang scales linearly up to 150 nodes and 1200 cores with relatively heavy data and computation loads when no global commands are used (Figure 4.9). Measuring the latency of commonly-used distributed Erlang commands reveals that the latency of *rpc* calls rises as cluster size grows. Our results also show that server processes like *gen_server* and *gen_fsm* have low latency and good scalability (Section 4.3).

Chapter 5 evaluates the scalability of improved distributed Erlang (SD Erlang). Firstly, a brief introduction on SD Erlang is given (Section 5.1). DE-Bench is employed to evaluate the s_group name registration up to 100 nodes (Section 5.2). We compare global name registration with group name registration and show that group name registration scales linearly up to 100 nodes (Figure 5.5) .

An SD Erlang version of the *Orbit* computation is designed, implemented, and evaluated in Section 5.3. We use a double hashing technique to route communication through gateway nodes to reduce the connections between peers belong to different s_groups. A comparison between the scalability of SD Erlang and Distributed Erlang Orbits up to 160 nodes on the

Kalkyl cluster is given in Figures 5.11 and 5.12. We demonstrate that SD Erlang Orbit outperforms the distributed Orbit on 160 nodes and 1280 cores.

We also develop a reliable distributed version of Ant Colony Optimisation (ACO), a technique that aims to arrange the sequence of jobs in such a way as to minimise the total weighted tardiness (Section 5.4). To evaluate the reliability of ACO, we employ Chaos Monkey. We show that reliability limits the scalability of ACO in traditional distributed Erlang (Figure 5.22). We alleviate the cost of reliability by employing techniques that SD Erlang offers for reducing the cost of name registration and mesh connectivity (Figure 5.24). Investigating the network performance shows that SD Erlang reduces the network traffic effectively (Section 5.4.6).

6.2 Limitations

In Chapter 3 we theoretically conclude that Dynamo-style DBMSs can provide scalable and available persistent storage for Erlang. We also measure the scalability of Riak 1.1.1 up to 100 nodes practically. However, measuring the scalability of the other DBMSs practically is out of the scope of the project and left as future work.

As Section 3.4.5 states, Basho has applied a number of techniques in the Riak versions 1.3 and 1.4 to improve scalability. Investigating the scalability of the new versions is out of the project schedule and remains as future work.

In Chapter 4 we investigate the scalability limits of distributed Erlang using DE-Bench up to 150 nodes and 1200 cores. We planned to run our measurement on the Blue Gene/Q computing architecture at larger scales but as distributed Erlang is not supported at Blue Gene (TCP/IP is not supported on the compute cards in the Blue Gene/Q and typical HPC applications use MPI for inter-node communication over the Blue Gene's very high-speed interconnects), we could not measure the scalability of distributed Erlang and SD Erlang beyond 150 nodes and 1200 cores.

While we have used several benchmarks to show that SD-Erlang improves the scalability of reliable distributed applications, we have not used it to improve the scalability of an existing real-world distributed Erlang application. We planned to employ SD-Erlang to improve the scalability of *Sim-Diasca*, a lightweight simulation platform, released by EDF R&D under the GNU LGPL licence. Sim-Diasca, which stands for *Simulation of Discrete Systems of All Scales*, offers a set of simulation elements, including a simulation engine, to simulate discrete event-based systems [1]. Our measurement shows that Sim-Diasca has some scalability issues (Appendix D.1). Until these scalability issues are solved, SD-Erlang cannot improve the scalability of Sim-Diasca because the issues are due to some problems in the design and implementation of Sim-Diasca and are not related to the programming language.

6.3 Future Work

Basho Bench has been developed to benchmark the Riak NoSQL DBMS, but it exposes a pluggable driver interface that makes it possible to be used as a benchmarking tool for other NoSQL DBMSs [91]. It can be used to evaluate the scalability, reliability, and elasticity of the other NoSQL DBMSs discussed in Chapter 3, i.e Mnesia, Cassandra, and CouchDB.

The idea of benchmarking distributed Erlang using DE-Bench can be extended to the other actor based frameworks such as AKKA [115]. Akka written in Scala is an open-source toolkit and runtime for building highly concurrent, distributed, and fault tolerant applications on the JVM. Akka handles concurrency based on the Actor Model inspired by the Erlang programming language. All DE-Bench's modules, except the *de_commands.erl* module, can be rewritten in AKKA with minimum expected changes. The module *de_commands.erl* contains distributed Erlang commands that should be replaced with distributed commands in AKKA [98].

The communication patterns we have used in our benchmarks, i.e. hierarchical tree structure in Ant Colony Optimisation and communication through gateways in P2P Orbit computation, can be generalized as reusable and common solutions for other SD Erlang applications. Generalizing these two categories of design patterns, with other possible patterns, can cover communication patterns in a considerable number of Erlang applications. Providing such general design patterns in future would persuade the Erlang community to employ SD Erlang with less possible effort to scale their distributed applications.

As Section 6.2 states, Sim-Diasca is a real-world distributed Erlang application that we consider as a case-study for SD Erlang. The following is a suggested design for creating an SD Erlang version of Sim-Diasca [2].

The *Time Manager* processes are one of the main components that impact the scalability of Sim-Diasca. The Time Manager processes are used to schedule the events in a simulation and to ensure uniform simulation time across all participating actors in the simulation. The Time Manager processes are organized in a hierarchical structure in a cluster. There is a Cluster Root Time Manager that spawns one Root Time Manager at all nodes in the cluster. The Root Time Manager at a node spawns a number of local Time Managers (Figure 6.1).

Figure 6.2 presents an SD Erlang design for grouping the time manager processes in Sim-Diasca. In the suggested design, a Time Manager process belongs to two groups, i.e. a group of its parent and siblings and also a group of its children. This approach reduces the connections and shared namespaces between nodes.

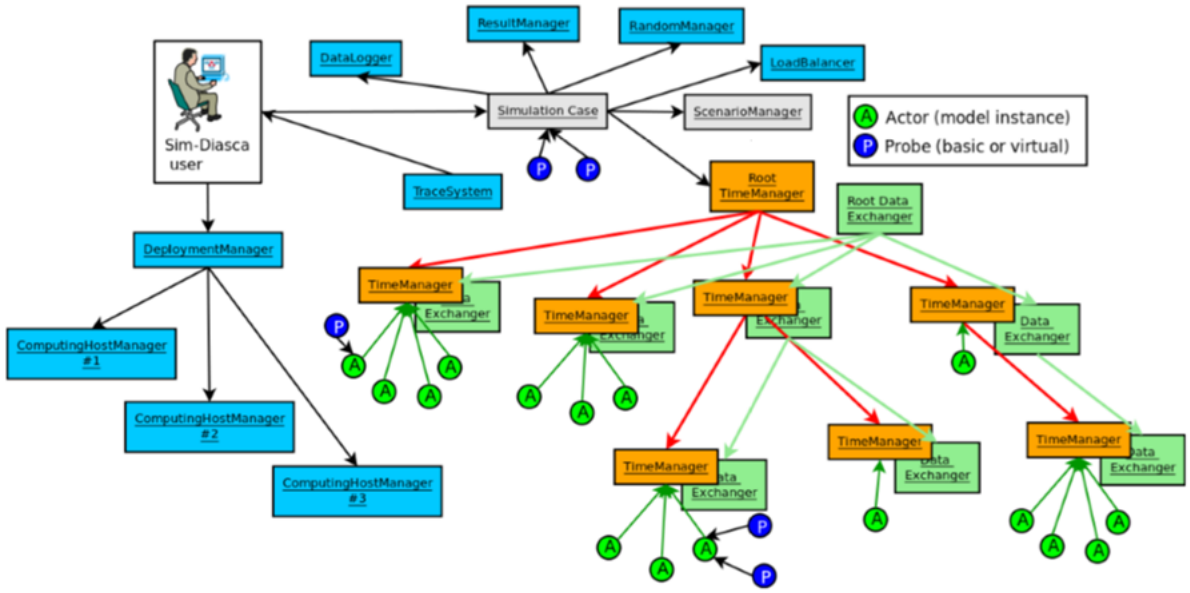


Figure 6.1: Hierarchical Structure of the Time Manager Processes in Sim-Diasca [1]

Figure 6.2: Time Manager Processes in the SD Erlang version of Sim-Diasca [2]

Appendix A

A.1 Comparing Riak Client Interfaces

There are two ways for communicating with a Riak node: *HTTP interface* and *Protocol Buffers interface* [68]. Figure A.1 compares both of theses interfaces on a 20-node Riak cluster with 7 traffic generators for 30 minutes. We run this benchmark three times and Figure A.1 represents the median of three executions. The result shows that Protocol Buffers is approximately 2 times faster than HTTP. Protocol Buffers is a binary protocol and in comparison with HTTP, request and response messages are more compact. On the other hand, the HTTP protocol is more feature-complete, i.e. setting bucket property and support secure connections (HTTPS). HTTP is also more familiar to developers and simpler for debugging. Moreover, HTTP is web-based protocol and clients are able to connect to the Riak nodes through firewalls and proxy servers.

A.2 Comparing Bitcask with LevelDB

In Riak, storage backends are pluggable and this allows us to choose storage engine that suits our requirement. *Bitcask* and *LevelDB* are two common persistent backends in Riak [68]. Figure A.2 compares the performance of these two backends on a 20-node Riak cluster with 7 traffic generators. We see from the figure that LevelDB's throughput is 1.1 more than Bitcask throughput. We repeat our measurement three times and the median values are presented in the figure.

Bitcask keeps all keys in memory and a direct lookup from an in-memory hash table point directly to locations on disk where the data lives and just one seek on disk is needed to retrieve any value [68]. On the other hand, keeping all keys in memory could be a limitation because system must have enough memory to host all the keys. In this benchmark, the keys are integer values and can be accommodated in RAM as they are not too large. The LevelDB backend is considered for systems with large number of keys.

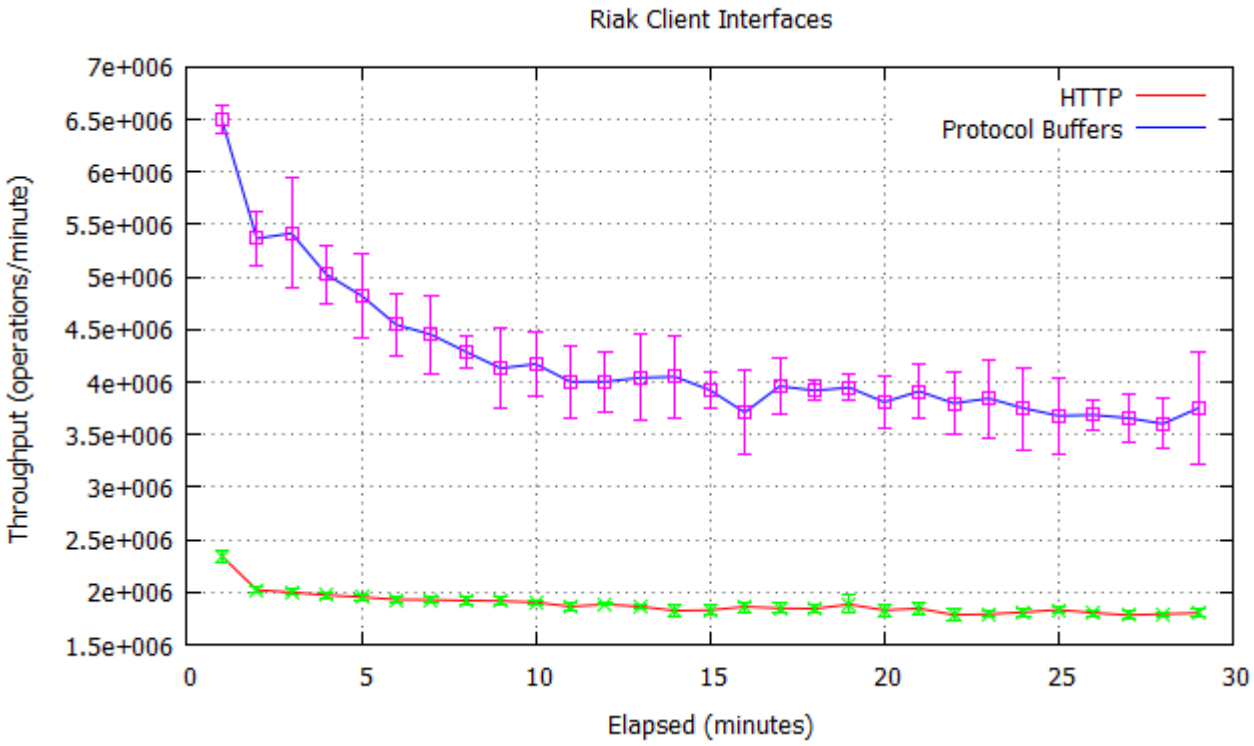


Figure A.1: Comparing Riak Client Interfaces (20 Riak nodes, 7 traffic generators)

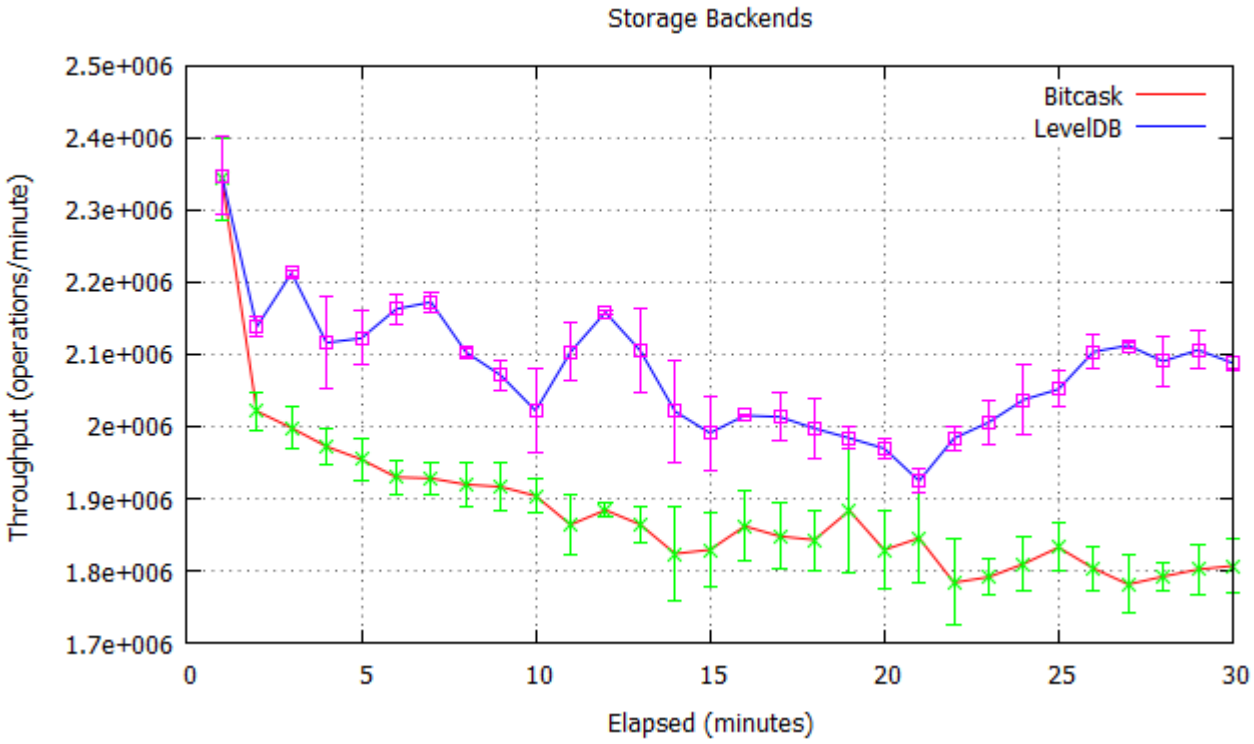


Figure A.2: Comparing Riak Storage Backends (20 Riak nodes, 7 traffic generators)

A.3 The Most Time-Consuming Riak *gen_server* Calls

To investigate the Riak source code, we profiled all the *gen_server* calls that Riak 1.1.1 makes. Table A.1 shows the 15 most time-consuming *gen_server* calls in Riak. There are two rows for each function: T_{mean} is the average time that each *gen_server:call* takes to complete (in microsecond) and N_{mean} shows the number of times that the function is called during the 5 minutes benchmarking. We see that the average time for the function *call* from module *rpc* increases proportionally when the size of cluster grows (marked by yellow bar in Table A.1).

In the next step, we narrow our investigation to function which are called through *rpc*. We aim to find out the most time-consuming function that are called by *rpc*. Table A.2 presents all the functions that are called through *rpc*. There are two rows for each function in the table, i.e. T_{mean} that shows the average time that each *rpc* call takes to complete (in microsecond) and N_{mean} that shows the number of times that a function is called during the 5 minutes benchmark. The results reveal that the latency of *rpc* call on function *start_put_fsm* grows when the size of cluster increases (marked by yellow bar in TableA.2). We see from the table that the call-time for function *start_put_fsm* grows from 2,019 microseconds on a 4-node cluster to 13,461 microseconds on a 24-node cluster. The *get* method from module *riak_core_capability* is called locally and its low call-time is because of this fact (marked by pink bar in the table).

Module	Function		Runtime on a cluster of			
			4 nodes	8 nodes	16 nodes	24 nodes
file_server_2	list_dir	T_{mean}	5210	3619	7265	10711
		N_{mean}	5127	5086	5137	3473
file_server_2	del_dir	T_{mean}	3395	2790	6208	12041
		N_{mean}	851	843	845	563
file_server_2	rename	T_{mean}	4261	2568	5450	9186
		N_{mean}	2553	2529	2535	1689
riak_core_vnode_manager	all_vnodes	T_{mean}	88814	85940	75983	74421
		N_{mean}	1810	1926	2023	1832
file_server_2	write_file	T_{mean}	4996	1831	3428	6213
		N_{mean}	1720	1711	1754	1218
riak_core_ring_manager	set_my_ring	T_{mean}	26874	28694	30895	58205
		N_{mean}	1	1	1	1
file_server_2	read_file_info	T_{mean}	2380	1408	2768	4357
		N_{mean}	6883	6827	6890	4662
file_server_2	open	T_{mean}	1973	1185	2559	3853
		N_{mean}	1706	1690	1694	1130
riak_core_stats_sup	start_child	T_{mean}	13816	16614	16933	16598
		N_{mean}	3	3	3	3
rpc	call	T_{mean}	1953	2228	8547	15180
		N_{mean}	106591	77468	49088	31104
riak_core_capability	register	T_{mean}	14321	13311	12783	13587
		N_{mean}	7	7	7	7
lager_handler_watcher_sup	start_child	T_{mean}	6002	6919	5501	8736
		N_{mean}	4	4	4	4
net_kernel	connect	T_{mean}	24613	11854	14303	10109
		N_{mean}	3	6	6	15
riak_core_ring_manager	ring_trans	T_{mean}	527	639	1844	5532
		N_{mean}	1476	1476	1667	1355
riak_kv_js_sup	start_child	T_{mean}	4648	4745	4562	5388
		N_{mean}	16	16	16	16

Table A.1: The 15 most time-consuming Riak *gen_server* calls

Module	Function		Runtime on a cluster of			
			4 nodes	8 nodes	16 nodes	24 nodes
riak_kv_put_fsm_sup	start_put_fsm	T_{mean}	2019	2284	8342	13461
		N_{mean}	104973	82070	48689	33898
riak_core_ring_manager	get_my_ring	T_{mean}	1045	813	1037	1080
		N_{mean}	1	1	1	1
riak_core_ring_manager	get_raw_ring	T_{mean}	889	983	1032	910
		N_{mean}	1	1	1	1
riak_core_gossip	legacy_gossip	T_{mean}	794	650	739	676
		N_{mean}	1	1	1	1
riak_core_capability	get (local)	T_{mean}	3	3	4	4
		N_{mean}	210001	164178	96782	68692

Table A.2: All *rpc* calls in Riak

Appendix B

B.1 DE-Bench Configuration File

Code 20 presents a sample configuration file for DE-Bench, a parametrized benchmarking tool that measures the throughput and latency of distributed Erlang commands on a cluster of Erlang nodes. The following gives a brief description of each of the configurable features:

- *duration*: the duration of the benchmark in minutes.
- *concurrent*: the number of concurrent worker processes on each DE-Bench node.
- *report_interval*: how often the benchmarking data should be recorded in *CSV* files (in second).
- *erlange_nodes*: list of Erlang nodes participating in the benchmark.
- *operations*: list of distributed Erlang commands using in the benchmark. The commands including *spawn*, *rpc*, *global name registration*, *global name unregisteration*, *global and local whereis*, *gen_server*, and *gen_fsm*. The percentage of each command can be specified as a tuple with the format of $\{Command,\ Percentage\}$.
- *sleep_time_before_ping*: a delay to make sure cluster of Erlang nodes becomes stable before starting the benchmark.
- *sleep_time_after_ping*: a delay to make sure DE-Bench is loaded on all participating nodes before starting the benchmark.
- *packet_siz*: the data size that is used in point-to-point commands, i.e *spawn*, *rpc*, and server process calls such as *gen_server* and *gen_fsm*.
- *delay_remote_function*: how long a remote function takes in microsecond. This is the function which is called by *spawn*, *rpc*, *gen_server* and *gen_fsm* remotely.

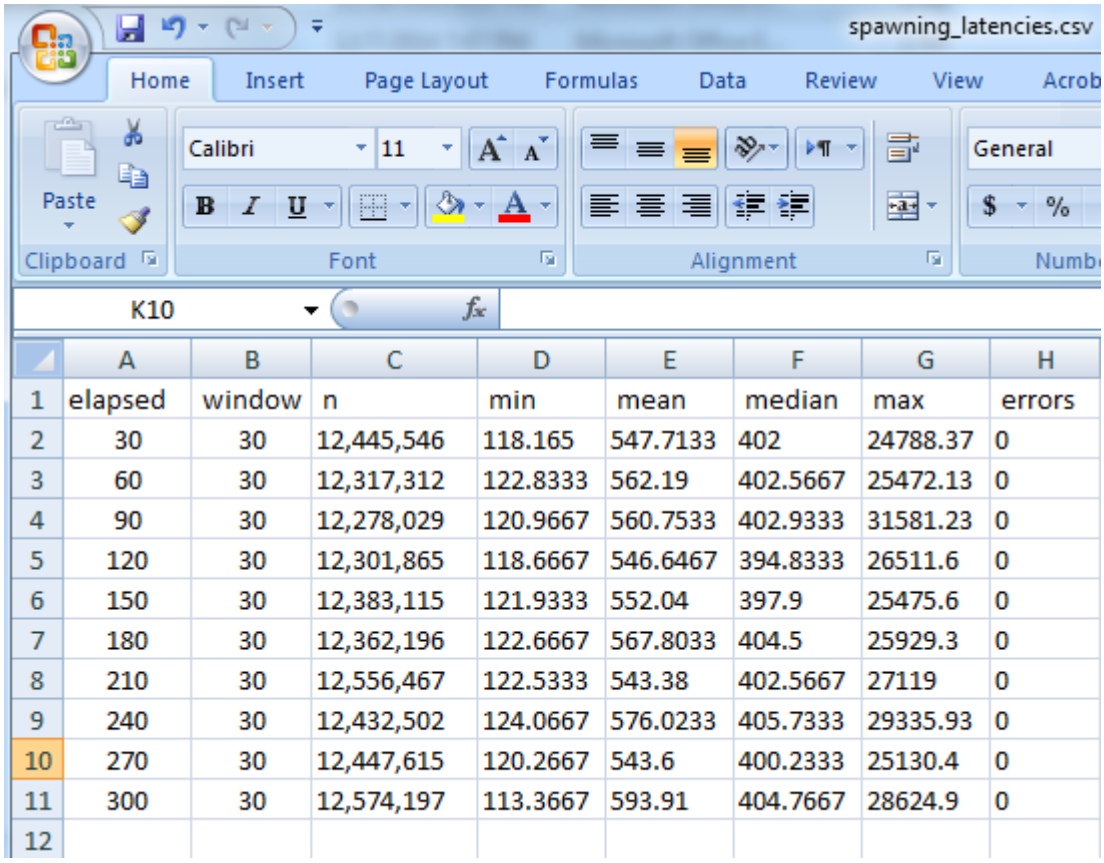
Code 20: DE-Bench Configuration File

```
{duration, minutes}.

{concurrent, number_of_worker_processes}.

{report_interval, report_interval_seconds}.

{erlange_nodes, [Here_put_VMs_names]}.

{operations, [{command1, command1_percentage} ,
{command2, command2_percentage}, ...]}.

{sleep_time_after_ping, sleep_time_after_ping_here}.

{sleep_time_before_ping, sleep_time_before_ping_here}.

{packet_size, packet_size_here}.

{delay_remote_function, delay_remote_function_here}.
```

B.2 Sample CSV Files Generated by DE-Bench

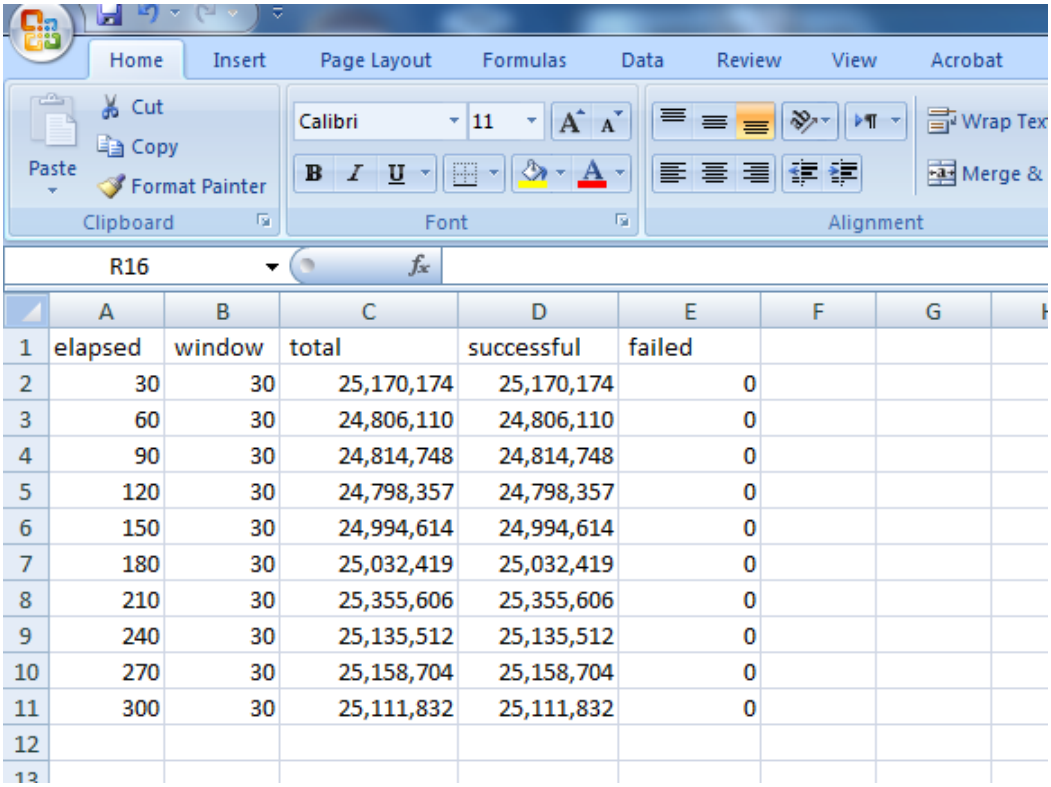
DE-Bench records the latency of each command in individual CSV files. Figure B.1 shows a sample file that records the latency of the *spawn* command during 5 minutes benchmarking. The columns *elapsed* and *window* show that each 30 seconds one record of statistical information is recorded (during 5 minutes benchmarking). The column *n* is the number of total *spawn* command run in each period of time (30 seconds in this case). The columns *min*, *mean*, *median*, and *max* show the minimum, average, middle, and the maximum latency recorded for the *spawn* command in each period of time (30 seconds in this case). The column *error* shows the number of failures that has happened in each period. There is no failure in our measurement.

Figure B.2 presents the total number of all commands used in the benchmark. It also shows the successful and failed commands in different columns.



	A	B	C	D	E	F	G	H
1	elapsed	window	n	min	mean	median	max	errors
2	30	30	12,445,546	118.165	547.7133	402	24788.37	0
3	60	30	12,317,312	122.8333	562.19	402.5667	25472.13	0
4	90	30	12,278,029	120.9667	560.7533	402.9333	31581.23	0
5	120	30	12,301,865	118.6667	546.6467	394.8333	26511.6	0
6	150	30	12,383,115	121.9333	552.04	397.9	25475.6	0
7	180	30	12,362,196	122.6667	567.8033	404.5	25929.3	0
8	210	30	12,556,467	122.5333	543.38	402.5667	27119	0
9	240	30	12,432,502	124.0667	576.0233	405.7333	29335.93	0
10	270	30	12,447,615	120.2667	543.6	400.2333	25130.4	0
11	300	30	12,574,197	113.3667	593.91	404.7667	28624.9	0
12								

Figure B.1: A sample CSV file that shows the latency of the *spawn* command



	A	B	C	D	E	F	G
1	elapsed	window	total	successful	failed		
2	30	30	25,170,174	25,170,174	0		
3	60	30	24,806,110	24,806,110	0		
4	90	30	24,814,748	24,814,748	0		
5	120	30	24,798,357	24,798,357	0		
6	150	30	24,994,614	24,994,614	0		
7	180	30	25,032,419	25,032,419	0		
8	210	30	25,355,606	25,355,606	0		
9	240	30	25,135,512	25,135,512	0		
10	270	30	25,158,704	25,158,704	0		
11	300	30	25,111,832	25,111,832	0		
12							

Figure B.2: A sample CSV file that shows the total number of commands

Appendix C

C.1 Distributed Orbit Source Code

This section contains the source code of distributed Orbit developed by Dr Patrick Maier from Glasgow University. The four main modules, i.e. *credit*, *master*, *worker*, and *table* are listed as Codes C.1, C.2, C.3, and C.4 respectively. All modules and deployment scripts are publicly available at [116].

```
%% orbit-int credits (for checking termination of orbit computation)
%%
%% Author: Patrick Maier <P.Maier@hw.ac.uk>
%%

-module(credit).

-export([credit/2, credit_atomic/2, debit_atomic/1, debit_atomic_nz/1,
         zero/0, one/0, is_zero/1, is_one/1]).

%% An *atomic credit* is represented as a non-negative integer k;
%% it stands for the credit 1/{2^k}.
%%
%% A *credit* is represented as list of non-negative integers, sorted in
%% strict descending order; it represents the sum of atomic credits
%% represented by the integers on the list, where zero credit is
%% represented by the empty list. The maximally possible credit, 1,
%% is represented by [0].


%% credit_atomic(K, C) adds the atomic credit 1/{2^K} to the credit C.
credit_atomic(K, [])     -> [K];
credit_atomic(K, [C|Cs]) -> if
                              K > C  -> [K, C | Cs];
                              K == C -> credit_atomic(K - 1, Cs);
                              K < C  -> [C | credit_atomic(K, Cs)]
                            end.

%% credit(C1, C2) returns a list representing the sum of the credit
%% represented by the lists C1 and C2.
credit(C1, C2) -> lists:foldl(fun credit_atomic/2, C2, C1).
% Alternative fomulation:
% credit([],     C2) -> C2;
% credit([K|Ks], C2) -> credit(Ks, credit_atomic(K, C2)).

%% debit_atomic(C) returns a pair {K',C'} where K' is an integer
%% representing some atomic credit and C' is a list of integers representing
%% some credit (which may be zero) such that the sum of the credits
%% represented by K' and C' equals the credit represented by C.
%% Precondition: C must represent non-zero credit.
debit_atomic([C|Cs]) -> {C, Cs}.  %% debit smallest unit of credit

%% debit_atomic_nz(C) returns a pair {K',C'} where K' is an integer
%% representing some atomic credit and C' is a list of integers representing
%% some non-zero credit such that the sum of the credits
%% represented by K' and C' equals the credit represented by C.
%% Precondition: C must represent non-zero credit.
debit_atomic_nz([C])    -> {C+1, [C+1]};  %% debit half the credit
debit_atomic_nz([C|Cs]) -> {C,   Cs}.     %% debit smallest unit of credit;
                                          %% case only applies if Cs non-empty


%% zero/0 produces zero credit.
zero() -> [].

%% one/0 produces credit one.
one() -> [0].

%% is_zero/1 tests whether its argument represents zero credit.
is_zero([]) -> true;
is_zero(_)  -> false.

%% is_one/1 tests whether its argument represents maximal credit 1.
```

```
is_one([0]) -> true;
is_one(_)   -> false.
```

Code C.1: Credit Module (credit.erl)

```
%% orbit-int master (controlling orbit computation)
%%
%% Author: Patrick Maier <P.Maier@hw.ac.uk>
%%

-module(master).

-export([orbit/3,
         get_gens/1, get_master/1, get_workers/1, get_spawn_img_comp/1,
         get_global_table_size/1, get_idle_timeout/1,
         set_idle_timeout/2, clear_spawn_img_comp/1,
         now/0
         ]).

-compile({no_auto_import, [now/0]}).

%% DATA
%%   Static Machine Configuration:
%%     {Gs,               %list of generators
%%      Master,           %pid of master process
%%      Workers,          %list of Worker
%%      GlobalTableSize,  %size of global hash table
%%      IdleTimeout,      %milliseconds this worker idles before sending 'done'
%%      SpawnImgComp}     %true iff this worker spawns image computations
%%
%%   Worker:
%%     {Pid,              %pid of worker process
%%      TableOffset,      %offset (= index 0) of local table into global table
%%      TableSize}        %size of local hash table
%%
%%   Host:
%%     {Node,             %atom naming Erlang node
%%      Procs,            %number of processors
%%      TableSize,        %size of hash table per processor
%%      IdleTimeout}      %milliseconds a processor idles before sending 'done'
%%
%%   Statistics:
%%     List of pairs where the first component is an atom, the second
%%     some data. Part of the data is the fill frequency of the table
%%     (a list whose ith element indicates frequency of filling degree i).


%% MESSAGES
%%   Master -> Worker:        {init, StaticMachConf}
%%
%%   Master/Worker -> Worker: {vertex, X, Slot, K}
%%                               %X is vertex
%%                               %Slot is slot of X on target worker
%%                               %K is atomic credit shipped with vertex
%%
%%   Worker -> Master:        {done, Cs}
%%                               %Cs is non-zero credit (rep as list of ints)
%%
%%   Master -> Worker:        {dump}
%%
%%   Worker -> Master:        {result, Xs, Stats}
%%                               %Xs is list of found orbit vertices
%%                               %Stats is statistics about worker's table


%% compute orbit of elements in list Xs under list of generators Gs;
%% the argument Hosts is either an integer N, a triple {P, N, T}, or
```

```
%% a non-empty list [{H, P, N, T} | ...] of quadruples:
%% * N:                          run the sequential algorithm with table size N
%% * {P, N, T, S}:               run the parallel algorithm on P processors
%%                               each with table size N, idle timeout T and
%%                               spawn image computation flag S;
%% * [{H, P, N, T, S} | ...]: run the distributed algorithm on the list of
%%                               hosts, where each quintuple {H, P, N, T, S}
%%                               specifies
%%                               * host name H (ie. name of Erlang node),
%%                               * number of processors P on H,
%%                               * table size N (per processor),
%%                               * idle timeout T, and
%%                               * spawn image computation flag S.
%% The function returns a pair consisting of the computed orbit and
%% a list of statistics, the first element of which reports overall statistics,
%% and all remaining elements report statistics of some worker.
orbit(Gs, Xs, Hosts) ->
  if
    is_integer(Hosts) ->
      TableSize = Hosts,
      sequential:orbit(Gs, Xs, TableSize);
    true ->
      par_orbit(Gs, Xs, Hosts)
  end.

par_orbit(Gs, Xs, Hosts) ->
  % spawn workers on Hosts
  {Workers, GlobTabSize} = start_workers(Hosts),

  % assemble StaticMachConf and distribute to Workers
  StaticMachConf = mk_static_mach_conf(Gs, self(), Workers, GlobTabSize),
  lists:foreach(fun({Pid, _, _}) -> Pid ! {init, StaticMachConf} end, Workers),

  % start wall clock timer
  StartTime = now(),

  % distribute initial vertices to workers
  Credit = worker:distribute_vertices(StaticMachConf, credit:one(), Xs),

  % collect credit handed back by idle workers
  collect_credit(Credit),

  % measure elapsed time (in milliseconds)
  ElapsedTime = now() - StartTime,

  % tell all Workers to dump their tables
  lists:foreach(fun({Pid, _, _}) -> Pid ! {dump} end, Workers),

  % collect results from all workers and return them
  collect_orbit(ElapsedTime, length(Workers)).


%% start_workers starts worker processes depending on the input Hosts:
%% * if Hosts is a quadruple {P, _, _, _} then P processes are forked on the
%%   executing Erlang node;
%% * if Hosts is a non-empty list {H1, P1, _, _, _}, {H2, P2, _, _, _}, ...
%%   then P1 processes are forked on Erlang node H1, P2 processes on node H2,
%%   and so on.
%% The function returns a pair {Workers, GlobalTableSize}, where
%% * GlobalTableSize is the total number of slots of the global hash table, and
%% * Workers is a list of Worker, sorted wrt. TableOffset in ascending order.
start_workers({Procs, TabSize, TmOut, SpawnImgComp}) ->
```

```
  {Workers, GlobalTableSize} = do_start_shm({Procs, TabSize, TmOut, SpawnImgComp}, {[], 0}),
  {lists:reverse(Workers), GlobalTableSize};
start_workers([Host | Hosts]) ->
  {Workers, GlobalTableSize} = do_start_dist([Host | Hosts], {[], 0}),
  {lists:reverse(Workers), GlobalTableSize}.

do_start_shm({0, _, _, _}, Acc) ->
  Acc;
do_start_shm({M, TabSize, TmOut, SpawnImgComp}, {Workers, GTabSize}) ->
  Pid = spawn_link(worker, init, [TabSize, TmOut, SpawnImgComp]),
  NewWorkers = [{Pid, GTabSize, TabSize} | Workers],
  NewGTabSize = GTabSize + TabSize,
  Acc = {NewWorkers, NewGTabSize},
  do_start_shm({M - 1, TabSize, TmOut, SpawnImgComp}, Acc).

do_start_dist([], Acc) ->
  Acc;
do_start_dist([{_, 0, _, _, _} | Hosts], Acc) ->
  do_start_dist(Hosts, Acc);

do_start_dist([{Node, M, TabSize, TmOut, SpawnImgComp} | Hosts], {Workers, GTabSize}) ->
  Pid = spawn_link(Node, worker, init, [TabSize, TmOut, SpawnImgComp]),
  NewWorkers = [{Pid, GTabSize, TabSize} | Workers],
  NewGTabSize = GTabSize + TabSize,
  Acc = {NewWorkers, NewGTabSize},
  do_start_dist([{Node, M - 1, TabSize, TmOut, SpawnImgComp} | Hosts], Acc).

%% collect_credit collects leftover credit from idle workers until
%% the credit adds up to 1.
collect_credit(Credit) ->
  case credit:is_one(Credit) of
    true  -> ok;  %% break loop and return dummy atom
    _Else ->
      receive
        {done, WorkersCredit} ->
          CollectedCredit = credit:credit(WorkersCredit, Credit),
          collect_credit(CollectedCredit)
      end
  end.


%% collect_orbit collects partial orbits and stats from N workers.
collect_orbit(ElapsedTime, N) ->
  {PartOrbits, WorkerStats} = do_collect_orbit(N, [], []),
  Orbit = lists:flatten(PartOrbits),
  Stats = [master_stats(ElapsedTime, WorkerStats) | WorkerStats],
  {Orbit, Stats}.

do_collect_orbit(0, PartOrbits, WorkerStats) -> {PartOrbits, WorkerStats};
do_collect_orbit(N, PartOrbits, WorkerStats) ->
  receive
    {result, PartOrbit, WorkerStat} ->
      do_collect_orbit(N - 1, [PartOrbit|PartOrbits], [WorkerStat|WorkerStats])
  end.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% auxiliary functions

%% functions operating on the StaticMachConf
mk_static_mach_conf(Gs, Master, Workers, GlobalTableSize) ->
   {Gs, Master, Workers, GlobalTableSize, 0, true}.
```

```
get_gens(StaticMachConf)              -> element(1, StaticMachConf).
get_master(StaticMachConf)            -> element(2, StaticMachConf).
get_workers(StaticMachConf)           -> element(3, StaticMachConf).
get_global_table_size(StaticMachConf) -> element(4, StaticMachConf).
get_idle_timeout(StaticMachConf)      -> element(5, StaticMachConf).
get_spawn_img_comp(StaticMachConf)    -> element(6, StaticMachConf).
set_idle_timeout(StaticMachConf, X)  -> setelement(5, StaticMachConf, X).
clear_spawn_img_comp(StaticMachConf) -> setelement(6, StaticMachConf, false).


%% produce readable statistics
master_stats(ElapsedTime, WorkerStats) ->
  Freq = table:sum_freqs([table:freq_from_stat(W) || W <- WorkerStats]),
  VertsRecvd = lists:sum([worker:verts_recvd_from_stat(W) || W <- WorkerStats]),
  CreditRetd = lists:sum([worker:credit_retd_from_stat(W) || W <- WorkerStats]),
  MinAtomicCredit = lists:max([worker:min_atomic_credit_from_stat(W) || W <- WorkerStats]),
  MaxInitIdle = lists:max([worker:init_idle_from_stat(W) || W <- WorkerStats]),
  MaxIdle = lists:max([worker:max_idle_from_stat(W) || W <- WorkerStats]),
  MaxTailIdle = lists:max([worker:tail_idle_from_stat(W) || W <- WorkerStats]),
  [{wall_time, ElapsedTime},
   {vertices_recvd, VertsRecvd},
   {credit_retd, CreditRetd},
   {min_atomic_credit, MinAtomicCredit},
   {max_init_idle_time, MaxInitIdle},
   {max_idle_time, MaxIdle},
   {max_tail_idle_time, MaxTailIdle} | table:freq_to_stat(Freq)].


%% current wall clock time (in milliseconds since start of RTS)
now() -> element(1, statistics(wall_clock)).
```

Code C.2: Master Module (master.erl)

```
%% orbit-int worker (computing vertices and holding part of hash table)
%%
%% Author: Patrick Maier <P.Maier@hw.ac.uk>
%%

-module(worker).

-export([init/3,
         distribute_vertices/3,
         send_image/4,
         verts_recvd_from_stat/1,
         credit_retd_from_stat/1,
         min_atomic_credit_from_stat/1,
         init_idle_from_stat/1,
         tail_idle_from_stat/1,
         max_idle_from_stat/1]).

%% DATA (see module master)


%% MESSAGES (see module master)


%% counters/timers record
-record(ct,
  {verts_recvd = 0,       %% #vertices received by this server so far
   credit_retd = 0,       %% #times server has returned credit to master
   min_atomic_credit = 0, %% minimal atomic credit received so far
   last_event = master:now(), %% time stamp [ms] of most recent event
   init_idle = -1,        %% idle time [ms] between init recv first vertex
   tail_idle = -1,        %% idle time [ms] between send last vertex and dump
   max_idle = -1}).       %% max idle [ms] time between vertices


%% initialise worker
init(LocalTableSize, IdleTimeout, SpawnImgComp) ->
  Table = table:new(LocalTableSize),
  receive
    {init, StaticMachConf0} ->
      StatData = #ct{},
      StaticMachConf1 = master:set_idle_timeout(StaticMachConf0, IdleTimeout),
      StaticMachConf = case SpawnImgComp of
               true  -> StaticMachConf1;
               _Else -> master:clear_spawn_img_comp(StaticMachConf1)
               end,
      Credit = credit:zero(),
      vertex_server(StaticMachConf, Credit, Table, StatData)
  end.


%% main worker loop: server handling vertex messages;
%% StaticMachConf -- info about machine configuration
%% Credit         -- credit currently held by the server,
%% Table          -- hash table holding vertices
%% StatData       -- various counters and timers for gathering statistics
vertex_server(StaticMachConf, Credit, Table, StatData) ->
  IdleTimeout = master:get_idle_timeout(StaticMachConf),
  receive
    {vertex, X, Slot, K} ->
      Credit_plus_K = credit:credit_atomic(K, Credit),

      Now = master:now(),
```

```
      {NewCredit, NewTable} = handle_vertex(StaticMachConf, X, Slot, Credit_plus_K, Table),
      VertsRecvd = StatData#ct.verts_recvd,
      MinAtomicCredit = StatData#ct.min_atomic_credit,
      LastEvent = StatData#ct.last_event,
      InitIdle = StatData#ct.init_idle,
      MaxIdle = StatData#ct.max_idle,
      NewStatData0 = StatData#ct{verts_recvd = VertsRecvd + 1,
                                 min_atomic_credit = max(MinAtomicCredit, K)},
      NewStatData1 = if
                       InitIdle < 0 -> NewStatData0#ct{init_idle = Now - LastEvent};
                       true         -> NewStatData0#ct{max_idle = max(MaxIdle, Now -
                       LastEvent)}
                     end,
      NewStatData = NewStatData1#ct{last_event = master:now()},
      vertex_server(StaticMachConf, NewCredit, NewTable, NewStatData);

    {dump} ->
      Now = master:now(),
      LastEvent = StatData#ct.last_event,
      NewStatData = StatData#ct{tail_idle = Now - LastEvent,
                                last_event = master:now()},
      dump_table(StaticMachConf, Table, NewStatData)

  after
    IdleTimeout ->
      CreditRetd = StatData#ct.credit_retd,
      NewCreditRetd = return_credit(StaticMachConf, Credit, CreditRetd),
      NewStatData = StatData#ct{credit_retd = NewCreditRetd},
      vertex_server(StaticMachConf, credit:zero(), Table, NewStatData)
  end.


%% handle_vertex checks whether vertex X is stored in Slot of Table;
%% if not, it is in inserted there and the images of the generators
%% are distributed among the workers.
%% Precondition: Credit is non-zero.
handle_vertex(StaticMachConf, X, Slot, Credit, Table) ->
  % check whether X is already in Table
  case table:is_member(X, Slot, Table) of
    true  -> {Credit, Table};  % X already in table; do nothing

    _Else ->                   % X not in table
      % insert X at Slot
      NewTable = table:insert(X, Slot, Table),

      % distribute images of X under generators to their respective workers
      NewCredit = distribute_images(StaticMachConf, X, Credit),
      % return remaining credit and updated table
      {NewCredit, NewTable}
  end.


%% return_credit sends non-zero Credit back to the master;
%% returns number of times credit has been returned so far
return_credit(StaticMachConf, Credit, CreditRetd) ->
  case credit:is_zero(Credit) of
    true  -> CreditRetd;
    false -> MasterPid = master:get_master(StaticMachConf),
             MasterPid ! {done, Credit},
             CreditRetd + 1
  end.
```

```
%% dump_table sends a list containing the local partial orbit to the master,
%% together with some statistics on the distribution of vertices in the table.
dump_table(StaticMachConf, Table, StatData) ->
  MasterPid = master:get_master(StaticMachConf),
  Stat = worker_stats(node(self()), table:get_freq(Table), StatData),
  PartialOrbit = table:to_list(Table),
  MasterPid ! {result, PartialOrbit, Stat}.


%% distribute_images distributes the images of vertex X under the generators
%% to the workers determined by the hash; some ore all of of the Credit is
%% used to send the messages, the remaining credit is returned;
%% computation and sending of vertices is actually done asynchronously.
%% Precondition: Credit is non-zero.
distribute_images(StaticMachConf, X, Credit) ->
  Gs = master:get_gens(StaticMachConf),
  do_distribute_images(StaticMachConf, X, Credit, Gs).

do_distribute_images(_StaticMachConf, _X, Credit, []) ->
  Credit;
do_distribute_images(StaticMachConf, X, Credit, [G]) ->
  {K, RemainingCredit} = credit:debit_atomic(Credit),
  case master:get_spawn_img_comp(StaticMachConf) of
    true  -> spawn(worker, send_image, [StaticMachConf, X, G, K]), ok;
    _Else -> send_image(StaticMachConf, X, G, K)
  end,
  RemainingCredit;
do_distribute_images(StaticMachConf, X, Credit, [G|Gs]) ->
  {K, NonZeroRemainingCredit} = credit:debit_atomic_nz(Credit),
  case master:get_spawn_img_comp(StaticMachConf) of
    true  -> spawn(worker, send_image, [StaticMachConf, X, G, K]), ok;
    _Else -> send_image(StaticMachConf, X, G, K)
  end,
  do_distribute_images(StaticMachConf, X, NonZeroRemainingCredit, Gs).


%% distribute_vertices distributes the list of vertices Xs to the workers
%% determined by the hash; some ore all of of the Credit is used to send
%% the messages, the remaining credit is returned.
%% Precondition: If Xs is non-empty then Credit must be non-zero.
distribute_vertices(_StaticMachConf, Credit, []) ->
  Credit;
distribute_vertices(StaticMachConf, Credit, [X]) ->
  {K, RemainingCredit} = credit:debit_atomic(Credit),
  send_vertex(StaticMachConf, X, K),
  RemainingCredit;
distribute_vertices(StaticMachConf, Credit, [X|Xs]) ->
  {K, NonZeroRemainingCredit} = credit:debit_atomic_nz(Credit),
  send_vertex(StaticMachConf, X, K),
  distribute_vertices(StaticMachConf, NonZeroRemainingCredit, Xs).


%% send_image sends image of X under G to the worker determined by
%% the hash of G(X); the message is tagged with atomic credit K.
send_image(StaticMachConf, X, G, K) ->
  Y = G(X),
  send_vertex(StaticMachConf, Y, K).


%% send_vertex hashes vertex X and sends it to the worker determined by
%% the hash; the message is tagged with atomic credit K.
```

```
send_vertex(StaticMachConf, X, K) ->
  {Pid, Slot} = hash_vertex(StaticMachConf, X),
  Pid ! {vertex, X, Slot, K},
  ok.


%% hash_vertex computes the two-dimensional hash table slot of vertex X where
%% the first dim is a worker pid and the second a slot in that worker's table.
hash_vertex(StaticMachConf, X) ->
  % get static info
  GlobalTableSize = master:get_global_table_size(StaticMachConf),
  Workers = master:get_workers(StaticMachConf),
  % compute raw hash and slot in global table
  Hash = erlang:phash2(X),
  GlobalSlot = Hash rem GlobalTableSize,

  % translate global slot into worker pid and local slot
  global_to_local_slot(Workers, GlobalSlot).


%% global_to_local_slot traverses the list Workers sequentially to translate
%% slot GlobSlot in the global hash table into a two-dimensional local slot
%% {pid, slot}, where 'pid' is the PID of a worker and 'slot' a the slot
%% in that worker's local hash table.
%% Precondition: GlobSlot < sum of TableSize in Workers.
%% Note: This procedure is horribly inefficient (linear in size of Workers);
%%       it should be log (size of Workers) at most.
global_to_local_slot([{Pid, _, TabSize} | Workers], GlobSlot) ->
  if
    GlobSlot < TabSize  -> {Pid, GlobSlot};
    true                -> global_to_local_slot(Workers, GlobSlot - TabSize)
  end.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% auxiliary functions

%% produce readable statistics
worker_stats(Node, Frequency, StatData) ->
  [{node, Node},
   {vertices_recvd, StatData#ct.verts_recvd},
   {credit_retd, StatData#ct.credit_retd},
   {min_atomic_credit, StatData#ct.min_atomic_credit},
   {init_idle_time, StatData#ct.init_idle},
   {max_idle_time, StatData#ct.max_idle},
   {tail_idle_time, StatData#ct.tail_idle} | table:freq_to_stat(Frequency)].

verts_recvd_from_stat(Stat) ->
  case lists:keyfind(vertices_recvd, 1, Stat) of
    {_, VertsRecvd} -> VertsRecvd;
    _Else           -> false
  end.

credit_retd_from_stat(Stat) ->
  case lists:keyfind(credit_retd, 1, Stat) of
    {_, CreditRetd} -> CreditRetd;
    _Else           -> false
  end.

min_atomic_credit_from_stat(Stat) ->
  case lists:keyfind(min_atomic_credit, 1, Stat) of
    {_, MinAtomicCredit} -> MinAtomicCredit;
```

```
    _Else               -> false
  end.

init_idle_from_stat(Stat) ->
  case lists:keyfind(init_idle_time, 1, Stat) of
    {_, InitIdle} -> InitIdle;
    _Else         -> false
  end.

tail_idle_from_stat(Stat) ->
  case lists:keyfind(tail_idle_time, 1, Stat) of
    {_, TailIdle} -> TailIdle;
    _Else         -> false
  end.

max_idle_from_stat(Stat) ->
  case lists:keyfind(max_idle_time, 1, Stat) of
    {_, MaxIdle} -> MaxIdle;
    _Else         -> false
  end.
```

Code C.3: Worker Module (worker.erl)

```
%% orbit-int hash table (storing vertices on a worker)
%%
%% Author: Patrick Maier <P.Maier@hw.ac.uk>
%%


%% Note: Hash tables have a fixed number of slots but each slot can store
%%       a list of vertices. The functions is_member/3 and insert/3
%%       expect its slot argument to be in range.


-module(table).

-export([new/1, to_list/1, is_member/3, insert/3, get_freq/1,
         sum_freqs/2, sum_freqs/1,
         freq_to_slots/1, freq_to_nonempty_slots/1, freq_to_vertices/1,
         max_freq/1, avg_freq/1, avg_nonempty_freq/1, fill_deg/1,
         freq_to_stat/1, freq_from_stat/1]).

%% new(Size) creates a table with Size slots, each containing an empty list.
new(Size) ->
  array:new([{size,Size}, {default,[]}, {fixed,true}]).

%% to_list(T) converts a table T into a list of its entries.
to_list(T) ->
  lists:flatten(array:to_list(T)).

%% is_member(X, I, T) is true iff X is stored in table T at slot I.
is_member(X, I, T) ->
  L = array:get(I, T), lists:member(X, L).

%% insert(X, I, T) inserts X into table T at slot I.
insert(X, I, T) ->
  L = array:get(I, T), array:set(I, [X|L], T).


%% get_freq computes the fill frequency of table T;
%% the output is a list of integers where the number at position I
%% indicates how many slots of T are filled with I entries;
%% the sum of the output lists equals the number of slots of T.
get_freq(T) ->
  F0 = array:new([{default,0}, {fixed,false}]),
  F = array:foldl(fun(_, L, F) -> inc(length(L), F) end, F0, T),
  array:to_list(F).

%% freq_to_slots computes the number of slots from a table fill frequency.
freq_to_slots(F) -> lists:sum(F).

%% freq_to_nonempty_slots computes the number of non empty slots
%% from a table fill frequency.
freq_to_nonempty_slots(F) -> lists:sum(tl(F)).

%% freq_to_vertices computes the number of vertices
%% from a table fill frequency.
freq_to_vertices(F) ->
  {_, V} = lists:foldl(fun(N, {I,X}) -> {I + 1, (I * N) + X} end, {0,0}, F),
  V.

%% max_freq returns the maximum fill frequency.
max_freq(F) -> length(F) - 1.

%% avg_freq returns the average fill frequency
```

```
avg_freq(F) -> freq_to_vertices(F) / freq_to_slots(F).

%% avg_nonempty_freq returns the average fill frequency of non empty slots.
avg_nonempty_freq(F) ->
  case freq_to_vertices(F) of
    Verts when Verts > 0 -> Verts / freq_to_nonempty_slots(F);
    _Verts               -> 0.0  %% Verts = 0 <=> freq_to_nonempty_slots(F) = 0
  end.

%% fill_deg determines the filling degree of the table.
fill_deg(F) -> freq_to_nonempty_slots(F) / freq_to_slots(F).

%% sum_freqs/2 sums two fill frequencies.
sum_freqs([],    SumF)     -> SumF;
sum_freqs(F,     [])       -> F;
sum_freqs([N|F], [M|SumF]) -> [N + M | sum_freqs(F, SumF)].

%% sum_freqs/1 sums a list of fill frequencies.
sum_freqs(Fs) -> lists:foldl(fun(F, SumF) -> sum_freqs(F, SumF) end, [], Fs).


%% freq_to_stat produces a readable statistics from a table fill frequency;
%% the input frequency F is itself part of the statistics
freq_to_stat(Frequency) ->
  [{freq, Frequency},
   {size, table:freq_to_vertices(Frequency)},
   {slots, table:freq_to_slots(Frequency)},
   {nonempty_slots, table:freq_to_nonempty_slots(Frequency)},
   {fill_deg, table:fill_deg(Frequency)},
   {max_freq, table:max_freq(Frequency)},
   {avg_freq, table:avg_freq(Frequency)},
   {nonempty_avg_freq, table:avg_nonempty_freq(Frequency)}].

%% freq_from_stat extracts a table fill frequency from a statistics Stat
%% (assuming Stat was produced by freq_to_stat/1, otherwise returns []);
freq_from_stat(Stat) ->
  case lists:keyfind(freq, 1, Stat) of
    {_, F} -> F;
    _Else  -> []
  end.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% auxiliary functions

inc(I, F) ->
  N = array:get(I, F),
  array:set(I, N + 1, F).
```

Code C.4: Table Module (table.erl)

C.2 SD Erlang Orbit Source Code

This section contains the source code of SD Erlang Orbit. The four main modules, i.e. *grouping*, *master*, *sub-master*, and *worker* are listed as Codes C.5, C.6, C.7, and C.8 respectively. All modules and deployment scripts are publicly available at [117].

```
%% creates dynamically s_gropus
%%
%% Author: Amir Ghaffari <Amir.Ghaffari@glasgow.ac.uk>
%%

-module(grouping).

-export([initiate/2,create_group/3, create_group_list/3, get_submaster_nodes/2]).


%% makes s_groups by dividing all the nodes into a number of s_groups
initiate(Nodes, NumGroup) when NumGroup>0, is_list(Nodes) ->
        L= length(Nodes),
        Size=L div NumGroup,
        if
                Size==0 ->
                  group_size_is_zero;
                true->
                        do_grouping(Nodes, Size, NumGroup,[])
        end.

do_grouping(Nodes, _Size, 1, Acc) ->
        {ok, [make_group(Nodes, length(Acc)+1)|Acc]};

do_grouping(Nodes, Size, NumGroup, Acc) ->
        Group=lists:sublist(Nodes, Size),
        Remain=lists:subtract(Nodes, Group),
        NewGroup = make_group(Group, length(Acc)+1),
        do_grouping(Remain, Size, NumGroup-1, [NewGroup|Acc]).

%% creates a s_group on Submaster and includes all Workers in it
make_group([Submaster|Workers], Counter) ->
        spawn(Submaster, grouping, create_group, [self(), [Submaster|Workers], Counter]),
        receive GroupName ->
                {Submaster, GroupName}
        end.

%% create a s_group on the current node and return the name of s_group back to the master
create_group(Master, Nodes, Counter) ->
        FixName=group, %% prefix for creating s_group name
        GroupName=list_to_atom(atom_to_list(FixName) ++ integer_to_list(Counter)),
        try
                {ok, GroupName, _Nodes}=s_group:new_s_group(GroupName,Nodes), %% creates
                group and add all nodes to it
                Master! GroupName %% Sends the group name to the submaster node
        catch
                 _:_ -> io:format("exception: SD Erlang is not installed at: ~p \n",[code:
                 root_dir()]),
                        sderlang_is_not_installed
        end.

%% divids the Orbit space among submasters
create_group_list(Sub_masters, N, NumberOfGroups) ->
        %Submaster_nodes=[node()]++[HostName || {HostName, _Group} <- Sub_masters],
        %{ok,master_group,_Submaster_nodes}=s_group:new_s_group(master_group,
        Submaster_nodes), %% creates a group and add master and all submasters to it
        List_of_groups=[{Host, GroupName, N div NumberOfGroups} || {Host, GroupName} <-
        Sub_masters], %% calculate the size that each group is responsible for
        R=N-((N div NumberOfGroups)*NumberOfGroups), %% when N is not divisible by
        NumberOfGroups -> R>0
        NewList_of_groups=sub_master:change_list(List_of_groups,length(List_of_groups),
        length(List_of_groups),3,(N div NumberOfGroups)+R),
```

```
        io:format("Number of groups: ~p\n", [length(NewList_of_groups)]),
        io:format("[{Host, GroupName, TableSize}] ~p\n", [NewList_of_groups]),
        NewList_of_groups.

%% returns a list of submaster nodes by dividing all the nodes into a number of s_groups
and returns the head of each group as submaster node
get_submaster_nodes(Nodes, NumGroup) when NumGroup>0, is_list(Nodes) ->
        L= length(Nodes),
        Size=L div NumGroup,
        if
                Size==0 ->
                    group_size_is_zero;
                true->
                    create_submaster_nodes(Nodes, Size, NumGroup,[])
        end.

create_submaster_nodes([Submaster|_Workers], _Size, 1, Acc) ->
    case lists:member(node(), [Submaster|Acc]) of
        true ->
            {ok, [Submaster|Acc]};
        _->
            {ok, [node()]++[Submaster|Acc]}
    end;

create_submaster_nodes(Nodes, Size, NumGroup, Acc) ->
        Group=lists:sublist(Nodes, Size),
        Remain=lists:subtract(Nodes, Group),
        [Submaster|_Workers]=Group,
        create_submaster_nodes(Remain, Size, NumGroup-1, [Submaster|Acc]).
```

Code C.5: Grouping Module (grouping.erl)

```
%% orbit-int master (controlling orbit computation)
%%
%% Author: Patrick Maier <P.Maier@hw.ac.uk>
%%
%% Modified by Amir Ghaffari <Amir.Ghaffari@glasgow.ac.uk> to create SD Erlang version of
Orbit
%%
%% RELEASE project (http://www.release-project.eu/)
%%

-module(master).

-export([orbit/3,
         orbit/7,
         get_gens/1, get_master/1, get_gateway/1, get_workers/1, get_spawn_img_comp/1,
         get_global_table_size/1, get_idle_timeout/1,
         set_idle_timeout/2, clear_spawn_img_comp/1,
         now/0]).

-compile({no_auto_import, [now/0]}).

%% DATA
%%   Static Machine Configuration:
%%     {Gs,               %list of generators
%%      Master,           %pid of master process
%%      Workers,          %list of Worker
%%      GlobalTableSize,  %size of global hash table
%%      IdleTimeout,      %milliseconds this worker idles before sending 'done'
%%      SpawnImgComp}     %true iff this worker spawns image computations
%%
%%   Worker:
%%     {Pid,              %pid of worker process
%%      TableOffset,      %offset (= index 0) of local table into global table
%%      TableSize}        %size of local hash table
%%
%%   Host:
%%     {Node,             %atom naming Erlang node
%%      Procs,            %number of processors
%%      TableSize,        %size of hash table per processor
%%      IdleTimeout}      %milliseconds a processor idles before sending 'done'
%%
%%   Statistics:
%%     List of pairs where the first component is an atom, the second
%%     some data. Part of the data is the fill frequency of the table
%%     (a list whose ith element indicates frequency of filling degree i).


%% MESSAGES
%%   Master -> Worker:        {init, StaticMachConf}
%%
%%   Master/Worker -> Worker: {vertex, X, Slot, K}
%%                               %X is vertex
%%                               %Slot is slot of X on target worker
%%                               %K is atomic credit shipped with vertex
%%
%%   Worker -> Master:        {done, Cs}
%%                               %Cs is non-zero credit (rep as list of ints)
%%
%%   Master -> Worker:        {dump}
%%
%%   Worker -> Master:        {result, Xs, Stats}
%%                               %Xs is list of found orbit vertices
```

```
%%                              %Stats is statistics about worker's table


%% compute orbit of elements in list Xs under list of generators Gs;
%% the argument Hosts is either an integer N, a triple {P, N, T}, or
%% a non-empty list [{H, P, N, T} | ...] of quadruples:
%% * N:                         run the sequential algorithm with table size N
%% * {P, N, T, S}:              run the parallel algorithm on P processors
%%                              each with table size N, idle timeout T and
%%                              spawn image computation flag S;
%% * [{H, P, N, T, S} | ...]: run the distributed algorithm on the list of
%%                              hosts, where each quintuple {H, P, N, T, S}
%%                              specifies
%%                              * host name H (ie. name of Erlang node),
%%                              * number of processors P on H,
%%                              * table size N (per processor),
%%                              * idle timeout T, and
%%                              * spawn image computation flag S.
%% The function returns a pair consisting of the computed orbit and
%% a list of statistics, the first element of which reports overall statistics,
%% and all remaining elements report statistics of some worker.

orbit(Gs, Xs, NumGateways, P, TimeOut, SpawnImgComp, Sub_masters) ->
    Credit=credit:one(),
    StartTime = now(), % start wall clock timer
    Group_Hash_Table= start_submasters(Sub_masters, {[], 0}, {Gs, Xs, NumGateways, P,
    TimeOut, SpawnImgComp,Credit}), %% start submaster nodes and returns Group Hash Table
    lists:foreach(fun({Pid,_,_,_}) -> Pid ! {start, Group_Hash_Table} end, Group_Hash_Table
    ), %% send Group_Hash_Table to all sub-master processes
    lists:foreach(fun({_,Pids,_,_}) -> send_list(Pids,{start, Group_Hash_Table}) end,
    Group_Hash_Table), %% send Group_Hash_Table to all gateway processes
    %% collect credit handed back by idle workers
    collect_credit([]), %% credit:zero()=[]
    ElapsedTime = now() - StartTime, % measure elapsed time (in milliseconds)
    lists:foreach(fun({Pid,_, _, _}) -> Pid ! {dump} end, Group_Hash_Table), %% after
    collecting all the credits, terminates all sub-master processes
    lists:foreach(fun({_,Pids, _, _}) -> send_list(Pids,{dump}) end, Group_Hash_Table), %%
    after collecting all the credits, terminates all gateway processes
    collect_orbit(ElapsedTime, length(Group_Hash_Table)). %% collects the Orbit
    information from all sub-master nodes

send_list([], _Data) ->
  ok;
send_list([Pid|Tail], Data) ->
  Pid ! Data,
  send_list(Tail, Data).

%% creates groups hash table
%% groups hash table is a list of tuples, i.e. {Pid of submaster process, Pid of gateway
process, Start, TableSize}
start_submasters([], { Group_Hash_Table, _GlobalSize}, {_Gs, _Xs, _NumGateways, _P,
_Timeout, _Spawn, _Credit}) ->
  Group_Hash_Table;

start_submasters([{Node, GroupName, TabSize}], { Group_Hash_Table, GlobalSize}, {Gs, Xs,
NumGateways, P, Timeout, Spawn, Credit}) ->
  Pid = spawn_link(Node, sub_master, init,[Gs, Xs, P, Timeout, Spawn, Credit, self(),
  GroupName]),
  Gateways=spawn_gateways(Node, NumGateways,[]), %% spawns a number of gateway processes
  [{Pid, Gateways,GlobalSize, TabSize+1} | Group_Hash_Table];

start_submasters([{Node, GroupName, TabSize} | Sub_masters], { Group_Hash_Table, GlobalSize
```

```
}, {Gs, Xs, NumGateways, P, Timeout, Spawn, Credit}) ->
  Pid = spawn_link(Node, sub_master, init,[Gs, Xs, P, Timeout, Spawn, Credit, self(),
  GroupName]),
  Gateways=spawn_gateways(Node, NumGateways,[]), %% spawns a number of gateway processes
  New_Group_Hash_Table = [{Pid, Gateways,GlobalSize, TabSize} | Group_Hash_Table],
  New_GlobalSize= GlobalSize+TabSize,
  New_Xs=[], %% just the first sub-master node kicks off the computation
  New_Credit=credit:zero(), %% other submasters get no credit at first
  start_submasters(Sub_masters, {New_Group_Hash_Table,New_GlobalSize}, {Gs, New_Xs,
  NumGateways, P, Timeout, Spawn, New_Credit}).

spawn_gateways(_Node, 0, Gateways)->
  Gateways;

spawn_gateways(Node, NumGateways, Gateways) when NumGateways>0 ->
  Pid = spawn_link(Node, sub_master, gateway,[]),
  spawn_gateways(Node, NumGateways-1, [Pid|Gateways]).

orbit(Gs, Xs, Hosts) ->
  if
    is_integer(Hosts) ->
      TableSize = Hosts,
      sequential:orbit(Gs, Xs, TableSize);
    true ->
      par_orbit(Gs, Xs, Hosts)
  end.

par_orbit(Gs, Xs, Hosts) ->
  % spawn workers on Hosts
  {Workers, GlobTabSize} = start_workers(Hosts),
  % assemble StaticMachConf and distribute to Workers
  StaticMachConf = mk_static_mach_conf(Gs, self(), Workers, GlobTabSize),
  lists:foreach(fun({Pid, _, _}) -> Pid ! {init, StaticMachConf} end, Workers),

  % start wall clock timer
  StartTime = now(),

  % distribute initial vertices to workers
  Credit = worker:distribute_vertices(StaticMachConf, credit:one(), Xs),
  % collect credit handed back by idle workers
  collect_credit(Credit),

  % measure elapsed time (in milliseconds)
  ElapsedTime = now() - StartTime,

  % tell all Workers to dump their tables
  lists:foreach(fun({Pid, _, _}) -> Pid ! {dump} end, Workers),

  % collect results from all workers and return them
  collect_orbit(ElapsedTime, length(Workers)).


%% start_workers starts worker processes depending on the input Hosts:
%% * if Hosts is a quadruple {P, _, _, _} then P processes are forked on the
%%   executing Erlang node;
%% * if Hosts is a non-empty list {H1, P1, _, _, _}, {H2, P2, _, _, _}, ...
%%   then P1 processes are forked on Erlang node H1, P2 processes on node H2,
%%   and so on.
%% The function returns a pair {Workers, GlobalTableSize}, where
%% * GlobalTableSize is the total number of slots of the global hash table, and
%% * Workers is a list of Worker, sorted wrt. TableOffset in ascending order.
start_workers({Procs, TabSize, TmOut, SpawnImgComp}) ->
```

```
    {Workers, GlobalTableSize} = do_start_shm({Procs, TabSize, TmOut, SpawnImgComp}, {[], 0}),
    {lists:reverse(Workers), GlobalTableSize};
  start_workers([Host | Hosts]) ->
    {Workers, GlobalTableSize} = do_start_dist([Host | Hosts], {[], 0}),
    {lists:reverse(Workers), GlobalTableSize}.

  do_start_shm({0, _, _, _}, Acc) ->
    Acc;
  do_start_shm({M, TabSize, TmOut, SpawnImgComp}, {Workers, GTabSize}) ->
    Pid = spawn_link(worker, init, [TabSize, TmOut, SpawnImgComp]),
    NewWorkers = [{Pid, GTabSize, TabSize} | Workers],
    NewGTabSize = GTabSize + TabSize,
    Acc = {NewWorkers, NewGTabSize},
    do_start_shm({M - 1, TabSize, TmOut, SpawnImgComp}, Acc).

  do_start_dist([], Acc) ->
    Acc;
  do_start_dist([{_, 0, _, _, _} | Hosts], Acc) ->
    do_start_dist(Hosts, Acc);

  do_start_dist([{Node, M, TabSize, TmOut, SpawnImgComp} | Hosts], {Workers, GTabSize}) ->
    Pid = spawn_link(Node, worker, init, [TabSize, TmOut, SpawnImgComp]),
    NewWorkers = [{Pid, GTabSize, TabSize} | Workers],
    NewGTabSize = GTabSize + TabSize,
    Acc = {NewWorkers, NewGTabSize},
    do_start_dist([{Node, M - 1, TabSize, TmOut, SpawnImgComp} | Hosts], Acc).

  %% collect_credit collects leftover credit from idle workers until
  %% the credit adds up to 1.
  collect_credit(Credit) ->
    case credit:is_one(Credit) of
      true  -> ok;  %% break loop and return dummy atom
      _Else ->
        receive
          {done, WorkersCredit} ->
            CollectedCredit = credit:credit(WorkersCredit, Credit),
            collect_credit(CollectedCredit)
        end
    end.

  %% collect_orbit collects partial orbits and stats from N workers.
  collect_orbit(ElapsedTime, N) ->
    {PartOrbits, WorkerStats} = do_collect_orbit(N, [], []),
    Orbit = lists:flatten(PartOrbits),
    Stats = [master_stats(ElapsedTime, WorkerStats) | WorkerStats],
    {Orbit, Stats}.

  do_collect_orbit(0, PartOrbits, WorkerStats) -> {PartOrbits, WorkerStats};
  do_collect_orbit(N, PartOrbits, WorkerStats) ->
    receive
      {result, PartOrbit, WorkerStat} ->
        do_collect_orbit(N - 1, PartOrbit++PartOrbits, WorkerStat++WorkerStats)
    end.


  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %% auxiliary functions

  %% functions operating on the StaticMachConf
  mk_static_mach_conf(Gs, Master, Workers, GlobalTableSize) ->
     {Gs, Master, Workers, GlobalTableSize, 0, true}.
  get_gens(StaticMachConf)              -> element(1, StaticMachConf).
```

```
get_master(StaticMachConf)            -> element(2, StaticMachConf).
get_gateway(StaticMachConf)           -> element(3, StaticMachConf).
get_workers(StaticMachConf)           -> element(4, StaticMachConf).
get_global_table_size(StaticMachConf) -> element(5, StaticMachConf).
get_idle_timeout(StaticMachConf)      -> element(6, StaticMachConf).
get_spawn_img_comp(StaticMachConf)    -> element(7, StaticMachConf).
set_idle_timeout(StaticMachConf, X)  -> setelement(6, StaticMachConf, X).
clear_spawn_img_comp(StaticMachConf) -> setelement(7, StaticMachConf, false).


%% produce readable statistics
master_stats(ElapsedTime, WorkerStats) ->
  Freq = table:sum_freqs([table:freq_from_stat(W) || W <- WorkerStats]),
  VertsRecvd = lists:sum([worker:verts_recvd_from_stat(W) || W <- WorkerStats]),
  CreditRetd = lists:sum([worker:credit_retd_from_stat(W) || W <- WorkerStats]),
  MinAtomicCredit = lists:max([worker:min_atomic_credit_from_stat(W) || W <- WorkerStats]),
  MaxInitIdle = lists:max([worker:init_idle_from_stat(W) || W <- WorkerStats]),
  MaxIdle = lists:max([worker:max_idle_from_stat(W) || W <- WorkerStats]),
  MaxTailIdle = lists:max([worker:tail_idle_from_stat(W) || W <- WorkerStats]),
  [{wall_time, ElapsedTime},
   {vertices_recvd, VertsRecvd},
   {credit_retd, CreditRetd},
   {min_atomic_credit, MinAtomicCredit},
   {max_init_idle_time, MaxInitIdle},
   {max_idle_time, MaxIdle},
   {max_tail_idle_time, MaxTailIdle} | table:freq_to_stat(Freq)].


%% current wall clock time (in milliseconds since start of RTS)
now() -> element(1, statistics(wall_clock)).
```

Code C.6: Master Module (master.erl)

```
%% sub-master for performing double hashing
%%
%% Author: Amir Ghaffari <Amir.Ghaffari@glasgow.ac.uk>
%%
%% RELEASE project (http://www.release-project.eu/)
%%

-module(sub_master).

-export([init/8, gateway/0,change_list/5]).


%% Initiates sub-master, gateway, and all worker processes inside the s_group
init(Gs, Xs, P, Timeout, Spawn, Credit, MasterID, GroupName)->
  case config:get_key(separate_node_for_submasters) of
        false ->
                process_flag(priority, high), %% set priority of gateway process to high
                Hosts= s_group:own_nodes(GroupName); %% get all worker nodes that this
                submaster node is responsible for
        _ ->
                AllHosts= s_group:own_nodes(GroupName), %% get all worker nodes that this
                submaster node is responsible for
                if
                        length(AllHosts)>1 ->
                                Hosts=lists:delete(node(),AllHosts); %% removes submaster
                                node from the list of worker nodes
                        true ->  %% there is just one node in the group
                                Hosts=AllHosts
                end
  end,
  io:format("Number of worker nodes in group ~p is: ~p \n",[GroupName, length(Hosts)]),
  receive
        {start, Group_Hash_Table} -> %% Receives the group hash table from Master node
                FoundGroup=search_sub_group(self(), Group_Hash_Table),
                case FoundGroup of
                    {_Pid, Gateways, GroupStart, GroupSize} ->
                        ok;
                     _ -> Gateways=0, GroupStart=0, GroupSize=0,
                                throw(group_not_found)
                end,
                ProcessTabSize=GroupSize div (length(Hosts)*P), %% find table size for
                each process
                WorkerTabSize2= GroupSize-(ProcessTabSize*length(Hosts)*P), %% find
                additional table size for the last worker node
                ProcessTabSize2=WorkerTabSize2 div P, %% find additional table size for
                the last node processes
                ProcessTabSize3=GroupSize-(ProcessTabSize*length(Hosts)*P)-(ProcessTabSize2
                *P), %% find additional table size for the last process on the last node
                Temp_Worker_Hash_Table=make_workers_hash_table(Hosts, P, ProcessTabSize),
                Temp_Worker_Hash_Table2=change_list(Temp_Worker_Hash_Table,P*(length(Hosts)
                -1)+1, P*length(Hosts)-1, 2, ProcessTabSize+ProcessTabSize2), %% increase
                the size of table for all processes on the last node
                Worker_Hash_Table=change_list(Temp_Worker_Hash_Table2,P*length(Hosts), P*
                length(Hosts), 2, ProcessTabSize+ProcessTabSize2+ProcessTabSize3), %%
                increase the size of table for the last processes on the last node
                {Workers, _GlobTabSize}=start_workers(Worker_Hash_Table,{[],GroupStart}),

                [{_, _, LastGroupStart, LastGroupSize}|_Tail] = Group_Hash_Table,

                [{_, FirstWorkerStart,_}|_Tail2]=lists:reverse(Workers),
                [{_, LastWorkerStart,LastWorkerSize}|_Tail3]=Workers,
```

```
                case GroupStart of
                     FirstWorkerStart ->
                        ok;
                     _ -> throw(group_start_and_first_process_start_not_equal)
                end,

                EndOfWorkersInGroup=LastWorkerStart+LastWorkerSize,
                case GroupStart+GroupSize of
                     EndOfWorkersInGroup ->
                        ok;
                     _ ->throw(group_is_not_divided_properly)
                end,

                distribute_gateways(Gs,Gateways,Gateways,Workers,Workers,LastGroupStart+
                LastGroupSize,Timeout,Spawn), %% assign a gateway to all worker processes

                StaticMachConf={Gs,self(),Gateways,Workers,LastGroupStart+LastGroupSize,
                Timeout,Spawn}, %% contains the process hash table for this s_group
                lists:foreach(fun(Gateway) -> Gateway! {hash_table, StaticMachConf } end,
                Gateways), %% process hash table is sent to the gateway processes

                case Xs of
                        [] -> ok;
                        _ ->
                                % distribute initial vertices to workers
                                [GatewayHead|_GatewayTail]=Gateways,
                                StaticMachConf2=setelement(3,StaticMachConf,GatewayHead),
                                worker:distribute_vertices(StaticMachConf2, Credit, Xs)
                end
  end,

  collect_credit(MasterID),
  % ask from all Workers to dump their tables
  lists:foreach(fun({Pid, _, _}) -> Pid ! {dump} end, Workers),
  collect_orbit(MasterID,length(Workers)),
  s_group:delete_s_group(GroupName). %% delete s_groups at the end


%% assign a gateway process to all worker processes
distribute_gateways(_Gs,_Gateways,_Gateways2,_Workers,[],_LastGroupMember,_Timeout,_Spawn) ->
    ok;

distribute_gateways(Gs,Gateways,[],Workers, Workers2,LastGroupMember,Timeout,Spawn) ->
    distribute_gateways(Gs,Gateways,Gateways,Workers,Workers2,LastGroupMember,Timeout,Spawn);

distribute_gateways(Gs,Gateways,[Gateway|GatewaysTail],Workers, [Worker|WorkersTail],
LastGroupMember,Timeout,Spawn) ->
    StaticMachConf={Gs,self(),Gateway,Workers,LastGroupMember,Timeout,Spawn}, %% contains
    the process hash table for this s_group
    {Pid,_,_}=Worker,
    Pid ! {init, StaticMachConf }, %% process hash table is sent to all worker processes
    distribute_gateways(Gs,Gateways,GatewaysTail,Workers,WorkersTail,LastGroupMember,
    Timeout,Spawn).

%% update elements of a list from (From) to (To) with value (Value)
change_list(Worker_Hash_Table, From, To, Element, Value) ->
        if
          From>To ->
            Worker_Hash_Table;
          true ->
            Temp=lists:nth(From,Worker_Hash_Table),
            Updated=setelement(Element, Temp, Value),
```

```
            NewWorker_Hash_Table=lists:sublist(Worker_Hash_Table,From-1) ++ [Updated] ++
            lists:nthtail(From,Worker_Hash_Table),
            change_list(NewWorker_Hash_Table, From+1, To, Element, Value)
        end.

%% makes the second level hash table for its own s_group
%% table is a list of tuples {HostName,ProcessTableSize}
make_workers_hash_table(Hosts,P,ProcessTableSize) ->
        Counter=P, %% runs P processes on each node
        do_make_workers_hash_table([],Counter,Hosts,P,ProcessTableSize).

do_make_workers_hash_table(Worker_Hash_Table,_Counter,[],_P,_ProcessTableSize) ->
        Worker_Hash_Table;

do_make_workers_hash_table(Worker_Hash_Table,0,[_Host|Remains],P,ProcessTableSize) ->
        do_make_workers_hash_table(Worker_Hash_Table,P,Remains,P,ProcessTableSize);

do_make_workers_hash_table(Worker_Hash_Table,Counter,[Host|Remains],P,ProcessTableSize) ->
        NewWorker_Hash_Table = [{Host,ProcessTableSize}|Worker_Hash_Table],
        do_make_workers_hash_table(NewWorker_Hash_Table,Counter-1,[Host|Remains],P,
        ProcessTableSize).

%% create a number of worker processes on the nodes inside the s_group and return a table
%% table is a list of tuples {process PID, process_start_from, process table size}
start_workers([], {Workers, GTabSize}) ->
        {Workers, GTabSize};

start_workers([{Node, TabSize} | Hosts], {Workers, GTabSize}) ->
        Pid = spawn_link(Node, worker, init, [TabSize]),
        NewWorkers = [{Pid, GTabSize, TabSize} | Workers],
        NewGTabSize = GTabSize+TabSize,
        start_workers(Hosts, {NewWorkers, NewGTabSize}).

%% when receives a credit return it back to the master node
collect_credit(MasterID) ->
        receive
          {done, Credit} ->
            MasterID! {done, Credit},
            collect_credit(MasterID);
          {dump} ->
            ok
        end.

%% collect_orbit collects partial orbits and stats from N workers.
collect_orbit(MasterID,N) ->
  {PartOrbits, WorkerStats} = do_collect_orbit(N, [], []),
  MasterID! {result, PartOrbits, WorkerStats}.

do_collect_orbit(0, PartOrbits, WorkerStats) -> {PartOrbits, WorkerStats};
do_collect_orbit(N, PartOrbits, WorkerStats) ->
  receive
    {result, PartOrbit, WorkerStat} ->
      do_collect_orbit(N - 1, [PartOrbit|PartOrbits], [WorkerStat|WorkerStats])
  end.

%% this is gateway process which is created on all sub-master nodes
gateway() ->
  process_flag(priority, high), %% set priority of gateway process to high
  receive
        {start, Group_Hash_Table} -> %% Receives the group hash table from Master node
                gateway(Group_Hash_Table)
  end.
```

```
%% gateway process receives the first level hash table in which range is divided among groups
gateway(Group_Hash_Table) ->
  receive
        {hash_table, StaticMachConf} -> %% Hash table for worker processes inside own group
                do_gateway(Group_Hash_Table, StaticMachConf)
  end.

%% a loop that receives a pair of {X,Credit} and finds appropriate group for it
do_gateway(Group_Hash_Table, StaticMachConf) ->
  receive
    {X,K} ->
      GlobalTableSize = master:get_global_table_size(StaticMachConf),
      Hash = erlang:phash2(X), %% erlang:phash2(X) returns hash value for X
      GlobSlot = Hash rem GlobalTableSize,
      Gateways=find_gateway(Group_Hash_Table,GlobSlot),

        if
            Gateways==not_found_appropriate_gateway ->
                io:format("Not found appropriate gateway for ~p \n",[{X,K}]),
                throw("not_found_appropriate_gateway");
            true ->
                case lists:member(self(), Gateways) of
                    true  ->    % X belongs to the current s_group
                                StaticMachConf2=setelement(3,StaticMachConf,self()),
                                worker:send_vertex(StaticMachConf2,X,K);
                    false ->    %% X belongs to another s_group
                                Gateway= lists:nth(getRandomm(length(Gateways)),Gateways),
                                Gateway! {X,K}
                end
        end,

      do_gateway(Group_Hash_Table, StaticMachConf);
    {dump} ->
                ok
  end.

%% A recursive search in group hash table and returns an appropriate gateway process
find_gateway([{_,Gateways,Start,_TabSize} | Group_Hash_Table],GlobSlot)->
  if
    GlobSlot >= Start ->
      Gateways;
    true ->
      find_gateway(Group_Hash_Table,GlobSlot)
  end;

find_gateway([],_GlobSlot)->
  not_found_appropriate_gateway.

%% A recursive search for a specific gateway PID
%% Group_Hash_Table=[{Pid, Gateway,GlobalSize, TabSize} | ...];
search_sub_group(Target_Pid, [Head|Group_Hash_Table]) ->
        {Pid, Gateways, GlobalSize, TabSize}=Head,
        case Pid of
                Target_Pid-> {Pid, Gateways, GlobalSize, TabSize};
                _ -> search_sub_group(Target_Pid, Group_Hash_Table)
        end;

search_sub_group( _Target_Pid, []) ->
        not_found_appropriate_group.

%% generates a random number
```

```
getRandomm(Max) ->
        {A, B, C} = erlang:now(),
        random:seed(A, B, C),
        random:uniform(Max).
```

Code C.7: Sub-master Module (sub_master.erl)

```
%% orbit-int worker (computing vertices and holding part of hash table)
%%
%% Author: Patrick Maier <P.Maier@hw.ac.uk>
%%
%% Modified by Amir Ghaffari <Amir.Ghaffari@glasgow.ac.uk> to create SD Erlang version of Orbit
%%
%% RELEASE project (http://www.release-project.eu/)
%%

-module(worker).

-export([init/1,
         distribute_vertices/3,
         send_vertex/3,
         send_image/4,
         verts_recvd_from_stat/1,
         credit_retd_from_stat/1,
         min_atomic_credit_from_stat/1,
         init_idle_from_stat/1,
         tail_idle_from_stat/1,
         max_idle_from_stat/1]).

%% DATA (see module master)


%% MESSAGES (see module master)


%% counters/timers record
-record(ct,
  {verts_recvd = 0,       %% #vertices received by this server so far
   credit_retd = 0,       %% #times server has returned credit to master
   min_atomic_credit = 0, %% minimal atomic credit received so far
   last_event = master:now(), %% time stamp [ms] of most recent event
   init_idle = -1,        %% idle time [ms] between init recv first vertex
   tail_idle = -1,        %% idle time [ms] between send last vertex and dump
   max_idle = -1}).       %% max idle [ms] time between vertices


%% initialise worker
init(LocalTableSize) ->
  Table = table:new(LocalTableSize),
  receive
    {init, StaticMachConf} ->
      StatData = #ct{},
      Credit = credit:zero(),
      vertex_server(StaticMachConf, Credit, Table, StatData)
  end.

%% main worker loop: server handling vertex messages;
%% StaticMachConf -- info about machine configuration
%% Credit         -- credit currently held by the server,
%% Table          -- hash table holding vertices
%% StatData       -- various counters and timers for gathering statistics
vertex_server(StaticMachConf, Credit, Table, StatData) ->
  IdleTimeout = master:get_idle_timeout(StaticMachConf),
  receive
    {vertex, X, Slot, K} ->
      Credit_plus_K = credit:credit_atomic(K, Credit),
      Now = master:now(),
      {NewCredit, NewTable} = handle_vertex(StaticMachConf, X, Slot, Credit_plus_K, Table),
```

```
      VertsRecvd = StatData#ct.verts_recvd,
      MinAtomicCredit = StatData#ct.min_atomic_credit,
      LastEvent = StatData#ct.last_event,
      InitIdle = StatData#ct.init_idle,
      MaxIdle = StatData#ct.max_idle,
      NewStatData0 = StatData#ct{verts_recvd = VertsRecvd + 1,
                                 min_atomic_credit = max(MinAtomicCredit, K)},
      NewStatData1 = if
                       InitIdle < 0 -> NewStatData0#ct{init_idle = Now - LastEvent};
                       true         -> NewStatData0#ct{max_idle = max(MaxIdle, Now -
                       LastEvent)}
                     end,
      NewStatData = NewStatData1#ct{last_event = master:now()},
      vertex_server(StaticMachConf, NewCredit, NewTable, NewStatData);

    {dump} ->
      Now = master:now(),
      LastEvent = StatData#ct.last_event,
      NewStatData = StatData#ct{tail_idle = Now - LastEvent,
                                last_event = master:now()},
      dump_table(StaticMachConf, Table, NewStatData)

  after
    IdleTimeout ->
      CreditRetd = StatData#ct.credit_retd,
      NewCreditRetd = return_credit(StaticMachConf, Credit, CreditRetd),
      NewStatData = StatData#ct{credit_retd = NewCreditRetd},
      vertex_server(StaticMachConf, credit:zero(), Table, NewStatData)
  end.


%% handle_vertex checks whether vertex X is stored in Slot of Table;
%% if not, it is in inserted there and the images of the generators
%% are distributed among the workers.
%% Precondition: Credit is non-zero.
handle_vertex(StaticMachConf, X, Slot, Credit, Table) ->
  % check whether X is already in Table
  case table:is_member(X, Slot, Table) of
    true  -> {Credit, Table};  % X already in table; do nothing

    _Else ->                   % X not in table
      % insert X at Slot
      NewTable = table:insert(X, Slot, Table),

      % distribute images of X under generators to their respective workers
      NewCredit = distribute_images(StaticMachConf, X, Credit),
      % return remaining credit and updated table
      {NewCredit, NewTable}
  end.


%% return_credit sends non-zero Credit back to the master;
%% returns number of times credit has been returned so far
return_credit(StaticMachConf, Credit, CreditRetd) ->
  case credit:is_zero(Credit) of
    true  -> CreditRetd;
    false -> MasterPid = master:get_master(StaticMachConf),
             MasterPid ! {done, Credit},
             CreditRetd + 1
  end.
```

```
%% dump_table sends a list containing the local partial orbit to the master,
%% together with some statistics on the distribution of vertices in the table.
dump_table(StaticMachConf, Table, StatData) ->
  MasterPid = master:get_master(StaticMachConf),
  Stat = worker_stats(node(self()), table:get_freq(Table), StatData),
  PartialOrbit = table:to_list(Table),
  MasterPid ! {result, PartialOrbit, Stat}.


%% distribute_images distributes the images of vertex X under the generators
%% to the workers determined by the hash; some ore all of of the Credit is
%% used to send the messages, the remaining credit is returned;
%% computation and sending of vertices is actually done asynchronously.
%% Precondition: Credit is non-zero.
distribute_images(StaticMachConf, X, Credit) ->
  Gs = master:get_gens(StaticMachConf),
  do_distribute_images(StaticMachConf, X, Credit, Gs).

do_distribute_images(_StaticMachConf, _X, Credit, []) ->
  Credit;
do_distribute_images(StaticMachConf, X, Credit, [G]) ->
  {K, RemainingCredit} = credit:debit_atomic(Credit),
  case master:get_spawn_img_comp(StaticMachConf) of
    true  -> spawn(worker, send_image, [StaticMachConf, X, G, K]), ok;
    _Else -> send_image(StaticMachConf, X, G, K)
  end,
  RemainingCredit;
do_distribute_images(StaticMachConf, X, Credit, [G|Gs]) ->
  {K, NonZeroRemainingCredit} = credit:debit_atomic_nz(Credit),
  case master:get_spawn_img_comp(StaticMachConf) of
    true  -> spawn(worker, send_image, [StaticMachConf, X, G, K]), ok;
    _Else -> send_image(StaticMachConf, X, G, K)
  end,
  do_distribute_images(StaticMachConf, X, NonZeroRemainingCredit, Gs).


%% distribute_vertices distributes the list of vertices Xs to the workers
%% determined by the hash; some ore all of of the Credit is used to send
%% the messages, the remaining credit is returned.
%% Precondition: If Xs is non-empty then Credit must be non-zero.
distribute_vertices(_StaticMachConf, Credit, []) ->
  Credit;
distribute_vertices(StaticMachConf, Credit, [X]) ->
  {K, RemainingCredit} = credit:debit_atomic(Credit),
  send_vertex(StaticMachConf, X, K),
  RemainingCredit;
distribute_vertices(StaticMachConf, Credit, [X|Xs]) ->
  {K, NonZeroRemainingCredit} = credit:debit_atomic_nz(Credit),
  send_vertex(StaticMachConf, X, K),
  distribute_vertices(StaticMachConf, NonZeroRemainingCredit, Xs).


%% send_image sends image of X under G to the worker determined by
%% the hash of G(X); the message is tagged with atomic credit K.
send_image(StaticMachConf, X, G, K) ->
  Y = G(X),
  send_vertex(StaticMachConf, Y, K).


%% send_vertex hashes vertex X and sends it to the worker determined by
%% the hash; the message is tagged with atomic credit K.
send_vertex(StaticMachConf, X, K) ->
```

```
  case hash_vertex(StaticMachConf, X, K) of
  {Pid, Slot} ->
    Pid ! {vertex, X, Slot, K};
  ok ->
    ok;
  Exception ->
    MasteID = master:get_master(StaticMachConf),
    MasteID! {print, {exception, Exception}}
  end.

%% hash_vertex computes the two-dimensional hash table slot of vertex X where
%% the first dim is a worker pid and the second a slot in that worker's table.
hash_vertex(StaticMachConf, X, K) ->
  % get static info
  GlobalTableSize = master:get_global_table_size(StaticMachConf),
  Workers = master:get_workers(StaticMachConf),
  %compute raw hash and slot in global table
  Hash = erlang:phash2(X),
  GlobalSlot = Hash rem GlobalTableSize,
  [{_, Start, _}|_Tail]= lists:reverse(Workers),
  [{_, End, TabSize}|_Tail2]= Workers,

  if
    GlobalSlot<Start ->   %% if X belongs to a process on another group
        Gateway=master:get_gateway(StaticMachConf),
        Gateway! {X,K},
        ok;
    GlobalSlot>= End+TabSize ->   %% if X belongs to a process on another group
        Gateway =master:get_gateway(StaticMachConf),
        Gateway! {X,K},
        ok;
    true ->   % translate global slot into worker pid and local slot
        global_to_local_slot(Workers, GlobalSlot)
  end.


%% global_to_local_slot traverses the list Workers sequentially to translate
%% slot GlobSlot in the global hash table into a two-dimensional local slot
%% {pid, slot}, where 'pid' is the PID of a worker and 'slot' a the slot
%% in that worker's local hash table.
%% Precondition: GlobSlot < sum of TableSize in Workers.
%% Note: This procedure is horribly inefficient (linear in size of Workers);
%%       it should be log (size of Workers) at most.
global_to_local_slot([{Pid, GTabSize, _TabSize} | Workers], GlobSlot) ->
  if
    GlobSlot >= GTabSize -> {Pid, GlobSlot-GTabSize};
    true                 -> global_to_local_slot(Workers, GlobSlot)
  end;

global_to_local_slot([], _GlobSlot) ->
  not_found_appropriate_slot.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% auxiliary functions

%% produce readable statistics
worker_stats(Node, Frequency, StatData) ->
  [{node, Node},
   {vertices_recvd, StatData#ct.verts_recvd},
   {credit_retd, StatData#ct.credit_retd},
   {min_atomic_credit, StatData#ct.min_atomic_credit},
   {init_idle_time, StatData#ct.init_idle},
```

```
    {max_idle_time, StatData#ct.max_idle},
    {tail_idle_time, StatData#ct.tail_idle} | table:freq_to_stat(Frequency)].

verts_recvd_from_stat(Stat) ->
  case lists:keyfind(vertices_recvd, 1, Stat) of
    {_, VertsRecvd} -> VertsRecvd;
    _Else           -> false
  end.

credit_retd_from_stat(Stat) ->
  case lists:keyfind(credit_retd, 1, Stat) of
    {_, CreditRetd} -> CreditRetd;
    _Else           -> false
  end.

min_atomic_credit_from_stat(Stat) ->
  case lists:keyfind(min_atomic_credit, 1, Stat) of
    {_, MinAtomicCredit} -> MinAtomicCredit;
    _Else                -> false
  end.

init_idle_from_stat(Stat) ->
  case lists:keyfind(init_idle_time, 1, Stat) of
    {_, InitIdle} -> InitIdle;
    _Else         -> false
  end.

tail_idle_from_stat(Stat) ->
  case lists:keyfind(tail_idle_time, 1, Stat) of
    {_, TailIdle} -> TailIdle;
    _Else         -> false
  end.

max_idle_from_stat(Stat) ->
  case lists:keyfind(max_idle_time, 1, Stat) of
    {_, MaxIdle} -> MaxIdle;
    _Else         -> false
  end.
```

Code C.8: Worker Module (worker.erl)

C.3 Distributed ACO Source Code

This section contains the source code of distributed ACO developed by Dr Kenneth MacKenzie from Glasgow University. The three main modules, i.e. *ant*, *colony*, and *master* are listed as Codes C.9, C.10, and C.11 respectively. All modules and deployment scripts are publicly available at [118].

```
%% Individual ant: constructs new solution based on pheromone matrix.

-module(ant).
-include ("types.hrl").
-export([start/3]).

% M&M -> Merkle & Middendorf: An Ant Algorithm with a new
% Pheromone Evaluation Rule for Total Tardiness Problems


%% ------------------------------ Random numbers -------------------------------- %%

-spec init_rng(params()) -> ok.
init_rng(Params) ->
    #params{rng_type=RNG_type} = Params,
    put (rng_type, RNG_type),
    case RNG_type of
    now ->
        {A, B, C} = now(),
        random:seed(A,B,C);
    crypto ->
        <<A:32,B:32,C:32>> = crypto:rand_bytes(12),
        random:seed(A,B,C);
            % This seems to be safer, but the HW machines don't have the
        % correct C crypto library.
    cyclic -> put (rng_index, 1)
    end.

-spec random () -> float().
random() ->
    case get(rng_type) of
    cyclic ->   % Fake RNG for benchmarking
        Vals = {0.09, 0.19, 0.29, 0.39, 0.49, 0.59, 0.69, 0.79, 0.89, 0.99},
        N = tuple_size(Vals),
        I = get(rng_index),
        I1 = if I <  N -> I+1;
            I >= N -> 1
         end,
        put (rng_index, I1),
        element(I, Vals);
    _ -> random:uniform()
    end.

-spec reset_cyclic_rng () -> ok.
reset_cyclic_rng() ->
    case get(rng_type) of
    cyclic -> put (rng_index, 1);
    _ -> ok
    end.


%% ---------------------------------------------------------------------- %%
% Strictly, P(i,j) is equal to Pij/Sigma; he're we're calculating both as we go along.

construct_pij([], _, _, _, Sigma, P) -> % Sigma is sum of terms over unscheduled jobs: see
M&M
    {lists:reverse(P), Sigma};
construct_pij([{J,Eta_J}|TEta], Alpha, Beta, TauI, Sigma, P) ->
    Pij = math:pow (element(J, TauI), Alpha) * math:pow(Eta_J, Beta),
    construct_pij(TEta, Alpha, Beta, TauI, Sigma+Pij, [{J,Pij}|P]).

% Eta contains info only for unscheduled jobs, so the above function constructs info only
```

```
for those jobs.


%% ---------------------------------------------------------------- %%
% Careful here: elements of Pij can be extremely small (10^-200 towards end).

-spec find_limit_index ([idxf()], index(), float(), float()) -> index().
find_limit_index(P, Idx, Total, Limit) ->
    if Total >= Limit -> Idx;
       true -> case P of
           [] -> error ("Fell off end of P in find_limit_index");
                       % This could happen if Limit is 1 (or very close)
                       % and rounding errors make the sum of pij slightly
                       % less than 1.
           [{K, Kval}|Rest] ->
               find_limit_index (Rest, K, Total+Kval, Limit)
           end
    end.


-spec find_random_job([idxf(),...], float()) -> index().
find_random_job(P_i, SumPij) ->
    case P_i of
    [] -> error ("find_rand_pij: P_i is empty");
    [{J1, J1val}|Rest] ->
        Limit = random() * SumPij,
        find_limit_index(Rest, J1, J1val, Limit)
    end.



%% ---------------------------------------------------------------- %%

% Find index j which maximises P[i,j] (i is fixed).
% Note that we're usually working with a partial schedule, in which case many of the
% elements of Pij are zero.  We need to be sure that at least one element is greater
% than zero when calling this function.


-spec find_index_max_P(index(), float(), [idxf()]) -> index().

find_index_max_P(MaxIndex, _, []) -> MaxIndex;
find_index_max_P(MaxIndex, MaxVal, [{K,Kval}|Rest]) ->
    if Kval =< MaxVal -> find_index_max_P (MaxIndex, MaxVal, Rest);
       Kval >  MaxVal -> find_index_max_P (K,        Kval,   Rest)
    end.


-spec find_index_max_P([idxf(),...]) -> index().

find_index_max_P ([]) -> error ("find_index_max_P: P_i is empty");
find_index_max_P ([{J1,J1val}|Rest]) -> find_index_max_P (J1, J1val, Rest).


%% ---------------------------------------------------------------- %%

-spec find_solution (I::integer(),
                     % We're trying to schedule a job at position I. We need I to index
                     tau here.
             Unscheduled_Jobs :: natlist(),
```

```
            Inputs::inputs(),
            Alpha::number(), Beta::number(), Q0::number(),
            Heuristic :: heuristic(),
            Partial_Schedule::natlist(),
            Scheduled_Time::integer()
           ) -> schedule().


find_solution(I, Unscheduled_Jobs, Inputs, Alpha, Beta, Q0, Heuristic,
         Partial_Schedule, Scheduled_Time) ->
    case Unscheduled_Jobs of
    [] -> error ("Ran out of unscheduled jobs in find_solution");
    [Last] -> lists:reverse(Partial_Schedule, [Last]);
    _ ->
        Eta = fitness:make_eta (Unscheduled_Jobs, Heuristic, Inputs, Scheduled_Time),
%       io:format("Eta=~p~n", [Eta]),
        [{I, TauI}] = ets:lookup(tau, I),  % Since tau is a set, we should always get a
        one-entry list.

        {P_i, SumPij} = construct_pij(Eta, Alpha, Beta, TauI, 0.0, []),

        Q = random(),
        New_Job = if Q<Q0 -> find_index_max_P (P_i);
                  % Value of J which % maximises %
                  % {tau^alpha}{eta^beta}[J] % for %
                  % unscheduled % jobs
             true -> find_random_job (P_i, SumPij)
                  % Choose a job randomly wrt to probability distribution
                  % given by P_i
              end,
%       io:format ("New_Job = ~p~n", [New_Job]),
        {Durations,_,_} = Inputs,
        Current_Time = Scheduled_Time+element(New_Job, Durations),
        Now_Unscheduled = lists:delete (New_Job, Unscheduled_Jobs),
        find_solution(I+1, Now_Unscheduled, Inputs, Alpha, Beta, Q0, Heuristic,
              [New_Job|Partial_Schedule], Current_Time)
    end.

-spec find_solution (numjobs(), inputs(), params()) -> solution().
find_solution(Num_Jobs, Inputs, Params) ->
    #params{alpha=Alpha, beta=Beta, q0=Q0, heuristic=Heuristic} = Params,
    Unscheduled_Jobs = lists:seq(1, Num_Jobs),
    Schedule = find_solution(1, Unscheduled_Jobs, Inputs, Alpha, Beta, Q0, Heuristic, [], 0),
    Tardiness = fitness:tardiness(Schedule, Inputs),
    {Tardiness, Schedule}.


%% ---------------------------------------------------------------- %%

-spec loop(numjobs(), inputs(), params()) -> ok.
loop(Num_Jobs, Inputs, Params) ->
    receive
    {From, find_solution} ->
        reset_cyclic_rng(),  % Reset cyclic "RNG" (if we're using it) to get same
        behaviour for all generations.
        Result = find_solution(Num_Jobs, Inputs, Params),
        From ! {ant_done, Result},
        loop(Num_Jobs, Inputs, Params);
    {_From, stop} -> ok;
        Msg ->       error ({unexpected_message, Msg})
    end.
```

Code C.9: Ant Module (ant.erl)

```
%% A single ant colony: spawns a set of ants, loops round getting them to construct new
solutions

-module(ant_colony).
-export([init/5]).

-include ("types.hrl").

% The value 'none' below is used to represent the state at the very start
% when we don't have a current best solution.

-spec best_solution (solution(), solution()|none) -> solution().

best_solution(Solution, none) -> Solution;
best_solution(S1 = {Cost1, _}, S2 = {Cost2, _}) ->
    if
    Cost1 =< Cost2 -> S1;
    true -> S2
    end.


-spec collect_ants(non_neg_integer(), solution() | none) -> solution().

collect_ants (0,Best) -> Best;
collect_ants (Num_left, Current_Best) -> % or use lists:foldl.
    receive
    {ant_done, New_Solution} ->
%       io:format ("collect_ants ~p: got solution ~p~n", [self(), Num_left]),
        collect_ants(Num_left-1, best_solution (New_Solution, Current_Best))
    end.

%-spec aco_loop (pos_integer(), [pid()], solution() | none) -> solution().
aco_loop (0, _, _, _, _, Best_Solution) ->
    Best_Solution;
aco_loop (Iter_Local, Num_Jobs, Inputs, Params, Ants, Best_Solution) ->
    lists:foreach (fun (Pid) -> Pid ! {self(), find_solution} end, Ants),
    New_Solution = collect_ants(length(Ants), none),
%    io:format ("Colony ~p got new solution~n",[self()]),

    {Cost1, _} = New_Solution,
    Improved_Solution = localsearch:vnd_loop(New_Solution, Inputs, Params),

    {Cost, _} = Improved_Solution,

    #params{vverbose=Vverbose} = Params,
    case Vverbose of
    true -> io:format ("Colony ~p: cost = ~p -> ~p~n", [self(), Cost1, Cost]);
    false -> ok
    end,

    New_Best_Solution = best_solution (Improved_Solution, Best_Solution),

    ok = update:update_pheromones(Num_Jobs, New_Best_Solution, Params),
    aco_loop (Iter_Local-1, Num_Jobs, Inputs, Params, Ants, New_Best_Solution).

main (Iter_Local, Num_Jobs, Inputs, Params, Ants, Current_Best) ->
    receive
    {Master, run} ->
%       io:format ("~p: received run in main~n", [self()]),
%       io:format ("Colony ~p got run from ~p~n", [self(),From]),
        New_Solution = aco_loop (Iter_Local, Num_Jobs,
                     Inputs, Params,
```

```
                    Ants, Current_Best),
%       io:format ("tau[1] = ~p~n", [ets:lookup(tau,1)]),
%       io:format ("Colony ~p returning solution ~p~n", [self(), New_Solution]),

        Master ! {colony_done, {New_Solution, self()}},
        main (Iter_Local, Num_Jobs, Inputs, Params, Ants, New_Solution);

    {_Master, {update, Global_Best_Solution}} ->
%       io:format ("~p: received update in main~n", [self()]),
        update:update_pheromones (Num_Jobs, Global_Best_Solution, Params),
        main (Iter_Local, Num_Jobs, Inputs, Params, Ants, Current_Best);

    %% NOTE!!!! Having updated tau according to the global best solution, the
    %% colony carries on with its OWN current best solution.  We could also
    %% carry on with the global best solution:  might this lead to stagnation?

    {Master, stop_ants} ->  % called by master at end of main loop
%       io:format ("~p: received stop_ants in main~n", [self()]),
        ok = lists:foreach (fun (Pid) -> Pid ! {self(), stop} end, Ants),
        ets:delete(tau),
        Master ! ok;  % Note that the value of X!msg is msg, so main returns ok.
    Msg -> error ({"Unexpected message in ant_colony:main", Msg})
    end.



%% For each VM, set up the ETS table and spawn the required number of ants.
%% Then get each VM to perform (say) 50 iterations and report best solution.
%% Send overall best solution to all VMs (or maybe get the owner of the best
%% solution to send it to everyone), and start a new round of iterations.
%% Stop after a certain number of global iterations.

-spec spawn_ants(integer(), list()) -> [pid(),...].
spawn_ants(Num_Ants,Args) -> lists:map (fun (_) -> spawn(ant, start, Args) end, lists:seq(1 ↵
,Num_Ants)).

-spec init(integer(), integer(), numjobs(), inputs(), params()) -> ok.
init (Num_Ants, Iter_Local, Num_Jobs, Inputs, Params) ->

    % Set up ets table: visible to all ants
    case lists:member(tau, ets:all()) of
          true -> ok;  % Already exists: maybe left over from interrupted run.
          false -> ets:new(tau, [set, public, named_table])
      end,
    #params{tau0=Tau0} = Params,
    Tau = lists:map(fun(I) -> {I, util:make_tuple(Num_Jobs, Tau0)} end, lists:seq(1,        ↵
    Num_Jobs)),
    ets:insert(tau, Tau),

%    io:format("~p~n", [ets:tab2list(tau)]),

    Ants = spawn_ants(Num_Ants, [Num_Jobs, Inputs, Params]),
    main(Iter_Local, Num_Jobs, Inputs, Params, Ants, none).

    % We could also put Num_Jobs, Durations, Weights, Deadlines,
    % Alpha, Beta, Q0 (all read_only) in an ETS table and reduce
    % the number of parameters in a lot of function calls.
```

Code C.10: Ant Colony Module (ant_colony.erl)

```
-module(ant_master).
-export([run/7]).
-include("types.hrl").

-spec best_solution ({solution(), pid()}, {solution(),pid()}|none) -> {solution(), pid()}.

best_solution(Solution, none) -> Solution;
best_solution(S1 = {{Cost1, _}, _}, S2 = {{Cost2, _}, _}) ->
    if
    Cost1 =< Cost2 -> S1;
    true -> S2
    end.
% The stuff with the pids is just so that we can see who's produced the best solution


-spec collect_colonies(non_neg_integer(), {solution(), pid()} | none) -> {solution(), pid()}.

collect_colonies (0,{Best, Pid}) ->
    {Best, Pid};
collect_colonies (Num_left, Current_Best) -> % or use lists:foldl.
    receive
    {colony_done, {New_Solution, Pid}} ->
%       {Cost, _} = New_Solution,
%       io:format("Solution ~p from ~p~n", [Cost, Pid]),
        collect_colonies(Num_left-1, best_solution ({New_Solution, Pid}, Current_Best))
    end.


-spec loop(non_neg_integer(), [pid()], {solution(), pid()} | none, inputs(), params()) ->
solution().

loop (0, Colonies, {Best_solution, _}, _Inputs, _Params) ->
    lists:foreach (fun (Pid) -> Pid ! {self(), stop_ants} end, Colonies),
    Best_solution;

loop (N, Colonies, Best_solution, Inputs, Params) ->
%    io:format ("Colonies -> ~p~n", [Colonies]),
    lists:foreach (fun(Pid) -> Pid ! {self(), run} end, Colonies),
    {{Cost, _} = New_Solution, Best_Pid} = collect_colonies (length(Colonies),
    Best_solution),

% Note that we're using the best solution from the previous generation.
% We could also try starting anew, relying on the pheromone matrix to guide
% ants back to earlier best solutions if they can't do any better.

%    Improved_Solution = localsearch:vnd_loop(New_Solution, Inputs, Params),
% local search now done in colony, after every generation of ants.

    Improved_Solution = New_Solution,
    {Cost, _} = Improved_Solution,

    #params{verbose=Verbose} = Params,
    case Verbose of
    true -> io:format ("Iteration ~p: cost = ~p from ~p~n", [N, Cost, Best_Pid]);
    false -> ok
    end,

    lists:foreach (fun(Pid) ->
              if Pid =/= Best_Pid ->
                  Pid ! {self(), {update, Improved_Solution}};
                 true -> ok
              end
```

```
            end, Colonies),

    loop (N-1, Colonies, {Improved_Solution, Best_Pid}, Inputs, Params).


-spec spawn_colonies (list(), [atom()]) -> [pid()].
spawn_colonies (Args, Nodes) ->
    lists:map (fun(H) -> spawn (H, ant_colony, init, Args) end, Nodes).

-spec run (numjobs(), pos_integer(), pos_integer(), pos_integer(), inputs(), params(), list
()) -> solution().
run(Num_Jobs, Num_Ants, Iter_Global, Iter_Local, Inputs, Params, Nodes) ->
    #params{heuristic=Heuristic} = Params,
    Colonies = case Heuristic of
            mixed ->
                Half = length(Nodes) div 2,
                {Nodes_mdd, Nodes_au} = lists:split (Half, Nodes),
                Args_mdd = [Num_Ants, Iter_Local, Num_Jobs, Inputs, Params#params{heuristic=
                mdd}],
                Args_au  = [Num_Ants, Iter_Local, Num_Jobs, Inputs, Params#params{heuristic=
                au}],
                spawn_colonies(Args_mdd, Nodes_mdd) ++ spawn_colonies(Args_au, Nodes_au);

            _ -> Args = [Num_Ants, Iter_Local, Num_Jobs, Inputs, Params],
             spawn_colonies (Args, Nodes)
            end,
    loop (Iter_Global, Colonies, none, Inputs, Params).
```

Code C.11: Ant Master Module (ant_master.erl)

C.4 Multi-Level ACO Source Code

This section contains the source code of Multi-Level Distributed ACO (ML-ACO). The main module, i.e. *ant-submaster*, is listed as Code C.12. All modules and deployment scripts are publicly available at [118].

```
%% Creates a multi-level tree of sub-master nodes to collect the results from Colonies.

%% Author: Amir Ghaffari <Amir.Ghaffari@glasgow.ac.uk>
%% RELEASE project (http://www.release-project.eu/)

-module(ant_submaster).
-compile(export_all).
-include("types.hrl").


%% compares two solutions and returns the better one
%-spec best_solution ({solution(), pid()}, {solution(),pid()}|none) -> {solution(), pid()}.
best_solution(Solution, none) -> Solution;
best_solution(none, Solution) -> Solution;
best_solution(S1 = {{Cost1, _}, _}, S2 = {{Cost2, _}, _}) ->
    if
    Cost1 =< Cost2 -> S1;
    true -> S2
    end.
% The stuff with the pids is just so that we can see who's produced the best solution


%-spec collect_ants(non_neg_integer(), solution() | none) -> solution().

%% collecting the collonies results from child nodes
collect_childs (0,{Best, Pid}) ->
    {Best, Pid};
collect_childs (Num_left, Current_Best) -> % or use lists:foldl.
    receive
    {colony_done, {New_Solution, Pid}} ->
        collect_childs(Num_left-1, best_solution ({New_Solution, Pid}, Current_Best))
    end.

%% processing and passing all the messages from parent to childs and vice versa
loop (ChildProcesses, Best_solution) ->
    receive
    {Parent, run} ->
        lists:foreach (fun(Pid) -> Pid ! {self(), run} end, ChildProcesses),
        {New_Solution, Best_Pid} = collect_childs (length(ChildProcesses), Best_solution),
        Parent ! {colony_done, {New_Solution, Best_Pid}},
        loop (ChildProcesses, {New_Solution, Best_Pid});

    {Best_Pid, {update, Global_Best_Solution}} ->

        lists:foreach (fun(Pid) ->
            if Pid =/= Best_Pid ->
                %Pid ! {self(), {update, Improved_Solution}};
                Pid ! {Best_Pid, {update, Global_Best_Solution}};
                true -> ok
            end
        end, ChildProcesses),

        %lists:foreach (fun(Pid) -> Pid ! {self(), {update, Global_Best_Solution}} end,
        ChildProcesses),
        loop (ChildProcesses, {Global_Best_Solution,Best_Pid});

    {Parent, stop_ants} ->  % called by master at end of main loop
        lists:foreach (fun (Pid) -> Pid ! {self(), stop_ants} end, ChildProcesses),
        lists:foreach (fun (ID) -> receive ok -> ID end end, lists:seq(1,length(
        ChildProcesses))),
        Parent ! ok
    end.
```

```
%% creates appropriate processes on child nodes
run(Num_Jobs, Num_Processes, Num_Ants, Iter_Global, Iter_Local, Inputs, Params, Nodes,
CurrentLevel, MaxLevel, NodeIndex,ProcessIndex) ->
    if
        CurrentLevel==MaxLevel-1 ->
            ChildNodes=get_childs(Num_Processes,CurrentLevel,MaxLevel,Nodes,NodeIndex),
            QuotaOfEachProcess=round(length(ChildNodes)/Num_Processes),
            Start=(ProcessIndex-1)*QuotaOfEachProcess+1,
            MyChildNodes=lists:sublist(ChildNodes, Start, Num_Processes),
            Colonies = lists:map (fun(H) -> spawn (H, ant_colony, init,
                                [Num_Ants, Iter_Local, Num_Jobs,
                                 Inputs, Params]) end, MyChildNodes),
            loop (Colonies,none);
        true ->
            ChildNodes=get_childs(Num_Processes,CurrentLevel,MaxLevel,Nodes,NodeIndex),
            ChildNode=lists:nth(ProcessIndex, ChildNodes),
            ChildProcesses = lists:map(
            fun(NextProcessIndex) ->
            spawn(ChildNode,ant_submaster,run,[Num_Jobs, Num_Processes, Num_Ants,
            Iter_Global, Iter_Local, Inputs, Params,Nodes, CurrentLevel+1, MaxLevel,
            Num_Processes*(NodeIndex-1)+ProcessIndex,NextProcessIndex])
            end,
            lists:seq(1,Num_Processes)),
            loop (ChildProcesses,none)
    end.

%% auxiliary functions to create a tree of submaster nodes

%% calculates the number of levels in the tree
%% each node has "Num_Processes" processes
%% each process supervise one node in lower level, except the last level that each process
supervises "Num_Processes" number of nodes

%% calculates the tree level based on the total number of nodes and node degree (degree of
vertices)
find_max_level(Num_Nodes,Num_Processes) ->
    S=speculate_level(Num_Nodes,Num_Processes),
    find_max_level(Num_Nodes,Num_Processes,S).

find_max_level(Num_Nodes,Num_Processes,Speculated_level) ->
    Result=calc_formula(Num_Processes,Speculated_level),
    if
        Result ==0 -> {_Level=0,_NumNodes=0};
        Result =< Num_Nodes -> {Speculated_level,Result};
        true -> find_max_level(Num_Nodes,Num_Processes,Speculated_level-1)
    end.

%% finds the largest possible power for Num_Processes to be less than Num_Nodes
speculate_level(Num_Nodes,Num_Processes)->
    speculate_level(Num_Nodes,Num_Processes,0).

speculate_level(Num_Nodes,Num_Processes,Acc)->
    Result=math:pow(Num_Processes,Acc),
    if
         Result<Num_Nodes ->
             speculate_level(Num_Nodes,Num_Processes,Acc+1);
        true ->
            round(Acc-1)
    end.

%% calculates 1+P^1+P^2+...+P^(N-2)+P^N
```

```
calc_formula(Num_Processes,Level) when Level>=2 ->
    calc_formula(Num_Processes,Level,_Acc=0,_Current_level=0);

%% No submaster can be allocated
calc_formula(_Num_Processes,_Level) ->
    0.

calc_formula(Num_Processes,Last_Level,Acc,Current_level) when Current_level=<Last_Level ->
    Num_Nodes=math:pow(Num_Processes,Current_level),
    case Current_level+2 of
        Last_Level ->
            calc_formula(Num_Processes,Last_Level,Acc+Num_Nodes,Current_level+2);
        _->
            calc_formula(Num_Processes,Last_Level,Acc+Num_Nodes,Current_level+1)
    end;

calc_formula(_Num_Processes,_Last_Level,Acc,_Current_level) ->
    round(Acc).

%% returns number of nodes for a specific level
nodes_in_level(Num_Processes, Level) ->
    round(math:pow(Num_Processes,Level-1)).

%% returns all the child nodes of a specific node. Node is specified by its level and its index in the level
%% How to test: Nodes=lists:seq(1, 277).
list_to_tuple(ant_submaster:get_childs(4,3,4,Nodes,1)).
get_childs(Num_Processes,CurrentLevel,MaxLevel,Nodes,IndexInLevel) -> %when CurrentLevel<MaxLevel ->
    Num_Node_in_Current_Level=nodes_in_level(Num_Processes,CurrentLevel),
    if
        Num_Node_in_Current_Level<IndexInLevel ->
            throw(index_in_level_is_more_than_num_nodes_in_level);
        true -> ok
    end,
    if
        CurrentLevel>=MaxLevel ->
            Num_Node_in_Next_Level=0,
            throw(current_level_must_be_less_than_max_level);
        CurrentLevel==MaxLevel-1 ->
            Num_Node_in_Next_Level=nodes_in_level(Num_Processes,CurrentLevel+2);
        true->
            Num_Node_in_Next_Level=nodes_in_level(Num_Processes,CurrentLevel+1)
    end,
    Childs_Per_Node=Num_Node_in_Next_Level/Num_Node_in_Current_Level,
    Index_For_Next_Line=Childs_Per_Node*(IndexInLevel-1)+1,
    After_Me_This_Level=Num_Node_in_Current_Level-IndexInLevel,
    Child_index=level_index(Num_Processes,CurrentLevel)+IndexInLevel+After_Me_This_Level+
    Index_For_Next_Line,lists:sublist(Nodes, round(Child_index), round(Childs_Per_Node)).

%% returns the index of the first node for a specific level
level_index(Num_Processes,Level) ->
    level_index(Num_Processes,Level,0,0).

level_index(Num_Processes,Level,Acc,CurrentLevel) when CurrentLevel<Level-1 ->
    R=math:pow(Num_Processes,CurrentLevel),
    level_index(Num_Processes,Level,Acc+R,CurrentLevel+1);

level_index(_Num_Processes,_Level,Acc,_CurrentLevel)->
Acc.
```

Code C.12: Sub-Master Module (ant_submaster.erl)

C.5 Reliable ACO Source Code

This section contains the source code of Reliable Distributed ACO (R-ACO). The main affected modules by reliability, i.e. *ant*, *colony*, *master*, *submaster*, and *supervisor* are listed as Codes C.13, C.14, C.15, C.16, and C.17. respectively. All modules and deployment scripts are publicly available at [118].

```
%% Individual ant: constructs new solution based on pheromone matrix.

-module(ant).
-include ("types.hrl").
-export([start/3]).

% M&M -> Merkle & Middendorf: An Ant Algorithm with a new
% Pheromone Evaluation Rule for Total Tardiness Problems

%% ----------------------------------------------------------------- %%
% Strictly, P(i,j) is equal to Pij/Sigma; he're we're calculating both as we go along.

construct_pij([], _, _, _, Sigma, P) -> % Sigma is sum of terms over unscheduled jobs: see M&M
    {lists:reverse(P), Sigma};
construct_pij([{J,Eta_J}|TEta], Alpha, Beta, TauI, Sigma, P) ->
    Pij = math:pow (element(J, TauI), Alpha) * math:pow(Eta_J, Beta),
    construct_pij(TEta, Alpha, Beta, TauI, Sigma+Pij, [{J,Pij}|P]).

% Eta contains info only for unscheduled jobs, so the above function constructs info only for those jobs.



%% ----------------------------------------------------------------- %%
% Careful here: elements of Pij can be extremely small (10^-200 towards end).

-spec find_limit_index ([idxf90], index(), float(), float()) -> index().
find_limit_index(P, Idx, Total, Limit) ->
    if Total >= Limit -> Idx;
       true -> case P of
           [] -> error ("Fell off end of P in find_limit_index");
                       % This could happen if Limit is 1 (or very close)
                       % and rounding errors make the sum of pij slightly
                       % less than 1.
           [{K, Kval}|Rest] ->
               find_limit_index (Rest, K, Total+Kval, Limit)
           end
    end.


-spec find_random_job([idxf()], float()) -> index().
find_random_job(P_i, SumPij) ->
    case P_i of
    [] -> error ("find_rand_pij: P_i is empty");
    [{J1, J1val}|Rest] ->
        Limit = random:uniform() * SumPij,
        find_limit_index(Rest, J1, J1val, Limit)
    end.



%% ----------------------------------------------------------------- %%

% Find index j which maximises P[i,j] (i is fixed).
% Note that we're usually working with a partial schedule, in which case many of the
% elements of Pij are zero.  We need to be sure that at least one element is greater
% than zero when calling this function.


-spec find_index_max_P(index(), float(), [idxf()]) -> index().
```

```
    Schedule = find_solution(1, Unscheduled_Jobs, Inputs, Alpha, Beta, Q0, Heuristic, [], 0),
    Tardiness = fitness:tardiness(Schedule, Inputs),
    {Tardiness, Schedule}.


%% ---------------------------------------------------------------- %%

%-spec loop(numjobs(), inputs(), params()) -> ok.

%loop(Num_Jobs, Inputs, Params) -> loop(Num_Jobs, Inputs, Params, 0).

%loop(Num_Jobs, Inputs, Params, 0) ->
%    receive
%   {From, find_solution} ->
%       Result = find_solution(Num_Jobs, Inputs, Params),
%       From ! {ant_done, self(), Result},
%       loop(Num_Jobs, Inputs, Params);
%   {_From, stop} -> ok;
%       Msg ->       error ({unexpected_message, Msg})
%   after
%       ?TimeOut ->
%           io:format ("Ant Timeout with pid ~p ~n", [self()]),
%           loop(Num_Jobs, Inputs, Params, 1)
%   end;

%loop(Num_Jobs, Inputs, Params, 1) ->
%    receive
%   {From, find_solution} ->
%       Result = find_solution(Num_Jobs, Inputs, Params),
%       From ! {ant_done, self(), Result},
%       loop(Num_Jobs, Inputs, Params);
%   {_From, stop} -> ok;
 %       Msg ->      error ({unexpected_message, Msg})
%    end.

-spec loop(numjobs(), inputs(), params()) -> ok.
loop(Num_Jobs, Inputs, Params) ->
    receive
    {From, find_solution} ->
        Result = find_solution(Num_Jobs, Inputs, Params),
        From ! {ant_done, self(), Result},
        loop(Num_Jobs, Inputs, Params);
    {_From, stop} -> ok;
       Msg ->       error ({unexpected_message, Msg})
    end.


-spec start(numjobs(), inputs(), params()) -> ok.
start(Num_Jobs, Inputs, Params) ->
    #params{seed_now=Seed_now} = Params,
    if Seed_now ->
        {A, B, C} = now();
       true ->
        <<A:32,B:32,C:32>> = crypto:rand_bytes(12)
            % This seems to be safer, but the HW machines don't have the
        % correct C crypto library.
    end,
    random:seed(A,B,C),
    loop(Num_Jobs, Inputs, Params).
```

```
    Schedule = find_solution(1, Unscheduled_Jobs, Inputs, Alpha, Beta, Q0, Heuristic, [], 0),
    Tardiness = fitness:tardiness(Schedule, Inputs),
    {Tardiness, Schedule}.


%% ---------------------------------------------------------------- %%

%-spec loop(numjobs(), inputs(), params()) -> ok.

%loop(Num_Jobs, Inputs, Params) -> loop(Num_Jobs, Inputs, Params, 0).

%loop(Num_Jobs, Inputs, Params, 0) ->
%    receive
%   {From, find_solution} ->
%       Result = find_solution(Num_Jobs, Inputs, Params),
%       From ! {ant_done, self(), Result},
%       loop(Num_Jobs, Inputs, Params);
%   {_From, stop} -> ok;
%       Msg ->       error ({unexpected_message, Msg})
%   after
%       ?TimeOut ->
%           io:format ("Ant Timeout with pid ~p ~n", [self()]),
%           loop(Num_Jobs, Inputs, Params, 1)
%   end;

%loop(Num_Jobs, Inputs, Params, 1) ->
%    receive
%   {From, find_solution} ->
%       Result = find_solution(Num_Jobs, Inputs, Params),
%       From ! {ant_done, self(), Result},
%       loop(Num_Jobs, Inputs, Params);
%   {_From, stop} -> ok;
 %       Msg ->      error ({unexpected_message, Msg})
%    end.

-spec loop(numjobs(), inputs(), params()) -> ok.
loop(Num_Jobs, Inputs, Params) ->
    receive
    {From, find_solution} ->
        Result = find_solution(Num_Jobs, Inputs, Params),
        From ! {ant_done, self(), Result},
        loop(Num_Jobs, Inputs, Params);
    {_From, stop} -> ok;
       Msg ->        error ({unexpected_message, Msg})
    end.


-spec start(numjobs(), inputs(), params()) -> ok.
start(Num_Jobs, Inputs, Params) ->
    #params{seed_now=Seed_now} = Params,
    if Seed_now ->
        {A, B, C} = now();
       true ->
        <<A:32,B:32,C:32>> = crypto:rand_bytes(12)
            % This seems to be safer, but the HW machines don't have the
        % correct C crypto library.
    end,
    random:seed(A,B,C),
    loop(Num_Jobs, Inputs, Params).
```

Code C.13: Ant Module (ant.erl)

```
%% A single ant colony: spawns a set of ants, loops round getting them to construct new solutions

-module(ant_colony).
-export([init/8]).

-include ("types.hrl").

% The value 'none' below is used to represent the state at the very start
% when we don't have a current best solution.

-spec best_solution (solution(), solution()|none) -> solution().

best_solution(Solution, none) -> Solution;
best_solution(none, Solution) -> Solution; % added just for test amir
best_solution(S1 = {Cost1, _}, S2 = {Cost2, _}) ->
    if
    Cost1 =< Cost2 -> S1;
    true -> S2
    end.


-spec collect_ants(non_neg_integer(), solution() | none, list(), colony_state(), list()) ->
 solution().

collect_ants (0,Best, Ants, _Colony_state, _ReceivedPIDs) ->
{Ants, Best};
collect_ants (Num_left, Current_Best, Ants, Colony_state, ReceivedPIDs) -> % or use lists:foldl.
    receive
    {ant_done, PID, New_Solution} ->
        collect_ants(Num_left-1, best_solution (New_Solution, Current_Best), Ants, Colony_state, [PID]++ReceivedPIDs);
    {'EXIT', FailedPID, Reason} ->
        io:format ("Failure of an ant process with PID ~p and reason ~p ~n", [FailedPID, Reason]),
        case recover_childs(FailedPID, Colony_state, Ants) of
        {no_updated, Ants}->
            collect_ants(Num_left, Current_Best, Ants, Colony_state, ReceivedPIDs);
        {updated, NewAnts}->
            case lists:member(FailedPID, ReceivedPIDs) of
                true-> collect_ants(Num_left, Current_Best, NewAnts, Colony_state , ReceivedPIDs);
                _->    collect_ants(Num_left-1, Current_Best, NewAnts, Colony_state, ReceivedPIDs)
            end
        end
    after
        ?TimeOut ->
            io:format ("Timeout has occurred on colony process ~p on node ~p ~n", [self(), node()]),
            collect_ants (Num_left, Current_Best, Ants, Colony_state, ReceivedPIDs)
    end.

%-spec aco_loop (pos_integer(), [pid()], solution() | none) -> solution().
aco_loop (0, _, _, _, Ants, Best_Solution, Colony_state) ->
    {Ants, Best_Solution, Colony_state};
aco_loop (Iter_Local, Num_Jobs, Inputs, Params, Ants, Best_Solution, Colony_state) ->
    lists:foreach (fun (Pid) -> Pid ! {self(), find_solution} end, Ants),
    {New_Ants, New_Solution} = collect_ants(length(Ants), none, Ants, Colony_state, _ReceivedPIDs=[]),
%    io:format ("Colony ~p got new solution~n",[self()]),
```

```
    {Cost1, _} = New_Solution,
    Improved_Solution = localsearch:vnd_loop(New_Solution, Inputs, Params),

    {Cost, _} = Improved_Solution,

    #params{vverbose=Vverbose} = Params,
    case Vverbose of
    true -> io:format ("Colony ~p: cost = ~p -> ~p~n", [self(), Cost1, Cost]);
    false -> ok
    end,

    New_Best_Solution = best_solution (Improved_Solution, Best_Solution),

    ok = update:update_pheromones(Num_Jobs, New_Best_Solution, Params),
    aco_loop (Iter_Local-1, Num_Jobs, Inputs, Params, New_Ants, New_Best_Solution,
    Colony_state).

main (Duplicating, Iter_Local, Num_Jobs, Inputs, Params, Ants, Current_Best, Colony_state) ->
    receive
    {Master, run} ->
%       io:format ("Colony ~p got run from ~p~n", [self(),From]),
        {New_Ants, New_Solution, New_Colony_state} = aco_loop (Iter_Local, Num_Jobs,
                    Inputs, Params,
                    Ants, Current_Best, Colony_state),
%       io:format ("tau[1] = ~p~n", [ets:lookup(tau,1)]),
%       io:format ("Colony ~p returning solution ~p~n", [self(), New_Solution]),

        %Master ! {colony_done, {New_Solution, self()}},
        lists:foreach (fun (_ID) -> Master ! {colony_done, {New_Solution, self()}} end,
        lists:seq(1, Duplicating)),
        main (Duplicating, Iter_Local, Num_Jobs, Inputs, Params, New_Ants, New_Solution,
        New_Colony_state);

    {_Master, {update, Global_Best_Solution}} ->
        update:update_pheromones (Num_Jobs, Global_Best_Solution, Params),
        main (Duplicating, Iter_Local, Num_Jobs, Inputs, Params, Ants, Current_Best,
        Colony_state);

    %% NOTE!!!! Having updated tau according to the global best solution, the
    %% colony carries on with its OWN current best solution.  We could also
    %% carry on with the global best solution:  might this lead to stagnation?

    {Master, stop_ants} ->  % called by master at end of main loop
        ok = lists:foreach (fun (Pid) -> Pid ! {self(), stop} end, Ants),
        ets:delete(tau),
        Master ! ok
    %Msg -> error ({"Unexpected message in ant_colony:main", Msg})
    end.



%% For each VM, set up the ETS table and spawn the required number of ants.
%% Then get each VM to perform (say) 50 iterations and report best solution.
%% Send overall best solution to all VMs (or maybe get the owner of the best
%% solution to send it to everyone), and start a new round of iterations.
%% Stop after a certain number of global iterations.

-spec spawn_ants(integer(), list()) -> [pid()].
spawn_ants(Num_Ants,Args) -> lists:map (fun (_) -> spawn_link(ant, start, Args) end, lists:
seq(1,Num_Ants)).
```

```
-spec init(integer(), integer(), integer(), numjobs(), inputs(), params(), list(), boolean)
 -> pid().
init (Num_Ants, Duplicating, Iter_Local, Num_Jobs, Inputs, Params, ProcessName, Recovery) ->
    process_flag(trap_exit, true),
    global:register_name(ProcessName,self()),
    % Set up ets table: visible to all ants
    case lists:member(tau, ets:all()) of
        true -> %ok;  % Already exists: maybe left over from interrupted run.
            ok;
        false -> ets:new(tau, [set, public, named_table])
      end,
    #params{tau0=Tau0} = Params,
    Tau = lists:map(fun(I) -> {I, util:make_tuple(Num_Jobs, Tau0)} end, lists:seq(1,
    Num_Jobs)),
    ets:insert(tau, Tau),

%    io:format("~p~n", [ets:tab2list(tau)]),

    Colony_state=#colony_state{num_Jobs=Num_Jobs, inputs=Inputs, params=Params},
    Ants = spawn_ants(Num_Ants, [Num_Jobs, Inputs, Params]),

    #params{chaos=Chaos} = Params,
    if
        Chaos==true andalso Recovery==false->
            Chaos_starter=whereis(chaos_starter),
            io:format("ant colony - sending ~p pids from node ~p to ~p ~n" , [length(Ants),
             node(), Chaos_starter]),
            Chaos_starter! {pids, Ants};
        true ->
            ok
    end,


    main(Duplicating, Iter_Local, Num_Jobs, Inputs, Params, Ants, none, Colony_state),
    self().

    % We could also put Num_Jobs, Durations, Weights, Deadlines,
    % Alpha, Beta, Q0 (all read_only) in an ETS table and reduce
    % the number of parameters in a lot of function calls.

%% Recovers a failed process
recover_childs(FailedPID, Colony_state, Ants) ->
    #colony_state{num_Jobs=Num_Jobs, inputs=Inputs, params=Params}=Colony_state,
    case lists:member(FailedPID, Ants) of
        true ->
            Index=ant_master:index_of(FailedPID, Ants),
            PID=spawn_link(ant, start, [Num_Jobs, Inputs, Params]),
            util:send_pid(PID),
            NewAnts=lists:sublist(Ants,1,Index-1)++[PID]++lists:sublist(Ants,Index+1,length
            (Ants)),
            io:format ("recovery of ant process ~p on node ~p by new length ~p and new PID
            ~p ~n", [FailedPID, node(), length(NewAnts), PID]),
            {updated, NewAnts};
        _->
            io:format ("No recovery for ant process ~p is needed on node ~p ~n", [FailedPID
            , node()]),
            {no_updated, Ants}
    end.
```

Code C.14: Ant Colony Module (ant_colony.erl)

```
-module(ant_master).
-compile(export_all).
-include("types.hrl").

%-spec best_solution ({solution(), pid()}, {solution(),pid()}|none) -> {solution(), pid()}.

best_solution(Solution, none) -> Solution;
best_solution(none, Solution) -> Solution;
best_solution(S1 = {{Cost1, _}, _}, S2 = {{Cost2, _}, _}) ->
    if
    Cost1 =< Cost2 -> S1;
    true -> S2
    end.
% The stuff with the pids is just so that we can see who's produced the best solution


%-spec collect_ants(non_neg_integer(), solution() | none) -> solution().

collect_colonies (0,{Best, Pid}, _Master_state, NewColonies, _ReceivedPIDs, _Duplicating) ->
    {Best, Pid, NewColonies};
collect_colonies (Num_left, Current_Best, Master_state, Colonies, ReceivedPIDs, Duplicating
) -> % or use lists:foldl.
    receive
    {colony_done, {New_Solution, Pid}} ->
%       {Cost, _} = New_Solution,
%       io:format("Solution ~p from ~p~n", [Cost, Pid]),
        collect_colonies(Num_left-1, best_solution ({New_Solution, Pid}, Current_Best),
        Master_state, Colonies, ant_submaster:key_value_increment(Pid, ReceivedPIDs),
        Duplicating);
    {'EXIT', FailedPID, Reason} ->
        #master_state{nodes=Nodes, num_Processes=Num_Processes} = Master_state,
        case ant_submaster:find_max_level(length(Nodes),Num_Processes) of
        {_MaxLevels=0,_NodeLength=0} ->
            io:format ("Failure of a colony process with PID ~p detected by master process
            and reason ~p ~n", [FailedPID, Reason]);
        {_MaxLevels,_NodeLength} ->
            io:format ("Failure of a sub-master process on the first level with PID ~p and
            reason ~p ~n", [FailedPID, Reason])
        end,
        case recover_childs(FailedPID, Master_state, Colonies) of
        {no_updated, Colonies}->
            collect_colonies(Num_left, Current_Best, Master_state, Colonies, ReceivedPIDs,
            Duplicating);
        {updated, NewColonies}->
            case ant_submaster:key_search(FailedPID, ReceivedPIDs) of
                []-> collect_colonies(Num_left-(1*Duplicating), Current_Best, Master_state,
                 NewColonies, ReceivedPIDs, Duplicating);
                Value-> collect_colonies(Num_left-(Duplicating-Value), Current_Best,
                Master_state, NewColonies, ReceivedPIDs, Duplicating)
            end
        end
    after
        ?TimeOut ->
            io:format ("Timeout has occured in master process ~p on node ~p with Num_left
            ~p and received ~p ~n", [self(), node(), Num_left, ReceivedPIDs]),
            collect_colonies (Num_left, Current_Best, Master_state, Colonies, ReceivedPIDs,
             Duplicating)
    end.


loop (_Duplicating, 0, Colonies, {Best_solution, _}, _Inputs, _Params, _Master_state) ->
    lists:foreach (fun ({Pid, _ProcessName, _ProcessIndex}) -> Pid ! {self(), stop_ants}
```

```
    end, Colonies),
    lists:foreach (fun (ID) -> receive ok -> ID; {'EXIT', _FailedPID, _Reason} -> ok end
    end, lists:seq(1,length(Colonies))), %% wait to collect all the acknowledges
    Best_solution;

loop (Duplicating, N, Colonies, Best_solution, Inputs, Params, Master_state) ->
    lists:foreach (fun({Pid, _ProcessName, _ProcessIndex}) -> Pid ! {self(), run} end,
    Colonies),
    {{Cost, _} = New_Solution, Best_Pid, NewColonies} = collect_colonies (length(Colonies)*
    Duplicating, Best_solution, Master_state, Colonies, _ReceivedPIDs=[], Duplicating),

% Note that we're using the best solution from the previous generation.
% We could also try starting anew, relying on the pheromone matrix to guide
% ants back to earlier best solutions if they can't do any better.

%    Improved_Solution = localsearch:vnd_loop(New_Solution, Inputs, Params),
% local search now done in colony, after every generation of ants.

    Improved_Solution = New_Solution,
    {Cost, _} = Improved_Solution,

    #params{verbose=Verbose, schedule=Print_schedule} = Params,
    case Verbose of
    true -> io:format ("Iteration ~p: cost = ~p from ~p~n", [N, Cost, Best_Pid]),
        case Print_schedule of
            true -> io:format ("Schedule -> ~p~n", [Improved_Solution]);
            false -> ok
        end;
    false -> ok
    end,

    lists:foreach (fun({Pid, _ProcessName, _ProcessIndex}) ->
              if Pid =/= Best_Pid ->
                  %Pid ! {self(), {update, Improved_Solution}};
                  Pid ! {Best_Pid, {update, Improved_Solution}};
                 true -> ok
              end
           end, Colonies),

    #params{chaos=Chaos} = Params,
    if
        Chaos==true ->
            #master_state{nodes=Nodes} = Master_state,
            lists:foreach (fun(Node) -> Chaos_starter=rpc:call(Node, erlang, whereis,[
            chaos_starter]), Chaos_starter! {run_the_chaos} end, Nodes),
            loop (Duplicating, N-1, NewColonies, {Improved_Solution, Best_Pid}, Inputs,
            Params#params{chaos=false}, Master_state);
        true ->
            loop (Duplicating, N-1, NewColonies, {Improved_Solution, Best_Pid}, Inputs,
            Params, Master_state)
    end.


run(Num_Jobs, Num_Processes, Duplicating, Num_Ants, Iter_Global, Iter_Local, Inputs, Params
, Nodes, Recovery) ->
    if
        Recovery -> io:format ("Master process restart at ~p on node ~p ~n", [time(), node
        ()]);
        true -> ok
    end,
    process_flag(trap_exit, true),
    Master_state=#master_state{num_Jobs=Num_Jobs, num_Processes=Num_Processes, duplicating=
```

```
Duplicating, num_Ants=Num_Ants, iter_Global=Iter_Global, iter_Local=Iter_Local, inputs=
Inputs, params=Params, nodes=Nodes},
case ant_submaster:find_max_level(length(Nodes),Num_Processes) of
{_MaxLevels=0,_NodeLength=0} ->
    io:format("Number of nodes (~p) with ~p processes per node is not enough to have
    any submaster node~n", [length(Nodes),Num_Processes]),
    Colonies = lists:map (fun(H) ->
                            ProcessIndex=index_of(H, Nodes),
                            ProcessName=list_to_atom("colony_node_"++integer_to_list(
                            ProcessIndex)),
                            case global:whereis_name(ProcessName) of
                            undefined ->
                                ChildPID=spawn_link(H, ant_colony, init, [Num_Ants,
                                Duplicating, Iter_Local, Num_Jobs, Inputs, Params,
                                ProcessName, Recovery]);
                            ChildPID -> link(ChildPID)
                            end,
                            {ChildPID, ProcessName, ProcessIndex}
                          end, Nodes),
    global:register_name(master,self()),
    ChildPIDs=lists:map (fun({ChildPID, _ProcessName, _ProcessIndex}) -> ChildPID end,
    Colonies),
    #params{chaos=Chaos} = Params,
    if
        Chaos==true andalso Recovery==false ->
            Chaos_starter=whereis(chaos_starter),
            %io:format("master - Sending ~p pids from node ~p to ~p ~n" ,
            [length(ChildPIDs)+1, node(), Chaos_starter]),
            %Chaos_starter! {pids, [self()]++ChildPIDs};
            io:format("master - Sending ~p pids from node ~p to ~p ~n" , [length(
            ChildPIDs), node(), Chaos_starter]),
            Chaos_starter! {pids, ChildPIDs};
        true ->
            ok
    end,
    Results=loop(Duplicating, Iter_Global, Colonies, none, Inputs, Params, Master_state);
{MaxLevels,NodeLength} ->
    io:format("We have ~p levels for ~p nodes and ~p processes per node (~p nodes will
    be used) ~n", [MaxLevels,length(Nodes),Num_Processes,NodeLength]),
    {UsingNodes,_Rest}=lists:split(NodeLength, Nodes),
    [Head|_Tail]=UsingNodes,
    ChildProcesses=lists:map(
        fun(ProcessIndex) ->
            ProcessName=ant_submaster:generate_submaster_name(_CurrentLevel=1,
            _NodeIndex=1, ProcessIndex),
            case global:whereis_name(ProcessName) of
            undefined ->
                ChildPID=spawn_link(Head, ant_submaster, run, [Num_Jobs, Num_Processes,
                 Duplicating, Num_Ants, Iter_Global, Iter_Local, Inputs, Params,
                UsingNodes, _CurrentLevel=1, MaxLevels, _NodeIndex=1, ProcessIndex,
                ProcessName, Recovery]),
                {ChildPID, ProcessName, ProcessIndex};
            ChildPID -> link(ChildPID),
                {ChildPID, ProcessName, ProcessIndex}
            end
        end,
    lists:seq(1,Num_Processes)),
    global:register_name(master,self()),
    ChildPIDs=lists:map (fun({ChildPID, _ProcessName, _ProcessIndex}) -> ChildPID end,
    ChildProcesses),
    #params{chaos=Chaos} = Params,
    if
```

```
            Chaos==true andalso Recovery==false ->
                Chaos_starter=whereis(chaos_starter),
                io:format("master - sending ~p pids from node ~p to ~p ~n" , [length(
                ChildPIDs)+1, node(), Chaos_starter]),
                Chaos_starter! {pids, ChildPIDs};
            true ->
                ok
        end,
        Results=loop(1, Iter_Global, ChildProcesses, none, Inputs, Params, Master_state)
    end,
    Starter=util:get_global_name(starter),
    Starter ! {self(), Results}.

%% Recovers a failed process
recover_childs(FailedPID, Master_state, Colonies) ->
    #master_state{
    num_Jobs=Num_Jobs,
    num_Processes=Num_Processes,
    duplicating=Duplicating,
    num_Ants=Num_Ants,
    iter_Global=Iter_Global,
    iter_Local=Iter_Local,
    inputs=Inputs,
    params=Params,
    nodes=Nodes} = Master_state,
    case get_process_name(FailedPID, Colonies) of
    not_found ->
        io:format ("No recovery for colony process ~p is needed on node ~p ~n", [FailedPID,
         node()]),
        {no_updated, Colonies};
    {ProcessName, ProcessIndex} ->
        case ant_submaster:find_max_level(length(Nodes),Num_Processes) of
        {_MaxLevels=0,_NodeLength=0} ->
            Node=lists:nth(ProcessIndex, Nodes),
            NewPID=spawn_link (Node, ant_colony, init,[Num_Ants, Duplicating, Iter_Local,
            Num_Jobs, Inputs, Params,ProcessName, _Recovery=true]),
            util:send_pid(NewPID),
            NewChildProcesses=update_process_PID(ProcessName, NewPID, Colonies),
            io:format ("recovery of a colony process ~p detected by master process on node
            ~p by new length ~p and new PID ~p ~n", [FailedPID, node(), length(
            NewChildProcesses), NewPID]);
        {MaxLevels,NodeLength} ->
            {UsingNodes,_Rest}=lists:split(NodeLength, Nodes),
            [Head|_Tail]=UsingNodes,
            NewPID=spawn_link(Head, ant_submaster, run, [Num_Jobs, Num_Processes,
            Duplicating, Num_Ants, Iter_Global, Iter_Local, Inputs, Params,UsingNodes,
            _CurrentLevel=1, MaxLevels, _NodeIndex=1, ProcessIndex, ProcessName, _Recovery=
            true]),
            util:send_pid(NewPID),
            NewChildProcesses=update_process_PID(ProcessName, NewPID, Colonies),
            io:format ("recovery of a sub-master process ~p detected by master process on
            node ~p by new length ~p and new PID ~p ~n", [FailedPID, node(), length(
            NewChildProcesses), NewPID])
        end,
        {updated, NewChildProcesses}
    end.

%% gets a process identity and return its registred name
get_process_name(_ProcessPID,_Colonies=[]) ->
    not_found;
get_process_name(ProcessPID, [{ProcessPID, ProcessName, ProcessIndex}|_Tail]) ->
    {ProcessName,ProcessIndex};
```

```
get_process_name(ProcessPID, [{_Process, _ProcessName, _ProcessIndex}|Tail]) ->
    get_process_name(ProcessPID,Tail).

%% updates the pid of a failed process with the new pid
update_process_PID(ProcessName, NewPID, Colonies) ->
    update_process_PID(ProcessName, NewPID, Colonies, _ACC=[]).

update_process_PID(_ProcessName, _NewPID, [], _ACC)->
        throw(cannot_find_process_name_to_update);

update_process_PID(ProcessName, NewPID, [{_Process, ProcessName, ProcessIndex}|Tail], ACC)->
        ACC++[{NewPID, ProcessName, ProcessIndex}]++Tail;

update_process_PID(ProcessName, NewPID, [{ProcessID, ProcessName2, ProcessIndex}|Tail], ACC
)->
        update_process_PID(ProcessName, NewPID, Tail, ACC++[{ProcessID, ProcessName2,
        ProcessIndex}]).

%% returns the index of an element of a list
index_of(Item, List) -> index_of(Item, List, 1).
index_of(_, [], _)  -> not_found;
index_of(Item, [Item|_], Index) -> Index;
index_of(Item, [_|Tl], Index) -> index_of(Item, Tl, Index+1).
```

Code C.15: Ant Master Module (ant_master.erl)

```
%% Creates a multi-level tree of sub-master nodes to collect the results from Colonies.

%% Author: Amir Ghaffari <Amir.Ghaffari@glasgow.ac.uk>

%% RELEASE project (http://www.release-project.eu/)

-module(ant_submaster).
-compile(export_all).
-include("types.hrl").


%% compares two solutions and returns the better one
%-spec best_solution ({solution(), pid()}, {solution(),pid()}|none) -> {solution(), pid()}.
best_solution(Solution, none) -> Solution;
best_solution(none, Solution) -> Solution;
best_solution(S1 = {{Cost1, _}, _}, S2 = {{Cost2, _}, _}) ->
    if
    Cost1 =< Cost2 -> S1;
    true -> S2
    end.
% The stuff with the pids is just so that we can see who's produced the best solution


%-spec collect_ants(non_neg_integer(), solution() | none) -> solution().

%% collecting the collonies results from child nodes
collect_childs (0,{Best, Pid}, _SubMaster_state, NewChildProcesses, _ReceivedPIDs,
_Duplicating) ->
    {Best, Pid, NewChildProcesses};
collect_childs (Num_left, Current_Best, SubMaster_state, ChildProcesses, ReceivedPIDs,
Duplicating) -> % or use lists:foldl.
    receive
    {colony_done, {New_Solution, Pid}} ->
        collect_childs(Num_left-1, best_solution ({New_Solution, Pid}, Current_Best),
        SubMaster_state, ChildProcesses, key_value_increment(Pid, ReceivedPIDs),
        Duplicating);
    {'EXIT', FailedPID, Reason} ->
        #submaster_state{currentLevel=CurrentLevel, maxLevel=MaxLevel} = SubMaster_state,
        if
            CurrentLevel==MaxLevel-1 ->
                io:format ("Failure of a colony process with PID ~p and reason ~p ~n", [
                FailedPID, Reason]);
            true ->
                io:format ("Failure of a sub-master process with PID ~p and reason ~p ~n",
                [FailedPID, Reason])
        end,
        case recover_childs(FailedPID, SubMaster_state, ChildProcesses) of
            {no_updated, ChildProcesses}->
                collect_childs(Num_left, Current_Best, SubMaster_state, ChildProcesses,
                ReceivedPIDs, Duplicating);
            {updated, NewChildProcesses}->
                %NewSubMaster_state=SubMaster_state#submaster_state{status=1},
                case key_search(FailedPID, ReceivedPIDs) of
                    []->    collect_childs(Num_left-(1*Duplicating), Current_Best,
                    SubMaster_state, NewChildProcesses, ReceivedPIDs, Duplicating);
                    Value-> collect_childs(Num_left-(Duplicating-Value), Current_Best,
                    SubMaster_state, NewChildProcesses, ReceivedPIDs, Duplicating)
                end
        end
    after
        ?TimeOut ->
        #submaster_state{currentLevel=CurrentLevel, maxLevel=MaxLevel} = SubMaster_state,
```

```
        if
            CurrentLevel==MaxLevel-1 ->
                io:format ("Timeout has occurred on a last level sub-master process ~p on
                node ~p ~n", [self(), node()]),
                collect_childs (Num_left, Current_Best, SubMaster_state, ChildProcesses,
                ReceivedPIDs, Duplicating);
            true ->
                io:format ("Timeout has occurred on a sub-master process ~p on node ~p ~n",
                 [self(), node()]),
                collect_childs (Num_left, Current_Best, SubMaster_state, ChildProcesses,
                ReceivedPIDs, Duplicating)
        end
    end.

%% return the value of a key in a list of {key,value} tuples
key_search(_Key, []) ->
    [];
key_search(Key, [{Key, Val}|_Tail]) ->
    Val;
key_search(Key, [{_Key2, _Val}|Tail]) ->
    key_search(Key, Tail).

key_value_increment(Key, List) ->
    key_value_increment(Key, List, _Acc=[]).

key_value_increment(Key, [], Acc) ->
    Acc++[{Key,1}];

key_value_increment(Key, [{Key, Val}|Tail], Acc) ->
    Acc++[{Key, Val+1}]++Tail;

key_value_increment(Key, [{Key2, Val}|Tail], Acc) ->
    key_value_increment(Key, Tail, Acc++[{Key2, Val}]).

%% processing and passing all the messages from parent to childs and vice versa
loop(ChildProcesses, Best_solution, Duplicating, SubMaster_state) ->
    receive
    {Parent, run} ->
        lists:foreach (fun({Pid, _ProcessName, _ProcessIndex}) -> Pid ! {self(), run} end,
        ChildProcesses),
        {New_Solution, Best_Pid, NewChildProcesses} = collect_childs (length(ChildProcesses
        )*Duplicating, Best_solution, SubMaster_state, ChildProcesses, _ReceivedPID=[],
        Duplicating),
        Parent ! {colony_done, {New_Solution, Best_Pid}},
        loop(NewChildProcesses, {New_Solution, Best_Pid}, Duplicating, SubMaster_state);

    {Best_Pid, {update, Global_Best_Solution}} ->

        lists:foreach (fun({Pid, _ProcessName, _ProcessIndex}) ->
            if Pid =/= Best_Pid ->
                %Pid ! {self(), {update, Improved_Solution}};
                Pid ! {Best_Pid, {update, Global_Best_Solution}};
                true -> ok
            end
        end, ChildProcesses),

        loop(ChildProcesses, {Global_Best_Solution,Best_Pid},Duplicating, SubMaster_state);

    {Parent, stop_ants} ->  % called by master at end of main loop
        lists:foreach (fun ({Pid, _ProcessName, _ProcessIndex}) -> Pid ! {self(), stop_ants
        } end, ChildProcesses),
        io:format("waiting for stop on node ~p ~n" , [node()]),
```

```
        lists:foreach (fun (ID) -> receive ok -> ID; {'EXIT', _FailedPID, _Reason} -> ok
        end end, lists:seq(1,length(ChildProcesses))),
        Parent ! ok
    end.

%% creates appropriate processes on child nodes
run(Num_Jobs, Num_Processes, Duplicating, Num_Ants, Iter_Global, Iter_Local, Inputs, Params
, Nodes, CurrentLevel, MaxLevel, NodeIndex,ProcessIndex, ProcessName, Recovery) ->
    process_flag(trap_exit, true),
    global:register_name(ProcessName, self()),
    SubMaster_state=#submaster_state{num_Jobs=Num_Jobs, num_Processes=Num_Processes,
    duplicating=Duplicating, num_Ants=Num_Ants, iter_Global=Iter_Global, iter_Local=
    Iter_Local, inputs=Inputs, params=Params, nodes=Nodes, currentLevel=CurrentLevel,
    maxLevel=MaxLevel, nodeIndex=NodeIndex, processIndex=ProcessIndex},
    if
        CurrentLevel==MaxLevel-1 ->
            ChildNodes=get_childs(Num_Processes,CurrentLevel,MaxLevel,Nodes,NodeIndex),
            QuotaOfEachProcess=round(length(ChildNodes)/Num_Processes),
            Start=(ProcessIndex-1)*QuotaOfEachProcess+1,
            MyChildNodes=lists:sublist(ChildNodes, Start, Num_Processes),
            Colonies = lists:map (fun(H) ->
                                    ProcessGlobalIndex=ant_master:index_of(H, Nodes),
                                    ChildProcessName=list_to_atom("colony_node_"++
                                    integer_to_list(ProcessGlobalIndex)),
                                    case global:whereis_name(ChildProcessName) of
                                    undefined ->
                                        ChildPID=spawn_link(H, ant_colony, init, [Num_Ants,
                                         Duplicating, Iter_Local, Num_Jobs, Inputs, Params,
                                         ChildProcessName, Recovery]);
                                    ChildPID -> link(ChildPID)
                                    end,
                                    {ChildPID, ChildProcessName, ProcessGlobalIndex}
                                end, MyChildNodes),
            ChildPIDs=lists:map (fun({ChildPID, _ProcessName, _ProcessIndex}) -> ChildPID
            end,Colonies),
            #params{chaos=Chaos} = Params,
            if
                Chaos==true andalso Recovery==false ->
                    Chaos_starter=whereis(chaos_starter),
                    io:format("submaster - Sending ~p pids of colonies from node ~p to ~p
                    ~n" , [length(ChildPIDs), node(), Chaos_starter]),
                    Chaos_starter! {pids, ChildPIDs};
                true ->
                    ok
            end,
            loop(Colonies, none, Duplicating, SubMaster_state);
        true ->
            ChildNodes=get_childs(Num_Processes,CurrentLevel,MaxLevel,Nodes,NodeIndex),
            ChildNode=lists:nth(ProcessIndex, ChildNodes),
            ChildProcesses = lists:map(
            fun(NextProcessIndex) ->
                ChildProcessName=ant_submaster:generate_submaster_name(CurrentLevel+1,
                Num_Processes*(NodeIndex-1)+ProcessIndex, NextProcessIndex),
                case global:whereis_name(ChildProcessName) of
                undefined ->
                    ChildPID=spawn_link(ChildNode,ant_submaster,run,[Num_Jobs,
                    Num_Processes, Duplicating, Num_Ants, Iter_Global, Iter_Local, Inputs,
                    Params,Nodes, CurrentLevel+1, MaxLevel,Num_Processes*(NodeIndex-1)+
                    ProcessIndex,NextProcessIndex, ChildProcessName, Recovery]),
                    {ChildPID, ChildProcessName, NextProcessIndex};
                ChildPID -> link(ChildPID),
                    {ChildPID, ChildProcessName, NextProcessIndex}
```

```
                end
            end,
            lists:seq(1,Num_Processes)),
            ChildPIDs=lists:map (fun({ChildPID, _ProcessName, _ProcessIndex}) -> ChildPID
            end,ChildProcesses),
            #params{chaos=Chaos} = Params,
            if
                Chaos==true andalso Recovery==false ->
                    Chaos_starter=whereis(chaos_starter),
                    io:format("submaster - Sending ~p pids from node ~p to ~p ~n" , [length
                    (ChildPIDs), node(), Chaos_starter]),
                    Chaos_starter! {pids, ChildPIDs};
                true ->
                    ok
            end,
            loop (ChildProcesses, none, _Duplicating=1, SubMaster_state)
    end.

%% auxiliary functions to create a tree of submaster nodes

%% calculates the number of levels in the tree
%% each node has "Num_Processes" processes
%% each process supervise one node in lower level, except the last level that each process
supervises "Num_Processes" number of nodes

%% calculates the tree level based on the total number of nodes and node degree (degree of
vertices)
find_max_level(Num_Nodes,Num_Processes) ->
    S=speculate_level(Num_Nodes,Num_Processes),
    find_max_level(Num_Nodes,Num_Processes,S).

find_max_level(Num_Nodes,Num_Processes,Speculated_level) ->
    Result=calc_formula(Num_Processes,Speculated_level),
    if
        Result ==0 -> {_Level=0,_NumNodes=0};
        Result =< Num_Nodes -> {Speculated_level,Result};
        true -> find_max_level(Num_Nodes,Num_Processes,Speculated_level-1)
    end.

%% finds the largest possible power for Num_Processes to be less than Num_Nodes
speculate_level(Num_Nodes,Num_Processes)->
    speculate_level(Num_Nodes,Num_Processes,0).

speculate_level(Num_Nodes,Num_Processes,Acc)->
    Result=math:pow(Num_Processes,Acc),
    if
         Result<Num_Nodes ->
             speculate_level(Num_Nodes,Num_Processes,Acc+1);
        true ->
            round(Acc-1)
    end.

%% calculates 1+P^1+P^2+...+P^(N-2)+P^N
calc_formula(Num_Processes,Level) when Level>=2 ->
    calc_formula(Num_Processes,Level,_Acc=0,_Current_level=0);

%% No submaster can be allocated
calc_formula(_Num_Processes,_Level) ->
    0.

calc_formula(Num_Processes,Last_Level,Acc,Current_level) when Current_level=<Last_Level ->
    Num_Nodes=math:pow(Num_Processes,Current_level),
```

```
    case Current_level+2 of
        Last_Level ->
            calc_formula(Num_Processes,Last_Level,Acc+Num_Nodes,Current_level+2);
        _->
            calc_formula(Num_Processes,Last_Level,Acc+Num_Nodes,Current_level+1)
    end;

calc_formula(_Num_Processes,_Last_Level,Acc,_Current_level) ->
    round(Acc).

%% returns number of nodes for a specific level
nodes_in_level(Num_Processes, Level) ->
    round(math:pow(Num_Processes,Level-1)).

%% returns all the child nodes of a specific node. Node is specified by its level and its index in the level
%% How to test: Nodes=lists:seq(1, 277).
list_to_tuple(ant_submaster:get_childs(4,3,4,Nodes,1)).
get_childs(Num_Processes,CurrentLevel,MaxLevel,Nodes,IndexInLevel) -> %when CurrentLevel<MaxLevel ->
    Num_Node_in_Current_Level=nodes_in_level(Num_Processes,CurrentLevel),
    if
        Num_Node_in_Current_Level<IndexInLevel ->
            throw(index_in_level_is_more_than_num_nodes_in_level);
        true -> ok
    end,
    if
        CurrentLevel>=MaxLevel ->
            Num_Node_in_Next_Level=0,
            throw(current_level_must_be_less_than_max_level);
        CurrentLevel==MaxLevel-1 ->
            Num_Node_in_Next_Level=nodes_in_level(Num_Processes,CurrentLevel+2);
        true->
            Num_Node_in_Next_Level=nodes_in_level(Num_Processes,CurrentLevel+1)
    end,
    Childs_Per_Node=Num_Node_in_Next_Level/Num_Node_in_Current_Level,
    Index_For_Next_Line=Childs_Per_Node*(IndexInLevel-1)+1,
    After_Me_This_Level=Num_Node_in_Current_Level-IndexInLevel,
    Child_index=level_index(Num_Processes,CurrentLevel)+IndexInLevel+After_Me_This_Level+Index_For_Next_Line,lists:sublist(Nodes, round(Child_index), round(Childs_Per_Node)).

%% returns the index of the first node for a specific level
level_index(Num_Processes,Level) ->
    level_index(Num_Processes,Level,0,0).

level_index(Num_Processes,Level,Acc,CurrentLevel) when CurrentLevel<Level-1 ->
    R=math:pow(Num_Processes,CurrentLevel),
    level_index(Num_Processes,Level,Acc+R,CurrentLevel+1);

level_index(_Num_Processes,_Level,Acc,_CurrentLevel)->
Acc.

generate_submaster_name(Level, NodeIndex, ProcessIndex) ->
    Name=integer_to_list(Level)++integer_to_list(NodeIndex)++integer_to_list(ProcessIndex),
    list_to_atom(Name).

%% Recovers a failed process
recover_childs(FailedPID, SubMaster_state, ChildProcesses) ->
    #submaster_state{
    num_Jobs=Num_Jobs,
    num_Processes=Num_Processes,
    duplicating=Duplicating,
```

```
num_Ants=Num_Ants,
iter_Global=Iter_Global,
iter_Local=Iter_Local,
inputs=Inputs,
params=Params,
nodes=Nodes,
currentLevel=CurrentLevel,
maxLevel=MaxLevel,
nodeIndex=NodeIndex,
processIndex=ProcessIndex} = SubMaster_state,

if
    CurrentLevel==MaxLevel-1 ->
        case ant_master:get_process_name(FailedPID, ChildProcesses) of
        not_found ->
            io:format ("No recovery for colony process ~p is needed on node ~p ~n", [
            FailedPID, node()]),
            {no_updated, ChildProcesses};
        {ChildProcessName, ProcessGlobalIndex} ->
            Node=lists:nth(ProcessGlobalIndex, Nodes),
            NewPID=spawn_link(Node, ant_colony, init, [Num_Ants, Duplicating,
            Iter_Local, Num_Jobs, Inputs, Params, ChildProcessName, _Recovery=true]),
            util:send_pid(NewPID),
            NewChildProcesses=ant_master:update_process_PID(ChildProcessName, NewPID,
            ChildProcesses),
            io:format ("recovery of a colony process ~p on node ~p by new length ~p
            and new PID ~p ~n", [FailedPID, node(), length(NewChildProcesses), NewPID]),
            {updated, NewChildProcesses}
        end;
    true ->
        case ant_master:get_process_name(FailedPID, ChildProcesses) of
        not_found ->
            io:format ("No recovery for sub-master process ~p is needed on node ~p ~n",
             [FailedPID, node()]),
            {no_updated, ChildProcesses};
        {ChildProcessName, NextProcessIndex} ->
            ChildNodes=get_childs(Num_Processes,CurrentLevel,MaxLevel,Nodes,NodeIndex),
            ChildNode=lists:nth(ProcessIndex, ChildNodes),
            NewPID=spawn_link(ChildNode,ant_submaster,run,[Num_Jobs, Num_Processes,
            Duplicating, Num_Ants, Iter_Global, Iter_Local, Inputs, Params,Nodes,
            CurrentLevel+1, MaxLevel,Num_Processes*(NodeIndex-1)+ProcessIndex,
            NextProcessIndex, ChildProcessName, _Recovery=true]),
            util:send_pid(NewPID),
            NewChildProcesses=ant_master:update_process_PID(ChildProcessName, NewPID,
            ChildProcesses),
            io:format ("recovery of a sub-master process ~p on node ~p by new length
            ~p and new PID ~p ~n", [FailedPID, node(), length(NewChildProcesses),
            NewPID]),
            {updated, NewChildProcesses}
        end
end.
```

Code C.16: Sub-Master Module (ant_submaster.erl)

```
%% Supervises a number of sub-master processes locally

%% Author: Amir Ghaffari <Amir.Ghaffari@glasgow.ac.uk>

%% RELEASE project (http://www.release-project.eu/)

-module(sup_submaster).
-include("types.hrl").
-compile(export_all).


run(Num_Jobs, Num_Processes, Duplicating, Num_Ants, Iter_Global, Iter_Local, Inputs, Params
,Nodes, CurrentLevel, MaxLevel,NodeIndex, ProcessIndex, Recovery, {ParentName, ParentPid})
->
    process_flag(trap_exit, true),
    if
        Recovery==false ->
            io:format("creating a sup_submaster process on node ~p ~n" , [node()]);
        true ->
            io:format("recovery a sup_submaster process on node ~p ~n" , [node()])
    end,
    ChildProcesses = lists:map(
    fun(NextProcessIndex) ->
        ChildProcessName=ant_submaster:generate_submaster_name(CurrentLevel+1,
        Num_Processes*(NodeIndex-1)+ProcessIndex, NextProcessIndex),
        case whereis(ChildProcessName) of
        undefined ->
            ChildPID=spawn_link(ant_submaster,run,[Num_Jobs, Num_Processes, Duplicating,
            Num_Ants, Iter_Global, Iter_Local, Inputs, Params, Nodes, CurrentLevel,
            MaxLevel, NodeIndex, NextProcessIndex, ChildProcessName, Recovery]),
            {ChildPID, ChildProcessName, NextProcessIndex};
        ChildPID -> link(ChildPID),
            {ChildPID, ChildProcessName, NextProcessIndex}
        end
    end,
    lists:seq(1, Num_Processes)),
    ReliablePid=get_Pid_reliably_sgroup(ParentName, ParentPid),
    ReliablePid!{sup_submaster,ChildProcesses},
    monitor(Num_Jobs, Num_Processes, Duplicating, Num_Ants, Iter_Global, Iter_Local, Inputs
    , Params, Nodes, CurrentLevel, MaxLevel, NodeIndex, Recovery, {ParentName, ParentPid},
    ChildProcesses, _Terminating=false).

monitor(Num_Jobs, Num_Processes, Duplicating, Num_Ants, Iter_Global, Iter_Local, Inputs,
Params, Nodes, CurrentLevel, MaxLevel, NodeIndex, Recovery, {ParentName, ParentPid},
ChildProcesses, Terminating) ->
    receive
        {ParentPid, stop_ants} ->
            monitor(Num_Jobs, Num_Processes, Duplicating, Num_Ants, Iter_Global, Iter_Local
            , Inputs, Params, Nodes, CurrentLevel, MaxLevel, NodeIndex, Recovery, {
            ParentName, ParentPid}, ChildProcesses, _Terminating=true);
        {after_recovery,NewParentPid} ->
            NewParentPid! {sup_submaster,ChildProcesses},
            monitor(Num_Jobs, Num_Processes, Duplicating, Num_Ants, Iter_Global, Iter_Local
            , Inputs, Params, Nodes, CurrentLevel, MaxLevel, NodeIndex, Recovery, {
            ParentName, NewParentPid}, ChildProcesses, Terminating);
        {'EXIT', FailedPID, Reason} ->
            if
                Terminating==false ->
                    io:format("Failure of a sub-master process with PID ~p and reason ~p
                    ~n", [FailedPID, Reason]),
                    case ant_master:get_process_name(FailedPID, ChildProcesses) of
                        not_found ->
```

```
                            io:format("No recovery for sub-master process ~p is needed on
                            node ~p ~n", [FailedPID, node()]),
                            monitor(Num_Jobs, Num_Processes, Duplicating, Num_Ants,
                            Iter_Global, Iter_Local, Inputs, Params, Nodes, CurrentLevel,
                            MaxLevel, NodeIndex, Recovery, {ParentName, ParentPid},
                            ChildProcesses, Terminating);
                        {ChildProcessName, NextProcessIndex} ->
                            NewPID=spawn_link(ant_submaster,run,[Num_Jobs, Num_Processes,
                            Duplicating, Num_Ants, Iter_Global, Iter_Local, Inputs, Params,
                             Nodes, CurrentLevel, MaxLevel, NodeIndex, NextProcessIndex,
                            ChildProcessName, _Recovery=true]),
                            NewChildProcesses=ant_master:update_process_PID(
                            ChildProcessName, NewPID, ChildProcesses),
                            io:format("recovery of a sub-master process ~p on node ~p by
                            new length ~p and new PID ~p ~n", [FailedPID, node(), length(
                            NewChildProcesses), NewPID]),
                            ReliablePid=get_Pid_reliably_sgroup(ParentName, ParentPid),
                            ReliablePid!{afterFailure,NewChildProcesses, FailedPID},
                            monitor(Num_Jobs, Num_Processes, Duplicating, Num_Ants,
                            Iter_Global, Iter_Local, Inputs, Params, Nodes, CurrentLevel,
                            MaxLevel, NodeIndex, Recovery, {ParentName, ParentPid},
                            NewChildProcesses, Terminating)
                    end;
                true ->
                    if
                        Reason=='normal' ->
                            monitor(Num_Jobs, Num_Processes, Duplicating, Num_Ants,
                            Iter_Global, Iter_Local, Inputs, Params, Nodes, CurrentLevel,
                            MaxLevel, NodeIndex, Recovery, {ParentName, ParentPid},
                            ChildProcesses, Terminating);
                        true->
                            ReliablePid=get_Pid_reliably_sgroup(ParentName, ParentPid),
                            ReliablePid! {'EXIT', FailedPID, Reason},
                            monitor(Num_Jobs, Num_Processes, Duplicating, Num_Ants,
                            Iter_Global, Iter_Local, Inputs, Params, Nodes, CurrentLevel,
                            MaxLevel, NodeIndex, Recovery, {ParentName, ParentPid},
                            ChildProcesses, Terminating)
                    end
            end
    end.

get_Pid_reliably_sgroup(ProcessName, ProcessPid) when not is_integer(ProcessPid)->
    if
        is_pid(ProcessPid)==true ->
            ProcessPid;
        true->
            get_Pid_reliably_sgroup(ProcessName, 3)
    end;

get_Pid_reliably_sgroup(ProcessName, 0)->
    io:format("After 3 times attempts, ant_submaster process is not available in global
    group with name ~p ~n", [ProcessName]),
    undefined;

get_Pid_reliably_sgroup(ProcessName, Num)->
    case global:whereis_name(ProcessName) of
    undefined ->
        timer:sleep(1),
        get_Pid_reliably_sgroup(ProcessName, Num-1);
    Pid ->
        if
            is_pid(Pid)==true ->
```

```
                Pid;
            true->
                timer:sleep(1),
                get_Pid_reliably_sgroup(ProcessName, Num-1)
        end
    end.
```

Code C.17: Local Supervisor Module (sup_submaster.erl)

C.6 Scalable Reliable ACO Source Code

This section contains the source code of Scalable Reliable Distributed ACO (SR-ACO). The main affected modules by SD Erlang, i.e. *ant*, *colony*, *master*, *submaster*, and *supervisor* are listed as Codes C.18, C.19, C.20, C.21, and C.22 respectively. All modules and deployment scripts are publicly available at [118].

```
%% Individual ant: constructs new solution based on pheromone matrix.

-module(ant).
-include ("types.hrl").
-export([start/4]).

% M&M -> Merkle & Middendorf: An Ant Algorithm with a new
% Pheromone Evaluation Rule for Total Tardiness Problems

%% ----------------------------------------------------------------- %%
% Strictly, P(i,j) is equal to Pij/Sigma; he're we're calculating both as we go along.

construct_pij([], _, _, _, Sigma, P) -> % Sigma is sum of terms over unscheduled jobs: see
M&M
    {lists:reverse(P), Sigma};
construct_pij([{J,Eta_J}|TEta], Alpha, Beta, TauI, Sigma, P) ->
    Pij = math:pow (element(J, TauI), Alpha) * math:pow(Eta_J, Beta),
    construct_pij(TEta, Alpha, Beta, TauI, Sigma+Pij, [{J,Pij}|P]).

% Eta contains info only for unscheduled jobs, so the above function constructs info only
for those jobs.



%% ----------------------------------------------------------------- %%
% Careful here: elements of Pij can be extremely small (10^-200 towards end).

-spec find_limit_index ([idxf90], index(), float(), float()) -> index().
find_limit_index(P, Idx, Total, Limit) ->
    if Total >= Limit -> Idx;
       true -> case P of
           [] -> error ("Fell off end of P in find_limit_index");
                       % This could happen if Limit is 1 (or very close)
                       % and rounding errors make the sum of pij slightly
                       % less than 1.
           [{K, Kval}|Rest] ->
               find_limit_index (Rest, K, Total+Kval, Limit)
           end
    end.


-spec find_random_job([idxf()], float()) -> index().
find_random_job(P_i, SumPij) ->
    case P_i of
    [] -> error ("find_rand_pij: P_i is empty");
    [{J1, J1val}|Rest] ->
        Limit = random:uniform() * SumPij,
        find_limit_index(Rest, J1, J1val, Limit)
    end.



%% ----------------------------------------------------------------- %%

% Find index j which maximises P[i,j] (i is fixed).
% Note that we're usually working with a partial schedule, in which case many of the
% elements of Pij are zero.  We need to be sure that at least one element is greater
% than zero when calling this function.


-spec find_index_max_P(index(), float(), [idxf()]) -> index().
```

```
find_index_max_P(MaxIndex, _, []) -> MaxIndex;
find_index_max_P(MaxIndex, MaxVal, [{K,Kval}|Rest]) ->
    if Kval =< MaxVal -> find_index_max_P (MaxIndex, MaxVal, Rest);
       Kval >  MaxVal -> find_index_max_P (K,        Kval,   Rest)
    end.


-spec find_index_max_P([idxf()]) -> index().

find_index_max_P ([]) -> error ("find_index_max_P: P_i is empty");
find_index_max_P ([{J1,J1val}|Rest]) -> find_index_max_P (J1, J1val, Rest).


%% ---------------------------------------------------------------- %%

-spec find_solution (I::integer(),
                     % We're trying to schedule a job at position I. We need I to index
                     tau here.
             Unscheduled_Jobs :: intlist(),
             Inputs::inputs(),
             Alpha::number(), Beta::number(), Q0::number(),
             Heuristic :: heuristic(),
             Partial_Schedule::intlist(),
             Scheduled_Time::integer()
            ) -> schedule().


find_solution(I, Unscheduled_Jobs, Inputs, Alpha, Beta, Q0, Heuristic,
         Partial_Schedule, Scheduled_Time) ->
    case Unscheduled_Jobs of
    [] -> error ("Ran out of unscheduled jobs in find_solution");
    [Last] -> lists:reverse(Partial_Schedule, [Last]);
    _ ->
        Eta = fitness:make_eta (Unscheduled_Jobs, Heuristic, Inputs, Scheduled_Time),
%       io:format("Eta=~p~n", [Eta]),
        [{I, TauI}] = ets:lookup(tau, I),  % Since tau is a set, we should always get a
        one-entry list.

        {P_i, SumPij} = construct_pij(Eta, Alpha, Beta, TauI, 0.0, []),

        Q = random:uniform(),
        New_Job = if Q<Q0 -> find_index_max_P (P_i);
                  % Value of J which % maximises %
                  % {tau^alpha}{eta^beta}[J] % for %
                  % unscheduled % jobs
             true -> find_random_job (P_i, SumPij)
                  % Choose a job randomly wrt to probability distribution
                  % given by P_i
              end,
%       io:format ("New_Job = ~p~n", [New_Job]),
        {Durations,_,_} = Inputs,
        Current_Time = Scheduled_Time+element(New_Job, Durations),
        Now_Unscheduled = lists:delete (New_Job, Unscheduled_Jobs),
        find_solution(I+1, Now_Unscheduled, Inputs, Alpha, Beta, Q0, Heuristic,
              [New_Job|Partial_Schedule], Current_Time)
    end.

-spec find_solution (numjobs(), inputs(), params()) -> solution().
find_solution(Num_Jobs, Inputs, Params) ->
    #params{alpha=Alpha, beta=Beta, q0=Q0, heuristic=Heuristic} = Params,
    Unscheduled_Jobs = lists:seq(1, Num_Jobs),
```

```
    Schedule = find_solution(1, Unscheduled_Jobs, Inputs, Alpha, Beta, Q0, Heuristic, [], 0),
    Tardiness = fitness:tardiness(Schedule, Inputs),
    {Tardiness, Schedule}.


%% ---------------------------------------------------------------- %%

%-spec loop(numjobs(), inputs(), params()) -> ok.

%loop(Num_Jobs, Inputs, Params) -> loop(Num_Jobs, Inputs, Params, 0).

%loop(Num_Jobs, Inputs, Params, 0) ->
%    receive
%   {From, find_solution} ->
%       Result = find_solution(Num_Jobs, Inputs, Params),
%       From ! {ant_done, self(), Result},
%       loop(Num_Jobs, Inputs, Params);
%   {_From, stop} -> ok;
%       Msg ->       error ({unexpected_message, Msg})
%   after
%       ?TimeOut ->
%           io:format ("Ant Timeout with pid ~p ~n", [self()]),
%           loop(Num_Jobs, Inputs, Params, 1)
%   end;

%loop(Num_Jobs, Inputs, Params, 1) ->
%    receive
%   {From, find_solution} ->
%       Result = find_solution(Num_Jobs, Inputs, Params),
%       From ! {ant_done, self(), Result},
%       loop(Num_Jobs, Inputs, Params);
%   {_From, stop} -> ok;
 %       Msg ->      error ({unexpected_message, Msg})
%    end.

-spec loop(numjobs(), inputs(), params()) -> ok.
loop(Num_Jobs, Inputs, Params) ->
    receive
    {From, find_solution} ->
        Result = find_solution(Num_Jobs, Inputs, Params),
        From ! {ant_done, self(), Result},
        loop(Num_Jobs, Inputs, Params);
    {_From, stop} -> ok;
       Msg ->        error ({unexpected_message, Msg})
    end.


-spec start(numjobs(), inputs(), params(), pid()) -> ok.
start(Num_Jobs, Inputs, Params, Parent) ->
    link(Parent),
    #params{seed_now=Seed_now} = Params,
    if Seed_now ->
        {A, B, C} = now();
       true ->
        <<A:32,B:32,C:32>> = crypto:rand_bytes(12)
            % This seems to be safer, but the HW machines don't have the
        % correct C crypto library.
    end,
    random:seed(A,B,C),
    loop(Num_Jobs, Inputs, Params).
```

Code C.18: Ant Module (ant.erl)

```
%% A single ant colony: spawns a set of ants, loops round getting them to construct new solutions

-module(ant_colony).
-export([init/8]).

-include ("types.hrl").

% The value 'none' below is used to represent the state at the very start
% when we don't have a current best solution.

-spec best_solution (solution(), solution()|none) -> solution().

best_solution(Solution, none) -> Solution;
best_solution(none, Solution) -> Solution; % added just for test amir
best_solution(S1 = {Cost1, _}, S2 = {Cost2, _}) ->
    if
    Cost1 =< Cost2 -> S1;
    true -> S2
    end.


-spec collect_ants(non_neg_integer(), solution() | none, list(), colony_state(), list()) ->
 solution().

collect_ants (0,Best, Ants, _Colony_state, _ReceivedPIDs) ->
{Ants, Best};
collect_ants (Num_left, Current_Best, Ants, Colony_state, ReceivedPIDs) -> % or use lists:foldl.
    receive
    {ant_done, PID, New_Solution} ->
        collect_ants(Num_left-1, best_solution (New_Solution, Current_Best), Ants, Colony_state, [PID]++ReceivedPIDs);
    {'EXIT', FailedPID, Reason} ->
        ?Print(io_lib:format("Failure of an ant process with PID ~p and reason ~p", [FailedPID, Reason])),
        case recover_childs(FailedPID, Colony_state, Ants) of
        {no_updated, Ants}->
            collect_ants(Num_left, Current_Best, Ants, Colony_state, ReceivedPIDs);
        {updated, NewAnts}->
            case lists:member(FailedPID, ReceivedPIDs) of
                true-> collect_ants(Num_left, Current_Best, NewAnts, Colony_state , ReceivedPIDs);
                _->    collect_ants(Num_left-1, Current_Best, NewAnts, Colony_state, ReceivedPIDs)
            end
        end
    after
        ?TimeOut ->
            ?Print(io_lib:format("Timeout has occurred on colony process ~p on node ~p", [self(), node()])),
            collect_ants (Num_left, Current_Best, Ants, Colony_state, ReceivedPIDs)
    end.

%-spec aco_loop (pos_integer(), [pid()], solution() | none) -> solution().
aco_loop (0, _, _, _, Ants, Best_Solution, Colony_state) ->
    {Ants, Best_Solution, Colony_state};
aco_loop (Iter_Local, Num_Jobs, Inputs, Params, Ants, Best_Solution, Colony_state) ->
    lists:foreach (fun (Pid) -> Pid ! {self(), find_solution} end, Ants),
    {New_Ants, New_Solution} = collect_ants(length(Ants), none, Ants, Colony_state, _ReceivedPIDs=[]),
```

```
    {Cost1, _} = New_Solution,
    Improved_Solution = localsearch:vnd_loop(New_Solution, Inputs, Params),

    {Cost, _} = Improved_Solution,

    #params{vverbose=Vverbose} = Params,
    case Vverbose of
    true -> ?Print(io_lib:format("Colony ~p: cost = ~p -> ~p", [self(), Cost1, Cost]));
    false -> ok
    end,

    New_Best_Solution = best_solution (Improved_Solution, Best_Solution),

    ok = update:update_pheromones(Num_Jobs, New_Best_Solution, Params),
    aco_loop (Iter_Local-1, Num_Jobs, Inputs, Params, New_Ants, New_Best_Solution,
    Colony_state).

main (Duplicating, Iter_Local, Num_Jobs, Inputs, Params, Ants, Current_Best, Colony_state) ->
    receive
    {Master, run} ->
        {New_Ants, New_Solution, New_Colony_state} = aco_loop (Iter_Local, Num_Jobs,
                    Inputs, Params,
                    Ants, Current_Best, Colony_state),

        lists:foreach (fun (_ID) -> Master ! {colony_done, {New_Solution, self()}} end,
        lists:seq(1, Duplicating)),
        main (Duplicating, Iter_Local, Num_Jobs, Inputs, Params, New_Ants, New_Solution,
        New_Colony_state);

    {_Master, {update, Global_Best_Solution}} ->
        update:update_pheromones (Num_Jobs, Global_Best_Solution, Params),
        main (Duplicating, Iter_Local, Num_Jobs, Inputs, Params, Ants, Current_Best,
        Colony_state);

    %% NOTE!!!! Having updated tau according to the global best solution, the
    %% colony carries on with its OWN current best solution.  We could also
    %% carry on with the global best solution:  might this lead to stagnation?

    {Master, stop_ants} ->  % called by master at end of main loop
        ok = lists:foreach (fun (Pid) -> Pid ! {self(), stop} end, Ants),
        ets:delete(tau),
        Master ! ok;
    {run_the_chaos} ->
        Chaos_starter=util:get_local_name(chaos_starter),
        Chaos_starter!{run_the_chaos},
        main (Duplicating, Iter_Local, Num_Jobs, Inputs, Params, Ants, Current_Best,
        Colony_state)
    end.



%% For each VM, set up the ETS table and spawn the required number of ants.
%% Then get each VM to perform (say) 50 iterations and report best solution.
%% Send overall best solution to all VMs (or maybe get the owner of the best
%% solution to send it to everyone), and start a new round of iterations.
%% Stop after a certain number of global iterations.

-spec spawn_ants(integer(), list()) -> [pid()].
spawn_ants(Num_Ants,Args) -> lists:map (fun (_) -> spawn_link(ant, start, Args) end, lists:
seq(1,Num_Ants)).

-spec init(integer(), integer(), integer(), numjobs(), inputs(), params(), list(), boolean)
```

```
 -> pid().
init (Num_Ants, Duplicating, Iter_Local, Num_Jobs, Inputs, Params, _ProcessName, Recovery) ->
    process_flag(trap_exit, true),
    #params{chaos=Chaos, printer=Printer} = Params,
    if
        Printer==false->
            put(parent_printer, no_print);
        true->
            put(parent_printer, Params#params.parent_printer)
    end,
    case s_group:s_groups() of
    undefined ->
        ?Print(io_lib:format("sd_erlang: ant colony on node ~p is not in any sgroups " , [
        node()]));
    {OwnSGroups, Reason} ->
        if
            is_list(OwnSGroups) ->
                case length(OwnSGroups) of
                    1->
                        ok;
                    _->
                    ?Print(io_lib:format("sd_erlang: ant colony on node ~p belongs to ~p
                    sgroups (more or less than one group!) ~p " , [node(), length(
                    OwnSGroups), OwnSGroups]))
                end;
            true ->
                ?Print(io_lib:format("sd_erlang: ant colony s_group:s_groups() returns ~p
                with reason ~p " , [OwnSGroups, Reason]))
        end
    end,
    % Set up ets table: visible to all ants
    case lists:member(tau, ets:all()) of
        true -> %ok;  % Already exists: maybe left over from interrupted run.
            ok;
        false -> ets:new(tau, [set, public, named_table])
      end,
    #params{tau0=Tau0} = Params,
    Tau = lists:map(fun(I) -> {I, util:make_tuple(Num_Jobs, Tau0)} end, lists:seq(1,
    Num_Jobs)),
    ets:insert(tau, Tau),

%    io:format("~p~n", [ets:tab2list(tau)]),

    Colony_state=#colony_state{num_Jobs=Num_Jobs, inputs=Inputs, params=Params},
    Ants = spawn_ants(Num_Ants, [Num_Jobs, Inputs, Params, self()]),

    #params{chaos=Chaos, printer=Printer} = Params,
    if
        Chaos==false andalso Printer==false ->
            ok;
        Recovery==false ->
            Chaos_starter=spawn(start_chaos, run, [get(parent_printer), Params#params.chaos
            , Params#params.printer]),
            ?Print(io_lib:format("ant colony - sending ~p pids from node ~p to ~p " , [
            length(Ants), node(), Chaos_starter])),
            if
                Chaos==true->
                    Chaos_starter=util:get_local_name(chaos_starter), %% chaos monkey is
                    running on this node
                    Chaos_starter! {pids, Ants};
                true ->
                    ok
```

```
            end;
        true->
            ok
    end,

    main(Duplicating, Iter_Local, Num_Jobs, Inputs, Params, Ants, none, Colony_state),
    self().

    % We could also put Num_Jobs, Durations, Weights, Deadlines,
    % Alpha, Beta, Q0 (all read_only) in an ETS table and reduce
    % the number of parameters in a lot of function calls.

%% Recovers a failed process
recover_childs(FailedPID, Colony_state, Ants) ->
    #colony_state{num_Jobs=Num_Jobs, inputs=Inputs, params=Params}=Colony_state,
    case lists:member(FailedPID, Ants) of
        true ->
            Index=ant_master:index_of(FailedPID, Ants),
            PID=spawn_link(ant, start, [Num_Jobs, Inputs, Params, self()]),
            util:send_pid(PID),
            NewAnts=lists:sublist(Ants,1,Index-1)++[PID]++lists:sublist(Ants,Index+1,length
            (Ants)),
            ?Print(io_lib:format("recovery of ant process ~p on node ~p by new length ~p
            and new PID ~p ", [FailedPID, node(), length(NewAnts), PID])),
            {updated, NewAnts};
        _->
            ?Print(io_lib:format("No recovery for ant process ~p is needed on node ~p ", [
            FailedPID, node()])),
            {no_updated, Ants}
    end.
```

Code C.19: Ant Colony Module (ant_colony.erl)

```
-module(ant_master).
-compile(export_all).
-include("types.hrl").

%-spec best_solution ({solution(), pid()}, {solution(),pid()}|none) -> {solution(), pid()}.

best_solution(Solution, none) -> Solution;
best_solution(none, Solution) -> Solution;
best_solution(S1 = {{Cost1, _}, _}, S2 = {{Cost2, _}, _}) ->
    if
    Cost1 =< Cost2 -> S1;
    true -> S2
    end.
% The stuff with the pids is just so that we can see who's produced the best solution


%-spec collect_ants(non_neg_integer(), solution() | none) -> solution().

collect_colonies (0,{Best, Pid}, _Master_state, NewColonies, _ReceivedPIDs, _Duplicating) ->
    {Best, Pid, NewColonies};
collect_colonies (Num_left, Current_Best, Master_state, Colonies, ReceivedPIDs, Duplicating
) -> % or use lists:foldl.
    receive
    {colony_done, {New_Solution, Pid}} ->
%       {Cost, _} = New_Solution,
%       io:format("Solution ~p from ~p~n", [Cost, Pid]),
        collect_colonies(Num_left-1, best_solution ({New_Solution, Pid}, Current_Best),
        Master_state, Colonies, ant_submaster:key_value_increment(Pid, ReceivedPIDs),
        Duplicating);
    {'EXIT', FailedPID, Reason} ->
        #master_state{nodes=Nodes, num_Processes=Num_Processes} = Master_state,
        case ant_submaster:find_max_level(length(Nodes),Num_Processes) of
        {_MaxLevels=0,_NodeLength=0} ->
            ?Print(io_lib:format("Failure of a colony process with PID ~p detected by
            master process and reason ~p ", [FailedPID, Reason]));
        {_MaxLevels,_NodeLength} ->
            ?Print(io_lib:format("Failure of a sub-master process on the first level with
            PID ~p and reason ~p ", [FailedPID, Reason]))
        end,
        case recover_childs(FailedPID, Master_state, Colonies) of
        {no_updated, Colonies}->
            collect_colonies(Num_left, Current_Best, Master_state, Colonies, ReceivedPIDs,
            Duplicating);
        {updated, NewColonies}->
            case ant_submaster:key_search(FailedPID, ReceivedPIDs) of
                []-> collect_colonies(Num_left-(1*Duplicating), Current_Best, Master_state,
                 NewColonies, ReceivedPIDs, Duplicating);
                Value-> collect_colonies(Num_left-(Duplicating-Value), Current_Best,
                Master_state, NewColonies, ReceivedPIDs, Duplicating)
            end
        end
    after
        ?TimeOut ->
            ?Print(io_lib:format("Timeout has occured in master process ~p on node ~p with
            Num_left ~p and received ~p ", [self(), node(), Num_left, ReceivedPIDs])),
            collect_colonies (Num_left, Current_Best, Master_state, Colonies, ReceivedPIDs,
             Duplicating)
    end.


loop (_Duplicating, 0, Colonies, {Best_solution, _}, _Inputs, _Params, _Master_state) ->
    lists:foreach (fun ({Pid, _ProcessName, _ProcessIndex}) -> Pid ! {self(), stop_ants}
```

```
    end, Colonies),
    lists:foreach (fun (ID) -> receive ok -> ID; {'EXIT', _FailedPID, _Reason} -> ok end
    end, lists:seq(1,length(Colonies))), %% wait to collect all the acknowledges
    Best_solution;

loop (Duplicating, N, Colonies, Best_solution, Inputs, Params, Master_state) ->
    lists:foreach (fun({Pid, _ProcessName, _ProcessIndex}) -> Pid ! {self(), run} end,
    Colonies),
    {{Cost, _} = New_Solution, Best_Pid, NewColonies} = collect_colonies (length(Colonies)*
    Duplicating, Best_solution, Master_state, Colonies, _ReceivedPIDs=[], Duplicating),

% Note that we're using the best solution from the previous generation.
% We could also try starting anew, relying on the pheromone matrix to guide
% ants back to earlier best solutions if they can't do any better.

%    Improved_Solution = localsearch:vnd_loop(New_Solution, Inputs, Params),
% local search now done in colony, after every generation of ants.

    Improved_Solution = New_Solution,
    {Cost, _} = Improved_Solution,

    #params{verbose=Verbose, schedule=Print_schedule} = Params,
    case Verbose of
    true -> ?Print(io_lib:format("Iteration ~p: cost = ~p from ~p", [N, Cost, Best_Pid])),
        case Print_schedule of
            true -> ?Print(io_lib:format("Schedule -> ~p", [Improved_Solution]));
            false -> ok
        end;
    false -> ok
    end,

    lists:foreach (fun({Pid, _ProcessName, _ProcessIndex}) ->
              if Pid =/= Best_Pid ->
                  %Pid ! {self(), {update, Improved_Solution}};
                  Pid ! {Best_Pid, {update, Improved_Solution}};
                 true -> ok
              end
           end, Colonies),

    #params{chaos=Chaos} = Params,
    #master_state{recovery=Recovery} = Master_state,
    if
        Chaos==true andalso Recovery==false->
            ?Print(io_lib:format("more than once on ~p", [node()])),
            lists:foreach (fun({Pid, _ProcessName, _ProcessIndex}) ->  Pid ! {run_the_chaos
            } end, Colonies),
            loop (Duplicating, N-1, NewColonies, {Improved_Solution, Best_Pid}, Inputs,
            Params#params{chaos=false}, Master_state);
        true ->
            loop (Duplicating, N-1, NewColonies, {Improved_Solution, Best_Pid}, Inputs,
            Params, Master_state)
    end.

run(Num_Jobs, Num_Processes, Duplicating, Num_Ants, Iter_Global, Iter_Local, Inputs, Params
, Nodes, Recovery, Starter) ->
    process_flag(trap_exit, true),
    #params{chaos=Chaos, printer=Printer} = Params,
    if
        Printer==false->
            put(parent_printer, no_print);
        true->
            put(parent_printer, Params#params.parent_printer)
```

```
    end,

    if
        Recovery -> ?Print(io_lib:format("Master process restarts at ~p on node ~p ", [time
        (), node()]));
        true ->
            ok
    end,
    Master_state=#master_state{num_Jobs=Num_Jobs, num_Processes=Num_Processes, duplicating=
    Duplicating, num_Ants=Num_Ants, iter_Global=Iter_Global, iter_Local=Iter_Local, inputs=
    Inputs, params=Params, nodes=Nodes, recovery=Recovery},
    case ant_submaster:find_max_level(length(Nodes),Num_Processes) of
    {_MaxLevels=0,_NodeLength=0} ->
        ?Print(io_lib:format("Number of nodes (~p) with ~p processes per node is not
        enough to have any submaster node", [length(Nodes),Num_Processes])),
        GroupName="total_group",
        case s_group:new_s_group(GroupName, Nodes) of
        {ok, GroupName, _Nodes} ->
            ok;
        Error ->
            if
                Recovery -> ok; %% previously the group has been created
                true ->
                    ?Print(io_lib:format("sd_erlang: cannot create total_group for all
                    nodes with error ~p" , [Error]))
            end
        end,
        Colonies = lists:map (fun(H) ->
                                ProcessIndex=index_of(H, Nodes),
                                ProcessName=list_to_atom("colony_node_"++integer_to_list(
                                ProcessIndex)),
                                case s_group:whereis_name(GroupName, ProcessName) of
                                undefined ->
                                    ChildPID=spawn_link(H, ant_colony, init, [Num_Ants,
                                    Duplicating, Iter_Local, Num_Jobs, Inputs, Params,
                                    ProcessName, Recovery]),
                                    s_group:register_name(GroupName, ProcessName, ChildPID);
                                ChildPID -> link(ChildPID)
                                end,
                                {ChildPID, ProcessName, ProcessIndex}
                            end, Nodes),
        ChildPIDs=lists:map (fun({ChildPID, _ProcessName, _ProcessIndex}) -> ChildPID end,
        Colonies),
        #params{chaos=Chaos} = Params,
        if
            Chaos==true andalso Recovery==false ->
                Chaos_starter=util:get_local_name(chaos_starter),
                ?Print(io_lib:format("master - sending ~p pids from node ~p to ~p " , [
                length(ChildPIDs), node(), Chaos_starter])),
                Chaos_starter! {pids, ChildPIDs};
                %Chaos_starter! {pids, []};
            true ->
                ok
        end,
        Results=loop(Duplicating, Iter_Global, Colonies, none, Inputs, Params, Master_state);
    {MaxLevels,NodeLength} ->
        ?Print(io_lib:format("We have ~p levels for ~p nodes and ~p processes per node (~p
        nodes will be used)", [MaxLevels,length(Nodes),Num_Processes,NodeLength])),
        {UsingNodes,_Rest}=lists:split(NodeLength, Nodes),
        [Head|_Tail]=UsingNodes,
        Groups=s_group:s_groups(),
        ChildProcesses=lists:map(
```

```
            fun(ProcessIndex) ->
                ProcessName=ant_submaster:generate_submaster_name(_CurrentLevel=1, _NodeIndex=1
                , ProcessIndex),
                case Groups of
                    undefined ->
                        ChildPID=spawn_link(Head, ant_submaster, run, [Num_Jobs, Num_Processes,
                         Duplicating, Num_Ants, Iter_Global, Iter_Local, Inputs, Params,
                        UsingNodes, _CurrentLevel=1, MaxLevels, _NodeIndex=1, ProcessIndex,
                        ProcessName, Recovery]),
                        {ChildPID, ProcessName, ProcessIndex};
                    {OwnSGroups,_} ->
                        if
                            is_list(OwnSGroups) ->
                                case length(OwnSGroups) of
                                    1 ->
                                        ok;
                                    _ ->
                                        ?Print(io_lib:format("sd_erlang: (warning) master
                                        process belongs to more or less than one node ~p ~p", [
                                        length(OwnSGroups),OwnSGroups]))
                                end,
                                [GroupName|_Tail]=OwnSGroups,
                                case s_group:whereis_name(GroupName, ProcessName) of
                                    undefined ->
                                        ChildPID=spawn_link(Head, ant_submaster, run, [Num_Jobs
                                        , Num_Processes, Duplicating, Num_Ants, Iter_Global,
                                        Iter_Local, Inputs, Params,UsingNodes, _CurrentLevel=1,
                                         MaxLevels, _NodeIndex=1, ProcessIndex, ProcessName,
                                        Recovery]),
                                        {ChildPID, ProcessName, ProcessIndex};
                                    ChildPID -> link(ChildPID),
                                        {ChildPID, ProcessName, ProcessIndex}
                                end;
                            true ->
                                ChildPID=spawn_link(Head, ant_submaster, run, [Num_Jobs,
                                Num_Processes, Duplicating, Num_Ants, Iter_Global, Iter_Local,
                                Inputs, Params,UsingNodes, _CurrentLevel=1, MaxLevels,
                                _NodeIndex=1, ProcessIndex, ProcessName, Recovery]),
                                {ChildPID, ProcessName, ProcessIndex}
                        end
                end
            end,
            lists:seq(1,Num_Processes)),
            ChildPIDs=lists:map (fun({ChildPID, _ProcessName, _ProcessIndex}) -> ChildPID end,
            ChildProcesses),
            #params{chaos=Chaos} = Params,
            if
                Chaos==true ->
                    Chaos_starter=util:get_local_name(chaos_starter),
                    ?Print(io_lib:format("master - sending ~p pids from node ~p to ~p" , [
                    length(ChildPIDs), node(), Chaos_starter])),
                    Chaos_starter! {pids, ChildPIDs};
                    %Chaos_starter! {pids, []};
                true ->
                    ok
            end,
            Results=loop(1, Iter_Global, ChildProcesses, none, Inputs, Params, Master_state)
        end,
        Starter ! {self(), Results}.

    %% Recovers a failed process
    recover_childs(FailedPID, Master_state, Colonies) ->
```

```
    #master_state{
    num_Jobs=Num_Jobs,
    num_Processes=Num_Processes,
    duplicating=Duplicating,
    num_Ants=Num_Ants,
    iter_Global=Iter_Global,
    iter_Local=Iter_Local,
    inputs=Inputs,
    params=Params,
    nodes=Nodes} = Master_state,
    case get_process_name(FailedPID, Colonies) of
    not_found ->
        ?Print(io_lib:format("No recovery for colony process ~p is needed on node ~p", [
        FailedPID, node()])),
        {no_updated, Colonies};
    {ProcessName, ProcessIndex} ->
        case ant_submaster:find_max_level(length(Nodes),Num_Processes) of
        {_MaxLevels=0,_NodeLength=0} ->
            Node=lists:nth(ProcessIndex, Nodes),
            NewPID=spawn_link (Node, ant_colony, init,[Num_Ants, Duplicating, Iter_Local,
            Num_Jobs, Inputs, Params,ProcessName, _Recovery=true]),
            util:send_pid(NewPID),
            NewChildProcesses=update_process_PID(ProcessName, NewPID, Colonies),
            ?Print(io_lib:format("recovery of a colony process ~p detected by master
            process on node ~p by new length ~p and new PID ~p", [FailedPID, node(), length
            (NewChildProcesses), NewPID]));
        {MaxLevels,NodeLength} ->
            {UsingNodes,_Rest}=lists:split(NodeLength, Nodes),
            [Head|_Tail]=UsingNodes,
            NewPID=spawn_link(Head, ant_submaster, run, [Num_Jobs, Num_Processes,
            Duplicating, Num_Ants, Iter_Global, Iter_Local, Inputs, Params,UsingNodes,
            _CurrentLevel=1, MaxLevels, _NodeIndex=1, ProcessIndex, ProcessName, _Recovery=
            true]),
            util:send_pid(NewPID),
            NewChildProcesses=update_process_PID(ProcessName, NewPID, Colonies),
            ?Print(io_lib:format("recovery of a sub-master process ~p detected by master
            process on node ~p by new length ~p and new PID ~p", [FailedPID, node(), length
            (NewChildProcesses), NewPID]))
        end,
        {updated, NewChildProcesses}
    end.

%% gets a process identity and return its registred name
get_process_name(_ProcessPID,_Colonies=[]) ->
    not_found;
get_process_name(ProcessPID, [{ProcessPID, ProcessName, ProcessIndex}|_Tail]) ->
    {ProcessName,ProcessIndex};
get_process_name(ProcessPID, [{_Process, _ProcessName, _ProcessIndex}|Tail]) ->
    get_process_name(ProcessPID,Tail).

%% updates the pid of a failed process with the new pid
update_process_PID(ProcessName, NewPID, Colonies) ->
    update_process_PID(ProcessName, NewPID, Colonies, _ACC=[]).

update_process_PID(_ProcessName, _NewPID, [], _ACC)->
        throw(cannot_find_process_name_to_update);

update_process_PID(ProcessName, NewPID, [{_Process, ProcessName, ProcessIndex}|Tail], ACC)->
        ACC++[{NewPID, ProcessName, ProcessIndex}]++Tail;

update_process_PID(ProcessName, NewPID, [{ProcessID, ProcessName2, ProcessIndex}|Tail], ACC
)->
```

```
        update_process_PID(ProcessName, NewPID, Tail, ACC++[{ProcessID, ProcessName2,
        ProcessIndex}]).

%% returns the index of an element of a list
index_of(Item, List) -> index_of(Item, List, 1).
index_of(_, [], _)  -> not_found;
index_of(Item, [Item|_], Index) -> Index;
index_of(Item, [_|Tl], Index) -> index_of(Item, Tl, Index+1).
```

Code C.20: Ant Master Module (ant_master.erl)

```
%% Creates a multi-level tree of sub-master nodes to collect the results from Colonies.

%% Author: Amir Ghaffari <Amir.Ghaffari@glasgow.ac.uk>
%% RELEASE project (http://www.release-project.eu/)

-module(ant_submaster).
-compile(export_all).
-include("types.hrl").


%% compares two solutions and returns the better one
%-spec best_solution ({solution(), pid()}, {solution(),pid()}|none) -> {solution(), pid()}.
best_solution(Solution, none) -> Solution;
best_solution(none, Solution) -> Solution;
best_solution(S1 = {{Cost1, _}, _}, S2 = {{Cost2, _}, _}) ->
    if
    Cost1 =< Cost2 -> S1;
    true -> S2
    end.
% The stuff with the pids is just so that we can see who's produced the best solution


%-spec collect_ants(non_neg_integer(), solution() | none) -> solution().

%% collecting the collonies results from child nodes
collect_childs (0,{Best, Pid}, _SubMaster_state, NewChildProcesses, _ReceivedPIDs,
_Duplicating) ->
    {Best, Pid, NewChildProcesses};
collect_childs (Num_left, Current_Best, SubMaster_state, ChildProcesses, ReceivedPIDs,
Duplicating) -> % or use lists:foldl.
    receive
    {afterFailure,NewChildProcesses, FailedPID} ->
        case key_search(FailedPID, ReceivedPIDs) of
            []->    collect_childs(Num_left-(1*Duplicating), Current_Best, SubMaster_state,
             NewChildProcesses, ReceivedPIDs, Duplicating);
            Value-> collect_childs(Num_left-(Duplicating-Value), Current_Best,
            SubMaster_state, NewChildProcesses, ReceivedPIDs, Duplicating)
        end;
    {colony_done, {New_Solution, Pid}} ->
        collect_childs(Num_left-1, best_solution ({New_Solution, Pid}, Current_Best),
        SubMaster_state, ChildProcesses, key_value_increment(Pid, ReceivedPIDs),
        Duplicating);
    {'EXIT', FailedPID, Reason} ->
        #submaster_state{currentLevel=CurrentLevel, maxLevel=MaxLevel} = SubMaster_state,
        if
            CurrentLevel==MaxLevel-1 ->
                ?Print(io_lib:format("Failure of a colony process with PID ~p and reason
                ~p ", [FailedPID, Reason]));
            true ->
                %?Print(io_lib:format("Failure of a sub-master process with PID ~p and
                reason ~p ", [FailedPID, Reason]))
                ?Print(io_lib:format("Failure of a supervisor process with PID ~p and
                reason ~p ", [FailedPID, Reason]))
        end,
        case recover_childs(FailedPID, SubMaster_state, ChildProcesses) of
            {no_updated, ChildProcesses}->
                collect_childs(Num_left, Current_Best, SubMaster_state, ChildProcesses,
                ReceivedPIDs, Duplicating);
            {updated, NewChildProcesses}->
                case key_search(FailedPID, ReceivedPIDs) of
                    []->    collect_childs(Num_left-(1*Duplicating), Current_Best,
                    SubMaster_state, NewChildProcesses, ReceivedPIDs, Duplicating);
```

```
                    Value-> collect_childs(Num_left-(Duplicating-Value), Current_Best,
                    SubMaster_state, NewChildProcesses, ReceivedPIDs, Duplicating)
                end;
            {updated, NewChildProcesses, NewSupervisorPid}->
                Num_failed=find_num_failed(ChildProcesses, NewChildProcesses, ReceivedPIDs,
                 Duplicating),
                collect_childs(Num_left-Num_failed, Current_Best, SubMaster_state
                #submaster_state{supervisor=NewSupervisorPid}, NewChildProcesses,
                ReceivedPIDs, Duplicating)
        end
    after
        ?TimeOut ->
        #submaster_state{currentLevel=CurrentLevel, maxLevel=MaxLevel} = SubMaster_state,
        if
            CurrentLevel==MaxLevel-1 ->
                ?Print(io_lib:format("Timeout has occurred on a last level sub-master
                process ~p on node ~p ", [self(), node()])),
                collect_childs (Num_left, Current_Best, SubMaster_state, ChildProcesses,
                ReceivedPIDs, Duplicating);
            true ->
                ?Print(io_lib:format("Timeout has occurred on a sub-master process ~p on
                node ~p ", [self(), node()])),
                collect_childs (Num_left, Current_Best, SubMaster_state, ChildProcesses,
                ReceivedPIDs, Duplicating)
        end
    end.

find_num_failed(ChildProcesses, NewChildProcesses, ReceivedPIDs, Duplicating)->
    ChildPIDs=lists:map (fun({ChildPID, _ProcessName, _ProcessIndex}) -> ChildPID end,
    ChildProcesses),
    NewChildPIDs=lists:map (fun({ChildPID, _ProcessName, _ProcessIndex}) -> ChildPID end,
    NewChildProcesses),
    find_num_failed(ChildPIDs, NewChildPIDs, ReceivedPIDs, Duplicating, 0).

find_num_failed([], _NewChildPIDs, _ReceivedPIDs, _Duplicating, Acc) ->
    Acc;

find_num_failed([Head|Tail], NewChildPIDs, ReceivedPIDs, Duplicating, Acc) ->
    case lists:member(Head, NewChildPIDs) of
        false ->
            case key_search(Head, ReceivedPIDs) of
                []->    find_num_failed(Tail, NewChildPIDs, ReceivedPIDs, Acc+Duplicating);
                Val ->  find_num_failed(Tail, NewChildPIDs, ReceivedPIDs, Acc+(Duplicating
                -Val))
            end;
        true->
            find_num_failed(Tail, NewChildPIDs, ReceivedPIDs, Duplicating, Acc)
    end.

%% return the value of a key in a list of {key,value} tuples
key_search(_Key, []) ->
    [];
key_search(Key, [{Key, Val}|_Tail]) ->
    Val;
key_search(Key, [{_Key2, _Val}|Tail]) ->
    key_search(Key, Tail).

key_value_increment(Key, List) ->
    key_value_increment(Key, List, _Acc=[]).

key_value_increment(Key, [], Acc) ->
    Acc++[{Key,1}];
```

```
key_value_increment(Key, [{Key, Val}|Tail], Acc) ->
    Acc++[{Key, Val+1}]++Tail;

key_value_increment(Key, [{Key2, Val}|Tail], Acc) ->
    key_value_increment(Key, Tail, Acc++[{Key2, Val}]).

%% processing and passing all the messages from parent to childs and vice versa
loop(ChildProcesses, Best_solution, Duplicating, SubMaster_state) ->
    receive
    {Parent, run} ->
        lists:foreach (fun({Pid, _ProcessName, _ProcessIndex}) -> Pid ! {self(), run} end,
        ChildProcesses),
        {New_Solution, Best_Pid, NewChildProcesses} = collect_childs (length(ChildProcesses
        )*Duplicating, Best_solution, SubMaster_state, ChildProcesses, _ReceivedPID=[],
        Duplicating),
        Parent ! {colony_done, {New_Solution, Best_Pid}},
        loop(NewChildProcesses, {New_Solution, Best_Pid}, Duplicating, SubMaster_state);

    {Best_Pid, {update, Global_Best_Solution}} ->
        lists:foreach (fun({Pid, _ProcessName, _ProcessIndex}) ->
            if Pid =/= Best_Pid ->
                Pid ! {Best_Pid, {update, Global_Best_Solution}};
                true -> ok
            end
        end, ChildProcesses),
        loop(ChildProcesses, {Global_Best_Solution,Best_Pid},Duplicating, SubMaster_state);

    {Parent, stop_ants} ->  % called by master at end of main loop
        case is_pid(SubMaster_state#submaster_state.supervisor) of
            false ->
                ok; %% this sub-master does not have a supervisor process
            _->
                SubMaster_state#submaster_state.supervisor!{self(), stop_ants}
        end,
        lists:foreach (fun ({Pid, _ProcessName, _ProcessIndex}) -> Pid ! {self(), stop_ants
        } end, ChildProcesses),
        ?Print(io_lib:format("waiting for stop on node ~p " , [node()])),
        lists:foreach (fun (ID) -> receive ok -> ID; {'EXIT', _FailedPID, _Reason} -> ok
        end end, lists:seq(1,length(ChildProcesses))),
        Parent ! ok;

    {run_the_chaos} ->
        if
            SubMaster_state#submaster_state.processIndex==1 ->
                Chaos_starter=util:get_local_name(chaos_starter),
                Chaos_starter!{run_the_chaos};
            true->
                ok
        end,
        lists:foreach (fun({Pid, _ProcessName, _ProcessIndex}) ->  Pid ! {run_the_chaos}
        end, ChildProcesses),
        loop(ChildProcesses, Best_solution, Duplicating, SubMaster_state)
    end.

%% creates appropriate processes on child nodes
run(Num_Jobs, Num_Processes, Duplicating, Num_Ants, Iter_Global, Iter_Local, Inputs, Params
, Nodes, CurrentLevel, MaxLevel, NodeIndex,ProcessIndex, ProcessName, Recovery) ->
    process_flag(trap_exit, true),
    #params{chaos=Chaos, printer=Printer} = Params,
    if
        Printer==false->
```

```
        put(parent_printer, no_print);
    true->
        put(parent_printer, Params#params.parent_printer)
end,
SubMaster_state=#submaster_state{num_Jobs=Num_Jobs, num_Processes=Num_Processes,
duplicating=Duplicating, num_Ants=Num_Ants, iter_Global=Iter_Global, iter_Local=
Iter_Local, inputs=Inputs, params=Params, nodes=Nodes, currentLevel=CurrentLevel,
maxLevel=MaxLevel, nodeIndex=NodeIndex, processIndex=ProcessIndex, processName=
ProcessName},
if
    Chaos==false andalso Printer==false ->
        Chaos_starter=no_print;
    true ->
        if
            ProcessIndex==1 andalso not Recovery->
                Chaos_starter=spawn(start_chaos, run, [get(parent_printer), Params
                #params.chaos, Params#params.printer]);
            true ->
                Chaos_starter=util:get_local_name(chaos_starter) %% chaos monkey is
                run on this node
        end
end,

if
    CurrentLevel==MaxLevel-1 ->
        ChildNodes=get_childs(Num_Processes,CurrentLevel,MaxLevel,Nodes,NodeIndex),
        QuotaOfEachProcess=round(length(ChildNodes)/Num_Processes),
        Start=(ProcessIndex-1)*QuotaOfEachProcess+1,
        MyChildNodes=lists:sublist(ChildNodes, Start, Num_Processes),
        GroupName="group_"++ProcessName,
        add_s_group_last_level(GroupName, [node()]++MyChildNodes),
        s_group:register_name(GroupName, ProcessName, self()),
        Colonies = lists:map (fun(H) ->
                                ProcessGlobalIndex=ant_master:index_of(H, Nodes),
                                ChildProcessName=list_to_atom("colony_node_"++
                                integer_to_list(ProcessGlobalIndex)),
                                case s_group:whereis_name(GroupName, ChildProcessName) of
                                undefined ->
                                    ChildPID=spawn_link(H, ant_colony, init, [Num_Ants,
                                     Duplicating, Iter_Local, Num_Jobs, Inputs, Params
                                    #params{parent_printer=Chaos_starter},
                                    ChildProcessName, Recovery]),
                                    s_group:register_name(GroupName, ChildProcessName,
                                    ChildPID);
                                ChildPID -> link(ChildPID)
                                end,
                                {ChildPID, ChildProcessName, ProcessGlobalIndex}
                              end, MyChildNodes),
        ChildPIDs=lists:map (fun({ChildPID, _ProcessName, _ProcessIndex}) -> ChildPID
        end,Colonies),
        if
            Chaos==true andalso Recovery==false->
                ?Print(io_lib:format("submaster - sending ~p pids of colonies from
                node ~p to ~p " , [length(ChildPIDs), node(), Chaos_starter])),
                Chaos_starter! {pids, ChildPIDs};
                %Chaos_starter! {pids, []};
            true ->
                ok
        end,
        loop(Colonies, none, Duplicating, SubMaster_state#submaster_state{chaos_starter
        =Chaos_starter});
    true ->
```

```
                ChildNodes=get_childs(Num_Processes,CurrentLevel,MaxLevel,Nodes,NodeIndex),
                ChildNode=lists:nth(ProcessIndex, ChildNodes),
                GroupName="group_"++atom_to_list(node()),
                add_s_group(GroupName, ProcessIndex, [node()]++[ChildNode]),
                s_group:register_name(GroupName, ProcessName, self()),
                SupervisorName=list_to_atom("sup_"++atom_to_list(ProcessName)),
                case s_group:whereis_name(GroupName, SupervisorName) of
                undefined ->
                    SupervisorPid=spawn_link(ChildNode,sup_submaster,run,[Num_Jobs,
                    Num_Processes, Duplicating, Num_Ants, Iter_Global, Iter_Local, Inputs,
                    Params#params{parent_printer=Chaos_starter}, Nodes, CurrentLevel+1,
                    MaxLevel, Num_Processes*(NodeIndex-1)+ProcessIndex, Recovery, {ProcessName,
                     self()}, GroupName]),
                    s_group:register_name(GroupName, SupervisorName, SupervisorPid);
                SupervisorPid ->
                    link(SupervisorPid),
                    SupervisorPid! {after_recovery,self()}
                end,

                receive
                    {sup_submaster,ChildProcesses} ->
                        ok
                end,

                %ChildProcesses = lists:map(
                %fun(NextProcessIndex) ->
                %   ChildProcessName=ant_submaster:generate_submaster_name(CurrentLevel+1,
                Num_Processes*(NodeIndex-1)+ProcessIndex, NextProcessIndex),
                %   case s_group:whereis_name(GroupName, ChildProcessName) of
                %   undefined ->
                %       ChildPID=spawn_link(ChildNode,ant_submaster,run,[Num_Jobs,
                Num_Processes, Duplicating, Num_Ants, Iter_Global, Iter_Local, Inputs,
                Params#params{parent_printer=Chaos_starter}, Nodes, CurrentLevel+1,
                MaxLevel,Num_Processes*(NodeIndex-1)+ProcessIndex,NextProcessIndex,
                ChildProcessName, Recovery]),
                %       s_group:register_name(GroupName, ChildProcessName, ChildPID),
                %       {ChildPID, ChildProcessName, NextProcessIndex};
                %   ChildPID -> link(ChildPID),
                %       {ChildPID, ChildProcessName, NextProcessIndex}
                %   end
                %end,
                %lists:seq(1, Num_Processes)),

                ChildPIDs=lists:map (fun({ChildPID, _ProcessName, _ProcessIndex}) -> ChildPID
                end,ChildProcesses),
                if
                    Chaos==true andalso Recovery==false ->
                        ?Print(io_lib:format("submaster - sending ~p pids of colonies from
                        node ~p to ~p " , [length(ChildPIDs), node(), Chaos_starter])),
                        Chaos_starter! {pids, ChildPIDs++[SupervisorPid]};
                        %Chaos_starter! {pids, []};
                    true ->
                        ok
                end,
                loop (ChildProcesses, none, _Duplicating=1, SubMaster_state#submaster_state{
                supervisor=SupervisorPid,chaos_starter=Chaos_starter})
    end.

add_s_group_last_level(GroupName, Nodes) ->
    case exist_group(GroupName) of
        true ->
            ok;
```

```
        false ->
            case s_group:new_s_group(GroupName, Nodes) of
                {ok, GroupName, _Nodes} ->
                    {ok, GroupName, _Nodes};
                Error ->
                    ?Print(io_lib:format("sd_erlang: an error occurs (~p) when trying to
                    create group ~p on node ~p " , [Error, GroupName, node()]))
            end
    end.

add_s_group(GroupName, ProcessIndex, Nodes) ->
    if
        ProcessIndex==1 ->
            case exist_group(GroupName) of
                true ->
                    ok;
                false ->
                    case s_group:new_s_group(GroupName, Nodes) of
                        {ok, GroupName, _Nodes} ->
                            {ok, GroupName, _Nodes};
                        Error ->
                            ?Print(io_lib:format("sd_erlang: an error occurs (~p) when
                            trying to create group ~p on node ~p for process index ~p " , [
                            Error, GroupName, node(), ProcessIndex]))
                    end
            end;
        true ->
            %% wait untill sgroup gets created
            case exist_group_with_wait(GroupName) of
                true->
                    [_Head|Tail]=Nodes,
                    s_group:add_nodes(GroupName, Tail);
                    %?Print(io_lib:format("sd_erlang:add nodes ~p to group ~p for process
                    index ~p on node ~p " , [Tail, GroupName, ProcessIndex, node()]));
                _->
                    ?Print(io_lib:format("sd_erlang: group name ~p is not in my groups on
                    node ~p and process index ~p " , [GroupName, node(), ProcessIndex]))
            end
    end.

exist_group(GroupName) ->
    case s_group:s_groups() of
    undefined ->
        false;
    {OwnSGroups,_} ->
        if
            is_list(OwnSGroups) ->
                case lists:member(GroupName, OwnSGroups) of
                    true->
                        true;
                    _->
                    false
                end;
            true ->
                false
        end
    end.

exist_group_with_wait(GroupName) ->
    exist_group_with_wait(GroupName, 10). %% waits 10 milliseconds for s_group to be created

exist_group_with_wait(_GroupName, 0) ->
```

```
    false;
exist_group_with_wait(GroupName, Times) ->
    case exist_group(GroupName) of
        true->
            true;
        _->
            timer:sleep(1),
            exist_group_with_wait(GroupName, Times-1)
    end.

%% auxiliary functions to create a tree of submaster nodes

%% calculates the number of levels in the tree
%% each node has "Num_Processes" processes
%% each process supervise one node in lower level, except the last level that each process supervises "Num_Processes" number of nodes

%% calculates the tree level based on the total number of nodes and node degree (degree of vertices)
find_max_level(Num_Nodes,Num_Processes) ->
    S=speculate_level(Num_Nodes,Num_Processes),
    find_max_level(Num_Nodes,Num_Processes,S).

find_max_level(Num_Nodes,Num_Processes,Speculated_level) ->
    Result=calc_formula(Num_Processes,Speculated_level),
    if
        Result ==0 -> {_Level=0,_NumNodes=0};
        Result =< Num_Nodes -> {Speculated_level,Result};
        true -> find_max_level(Num_Nodes,Num_Processes,Speculated_level-1)
    end.

%% finds the largest possible power for Num_Processes to be less than Num_Nodes
speculate_level(Num_Nodes,Num_Processes)->
    speculate_level(Num_Nodes,Num_Processes,0).

speculate_level(Num_Nodes,Num_Processes,Acc)->
    Result=math:pow(Num_Processes,Acc),
    if
         Result<Num_Nodes ->
             speculate_level(Num_Nodes,Num_Processes,Acc+1);
        true ->
            round(Acc-1)
    end.

%% calculates 1+P^1+P^2+...+P^(N-2)+P^N
calc_formula(Num_Processes,Level) when Level>=2 ->
    calc_formula(Num_Processes,Level,_Acc=0,_Current_level=0);

%% No submaster can be allocated
calc_formula(_Num_Processes,_Level) ->
    0.

calc_formula(Num_Processes,Last_Level,Acc,Current_level) when Current_level=<Last_Level ->
    Num_Nodes=math:pow(Num_Processes,Current_level),
    case Current_level+2 of
        Last_Level ->
            calc_formula(Num_Processes,Last_Level,Acc+Num_Nodes,Current_level+2);
        _->
            calc_formula(Num_Processes,Last_Level,Acc+Num_Nodes,Current_level+1)
    end;

calc_formula(_Num_Processes,_Last_Level,Acc,_Current_level) ->
```

```
    round(Acc).

%% returns number of nodes for a specific level
nodes_in_level(Num_Processes, Level) ->
    round(math:pow(Num_Processes,Level-1)).

%% returns all the child nodes of a specific node. Node is specified by its level and its index in the level
%% How to test: Nodes=lists:seq(1, 277).
list_to_tuple(ant_submaster:get_childs(4,3,4,Nodes,1)).
get_childs(Num_Processes,CurrentLevel,MaxLevel,Nodes,IndexInLevel) -> %when CurrentLevel<MaxLevel ->
    Num_Node_in_Current_Level=nodes_in_level(Num_Processes,CurrentLevel),
    if
        Num_Node_in_Current_Level<IndexInLevel ->
            throw(index_in_level_is_more_than_num_nodes_in_level);
        true -> ok
    end,
    if
        CurrentLevel>=MaxLevel ->
            Num_Node_in_Next_Level=0,
            throw(current_level_must_be_less_than_max_level);
        CurrentLevel==MaxLevel-1 ->
            Num_Node_in_Next_Level=nodes_in_level(Num_Processes,CurrentLevel+2);
        true->
            Num_Node_in_Next_Level=nodes_in_level(Num_Processes,CurrentLevel+1)
    end,
    Childs_Per_Node=Num_Node_in_Next_Level/Num_Node_in_Current_Level,
    Index_For_Next_Line=Childs_Per_Node*(IndexInLevel-1)+1,
    After_Me_This_Level=Num_Node_in_Current_Level-IndexInLevel,
    Child_index=level_index(Num_Processes,CurrentLevel)+IndexInLevel+After_Me_This_Level+Index_For_Next_Line,lists:sublist(Nodes, round(Child_index), round(Childs_Per_Node)).

%% returns the index of the first node for a specific level
level_index(Num_Processes,Level) ->
    level_index(Num_Processes,Level,0,0).

level_index(Num_Processes,Level,Acc,CurrentLevel) when CurrentLevel<Level-1 ->
    R=math:pow(Num_Processes,CurrentLevel),
    level_index(Num_Processes,Level,Acc+R,CurrentLevel+1);

level_index(_Num_Processes,_Level,Acc,_CurrentLevel)->
Acc.

generate_submaster_name(Level, NodeIndex, ProcessIndex) ->
    Name=integer_to_list(Level)++integer_to_list(NodeIndex)++integer_to_list(ProcessIndex),
    list_to_atom(Name).

%% Recovers a failed process
recover_childs(FailedPID, SubMaster_state, ChildProcesses) ->
    #submaster_state{
    num_Jobs=Num_Jobs,
    num_Processes=Num_Processes,
    processName=ProcessName,
    duplicating=Duplicating,
    num_Ants=Num_Ants,
    iter_Global=Iter_Global,
    iter_Local=Iter_Local,
    inputs=Inputs,
    params=Params,
    nodes=Nodes,
    currentLevel=CurrentLevel,
```

```
        maxLevel=MaxLevel,
        nodeIndex=NodeIndex,
        processIndex=ProcessIndex,
        supervisor=SupervisorPid,
        chaos_starter=Chaos_starter} = SubMaster_state,
        if
            CurrentLevel==MaxLevel-1 ->
                case ant_master:get_process_name(FailedPID, ChildProcesses) of
                not_found ->
                    ?Print(io_lib:format("No recovery for colony process ~p is needed on node
                    ~p ", [FailedPID, node()])),
                    {no_updated, ChildProcesses};
                {ChildProcessName, ProcessGlobalIndex} ->
                    Node=lists:nth(ProcessGlobalIndex, Nodes),
                    NewPID=spawn_link(Node, ant_colony, init, [Num_Ants, Duplicating,
                    Iter_Local, Num_Jobs, Inputs, Params, ChildProcessName, _Recovery=true]),
                    util:send_pid(NewPID),
                    GroupName="group_"++ProcessName,
                    s_group:register_name(GroupName, ChildProcessName, NewPID),
                    NewChildProcesses=ant_master:update_process_PID(ChildProcessName, NewPID,
                    ChildProcesses),
                    ?Print(io_lib:format("recovery of a colony process ~p on node ~p by new
                    length ~p and new PID ~p ", [FailedPID, node(), length(NewChildProcesses),
                    NewPID])),
                    {updated, NewChildProcesses}
                end;
            true ->
                if
                    SupervisorPid==FailedPID ->
                        ?Print(io_lib:format("Failure of a supervisor process with pid ~p on
                        node ~p", [FailedPID, node()])),
                        ChildNodes=get_childs(Num_Processes,CurrentLevel,MaxLevel,Nodes,
                        NodeIndex),
                        ChildNode=lists:nth(ProcessIndex, ChildNodes),
                        GroupName="group_"++atom_to_list(node()),
                        NewSupervisorPid=spawn_link(ChildNode,sup_submaster,run,[Num_Jobs,
                        Num_Processes, Duplicating, Num_Ants, Iter_Global, Iter_Local, Inputs,
                        Params#params{parent_printer=Chaos_starter}, Nodes, CurrentLevel+1,
                        MaxLevel, Num_Processes*(NodeIndex-1)+ProcessIndex, _Recovery=true, {
                        ProcessName, self()}, GroupName]),
                        SupervisorName=list_to_atom("sup_"++atom_to_list(ProcessName)),
                        s_group:register_name(GroupName, SupervisorName, NewSupervisorPid),
                        receive
                            {sup_submaster,ChildProcesses} ->
                                ok
                        end,
                        {updated, ChildProcesses, NewSupervisorPid};
                    true ->
                        case ant_master:get_process_name(FailedPID, ChildProcesses) of
                        not_found ->
                            ?Print(io_lib:format("No recovery for sub-master process ~p is
                            needed on node ~p ", [FailedPID, node()])),
                            {no_updated, ChildProcesses};
                        {ChildProcessName, NextProcessIndex} ->
                            ChildNodes=get_childs(Num_Processes,CurrentLevel,MaxLevel,Nodes,
                            NodeIndex),
                            ChildNode=lists:nth(ProcessIndex, ChildNodes),
                            NewPID=spawn_link(ChildNode,ant_submaster,run,[Num_Jobs,
                            Num_Processes, Duplicating, Num_Ants, Iter_Global, Iter_Local,
                            Inputs, Params,Nodes, CurrentLevel+1, MaxLevel,Num_Processes*(
                            NodeIndex-1)+ProcessIndex,NextProcessIndex, ChildProcessName,
                            _Recovery=true]),
```

```
                        util:send_pid(NewPID),
                        GroupName="group_"++atom_to_list(node()),
                        s_group:register_name(GroupName, ChildProcessName, NewPID),
                        NewChildProcesses=ant_master:update_process_PID(ChildProcessName,
                        NewPID, ChildProcesses),
                        ?Print(io_lib:format("recovery of a sub-master process ~p on node
                        ~p by new length ~p and new PID ~p ", [FailedPID, node(), length(
                        NewChildProcesses), NewPID])),
                        {updated, NewChildProcesses}
                end
        end
end.
```

Code C.21: Sub-Master Module (ant_submaster.erl)

```
%% Supervises a number of sub-master processes locally

%% Author: Amir Ghaffari <Amir.Ghaffari@glasgow.ac.uk>
%% RELEASE project (http://www.release-project.eu/)

-module(sup_submaster).
-include("types.hrl").
-compile(export_all).


run(Num_Jobs, Num_Processes, Duplicating, Num_Ants, Iter_Global, Iter_Local, Inputs, Params
, Nodes, CurrentLevel, MaxLevel, NodeIndex, Recovery, {ParentName, ParentPid}, GroupName) ->
    process_flag(trap_exit, true),
    #params{printer=Printer} = Params,
    if
        Printer==false->
            put(parent_printer, no_print);
        true->
            put(parent_printer, Params#params.parent_printer)
    end,
    case ant_submaster:exist_group_with_wait(GroupName) of
        true->
            ok;
        _->
            ?Print(io_lib:format("sd_erlang: group name ~p is not available on node
            sup_submaster process ~p " , [GroupName, node()]))
    end,
    if
        Recovery==false ->
            ?Print(io_lib:format("creating a sup_submaster process on node ~p " , [node()]));
        true ->
            ?Print(io_lib:format("recovery a sup_submaster process on node ~p " , [node()]))
    end,
    ChildProcesses = lists:map(
    fun(NextProcessIndex) ->
        ChildProcessName=ant_submaster:generate_submaster_name(CurrentLevel, NodeIndex,
        NextProcessIndex),
        %case s_group:whereis_name(GroupName, ChildProcessName) of
        case whereis(ChildProcessName) of
        undefined ->
            ChildPID=spawn_link(ant_submaster,run,[Num_Jobs, Num_Processes, Duplicating,
            Num_Ants, Iter_Global, Iter_Local, Inputs, Params, Nodes, CurrentLevel,
            MaxLevel, NodeIndex, NextProcessIndex, ChildProcessName, Recovery]),
            %s_group:register_name(GroupName, ChildProcessName, ChildPID),
            register(ChildProcessName, ChildPID),
            {ChildPID, ChildProcessName, NextProcessIndex};
        ChildPID -> link(ChildPID),
            {ChildPID, ChildProcessName, NextProcessIndex}
        end
    end,
    lists:seq(1, Num_Processes)),
    ReliablePid=get_Pid_reliably_sgroup(GroupName, ParentName, ParentPid),
    ReliablePid!{sup_submaster,ChildProcesses},
    %Parent!{sup_submaster,ChildProcesses},
    monitor(Num_Jobs, Num_Processes, Duplicating, Num_Ants, Iter_Global, Iter_Local, Inputs
    , Params, Nodes, CurrentLevel, MaxLevel, NodeIndex, Recovery, {ParentName, ParentPid},
    GroupName, ChildProcesses, _Terminating=false).

monitor(Num_Jobs, Num_Processes, Duplicating, Num_Ants, Iter_Global, Iter_Local, Inputs,
Params, Nodes, CurrentLevel, MaxLevel, NodeIndex, Recovery, {ParentName, ParentPid},
GroupName, ChildProcesses, Terminating) ->
    receive
```

```
        {ParentPid, stop_ants} ->
            monitor(Num_Jobs, Num_Processes, Duplicating, Num_Ants, Iter_Global, Iter_Local
            , Inputs, Params, Nodes, CurrentLevel, MaxLevel, NodeIndex, Recovery, {
            ParentName, ParentPid}, GroupName, ChildProcesses, _Terminating=true);
        {after_recovery,NewParentPid} ->
            NewParentPid! {sup_submaster,ChildProcesses},
            monitor(Num_Jobs, Num_Processes, Duplicating, Num_Ants, Iter_Global, Iter_Local
            , Inputs, Params, Nodes, CurrentLevel, MaxLevel, NodeIndex, Recovery, {
            ParentName, NewParentPid}, GroupName, ChildProcesses, Terminating);
        {'EXIT', FailedPID, Reason} ->
            if
                Terminating==false ->
                    ?Print(io_lib:format("Failure of a sub-master process with PID ~p and
                    reason ~p ", [FailedPID, Reason])),
                    case ant_master:get_process_name(FailedPID, ChildProcesses) of
                        not_found ->
                            ?Print(io_lib:format("No recovery for sub-master process ~p is
                            needed on node ~p ", [FailedPID, node()])),
                            monitor(Num_Jobs, Num_Processes, Duplicating, Num_Ants,
                            Iter_Global, Iter_Local, Inputs, Params, Nodes, CurrentLevel,
                            MaxLevel, NodeIndex, Recovery, {ParentName, ParentPid},
                            GroupName, ChildProcesses, Terminating);
                        {ChildProcessName, NextProcessIndex} ->
                            NewPID=spawn_link(ant_submaster,run,[Num_Jobs, Num_Processes,
                            Duplicating, Num_Ants, Iter_Global, Iter_Local, Inputs, Params,
                             Nodes, CurrentLevel, MaxLevel, NodeIndex, NextProcessIndex,
                            ChildProcessName, _Recovery=true]),
                            s_group:register_name(GroupName, ChildProcessName, NewPID),
                            NewChildProcesses=ant_master:update_process_PID(
                            ChildProcessName, NewPID, ChildProcesses),
                            ?Print(io_lib:format("recovery of a sub-master process ~p on
                            node ~p by new length ~p and new PID ~p ", [FailedPID, node(),
                            length(NewChildProcesses), NewPID])),
                            ReliablePid=get_Pid_reliably_sgroup(GroupName, ParentName,
                            ParentPid),
                            ReliablePid!{afterFailure,NewChildProcesses, FailedPID},
                            monitor(Num_Jobs, Num_Processes, Duplicating, Num_Ants,
                            Iter_Global, Iter_Local, Inputs, Params, Nodes, CurrentLevel,
                            MaxLevel, NodeIndex, Recovery, {ParentName, ParentPid},
                            GroupName, NewChildProcesses, Terminating)
                    end;
                true ->
                    ReliablePid=get_Pid_reliably_sgroup(GroupName, ParentName, ParentPid),
                    ReliablePid! {'EXIT', FailedPID, Reason},
                    monitor(Num_Jobs, Num_Processes, Duplicating, Num_Ants, Iter_Global,
                    Iter_Local, Inputs, Params, Nodes, CurrentLevel, MaxLevel, NodeIndex,
                    Recovery, {ParentName, ParentPid}, GroupName, ChildProcesses,
                    Terminating)
            end
    end.

get_Pid_reliably_sgroup(GroupName, ProcessName, ProcessPid) when not is_integer(ProcessPid)->
    if
        is_pid(ProcessPid)==true ->
            ProcessPid;
        true->
            get_Pid_reliably_sgroup(GroupName, ProcessName, 3)
    end;

get_Pid_reliably_sgroup(GroupName, ProcessName, 0)->
    ?Print(io_lib:format("After 3 times attempts, ant_submaster process is not available
    in group ~p with name ~p", [GroupName, ProcessName])),
```

```
    undefined;
get_Pid_reliably_sgroup(GroupName, ProcessName, Num)->
    case s_group:whereis_name(GroupName, ProcessName) of
    undefined ->
        timer:sleep(1),
        get_Pid_reliably_sgroup(GroupName, ProcessName, Num-1);
    Pid ->
        if
            is_pid(Pid)==true ->
                Pid;
            true->
                timer:sleep(1),
                get_Pid_reliably_sgroup(GroupName, ProcessName, Num-1)
        end
    end.
```

Code C.22: Local Supervisor Module (sup_submaster.erl)

Appendix D

D.1 Scalability of Sim-Diasca on the GPG Cluster

This section measures the scalability of Sim-Diasca at the GPG cluster [113]. We use the Sim-Diasca City benchmark at different scales, i.e *medium*, *large*, and *huge* [1]. In our measurement, the runtime of the benchmark is compared at different cluster sizes, i.e. 1, 2, 3, 4, 5, 10, and 15 nodes. The city benchmark has two phases, i.e. *initialisation* and *execution*. The initialisation phase is excluded from our measurement to reduce the runtime and increase parallelism. Code 21 and 22 are used for the initialisation and execution phases respectively.

Code 21: Initialisation Phase

```
make generate CASE_SCALE=${Scale} EXECUTION_TARGET=production
```

Code 22: Execution Phases

```
make run CASE_SCALE=${Scale} CASE_DURATION=${Duration} EXECUTION_TA
```

Figures D.1, D.2, and D.3 show the scalability of the City benchmark for different sizes. As we see from the figures, the medium size does not scale at all. This can be due to the size of the problem. It seems, the medium size is not large enough to take benefit from all the available resources at the cluster. The large size scales up to 4 nodes and beyond that the runtime rises despite increasing the number of nodes. The huge size scales up to 3 nodes and beyond that the runtime remains roughly constant.

D.1.1 Resource Usage

To identify possible bottlenecks of the simulation scalability we measure the usage of random access memory (RAM) and cores. Figure D.4 shows core utilisation of nodes in the cluster. We see from the figure that as cluster grows, core utilisation declines. The maximum core

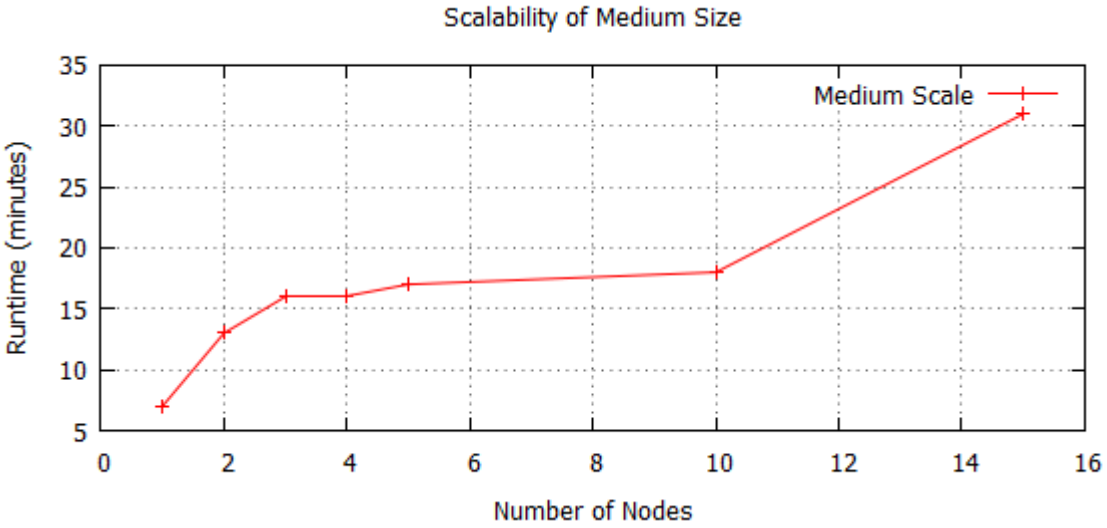


Figure D.1: The Scalability of Medium Size

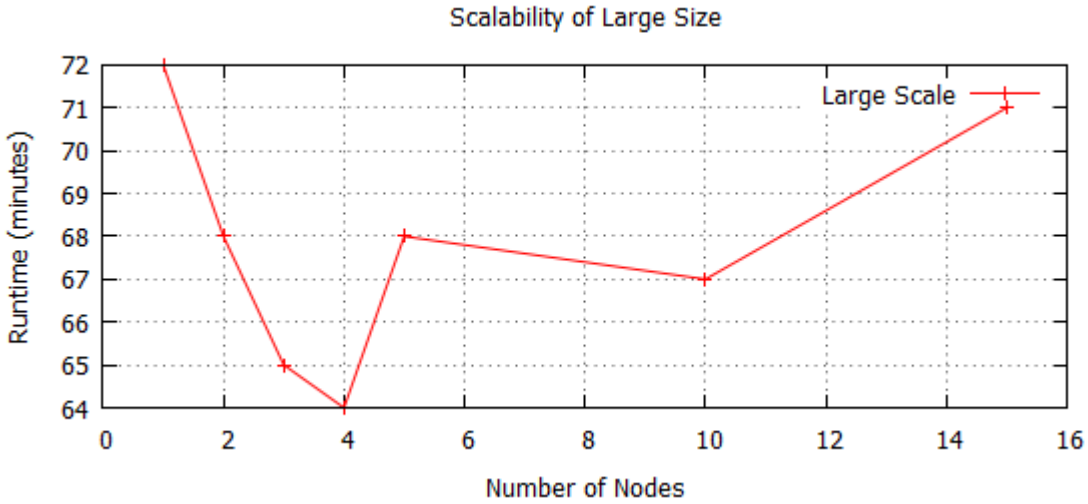


Figure D.2: The Scalability of Large Size

unitilisation belongs to single node which is 320% (3.2 cores out of 16 available cores). Memory utilisation is depicted in figure D.5. We see from the figure that the maximum memory usage belongs to 3-node cluster which is 45% percentage of the total memory (each node has 64 gigabyte memory).

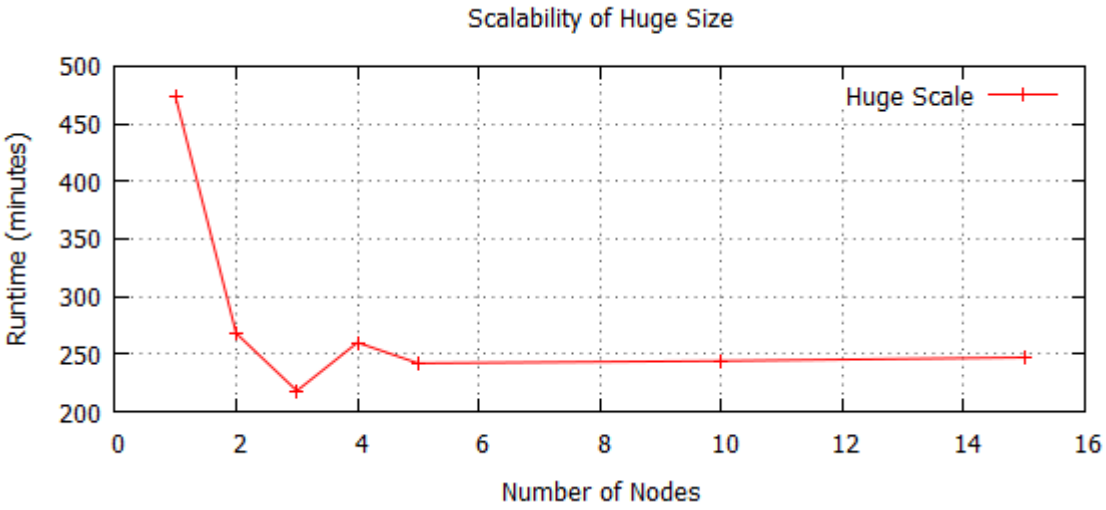


Figure D.3: The Scalability of Huge Size

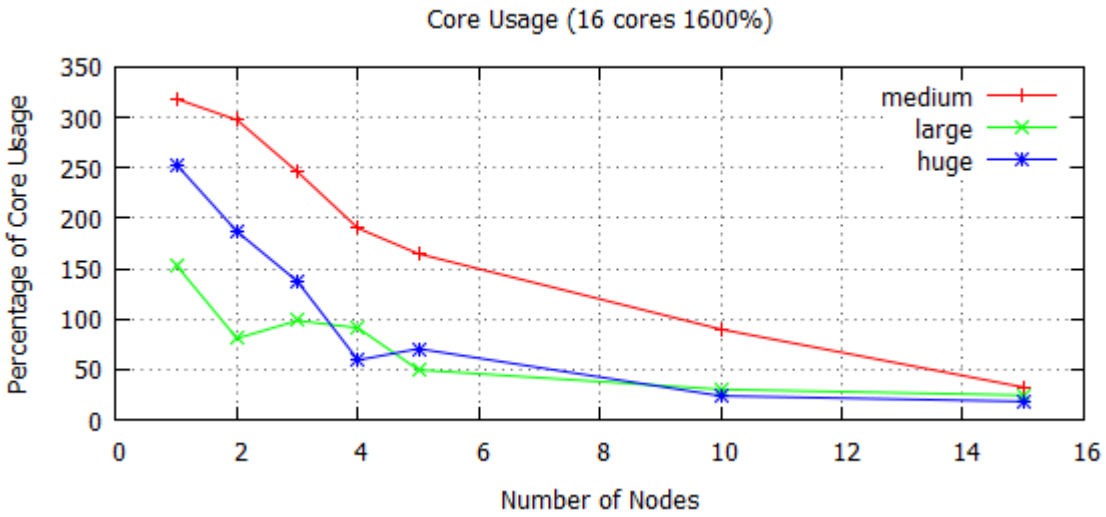


Figure D.4: Core Unitilisation

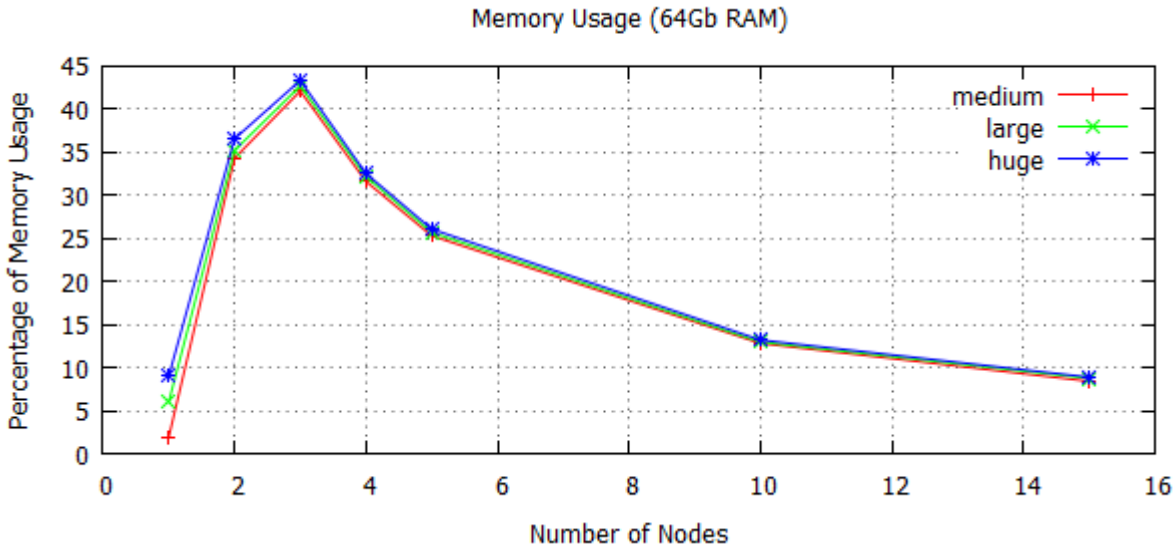


Figure D.5: Memory Unitilisation

Bibliography

[1] O. Boudeville, "Technical Manual of the Sim-Diasca Simulation Engine," EDF R&D, Tech. Rep., 2010.

[2] RELEASE, "RELEASE Project. Deliverable D3.4: Scalable Reliable OTP Library Release," Tech. Rep., 2014. [Online]. Available: http://release-project.softlab.ntua.gr/documents/D3.4.pdf

[3] Ericsson, "Who uses Erlang for product development?" 2014. [Online]. Available: http://www.erlang.org/faq/introduction.html#idp27538960

[4] O. Boudeville, F. Cesarini, N. Chechina, K. Lundin, N. Papaspyrou, K. Sagonas, S. Thompson, P. Trinder, and U. Wiger, "RELEASE: a high-level paradigm for reliable large-scale server software," in *Proceedings of the Symposium on Trends in Functional Programming*, St Andrews, UK, 2012.

[5] A. Ghaffari, N. Chechina, P. Trinder, and J. Meredith, "Scalable Persistent Storage for Erlang : Theory and Practice," in *Proceedings of the Twelfth ACM SIGPLAN Erlang Workshop*. Boston: ACM, 2013, pp. 73–74.

[6] RELEASE, "RELEASE Project. Deliverable D3.1: Scalable Reliable SD Erlang Design," Tech. Rep., 2012. [Online]. Available: http://release-project.softlab.ntua.gr/documents/D3.1.pdf

[7] A. Ghaffari, N. Chechina, P. Trinder, and J. Meredith, "Scalable Persistent Storage for Erlang: Theory and Practice," Heriot-Watt University, Edinburgh, Tech. Rep., 2013. [Online]. Available: http://www.macs.hw.ac.uk/cs/techreps/doc0098.html

[8] N. Chechina, H. Li, A. Ghaffari, S. Thompson, and P. Trinder, "Improving Network Scalability of Erlang," *Journal of Parallel and Distributed Computing*, 2015.

[9] A. Ghaffari, "Investigating the Scalability Limits of Distributed Erlang," in *Proceedings of the Thirteenth ACM SIGPLAN Erlang Workshop*. Gothenburg: ACM, 2014, pp. 43–49.

[10] RELEASE, "RELEASE Project. Deliverable D3.2: Scalable SD Erlang Computation Model," Tech. Rep., 2013. [Online]. Available: http://release-project.softlab.ntua.gr/documents/D3.2.pdf

[11] T. Hof, "Netflix: Continually Test by Failing Servers with Chaos Monkey," 2010. [Online]. Available: http://highscalability.com/blog/2010/12/28/netflix-continually-test-by-failing-servers-with-chaos-monke.html

[12] M. O. Tokhi, M. A. Hossain, and M. H. Shaheed, *Parallel Computing for Real-time Signal Processing and Control*, 1st ed. Springer, 2003.

[13] T. Rauber and G. Runger, *Parallel Programming for Multicore and Cluster Systems*. Springer, 2010.

[14] H. El-Rewin and M. Abd-El-Barr, *Advanced Computer Architecture and Parallel Processing*, 1st ed. Wiley-Interscience, 2005.

[15] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "Exploration of Distributed Shared Memory Architectures for NoC-based Multiprocessors," in *Embedded Computer Systems: Architectures, Modeling and Simulation*. Samos: IEEE, 2006, pp. 144 –151.

[16] R. Buyya, *High Performance Cluster Computing: Architectures and Systems*. NJ, USA: Prentice Hall PTR Upper Saddle River, 1999.

[17] N. Sadashiv and S. M. D. Kumar, "Cluster, grid and cloud computing: A detailed comparison," in *The 6th International Conference on Computer Science & Education (ICCSE 2011)*. Singapore: IEEE, 2011, pp. 477–482.

[18] L. Chai, Q. Gao, and D. K. Panda, "Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System," in *Seventh IEEE International Symposium on Cluster Computing and the Grid(CCGrid'07)*. Rio De Janeiro: IEEE, 2007, pp. 471–478.

[19] R. Buyya, T. Cortes, and H. Jin, "SINGLE SYSTEM IMAGE (SSI)," *The International Journal of High Performance Computing Applications*, vol. 15, pp. 124–135, 2001.

[20] K. Krauter, R. Buyya, and M. Maheswaran, "A taxonomy and survey of grid resource management systems for distributed computing," *Software Practice and Experience*, vol. 32, pp. 135–164, 2002.

[21] R. Buyya, J. Giddy, and D. Abramson, "An Evaluation of Economy-based Resource Trading and Scheduling on Computational Power Grids for Parameter Sweep Applications," *Proceedings of the 2nd Annual Workshop on Active Middleware Services*, vol. 583, pp. 221–230, 2000.

[22] Globus, "About the Globus Toolkit," 2014. [Online]. Available: http://toolkit.globus.org/toolkit/about.html

[23] S. J. Chapin, D. Katramatos, J. Karpovich, and A. S. Grimshaw, "The Legion Resource Management System," in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds., vol. 1659. Puerto Rico: Springer, 1999, pp. 162–178.

[24] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, "Condor-G: A Computation Management Agent for Multi-Institutional Grids," *Cluster Computing*, vol. 5, no. 3, pp. 237–246, 2002.

[25] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, 2009.

[26] R. Buyya, *High Performance Cluster Computing: Programming and Applications*, 1st ed., R. Buyya, Ed. NJ, USA: Prentice Hall, 1999.

[27] S. Akhter and J. Roberts, *Multi-Core Programming : Increasing Performance through Software Multi-threading*, 1st ed. Intel Corporation, 2006.

[28] Oracle, "Developing Parallel Programs: A Discussion of Popular Models," 2010. [Online]. Available: http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/oss-parallel-programs-170709.pdf

[29] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine*, J. Kowalik, Ed. MIT Press, 1994.

[30] M. Collette, B. Corey, and J. Johnson, "High Performance Tools & Technologies," Lawrence Livermore National Laboratory, Tech. Rep., 2004.

[31] A. Mainwaring and D. Culler, "Active Message Applications Programming Interface and Communication Subsystem Organization," University of California at Berkeley, Tech. Rep.

[32] A. D. Selvakumar, P. M. Sobha, G. C. Ravindra, and R. Pitchiah, "Design, Implementation and Performance of Fault-Tolerant Message Passing Interface (MPI)," *HPCASIA*, pp. 120 – 129.

[33] D. Padua, *Encyclopedia of Parallel Computing*. Springer, 2011.

[34] J. Armstrong, *Programming Erlang: Software for a Concurrent World*, 1st ed. Pragmatic Bookshelf, 2007.

[35] D. Pollak, *Beginning Scala*. Apress, 2009.

[36] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*, 2nd ed. ARTIMA PRESS, 2010.

[37] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed., M. Horton, Ed. Addison-Wesley, 2011.

[38] S. Abiteboul, I. Manolescu, P. Rigaux, M.-C. Rousset, and P. Senellart, *Web Data Management*. Cambridge University Press, 2011.

[39] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[40] W. Jia and W. Zhou, *Distributed Network Systems: From Concepts to Implementations*. Springer, 2005.

[41] M. Arango and B. Kaponig, "Ultra-scalable architectures for Telecommunications and Web 2.0 services," in *Intelligence in Next Generation Networks*. Bordeaux: IEEE, 2009, pp. 1–7.

[42] L. Kari and G. Rozenberg, "The many facets of natural computing," *Communications of the ACM*, pp. 72–83, 2008.

[43] WhatsApp, "WhatsApp," 2014. [Online]. Available: http://www.whatsapp.com/

[44] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Pena, S. Priebe, A. J. Rebon, and P. W. Trinder, "Comparing Parallel Functional Languages: Programming and Performance," *Higher-Order and Symbolic Computation*, vol. 16, no. 3, pp. 203–251, 2003.

[45] T. Petricek and J. Skeet, *Real-World Functional Programming: With Examples in F# and C#*, 1st ed. Manning Publications, 2010.

[46] Ericsson, "Processes," 2014. [Online]. Available: http://www.erlang.org/doc/efficiency_guide/processes.html

[47] F. Cesarini and S. Thompson, *Erlang Programming*, 1st ed. O'Reilly Media, 2009.

[48] M. L. Scott, *Programming Language Pragmatics*, 3rd ed. Morgan Kaufmann, 2009.

[49] C. Hewitt, P. Bishop, and P. Bishop, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd international joint conference on Artificial intelligence*. San Francisco: Morgan Kaufmann Publishers Inc, 1973, pp. 235– 245.

[50] Ericsson, "Distributed Erlang," 2014. [Online]. Available: http://www.erlang.org/doc/reference_manual/distributed.html

[51] Ericsson, "Communicates with a running port mapper daemon," 2014. [Online]. Available: http://www.erlang.org/doc/man/epmd.html

[52] Ericsson, "Using SSL for Erlang Distribution," 2014. [Online]. Available: http://www.erlang.org/doc/apps/ssl/ssl_distribution.html

[53] Ericsson, "Remote Procedure Call Services," 2014. [Online]. Available: http://erlang.org/doc/man/rpc.html

[54] Ericsson, "A Global Name Registration Facility," 2014. [Online]. Available: http://www.erlang.org/doc/man/global.html

[55] Ericsson, "Built-In Term Storage," 2014. [Online]. Available: http://www.erlang.org/doc/man/ets.html

[56] Ericsson, "A Disk Based Term Storage," 2014. [Online]. Available: http://www.erlang.org/doc/man/dets.html

[57] Ericsson, "OTP Design Principles User's Guide," 2014. [Online]. Available: http://www.erlang.org/doc/design_principles/des_princ.html

[58] Ericsson, "Generic Server Behaviour," 2014. [Online]. Available: http://erlang.org/doc/man/gen_server.html

[59] Ericsson, "Gen_Fsm Behaviour," 2014. [Online]. Available: http://www.erlang.org/doc/design_principles/fsm.html

[60] Ericsson, "Generic Finite State Machine Behaviour," 2014. [Online]. Available: http://erlang.org/doc/man/gen_fsm.html

[61] Ericsson, "Generic Event Handling Behaviour," 2014. [Online]. Available: http://www.erlang.org/doc/man/gen_event.html

[62] Ericsson, "Supervisor Behaviour," 2014. [Online]. Available: http://www.erlang.org/doc/design_principles/sup_princ.html

[63] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo : Amazon's Highly Available Key-value Store," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. Stevenson: ACM, 2007, pp. 205–220.

[64] A. Lakshman and P. Malik, "Cassandra - A Decentralized Structured Storage System," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[65] F. A. Y. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable : A Distributed Storage System for Structured Data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, 2008.

[66] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s Hosted Data Serving Platform," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, 2008.

[67] D. McMurtry, A. Oakley, J. Sharp, M. Subramanian, and H. Zhang, *Data Access for Highly-Scalable Solutions: Using SQL, NoSQL, and Polyglot Persistence*, 1st ed. Microsoft patterns & practices, 2013.

[68] Basho, "Riak," 2014. [Online]. Available: http://basho.com/riak/

[69] Citrusbyte, "The Redis Documentation," 2014. [Online]. Available: http://redis.io/documentation

[70] Amazon, "Amazon SimpleDB," 2014. [Online]. Available: http://aws.amazon.com/nosql/

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

[72] MongoDB Inc, "MongoDB," 2014. [Online]. Available: http://www.mongodb.org/

[73] Apache Software Foundation, "The Apache HBase Reference Guide," 2014. [Online]. Available: http://hbase.apache.org/book/book.html

[74] Cassandra Apache, "Apache Cassandra," 2014. [Online]. Available: http://cassandra.apache.org/

[75] NeoTechnology, "Neo4j," 2014. [Online]. Available: http://neo4j.com/

[76] FactNexus, "GraphBase," 2014. [Online]. Available: http://graphbase.net/

[77] Objectivity, "InfiniteGraph," 2014. [Online]. Available: http://www.objectivity.com/infinitegraph

[78] E. Hewitt, *Cassandra: The Definitive Guide*, 1st ed. O'Reilly Media, 2010.

[79] D. Kargerl, T. Leightonl, and D. Lewinl, "Consistent Hashing and Random Trees : Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. El Paso, TX, USA: ACM Press, 1997, pp. 654–663.

[80] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*. Portland, Oregon: ACM, 2000, p. 7.

[81] P. A. Bernstein and S. Das, "Rethinking eventual consistency," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. New York, USA: ACM, 2013, pp. 923–928.

[82] W. Vogels, "Eventually Consistent," *Communications of the ACM - Rural engineering development*, vol. 52, no. 1, pp. 40–44, 2009.

[83] M. Takada, "Replication: weak consistency model protocols," 2014. [Online]. Available: http://book.mixu.net/distsys/eventual.html

[84] T. M. Connolly and C. E. Begg, *Database Systems: A Practical Approach to Design, Implementation and Management*, 4th ed. Addison Wesley, 2005.

[85] M. T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems*, 3rd ed. Springer, 2011.

[86] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2004, pp. 137–149.

[87] Ericsson, "Mnesia," 2014. [Online]. Available: http://www.erlang.org/doc/man/mnesia.html

[88] Ericsson, "Query List Comprehensions (QLC)," 2014. [Online]. Available: http://www.erlang.org/doc/man/qlc.html

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

[90] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*. Bolton Landing, NY: ACM Press, 2003, pp. 29–43.

[91] Basho, "Basho Bench," 2014. [Online]. Available: http://docs.basho.com/riak/latest/ops/building/benchmarking/

[92] SNIC-UPPMAX, "The Kalkyl Cluster," 2012. [Online]. Available: http://www.uppmax.uu.se/hardware

[93] E. Krevat, V. Vasudevan, A. Phanishayee, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan, "On application-level approaches to avoiding TCP throughput collapse in cluster-based storage systems," in *Proceedings of the 2Nd International Workshop on Petascale Data Storage*, New York, USA, 2007, pp. 1–4.

[94] S. L. Fritchie, "TCP incast: What is it? How can it affect Erlang applications?" 2012. [Online]. Available: http://www.snookles.com/slf-blog/2012/01/05/tcp-incast-what-is-it/

[95] Basho, "Basho Technologies, Inc." 2014. [Online]. Available: http://basho.com/

[96] Basho, "sidejob," 2014. [Online]. Available: https://github.com/basho/sidejob

[97] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," *SoCC '10 Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154, 2010.

[98] A. Ghaffari, "DE-Bench," 2014. [Online]. Available: https://github.com/amirghaffari/DEbench

[99] Ericsson, "Supervisor," 2014. [Online]. Available: http://www.erlang.org/doc/man/supervisor.html

[100] Ericsson, "Global Name Registration Facility," 2014. [Online]. Available: http://www.erlang.org/doc/man/global.html

[101] N. Chechina, P. Trinder, A. Ghaffari, R. Green, K. Lundin, and R. Virding, "The Design of Scalable Distributed Erlang," in *the Symposium on Implementation and Application of Functional Languages*, 2012.

[102] F. Lubeck and M. Neunhoffer, "Enumerating Large Orbits and Direct Condensation," *Experimental Mathematics*, vol. 10, no. 2, pp. 197–205, 2001.

[103] J. Matocha and T. Camp, "A taxonomy of distributed termination detection algorithms," *Journal of Systems and Software*, vol. 43, no. 221, pp. 207–221, 1998.

[104] R. McNaughton, "Scheduling with Deadlines and Loss Functions," *Management Science*, vol. 6, no. 1, pp. 1–12, 1959.

[105] M. Dorigo and T. Stützle, *Ant Colony Optimization*. Scituate, MA, USA: Bradford Company Scituate, 2004.

[106] G. Ghiani, D. Lagana, G. Laporte, and F. Mari, "Ant colony optimization for the arc routing problem with intermediate facilities under capacity and length restrictions," *ournal of Heuristics*, vol. 16, no. 2, pp. 211–233, 2010.

[107] W. Liao, E. Pan, and L. Xi, "A heuristics method based on ant colony optimisation for redundancy allocation problems," *International Journal of Computer Applications in Technology*, vol. 40, no. 1, pp. 71–78, 2011.

[108] M. J. Geiger, "On heuristic search for the single machine total weighted tardiness problem, Some theoretical insights and their empirical verification," *European Journal of Operational Research*, vol. 207, no. 3, pp. 1235–1243, 2010.

[109] T. Abdul-Razaq, C. Potts, and L. V. Wassenhove, "A survey of algorithms for the single machine total weighted tardiness scheduling problem," *Discrete Applied Mathematics*, vol. 26, no. 2, pp. 235–253, 1990.

[110] A. Bauer, B. Bullnheimer, R. F. Hartl, and C. Strauss, "Minimizing Total Tardiness on a Single Machine Using Ant Colony Optimization," *Central European Journal of Operations Research*, vol. 8, pp. 125–141, 2000.

[111] M. den Besten, T. Stutzle, and M. Dorigo, "Ant Colony Optimization for the Total Weighted Tardiness Problem," *Parallel Problem Solving from Nature*, vol. 1917, pp. 611–620, 2000.

[112] D. Merkle and M. Middendorf, "An Ant Algorithm with a New Pheromone Evaluation Rule for Total Tardiness Problems," *Real-World Applications of Evolutionary Computing*, vol. 1803, pp. 290–299, 2000.

[113] P. Trinder, "GPG Cluster," 2014. [Online]. Available: http://www.dcs.gla.ac.uk/research/gpg/cluster.htm

[114] Daniel Luna, "Chaos Monkey," 2012. [Online]. Available: https://github.com/dLuna/chaos_monkey

[115] Typesafe, "Akka Documentation," 2014. [Online]. Available: http://akka.io/docs/

[116] P. Maier, "Distributed Orbit," 2014. [Online]. Available: https://github.com/amirghaffari/Orbit

[117] Ghaffari, Amir, "Scalable Distributed Orbit," 2014. [Online]. Available: https://github.com/amirghaffari/SD-Orbit

[118] A. Ghaffari, "Ant Colony Optimisation," 2015. [Online]. Available: https://github.com/amirghaffari/ACO