Self Maintained Replication In a Distributed File System

*A Live-Forever Replication Framework For Emerald*

Ola Edvardsen Solberg

Thesis submitted for the degree of Master in Informatics: Programming and Networks
(Design Patterns For Cloud Computing)
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2017
Self Maintained Replication In a Distributed File System

A Live-Forever Replication Framework For Emerald

Ola Edvardsen Solberg
Contents

Table of Contents .......................................................... 1
1 Introduction ........................................................................ 4
  1.1 10 line Description ...................................................... 4
  1.2 Motivation ..................................................................... 4
  1.3 Problem Statement ....................................................... 5
    1.3.1 The problem we have/and are trying to solve .......... 5
  1.4 Goal .............................................................................. 5
    1.4.1 Platform to solve issue ........................................ 5
  1.5 Approach ....................................................................... 5
  1.6 Evaluation ..................................................................... 6
  1.7 Result ............................................................................ 6
  1.8 Outline .......................................................................... 6
2 Background .......................................................................... 6
  2.1 Distributed systems ...................................................... 7
  2.2 Design Patterns .......................................................... 7
  2.3 Replication .................................................................... 9
    2.3.1 Desired functionality ............................................ 9
  2.4 Emerald .......................................................................... 9
  2.5 Planetlab ......................................................................... 11
3 State of The Art ..................................................................... 11
  3.1 Survey of Erl, Cope, and Naserpour, Design Patterns for Cloud Computing ..................................................... 11
  3.2 General Design Pattern Areas ....................................... 11
    3.2.1 Sharing, scaling and elasticity Patterns ................. 11
    3.2.2 Reliability, Resiliency and Recovery Patterns ..... 12
    3.2.3 Data Management and Storage Device Patterns 12
    3.2.4 Virtual Server and Hypervisor Connectivity and Management Patterns ........................................ 12
    3.2.5 Monitoring, Provisioning and Administration Patterns ................................................................. 12
    3.2.6 Cloud Service and Storage Security Patterns ..... 13
    3.2.7 Network Security, Identity and Access Management and Trust Assurance Patterns 13
  3.3 Compound Design Patterns ............................................ 13
    3.3.1 Common Compound Patterns .............................. 14
  3.4 Replication Patterns ..................................................... 14
    3.4.1 Passive-replication .............................................. 14
    3.4.2 Active-replication .............................................. 15
    3.4.3 Self-replication .................................................. 15
Preface

The scope of this paper started with the general headline of "Design patterns for cloud computing" and the initial approach to tackle the scope was the reading and study of the book, "Design patterns for cloud computing" by Erl, Cope, and Naserpour, and writing a disposition of their work. This gave a good starting position and point of view on state of the art design patterns for cloud computing. After consulting with my supervisor we landed on looking at the possibilities of creating a framework for replication that could support itself, for the program language Emerald. With this as a basis and also experience with the primary replication scheme (passive replication) from my home exam in the course INF5510, which was also done in Emerald, I decided to work on this project. The work then started on planning the framework itself. Reading through several papers on distribution and replication, including understanding both the design patterns for passive- and active replication, studying the passive replication program I had previously built, and attempting to get a grip on how to create the self-replicating framework.

Initially I wanted to create a complete system, with support for clients to connect, and services to actually use the framework. I ended up staggering for a while, trying to figure out how clients should be able to get the reference list for the different replications, without the use of a single point of failure, like a naming service/server, which is normally used within the area of replication to group several different copies together. After more consultations with my supervisor, we came to the conclusion that this part actually did not have to be part of the project, since the construction of the replication framework itself was the main goal for the paper, and that other the inking of clients to the system was a completely different work, hence scaling down the scope of the project yet again, to only surround the framework for replication itself.

Work on the actual programming started with the base of the passive replication program I had previously created and with the use of the active replication design pattern, change the program from passive- to active replication. While working and on completion of the active replication program extensive testing was done through planetlab, with both a small and large amount of test nodes. From this point forward I had to implement the self-replication part of the framework and the the same time, planetlab which had been my test bed went down. All further testing from this point had to be done by simulating several nodes on a local machine, instead of on a real distributed system, up until the very last week before delivery.

Special thanks

I would like to extend a special thanks to Eric B Jul, at the University of Oslo, whom served as my supervisor. Being able to help me out within a short time limit, when crisis broke out, made this whole thesis possible.
1 Introduction

In distributed systems replication serves as a solution to maintaining high grade of availability, reliability and performance. This is due to several central aspects of replication. First and foremost, replication within the area of, for instance a distributed file system, will leave you with several copies of your data, securing the loss data, should one machine or node crash, hence enhancing availability. Further replication can help you spread copies out over a wide area, making sure that a nearby copy is always available for quicker and more stable connections to the data for different services and clients, this also leads to reduced network traffic, due to the fact that there is no single point of connections for communication with the data and also allowing for parallel connections to data stored on different nodes, further increasing the performance. Performance can be measured as the speed of accessing the data and includes both local and network based operations. It is vital to try and keep delay due to network speed as low as possible, since this will be the slowest part of any operation, due to the advanced speed of modern CPU’s. Two major challenges to solve are, consistency among copies and the single point of failure/rely on a stable node.

1.1 10 line Description

When attempting to create any framework for replication in a distributed environment there are several aspects to address, both in terms of expected functionality, but also in regards to development. Utilizing the strengths of each part in development, trying not to over-complicate the system, can be a challenge. Emerald is a programming language created for the purpose of making distribution and remote method invocation as easy as possible for the programmer, making distributed and local procedure calls as similar as possible, hence removing the need for the programmer to take special care and considerations when altering between local and distributed calls. This makes Emerald ideal for the purpose of creating such a prototype, allowing for the shift in work load from necessities, such as work on how distributed calls are made, to the main focus of the project, which is to create a “keep-alive” replication framework. This paper aims to describe the different aspects and areas of interests needed to create a framework for replication that can support its own existence without the need of any third part service or server to maintain the copy. Further, the prototype will aim to build upon the state-of-the-art within the area of replication to create a self sustained replication framework. The framework should use active-replication (master-master) as its base, making sure all copies have the functionality required to maintain the replication set as a whole. Each copy should also have the functionality to handle failures, by detecting, deciding and acting upon machine/node failures and crashes.

1.2 Motivation

Today’s computing world have shifted its focus to the aspect of cloud computing, and distribution is a necessity when dealing with cloud computing. Within the area of distribution there are two major obstacles to counter when creating a distributed system. First and foremost a high availability. This means making
sure that data is available, even in the event of a network outage and also not lost, should a machine or storage device crash completely. The other is enhancing performance, with concurrency between components at its core. Concurrency between components means different components can advance their procedures without waiting in queue for others to finish. Replication can ensure both of these, and is an everlasting importance in today's cloud computing, since no business would accept the loss of data in the event of an outage, because their systems now are services hosted elsewhere. Since replication is nothing new, there are several well documented approaches already out there, through what is known as design patterns. These patterns describe a reoccurring issue and provides the means to solve them. The problem with many of these replication schemes are that they rely on other services to handle the actual replication and maintaining the number i case of failures. In this paper I will therefore aim to create a framework which is able to survive completely on its own. hence tuning the existing patterns for replication to support self-replication, where each copy itself can detect failures and create new copies to replace the lost ones, and support all functionality without the use of hypervisors. I will argue that this will allow for greater fault tolerance. This paper will investigate that statement and also try and compare the possible pros and cons, in regards to availability and speed.

1.3 Problem Statement

Standard replication schemes rely on a third party hypervisor to handle failures and maintain copies. Is it possible to create a "keep-alive" framework that can support its own existence, without a big speed turnover?

1.3.1 The problem we have/and are trying to solve

Adding the functionality of detecting and handling failures within each copy, maintaining the number of copies and ensure strict consistency without the need for a third party service.

1.4 Goal

The goal of this paper is to use state of the art design patterns and build upon them to create a self-replicating live-forever framework for the programming language Emerald.

1.4.1 Platform to solve issue

The platform for solving this issue has several layers. Planetlab for distribution, Emerald language, Master-Master replication framework/design pattern, strict-consistency, Move Copy of Data pattern, Data Replication pattern, .

1.5 Approach

Create a general framework that can take any data type, replicate a number on demand copies, and detect failures and correct these within the framework, without the need of manual operations, clients or naming/grouping services. The main part and sole gola of the thesis is meant to be a programming solution,
and the additional written paper is meant to give insight into which technologies and pre-done work that creates the bases for the created framework, and also explain some relevant details into the actual implementation of the framework.

1.6 Evaluation

There are both pros and cons to the proposed framework for replication presented in this paper. In terms of the main goals of the framework perform as expected. Strict consistency between copies is ensured due to the required claiming of master, this have been tested by calling several operations, update, add and remove, without the occurrence of error. Keeping the copies alive when failures occur, hence ensuring high reliability and availability within the system using the framework is working as expected. Several trials both locally and on the testbed for distribution, PlanetLab, have proved that as long as there are available nodes the framework will regenerate lost copies. This have been tested by killing off one node every minute of as long as 15 minutes, without failure to regenerate the lost copies. However the somewhat slow communication allowed for a sequential system that always needs an accept from other copies before being able to do any other operations than the read command does incline a somewhat slow framework, this can be seen by the fact that from a node crashes to another copy being up the average time is about 53 seconds.

1.7 Result

Results show that the framework proposed are well suited for ensuring consistency, availability and reliability. However it have become clear that in terms of performance, the speed of a sequential system that heavily rely on messages between copies to be able to perform actions is significantly effected. This might put possible limitations on the solution, because the maximum number of copies tested to hold at any time have been set to 15 (in a distributed environment, 20 locally). It is possible that a larger number of nodes could significantly harm the performance, even though the claiming of a master position is capped at the number of copies divided by two (excluding the caller).

1.8 Outline

The following two sections of this paper will lay out the research and the already proven work done within the relevant areas connected to, which served as the base for building the creation of a framework for replication. The next sections will include design choices made when creating the framework and information about the actual implementation. Last the framework will be evaluated and a conclusion will be given.

2 Background

In this section I will present and evaluate the background material making up the base of this thesis. The areas described are components and designs the proposed framework is based on, and exclude some features that should be included when using the framework, e.g. relevant parts when using, but
not directly involved with the construction on the framework. These include architecture and security.

2.1 Distributed systems

At the very root of this framework lays the concept of distribution and distributed systems. A distributed system can most easily be view as a collection of computers working together and acting as one entity. By setting up a system in this matter, one can from first glance see that there can potentially be several benefits from such a design. These are best explained by looking at the main goals of a distributed system.

The first one is making resources accessible. What it entails is making accessing and sharing of resources easy.

The second is, distribution transparency. Distribution transparency means hiding the fact that resources are in fact distributed over several machines, and possible large distances. The end user of a system should not be made aware of, or have to deal with the fact that resources are spread out, but instead interact with them in the same manner as if they were locally accessible.

The third is openness. Openness addresses reuseability withing a distributed environment, meaning that the distributed system consists of components that are easily reused or integrated into other systems. As an example the replication framework proposed by this paper, is in fact such a component, if any distributed system needs replication the framework can easiliy be applied to any of them. Distributed systems are usually built up byseveral such components.

The fourth and last goal of a distributed system is scalability. Scalability can in turn be seperated into three different types, numberical, geographical and administrative [1].

The numerical scalability of a distributed systems refers to the number of nodes in the system. Meaning that a system that is numerically scalable can easily add more resources to the system without loss of performance.

Geographical scalability allows a system to expand geographically. Meaning that resources can be spread out accross the globe, which for instance is the case with the internet.

Administrative scalability addresses the capability of managing a system as it grows and expand across numerous independent organizations.

Pitfalls 16

[2, 3-16] Typical issues are concurrency between components, no global cloack for timestamps, failure/crashes of individual components

2.2 Design Patterns

What is a design pattern? Christopher Alexander describes what a design pattern when describing building patterns for buildings and towns, he says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [3, 12]

This of course applies to many general areas in common day to day life where you want a description of how to do, whatever you are supposed to do.

Erl, Cope and Naserour describes design patterns in the context of software
development as, “The simplest way to describe a pattern is that it provides a proven solution to a common problem individually documented in a consistent format and usually as part of a larger collection” [4, 9].

This means that a design pattern serves as a template for how to implement a solution to a problem. On the other hand, design patterns are not a piece of prewritten code that may be copy pasted into your work, and how to implement them may vary based on what system you are working on or what programming language you are using, but they will always give an idea of how to implement a solution. Erl, Cope and Naserour lists ten reasons for the usefulness of design patterns [4, 9–10].

Represent field-tested solutions to common design problems
Organize design intelligence into a standardized and easily referenced format
Are generally repeatable by most IT professionals involved with design
Can be used to ensure consistency in how systems are designed and built
Can become a basis for design standards
Are usually flexible and optional (and openly document the impact of their application and even suggest alternative approaches)
Can be used as educational aids by documenting specific aspects of system design (regardless of whether they are applied)
Can sometimes be applied prior and subsequent to the implementation of a system
Can be supported via the application of the design patterns that are part of the same collection
Enrich the vocabulary of a given IT field because each pattern are given a meaningful name

The general format of design patterns as described by Vlissides, Helm, Johnson and Gamma in their book Design patterns: Elements of reusable object-oriented software are, the name of the pattern, which aim is to shortly describe the pattern. The problem, describing when and for what this pattern is useful. The solution, describing the different elements and their relations and lastly the consequences of applying the certain pattern [3, 12].

So, plainly speaking, a design pattern can be viewed in comparison with a cooking recipe. It has a descriptive name (i.e “Coq au vin”, Rooster in whine), it describes what you want to create (the problem) and it describes how to do it (the solution).

It is quite easy to comprehend both the comfort and importance of a design pattern as it makes implementation easier for developers and reduces both cost and chance for error for businesses. The question is then if these common problems are occurring often enough for there to be any motivation for studying and developing these patterns. In this case we are specifically talking about design patterns for cloud computing. One study done by CompTIA has shown how much cloud computing is used by businesses in the U.S.

“Both channel firms and end users are experiencing disruption as they move towards cloud-enabled infrastructure and solutions. Forty-one percent of channel firms cite cloud as a catalyst for new directions requiring business transformation. From an end user perspective, over 90% of U.S. firms claim some form of cloud computing, and over 60% of those firms stating that cloud components represent at least a third of their overall IT architecture” [5]

This goes to show the huge relevance cloud computing has today, and so shows the importance of design patterns for often reoccurring problems.
2.3 Replication

As described in a previous sub section, availability and reliability has grown into one of the most important parts of modern distributed systems. This is where replication serves a purpose. Replication will provide several layers of extra security in terms of both availability and reliability. If you for instance are creating a distributed file system, replication will allow for copies of the same file to be distributed over several servers, this grants several benefits to the user. First and foremost, in the event of a machine failure, backups are stored elsewhere and no data is lost. Further, replication may enhance both performance and uptime. If a centralized server containing your files should for some reason be unavailable, you can access your files on one of the other servers. In terms of performance, a server closer to your location, and with a more stable connection, will allow for faster interaction with the resource.

There are several aspects to take into consideration when creating a framework for replication, in terms of services that should be provided, but are invisible or transparent to the user.

2.3.1 Desired functionality

Scalability - Any replication framework should support a large number of copies, without disruption of service.

Transparency: The fact that a file is replicated should not be visible to any clients, nor should their location. This means that as far as clients are concerned, there is only the one resource they are communicating with.

Performance - The amount of time needed to process a client request should not be significantly hurt by the fact that the system is a distributed system.

Availability and reliability - The main point of replication is to ensure availability and reliability. Should some nodes crash others should be able to process client requests, and in case of crashes data should not be completely lost, and should be regained through copies.

Concurrency and consistency - The framework should be able to deal with several requests (read commands) at the same time, but also ensure that multiple simultaneous update commands does not create errors within the resources held by the replication system.

Design patterns are proven templates for re-occurring issues, often smart to use when dealing with difficult issues. Hence, in the next section, proved schemes for replication will be presented as a basis for the work done in this paper.

2.4 Emerald

Emerald is a programming language based on a single object model. This means that the entire language and all its content, from integers to entire file systems, are of the same base type, namely objects. What Emerald aims to do is to use this single object model within a distributed environment. Each object is built up from three obligatory parts, one optional part and a set of attributes. All of these serve their purpose, but I will only unfold on those serving a big role within the scope of this study [6, 14–15].

Each object consists of a name. This name serves as an object's unique
identifier within a network.

A representation part, which include the reference to other object. In Emerald these references does not care about the location of the object, and the reference will be the same whether the object referenced is located at the same or a different node than the object containing the reference. For these object references Emerald supports a special function called "attached" the point of this function is to solve the grouping problem. This means that all variables/references declared within another object will be group together with that object in terms of object mobility, and move to the same location, should the object containing the reference be moved [7, 44]. Not including the attached function will leave object referenced at whatever location it is currently at. This distinction i particularly useful in the framework purposed in this paper, because it will allow all data referenced within a replication to easily be moved with it, while leaving the list of references to other replications where they are.

Several operations, which are the functionality provided by the object, allowing for data manipulation or retrieval. These functions can be tagged with "export", which is equivalent to the "public" tag, in for instance the java programming language, making the operation or function invokable for outsside objects.

An optional process part, which will be executed after initialization of the object. This process will execute in a separate thread. This will allow for operations within the object to be called and executed in parallel with the job being done by the process. This simple, but powerful idea is the basis of why Emerald is well suited for the framework proposed in this paper. Allowing the replications to be accessable through calls to its operations to wither update or retrieve data, while the process will be able to run simultaneously and keep the node cluster.

The attribute, location, is available in to each object in Emerald. The location attribute will contain information on which node the object is currently located on. This is the second built-in property that makes Emerald ideal for the "Live-Forever" framework. Emerald has a language construct that can be called to attempt and retrieve this location [7, 35].

These represent the way objects a represented in Emerald. The reason is to support distribution, with semantics built into the language to allow for a simpler way to deal with object mobility for the programmer. Additionally Emerald supports a special operation for failure handling. The failure handler may be attached to any operation and will be invoked by a number of events not allowed within the langauge. Thus executing the program statement within the handler. This will allow the programer to take special action in the event of failures [7, 23-24].

A specialized sub-function of the failure handler is the unavailable handler. It works exactly with as the failure handler, but instead of being invoked by a number of failures, it instead interact with the Emerald built-in location, and an object is interpreted as unavailable if it cannot be located at any of the nodes now to be available by the system, hence executing the unavailable clause [7, 23]. This means that together with the representation part, which consists of object references and the location attribute, one can easily determine if one of the nodes have become unabailable, and execute failover or other appropriate fault handling. For the suggested framework in this paper, this is the key aspect of the keep-alive functionality of the replication framework.
2.5 Planetlab

PlanetLab presentation and facts

3 State of The Art

3.1 Survey of Erl, Cope, and Naserpour, Design Patterns for Cloud Computing

This part will go through some of the different design patterns from the catalog (Cloud Computing Design Patterns, Erl, Cope and Naserour). The book distinguishes normal patterns from what they call compound patterns.

“A compound pattern is a coarse-grained pattern comprised of a set of finer-grained patterns.”[4]

This part will take a close look at each of these individual collections of design patterns presented in Cloud Computing Design Patterns by Erl, Cope and Naserour. The patterns not listed as compound, will only be view shortly, by listing their applicable area and their main goals. For the compound patterns, I will look a bit deeper and show a few examples of the structure and content of the patterns as well as their specific focus area.

“The compound patterns covered in this book are classified as such because they relate to recognizable models, environments, and technology-sets in the contemporary cloud computing industry.” [5]

Furthermore, it is important to clarify how patterns can be combined into compounds. A compound pattern can represent a set of patterns that are applied together to a particular program or implementation in order to establish a specific set of design characteristics. This would be referred to as joint application. Alternatively, the member patterns that comprise a compound pattern can represent a set of related features provided by a particular program or environment. In this case, a coexistent application of patterns establishes a “solution environment” that may be realized by a combination of tools and technologies.


3.2 General Design Pattern Areas

3.2.1 Sharing, scaling and elasticity Patterns

This collection of design patterns focuses on providing solutions for maximizing the potential usage of available IT resources in response to unpredictable usage requirements across multiple cloud consumers. Shared Resources, Dynamic Data Normalization, Memory Over-Committing, and NIC Teaming directly enable and support the realization of multitenancy over large pooled resources, whereas ubiquitous cloud consumer access is enabled through the application of the Broad Access pattern. The majority of patterns in this chapter directly or indirectly enable the elasticity characteristic of cloud computing to support the automated ability of a cloud to transparently scale IT resources, as required in response to runtime.[6]
3.2.2 Reliability, Resiliency and Recovery Patterns

Contingency planning efforts for continuity of operations and disaster recovery are concerned with designing and implementing cloud architectures that provide runtime reliability, operational resiliency, and automated recovery when interruptions are encountered, regardless of origin. The patterns in this chapter address different aspects of these requirements. Starting with foundational patterns, such as Resource Pooling, Resource Reservation, Hypervisor Clustering, and Redundant Storage, which address basic failover and availability demands, the chapter continues with more specialized and complex patterns, such as Dynamic Failure Detection and Recovery and Zero Downtime, which establish resilient cloud architectures that act as pillars for enterprise cloud solutions. It is also worth noting that this set of patterns establishes and contributes to the availability leg of the security triad of confidentiality, integrity, and availability and is further complemented by several cloud security patterns in maximizing the reliability and resiliency potential by protecting against attacks that can compromise the availability of an organization’s cloud-hosted IT resources.[7]

3.2.3 Data Management and Storage Device Patterns

The fundamental cloud computing model for enabling ubiquitous, on-demand, scalable network access to shared pools of configurable IT resources typically demands for the existence of and access to vast amounts of inexpensive storage that itself must be highly flexible, scalable, and configurable. As with other members of typical cloud architectures, cloud storage devices must have the ability to be rapidly provisioned and release storage resources and large amounts of data with minimal management effort or cloud provider interaction. This set of patterns addresses key issues that pertain to common challenges and optimization requirements when configuring and managing cloud-based storage devices and the datasets they store.[8]

3.2.4 Virtual Server and Hypervisor Connectivity and Management Patterns

Depending on which cloud delivery model a cloud resource administrator is working with, the extent to which hypervisor and virtual server configuration can be controlled can vary significantly. This set of patterns focuses on hypervisors and virtual servers. Through the hypervisor, the cloud provider delivers interfaces to networking features, such as virtual network switches, that cloud consumers may use to configure custom virtual networks within the provider’s infrastructure. This structure varies from SaaS and PaaS environments where much of the configuration and maintenance of IT resources is handled transparently for cloud consumers. In an IaaS environment, on the other hand, cloud consumers will typically maintain complete control over the operation of the guest operating system in each virtual server, and all software layers above it.[9]

3.2.5 Monitoring, Provisioning and Administration Patterns

A primary goal for cloud providers is to deliver not only affordable but easy-to-use resources for organizational computing requirements. The patterns in this
chapter are primarily provided in support of that goal. Monitoring and situational awareness are enabled by patterns such as Usage Monitoring (285), Pay-as-You-Go (288), and Realtime Resource Availability (292) supporting cloud consumers with critical SLA assessment and verification capabilities. Automated Administration (310) is an example of a solution to how the cloud provider can automate provisioning requirements met on-demand. This pattern can be combined with Centralized Remote Administration (315) and Rapid Provisioning (295) in support of Platform Provisioning (301) to relieve cloud consumers of the burden of implementing the underlying infrastructure of their cloud environments.[10]

3.2.6 Cloud Service and Storage Security Patterns

The foundations of security consist of supplying confidentiality, integrity, and availability of services. The patterns contained in this chapter provide important solutions for these key security requirements and aggressively approach threat mitigation by enabling security features suitable for cloud service and cloud storage architectures that are part of potentially volatile cloud environments.[11]

3.2.7 Network Security, Identity and Access Management and Trust Assurance Patterns

This chapter acts as a continuation of the previous in that it continues to focus on cloud security architectures and solutions. This time, the areas covered are network security with an emphasis on external connectivity issues, the management of identities and access levels, and patterns that are applied to establish trust boundaries and characteristics.[12]

3.3 Compound Design Patterns

A compound design pattern is most easily described as, a combination of several singular design patterns. A compound can then be viewed in the same way a normal design pattern, but instead often lead to design solutions for a bit larger problems, by combining patters for many of the smaller parts. The compound design patterns to be researched in this paper are all from, Cloud Computing Design Patterns by Erl, Cope and Naserpour. Hence, these compound patterns are different combinations of the design patterns discussed in the previous section.

As previously stated we need to combine several design patterns to create a compound pattern. According to Eral, Cope and Naserpour, there are two different ways of making these combinations. “When we discuss the notion of combining patterns into compounds, it is important to clarify how patterns can be combined. A compound pattern can represent a set of patterns that are applied together to a particular program or implementation in order to establish a specific set of design characteristics. This would be referred to as joint application. The compound patterns with patterns that are jointly applied are: • Cloud Bursting (492) • Burst Out to Private Cloud (493) • Burst Out to Public Cloud (496) • Burst In (499) • Secure Burst Out to Private Cloud/Public Cloud (501) • Cloud Balancing (503) Alternatively, the patterns that comprise a compound pattern can represent a set of related features provided
by a particular program or environment. In this case, a coexistent application of patterns establishes a “solution environment” that may be realized by a combination of tools and technologies.” [13]

3.3.1 Common Compound Patterns

This section will discuss one example of a compound design pattern, namely the “Private Cloud” pattern, and the buildup of one of the patterns it contains. The Private Cloud pattern consists of fourteen core patterns and three extension patterns. Extension patterns are optional additions to the collection to enhance functions or features of the given environment. For our example, we will look closer at one of the core patterns that this compound pattern consists of, the Automated Administration pattern. The Automated Administration pattern is meant to answer the question, “How can common administrative tasks be carried out consistently and automatically in response to pre-defined events?” [14] The pattern (as well as any pattern covered by this catalog) is built up of four parts. A Problem, a solution, an application and mechanisms. The problem states the issue at hand, or what you would like the pattern to solve (as the question in the start of this example). The solution part lists how we answer this question. The application part stats what application is needed to answer the question and the mechanisms part lists the actual parts needed for the application.

3.4 Replication Patterns

Within the area of replication there are two main ideas for proposed solutions, these are the passive-replication and active-replication schemes. This section will look closes at both of these schemes as described in [8, 618–630].

3.4.1 Passive-replication

In the passive-replication, or primary-copy replication, scheme for replication there is a master-slave relationship between replicas. This means that at any time, there will only be one master (primary) replica, and the rest will be slaves. The idea of the passive-replication scheme is that only the primary-copy will ever execute operations, and communication with a client is done through this object. The primary-copy will then send a copy of the updated data to all the backups within the system. The execution sequence in event of a client request is as follows:

Request: The front-end issues a request to the primary-copy, each of these requests will contain a unique request identifier.

Coordination: The primary-copy will process each request in the order of which they are received. The unique identifier will then be used to check whether the request have already been processed, and if so the response will be re-sent to the client, else the request will be processed.

Execution: The primary copy will execute the request and store the response.

Agreement: If the client request was an update. The primary will then forward the updated state to all backup copies, whom in turn respond with an acknowledgement when their state have been updated.

Response: The primary copy will return a response to the front-end, which
then will return the response to the client. The drawback of the passive-replication scheme is that it relies heavily on the uptime of the primary copy. Therefore mechanisms need to be in place to ensure a failover in the case of a primary-copy crash. One of the backups will need to be made the new primary-copy. The special care that needs to be taken here is that only one backup is chosen for the new primary-copy and that all replications agree upon which requests that had already been performed at the point of the failover.

3.4.2 Active-replication

The other branch of replication is the active-replication scheme. In active replication each replica serves a similar role, in comparison to the passive-replication scheme, this solution will have all replicas as the primary-copy. The difference is that in an active-replication scheme each copy processes the request independently, even though they are identical, and reply. These are organized as groups and the front-end communicate using a multicast for message passing to all replications.

Request: As with the passive-replication, requests are marked with a unique identifier, but the message is multicast to all replications within a group instead of only communicating with the primary. Only one request will be handled at a time, since no new request will be issued before receiving a response.

Coordination: With the group communication system, all requests are delivered to all replications in the same order.

Execution: Every replication executes the request independently. Because these are state machines and requests are delivered in the same order, the requests will be executed identically.

Agreement: There is no agreement phase. This is because of the multicast communication system, ensuring all requests are similarly sent to all replications.

Response: Every copy sends a response to the front-end. Depending on the specified failure assumptions and wanted functionality, the front-end can respond based on the number of replies collected or use the first received and discard the rest.

3.4.3 Self-replication

The idea of self-replication is not a normal one as far as replication for fault tolerance goes. In this paper the term "self-replication" refers to the fact that each replica within the replicaton framework should have the possibility to both detect failures of other nodes and respond by creating a new copy on an available node, if any.

4 Design Choices

The framework proposed in this paper differ from, but are also built upon the two main schemes for replication introduced in sections 3.4.1 and 3.4.2. Since the whole idea of the framework is a self-sustained framework for replication, I
argue that the core needs to be of the active-replication nature. By this I mean, each copy should contain all functionality, both in order of manipulation data and being able to detect crashes. This will keep the framework able to not rely on any other part to function, as with the active-replication scheme, and also being able to reproduce to other available nodes in the event of a node failure. However, in contrast to the active-replication scheme, which relies on a multicast communication system at the front-end side, a self-sustained framework should keep as much as the requirements to fulfill its purpose as possible. Here is where the framework retrieves inspiration from the passive-replication scheme. Each copy will be treated exactly like both the primary and backup copies of the passive-replication scheme. Requests will be done to either copy, which in turn will attempt to claim the master position (claim the right of being the primary), and then execute the request as with the passive-replication scheme. Following the same syntax as described in [8], the workflow of the framework will be as follows.

Request: A request is issued from the front-end to any replication.

Coordination: The replica receiving the request will call on all other replicas in an attempt to claim the master position. The required positive responses is set to the number of nodes divided by two (excluding itself). Because the system is set up in such a way that a replica that have already given permission to one node cannot give it to another before released, no other replication can be able to receive that number of positive responses at the same time, thus saving computing time.

Execution: The now proclaimed master (primary copy) will execute the request by calling the update function on all other replication, before updating its own state.

Agreement: As each node will update their value when the selected master makes the call, there is no need for an agreement.

Response: The selected master will return a response to the front-end, which then will return the response to the client.

5 Implementation

In the following section I will elaborate on my implementation of the live-forever framework, proposed to solve the problem statement. Digging deeper into similarities and differences from the general design choices, described in the previous section.

In distributes systems, replication is the means to ensure availability, performance and security for your data. Replication itself will help keeping your data safe in the event of machine crashes or failures, they will let you access your data with greater speed, by having several copies spread over a, possibly, wide area, making access to the closest, or most stable node, quicker than working with a dataset far away on an unstable network machine. A general replication framework can work in a highly heterogeneous environment, in other words it is compatible with any type of data-sets or types, making for a one-fits-all solution, which serves the purpose of modern cloud computing perfectly.
5.1 General Design

There are several difficulties when it comes to implementing a platform for replication. First and foremost consistency throughout the replications are key. The value stored in each separate replication, for a certain data-set, needs to follow the same mapping, in a way that each replication contains the same value, both after initialization and after update calls, hence securing strict consistency, throughout the system and also try and keep the computation time as low as possible. Further more, there is also the issue of handling failures, throughout all nodes in the system, minimizing the chance for each data-set to be completely lost.

This design aims to use the active-replication scheme as base for the framework, and add additional layers to it, to create an active replication framework, which will self-replication in case of machine crashes, and keep the total replication count to the desired number. The framework has two main parts, the first, a message operation, used to handle all distributed calls throughout the system, and all other operations are called locally for each node. The other is a process, which contains a "listening" loop, which will at all times attempt to keep the actual and wanted number of replications at the same level.

The framework is built for the sole purpose of replicating, detecting machine crashes and taking simple commands from a client, adding, removing and updating. It is not built as a full system containing the needs to process and store several different replications at the same time, but serves as a framework that may be used to create such a system.
5.2 Initialization

Initialization of a data-set using the framework is done by creating one object. This will typically be done through a client. The initial object will take several parameters, a reference array, where all future replications will be referenced. The array is sent by the client and not created within the initial object to make the reference array easily accessible by the client. The value you want to be stored with the replications. Here all data types are allowed, through the Emerald type Any. In order to use the framework in a complete system, one would need to add several conformity tests and "view as" operations in order to use these values when extracted, each one tailored after the specific system or data-set in question. The number of wanted replications, since the framework is "self-replicating", it will need to know how many copies of a certain data-set is wanted, so it can create replications of itself. It needs a destination node, each object will move itself after initialization, hence a reference to the destination node is needed for this to be possible. The idea around this design is for the listening process not to be started before the move has been complete, which would have been the case if each object were moved after the complete initialization. For the initial object this is typically the location of the client itself, making the chance for the referenced node to be up, a certainty.

A timestamp, this is not currently in use within the framework, the initial implementation attempted to use timestamp to ensure consistent throughout all node, but the final implementation, using a claim master position operation rendered this superficial. However the timestamp was kept, because knowing the updated time, or version date of your data might prove useful when using this framework to create complete systems. The last parameter is a pseudo-random number generator. This is used in an attempt to stop loops when claiming the master position. If a replication is unable to claim the master position the generator is used to delay the node for a given amount of time, letting other nodes try an claim the master position, without interference.

Post creation of an initial object the framework will itself create additional replications, up until either the selected number have been met or until there is no more available machines on the network.

5.3 Self Replication

As mentioned in the general design section, the framework uses "self-replication". By "self-replication" I mean, each node have the functionality to make another copy and also the functionality to detect when another node becomes unavailable, due to machine or network crashes. This is done through the listening loop. This loop will live for as long as the node is active. For the first part the loop will check if the wanted replication number meets the actual replication number, and if not, will attempt to create a new replication at an available node. The other part, will continuously try and locate all the referenced replications (find their node), if this fails the error handling specific to Emerald will execute and deal with the fault handling.
Workflow of the replication framework

Figure 2: Unavailable Node Detection Workflow. The “create new copy” in this figure will also utilize the claim master steps as shown in 4 on page 17.

5.4 Failures

Emerald uses a language integrated fault handling, called failures. In this framework I only use the sub part of failure called unavailable. Any operation can have an unavailable section, which will execute if one of the objects used or accessed within the operation is unavailable. The fact that Emerald have this built-in error handling makes handling node crashes much easier than it would otherwise be. As mentioned above in the Self replication section, a process in the framework will attempt to locate all the replications referenced in the reference array. If unable to locate one of these, the unavailable clause will automatically kick in, and allow for error handling. In case of an unavailable clause being executed the framework will remove the object in question from the reference array and then call itself again, making sure that, if several node should crash at the same time, all of these are removed from the reference array before allowing for creation of a new replication. When all unavailable nodes have been removed the fault handler have done its job and the listening loop will once again recognize that the actual number of replications are lower than the target number and attempt to create a new replication on an available node.
5.4.1 Temporary Unavailable vs Actual Crashes

There are several reasons why a node could become unavailable, either it be a machine crash, a network blackout or just a small window of unavailability due to latency (which would mean the node have not really crashed). The system will handle all these in the same manner. Removing the reference to the object from the reference array and attempt to find a new node and create a new replication on it. This could potentially lead to the framework removing a reference to an object then being able to locate the same machine when attempt to add a new object and create a new object on that node. This is a shortcoming of the system that could lead to several redundant procedure calls, and could potentially also lead to several objects on the same machine. From within the system this would not be noticeable, since only the new object would be referenced, but could potentially lead to a machine with a slow network connection wasting space on dead objects, since there is no garbage collection.

5.5 Master Position/Strict consistency

As discussed in the opening of the implementation section, strict consistency is important in replication with a distributed file system. The way the framework ensure consistency is by the use of a three-part variable system, the master position, the contender position and the waiting position. These are meant to make sure only one node can be able to make changes at any given moment, through being the master. The way they work together is that, each attempt to claim the master position will first check if the waiting variable is set, if not, the nodes contender variable will be set, the node is now a contender for the master position and is allowed try and claim it. This is done by sending messages to all other nodes, which will reply either yes, if the node in question is either the master or a contender, or no otherwise. If a total number of actual number/2 is met, the contender becomes the master. At this point there have been added an additional test, testing the number of nodes, and if the amount is low the number of yes’es required will be the total number instead of a half. This have been implemented due to issues with the testing platform, PlanetLab, which crashed and were unavailable for the last few months of development. The point here is that, local testing, through simulations of several nodes on a local computer, was too quick for only test for half the number, leaving more than one node to be allowed the master position, hence crashing the system. The test, which required a certain amount of nodes before using the desired function amount/2, is a work around for the planetlab crash, and was not planned to be part of the final implementation, but due to the persisting issues on planetlab, I decided to keep it within the program, to allow for local testing, if need be. The point of only need a yes from half the nodes is to shorten the execution time for larges systems, with a large number of nodes, since every node with a yes from self and half of the amount of others, will with certainty have yes from more than what is possible for an other contender, this is a lean way of cutting potential run time of this part by half, since the framework is set up is a sequential way. A Yes from any node will put that node in a waiting position. If a node does not receive the required number of yes, the pseudo-random number generator is used to halt the node from attempting another claim for a given amount of time, allowing other nodes at finish their attempt, before executing
again. This is used to ensure that there are not too many contesters that would answer no, hence making it impossible to claim the master position. When the master position have been successfully claimed, the node is free to make the required changes, either it be an operational call from a client or the self-replication.

5.6 Client Operations

All operations that make changes will first call the claim master operation. Once a node have become the master, it will be allowed to add a new replication due to the listening loop, or execute several client call operations.

5.6.1 Add Replication

The add replication operation is a client supported operation where a node is asked to add another replication to the data-set, this can be done in two ways. The first one is the add operation without any parameter, this will make the master call all other nodes to increase the value of the wanted number of replications by 1, which again will lead the listening loop to create a new replication on an available node. The other way is for a call to the add operation with a destination node as parameter. The master will then check if the destination node is available and also check if there is a replication there already. If not the master will go ahead and create a new replication on the destination node, adding it to the reference set.

5.6.2 Data updates

The update operation requires an input parameter. The master will then take a timestamp and call to all other replications to update their values and timestamp variable, before changing the value of its own variables. The framework does not allow for any protection for conformity of data, hence if the framework is used to store Strings and the update is called with Integer as input, the entire system will still update its value to the received integer value. This is due to the frameworks goal of being generalized, and when used to create different systems, such security could be implemented to fit the needs of the user.

5.6.3 Replication removal

The remove replication operation requires a node as input. And will then go through all the referenced data to attempt to locate the destination specified. If found this node will be called to shut down, and then removed from the reference array.

5.6.4 Get data

This is the only client supported function that does not require the claiming of the master position. Since no changes are done, allowing several clients to connect and retrieve the data at the same time poses little risk to the reliability and consistency of the framework. The function only returns the stored data within the connected replication. This would in a distributed system mean that
several clients could retrieve data in parallel from different nodes, without any queue time.

5.7 Code Overview

5.7.1 Ensuring strict consistency

The pseudo-code provided here is the code used to claim the master (primary) position. The idea is that since all changes require a copy to be the master the entire framework is bound by strict consistency. In your code there is also added a pseudo-random number generator, used to stall a node from claiming the master position again. This is used in case of collision. Normally a collision should now occur, but local testing has revealed that the speed of each node's request is too high, and the variable set to hold other copies from claiming the master too slow in comparison. Thus leaving a possibility of an infinite loop of copies trying to claim the master position. In relevant cases this is meant for a distributed
5.7.2 Checking for crashes

This part of the code is the loop ran inside each copy’s process. The loop will check if the desired number, possible number and actual number are in fact similar, and will take action if they are not and if it is able. If the numbers correspond the loop will call the method isUp() over and over, which check for crashes.

This is the operation allowing each copy to check for failures in other copies. The operation utilizes the built-in Emerald language functionality of location and unavailable, as described in section ???. The idea is that if any node’s location should become unavailable the unavailable clause will kick in, removing the reference to the copy on that node and allow the automatic creation of a new copy from the main process loop described above. This is possible, because of the sequential nature of the system.

6 Evaluation

The main goal of this paper was to create a framework that were able to keep itself alive. As far as this functionality goes the framework, using Emeraldrd special build in operations, delivers really good, and tests have shown that as long as there are available nodes a new copy will be created in case of a failure. As all frameworks for replications, other features are also required. In terms of strict consistency, the framework ensures this using an operation to calim the
<table>
<thead>
<tr>
<th>Number</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>61:965024</td>
</tr>
<tr>
<td>2</td>
<td>49:136930</td>
</tr>
<tr>
<td>3</td>
<td>41:231529</td>
</tr>
<tr>
<td>4</td>
<td>57:192421</td>
</tr>
<tr>
<td>5</td>
<td>62:215302</td>
</tr>
<tr>
<td>6</td>
<td>63:012531</td>
</tr>
<tr>
<td>7</td>
<td>44:102592</td>
</tr>
<tr>
<td>8</td>
<td>45:726124</td>
</tr>
<tr>
<td>9</td>
<td>56:351249</td>
</tr>
<tr>
<td>10</td>
<td>51:290041</td>
</tr>
<tr>
<td>Avg</td>
<td>53:222374</td>
</tr>
</tbody>
</table>

Table 1: Time Result: List of showing the time for the system to regenerate a copy after a crash. Shown in s and ms

master position. This operation runs in a sequential manner and successfully hinder several write requests to be executed at once. However, many messages having to be sent across a network is expensive, and the framework is somewhat hurt by this, performance wise. Tests show that with a total of five copies, a kill and regenerate sequence will on average take 53 seconds. This might arguably be tolerable, if write requests are rare. If the increase of copies significantly increase the time needed for write requests, this might not be the best solution in that particualt case, but this have not been tested. I would however argue that the decrease of performance would not be that significant due to most time used by the system is not used here. The tests shown below shows a difference in about 22 seconds, which is a significant error margin, even for a small number of nodes.

The table below shows a list of ten trials, from killing off a node until a new copy is baack up. These tests were run in an environment where only 5 active copies were held.

7 Conclusion

In this paper I have presented a keep alive replication framework for the program language Emerald. It supports strict consistency through the use of sequential consistency. Since the framework is base of off the master-master replication framework all copies/nodes contains all functionallity, and this together with the "Dynamic Failure Detection and Recovery" pattern from Erl, Cope and Naserpour, the system will be kept alive as long as atleast one node survives, since each node also carries this functionality. The framework allows for a number of services/clients to connect and retrienve the date from a single node, with the need of communication within the system, thus increasing the performance for all read requests. On the other hand, since the framework is set up using sequential consistency together with the need for a node to be able to claim the master position before any write can be done there are bound to be some performance penalty for all write commands. I would argue that this is a tolerable trade off to be able to keep strict consistency between components, due
to the fact that in general there is a much larger demand for read commands opposed to write commands. Most systems, when set up, will not usually change the amount of copies on demand, making these write commands a rarity. The update command is the only write command that in some cases might be in use regularly. If the framework should be applied on a distributed file system, where files regularly are checked out, edited, then checked back in the update command could see the light of day on a regular basis. However in a system like that strict consistency is high on demand, making for a possible fair trade off. Failures should also be rare, making the expensive strict consistency through the claiming of the master position a smaller issue. However the framework have only been tested in a distributed environment consisting of few nodes, and the speed might be significantly reduced by adding more, even though you only need to response from half of the nodes.

7.1 Limitations

The framework have several limitations, when it comes to both scope and functionality. In terms of scope, the framework is set up to be generalized, being able to work with all, but no specific data. Hence the framework takes no care in what data that are stored, nor updated. If you use the framework to store document objects, but then call the updated command with the input as a single integer, the whole framework would change their values to that integer and all document objects would be lost, this is due to the fact that the scope is not set to any specific data. Further the framework is setup as a keep alive, master-master replication scheme, thus maintaining a reference to all copies within each individual copy, and the idea is that the replication set should self maintain its existence without a single point of failure. As of now however there is no way for clients to retrieve the reference list without creating the reference on a single location when taking the framework into use, I will discuss this further under 7.2 Future Work on page 25. The fact that the framework at max, only have been tested on 15 nodes in a distributed environment, might put additional limitations on the framework, due to the possibility of even larger loss of performance.

7.2 Future Work

There are one aspect of the current framework that might be up for review in future work. that is the fact that the update command will take a lot of time. If the framework is to be used for a distributed file system, one might want to look at additional possibilities of increasing performance in that command. Other future work is related to making the framework as a whole, standalone, middleware. By this i mean figuring out a clever way for client reference retrieval. If one were able to figure out a way without the use of a third party application the keep-alive framework would be able to serve as its own entity and could be applied or integrated, without the use of additional services.
List of Tables

1 Time Result: List of showing the time for the system to regenerate a copy after a crash. Shown in s and ms . . . . . . . . . . . . 24
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Client Request Workflow</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>Unavailable Node Detection Workflow. The &quot;create new copy&quot; in this figure will also utilize the claim master steps as shown in 4 on page 17.</td>
<td>19</td>
</tr>
</tbody>
</table>
Bibliography


