Welcoming the Unknown

Dynamically Merging Distributed Programs

Gaute Svanes Lunde
Olav Johan Myklestad Ekblom

Thesis submitted for the degree of
Master in Software
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2021

Welcoming the Unknown

Dynamically Merging Distributed Programs

Gaute Svanes Lunde
Olav Johan Myklestad Ekblom

© 2021 Gaute Svanes Lunde , Olav Johan Myklestad Ekblom

Welcoming the Unknown

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Distributed object oriented programming languages, such as Emerald, facilitate the creation of distributed systems by making the migration and invocation of remote objects part of the language. Although objects may travel freely within their respective system, they have no way of discovering, interacting with or migrating to other systems.

The WELCOME language mechanism introduced in this thesis provides support for interaction *between* distributed systems, and is implemented as an extension to the Emerald programming language. The interaction is achieved through syntactic support at the language level, which allows separate systems to discover each other and exchange object references directly during runtime.

Contents

I Introduction and Background 1

1 Introduction 2
1.1 Motivation . . . 2
1.2 Problem Statement . . . 3
1.3 Goal . . . 3
1.4 Approach . . . 3
1.5 Work Done . . . 3
1.6 Evaluation Criteria . . . 4
1.7 Results . . . 4
1.8 Conclusion . . . 4
1.9 Contribution . . . 4
1.10 Limitations . . . 4
1.11 Thesis Outline . . . 5

2 Background 6
2.1 Distributed Systems . . . 6
2.2 Creating a Compiler . . . 7
2.3 Virtual Machines . . . 8
2.4 PlanetLab . . . 8
2.4.1 Using PlanetLab . . . 8
2.4.2 Emerald and PlanetLab . . . 9
2.4.3 Security . . . 9

3 The Emerald Programming Language 10
3.1 Introduction . . . 10
3.1.1 Terminology . . . 10
3.2 Language Background . . . 10
3.3 Object Mobility . . . 11
3.3.1 Attached Objects . . . 12
3.4 Object Creation . . . 12
3.5 Types and Conformity . . . 13
3.6 Object Structure . . . 13
3.7 Distribution and Location . . . 14
3.8 Summary . . . 16

II The Problem 17

4 Inter-Program Communication 18
4.1 The Limits of a Program . . . 18
4.2 Inter-Program Referencing . . . 19
4.3 System Extensibility . . . 19
4.4 Summary . . . 20

5 The Problem of the Unknown 22
5.1 Connecting Nodes . . . 22
5.2 Connecting to Unknown Services . . . 22
5.3 Summary . . . 24

III The Solution 25

6 WELCOME - A Multilevel Solution 26
6.1 Introduction . . . 26
6.1.1 New Terminology . . . 26
6.2 Merging Object Graphs . . . 27
6.2.1 Exploring Semantics . . . 27
6.2.2 Syntax of the `welcome` Expression . . . 29
6.2.3 Welcomable Objects . . . 32
6.3 Merging Node Graphs . . . 33
6.3.1 Extending the `Node` Interface . . . 34
6.3.2 Consequences of Merging . . . 35
6.3.3 Dynamically Merging Node Graphs Using Identity . . . 37
6.4 Complications of Merging . . . 37
6.4.1 Best Effort Merging . . . 40
6.4.2 The Node Synchronization Algorithm . . . 40
6.5 Future Work . . . 42
6.6 Summary . . . 42

7 Implementation 44
7.1 Introduction . . . 44
7.2 Getting to a Bootstrapping Compiler . . . 44
7.2.1 Understanding the source code . . . 45
7.3 Implementing the `welcome` Expression . . . 45
7.3.1 Changing the compiler . . . 45
7.3.2 Changing the Emerald Virtual Machine . . . 47
7.3.3 Implementing Welcomable Objects . . . 49
7.4 Implementing Node Graph Merging . . . 49
7.4.1 The `mergeWith` Operation . . . 49
7.4.2 Node Discovery . . . 50
7.4.3 Extending the Emerald Event System . . . 51
7.4.4 The Emissary Move Request . . . 52
7.4.5 Silent Node Connections . . . 54
7.4.6 3rd Party Emissary Move . . . 55
7.4.7 Discovered Node References . . . 56

7.4.8 Modified Cascading Search Algorithm . . . 56
7.5 Limitations . . . 57

8 Evaluation and Results **58**
8.1 Evaluation Criteria . . . 58
8.2 Results & Evaluation . . . 58
8.2.1 The Welcome Expression . . . 59
8.2.2 Move Welcoming Processes . . . 61
8.2.3 Node Discovery . . . 62
8.2.4 The `mergeWith` Operation . . . 63
8.2.5 Merge by Emissary Objects . . . 65
8.2.6 Third Party Emissary Move . . . 66
8.3 A Discussion of Novelty . . . 67
8.4 Summary . . . 68

IV Conclusion **69**

9 Conclusion **70**
9.1 Project Outcome . . . 70
9.1.1 Design . . . 70
9.1.2 Prototype . . . 71
9.1.3 Evaluation . . . 71
9.2 Limitations . . . 71

Part I

Introduction and Background

Chapter 1

Introduction

In this project, we aim to enable distributed disjoint programs to become acquainted by introducing a mechanism for exchanging object references between them. We will be using Emerald, an object oriented (OO) programming language especially designed for creating distributed systems.

1.1 Motivation

The Emerald programming language was designed in Seattle, WA. in the early 80s by a small research group. The purpose of the language was both to facilitate distributed programming without sacrificing any more performance than necessary and to prove that OO design could be efficient and elegantly incorporated into the language [8]. Emerald programs run on a set of nodes that can be distributed across the world and connected at launch. Each node represents a physical machine, and is eligible to host Emerald objects and processes. Emerald is a rich and powerful language when it comes to distributed systems, but lacks a fundamental feature of connecting disjoint programs.

Although processes within an Emerald program are easily created and distributed, they can never interact across programs. An object reference in one program cannot in any way be passed to another. This creates rigidity in Emerald programs, as they do not allow any agency for connecting foreign users; all interactions must be predetermined at compile time, limiting the usability of such programs.

Connecting to a node without knowing its identification and port number is also impossible, thus to connect to an unknown node one first needs to know the node. We consider this paradox to be a fundamental flaw in Emerald.

To better illustrate these problems, consider figure 1.1. Here one program has created three objects A, B and C. Another program is running on a different node, and has created the objects X, Y and Z. With the current implementation there is no way for any of the objects ABC to become acquainted with XYZ and vice versa, and the two nodes will never know about each others existence. Even if the nodes were in the same node graph, and all the objects XYZ were moved to `node 1`, the two programs would not be able to communicate.

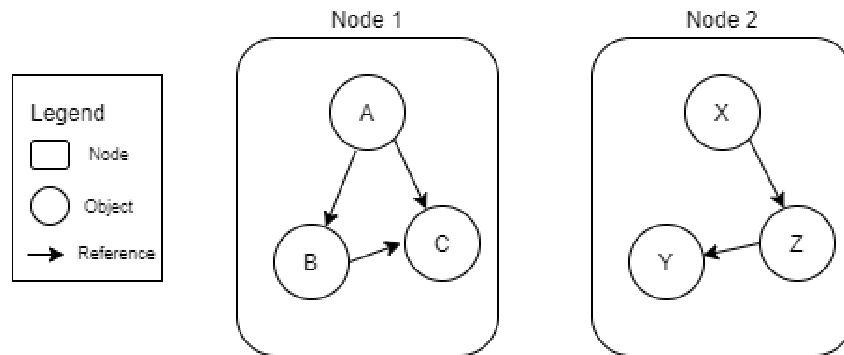

Figure 1.1: Disjoint object and node graphs

1.2 Problem Statement

Objects from different Emerald programs cannot become acquainted. Similarly, two Emerald nodes cannot become acquainted unless they are connected at the time of launch. We intend to address these limitations by introducing a language mechanism allowing both object and node graphs to merge.

1.3 Goal

The main goal of the project is to allow Emerald programs to discover and communicate with one another. We intend to achieve this goal by designing a new language mechanism, which we call WELCOME. This mechanism solves problems on two levels. The first enables disjoint programs to become acquainted through emissary objects. The second lets one node discover another and merges their node graphs. To demonstrate the WELCOME language mechanism, we intend to create a working proof-of-concept prototype of an extended Emerald compiler and virtual machine.

1.4 Approach

We have been using an experimental approach in this project. We worked incrementally by creating and testing various implementations for each problem until we discovered a satisfactory solution. A criteria for the implementation was that it needed to be grounded in research literature of similar problems and adapted to fit the Emerald environment. The working solution also needed to provide reproducible results.

1.5 Work Done

In this project we have designed a language mechanism that allows programs to exchange object references using emissary objects. Furthermore, we have rewritten and added functionality to parts of the Emerald compiler and virtual machine to support the new WELCOME language mechanism, as well as provided documentation for its usage. We have also created and tested Emerald programs that demonstrate the new features.

1.6 Evaluation Criteria

We have evaluated our work based on the extent of how the functionality provided by our solution achieves our goal, i.e. to exchange object references across disjoint programs, merging their object and node graphs in the process. This includes the usefulness and robustness of the working implementation, as well as a discussion of whether our features introduce a novelty to the Emerald programming language.

1.7 Results

We have enabled Emerald programs to discover one another and exchange object references across distributed disjoint programs by implementing the WELCOME language mechanism. The implementation consists of two parts, which allow for both Emerald node and object graphs to be merged.

1.8 Conclusion

We find that the WELCOME language mechanism solves the problem of exchanging object references between distributed disjoint programs, and provides a way for unrelated programs to become acquainted. Furthermore, a prototype implementing the WELCOME language mechanism demonstrates that service discovery and communication between distributed systems can be performed at the language level.

1.9 Contribution

We have expanded the Emerald programming language to include the WELCOME language mechanism. This includes a working prototype Emerald compiler and virtual machine that can use the `welcome` expression and the `welcomable` object prefix. These are language level features that allows communication across distributed programs and provide support for service discovery. The source code of the prototype can be found on GitHub [13].

1.10 Limitations

The WELCOME language mechanism does not consider splitting merged node graphs: if two node graphs are merged, they cannot be separated. Additionally, a distributed system using the WELCOME language mechanism cannot guarantee that it will become acquainted with all systems attempting to do so.

The implementation of the prototype does not account for all concurrency issues. Furthermore, all Emerald programs that are going to interact must be compiled in the same environment. Such environments can be stored into files and distributed separately for future compilation.

1.11 Thesis Outline

The thesis is divided into four parts and nine chapters:

Part 1: Introduction and Background

1. **Introduction:** A brief introduction of our main problem, goals and results of the project.
2. **Background:** A brief introduction to distributed systems, compilation and virtual machines, as well as the test environment we have been working with.
3. **The Emerald programming language:** An overview of the Emerald programming language.

Part 2: The Problem

4. **Inter-Program Communication:** A discussion on the rigidity of Emerald programs and the lack of agency for connecting users.
5. **The Problem of The Unknown:** A discussion on the topic of discovering Emerald nodes.

Part 3: The Solution

6. **WELCOME - A Multilevel Solution:** A proposal of a language mechanism that enables merging object and node graphs in Emerald.
7. **Implementation:** Technical details about the implementation of our solution.
8. **Evaluation and Results:** A Presentation of the results and test cases of our implementation as well as a discussion of whether our solution helped us reach our goal.

Part 4: Conclusion

9. **Conclusion:** This chapter concludes our report and provides a summary of our results.

Chapter 2

Background

In this chapter we discuss the benefits and pitfalls of distributed systems, aspects of compilers and how to create them, the concept of virtual machines and the PlanetLab testbed system.

2.1 Distributed Systems

With the rise of the world wide web, use of distributed systems has become commonplace. Despite its prevalence, the definition of a distributed system is somewhat elusive. In this thesis, we follow the characterization given by Tanenbaum & Van Steen [19] in which a distributed system is described as a collection of independent computers that appear to its users, whether they are programs or people, as a single coherent system. This means that the computers must be connected in some way, and that the communication between them is hidden for the user.

The benefits of a distributed system are many, and are reflected through its goals. Tanenbaum & Van Steen describes the goals of a distributed system as the following:

- **Sharing resources:** Providing easy access to remote resources is the primary goal of a distributed system.
- **Transparency:** Implementation details should be hidden from the user, presenting them with an easy and uniform interface. Location, replication, migration, means of storage, partial failure etc. could all be hidden in varying degrees.
- **Openness:** An open distributed system offers services through a strictly defined interface. Thus, any application that uses this interface exactly as it is described should be able to gain access to the services.
- **Scalability:** Adding more resources or users to the system should not have a significant impact on the performance of the system. Its distributed nature allows for dynamic workload sharing, and a decentralized and autonomous structure can allow computers to be added or removed as needed.

Though the possible benefits seem promising, distributed systems creates their own set of problems. For example, one or more of the connected computers could go offline any time, which may result in a partial crash of the system. When creating distributed systems, this should be considered as an expected event.

Additionally, creating distributed programs requires extra attention so as not to fall into pitfalls unique for such programs. These pitfalls have been known as the eight fallacies of distributed computing [17], and concern network reliability, security and homogeneousness, changes to topology, latency, bandwidth usage, transportation cost and administration.

The Emerald distributed system and programming language achieves several of the goals listed above, but deliberately provides the user with control over location and migration. Moreover, the Emerald programming language minimizes the possible mistakes when creating a distributed system, as the underlying virtual machine takes care of most of the pitfalls listed above. Furthermore, programmers of Emerald are provided tools to handle hazardous distributed environments, in which the topology change frequently.

2.2 Creating a Compiler

A compiler is a computer program that translates a computer program written in one language to a computer program written in another language [5]. Many compilers, such as the GNU Compiler Collection [20], compiles source code into executable machine instructions for a specific instruction set. Running programs compiled in this way is typically very fast, as they can be run directly by the CPU. Drawbacks with these types of compilers are that they must be updated or recreated for every new instruction set that is released, and the compiled code is not portable. Therefore, some programming languages, like Java, are instead compiled into a set of *bytecode* [5], which is a compact representation of the code that must be run on a custom built virtual machine. As running this code requires running an additional program, it is inherently slower than the previous alternative. However, only the virtual machine will need to be recompiled for a specific instruction set as the compiler itself does not need to change.

The Emerald compiler previously compiled Emerald source code directly to executable machine instructions, but has since been rewritten to output bytecode for the Emerald virtual machine. It is implemented as a bootstrapping compiler, i.e. a compiler that can compile itself to produce a newer version. As such a compiler naturally brings with it the chicken-or-egg problem, there must at some point have been an original compiler created in another language.

Any programming language, compiled or interpreted, consist of a collection of grammatical rules that define the syntax of the language. For each rule, an unambiguous definition of the semantic must follow. When compiling a program, each atomic element in the program is transformed to a token by a scanner, and these tokens are organized into a syntax tree by a parser.

The Yet Another Compiler Compiler (YACC) [9] is a parser generator that inputs a set of rules and corresponding code to be invoked when a rule is recognized. The Emerald compiler performs type checking and code generation based on the syntax tree generated by the output of YACC.

2.3 Virtual Machines

Virtual machines serve as an abstraction between running software and an underlying system, and can be divided into two types: system virtual machines and process virtual machines. Emerald programs are run on the latter type, which can be defined as virtual platforms that execute individual processes [18].

To achieve cross-platform compatibility, high-level programming languages like Java [12] can be compiled to run on process virtual machines instead of actual hardware. By doing so, performance is traded for portability. Programs are compiled into a bytecode representation and interpreted by the virtual machine. In a distributed system, this acts as a middleware and provides homogeneousness as every computer is represented through the same virtual machine, no matter what the underlying operating system and architecture is. When hosting processes on a virtual machine, the underlying OS of the physical machine views all processes as one. Thus, they technically run in the same address space and can easily share resources with each other. Additionally, the underlying OS only sees one executable program being run, as the high level executables are merely data files parsed by the virtual machine. This enables executable code to be changed during runtime, if desired.

2.4 PlanetLab

Testing distributed systems can be difficult, as access to several machines in different locations can be hard to acquire and operate. Also, using a virtual network may not reflect real world challenges such as unexpected node failure, network failures and differences in latency.

PlanetLab is a global overlay network that offers a collection of distributed computers. It is designed to facilitate testing and development of various distributed services from an early prototype through iterations of design to finally hosting popular services. By design it supports network services that benefit from being widely distributed. Examples of such applications include scalable object location, scalable event propagation, network embedded storage, peer-to-peer file sharing etc. [4].

2.4.1 Using PlanetLab

Through registration a user can gain access to a PlanetLab slice. This will grant semi-root access to a virtual machine on a set of nodes placed at different geographical locations. With a single user account one is granted access to resources on hundreds of computers spread across the globe (see figure 2.1). Each PlanetLab machine has multiple virtual machines running, and a virtual machine monitor managing their resources. Each virtual machine will have access to a portion of the resources available on the physical machine it is running on, and can be set up like a regular server. Semi-root access enables users to install any software of their choice such as Emerald on the various machines.

Although the PlanetLab system simplifies testing of distributed systems, resources are limited. Simple concepts and small programs may easily be tested, but as each of the distributed computers are shared among multiple users, heavy duty operations should be avoided.

Figure 2.1: Distribution of PlanetLab machines.
Downloaded 15.09.2020 from https://www.planet-lab.org/

Due to the shut down of PlanetLab in May 2020 [15], we use PlanetLab Europe for testing in our project. This version still provides machines across Europe and Canada.

2.4.2 Emerald and PlanetLab

We are using Emerald to write distributed programs, and PlanetLab to properly test our solutions in a real world environment. We believe that by using PlanetLab, we can encounter and solve more realistic challenges and get more accurate results when testing.

When working with Emerald in such an environment, objects can be moved between nodes hundreds of kilometers apart. Any alterations we make to the language that requires further synchronization need to be tested in a fault-prone scenario where simple `move`-instructions are time consuming and inefficient.

2.4.3 Security

While the users of various slices of PlanetLab are protected from each other, every PlanetLab node must be placed outside any firewall to function properly. Thus, any PlanetLab session must be considered an easy target for outsiders. As the PlanetLab nodes only provide a weak form of rooting to its users, the security of the underlying machine is not compromised [4]. In other words, while user-files uploaded to a PlanetLab machine may be compromised, the general infrastructure of PlanetLab is well secured. We do not consider the security aspect of either PlanetLab or Emerald in this thesis.

Chapter 3

The Emerald Programming Language

3.1 Introduction

Emerald is a general purpose OO programming language with a variety of built-in features for facilitating distribution, and uses a non-traditional object model [16]. The language forms a branch in a research tree on distributed systems [1], and distinguishes itself on some key features that we discuss further in this chapter:

- **Object mobility**: Objects can easily be moved around in a network and can be invoked remotely.
- **Object creation**: Objects are created without the notion of classes.
- **Types and conformity**: Type checking is done through structural conformity.
- **Emerald nodes**: Location in distributed programs is represented through the node abstraction.

3.1.1 Terminology

Some of the terminology used here may deviate from that used in other programming languages or articles on the Emerald programming language. In this thesis we use the word *operation* to refer to all kinds of object-bound methods and functions. Additionally, the term *abstract type* is used to describe a concept similar to interface in Java [6]. *Concrete type* is used to describe the implementation of an object. Finally, we refer to the Emerald kernel as the *Emerald virtual machine*.

3.2 Language Background

The Emerald system and programming language emerged as a response to the Eden research project [1]. At the time of its creation, most distributed programming languages and operating systems that supported object mobility had a binary object model. One type of objects were large objects, typically including a process with its entire address space, and this was used as the unit of mobility. The other type was used to create small data structures and did not support mobility. Thus, the programmer would need to know and explicitly state what kind of object to use for a given situation a priori. Emerald sought to remove this distinction by hiding the two object models

```
const Kilroy ← object Kilroy
    process
        const origin ← locate self
        const up ← origin.getActiveNodes
        for e in up
            const there ← e.getTheNode
            move self to there
        end for
        move self to origin
    end process
end Kilroy
```

Figure 3.1: The Kilroy example

from the programmer. Instead, a single object model is presented and it is up to the compiler and runtime system to decide which object model that should be used for a given object based on its behavior. Moreover, the apparent single object model allows for fine-grained mobility, as the unit for mobility can be much smaller than e.g. a migrating process [10][11]. A single object model also facilitates the creation of distributed applications by making location transparent; The programmer does not need to know whether an object is local to the node or not to invoke one of its operations. At the same time, the location of an object may be explicitly retrieved or set through the mobility constructs in the Emerald programming language.

At the time, object oriented programming languages were considered slow, as it was commonly assumed that the OO-model required too much overhead [1]. The creators of Emerald wanted to demonstrate that invoking local objects could achieve performance comparable to that of C [7]. They also wanted Emerald to be a high-performance *distributed* system, and aimed at outperforming Eden in both performance and memory usage. One key design choice that enables Emerald programs to be optimized is including the concept of location. Retrieving the location of an object in combination with Emerald's set of mobility constructs enables the programmer to decide if an invocation should be remote or not [1].

3.3 Object Mobility

One of the most important design choices in Emerald is the simplistic syntactic support for object mobility. Whereas other languages need multiple lines of code to prepare and move an object from one machine to another, Emerald bakes this feature into one single line: `move [object] to [location]`. Emerald offers fine-grained mobility without sacrificing the speed of invoking local objects [11].

The Kilroy example (see figure 3.1) showcases how an Emerald program can efficiently move objects between nodes (computers). The program consists of a single object containing a process, which moves itself through every available node in the system. Finally it moves back to its original node and terminates.

To avoid unnecessary overhead on the network, the Emerald virtual machine attempts to optimize object movement. When using the `move` statement, the objects

```
const myObjReference ← <immutable | monitor> object anObj
    const myConst ← 5
    var myVar : Integer ← 6

    export operation myOperation[arg : Type] -> [retVal : Type]
        % Operation code
    end myOperation

    initially
        % Initial code
    end initially
    process
        % Process code
    end process
end anObj
```

Figure 3.2: The object constructor in Emerald. The immutable and monitor attributes are optional.

may or may not be moved depending on the optimizations. The `move` statement is therefore a "best effort" move, and does not provide a guarantee that the object actually moved. If the programmer wants to ensure that an object is moved, the `fix` statement should be used. Using this statement, the object is moved as long as the destination is available. Additionally, when an object is fixed at a location, it cannot be moved or fixed again, unless it is `unfixed` first.

Emerald was made to function properly even in a hazardous environment where connections may unexpectedly disappear. To let programmers plan for such incidents, Emerald includes a language construct that tells the program what to do when trying to invoke an unavailable object. This is done by using the built-in `unavailable` handler at the end of any process or operation.

3.3.1 Attached Objects

To minimize remote invocations, it may be desirable to move certain objects along with another. Emerald facilitates this through the `attached` keyword, which may precede any variable declaration. An object may have an attached object reference, which in turn has its own attached references, forming a tree structure. When an object is moved, its sub-tree of attached objects moves with it. However, if the object itself is attached to another object, the parent will not move.

3.4 Object Creation

Objects are created on the fly in Emerald without the need for classes, and every object is assigned a unique object identifier (OID). As can be seen in figure 3.2, the object constructor itself contains the details and internals of the object. Once the constructor runs, the object is created. Multiple similar objects can be created by running the constructor multiple times. The notion of classes can be replicated by placing an

object constructor inside an operation and return a reference to the object created [1]. Although objects made by the same constructor are unrelated, the conformity system in Emerald creates a relation between them.

The object constructor contains the implementation of the operations of the object, as well as the optional `initially` and `process` sections. While the `initially` section is run by the creator of the object [10], the `process` section creates and is run by a new thread. For the rest of this thesis we refer to such threads as *processes*, as this is how they are referred to in the Emerald programming language.

Two attributes can be specified in the object constructor to specify certain behavior. The *immutable* attribute makes an object unchangeable and omnipresent. As there are no primitive data types in Emerald, this attribute is especially important for objects of type Integer, Real, boolean etc. Without this attribute, the number (and object) `4` could change its value, be sent across the network and even become unavailable. Immutable objects allows for object duplication, and makes sharing object code between nodes convenient. The *monitor* attribute specifies that all operations in an object should be thread safe.

3.5 Types and Conformity

Interacting with objects created after compile time can be a problem in long running systems as the type of the new object needs to be known at compile time. The structural conformity system in Emerald solves this issue: Object A conforms to type B (and can be viewed as an object of Type B) if A contains at least all the same operations as B. Thus, if A implements B's interface, object A may be used in place of any object of type B. Although this relation looks similar to inheritance, A need not explicitly state its relation to B: "[...] inheritance is a relationship between implementations, while conformity is a relationship between interfaces" [2, p. 70].

The Emerald language is strongly typed and based on the concept of abstract types [10]. An abstract type describes a set of operations and consists of the name, the number of parameters and return values of each operation, and their respective types. Abstract types allows multiple implementations of the same type as types are compared using structural conformity.

> "The basic question that the type system attempted to answer was whether or not a given object (characterized by a concrete type) supported enough operations to be used in a particular context (characterized by an abstract type)." [1, p. 14].

As can be seen in figure 3.3, the objects `Cat` and `Dog` do not in any way state a relation to each other or to the abstract type `Animal` (defined using the keyword `typeobject`). Still, both objects conform to `Animal` and may be used in place of each other.

3.6 Object Structure

Object are represented in memory through a collection of structures. Instance variables and the code are stored separately to optimize memory usage when having multiple

```
const Animal ← typeobject Animal
    op eat
end Animal

const Cat ← object Cat
    export op eat
    end eat
end Cat

const Dog ← object Dog
    export op eat
    end eat
end Dog
```

Figure 3.3: Structural conformity in Emerald

instances of objects created by the same object constructor. These objects share their code, as the implementation of all operations and processes are the same. We call this the *concrete type* of an object, but it has also been called the implementation or code in various articles. Static attributes shared by all objects with the same concrete type, such as whether the object is immutable or a monitor, is also placed here.

On creating a variable in Emerald, an abstract type must be specified, of which object references stored in the variable must conform to. When an object reference is assigned to the variable, its concrete type can be accessed through the object. The memory layout for object references is illustrated in figure 3.4 and 3.5.

Abstract and concrete types are themselves implemented as immutable objects and are not sent along when moving an object [10, p. 59]. Due to the nature of the immutable property, types need only be shared once between nodes. This makes objects compact, and only the instance data fields of the object are sent.

3.7 Distribution and Location

An Emerald program runs on an Emerald node (hereinafter just called node), which is an abstraction of a physical machine. The node is used to express location, although multiple nodes can be hosted on a single physical machine [1, p. 26]. By connecting nodes, several machines can form a network creating a distributed environment for Emerald programs. Every node runs on an interpreter for Emerald bytecode, namely the Emerald virtual machine. It handles process and object creation, process scheduling etc.

A network of nodes can be created by starting new nodes and directly connecting them to a known node. One initial node is launched, and by knowing its IP address and port number other nodes may connect to it. These nodes may in turn be connected to by other new nodes, creating a network. We refer to such networks as *node graphs*.

In the Emerald programming language, nodes are represented by the built-in `Node` object. This object provides various operations through its interface, e.g. the ability to output text in the terminal, open local files, acquire references to all nodes in the node

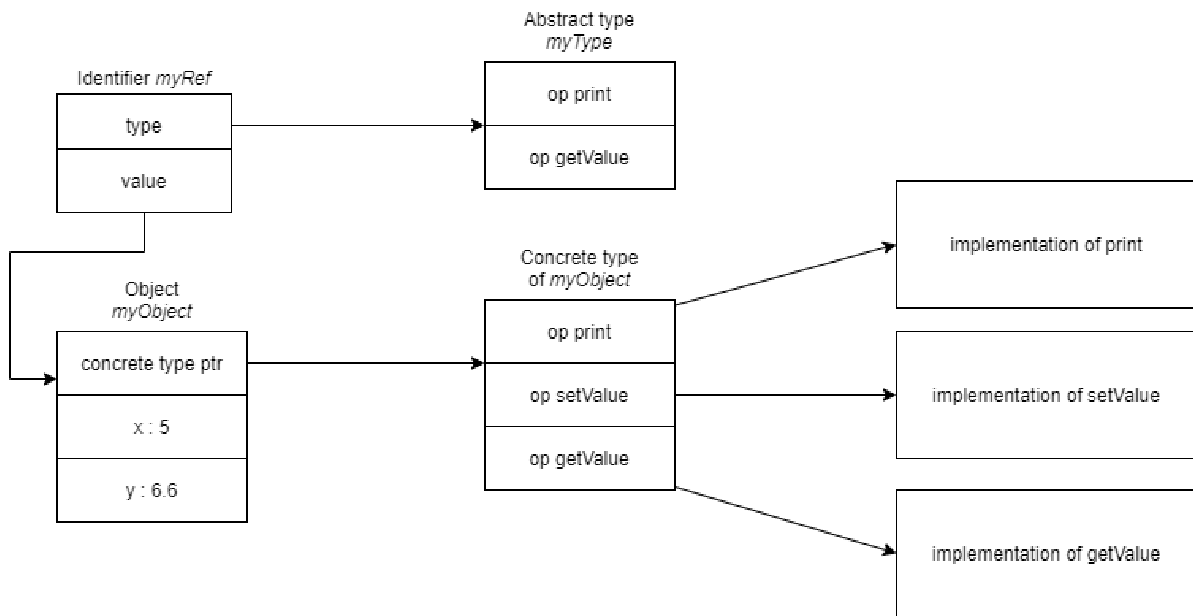

Figure 3.4: The memory layout of an object and its concrete and abstract type

```
1  const myType ← typeobject myType
2      op print
3      op getValue -> [Integer]
4  end myType
5
6  const myRef : MyType ← object myObject
7      var x : Integer ← 5
8      var y : Real ← 6.6
9
10     export operation print
11         % Implementation of print
12     end print
13
14     export operation getValue -> [res : Integer]
15         % Implementation of getValue
16     end getValue
17
18     export operation setValue[param : Integer]
19         % Implementation of setValue
20     end setValue
21
22 end myObject
```

Figure 3.5: Code producing the object structure illustrated in figure 3.4

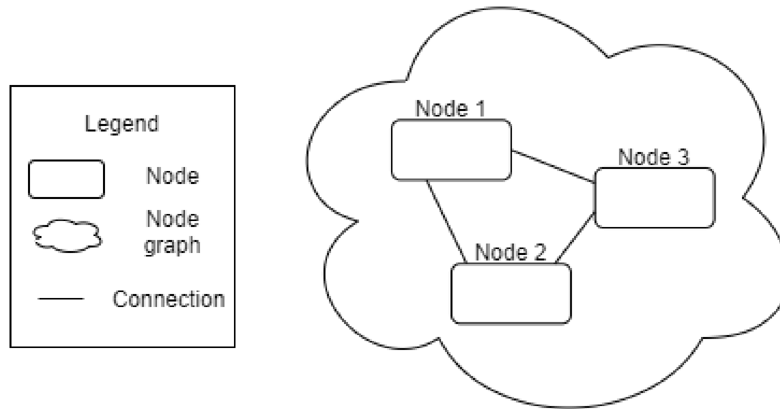

Figure 3.6: An Emerald node graph. Node graphs are always complete.

graph and more.

3.8 Summary

In this chapter we have discussed some key features in the Emerald programming language that distinguish it from other OO languages. Object mobility is achieved through simple built-in syntax, and commonly occurring network failures are handled using dedicated handlers. Objects are created using object constructors without the traditional notion of classes, and are an abstraction of data and types. Types are compared through structural conformity, and the concept of abstract types describes a certain behavior of any conforming object. Lastly, distribution is handled by the abstraction of nodes, which is used to express location. A network is created by multiple connected nodes, forming a node graph.

Part II

The Problem

Chapter 4

Inter-Program Communication

Emerald was meant to simplify the programming of distributed systems and applications by providing language support for distribution [3]. According to Tanenbaum & Van Steen, one of the main goals of a distributed system is to make it easy for users to access remote resources [19]. However, connecting users to a running service is not possible in Emerald without drastic workarounds, limiting the use cases of Emerald programs.

4.1 The Limits of a Program

An Emerald program is a series of bytecode that is interpreted on an Emerald virtual machine. Every program can have multiple references to objects, which can have further internal object references. An Emerald program can therefore be viewed as a directed object graph. Objects in an Emerald program can only reference objects in its residing object graph. This becomes a problem when we want to allow communication between programs.

There are a few ways of communicating with a distributed system written in Emerald, however these are unintuitive and not designed for that purpose. For example, to create a server and client application in Emerald, the two must be part of the same program. A user connects a node to a server and the server responds by sending a client application in return. The client application is created and compiled together with the server. If an update is needed on the server, the user must reconnect and receive a new client.

To create programs that can communicate, a possible solution is to utilize the local file system of a common node to implement a sort of message passing by reading and writing to a commonly known file. This is a problematic solution for several reasons. One is that the programs are dependent on the file system of a physical machine, creating a single point of failure. Also, the Emerald virtual machine can be said to provide a middleware abstraction, as details on which architecture the node runs on is transparent to the user [19]. If one uses the local file system of a node to communicate, this abstraction is broken. Additionally, utilizing a local file system for communication is prone to both security and concurrency issues. Lastly, such a solution requires a common protocol to be known, and data needs to be serialized before it is sent. If objects are sent this way, the reference is not sent and any changes made to the object would not be registered. The main problem with serialization is that only the state of

an object can be recreated on the receiving side, and not the implementation.

While communicating using a common file has its own set of issues, providing users access to more conventional network communication tools like network sockets would also not be sufficient; The implementation of a serialized object cannot properly be recreated at the receiving end if it is not serialized and sent by the Emerald virtual machine itself. Furthermore, references cannot be sent, only snapshots of an object's state. In the next section we discuss why using a reference is important, and show an example of a specific use case that highlights the need for passing object references across programs.

4.2 Inter-Program Referencing

Emerald features an encapsulation mechanism that allows multiple programs to be run on the same Emerald virtual machine without knowing about each others existence. This way, programs can share infrastructure without risking interference from unknown sources.

Accessing objects in another object graph is technically possible if the programs run in the same address space, however Emerald was designed so that this should not be possible. Therefore, using object references is the only way to access and modify data in Emerald. This encapsulation mechanism upholds the principle that an object's data should only be accessible to those who have been given a reference to it. If an object's data could be accessed across object graphs without a reference, this principle would be broken.

There is a clear distinction between receiving an object reference and receiving a serialized representation of an object; While a serialized representation merely contains the state of a, possibly unknown, object, a reference provides direct access to the object and its interface. This distinction becomes important when considering the following example: We want to create a framework for replicating and distributing objects. The framework should be able to receive any unknown object, create replicas of it and distribute the replicas to all known nodes. We also want to create an application, and we intend to use the framework to replicate objects in the application. To do this, we need to send objects from the application to the framework. As these are two different programs, we cannot use the method of sending a user client to a newly connected node, as discussed in the previous section. If we serialize the objects and use a common file for communication, we cannot recreate the objects in the framework, only store their state.

The example above demonstrates a need for passing objects across programs. As this is not possible in Emerald, and merely sending an object's state is not sufficient, we argue that a way for programs to exchange object references is needed. This would merge the object graphs of the programs, and thus merge the programs. For clarity, we henceforth refer to unmerged programs as *disjoint*.

4.3 System Extensibility

So far we have discussed the problems of connecting users to existing distributed systems created in Emerald. However, a related problem is that of modifying and

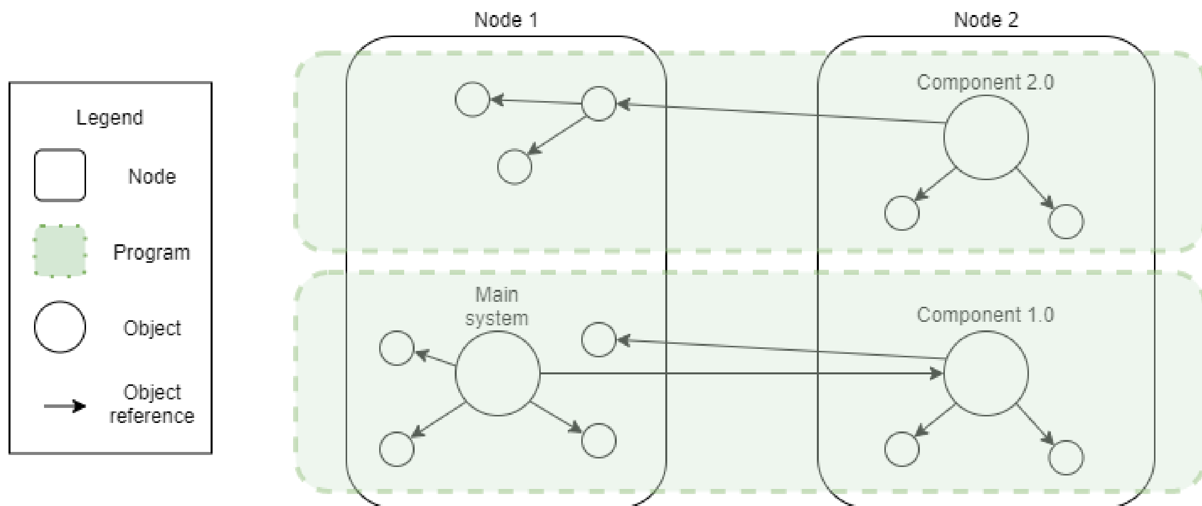

Figure 4.1: The extensibility problem in Emerald

expanding a running system without shutting it down, which has been an intended feature in Emerald for decades:

> "[...] since shutting down and recompiling an entire distributed system in order to modify some component is unacceptable, the [distributed] language must permit system extensibility without recompilation; existing programs must continue to work in collaboration with new programs" [3, p. 65].

For existing programs to work in collaboration with new disjoint programs, they need to access objects within each other's object graphs. As we have seen in the previous sections, exchanging object references across programs is not currently possible in Emerald, even if they reside in the same set of nodes. Figure 4.1 illustrates this problem: A running system needs to update a component. The new and updated component is ready and running, but the system has no way of accessing it. If the new component was compiled in the same environment as the running system, it would also be a part of the running program, as it would have references to its objects. However, other disjoint programs wanting to use the component would suffer from the same problem, making program level modularity difficult. This reinforces our view that being able to exchange object references across programs would be beneficial.

4.4 Summary

In this chapter we have discussed the current possible ways of connecting users and user programs with existing distributed systems written in Emerald, and why these methods are not always sufficient. We have also considered the differences between exchanging serialized objects and object references. Moreover, we have examined the benefits and limitations of encapsulation through object references and argued that exchanging object references is necessary for accessing objects across programs. We have also shown two examples that illustrates this need: One for connecting users to

a running service, and another for replacing or updating components in a distributed system.

Chapter 5

The Problem of the Unknown

In chapter 4 we discussed the problem of letting disjoint programs become acquainted. However, any solution to this problem would presuppose that the disjoint programs are in the same node graph. In situations where it is desirable for a program to become acquainted with programs from an unknown node graph, like programs running on nearby mobile or IoT devices, such a solution would not be sufficient.

5.1 Connecting Nodes

In Emerald, a node graph can be created by launching new nodes one by one and connecting them on startup. A node can connect to another node by specifying its *identity*, where the identity consists of the node's hostname/IP address and port number. This adds the new node to the other node's node graph. However, once a node is running, it cannot connect to nodes running in other node graphs. We refer to nodes residing in other node graphs as *foreign* nodes. Every node graph is isolated and limited to host programs that were started on its nodes, or that will be started on any new node that connects to the node graph.

As no node can bridge the gap between two node graphs, they will always remain *disjoint*, and every node's resources will be limited to its residing node graph. If disjoint programs from disjoint node graphs are to become acquainted, information must pass between the graphs. As this is not possible in Emerald, a new mechanism is required to allow communication across disjoint node graphs.

Consider the following example: We want to create a program that gives any movie a score based on its director and the composer of its soundtrack. We are familiar with two services; one that provides the director of any given movie, and another that provides the composer of the soundtrack. After creating an Emerald client, we may connect it to either one of the two services, as both of their identities are known. However, as can be seen in figure 5.1, the client cannot connect to both services at the same time because they run in different node graphs.

5.2 Connecting to Unknown Services

A mechanism that allows for communication across nodes from disjoint node graphs would require the identity of at least one foreign node to be known. This works for

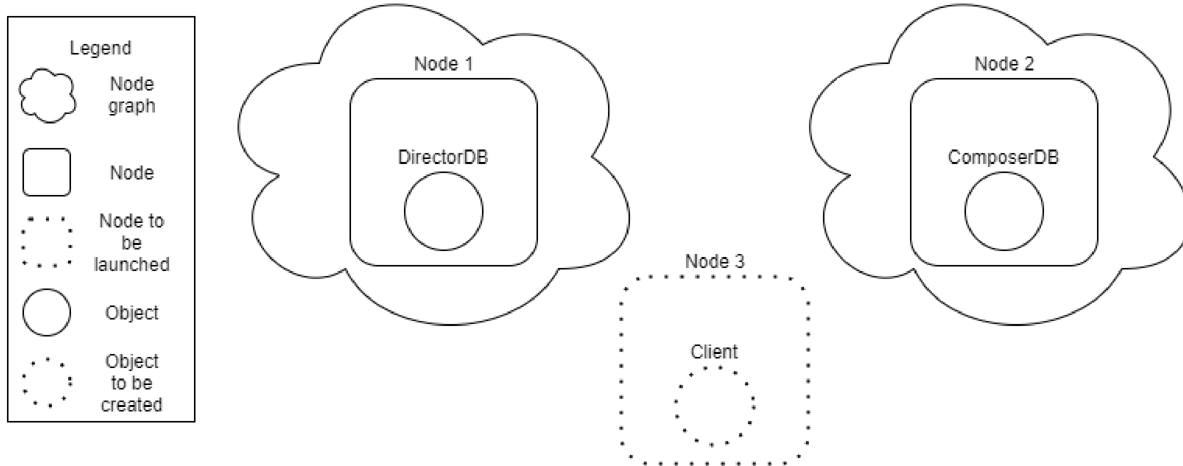

Figure 5.1: Isolated node graphs. The Emerald client must choose a single node graph to join.

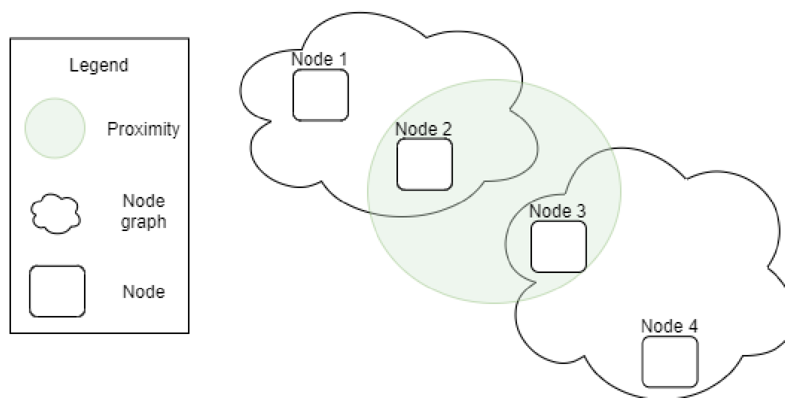

Figure 5.2: Foreign nodes in proximity.

known services, such as the movie scoring example from 5.1. However, obtaining the identity of a node in another node graph is not always possible.

Two devices may want to connect even if they do not know each others identity. IoT-devices often fall into this category, e.g. a printer on a local network. When users want to use the printer, they may not know how to obtain its identity. In such a scenario a connection must be formed in a different way, i.e. the user needs a mechanism to discover the identity of the printer.

Discovering nearby nodes without knowing their identity is currently not possible in Emerald. In figure 5.2 two nodes from disjoint node graphs are in proximity, but have no way of communicating. Using the example above, one of the nodes could belong to a printer and the other to a client device that is ready to print. Even if a mechanism to connect to a foreign node is implemented, the client device has no way of discovering the printer's identity, even though they are in close proximity. Thus, for an Emerald program to connect to an unknown service, a non-identity based node discovery mechanism is needed, in addition to a way of communicating across disjoint node graphs when the identity of a foreign node is known.

5.3 Summary

In this chapter we have discussed why it is not adequate to limit a program to its residing node graph, and highlighted the problem of connecting to unknown services. The problem is twofold; One is that Emerald provides no way for nodes to connect to foreign nodes, and the other is that discovering nodes is not possible in Emerald. A node discovery mechanism would in itself not be sufficient, as there would not be any way to communicate with the discovered foreign node. Likewise, providing a way for foreign nodes to communicate would only solve the problem when the identity of the foreign node is known. Thus, any solutions to these problems would need to work in tandem.

Part III

The Solution

Chapter 6

WELCOME - A Multilevel Solution

6.1 Introduction

In the previous chapters we identified some shortcomings of the Emerald programming language that makes it hard, or in some cases impossible, for distributed programs to become acquainted. We argue that a new language mechanism for exchanging object references across disjoint programs, which are possibly located in disjoint node graphs, is required to address these shortcomings.

We believe that introducing a new language mechanism is an advantageous approach to addressing the shortcomings, as it provides more agency to the programmer; The timing and degree of its usage is a deliberate choice, and an optional language mechanism is backwards compatible with existing Emerald programs that do not use this feature.

As discussed in chapter 4, object data is encapsulated through object references. This is a deliberate choice in the design of Emerald, so that only those who receive an object reference are entitled to access its data. With a language mechanism that enables disjoint programs to acquire each other's object references, this principle is easily broken. Therefore, any solution must ensure that object references are exchanged in a controlled manner, and that the exchange is deliberate and favorable by both sides.

Our proposed solution, which we call the WELCOME language mechanism, is split into two related parts: exchanging object references across disjoint programs and dynamically merging disjoint node graphs without necessarily knowing the identity of a foreign node. Each part could function as a stand-alone solution for its problem, but both are needed to form a coherent solution.

A new language mechanism would be an extension to the Emerald programming language, and would thus require changes to both the Emerald compiler and virtual machine. In this chapter we describe our proposed new syntax and semantics of the WELCOME language mechanism. We also outline some necessary changes to existing built-in objects. The changes presented here will not impact existing programs, meaning that the resulting Emerald compiler and virtual machine can be used on existing programs without requiring any modifications to the source code.

6.1.1 New Terminology

For the remainder of this thesis, we regularly use new terms that relates to the WELCOME language mechanism. Although these terms are introduced as needed,

table 6.1 contains their precise definition and can be used as a reference.

Term	Definition
The `welcome` expression	A proposed blocking expression for the Emerald programming language.
Welcoming process	A process blocking on a `welcome` expression.
Welcoming object	An object containing a welcoming process.
To be welcomed	For an object to have its reference passed to a welcoming process, causing it to unblock.
Welcomable object	An object with the property of being able to be welcomed.
Unwelcome object	An object *without* the property of being able to be welcomed.
To welcome	For a process to block on a `welcome` expression, waiting for a welcomable object conforming to a given type to move to its residing node.
Welcome object	A welcomable object on the move to a node hosting a welcoming object, and that conforms to the type the welcoming process welcomes.
Welcomed object	An object that has been welcomed by a welcoming process.
Emissary object	An object intended to be welcomed in order to serve as a link between programs or node graphs.
The WELCOME language mechanism	A proposed language mechanism that allows emissary objects to move across disjoint node graphs, merging the graphs if welcomed at the destination.

Table 6.1: Definition of terms introduced in this thesis.

6.2 Merging Object Graphs

In chapter 4 we discussed the problem of connecting disjoint programs, and found that a mechanism to exchange object references is needed for such a connection. Our proposed WELCOME language mechanism includes a new expression that allows a program to obtain references to relevant objects that are moved to its node. We also propose a syntactic addition to the object constructor that labels objects available for others to reference.

6.2.1 Exploring Semantics

The WELCOME language mechanism should provide a syntactic extension that enables the ability for a program to obtain object references from disjoint programs. Before we decide what the syntax will look like, we first need to decide on the semantics. In the following, we discuss the various possible semantics of this extension, and why we decided on using object mobility as a trigger for the exchange.

Firstly, the programmer should be able to have some knowledge about the behavior of the object that they receive a reference to, even though the object itself is unknown.

The type system in Emerald is helpful for achieving this, as there need not be a predetermined relationship between the abstract type and the object. By using an abstract type as a parameter to the language mechanism, the programmer can specify the behavior of the object without knowing its implementation beforehand. If the programmer does not know anything about the behavior of the object, the `Any` type can be used.

Secondly, the language mechanism should provide the programmer with the ability to communicate with other running programs. A possibility for providing a reference to an unknown object is to search all known nodes for objects that conform to the specified type. This would allow the programmer to obtain information about other programs at the time of their choosing, giving control to the programmer. There are however several problems with this approach: It would affect the performance significantly, as network packets would need to be sent to each node asking for an object. Additionally, the structure of how objects are stored would also need to change so that every object can be retrieved given its type. This method also brings further complications, such as how to decide which object should be chosen if there are multiple candidates. Finally, this option does not provide any useful way of actually communicating with other programs, as there is no way for the programmer to know if any unknown program is currently running on the same set of nodes. Thus, the language mechanism must be able to *listen* for some form of activity from another program.

Another possibility is to allow listening for newly created objects. When objects are created, any listeners listening to a type conforming to the object is notified. This allows for communication across the programs without the programmer needing to know where and when the other program is launched. However, this too is a problematic solution, as it would harm the performance of object creation. When an object is created, multiple network packets would need to be sent, and conformity checking would have to be done for each of the listening types. This would not scale well, as the cost of creating an object would depend on the size of the node graph. Moreover, objects would be exposed beyond the control of the programmer. Even if exposing the object to this language mechanism is optional, there would be no way to restrict who gets a reference to the newly created object. For example, if the programmer wants to share an object with a program located on node A, but not with programs on node B, they have no way of doing so.

We argue that a more favourable solution would be to listen for objects that are moved onto the local node. This requires no additional overhead on the network, and lets the programmer decide which node(s) the object should be shared with. In addition to this, the act of moving an object can be used as a communication tool, giving the programmer agency for when the communication should take place. The moved object then becomes an *emissary object*, as it serves as a link between the two programs. This method also has the added benefit that the listener and the sent object is located on the same node. As objects organized by the same Emerald virtual machine share the same address space, the reference can be passed internally without any indirection through global IDs or address translation.

```
const main ← object main
    const arrivedObjects ← Array.of[SomeType].empty
    const me : Node ← locate self

    me.setWelcomeEventHandler[SomeType,
        object welcomeObject
            export op doWelcome[obj : SomeType]
                arrivedObjects.addUpper[obj]
            end doWelcome
        end welcomeObject
    ]
end main
```

Figure 6.1: Proposal of welcome as an integrated event.

6.2.2 Syntax of the `welcome` Expression

In the previous section we established that the new language mechanism should provide programmers with a way to listen for objects that are moved onto the local node and conform to a specific type. As the move operation triggers the exchange, the syntactic extension should manifest as a mean to handle this event.

If the new language mechanism should take form as an event, we could look to other events in Emerald for inspiration when deciding the syntax. There are multiple ways of creating event handlers in Emerald: one is using the interface of a node, and another is creating a specialized block body handler. An alternative is to implement the event handler as a blocking expression. In the following, we discuss whether the new language mechanism should be blocking or not, explain why we prefer a blocking expression and provide examples of how the new syntax could look.

Extending the `Node` Interface

The interface of the `Node` type provides a `setNodeEventHandler` operation that allows programmers to add an event handler that listens for new nodes connecting to the node graph. The operation takes an object conforming to a predetermined type, and calls its functions as appropriate whenever changes occur in the node graph. Multiple handlers can be registered, and so a single connecting node can trigger multiple events. By extending the interface of the `Node` type, we can create support for a new type of event handler. In figure 6.1 we show an example of what a new event handler in the `Node` interface could look like. The `setWelcomeEventHandler` takes two arguments: the type of the object of which we will obtain a reference to, and an object conforming to some predetermined handler type. The handler object has an operation that is invoked when an object of the desired type is moved onto the local node.

This approach has the benefit of not needing any syntactic alterations to the language, only minor additions to the existing built-in type `Node` is required. However, one downside is that the handler is tied to the node and not to the program itself. Whereas the program can move around when it needs to, the node cannot.

```
const main ← object main
    const arrivedObjects ← Array.of[SomeType].empty

    welcome [obj : SomeType]
        arrivedObjects.addUpper[obj]
    end welcome
end main
```

Figure 6.2: Proposal of welcome as a block body.

Specialized Block Body

Emerald offers two specialized block body event handlers: an `unavailable` handler, and a `failure` handler. These are located at the end of a scope, and triggers whenever a remote invocation is performed on an unavailable node, or an action results in an error respectively. An important difference between these handlers and handlers set through the `Node` interface, is that they are not fired asynchronously; The handler is run sequentially as a result of code execution elsewhere. Another difference is that their lifetime is limited to the lifetime of the innermost scope that they reside in. This behavior is different from what we want to achieve when expecting references to foreign objects. Therefore, minor changes would need to be made to accommodate this event, if using a specialized block body.

Figure 6.2 illustrates a possible syntax for a `welcome` block body construct. Here the block body is placed directly in the object constructor, in contrast to the unavailable and failure handlers. This will tie the welcome event to the object itself, and not a specific scope within the object. The `welcome` block runs every time a welcome object arrives at the local node, and multiple `welcome` blocks could be created if objects of different types are desired.

A problem with this solution is that there is no way for the object to stop welcoming objects. The block would run indefinitely, limiting the programmer's ability to choose the quantity of received objects. This would also require alteration of the garbage collector, as an object with a `welcome` block should never be collected to avoid unexpected behavior.

`welcome` as an Expression

Finally, `welcome` could also manifest as a unary expression. The expression could take an abstract type as its operand and return an object reference of that type, as can be seen in figure 6.3. The expression would need to block until an object conforming to the specified type moves onto the local node.

This solution differs from the two previous alternatives. A blocking expression stops the execution of a process until a welcome object arrives, while the asynchronous nature of the previous solutions allows them to continue execution while simultaneously waiting for new objects, fully using the resources available at all times. However, as objects are easily created in Emerald, a new object with a blocking process can be made to imitate the behavior of an asynchronous event handler. This is illustrated in figure 6.4.

```
const main ← object main
    process
        const newObject ← welcome SomeType
    end process
end main
```

Figure 6.3: Proposal of welcome as an expression.

```
const main ← object main
    field unknown : SomeType ← NIL
    process
        object a
            process
                main$unknown ← welcome SomeType
            end process
        end a

        loop
            exit when unknown !== NIL
            % do heavy computations
        end loop

        % use the welcomed object
    end process
end main
```

Figure 6.4: Welcome imitating an asynchronous event.

```
const anObject ← welcomable object anObject
end anObject
```

Figure 6.5: A welcomable object

One downside with this solution is that there is no way to guarantee that all arriving objects on a given node is welcomed. A process may be designated to welcome all incoming objects, but as some time must be spent handling the incoming object, and since two or more objects may arrive simultaneously, the welcoming process will only be able to welcome one of the arrived objects before it can welcome again.

We find that the syntax of this alternative is simpler than the former alternatives. This combined with the ability to choose whether or not an asynchronous event handler is required, and the option to choose the number of objects that are to be welcomed, makes this solution a preferable option. Additionally, with this solution we are guaranteed that any object that is welcomed is always present at the same node as the welcoming object, which is likely to require less restructuring of the Emerald virtual machine and creates no extra overhead on the network.

6.2.3 Welcomable Objects

In some situations it may not be appropriate to allow disjoint programs to obtain references to each other's objects. The WELCOME language mechanism should ensure that object references are exchanged in a controlled manner, and that the exchange is deliberate and favorable by both sides. However, the `welcome` expression as suggested above would allow anyone to listen for and modify any given object, compromising the natural boundary of an Emerald program and its incorporated encapsulation mechanism.

To address this security issue, the programmer should be allowed to choose whether an object should be welcomed or not. One way of implementing this is marking an object as *unwelcome*, either through a statement or by specifying it as a property when creating an object. An unwelcome object is an object that cannot be welcomed. This would solve the issue, but would require every object that is not intended for use with the WELCOME language mechanism to be marked as unwelcome.

The inconvenience of this opt-out unwelcome solution escalates further with the fact that it is not backwards compatible, i.e. any program written before this mechanism was introduced will have all of their objects exposed. Therefore an opt-in solution seems more viable, as it is both backwards compatible and only requires the emissary objects to be marked. We call objects that may be welcomed *welcomable* objects.

One way of implementing this in the prototype is to create a statement that marks an object as welcomable. Similarly, another statement could be added to mark the object as unwelcome again. This solution is dynamic as the property can change over time, but comes with its own problem; an object intended to remain unwelcome may at some point be marked as welcomable. Statically and permanently marking an object

```
const privateObj ← object privateObj
    export operation performAction
        % performing some action
    end performAction
end privateObj

const wrapper ← welcomable object wrapper
    attached const original ← privateObj

    export operation performAction
        original.performAction[]
    end performAction
end wrapper
```

Figure 6.6: A welcomable object wrapper

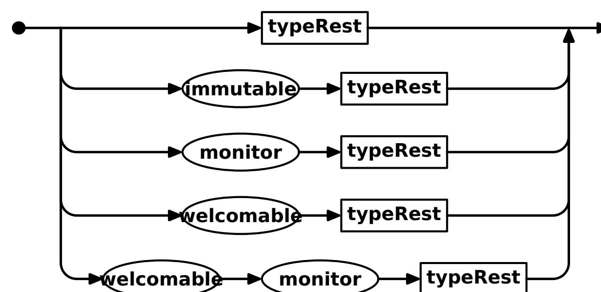

Figure 6.7: Grammar of available object attributes in Emerald. `typeRest` refers to the rest of the object constructor.

as welcomable at its creation eliminates this problem (see figure 6.5), but removes the possibility of dynamically changing the property. Even with this limitation we prefer the latter solution, as a welcomable wrapper object can be used to welcomably move an unwelcome object. Thus it is up to the programmer to determine if an object should be welcomable in a static or dynamic way, while simultaneously minimizing the required new syntax for the Emerald programming language. A downside to this approach, is that existing built-in objects cannot be marked as welcomable. However, this can also be solved using a welcomable wrapper object. Figure 6.6 shows an example of creating a wrapper object using the `welcomable` keyword.

The grammar of the `welcomable` keyword is visualized in figure 6.7. As can be seen in the figure, immutable objects cannot also be made welcomable. This is because of the omnipresent nature of immutable objects in Emerald; Objects that are everywhere cannot be moved, thus it makes no sense to make them emissary objects.

6.3 Merging Node Graphs

In chapter 5 we discussed the problem of connecting to both known and unknown services. As there are no way of communicating across disjoint node graphs, a

```
const Discover ← object Discover
    const home ← locate self

    process
        const nearbyNodes ← home$discoveredNodes
    end process
end Discover
```

Figure 6.8: Example of use of `getDiscoveredNodes`

mechanism for dynamically merging node graphs is needed. In the following, we explore the options for allowing communication across disjoint node graphs and how, when and if they should be merged.

6.3.1 Extending the `Node` Interface

To allow communication across disjoint node graphs, we need a way of discovering foreign nodes without knowing their identity, as well as a way for the programmer to access the newly discovered node. While discovering nearby nodes can be achieved through a broadcasting system on the local area network, the latter problem warrants a discussion of the available options.

In 6.2.2 we discussed options for how to present the programmer with a newly introduced object reference from another program. As this problem is similar (we want to introduce a new node from another node graph), we can revisit these options to evaluate them in the context of discovering nodes.

We argue that the discovery mechanism should be tied to the node and *not* the object. This rules out both a dedicated statement or expression, and the block body solution. Furthermore, we do not want to extend the syntax of the Emerald programming language more than necessary. For obtaining object references, we dismissed extending the node interface as we did not want to tie the action of welcoming to a specific node, but instead allow welcoming objects to move around. However, managing node connections is done by the nodes themselves, and so the action of discovering nodes should not be delegated to objects directly. Instead, objects could access discovered nodes through a known node's interface.

The `Node` interface includes several operations for gathering information on nodes in the node graph. Two of these can be used as inspiration for equivalent operations handling discovered nodes: `getActiveNodes` and `setNodeEventHandler`.

The `getActiveNodes` operation returns a list of all the active nodes in the node graph. A similar `getDiscoveredNodes` operation could be a non-blocking expression that returns a list of all the geographically nearby nodes regardless of whether they reside in the current node graph or not (see figure 6.8). Although this approach lets us access the nodes at any time, figuring out if there is a new node nearby requires a polling process.

The `setNodeEventHandler` operation could also be used as a base for a `setDiscoveredNodeEventHandler` operation, running a callback function whenever a new nearby node is discovered or moves out of range (see figure 6.9). This solution removes the need for a polling process, but does not account for nearby nodes present

```
const Discover ← object Discover
    const home ← locate self

    process
        home$discoveredNodeEventHandler[object EventHandler
            export op nodeUp[n : Node, t : Time]
                % handle a new nearby node
            end nodeUp

            export op nodeDown[n: Node, t : Time]
                % handle the disappearance of a nearby node
            end nodeDown
        end EventHandler]
    end process
end Discover
```

Figure 6.9: Example of use of `setDiscoveredNodeEventHandler`

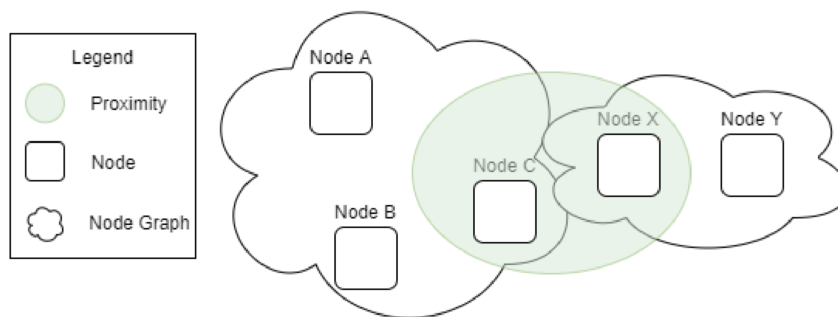

Figure 6.10: Two nodes in proximity of each other.

before any objects have subscribed to the event. As the `getDiscoveredNodes` and `setDiscoveredNodeEventHandler` operations compliment each other, both of these options should be implemented for a cohesive solution.

6.3.2 Consequences of Merging

Discovering a node is only useful if it can be made available to the discoverer. Objects should be able to move to the new node, and there should be some restrictions if the new node does not not want to host data from foreign nodes. In the following we discuss the details of whether disjoint node graphs needs to be merged, as well as provide a solution for how the merging can be made consensual.

Merging Node Graphs at Discovery

A way to instantly gain access to all of a discovered node's resources is to automatically join the discovered node into the node graph. However, if the discovered node also has multiple nodes in its node graph, the graphs will need to

be merged. Consider figure 6.10; When node C moves into proximity of node X, all the nodes A, B, C, X and Y should be merged into a complete graph. While this is a simple solution, it does not scale well and node graphs may merge where it is not intended.

Private Connection at Discovery

To prevent unnecessary connections, we could avoid merging the graphs, and instead create a private connection between the nodes in proximity. This would scale better, but creating incomplete node graphs has its own set of issues; If node B and C have a reference to a common object, and it is moved to node X, node B now has a reference to an object located on an unknown node. Furthermore, X could move the object to Y, leaving both B and C with an invalid object reference.

Consensually Merging Node Graphs

An additional problem with the two suggestions above is that neither of the nodes have any agency in whether they would like to establish a connection to the other node graph. A merging of the node graphs would need to be mutually consensual, similarly to the merging of object graphs outlined in 6.2.3. A possible solution to this is to only merge the node graphs when an object from one node graph is welcomed by another object from the other node graph. As welcoming an object requires a move operation, a limited private connection should be established for discovered nodes. Such a connection could for example cause move operations to fail if the moved object is not welcomed. Moreover, by using consensual connections the problem of scaling is greatly diminished as no unintended connections are established.

As node objects represents location in Emerald, and we are depending on the move operation to establish consent, something similar to a node object must become available through the private connection. However, we do not want to use the existing node object as it would enable too much access before consent is made. For example, using the regular `Node` interface, a malicious program could set the event handlers for discovering nodes to welcome any object, thus bypassing the consensual agreement altogether. A restricted variant of the `Node` type could still represent location, in addition to some basic operations, without exposing the entire interface. Additionally, using the `locate` operator on a restricted node should not escalate the privileges in any way, only return the restricted node itself.

Using this method, node C would receive a restricted node reference when it moves into proximity of node X. It would then move an emissary object to X. If the object is welcome on X, the two node graphs are merged into a complete graph. If it is not welcome, the move operation fails, the object remains on C, and the graphs remain disjoint.

It may not be appropriate for all nodes to be discoverable. Some nodes may need to stay hidden, or at least have no use of being discovered. Additionally, to support Emerald programs older than this feature, having nodes being discoverable as an opt-in option seems favourable. This can be achieved through an option when launching a node, similar to how distribution is optional.

```
const Connect ← object Connect
    const home ← locate self

    process
        home.mergeWith["129.240.118.130", 17099]
    end process
end Connect
```

Figure 6.11: Dynamically connecting to a foreign node using its identity

6.3.3 Dynamically Merging Node Graphs Using Identity

In cases where the identity of a node is known, it should not be required to move into its proximity in order to connect to it. The problem of establishing connections dynamically must already be solved when merging node graphs. Thus, adding the ability to connect to a foreign node given its identity should be trivial. This is useful in cases like the one described in 5.1 and illustrated in figure 5.1, and could also be implemented as an extension to the `Node` interface, see figure 6.11.

Connecting to a foreign node while a node is running will require both of the nodes' node graphs to merge into one complete graph. As there are currently no restrictions in place for connecting to a node given its identity, we argue that this mechanism does not require mutual consent. If this is wanted, one would also need to demand consent when connecting nodes at launch.

6.4 Complications of Merging

An invariant for Emerald node graphs is that they are complete; All nodes have a direct reference to all other nodes. When extending a node graph one node at a time, the single new node is at launch introduced to all the nodes in the node graph. However, if two nodes are launched and connect to two other nodes at exactly the same time, this invariant is broken. When a new node A connects to another node B, it receives a complete list of all nodes in B's node graph. A uses this list to become acquainted with each individual node in the graph. If two nodes requests this list at the exact same time, the list will not include either of the new nodes, and thus they will not become acquainted with each other. This is however extremely unlikely to happen, and is an edge case that has not been accounted for in the current implementation of Emerald. With the ability to merge node graphs directly within an Emerald program, this problem becomes more prominent, as two consecutive invocations of the `mergeWith` operation can increase the likelihood of performing concurrent merging on three disjoint node graphs. In the following we explore some options for possible algorithms dealing with this issue.

By only allowing a single merge at a time, this issue would solve itself. A call on `mergeWith` would simply block until the merge is complete, and all other merge-attempts from other processes would wait in line until each preceding merge is complete. However, implementing such a simple idea in a distributed system is not straightforward. First, one has to determine when a merge is complete. This requires

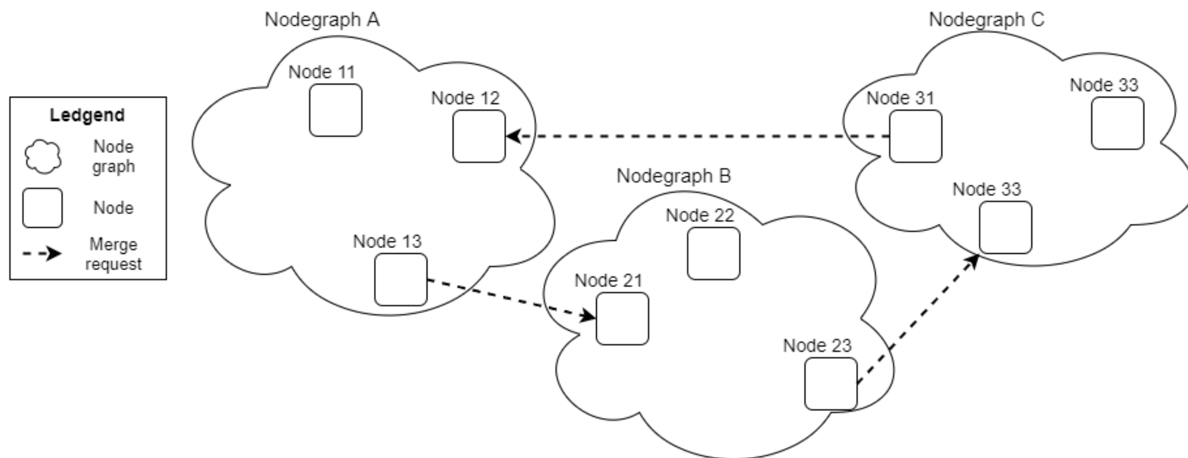

Figure 6.12: Three node graphs waiting for the next to complete its merging.

all nodes in both node graphs to know how many nodes they should connect to, and to send a confirmation message when they are done. Another, more prominent, problem is resolving potential distributed race conditions. If two nodes in the same node graph want to merge with two disjoint node graphs at the same time, each must coordinate with its node graph to stop any other node from both initiating a merge and respond to merge requests. This could potentially stop both of the nodes from merging, resulting in a deadlock. To solve this problem, the node graph should first agree on which merge that should be performed first, e.g. through using a unique ID per merge request and picking the lowest one during conflicts. Only after this is agreed upon by all the nodes in the node graph should the merge request be sent to the other node graph, on which other potential merge conflicts must also be resolved. While this solution resolves the problem of merge-deadlocks internally within the node graph, it does not remove the possibility of deadlocks completely. Figure 6.12 shows three node graphs that have all decided to merge with a node graph, and thus frozen all incoming merge requests. This will result in none of them completing their merge, all waiting for the next node graph to unfreeze. A way to temporarily cancel the merge and try again later at a random time (e.g. exponential backoff [14]) can fix the problem, but freezing a distributed system is in itself undesirable and should be avoided if possible.

As coordinating a set of nodes comes at a great cost, a better solution may be to rethink the complete node graph invariant. If each node acts independently with its own set of node relations, connecting two nodes would be trivial. A connection is simply a one-to-one relation between two nodes, and whether a node graph is complete or not is irrelevant. Figure 6.13 demonstrates an example of an incomplete graph. When node B sends a merge request to node X, only a single connection needs to be made to merge the two graphs. Any node must then be able to handle references to objects located on nodes to which there is no direct connection.

One way to implement this is to follow the chain of connections back to its origin whenever a remote invocation or move-statement is performed. If a node along the way becomes unavailable, a modified version of the Cascading Search Algorithm (*CSA*) [10, p. 83] can be used to recursively search through the node graph. The original CSA provides strong location semantics through performing a progressively reliable search, at the expense of performance. It will start by asking the last known location of the object, and follow a chain of forwarding pointers to the location. If a

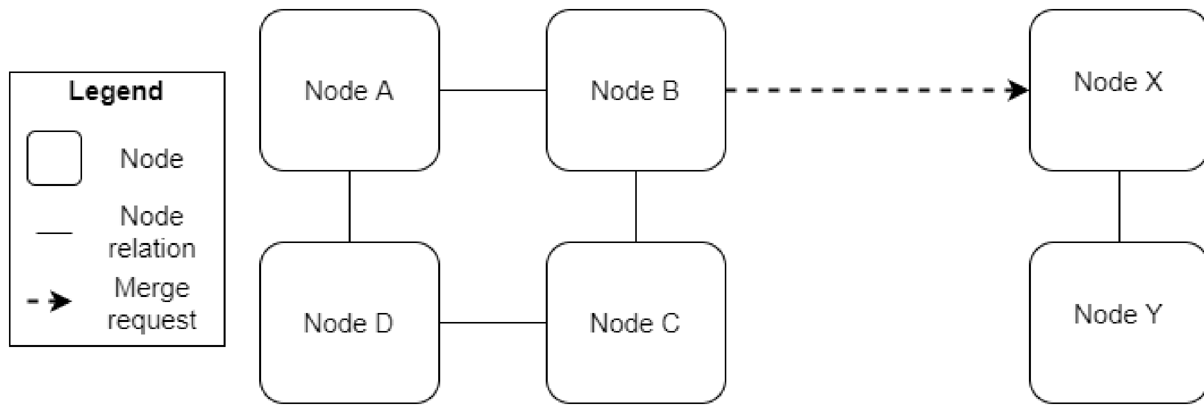

Figure 6.13: Two node graphs where two nodes are merging.

node along the way is unavailable, the CSA will progress to the next stage, and ask every node in the node graph if they have the object. Finally, if no answer is received, the CSA will ask every node for it's last known location of the object. If a new location is found in any of these steps, the CSA repeats.

The node performing the CSA assumes that it has a direct connection to the node currently hosting the object. Given our restructuring of node relations to avoid merge conflicts, this assumption is no longer true, and the algorithm may no longer provide strong location semantics. Assume that the nodes B and X from figure 6.13 have successfully connected. Object α and β are located on node D and they have a reference to each other. β is moved to node Y through A, B and X. If β invokes α, a connection is made to D, as β remembers D as α's last known location and α has not moved. If α invokes β, the CSA ensures that a path is found to Y, as long as A, B and X are available. However, if A becomes unavailable, there is still a valid path to Y through C, but the CSA will not find it. As node D's last known location of β is unavailable, D will ask all of its relations (i.e. C) if they host or know a path to β. As C never hosted β, it has no record of it, and the original CSA would determine the object as unavailable. Thus, we propose an extension of the algorithm, where all neighbours recursively repeats the last step. To avoid endless recursion, each initiated recursive search should be marked with an identity. Any duplicate recursive search requests are ignored. With this alteration, C will ask B, B will ask A (which is unavailable) and X, and X will ask Y, where the object is found.

While a modified CSA solves the problem of references to objects on unknown nodes, restructuring node relations to be on a "need to know" basis is a drastic change from the original Emerald implementation. Furthermore, this change would remove the fault-tolerant property of the distributed system, introducing possible single-point-of-failures and thus alter the required perspective of Emerald users to include information on how the node graph is structured. E.g. if Node X from figure 6.13 becomes unavailable after it successfully connected to B, node Y will also become unavailable for A, B, C and D and vice versa. Built in functions like `getActiveNodes` would also have to either change its behavior or obtain information on the node graph by recursively searching through it. The former alternative would drastically change a base philosophy of the Emerald programming language, and the latter would cause network overhead on something that used to be a local operation invocation.

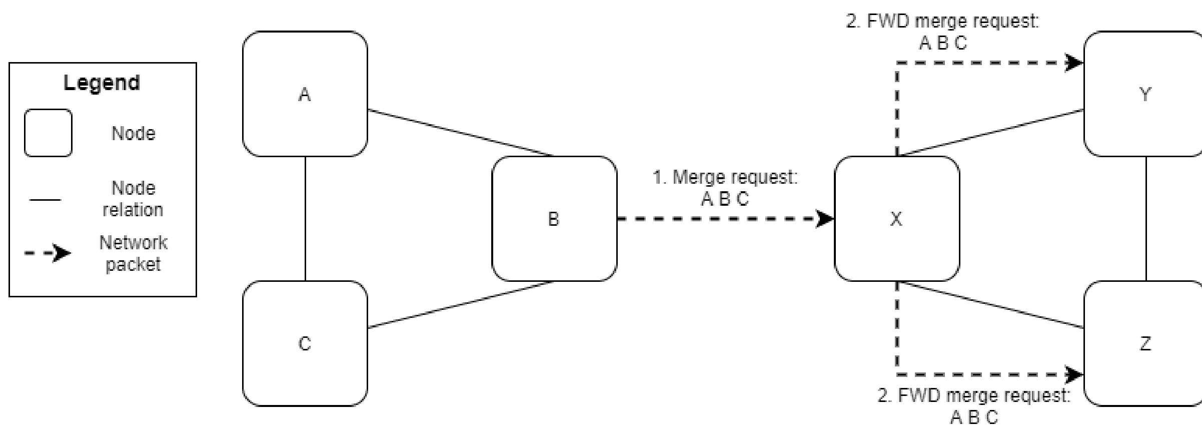

Figure 6.14: A best effort merge algorithm. All nodes from one node graph is sent to all of the nodes in the other. The recipients of the merge request will initiate a connection to the received nodes.

6.4.1 Best Effort Merging

We want to avoid both freezing Emerald node graphs and drastically change the Emerald programming language and its base philosophies. In the following we consider less extreme alternatives, where node graphs are generally thought of as, but not guaranteed to be, complete at all times.

In one such solution, merging node graphs would be a best-effort process, where all nodes at the time of the merge-request are introduced to the merging node graph. Figure 6.14 illustrates a merge request from node B to node X. Only three merge-specific network packets are needed to merge the two node graphs: B sends the nodes in its node graph to X, which becomes acquainted with these nodes. Furthermore, as X is the initial recipient of the merge request, it will also forward the request to all nodes in its node graph (Y and Z). Recipients of a forwarded merge request will not forward the request again. This eliminates the need for freezing the node graphs, and removes the possibility of a deadlock.

If two nodes initiate a merge at the same time, not all nodes in the resulting node graph would become acquainted. This is illustrated in figure 6.15. Here node P and B attempts to merge their respective graphs with XYZ at the same time. This leads to a state where the nodes XYZ know all nodes, while OPQ and ABC do not know about each other. However, nodes would also be capable of locating objects residing on nodes to which they have no direct connection through the modified CSA.

This solution does not solve the issue of guaranteeing a complete set of nodes through the `getActiveNodes` operation, as concurrent merge requests may result in incomplete node graphs. It would still be possible for programmers to ensure that they are working with a complete node graph, but this would require them to regularly search the node graph recursively. This moves the responsibility of maintaining the underlying node graph from the Emerald virtual machine to the programmer.

6.4.2 The Node Synchronization Algorithm

To ensure that one or more concurrent merges eventually results in a single complete node graph, we propose an algorithm that provides strong semantics for complete

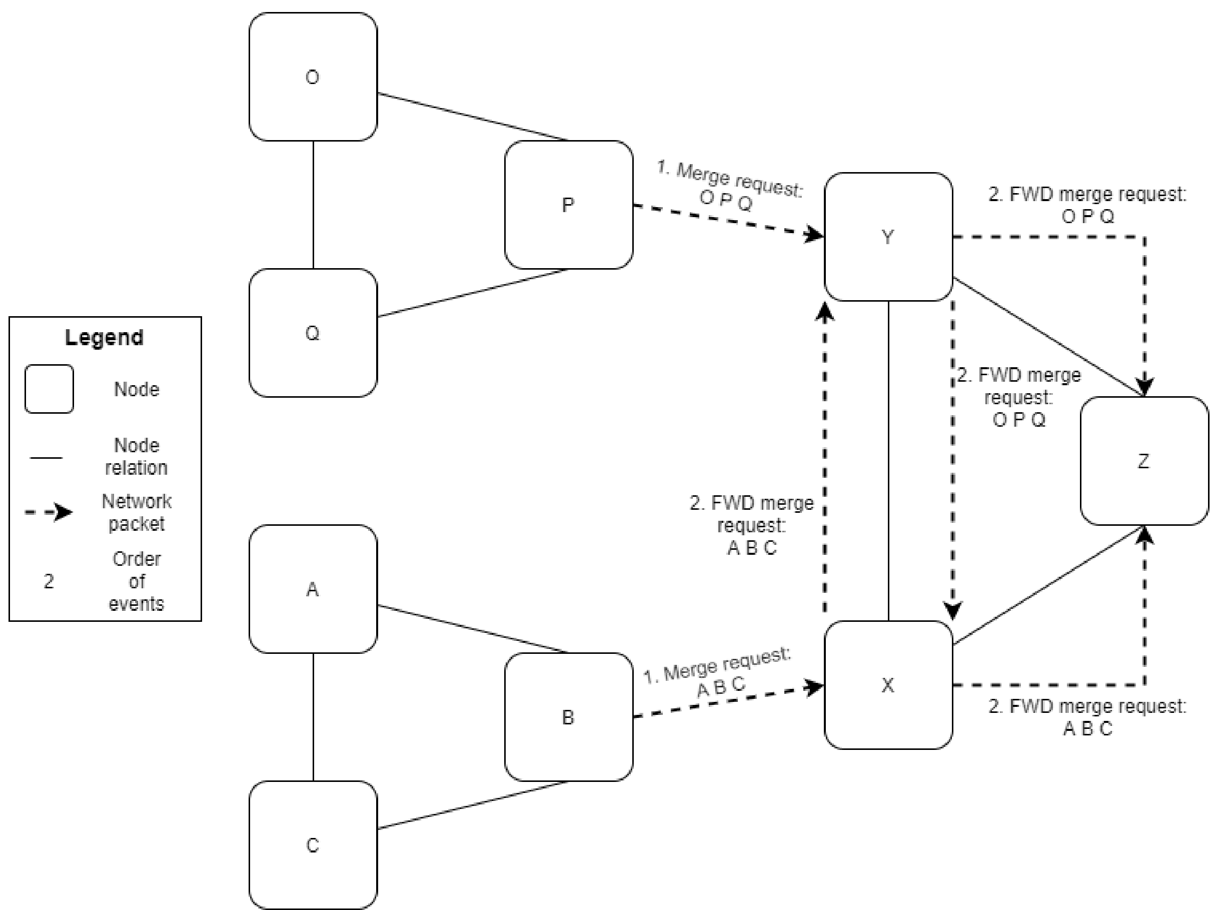

Figure 6.15: An illustration of a scenario where the best-effort merge algorithm results in an incomplete node graph.

merging. We call this algorithm the *node synchronization algorithm*, and it has the following steps:

- When merging, a merge request packet is sent containing all known nodes.
- When receiving a merge request packet, connections are established to all new nodes and a node synchronization message (*NodeSync*) is sent to all nodes.
- When receiving a NodeSync, connections are established to all new nodes.
  - A NodeSync is sent in reply if at least one known node is missing in the received message.
  - A NodeSync is sent to all new nodes received from the original message.

With this algorithm, the problem illustrated in figure 6.15 is solved. Instead of X forwarding the merge request to its original node graph, it now sends a NodeSync to all of the nodes ABCYZ. Similarly, Y sends a NodeSync to OPQZX. Any disagreements between nodes is resolved by replying with a NodeSync containing the updated node graph. For example, Y will disagree with X, as Y also know about OPQ (and vice versa). Y proceeds to send a NodeSync to X containing all the nodes in the final node graph. Additionally, A will upon receiving a NodeSync not only connect to Y and Z, but also send them a NodeSync. This will result in Y and Z realizing that A does not know about OPQ, and will reply with a NodeSync containing the complete node graph. When all disagreements are resolved, the node graph is complete.

The node synchronization algorithm ensures that the sender and receiver both have a unified view of the node graph, and any additions to the receiver's node graph will themselves become a receiver of a NodeSync. The first two steps of the node synchronization algorithm are required if one tries to make an existing incomplete node graph complete, as the original recipient may know about the same set of nodes as the sender. When merging disjoint node graphs, these steps are not required to end up with a complete node graph, as the base premise of the merge is that their set of nodes are different. However, they will speed up the process as in most cases all nodes can become acquainted simultaneously.

6.5 Future Work

Permanently merging disjoint node graphs may not always be desired. If a server expects to connect to many incoming clients, the resulting node graph will quickly become larger than necessary, and all clients will know about each other. The solution for merging node graphs does not provide for or anticipate a way to split node graphs. We consider the problem of splitting node graphs to be a separate issue that could be solved in a future project.

6.6 Summary

In this chapter we have proposed and detailed a new language mechanism for the Emerald programming language that introduces the ability to merge both object and node graphs. This language mechanism allows disjoint programs to communicate,

whether they are in the same node graph or not. The WELCOME language mechanism includes a new dedicated expression, as well as some additions to the `Node` interface and object constructor. The `welcome` expression blocks until a welcome object conforming to the expected type is moved onto the same node as the welcoming object, and returns a reference to that object. Objects can only be welcomed if they are welcomable, which is specified upon their creation. The `Node` interface is extended to support event handlers for handling the discovery of nearby nodes. Nodes may be discovered by other nodes in its proximity if configured for it, and node graphs merge if objects are exchanged using the `welcome` expression between them. Using the node synchronization algorithm, concurrent node graph merging may result in a temporary incomplete node graph, and for the Emerald programming language to run in such an environment, a mechanism like the modified CSA is required.

Chapter 7

Implementation

7.1 Introduction

To demonstrate the solution described in chapter 6, we have implemented a working prototype based on an existing implementation of the Emerald programming language. To implement the changes needed for the WELCOME language mechanism to work, we first needed an understanding of the base implementation, i.e. how the code is systematized and compiled.

There are multiple implementations of the Emerald programming language, and the original was implemented as a compiled language with a performance comparable to C [1, p. 11]. The language was later implemented as an interpreted language and this is the implementation we have been working with.

The implementation is split into two separate parts, a compiler and an interpreter. The compiler compiles Emerald source code into a bytecode representation and stores it into a separate file. The interpreter interprets and runs the bytecode created by the compiler. For the WELCOME language mechanism to work, both parts needed to be altered.

Although the base implementation was written with an effort to support as many CPU architectures and operating systems as possible, we have only ensured support for systems running Linux with GNU.

The prototype consists of an altered Emerald compiler and virtual machine, and supports the creation of welcomable objects and the `welcome` expression. Additionally, it features an extended built-in `Node` interface, which includes the `mergeWith`, `getDiscoveredNodes`, `getAllDiscoveredNodes` and `setDiscoveredNodeEventHandler` operations. It also enables the possibility to launch discoverable nodes, to discover foreign nodes on a local network, and merge disjoint node graphs by moving emissary objects to discovered nodes.

7.2 Getting to a Bootstrapping Compiler

Our first goal in changing the Emerald compiler was to recreate the current compiler to familiarize ourselves with the process of compiling the compiler. The source code for the compiler is itself written in Emerald, and thus compiling the compiler requires a working compiler. Before we started on this project, the Emerald compiler was last compiled in 2007 and since then some undocumented changes were made.

This, accompanied by undocumented scripts, proved building a compiler that could compile itself difficult.

On the Emerald GitHub page, we found five different versions of the Emerald source code and varying instructions for how to install and compile the Emerald virtual machine. However, instructions for how to compile the compiler were limited. Given the information we had, we were not able to compile a compiler with satisfactory functionality.

By reaching out to some of the creators of Emerald, we obtained a working version of the source code along with instructions on how to compile the compiler. The new version and its instructions have been documented and published on the Emerald GitHub page [13].

7.2.1 Understanding the source code

To alter the source code we first needed to understand it. Although documentation of the code is limited, the whole project is written in two programming languages that we are familiar with; C and Emerald. A substantial amount of time went into reading and understanding the source code, and although a lot was learned this way, it proved impossible to get a complete overview of the whole project within our time frame using this approach. Therefore, we also used a process of trail and error, changing values in one end to see how it affected the other. We also limited our focus to understanding the relevant components rather than trying to achieve a comprehensive understanding. For the `welcome` expression, this included process scheduling and the structuring and movement of objects. For discovering and merging nearby nodes, we focused on the node relation infrastructure and the implementation of the event system.

7.3 Implementing the `welcome` Expression

The `welcome` expression takes an abstract type as its argument and blocks until a welcomable object of the specified type is moved onto the same node. A reference to that object is then returned. To implement this new expression, we first had to add support for the `welcome` keyword in the compiler, and make sure that the return value remained consistent with the type checking system. After making the Emerald compiler successfully parse the `welcome` expression, the Emerald virtual machine had to be modified to reflect the expected behavior. In the following we discuss the details related to implementing these additions, as well as present the challenges we encountered along the way.

7.3.1 Changing the compiler

The Emerald compiler parses the source code into tokens, and generates a syntax tree based on a YACC [9] grammar file. On completion, it goes through the main stages of the compilation in the following order:

1. Remove sugar
2. Define symbols

3. Resolve symbols
4. Assign IDs
5. Find manifest objects
6. Type assignment
7. Type checking
8. Assign IDs again
9. Perform necessary allocations
10. Code generation
11. Environment exports

We found two of the stages listed above relevant to us: type assignment (6) and code generation (10). The `welcome` expression should only be able to take an abstract type as its argument, and the type of the returned object should conform to the argument type. We also needed to add a new bytecode-instruction so that the Emerald virtual machine was able to recognize the new expression.

The first change the compiler needed was a way to recognize our new keywords. The compiler holds a list of all reserved keywords and symbols in the Emerald programming language. After adding our two new keywords `welcome` and `welcomable`, the compiler recognized our keywords. When compiling a program using either one of the keywords we then got a syntax error instead of an error message claiming that an unidentified token is present.

Next, we wanted the compiler to recognize the new syntax described in 6.2.2. The grammar is specified in a YACC-file that needs to be altered when adding syntactic elements to the language. We added a new option in the expression rule to include the unary expression `welcome` (shown in 6.3).

We wanted to make sure that any argument given to the `welcome` expression is guaranteed to be an abstract type. Next, we needed to specify the abstract and concrete type of the return value of the `welcome` expression. Determining the abstract type of the return value was easy, as it should be the same as the argument. However, deciding the concrete type proved a problem as it cannot be determined during compile time. Instead, a placeholder value is returned and later overwritten by the Emerald virtual machine.

Lastly, compiling the `welcome` expression should produce bytecode for both the `welcome` keyword and the argument. We added the new bytecode `WELCOME`, and expanded the code generation to produce this bytecode along with the bytecode for the argument. This way the Emerald virtual machine has all the information required to handle the `welcome` expression.

The `welcomable` Keyword

Implementing the `welcomable` object prefix required much of the same steps as those listed above. The YACC-file specifying the grammar was updated to reflect the grammar shown in figure 6.7. From the list of compilation stages above, `welcomable`

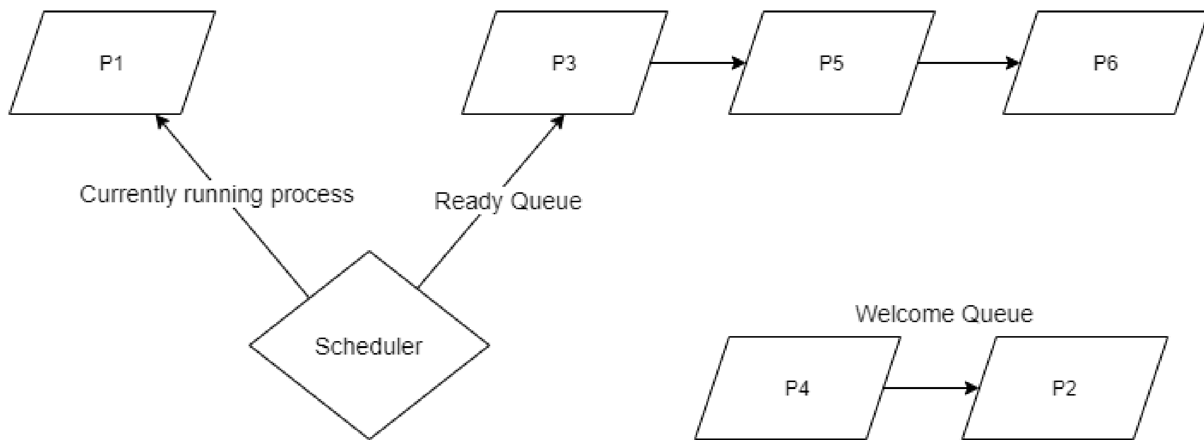

Figure 7.1: The scheduler and the welcome queue. Scheduled processes are moved out of the ready queue.

required alteration of define symbols (2) and code generation (10). We altered the object creation by adding a test controlling that no objects are both immutable and welcomable at the same time.

When working with the Emerald virtual machine we found that the welcomable attribute could be stored as a flag in the concrete type of the object. This is described in 7.3.3. As the concrete type is static and created on compile time, the flag can be set by the compiler and added to the bytecode representation of an object. As the immutable attribute is created and stored in the same way, we used it as a reference point when implementing the welcomable attribute.

7.3.2 Changing the Emerald Virtual Machine

With the changes made to the compiler described above, the bytecode WELCOME is produced for each `welcome` expression. We could then define the behavior of the WELCOME instruction in the Emerald virtual machine; The process should block until a welcome object is moved onto the local node, upon which a reference to that object should be returned. In the following we describe which changes needed to be made to the Emerald virtual machine to accomplish this goal.

To make the Emerald virtual machine recognize the new bytecode produced by the compiler, we needed to add it to the interpretation loop. The Emerald virtual machine runs a number of processes, where each process has its own stack, stack pointer, program counter etc. A scheduler is maintaining the processes in a ready queue, and removes one at a time to be run by the interpreter. When a process is finished, it is discarded and the next process from the ready queue is retrieved. The scheduler is visualized in figure 7.1. If a process should block for some time, it is temporarily stored in a separate queue and reinserted into the ready queue when the time is right.

When interpreting a WELCOME instruction, we know that its argument has been pushed on the stack. As the `welcome` expression should be blocking, we want to remove the process from the ready queue of the scheduler. This is done by simply returning from the interpretation loop. However, as we want to resume the process at a later point, we need to store the process. To do this we created a dedicated queue called the *welcome queue*, located in a globally accessible file. Then, all we needed to do

```
void handleMoveRequest(...):
    acquire object;
    acquire concrete type;
    if object is welcomable:
        for each process in welcome queue:
            if process welcomes object:
                push object on stack;
                push concrete type on stack;
                schedule process;
```

Figure 7.2: Pseudocode for the relevant parts of the `handleMoveRequest` function

```
case WELCOME:
    read argument from stack;
    push copy of argument on stack;
    add process to welcome queue;
    return;
```

Figure 7.3: Pseudocode for the WELCOME case in the interpretation loop

was to retrieve and resume the process at the correct time.

When handling incoming objects, the virtual machine checks if the object conforms to the expected type of any of the processes in the welcome queue. If the type of the incoming object conforms to the expected type, the process is rescheduled back into the ready queue. This way, multiple blocking processes can be unblocked simultaneously. Lastly, a reference to the incoming object should be returned when the process is unblocked, and this is done by pushing the object and its concrete type onto the process' stack. This process is demonstrated in figure 7.2.

There was one more case to consider when implementing the `welcome` expression, and that was moving a welcoming object. As a welcoming object is an object with a process that is currently in the node's local welcome queue, the object should continue blocking on the destination node and be inserted into its welcome queue. The Emerald virtual machine already knows whether an incoming process should run or not based on its status, so we only need to insert it into the welcome queue if appropriate. To find out if an incoming object is blocking on a `welcome` expression, we check its previous instruction pointed to by its program counter. If this instruction is a WELCOME instruction, it is moved into the new node's welcome queue. When moved, the object is also removed from the welcome queue.

As mentioned, we used the stack of a given process to store the argument. However, when moving a process, the Emerald virtual machine seems to expect the size of the stack to be divisible by 8. A reference to an abstract type requires 4 bytes, thus the argument may not always be sent. In this case, we used a workaround where we push a copy of the argument reference on the stack. This ensures that if the object is moved, the argument is always sent along with it. Figure 7.3 demonstrates this workaround.

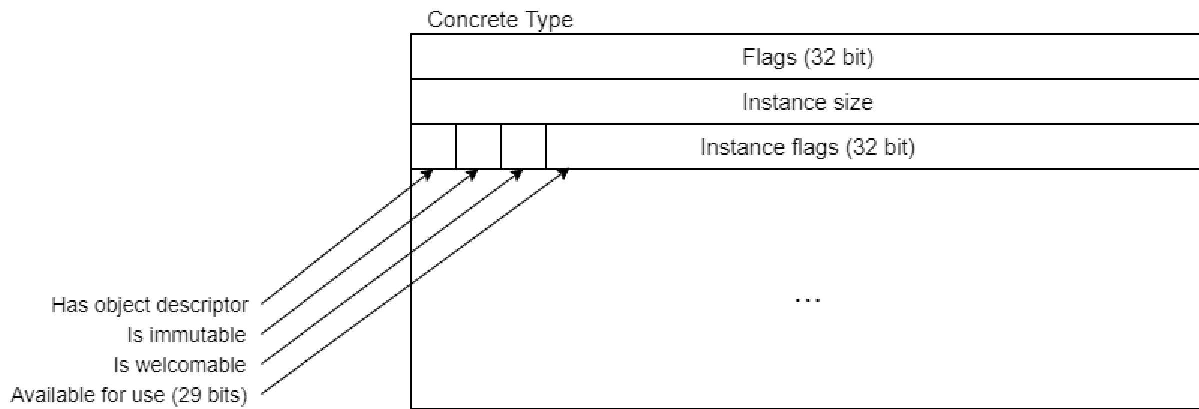

Figure 7.4: Layout of the Concrete Type object.

7.3.3 Implementing Welcomable Objects

As described in 6.2.3, not all objects should be able to trigger the `welcome` expression, only those created using the `welcomable` prefix. This is beneficial performance-wise as only objects intended to be used with the `welcome` expression needs to be type checked with the welcome queue, but makes the implementation slightly more intricate. Any moved object must be checked for whether it is welcomable or not by its receiving node, and trigger a search through the welcome queue if it is.

The concrete type provides a description of the data fields and the code of an object, as well as some attributes that never change for any instance of the concrete type. As the welcomable attribute is static and only determined at an objects creation, it seemed like a natural choice to store this property in the concrete type. Figure 7.4 shows the location of the welcomable-bit used to indicate welcomability. The first 32 flag-bits of the concrete type cannot be used, as the concrete type is itself an object and this field is reserved for all objects. Thus, the instance size is the actual first unique data field of the concrete type. The instance flags field previously only hosted two flags, leaving plenty of room for future attributes.

7.4 Implementing Node Graph Merging

Emerald node graphs should be able to merge using two different techniques: by using the `mergeWith` operation, and by discovering a node and sending an emissary object. As merging two node graphs is the base of both of these techniques, we started with implementing the `mergeWith` operation. After successfully merging two node graphs, we moved on to implementing a discovery mechanism for nearby Emerald nodes, and extended the event-handler system already present in Emerald to include the event of discovering new nodes. Lastly, we made sure that interaction with a discovered node was distinct from a regular node, with moving a welcomable object to it as the only allowed operation.

7.4.1 The `mergeWith` Operation

The `mergeWith` operation is not a syntactic extension like the `welcome` keyword, but an extension to the interface of the built-in `Node` object. Built-ins are objects that are

```
export operation mergeWith [ip : String, port : Integer]
   primitive "SYS" "JMERGEWITH" 2 [] ← [ip, port]
end mergeWith
```

Figure 7.5: Implementation of the `mergeWith` operation

accessible in every Emerald program, like e.g. `String`, `Array`, `Time` or `Node`. Although the functionality is added to the node object, the actual implementation is written in the Emerald virtual machine. This is possible using the `primitive` statement in the Emerald programming language, which allows us to write bytecode to the instruction stream and push arguments onto the stack directly from an Emerald program. Using the bytecode `SYS` we are able to make a system call. System calls are placed in a list and indexed through a constant given in the primitive statement. The `SYS` bytecode also expects an integer stating how many arguments are to be pushed on to the stack. We defined a unique system call for the `mergeWith` operation and mapped the bytecode `JMERGEWITH` to it. The only required extension to the built-in `Node` object is the single line operation shown in figure 7.5.

The `mergeWith` function expects a hostname or an IP address (as a string) and a port to be present on the stack. If possible, a TCP connection is established with the node matching the identity given. The Emerald programming language was made so that it would extend the node graph whenever the Emerald virtual machine learned about a new node. When a new connection between two nodes is established, they instantly exchange node information. We call this exchange the *Emerald handshake*. After a successful Emerald handshake, it is assumed by both parts that all subsequent messages will be of an Emerald-specific message type and be preceded by an Emerald-specific header. The first message of a connecting node is always an *echo request*, in which the connecting node requests a complete list of all nodes in the graph it is connecting to.

When implementing the `mergeWith` system call, we created two new message types inspired by this request: the *merge request* and the *node synchronization*. When merging, a merge request message containing all the nodes in the senders node graph is sent. The recipient then sends a node synchronization message to all nodes in both node graphs. The receiver of a node synchronization message updates its node graph accordingly, and sends another node synchronization message to all previously unknown nodes. If the sender's node graph is lacking, a node synchronization message is sent in reply. This behavior is according to the node synchronization algorithm presented in 6.4.2.

7.4.2 Node Discovery

To be able to connect to unknown services, we have implemented support for discovering nearby nodes. When launching a node, it is now possible to use the option '-D' to make the node discoverable. All discoverable nodes will periodically send out a broadcast message to all of its available broadcast addresses. A list of available broadcast addresses is updated regularly, and the discovery socket is marked reusable so that multiple nodes hosted on the same computer can broadcast.

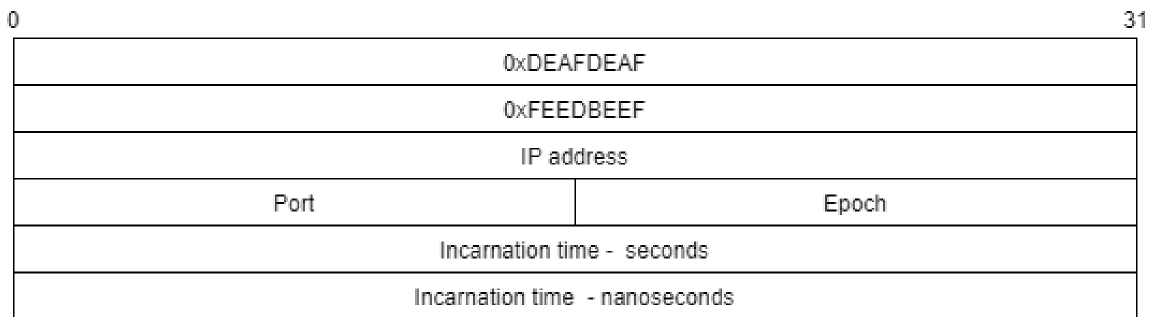

Figure 7.6: Format of the broadcast message.

The broadcast message itself contains information about the broadcasting node, and is visualized in figure 7.6. The message begins with a signature used in all Emerald messages, the 4 byte integer `0xDEAFDEAF`, and is followed by a unique discovery signature `0xFEEDBEEF` that is only used for these broadcast messages. Data fields with information about the broadcasting node follows the signature. As all data fields have a set size, every packet has a fixed length of 24 bytes.

A socket dedicated to reading the broadcast messages is initialized upon launching a node, and maintained along with the regular network sockets used by Emerald. To distinguish the behavior on different sockets, each one is mapped to its own handler-function. We created a handler that ignores any messages originating from a node within its current node graph. Each node in the node graph is represented by a structure called a *node record*, which has a reference to the remote node object, a boolean describing whether the node is declared unavailable or not, as well as the required information for contacting the node. All available node records are contained in a linked list, and all non-present objects are directly linked to the node record of their last known location. When receiving a broadcast message from an unknown node, a similar structure is created and put in a separate linked list. We call this structure a *discovered node record*, and it contains a regular node record and the time since the last received discovery message from that node. Whenever a new discovered node record is created, a discovered node event is triggered. If no broadcasting message is received from a discovered node within a set time interval, a discovered node down event is triggered and the node is declared as no longer discoverable.

7.4.3 Extending the Emerald Event System

The Emerald programming language already supports one pair of events for keeping track of changes in the node graph. The `setNodeEventHandler` operation is implemented as a part if the interface to the built-in `Node` object. The operation takes an object conforming to the `Handler` type as parameter. The `Handler` type requires the operations `nodeUp` and `nodeDown`, which are triggered when a new node presents itself and when an existing node becomes unavailable respectively.

To implement the handler for the discovered node events, we used the same approach and extended the `Node` object's interface by adding a `setDiscoveredNodeEventHandler` operation. Like the `setNodeEventHandler` operation, it takes an object conforming to the `Handler` type as a parameter, and triggers its `nodeUp` and `nodeDown` operation when a discovered node appears and disappears respectively. The `Node` object maintains a list of `Handler` objects that should be called when an event triggers.

```
const Runnable ← typeobject Runnable
    op run[Handler]
end Runnable

export operation discoveredNodeUp [n : Node, t : Time]
    const runnableObject ← object runnableObject
        export operation run[ h : Handler ]
            const invokeUp ← object invokeUp
                process
                    h.nodeUp[n, t]
                end process
            end invokeUp
        end run
    end runnableObject

    self.fireEvent[EventType.DISCOVERED_NODE_EVENT, runnableObject]
end discoveredNodeUp
```

Figure 7.7: The Runnable typeobject and the implementation of the `discoveredNode-Up-Event`. All event related operation called by the Emerald virtual machine calls the `fireEvent` operation, which invokes the `run` operation with the correct handler.

We generalized this list to contain tuples of an *event type* and a handler. The event type is a constant that describes the kind of event that triggered, allowing all handlers to be kept in the same list. When an event triggers, a corresponding operation in the `Node` object is invoked by the Emerald virtual machine. This operation goes through the list of events, and fires any that has the correct event type associated with it.

To remove an event handler, the `Node` object provides the operation `removeNode-EventHandler`, which takes an event handler as parameter and removes it from the list of handlers. This operation remains unchanged, and supports both discovered and regular node events.

Every event handler runs as a separate process. This is done by creating an object conforming to the `Runnable` interface (see figure 7.7). When its `run` operation is invoked, a process is created and runs one of the handler's operations.

If any future events were to be implemented, one would only need to create a new constant for the event type and an operation to trigger it. However, as the current `Handler` type is specific to events related to node graph changes, any new type of event would require a restructuring of the `Handler` type.

7.4.4 The Emissary Move Request

Using the `discoveredNodeEvent` handler, we can obtain references to discovered nodes. Emissary objects may be moved to a discovered node and, if welcomed by a process, the node graphs will merge.

When moving an object in Emerald, the last known location of the destination object is retrieved, and the object is moved to that location. As every non-resident object is linked with the node record of its last known location, we similarly linked the

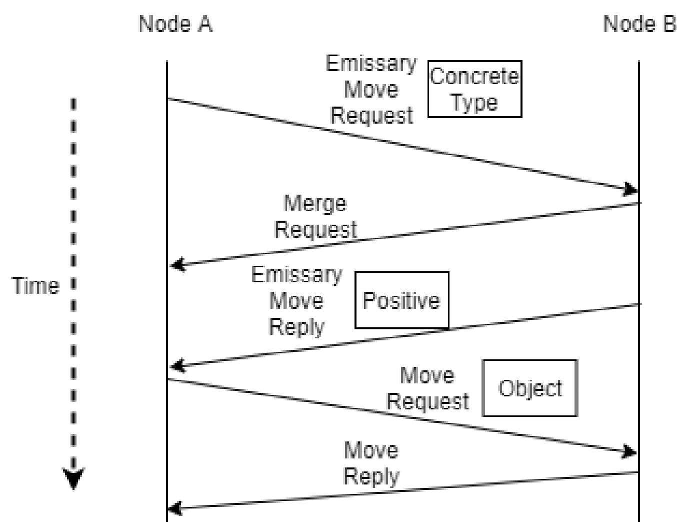

Figure 7.8: The Emissary Move protocol.

discovered node object to its corresponding discovered node record. We can use the fact that the discovered node record is not present in the list describing the current node graph to distinguish move operations to nodes inside and outside the node graph.

The function handling move operations to discovered nodes queries the destination node for whether there exists a process in its welcome queue that would welcome the object, if moved. The response is a binary, positive or negative. If positive, the object is moved through the regular move function and the node graphs are merged. If negative, the object remains unmoved.

As only welcomable objects may be welcomed, any unwelcome objects are immediately ignored before any network packets are sent. To query the discovered node, we created a new set of Emerald message types: the `EmissaryMoveRequest` and `EmissaryMoveReply`. The packets are similar to regular move requests and replies, with the only difference being the message type. When sending an emissary move request, the concrete type is sent to the destination node, which handles the packet using a dedicated callback function. The reply is merely a message header of type `EmissaryMoveReply`. The binary answer is a flag in the header. If at least one process welcomes the object, the answer is positive and the node initiates a merge request. As the emissary message types don't require many header options, the options for the potential move request are stored in the headers of the packets throughout the interaction. When receiving a positive `EmissaryMoveReply`, a regular move request can be performed using the information from the received header. This way, no header options for the resulting move need to be temporarily stored while waiting for a reply.

Figure 7.8 shows the interaction described above. The move request could potentially be omitted if the emissary move request also moved the object, and sent it back if the object was not welcomed. However, when an object is moved, all subsequent invocations on that object is performed as remote invocations to the node it was sent to. If the object's new location is a discovered node, invocations on this node are invalid (see 7.4.7). A solution to this could be to not remove the object from the original node, but this could result in duplicates of the same object existing on different nodes. We consider sending an extra network packet easier than solving the

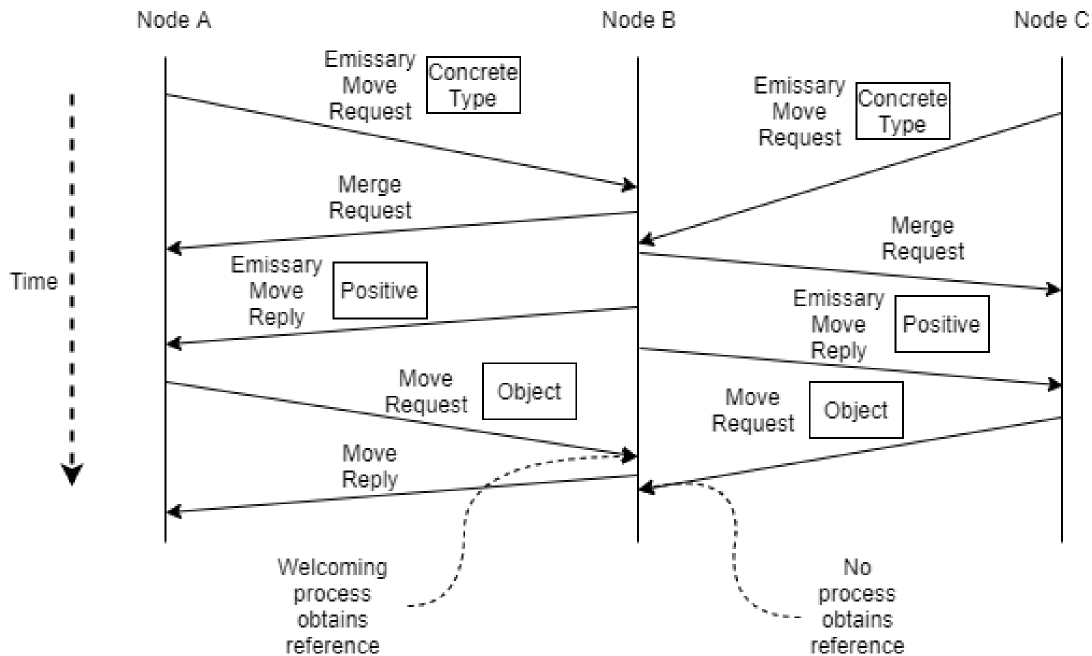

Figure 7.9: A visualization of a potential concurrency issue in the Emissary Move protocol.

issues related to duplicate objects.

Concurrency Issues

Although we consider the emissary move visualized in figure 7.8 to be the better solution, it does pose a potential concurrency problem. Figure 7.9 illustrates this issue, where two nodes *A* and *C* each perform an emissary move request towards node *B* at the same time. *B* has only one process in its welcome queue, and the emissary objects of *A* and *C* are both welcome objects on *B*. As illustrated in the figure, both *A* and *C* will receive a merge request, though only one of the objects will actually be welcomed.

A possible solution for this problem is to reserve the welcoming process by assigning an object ID to it when receiving an emissary move request. This way, no subsequent emissary move requests would receive a positive reply based on the same welcoming process, and the reserved process would only unblock when the object with the expected object ID arrives. In addition, the time of the reservation could be stored along with object ID so that the reservation could time out if the promised object is not provided within a certain time interval. As the concurrency issue described is unlikely to occur, we have not implemented this solution in the prototype.

7.4.5 Silent Node Connections

Before any messages can be sent to an unknown node, a TCP connection is made and the Emerald handshake is exchanged. This handshake will trigger a node up event, and create a node record for the new node. When sending an emissary object however, we do not want to alert the target node of our presence before the object is welcomed, as the emissary move request is not guaranteed to result in a merge between the node

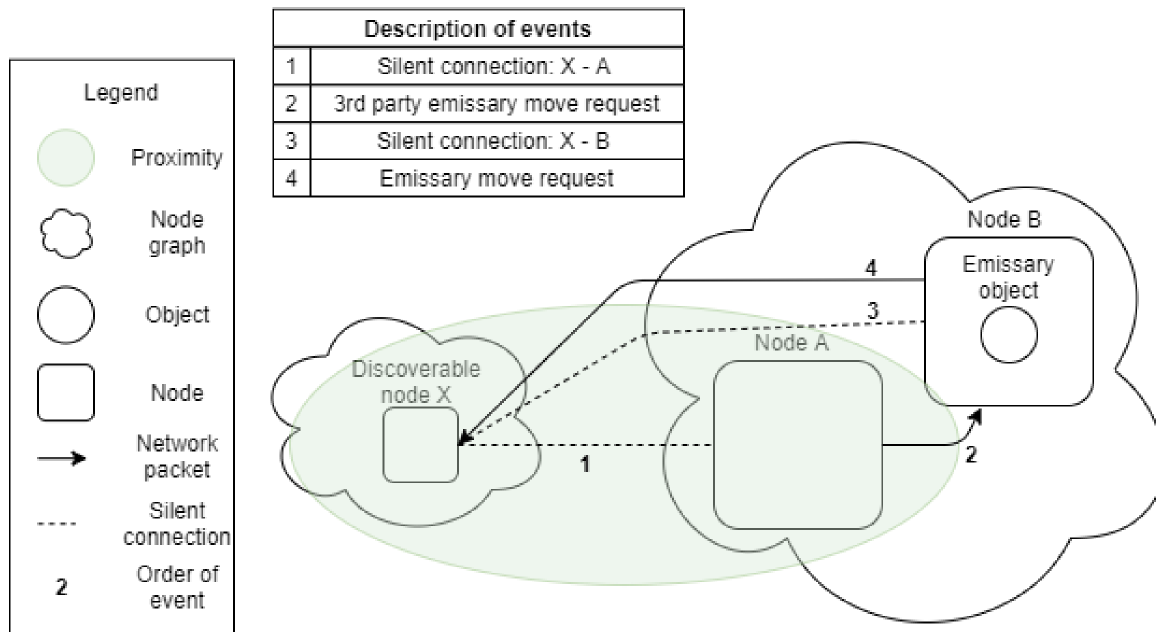

Figure 7.10: A 3rd party emissary move.

graphs. Therefore we added a new element to the Emerald handshake, so that nodes can communicate without being a part of each other's node graph.

The Emerald handshake consists of exchanging the IP address, port, epoch and the user ID of the nodes. Instead of adding a new data field to the handshake, we reserved a bit from the 32-bit user ID-field that is highly unlikely to be used in any real world scenario. If the bit is set, it marks the connection as being in *silent mode*. We used the 31st bit in the user ID-field, as this would still allow for over one billion unique users per node, while still preserving its signed property. User ID used to be a 16-bit value on Unix and Unix-like systems, and it seems most user IDs are still within this range. On Windows the user ID is always set to 5001. When a node receives a handshake with the silent mode bit set, it will not create a node record for the new connection, and will not trigger a node up event. When two node graphs are merged, all relevant connections currently in silent mode are changed to regular connections, a regular node up event is triggered per new node and the necessary node records are created.

All distributed operations in Emerald are able to handle destinations to which there are no connection by initiating an Emerald handshake. No Emerald specific messages sent over a silent connection will cause a node up event to trigger, but if a message is sent before the silent connection is created, a regular connection is established instead. Therefore, we had to ensure that all messages to a discovered node followed an emissary move request, as this is the only place silent connections are created. Silent connections could also be created upon discovery of a nearby node, but this would cause unnecessary connections.

7.4.6 3rd Party Emissary Move

An emissary move request is only sent if the emissary object is present at the location of the sender. During a regular move, a `Move3rdPartyRequest` is sent instead if the object

is not present. A message is sent to the node hosting the object with a request to send it to the destination. From this point, the hosting nodes perform a regular move request. Sending a `Move3rdPartyRequest`-message during an emissary move is problematic, as the node hosting the emissary object may not have discovered the target node and thus have no discovered node record describing it. Instead of creating another new message type, we set a flag in the message header denoting that the target destination is a discovered node. When receiving a `Move3rdPartyRequest` with the discovered node flag set, a silent mode connection is created before performing an emissary move. Figure 7.10 illustrates this process, where a node A wants to send an emissary object located on B to the discovered node X. Even though B creates a connection to X, it does not create a discovered node record and does not trigger a discovered node up event. If the emissary object is not present on B, it would forward the request to the emissary object's last known location without creating a silent connection to X.

7.4.7 Discovered Node References

A discovered node event provides a reference to a discovered node object. However, the node it refers to is not part of the node graph and invoking the object should not be possible. The only real use of the reference is providing the location of the discovered node to be used in move, fix or refix statements. When the discovered node's node graph is merged, the `Node` object's associated discovered node record is replaced by a regular node record, and all normal interactions with the object is possible.

To prohibit remote invocations on discovered nodes, we check the associated node record of the object before the invocation is made. If the node record is a discovered node record, an unavailable exception is raised instead. There is also a set of bespoke expressions in the Emerald programming language that needed to change their behavior when evaluated with a discovered node as its operand. The `isfixed` and `unfix` expressions are both distributed operations, and messages to discovered node records must be sent over silent connections. Instead of ensuring a silent connection for the `isfixed` and `unfix` expressions, we instead simplified the expressions so that `Node` objects always evaluate as fixed, and cannot be unfixed. This is in accordance with the Emerald programming language report [8, p. 40], but was not previously implemented. Furthermore, we changed the locate expression, so that whenever the argument is a node reference, the node reference itself is returned before the distributed search algorithm is initiated. This change was safe to make because node objects represent location in the Emerald programming language, and locating a location would be redundant.

7.4.8 Modified Cascading Search Algorithm

An edge case when merging node graphs with both the `mergeWith` operation and when using emissary objects is that concurrent merging may cause temporary incomplete node graphs. The modified Cascading Search Algorithm (CSA) described in chapter 5, is a solution that deals with the side effects of incomplete node graphs. As this is an edge case likely to require a lot of work and restructuring in the Emerald virtual machine, we did not prioritize implementing it in the prototype. Although it is theoretically possible to obtain references to objects residing on unknown nodes, it

is very unlikely to happen, as node graphs remain incomplete for a very short period of time.

7.5 Limitations

As previously mentioned, the WELCOME language mechanism lacks implementation for the modified CSA (see 7.4.8) and for fixing the concurrency issues related to emissary move requests (see 7.4.4). Additionally, it is also limited in its usability by the current implementation of the Emerald compiler. When compiling Emerald programs, the concrete type of every object is given a unique object ID (OID). These OIDs are not randomly generated, but are starting from a predetermined number. If two Emerald programs are compiled in separate environments, they may share OIDs for different concrete types, resulting in undefined behavior and a potential node crash if they are moved to, or run on, the same node.

The compiler recognizes equal manifest expressions and stores them in a single executable file, even if they occur in multiple source files. If such files are run separately, some files may be missing crucial parts of their executable code. Thus, files that are compiled together should also be run together.

The above problems leaves us with a conundrum. Files must be compiled together to be able to run on the same node, and files that are compiled together must be run together. However, for the WELCOME language mechanism to reach its full potential, it should handle programs that are compiled separately and that run on different nodes. With this limitation the WELCOME language mechanism is only useful when it is needed by two programs that are compiled and run together. In such scenarios the programs may already have references to each other using the `export` statement, leaving the WELCOME language mechanism redundant.

Fortunately, the Emerald compiler provides functionality that we can utilize as a work around, minimizing this limitation. The Emerald compiler's `load` command stores environmental variables into separate environment files, which can be shared and loaded in later compilation sessions. This fixes the problem of different concrete types sharing OIDs. It also allow us to split compilation between multiple session, forcing the compiler to store manifest expressions multiple times. By compiling only a single file in each session, we can be certain that no bytecode is missing in any of the executable files. The only limitation left is that the environment files must be shared and kept updated between all interacting parts.

Chapter 8

Evaluation and Results

8.1 Evaluation Criteria

We have created a prototype and evaluated our work based on the extent of how the functionality provided by the prototype achieves our goal, i.e. to exchange object references across disjoint programs, merging both their object and node graphs in the process. We have written a set of tests that isolate each of the new features, and performed them in a fault-prone environment using PlanetLab. The tests themselves are based on whether the correct output appears, or does not appear, when running a set of programs. As our prototype should not interfere with any of the existing features, it was important to test that the new functionality worked when it should, and only then. In the following we discuss the details and expected output of each of the tests and evaluate their results.

8.2 Results & Evaluation

All tests discussed in this section can be found under the **tests** folder on GitHub [13], and are split into six separate parts:

1. The `welcome` expression
2. Moving a welcoming process
3. Node discovery on local networks
4. Merging node graphs using the `mergeWith`-operation
5. Merging node graphs using node discovery and emissary objects.
6. Merging node graphs using node discovery and third party emissary objects.

For each part we discuss the purpose and scope of the related tests. We also present a table showing the conditions and results of each test, followed by a section evaluating the results and their implications for the usability of the prototype.

As most of the tests in each part test the interaction between programs using the `welcome` expression, a recurring set of tests involves the catcher and thrower programs. The catcher blocks on a `welcome` expression and invokes arriving welcome objects. The

thrower is placed on a different node in the same node graph. It creates one or more objects and moves them to another available node. Another recurring test program is a modified version of the Kilroy example shown in 3.1. As these tests often involve changes in the node graph, the Kilroy program used in these tests waits for user input before traveling through the available set of nodes. Additionally, it outputs a string on each node to mark its presence.

Figure 8.1 describes the location and latency between each node used on PlanetLab, and figure 8.2 shows each PlanetLab machine used in the testing. Additionally, a local setup of two computers connected through a home router has been used to test functionality related to node discovery.

	UK	Canada	Germany	Ireland	Sweden	Greece
UK	42	141 000	31 500	14 900	37 600	51 300
Canada	142 000	13	163 000	151 000	164 000	165 000
Germany	32 000	163 000	404	42 500	29 900	52 900
Ireland	14 800	151 000	42 100	23	48 200	60 700
Sweden	37 600	164 000	29 500	48 200	40	75 200
Greece	55 200	170 000	62 500	67 700	82 200	4 830

Table 8.1: Latency in microseconds between all PlanetLab computers used in the testing. Measurements performed using Ping.

Location	**Hostname**
UK	planetlab2.xeno.cl.cam.ac.uk
Canada	planetlab3.cs.ubc.ca
Germany	mars.planetlab.haw-hamburg.de
Ireland	planetlabeu-2.tssg.org
Sweden	planetlab-1.ida.liu.se
Greece	vicky.planetlab.ntua.gr

Table 8.2: The hostnames for each of the PlanetLab machines used in the testing.

Model	Architecture	OS	Version	Network Interface
Raspberry Pi Model 3B+	armv7l	Linux: Raspbian	10	Ethernet
Lenovo ThinkPad x201	x86_64	Linux: Debian	10	WiFi

Table 8.3: Setup of the local computers.

8.2.1 The Welcome Expression

In this part we refer to the tests located in the **tests/welcome_expression** folder [13]. As described in chapter 7, the `welcome` expression should block until a welcome object is moved onto the local node and return a reference to that object upon unblocking. To test this functionality, we look at each of the `welcome` expression's properties atomically.

Firstly, every process executing a `welcome` expression should block until a welcome object arrives. To test this, we created a process that uses a timer to measure the time spent blocking, and had another process move a welcome object to the local node after a certain time interval. The time spent blocking must be at least as long as the specified time interval, if not we consider the test to have failed.

Additionally, only welcome objects should be welcomed by the expression, and the received object should behave as expected. As welcome objects must be both welcomable and conforming to a given type, the tests include scenarios where the moved object only fulfills one of these criteria.

Using the `attached` keyword, a tree-structure of objects can be moved by a single move statement. If an object with an attached reference to a welcomable object is moved, and the welcomable object is welcome on the destination node, the welcoming process should obtain a reference to the welcome object.

When multiple processes on a single node block on a `welcome` expression, an object may be welcomed by more than one process at the same time. Each process welcoming the object should obtain a reference to the welcomed object. Additionally, monitor objects may also be welcomable. When multiple processes invoke a welcomed monitor object, all invocations should be performed in a thread safe manner. If multiple processes attempt to welcome an object through invoking a monitor object, they should all receive references to different objects.

Results

Functionality	**Files used**	**Test locations**	**Results**
The `welcome` expression puts the process in a blocking state	**block.m**	Canada Sweden	Works
Invoke object obtained from `welcome` expression	**catcher.m** **thrower.m**	Canada Sweden	Works
The `welcome` expression does not welcome unwelcome objects	**catcher.m** **unwelcome_throw.m**	Canada Sweden	Works
The `welcome` expression does not welcome non-conforming objects	**catcher.m** **nonconforming_throw.m**	Canada Sweden	Works
Welcome an attached object	**catcher.m** **attached.m**	Canada Sweden	Failed
Multiple welcoming processes all obtain the same object reference	**catchers.m** **monitor_throw.m**	Canada Sweden	Works
Multiple welcoming processes welcoming through the same monitor object all obtain different references	**throwers.m** **monitor_catchers.m**	Canada Sweden	Works

Evaluation

The results show that most of the functionality related to the properties of the `welcome` expression works as intended, with the exception of welcoming attached objects. The welcome statement did not unblock after the attached object arrived, and no reference was obtained. To solve this problem, the general implementation of receiving attached objects would need to be altered. In the current implementation of Emerald, attached objects are not processed upon arrival, they are only inserted into the receiving node's object table. Thus, moving e.g. an attached object containing a process will crash upon arrival.

While the attached test failed, this bug may be considered a feature. By design, a welcomable object had no way of becoming unwelcome. By instead moving it through an attached reference, the welcomable object may be silently moved without being welcomed on arrival.

8.2.2 Move Welcoming Processes

In this part we refer to the tests located in the **tests/move_welcoming_process** folder [13]. When a process is blocking on a `welcome` expression, other processes may move the object containing the blocking process. By design, the blocking process should stop welcoming on its former residing node and start blocking again at the destination. Welcome objects arriving on the former node should not be welcomed, and welcome objects arriving at the destination node should.

The order of processing a newly arrived object matters in cases where the moved process belongs to an object that conforms to the same type the process is welcoming. In these cases, the welcome queue on the destination node should be checked before the process is inserted into it, so that the welcoming process does not welcome itself.

Results

Functionality	Files used	Test locations	Results
A moved process blocking on a `welcome` expression continues to block on the destination node	**block.m**	UK Greece	Works
A moved welcoming process does not welcome itself	**welcome_self.m**	UK Greece	Works
A moved welcoming process can obtain a reference to and invoke a welcome object on the destination node	**destination_welcome.m** **thrower.m**	UK Greece Canada	Works
The moved welcoming process does not unblock when a welcome object arrives at its previous destination	**destination_welcome.m** **thrower.m**	UK Greece Canada	Works

Evaluation

As can be seen in the results, we did not find any inconsistencies when moving welcoming processes. As described in the previous section, moving attached processes results in a crash in the current implementation of Emerald. Therefore, we were not able to test moving welcoming processes through an attached reference, though we suspect that this would not work for the same reasons.

8.2.3 Node Discovery

In this part we refer to the tests located in the **tests/node_discovery** folder [13]. To make a node advertise its presence, it can broadcast discovery messages on its local network. Any node on that network should recognize these broadcast messages and be able to differentiate between multiple broadcasting nodes. Broadcasting should be done regularly so that nearby nodes becomes aware when it is no longer present.

Some of these tests use the trace option `-Tdiscovery=8`, which prints out useful debugging information about the discovery messages being sent and received.

Upon discovering a new nearby node and when an existing node disappears, the appropriate user-created handlers should be called with a `Node` and `Time` object as parameters. By design, invocations performed through the discovered `Node` object reference should result in a failure, unless the node graphs are merged first. Already discovered nodes should always be referenceable.

As all the tests in this part use a local network for communication, most tests are performed on a home network with two local machines. See table 8.3 for details about this setup. The two computers are connected through a router supporting both WiFi and Ethernet.

Results

Functionality	**Files used**	**Test locations**	**Results**
Nodes broadcast themselves to the local network when run with the -D flag	**broadcasting.sh**	One local machine	Works
A broadcasting node is discovered on the local area network	**discovering.sh** **broadcasting.sh**	PlanetLab	Failed
A broadcasting node is discovered on the local area network	**discovering.sh** **broadcasting.sh**	Two local machines	Works
The node can differentiate between new and old broadcasting nodes	**discovering.sh** **broadcasting.sh**	Two local machines	Works
The node recognizes when a broadcasting node disappears	**discovering.sh** **broadcasting.sh**	Two local machines	Works
The handlers set by the `setDiscoveredNodeEventHandler` operation are triggered at the appropriate time	**event_handlers.m**	Two local machines	Works
The parameters passed to the event handlers behave as expected	**event_handler-_parameters.m**	Two local machines	Works
All available discovered nodes are always referenceable	**disc_node_list.m** **broadcasting.sh**	Two local machines	Works

Evaluation

From the first three tests we can see that discovering a nearby node works as expected on a home network, but fails when using a PlanetLab machine. We suspect this is a network configuration matter, and that these networks do not permit broadcasting messages. Even so, we can conclude that the node discovery feature is limited to networks with the ability to broadcast freely. As the node discovery test failed on the PlanetLab machines, all further tests involving node discovery are carried out using the home network.

Although the remainder of the tests show that the implementation works as intended, one peculiarity about the discovery events should be noted: If a handler is set after a node has been discovered, it will not be triggered by that node. Therefore, any handlers should be set at the beginning of a node's life cycle if discovering all available devices is desired. Should the user want to only discover new devices, the handlers can be set at a later stage. This problem is solved by the `getDiscoveredNodes`-operation described in 6.3.

8.2.4 The `mergeWith` Operation

In this part we refer to the tests located in the **tests/merge_with** folder [13]. By having one node know the identity of a foreign node, one can merge their node graphs by using the `mergeWith`-operation.

To test whether the node graphs have merged successfully, we ran a **nodeup.m**-program on each node, which outputs a string when another node enters the node graph. Furthermore, we used a modified version of the Kilroy program, which traverses the node graph and outputs a string on every node. While the **nodeup.m**-program shows whether a change in the node graph is properly detected, the **kilroy.m**-program shows whether all nodes in the merged node graph are responsive after the merge.

As the node synchronization algorithm ensures that the node graphs eventually will become complete, we tested node graph merging with different permutations: merging two graphs where both consist of a single node, merging two graphs where one contains one node and the other several, and merging two nodes where both contains several nodes. We also tested concurrently merging three node graphs where each graph had several nodes.

All nodes used in the testing were launched from different PlanetLab machines to include network latency as a part of the test. See table 8.1 for a complete overview of the network latency at the time of the testing.

Results

Functionality	Files used	Test locations	Results
Attempting to merge with an invalid node address yields an error message	**invalid_merge.m**	Germany	Works
Merging 1-to-1 graphs	**nodeup.m** **merge.m** **kilroy.m**	Germany Canada	Works
Merging 1-to-many graphs	**nodeup.m** **merge.m** **kilroy.m**	Germany Canada UK	Works
Merging many-to-1 graphs	**nodeup.m** **merge.m** **kilroy.m**	Germany Canada UK	Works
Merging two graphs with multiple nodes (many-to-many)	**nodeup.m** **merge.m** **kilroy.m**	Germany Canada UK Greece Sweden	Works
Concurrently merging one graph with two other graphs (many-to-many-to-many)	**nodeup.m** **concurrent_merge.m** **kilroy.m**	Germany UK Canada Greece Sweden Ireland	Works

Evaluation

When testing node graph merging on machines located in different countries, and with real-world latency issues, we did not encounter any problems. While the node

synchronization algorithm will eventually make sure that the resulting node graph is complete, it should be noted that an Emerald program may try to traverse the node graph before it becomes complete. This will not result in a crash, but may lead to the program leaving out some of the nodes in the final complete graph. As such, programmers should avoid handling changes in the node graph directly after a call to `mergeWith`, but rather deal with each new individual node when they become available through the node event handler.

8.2.5 Merge by Emissary Objects

In this part we refer to the tests located in the **tests/emissary_object** folder [13]. Upon discovering a new node, Emerald programs should have the ability to move an emissary object to the new node in an attempt to merge the two node graphs. Emissary objects must be welcomable, and for the node graphs to merge they must also be welcomed at the destination node.

We combine the functionality tested in the previous parts to test the WELCOME language mechanism as a whole. First, a node must be discovered. Second, a welcomable emissary object must be sent to the discovered node. Finally, the node graphs should merge using the node synchronization algorithm. The novelty of this part is to attempt to move an emissary object across disjoint node graphs.

Unwelcome or non-conforming emissary objects should not be able to cause the node graphs to merge. Additionally, both consecutively and concurrently sent emissary objects from different node graphs should only lead to one of the node graphs successfully merging, provided that there is only one welcoming process on the target node.

After successfully merging with a discovered node, the node-reference passed as an argument to the discovery-handler should reference a node within the node graph. Thus, invoking the node through this reference should behave the same as invoking any other valid node-reference.

Results

Functionality	Files used	Test locations	Results
Moving a welcome emissary object across node graphs merges the graphs	**nodeup.m** **thrower.m** **catcher.m**	Two local machines	Works
Moving an unwelcome emissary object across node graphs does not merge the graphs	**nodeup.m** **unwelcome_thrower.m** **catcher.m**	Two local machines	Works
Moving a non-conforming welcomable emissary object across node graphs does not merge the graphs	**nodeup.m** **non_conforming_thrower.m** **catcher.m**	Two local machines	Works
Consecutive emissary moves on a single `welcome` expression does not result in two merges	**nodeup.m** **thrower.m** **catcher.m**	Two local machines	Works
Concurrent emissary moves on a single `welcome` expression does not result in two merges	**nodeup.m** **thrower.m** **catcher.m**	Two local machines	Failed
Invocations on a discovered node are successful only after an emissary object is welcomed	**nodeup.m** **disc_node_invocation.m** **catcher.m**	Two local machines	Works

Evaluation

Most of the functionality of the WELCOME language mechanism works as expected and is in compliance with the design specification. Most important is that the discovered node's node graph does merge when the emissary object is welcomed, and does not merge when it is not. Consecutive attempts to merge with a node with only a single welcoming process result in a single merge, as the latter emissary object is dismissed. However, when two processes concurrently attempt to move a welcome emissary object, they result in two merges. This concurrency issue, and a possible solution for it, are described in 7.4.4, but the solution is not implemented in the prototype.

8.2.6 Third Party Emissary Move

In this part we refer to the tests located in the **tests/3rd_party_emissary** folder [13]. When moving an emissary object from the same node that discovered the destination node, the communication is done over a silent connection that exists only between the two nodes. However, if the discoverer wants to move an object from another node in the node graph, that node must also communicate with the discovered node. This part tests these scenarios.

To test this with two computers on a single local network, one of the nodes had to run a modified version of Emerald that ignores discovery messages. This was to ensure that the 3rd party node did not know about the discoverable node before the 3rd party move was initiated.

Results

Functionality	Files used	Test locations	Results
Moving a remote emissary object to a discovered node merges the graphs	**nodeup.m** **thrower.m** **catcher.m**	Two local machines	Works
Moving a remote unwelcome object to a discovered node does not merge the graphs	**nodeup.m** **unwelcome_thrower.m** **catcher.m**	Two local machines	Works
Moving a remote non-conforming object to a discovered node does not merge the graphs	**nodeup.m** **non_conforming_thrower.m** **catcher.m**	Two local machines	Works

Evaluation

The 3rd party emissary move works as intended, and looks like a regular emissary move to the user. This is in accordance with the transparent support for object mobility in the Emerald programming language.

8.3 A Discussion of Novelty

With the additions of the WELCOME language mechanism provided in the prototype, programmers are now able to write programs that were previously not possible.

In chapter 4 we discussed how the extensibility problem could lead to running programs not being able to collaborate with new programs, e.g. an updated component. With the newly introduced `welcome` expression new components can be added to running programs without shutting down or recompiling the system. Instead, the program could have a process welcome all updated components, hot swapping component code on the fly.

In chapter 5 we described how a client application could not connect to more than a single service, thus making it impossible to create programs reliant on information from two different sources. By dynamically merging node graphs, it is now possible to connect a program to multiple services.

Furthermore, discovering the identity of unknown devices was previously not possible. Now, local services may advertise themselves to the local network, enabling nearby devices to connect to them without knowing their identities.

The additions of the WELCOME language mechanism elevates the ability to perform service discovery, and connect running programs, to a language level feature. To our knowledge, this has never been done prior to this project. Although the ability to exchange the state of an object, typically through serialization, is

common, exchanging object references across programs is a novelty introduced by the WELCOME language mechanism.

8.4 Summary

We have developed and performed a set of tests related to each of the novel features introduced to the Emerald programming language by our prototype, and evaluated their results. Considering the results of the tests, it is clear that our prototype achieves our goal. It is now possible to exchange object references across disjoint programs, merging both their object and node graphs in the process. Dynamically merging node graphs has been tested with real world latency issues, and node discovery has been tested on a local area home network.

Furthermore, the test results demonstrate that the prototype still has some limitations:

- Welcomable objects moved indirectly through attached references are not welcomed at the destination.
- Broadcasting may not be supported on all local networks.
- Concurrent emissary moves to the same node may cause both moves to result in a merge.

Part IV

Conclusion

Chapter 9

Conclusion

Through this project we have enabled distributed programs to become acquainted and exchange object references among each other. To do this, we have designed a new language mechanism called WELCOME, which enables unknown nodes to become acquainted through emissary objects. To demonstrate the language mechanism, we have extended the Emerald programming language and implemented a prototype that supports the new features.

The language mechanism described in this thesis elevates communication to the language level. Service discovery and the ability to communicate using object references is supported using high level abstractions through syntactic extensions, facilitating the creation of and communication between distributed systems.

9.1 Project Outcome

The overall goal of the project was to allow Emerald programs to discover and communicate with one another. To reach this goal, we have designed the new WELCOME language mechanism, enabling distributed programs to exchange object references and merge node graphs. We have also implemented support for the WELCOME language mechanism in the Emerald compiler and virtual machine, and tested the resulting prototype. The outcome of the project can be summarized by the following points:

- The design of the WELCOME language mechanism
- The implementation of a working prototype
- The testing and evaluation of the prototype

9.1.1 Design

As part of the WELCOME language mechanism we have designed an extended syntax for the Emerald programming language that allows for programs to exchange object references. We have also designed a node discovery mechanism, and the ability for programs in disjoint node graphs to become acquainted through emissary objects. Finally we have introduced the node synchronization algorithm, which provides strong semantics for a complete node graph merge. This design enables discovery and communication between distributed systems at the language level.

9.1.2 Prototype

To demonstrate our design we have implemented a prototype based on a preexisting version of the Emerald programming language. The prototype consists of a working compiler and virtual machine supporting most of the functionality proposed in this thesis. The implementation of the WELCOME language mechanism features the `welcome` expression and its ability to exchange object references, node discovery and dynamically merge node graphs through the `mergeWith`-operation and by moving emissary objects to discovered nodes. Additionally, event handlers for node discovery and the node synchronization algorithm are implemented in the prototype. The prototype serves as a proof-of-concept for the design of the WELCOME language mechanism.

9.1.3 Evaluation

To evaluate our solution, we have created and carried out an extensive set of tests for each of the novelty features introduced with the prototype. These tests demonstrate the robustness, but also the limitations of our solution. Programs are now able to exchange object references, merging their object graphs. This introduces the possibility for running programs to communicate or perform an update without requiring a restart. Unknown devices can discover each other through a local area network, and interact using emissary objects. If the interaction is consensual, the node graphs of the devices merge. Additionally, the node synchronization algorithm ensures strong semantics for concurrent node graph merges to result in a complete node graph.

9.2 Limitations

Throughout this thesis we outline some limitations of the design of the WELCOME language mechanism, its implementation in the prototype, and in running distributed programs compiled with the Emerald compiler. Although these limitations do not hinder the overall project goal, they do restrict the possible use cases of the WELCOME language mechanism.

Using the `welcome` expression, there is no way to guarantee that all incoming objects are welcomed. When multiple welcome objects arrive at the same time, only one object is welcomed.

Although we have outlined solutions for both concurrent handling of emissary objects and referencing objects located on unknown nodes, these solutions are not implemented in the prototype. Additionally, not all emissary objects are properly welcomed, as welcome objects moved indirectly through attached references are overlooked by the Emerald virtual machine.

Emerald programs must be compiled in the same environment if they are to run on the same set of nodes. This limits the usefulness of the WELCOME language mechanism in the prototype, as the distributed systems that are to be connected must be compiled using a shared set of environment files. The limitation further restricts the ability to connect unknown systems, as these supposedly unknown systems must have been related at the compilation stage.

Bibliography

[1] Andrew P. Black et al. "The Development of the Emerald Programming Language." In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: Association for Computing Machinery, 2007, 11–1–11–51. ISBN: 9781595937667. DOI: 10.1145/1238844.1238855. URL: https://doi.org/10.1145/1238844.1238855.

[2] Andrew Black et al. "Distribution and abstract types in Emerald." eng. In: *IEEE transactions on software engineering* 13.1 (1987), p. 65. ISSN: 0098-5589.

[3] Andrew Black et al. "Distribution and abstract types in Emerald." eng. In: *IEEE transactions on software engineering* 13.1 (1987), p. 65. ISSN: 0098-5589.

[4] Brent Chun et al. "PlanetLab: An Overlay Testbed for Broad-Coverage Services." In: *SIGCOMM Comput. Commun. Rev.* 33.3 (July 2003), pp. 3–12. ISSN: 0146-4833. DOI: 10.1145/956993.956995. URL: https://doi.org/10.1145/956993.956995.

[5] Keith D Cooper. *Engineering a compiler*. eng. Amsterdam, 2012.

[6] Cay Horstmann. *Big Java: late objects*. eng. New York: Wiley, 2013. ISBN: 9781118087886.

[7] N.C. Hutchinson. *Emerald: An Object-based Language for Distributed Programming*. UMI order. University of Washington, 1987. URL: https://books.google.no/books?id=sHEyQwAACAAJ.

[8] Norman C. Hutchinson et al. *The Emerald Programming Language*. 1991.

[9] Stephen C Johnson et al. *Yacc: Yet another compiler-compiler*. Vol. 32. Bell Laboratories Murray Hill, NJ, 1975.

[10] Eric Jul. *Object mobility in a distributed object-oriented system*. 1989.

[11] Eric Jul et al. "Fine-Grained Mobility in the Emerald System." In: *ACM Trans. Comput. Syst.* 6.1 (Feb. 1988), pp. 109–133. ISSN: 0734-2071. DOI: 10.1145/35037.42182. URL: https://doi.org/10.1145/35037.42182.

[12] Tim Lindholm et al. *The Java® Virtual Machine Specification, Java SE 7 Edition, Third Edition*. eng. 1st ed. Addison-Wesley Professional, 2013. ISBN: 0133260445.

[13] Gaute Svanes Lunde and Olav Johan Ekblom. *Emerald: Welcome*. May 2021. URL: https://github.com/emerald/welcome.

[14] Robert Metcalfe and David Boggs. "Ethernet: distributed packet switching for local computer networks." eng. In: *Communications of the ACM* 19.7 (1976), pp. 395–404. ISSN: 0001-0782.

[15] Larry Peterson. *It's Been a Fun Ride*. Mar. 2020. URL: https://www.systemsapproach.org/blog/its-been-a-fun-ride.

[16] Rajendra K Raj et al. "Emerald: A general-purpose programming language." eng. In: *Software, practice & experience* 21.1 (1991), pp. 91–118. ISSN: 0038-0644.

[17] Arnon Rotem-Gal-Oz. "Fallacies of Distributed Computing Explained." In: *Doctor Dobbs Journal* (Jan. 2008).

[18] J.E. Smith and Ravi Nair. "The architecture of virtual machines." In: *Computer* 38.5 (2005), pp. 32–38. DOI: 10.1109/MC.2005.173.

[19] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems : principles and paradigms*. eng. Upper Saddle River, N.J, 2007.

[20] The GCC Team. *GCC, the GNU Compiler Collection*. Apr. 2021. URL: https://gcc.gnu.org/.