Model-Driven Development of Distributed Systems in Umple

Amid Zakariapour

A thesis submitted
in partial fulfillment of the requirements for the degree of

Masters of Computer Science

University of Ottawa
Ottawa, Ontario, Canada
January 2018

Abstract

Model-driven software development can help tackle complexity when developing large software systems. Model-driven development tools facilitate this. Such tools support multiple features and languages; some are multi-platform and support multi-language code generation from models. Umple is a full-featured open source language and modelling tool that we used as a basis for this thesis.

Distribution concerns have become a critical part of modern software systems. In this thesis, we present how we extended Umple to support the development of model-driven synchronous or asynchronous distributed systems. Our contributions provide simple syntax, model analysis capabilities, and programming APIs, which allow users to change the configuration of systems both at development and deployment stages. We also demonstrate how a system can be modeled without distribution concerns and easily be transformed to a distributed system through our approach.

The contributions of this thesis are: a) Creating a mechanism to distribute objects in Umple; b) Developing new semantics for modelling of distributed objects and providing supporting syntax for this in Umple; c) Investigating different patterns and technologies to implement code generation for distributed systems; d) Implementation, testing, and comparison of the distributed feature in Umple for executable Java code; and e) implementing a mechanism to dynamically modify the distribution plan at runtime.

Acknowledgement

Firstly, I would like to thank my supervisor Dr. Gregor Von Bochmann for his guidance, financial support and technical support throughout my Master's studies. I would also want to express my sincere gratitude to my head supervisor Dr. Timothy C. Lethbridge. His support, and assistance and guidance during my M.Sc thesis shaped my view on software engineering and research.

I would also like to thank my research team, Complexity Reduction Software Engineering (CRuiSE), and my colleagues, especially Dr. Vahdat Abdelzad.

I also thank my family, specially my father and my mother for helping me both morally and financially during my studies. Their unconditional support and encouragement helped me be strong and endure all the challenges.

Table of Contents

Table of Figures

Table of Tables

Table of Snippets

1 Introduction

In this thesis we discuss modeling technology that extends the Umple [1] open-source technology to enable generation of code that can run in a distributed environment.

Model-driven approaches are used widely in the development of large and complex software systems. Modeling tools play an important role in the development cycle; they provide mechanisms to model systems (textually or visually) and in many cases automatically generate some of or an entire executable system in a target platform, a virtual machine, or a simulation environment. Umple [1] is an example of such tools. It takes UML [2] models and generates code for multiple platforms.

Distribution of software systems on multiple hosts is a way to deal with ever-growing needs for better performance, reliability, and scalability. A distributed system brings its own complexity to the design and development processes, and therefore requires modern approaches such as model-driven software development.

Existing open source model-driven modeling approaches have limited support for distributed applications and usually require the user to modify the generated code. Moreover, there are several code generation tools for distributed parallel programming that distribute the objects automatically on multiple hosts and try to provide transparency to the developer. However, they try to give the user some control to configure every step of the transition of the system to the distributed system (e.g. [3], [4]). While these tools focus on transparently distributing the objects, none of them support model-driven development.

Distributed systems can be implemented in several architectures [5]. Distributed object-based systems (DOBS) [6] have been created to distribute the objects of an object oriented system. In the popular object-oriented programming languages, there are frameworks to create distributed systems (eg. Jax.ws [7] or Java RMI [8] in Java). These frameworks usually need the developer to configure the system based on the execution environment and number of machines. They might also require extra code and annotations compared to a regular non-distributed program.

Although one way of creating a distributed application is to automatically distribute the objects on different processors to achieve the highest efficiency possible, there are several cases in which the distribution structure of the objects and some of the communication aspects should be modelled by

the developer based on the requirements. For example, some data might need to reside only at a specific location. Furthermore, in order to have a general distributed application development tool, the goal would be to allow the developer to program distributed applications the same way as regular applications and adjust the system incrementally so as to increase its distribution as needed.

In this thesis, we show how we extended Umple to support distributed application development in order to achieve the aforementioned goals. The distributed system feature is implemented for the generation of Java programs with Java RMI and Web services. The system can also be extended for other communication technologies such as simple TCP connection, or newer technologies like Restful web services.

1.1 Some definitions

The followings are descriptions of some key terms we use in this thesis:

1.1.1 Node

Every address space in which the system runs on is a node in the system. When a node starts, it means part of the system starts running on that node. Each node may have a different location (IP address, port number, and URL), and different objects running on it. In Java, each Java virtual machine is a node. Therefore, there can be multiple nodes on a physical machine.

1.1.2 Synchronous method call:

In a synchronous method call, the caller waits for the completion of the method's execution. Whether the method returns a value or not, the calling thread will be blocked until the method finishes its execution.

1.1.3 Asynchronous method call:

In an asynchronous method call, the caller calls the method and continues working without waiting for the method to finish. It does not return any value and the method can work in parallel with the caller if they run in separate threads.

1.1.4 Real Object

In an object-oriented system, objects contain data and are operated on by procedures of their class (or superclasses). In this thesis, a “real” object represent an object that is created on a certain node as desired by the developer. It performs certain behaviors and might change state. It should be contrasted with a Proxy Object.

1.1.5 Implementation class

In this thesis, there are sometimes multiple classes generated for one intended class. The class that contains the implementation of the intended class with all its attributes and methods is called the implementation class. A real object is always an object of an implementation class.

1.1.6 Proxy Object, Proxy Class

Contrary to real objects, proxy objects (as introduced in the proxy pattern [9]) only delegate the procedures to other objects. The other object could be a real object or another proxy object. In this thesis, we mention generated proxy classes for distributable classes. A proxy object is an instance of a proxy class.

1.1.7 Remote class, Remote Objects

In this thesis, any class for which methods of its instances can be called remotely is called a remote class and the object are called remote objects.

1.2 Goals of the Thesis

The goal of this research is to add a set of features to the Umple modeling language to support the implementation of distributed object systems. The Umple development environment should be able to generate a distributed system out of any object-oriented system modelled and developed using the Umple language. The generated code of a distributed system needs to be able to run on multiple hosts.

One of the biggest challenges is to create a straightforward and specific pattern of code generation so that any program written in Umple can be distributed. On the other hand, we need the distribution capability be integrated with the current Umple syntax and semantics.

1.3 Research Questions

- How can Umple generate code that can run on different nodes?
- What should the architecture be for a distributed system generated by Umple?
- What communication technologies should be used to distribute objects in Umple?
- What kind of simple annotations could be used by the developers to define distributed systems.
- How can the system generate distributed and non-distributed executable code out of the same model? How should the generated code differ from generated code for a non-distributed system?
- How can we defer decision of object placement and number of machines to the run-time instead of during modeling and development?
- What should the mechanism of object creation be on a remote machine?
- What constraints are added to the model when the system is distributed?

1.4 Outline of the Thesis

This thesis is organized in the following order:

In Chapter 2, we explain the concepts and background related to the thesis. We introduce syntax and semantics of Umple. The popular technologies and tools of distributed system development are described. There is also a brief review of similar works and tools with similar ideas including distributed systems in model-driven tools and distributed object-oriented Java development.

In Chapter 3, semantics and syntax of distributed systems in Umple are explained. The concepts and keywords for creating distributed systems in Umple, how to transform a legacy system into a distributed one, how to define which objects should be on which nodes, and other special cases are described in this chapter.

In Chapter 4, we explain how Umple generates code for a distributed system. We explain how we tackled the challenges of hiding the distributed code. The system is implemented with multiple approaches and each approach is addressed in detail.

In Chapter 5, a case study system is introduced and multiple test cases are performed to verify how the system works, that the same trace is generated in distributed and non-distributed versions, and to compare performances of different approaches.

2 Background

We organize the background chapter of this thesis into four parts. The first part (Section 2.1) is a short overview of Umple, to which this work is an extension. The second part (Section 2.2) discusses the primary communication architectures and tools to create distributed systems. Section 2.3 describes modelling tools similar to Umple and other similar distributed system ideas in modelling tools. Section 2.4 introduces middleware systems that generate distributed code from Java code. These have similarities with the distributed system feature in Umple.

2.1 Umple: A Model-Oriented Programming Language

Umple is an open-source modelling tool and a model-oriented programming language. It brings software modelling and programming into alignment by integrating the model and code written in various programming languages. It provides a textual syntax for modelling and developing object-oriented programs using modelling elements like classes, interfaces, state machines, traits, methods, attributes, and associations.

The Umple code contains both the model in the Umple language and the executable code of methods in the target programming languages. The developer develops the system by incrementally editing the model and adding the code of their preferred language inside the body of the methods; this all occurs in one integrated environment. Umple can compile the model-code combination into the executable target language. Umple can be used as a stand-alone command-line application, as well as web-based application and an Eclipse plugin.

Umple provides a textual syntax for modelling and developing object-oriented programs using modelling elements like classes, interfaces, state machines, traits, methods, attributes, and associations. Umple also includes concepts such as aspect orientation, constraints, and active objects. Although most of the concepts are borrowed from UML, Umple adds some concepts that are not in UML explicitly (e.g. modifiers for attributes, and traits). For multithreaded concurrent systems, prior to this work, Umple already supported the active object notion, queued state machines, and pooled state machines.

The model can be blended with C-family languages such as C++, Java, and PHP to generate high quality, object-oriented code executable on multiple platforms. It also generates code for model

checking tools like Alloy [10] and SMV [11]; the latter allows the model to be formally verified by these tools. Umple also can generate SQL and XML code of the model.

```
1   namespace accidents;
2
3   class AccidentType
4   {
5     code;
6     description;
7     key {code}
8   }
9   class SeriousnessLevel
10  {
11    code;
12    description;
13    key {code}
14  }
15  class Employee
16  {
17    id;
18    department;
19    name;
20    supervisor;
21    other_employee_details;
22  }
23  class Accident
24  {
25    id;
26    description;
27    Date date;
28    Time time;
29    other_details;
30
31    * -- 1 Employee;
32    * -> 1 AccidentType;
33    * -> 1 SeriousnessLevel;
34    key {id}
35  }
```

Snippet 1: Umple code example from UmpleOnline [12]

Umple presents graphical class diagrams and state machine diagrams based on the textual model. The textual model and the graphical model are both editable and are synched with each other. The developer can model the system using both textual and the graphical interfaces.

An example of Umple code is shown in Snippet 1. Umple code is intended to be saved in files with the ".ump" extension.

We will not review the entire syntax of Umple here; there are many papers with such a review, as well as many online resources. However, we will explain non-obvious syntax as needed.

2.1.1 **Classes**

Classes in Umple have the same semantics as UML classes. They are defined using the keyword "class" followed by the name of the class. The contents (such as methods, attributes, associations, generalizations and state machines) of the class are wrapped inside curly brackets (the body of the class). In the graphical view of Umple, a class is shown as in standard UML class diagrams.

A class can be an implementation of an interface if it is defined by the "isA" keyword followed by the name of the interface. A class can be abstract which is described by the keyword "abstract". Abstract classes cannot be instantiated but can be inherited by other classes. The "isA" keyword is also used to indicate a generalization relationship between two classes.

2.1.1.1 Association

Associations between classes, following UML semantics, are supported in Umple. Associations can be described in a graphical way (as in UML) or textual syntax. Associations can be directional, or bidirectional; aggregations are supported, as are composition, wherein destroying the 'whole' destroys the 'parts'. The multiplicity of the associations must be defined.

The symbol **– (as in line 31 of** Snippet 1**)** is used for bidirectional associations, **-> (as in line 32 and 33 of** Snippet 1**) (l)** or **<-** symbols are used to describe unidirectional associations. The **<@>-** symbol shows a composition relationship. This syntax is used to define an association starts with multiplicity, the name of the class, the symbol of association and is followed by the multiplicity and name of the other class. An association or composition relationship can be defined inside each of the two associated classes, or alternatively as a standalone association, or in an **association class** dedicated to describing associations that have their own attributes.

Role names are generated based on the names of the classes and multiplicity. However, user-defined role names can be added after the name of the class. For example, the role name of the warehouse in the association in line 4 of Snippet 2 is "location".

```
1  class Warehouse
2  {
3    name;
```

```
4    0..1 location -> * Product;
5  }
6  class Product
7  {
8  }
```

Snippet 2: Basic assocations in Umple

Umple generates API methods based on the associations. These APIs include CRUD functionality as well as methods to search for a specific associated object. For example, a get method is generated for each association and returns the reference of the associated object (i.e. “getProduct” method for Warehouse in Snippet 2). Table 1 shows the list of methods that are generated based on a typical association. The exact API varies depending on the multiplicity and certain special stereotypes of the association, such as being immutable.

Table 1: List of generated APIs for a class with an association that has a many (*) end with a class named “Product”

public GeometricObject getProduct(**int** index)
public List<Product> getProducts()
public int numberOfProducts()
public boolean hasProducts()
public int indexOfProduct (Product aProduct)
public static int minimumNumberOfProducts()
public boolean addProduct (Product aProduct)
public boolean removeProduct (Object aProduct)
public boolean addObjectAt(Product aProduct, **int** index)
public boolean addOrMoveProductAt(Product aProduct, **int** index)

2.1.1.2 Attributes

Attributes represent simple data stored in each instance of the class. Each attribute has a type and a name. Attributes can be settable or immutable, private or public. Immutable attributes are attributes that will not change after initialization in the constructor. There are several more stereotypes to

specify the attributes as described in the Umple user manual [13]. The type of the class can be any of the primitive types defined by Umple (including Integer, String, Boolean, Float, Date and Time). By default, the type of the attribute is String.

Attributes can be initialized to have specific values as seen in line 3 of Snippet 2. If there is no initialization value defined by the user, a parameter with the same type of the attribute is added to the constructor of the class to set the value. On the other hand, the "lazy" stereotype means there is no initialization and the value can be set any time after the object is created. Umple generates get/set APIs for each attribute of the class.

2.1.1.3 Generalization

A class can inherit another class using the "isA" keyword followed by the name of the more general class. A class can only have one parent.

2.1.1.4 Singleton pattern

The Singleton design pattern [9] enforces a class to have only one instance in the entire program. Umple generates code for singleton classes. A singleton class is specified in Umple syntax by the "singleton" keyword (Snippet 3).

```
1  class Manager
2  {
3    singleton;
4  }
```

Snippet 3: Singleton class in Umple

Umple generates different code for a singleton class. The constructor of a singleton class is private. Moreover, a static variable called "theInstance" exists in the class and is instantiated by a static method called 'getInstance()'. This method makes sure there is only one instance of the class (theInstance) and returns it. It instantiates the theInstance only if it has not been instantiated.

```
1  public class Manager
2  {
3
4  private static Manager theInstance = null;
5
6    private Manager()
     {}
```

```
7
8     public static Manager getInstance()
9     {
10      if(theInstance == null)
11      {
12        theInstance = new Manager();
13      }
14      return theInstance;
15    }
  }
```

Snippet 4: Generated singleton class in Java

2.1.1.5 Code Blocks

Umple accepts blocks of code in the target language. The block of code is wrapped inside a pair of curly brackets. The target language can also be specified so that the block of code only appears when the code is generated in the same language. This can be done by mentioning the name of the language (such as Java or CPP) before the curly brackets. The blocks of code can be used as the body of the methods, entry or exit actions for states and events, as well as active objects.

```
1  void print(String text)
2  Cpp{cout<<text;}
3  Java{System.out.println(text);}
```

Snippet 5: Multiple code blocks for a method

2.1.1.6 Methods

Classes can have user-written methods to allow the developer to add functionality. The syntax of definition of methods is borrowed from Java with modifiers, followed by the return type and name of the method. The arguments of the method are listed inside round brackets. By default, methods are instance methods and public, but the user can define them as static (class methods) or private by using “static” or “private” modifiers.

The body of the method comes after the signature of the method inside a pair of curly brackets (code block). The body of the method can be implemented by any target language supported by Umple. This can also define different implementations for one method by defining multiple code blocks. Umple can then use the proper body of the method based on the target language in the code generation.

The developer can call any of the API's which Umple generates based on the different elements of the class such as attributes, associations, or state machines. The developer can also generate Javadoc from the Umple file or refer to the API reference to get information about the usable API, both the methods that are generated and the ones the developer has written.

Abstract methods can be defined by using the "abstract" keyword as a modifier. Abstract methods are methods which do not have an implemented body. Therefore, the signature of an abstract method ends with a semicolon instead.

2.1.1.7 Keys

The developer can define *keys* to specify attributes and associations as the identifiers of the objects. Umple then generates the methods needed to specify which objects are equal by generating 'equals' and 'hashCode' methods in the class based on the defined key.

In the example of Snippet 6, two players are equal if they have the same id. But 'PlayerOnTeam' objects are associated with a given player and team, as well as a given year.

```
1   class Team {
2     name;
3     * -- 1 League;
4   }
5
6   class Player {
7     name;
8     Integer id;
9     key { id }
10    }
11
12  class PlayerOnTeam {
13    Integer year;
14    * -- 1 Player;
15    * -- 1 Team;
16    key { year, player, team }
17  }
```

Snippet 6: Use of Keys in Umple

2.1.2 **Interfaces**

The interface concept of UML is defined in Umple as a list of methods (without body). Interfaces are defined in Umple using a syntax similar to the classes using an "interface" keyword. An interface can

extend other interfaces using the “isA” keyword. The “isA” keyword is also used for the classes that implement Interfaces (see Snippet 7).

```
1  interface MovingAnimal
2  {
3    public void walk(int distance);
4  }
5  interface Mammal
6  {
7    isA MovingAnimal;
8    public void eat();
9  }
10 class Dog {
11   isA Animal;
12   public void bark()
13   {}
14 }
```

Snippet 7: Example of extending and implementing interfaces in Umple

2.1.3 **Active objects**

An active object is an object that runs its own thread after being created. In Umple, the user can define the object as active with the “active” keyword. A block of code can follow the active keyword. This block of code will start running when the object is instantiated. Active objects can be used to create a concurrent system. An example of an active object is shown in Snippet 8.

```
1  class A {
2    name;
3    active {
4      System.out.println("Object "+getName()+"is now active");
5      for (int i = 1; i < 10; i++) {
6        System.out.println(" "+name+" was here");
7        Thread.sleep(1000);
8      }
9    }
10   public static void main(String[] argv) {
```

```
11      new A("1");
12      new A("2");
13    }
14  }
```

Snippet 8: Active object example in Umple [13]

2.1.4 State machines

State machines are used to describe and model the behavior of classes of an object-oriented system in an event-driven environment. A state machine can be defined either as an element of a class or as a separate element in the system. Each class can have multiple state machines that describe the functionality of the class. Umple generates a graphical state diagram, and can also generate state tables and simulations of the state machine.

State machines are generally used to describe the behavior of the system. Modeling the behavior of the system using state machines allow developers to verify and validate the system more easily.

Umple uses UML semantics for state machines and the UML notations is used in the graphical diagram. Umple supports code generation for state machines so that the model can be executed. The generated code includes methods and variables to implement the logic of the state machine. A state machine is defined by a name and the body of the state machine which is inside a pair of curly brackets.

```
1  class TicTacToe {
2   state {
3     xTurn {
4       xPlays [isThreeInARow()] -> xWin;
5       xPlays [!isThreeInARow() && spaceAvailable()] -> oTurn;
6       xPlays [!spaceAvailable()] -> tie;
7     }
8     oTurn {
9       oPlays [isThreeInARow()] -> oWin;
10      oPlays [!isThreeInARow() && spaceAvailable()] -> xTurn;
11      oPlays [!spaceAvailable()] -> tie;
12    }
13    xWin {
14    }
15    oWin {
16    }
17    tie {
18    }
19  }
20  // The following need to be implemented
21  Boolean isThreeInARow() {return false;}
22  Boolean spaceAvailable() {return true;}
23  }
```

Snippet 9: State machine (booking) example in Umple[12]

State machines contain one or more states inside their bodies. Each state is defined by a unique name followed by a pair of curly brackets. Inside each state the events and transitions are shown in Snippet 9. Umple generates a method corresponding to each event name. A transition occurs in a ‘source’ state and has the -> symbol pointing to a ‘destination’ state. When the event method is called while the state machine is in the source state, then the state machine will perform the actions required for the transition and sets the destination state.

Transitions can have guards which are Boolean statements; a transition is only taken if the guard evaluates to true. States can also have entry or exit actions which are code blocks that run when the state machine enters the state or exits the state. A transition can also have an action. Actions are programmer defined code written in a code block; the programmer is required to make these blocks do computations that take negligible time, such as setting a variable.

States can also have ‘do’ activities. ‘Do’ activities are blocks of code that take non-negligible time to execute. In languages that support multithreading (i.e. Java, C++) Umple runs the ‘do’ activity block in a separate thread. The thread will be interrupted as soon as the state is changed. This allows the state machine to stay alive and accept other events even when running the activity code on a given

state. Therefore, it is possible to create concurrent (multi-threaded) system by using multiple do activities in a single state.

2.1.4.1 Regions and sub-states

Each state machine has a default start state, which is the first state listed.

A state can also have an internal set of states that acts like its own mini state machine. The overall state machine containing this state is called a compound state machine. Transitions can have as destination the border of a compound state machine, indicating that the destination is to be the default start state. A transition can also have a sub-state as its destination.

Regions are an idea permitting parallelism in state machines. When there is more than one region in a state this means that there is more than one state machine that runs concurrently; the system is in two or more states (one in each region). A set of regions in a state are called Orthogonal regions and are defined by the "||" symbol. Each region can have its own 'do' activity. An example is shown in Snippet 10.

```
1  class X {
2    activesm {
3      topstate {
4        thread1 {
5          do {
6            System.out.println("Doing the activity - Normally this
7  would last a long time");
8            Thread.sleep(1000);
9            System.out.println("Done thread 1");
10         }
11       }
12       ||
13       thread2 {
14         do {
15           System.out.println("Doing the other activity - Again
16 normally this would last a long time");
17           Thread.sleep(3000);
18           System.out.println("Done thread 2");
19         }
20       }
21     }
22   }
23   public static void main(String[] argv) {
24     new X();
25   }
26 }
```

Snippet 10: State machine with a state with two concurrent regions (example from Umple [13])

2.1.4.2 Queued and pooled state machines

Umple supports generation of asynchronous multithreaded state machines by introducing queued and pooled state machines. The user can define a state machine as queued or pooled using the “queued” and “pooled” keyword. In the generated code for the state machine, the method calls are registered in a queue or a pool and a separate thread runs them in a specific order. In the queued state machine, the event on the front of the queue is handled first. In a pooled state machine, it handles the first event in the queue that can be handled in that state, leaving others for being handled in later states.

2.1.5 Aspect Orientation

The developer can inject code to run before or after the APIs that Umple provides for associations and attributes. The code can also be added before or after the state machine components such as states, events, and event methods. An example of using the ‘after’ keyword is shown in Snippet 11. Snippet 12 shows an example of how the developer can inject code as actions in transitions when an event is

called (line 4) and event methods (line 11). Note that the injected code on the event method runs no matter the state of the machine but the injected code on the transition only runs when the state is ‘on’.

```
1  class X
2  {
3    a;
4    whenWasASet;
5
6    after setA {
7      setWhenWasASet(getA() + System.currentTimeMillis());
8    }
9  }
```

Snippet 11: Using "after" keyword on 'set' method in the Umple User Manual [13]

```
1  class Car {
2    queued radio {
3      on {
4        radioToggle /{lightBlink();} -> off;
5      }
6      off {
7        radioToggle -> on;
8      }
9    }
10   before radioToggle {soundBeep();}
11 }
```

Snippet 12: Using "before" keyword on a transition and an event method [13]

This code injection can be done with pattern matching to reduce redundancy in the code. For example, Snippet 13 shows how the developer can inject the same code before all the ‘get’ methods for attributes ending with “name” (first name and last name).

```
1  class Student
2  {
3    const Boolean DEBUG = true;
4    firstName;
5    lastName;
6    cityOfBirth;
7    before get*Name {
8      if ( DEBUG) { System.out.println("accessing the name"); }
9    }
10 }
```

Snippet 13: Using "before" keyword for get method with pattern matching in Umple [13]

2.1.6 **Traits**

A trait [14] is a set of behaviors; in its classic form it contains pure methods, although Umple has enhanced them such that they can contain attributes, associations and state machines. Traits represent a part of a class and can be used to add some functionality to multiple classes to improve reusability. They can be stateless (without attributes, state machines and associations) or stateful (with some of these elements). In Umple, traits are defined using the keyword “trait” followed by the name of the trait. The contents (such as methods, attributes, associations, generalizations and state machines) of the trait are wrapped inside curly brackets (the body of the trait).

Traits cannot be instantiated and be used on their own. Instead, they are added to classes using “isA” keyword. The body of the class is merged with the content of the trait. However, traits follow the same syntax and semantics of classes. Therefore, each trait can be transformed to a class by changing the keyword “trait” to “class”. One example of trait is shown in Snippet 14. The ‘Person’ class only uses the ‘Identifiable’ trait but Company uses Organization trait as well.

```
1  trait Identifiable {
2    firstName;
3    lastName;
4    address;
5    phoneNumber;
6    fullName = {firstName + " " + lastName}
7    Boolean isLongName() {return lastName.length() > 1;}
8  }
9  class Person {
10   isA Identifiable;
11 }
12
13 trait Organization {
14   Integer registrationNumber;
15 }
16
17 class Company {
18   isA Organization, Identifiable;
19 }
```

Snippet 14: Example of traits in Umple [13]

Traits can use the state machines in traits to create composite states.

2.1.7 **Mixins**

Umple allows splitting the model into different files, and also allows splitting the specifications of elements such as classes into multiple parts that might be in different files. In other words, two definitions of the same class are actually two parts of the same class definition. This brings flexibility and is very useful for large systems. It permits separation of concerns. For example, the definition of the attributes of a class can be separated from its associations, or its implemented methods.

The developer can choose which files Umple should use to build the model. Umple then mixes all the specifications of each element out in the different files and creates the elements of the model. Therefore, the developer can add new specifications to the existing system by adding files to the system instead of changing the current files.

For example, if the code in the Snippet 15 is in one file called 'trafficSystem-file.ump', the model created out of that is going to be the same as using the code in Snippet 16 and Snippet 17 when they are on separate files 'trafficSystem-file1' and 'trafficSystem-file2'. Because the specifications of the class 'Accident' is the same in both Snippet 15 with Snippet 16 and Snippet 17 together. To tell Umple to use both trafficSystem-file1 and trafficSystem-file2 the directive below should be used in the main Umple file (Snippet 18):

```
1  class Accident
2  {
3    id;
4    description;
5    Date date;
6    Time time;
7    other_details;
8
9    * -- 1 Employee;
10   * -> 1 AccidentType;
11   * -> 1 SeriousnessLevel;
12   key {id}
13 }
14
```

Snippet 15: trafficSystem-file.ump

```
1  class Accident
2  {
3    id;
4    description;
5    Date date;
6    Time time;
7    other_details;
8    }
```

Snippet 16: trafficSystem-file1.ump

```
1  class Accident
2  {
3    * -- 1 Employee;
4    * -> 1 AccidentType;
5    * -> 1 SeriousnessLevel;
6    key {id}
7  }
```

Snippet 17: trafficSystem-file2.ump

```
1  use trafficSystem-file1.ump;
2  use trafficSystem-file2.ump;
```

Snippet 18: trafficSystem-file-new.ump

2.1.8 **Tracing**

Umple allows the developer to trace the methods, events, and attributes with a simple syntax. The developer can also trace the generated APIs for associations and attributes. The keyword “trace” must be specified, followed by the name of the method, attribute, event, or API. For example, in Snippet 19 method() and id are traced. Whenever id changes or method() is called, the system will print it. In Snippet 22 line 8 the addSupervisor() method is being traced. The trace message will include the time, thread, the name of the Umple file and line number of the trace command in the source, the name of class, object name (key), and the operation (method, event, …), with its name. If the traced item is an attribute its value will also be printed.

```
1   class JavaMethod
2   {
3     trace method();
4     trace id;
5
6     id;
7     int method( int y ) {
8       x += 1;
9       return x;
10    }
11  }
```

Snippet 19: Tracing methods and attributes in Umple [13]

```
1   class Student {
2     Integer id;
3   }
4
5   class Professor {
6     1 -- * Student supervisor;
7
8     trace add supervisor;
9   }
```

Snippet 20: Example of tracing associations in Umple [13]

2.1.9 Umple Grammar

Umple is specified using a Backus–Naur form (EBNF) grammar with a slightly specialized syntax. Umple has its own parser tool, which can parse any language defined using this grammar. Umple itself is defined by this grammar. Like any EBNF, the grammar is constructed of terminal and non-terminal symbols.

2.1.9.1 Terminals

Terminals can be any words or characters except '+', '*', '(', ')' characters and words including these characters. The terminals are matched by the parser exactly (case-sensitive). The words "OPEN_ROUND_BRACKET" and "CLOSE_ROUND_BRACKET" are used to match the open and close round bracket characters.

2.1.9.2 Non-terminals

There are two types of non-terminals in the Umple grammar: Simple and rule-based. Simple Non-terminals are not defined by a full grammar rule. Instead, the parser only matches a particular

predefined pattern (such as a regular expression or sequence of alphanumeric characters) or one of a set of terminals. For example, ‘distributeTech’ in line 1 of Snippet 21 is compared to match with either “RMI” or “WS”.

```
1  distributable- : [=distributable:distributable] [=distributeTech:RMI|WS]? ;
2  methodThrowsExceptions : throws [~exception] ( , [~exception] )*
```

Snippet 21: Part of the Umple grammar

The minus (“-”) after the name of the non-terminal tells Umple parser not to add the non-terminal as a token in the Umple internal model. “?” means the component is optional, “*” means the component can repeat zero or multiple times. The “+” symbol means the component appears at least once.

Rule-based non-terminals are named on the left-hand side of a grammar rule and are defined by an expression after a colon. Snippet 21 shows examples of defining two rules. Non-terminals are ‘distributable’ and ‘methodThrowsExceptions’, visible on the left-hand side of the rules.

2.1.9.3 Special Matching Cases

The Umple parser can match terminals with special patterns. For example, tilde (“~”) before the terminal means the terminal must be alphanumeric (line 2 of Snippet 21 Umple also allows other regular expression syntax and semantics. For example, “\d” means digit, “!” means not and “\d+” means at least one digit.

2.2 Tools for Creating Distributed Applications

There are several different approaches to implement distributed applications. Figure 1 shows some of the most popular of these approaches ordered based on the level of exposure to the developer. The more the developer is exposed to the distributed nature of the system, the less transparent it is.

Direct TCP connection is the least transparent and least sophisticated approach, in which all the connections and data transmission must be handled by the developer. It can be implemented so that the nodes are completely independent and only communicate when needed. On the other hand, distributed shared memory systems hide the distribution completely.

Message passing provides an underlying platform for sending and receiving messages and hides the TCP connection configurations. However, the sending and receiving nodes must both recognize the same message passing mechanism.

Although any of these approaches can be used to create distributed systems in different architectures, RPC is specifically designed for remote method calling and are most suitable for object-oriented systems. They are not as transparent as DSM so that the developer is able to control over the distribution. They provide mechanisms for fault-tolerant remote calls in remote objects. Each of the technologies shown in Figure 1 is explained in the following sections.

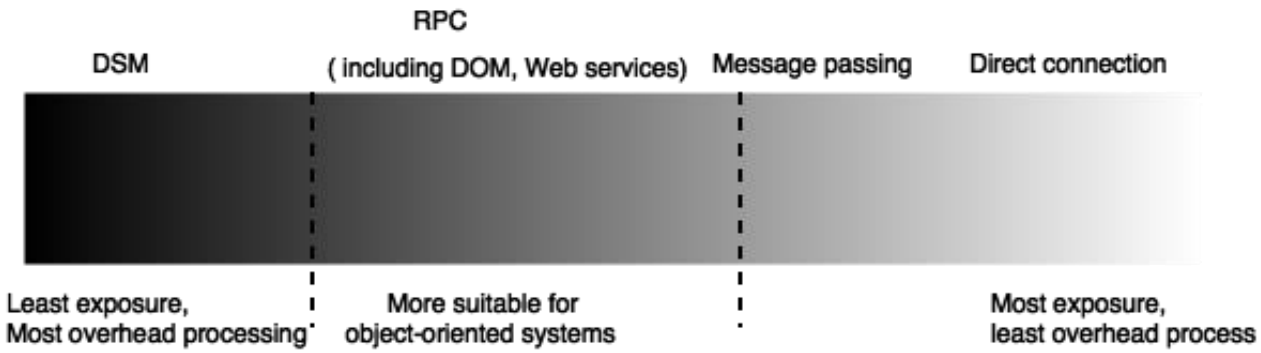

Figure 1: Different distributed application approaches based on level of exposure of the user to the details of implementation

2.2.1 Direct Network Communication

Programmers can send data through the network using the APIs provided by operating systems. The connection can either use the TPC or UDP protocol. To create a remote method call, some data must be passed over to represent the method call and the other side must interpret it as a method call as well. The developer is responsible for encoding and decoding the data that is being passed. This encoding mechanism must also take into consideration the differences in the platforms of the two nodes.

Using direct network communication, the user is most exposed to communication details of the distributed system and any change in the distribution and number of hosts should be foreseen in the code.

2.2.2 Message Passing

Message passing libraries such as Message Passing Interface (MPI) [15], and Parallel Virtual Machine (PVM)[15] formalize the message passing mechanism and hide the underlying communications. The developer uses the provided API and sends the data as a message. The message is received on the remote side and results in a signal in the program. The developer must still encode and decode the method calls into a meaningful message, but there is an abstraction over the platform differences and initial connections.

Message-oriented middleware systems [16] extend the message passing libraries and provide an abstraction over the location of the nodes. The nodes can send messages without knowing about their location. These middleware systems also provide a publish/subscribe model which allows more than one client to receive messages. The publisher sends messages with certain topics and any subscriber to that topic receives the messages.

2.2.3 Remote Procedure Call (RPC)

RPC systems provide mechanisms to identify the possible remote methods that can be called, and provide a reference to the remote server on which to perform procedure calls. The client calls the method on a remote node and sends the arguments as a serialized request message. The call is performed on the server side on a static code in an environment based on the arguments of the method.

SUN RPC [17] is an example RPC system developed by Sun microsystems. XML RPC [18] is another more modern RPC that uses XML documents for sending the method signature and the arguments. Web services and distributed object middleware systems also provide remote procedure call capabilities.

RPC frameworks facilitate the underlying network communications and provide the user with a proxy object with the same interface as the remote object. The user can call methods on the proxy and the RPC framework delegates the calls on the real object on the target host. The programmer defines the methods that are going to be called remotely by defining the classes and interfaces.

Web services and RMI are different groups of technologies for remote procedure call (RPC). They provide a mechanism for the developer to be able to call methods on remote machines. Web services use XML based messages to encode the method calls and are platform independent. Each

method call is a separate HTTP request. However, web services require more configurations and annotations by the developer comparing to other RPC systems.

Among RPC technologies, Distributed object middleware systems provide an object-like proxy for the developer so that the developer can treat the proxy of the remote object as the object itself. It provides more transparency but requires the nodes to always be connected.

2.2.3.1 Web Services

Web services provide remote procedure calling by encoding the methods and the parameters with SOAP (Simple Object Access Protocol). SOAP is an XML-based protocol for exchanging messages between client and server.

The Client connects to the web server, which is on a specific URL. The server sends an XML file, which follows the "web service description language" (WSDL) schema. WSDL describes the method definitions, the URLs of the services, and the transport protocols to use the services. The client then can call methods on the services using those URLs and http get/post.

Clients can generate proxies to access the services and call methods.

2.2.3.2 Distributed Object Middleware (DOM)

Distributed object middleware (DOM) is based on distribution of objects in an object-oriented system. Java RMI [8], Microsoft .NET remoting [19] and CORBA [20] are some examples of DOM frameworks.

The programmer must indicate the following for distributed objects [21]:

- The programmer must define which classes should be the "remote" classes. The objects of a remote class are going to be accessed remotely
- The interfaces of each node, by defining the interfaces of each remote class in the node.
- The programmer must register (bind) each object with the middleware's infrastructure with a name and use the name to access the object on the other node.

The main difference between RPC and DOM is that DOM proxies support more features than just delegating the calls to the remote host. DOM proxies can be treated like any other objects with greater transparency than RPC.

2.2.4 Distributed shared memory (DSM)

Distributed shared memory systems hide the communication of the nodes in the system from the programmer by hiding the distributed nature of the memory. The address space is shared among the nodes and the programmer does not know where the data resides. Java/DSM [22] is an example of a distributed shared memory system. The shared memory can be a virtual shared memory that each node can access which is managed by the DSM system. It can also be shared variables that other nodes can access. The DSM system must guarantee the mutual exclusion when nodes try to access the variables.

Tuple-space systems are a type of shared memory that uses a blackboard access mechanism to provide the mutual exclusion. It provides a repository of tuples that can be accessed by different nodes. Each node puts the data as a tuple in the space and any node can access the data in the space. JavaSpace[23] is an example of a Tuple-space systems.

Shared memory systems are scalable, fault-tolerant and portable. The user is sheltered from sending and receiving data in the network. On the other hand, they are usually slow because of the necessary locks needed to prevent simultaneous access to shared data. Furthermore, the distribution of data cannot be controlled by the developer.

2.3 Model-driven Development Tools

There are many model-driven development tools, and the tools have a variety of different purposes. Open-source UML model-driven development tools such as ArgoUML [8], StarUml [9], Bouml [10], and Green code generator [11] generate partial code, typically stubs, for classes, interfaces, and attributes and have some similarity to Umple in this matter, although Umple generates complete code. They allow visual modelling and code generation for different platforms. Code to complete the implementation needs to be added to the generated code later. As far as we know, none of these tools have code generation features for distributed applications

Comodo [12] (component modelling for observatory) introduces UML profiling for distributed component-based systems. It defines elements for distributed components to separate the platform-specific concerns from the system requirements. It defines two sets of model elements to describe the system: platform-independent elements and platform-specific elements. This way the system can be developed for different targets without the need to change the platform-independent model. For

example, the components and interfaces are platform independent but the components' implementations are platform-specific.

To describe the placement of the software components and deployment, ‘container’ and ‘machine’ elements are defined. Container is an environment that one or more components can run on. The deployments of components are modelled as attributes of the machines which are physical systems that one or more containers can run on. Containers are similar to the runtime components we introduce in Umple. Comodo is more of a modelling tool and does not produce generated code for distributed systems with objects distributed automatically.

Umple also has composite diagrams with textual modelling support for component-based development [24]. It supports C++ code generation for each component. The components can run on separate machines as a distributed system. The components have active methods that receive messages. However, the contributions of distributed system feature in this thesis focus on distributing the objects in an object-oriented environment with classes having associations with each other.

2.4 Distributed Objects systems with Java code generation

The tools introduced in this section are middleware systems that transform an object-oriented Java program into a distributed system. While these tools are not model-driven development tools, they have similarity with our work in that they allow for generating programs that can create objects on different hosts without the need for the user to change the code significantly.

2.4.1 **Do!**

Do! [25] is a Java RMI-based programming language that transforms a Java code to a distributed parallel system by distributing the objects. It uses simple keywords to specify the remote classes and the host nodes on which the objects should be created. To indicate a remote class, the user must use the following syntax by implementing the Accessible interface:

```
public MyClass implements Accessible{}
```

Objects then can be created on any host using the new method on each runtime server (factory) on the local node. For example, a client on node 0 can create an object of MyClass on node 1 using the following syntax:

```
myObject=(MyClass)DoRuntime.remoteNew(1,”MyClass”,new Object[0]);
```

After that DoRuntime calls the object creation method on the desired node. It will return a proxy of the remote object in the MyClass type object. MyClass is a wrapper class that the methods only delegate the calls to the remote object.

The wrapper object works as a proxy of the object and handles remote calls to real objects using RMI. The wrapper uses the same name of the class but the implementation of the class is renamed to another name. The runtime classes on each host are responsible for object creation.

Do! suggests programming in a multithread way to achieve a distributed parallel programming environment.

This is similar to the second approach of this thesis. However, in Umple we use the same 'new' statement with an optional extra constructor parameter instead of calling the factory method directly. The other difference is that in Umple, it is not the host that is being specified, it is the runtime component.

2.4.2 **JavaParty**

JavaParty [4] generates a RMI-based distributed system without the need for the user to write the code related to RMI. JavaParty works as a pre-processing phase of the Java compiler. It can also generate Java code that can be compiled later. For each remote class, JavaParty generates set/get methods for all the variables and makes all methods remotely accessible. It creates five generated classes or interfaces for each remote class: These include an implementation class of all static methods and variables and its Java RMI compliant interface, an implementation class for instance members and their interface in compliance with Java RMI. There is also a need for wrapper classes with the same name as the class that redirects the method calls (static or non-static methods) to the correct remote objects.

JavaParty supports the remote call of static methods in a distributed system. It assumes the static object of the class resides on a specific node and creates a remote object that represents the static object. Therefore, for every static variable and method of the class, there is an instance member in the implementation class (MyClass_class in Figure 2). There is always an object of MyClass_class_impl in the system on a specific node. For every object creation, an object of MyClass

(wrapper) and an object of MyClass+impl will be created. The wrapper redirects the static method calls to an object of MyClass_class_impl through MyClass_class interface. It redirects instance method calls to an object of MyClass_impl through MyClass_Intf interface.

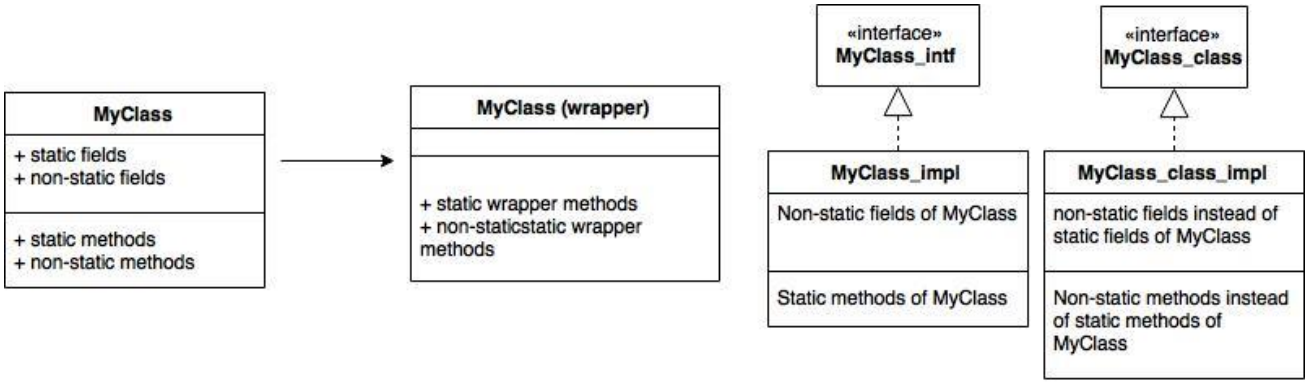

Figure 2: The generated classes for each accessible class in JavaParty

In JavaParty, object placement policies can be defined to manipulate where the objects should be created. For example, it can be set for objects of a class to be placed on different nodes in a round-robin way. JavaParty also supports migration of objects. The object can migrate from one node to another and the proxies will not lose references to it.

The developer must specify which classes are going to be distributed so that JavaParty can transform these classes into the five new classes and interfaces of Figure 2. Non-transformed classes can have references of objects of the transformed classes and call objects on them as if it were a local call. However, after transformation, the objects only have a reference to the instances of the wrapper class which keeps the name of the original class.

2.4.3 **Fargo**

Fargo [26] is an extension to the Java compiler that generates a program with distributed objects. The system is generated based on the Java code and a separate layout set by the developer. This layout is described in an event handling scripting language that the system uses to map the objects and migration policies. The developer uses this layout to dynamically define where the objects should be created and where they should migrate to. Objects in the same node interact with each other locally and cross-node method calls are made by Java-RMI method invocation.

Fargo defines ‘complets’ as the building blocks of the application. Programmers create complet objects which are the remote objects. The implementation of generated code is similar to Do! and

JavaParty. For each complet a proxy called ‘complet reference’ which is a reference to the implemented object called ‘anchor’ is generated. It also has a runtime system to handle the object creation. The complet reference can be reference to a local object or remote object based on its location.

Fargo fixes the problem of the object passing itself using Java's “this” (which is a reference to anchor) in compile time. It has a parameter passing system which detects these cases and replaces the reference with the complet reference.

2.4.4 **J-orchestra**

J-orchestra [3] transforms non-distributed Java programs to distributed systems using byte-code transformation and Java RMI. During the transformation, J-orchestra analyses the system to find all object creation (constructor call) occurrences. It then shows the user a list of all constructor calls and the user can decide where each object should be created and set object migration polices for that object. The user creates the distribution plan using a graphical tool.

J-Orchestra transforms all the possible classes in the program into remote classes by replacing direct field accesses to other objects with method calls, replacing constructor calls with calls to factory methods, replacing references to objects with references to special remotely-accessible wrapper objects, and running the objects in different threads.

J-Orchestra transforms all the classes in the system, therefore all objects are remote objects by default. J-Orchestra is useful for distributing non-distributed Java systems but it cannot be used for general distributed systems with multiple entry points because it only transforms non-distributed classes into distributed systems and provides no API for architecture of the distributed system.

2.4.5 **Pangea**

Pangea [27] transforms non-distributed programs into distributed systems automatically. It uses JavaParty to create objects remotely. Therefore, Pangea is focused on analyzing the code and provide an optimized object placement strategy. It also uses the object migration feature of JavaParty to move objects among nodes.

2.4.6 **JavaSymphony**

JavaSymphony (JS) [28] facilitates development of distributed parallel systems using Java RMI for remote method calls. It is a Java framework that supports distribution of objects. It does not hide the distributed characteristics of the system like Do! and JavaParty. JavaSymphony defines virtual components of the system to describe a virtual architecture. These components are nodes, clusters which are collections of nodes, sites which are collections of clusters, and domains which are collections of sites.

The developer must instantiate a JavaSymphony registration object and register the application manually. The node, cluster, site, and domain of the objects are defined as classes in the library and user can create instances of them.

JS-objects are defined as objects that can be distributed and run on different threads. The code for the JSObj class exists in the library but the developers must create JS-objects and pass their own objects that are needed to be called remotely to the JS-objects. Any method call on the JS objects will be redirected to the methods of a real object wrapped inside the JS object. An example of developer code for creating remote JS-objects is shown in lines 11, 14, and 17 of Snippet 22. The “class name” is the name of the class of the object that the user wants to call methods on and the object is created by the JS object using Java reflection.

```
1   // get node on which this application is being executed
2   Node local = JS.getLocalNode();
3   JSConstraints constr;
4   Node node;
5   Cluster cluster;
6    Site site;
7    Domain domain;
8
9   // generate an object of class “class name” at
10  // a node decided by JRS or restricted to constraints
11  JSObj obj1 = new JSObj(“class name” [, constr]);
12
13   // generate object on the local node
14  JSObj obj2 = new JSOBj(“class name”,local);
15
16   // generate object on a specific node
17  JSObj obj3 = new JSOBj(“class name”,node);
```

Snippet 22: example of programming with JavaSymphony [28]

JavaSymphony provides APIs enabling the developer to invoke synchronous and asynchronous calls on the JS-objects (for example on obj1). JavaSymphony supports object migration. The developer can map the objects to different nodes manually in the code, or the system can map and migrate them automatically.

2.4.7 **Java//**

Java// (Java parallel) [29] is a Java library to create distributed systems by distributing the objects similarly to JavaSymphony. Java// cannot transform a non-distributed Java program into a distributed system, but it provides wrapper objects that use RMI or Java message passing (JMS) to translate the method calls. Java// is heavily based on reflection in Java and reifying the method calls. Using reflection, it enables the object to work as a proxy to the remote object at runtime. The proxy calls the remote object using RMI or JMS.

Java// provides active objects that are remotely accessible. Active objects follow the active object pattern with each object in a separate thread. Java// queues the method calls and serializes the method call to the remote object; using active object pattern, it provides asynchronous method invocation. On the other hand, passive objects are created normally as any object in Java.

The programmer creates some objects as active objects by using the factory class called “Javall" that is provided in the library. The user must specify the parameters of the constructor of the object and call the object creation method on the factory class (Snippet 23). The programmer specifies the placement of the objects on nodes by instantiating an object of the “Mapping” class provided by Java.

```
1  Object[] params = {“name", new Integer (12)};
2  A a = (A) Javall.newActive ("A", params, myMapping);
```

Snippet 23: Creating an active object in Java// in a certain mapping [29]

2.5 Comparison to similar technologies

Error! Reference source not found. Shows a comparison of the related work and Umple. Umple is he only one that supports model-driven development and provides model-based analysis of the model. In particular, is the only one that can detect singleton objects and provide special API for them.

Umple does not have object migration. However, Umple can create a distributed system automatically without changing the user code. Umple gives the ability to the developer to put constraints on object placement but defers object placement to runtime using a configuration file. Pangea, Javaparty and J-orchestra provide a GUI environment to place objects on certain nodes before runtime. JavaSymphony and Fargo provide a rule-based definition of object placement. Since they have object migration, the rules also apply to runtime and migration policies.

Umple can be used as a tool to create a distributed system. It can also distribute a non-distributed system automatically. However, distributing a non-distributed system does not make it parallel. Other systems make every distributed object run on a separate thread. In Umple, the program will still run sequentially unless the developer defines an object as an active object.

Table 2: Summary of the comparison of similar tools

Tool	Model-driven	Object mig-ration	Object placement policy definition	Multiple entry-point	Choose classes to distribute	Underlying technology	Automatic distribution	Support singleton

Umple [1]	Yes	No	Text-based	Yes	Yes	RMI, WS	Yes	Yes
Do! [1]	No	No	No	Yes	Yes	RMI	No	No
JavaParty [2]	No	Yes	GUI	No	Yes	RMI	Yes. parallel	No
Fargo [26]	No	Yes	Rule-based	No	Yes	RMI	yes, parallel	No
Jorchestra[3]	No	Yes	GUI	No	Yes	RMI	Yes, parallel	No
Pangea [27]	No	Yes	GUI	No	No	RMI	Yes, parallel	No
Java Symphony[4]	No	Yes	Rule-based	Yes	No	RMI	No	No
Java Parallel[5]	No	Yes	No	Yes	No	RMI, JMS	No	Yes

In the next chapter, we show how to create a distributed system in Umple and how we implemented this feature in Umple.

[1] http://www.irisa.fr/caps/PROJECTS/Do/

[2] https://svn.ipd.kit.edu/trac/javaparty/wiki/JavaParty/Download

[3] https://yanniss.github.io/j-orchestra/

[4] http://www.dps.uibk.ac.at/projects/javasymphony/

[5] https://www.inria.fr/sloop/javall/

3 Semantics and syntax of Distributed systems in Umple

An object-oriented system is based on one or more objects and their interactions. An object describes a structure of data as well as set of behaviors. Therefore, a straightforward way of distributing an object-oriented system is by distributing the objects: a distributed system is a system with at least two objects on different nodes. The system in Figure 3 is distributed by distributing the object on two nodes shown in Figure 4.

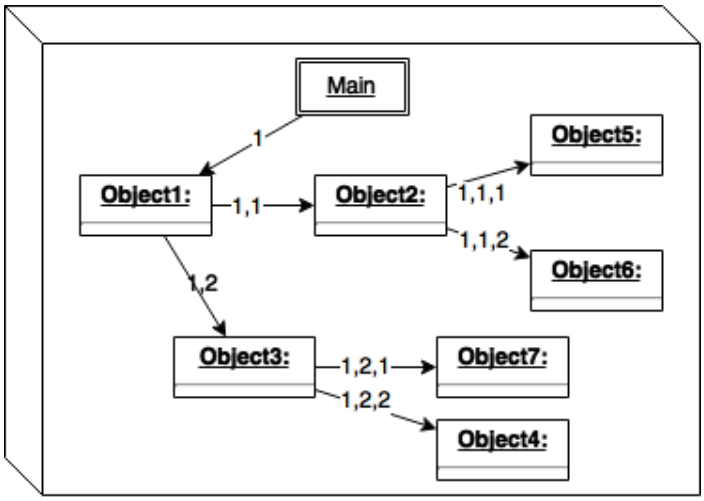

Figure 3: An example of sequential control flow in a non-distributed system (Arrows show method calls)

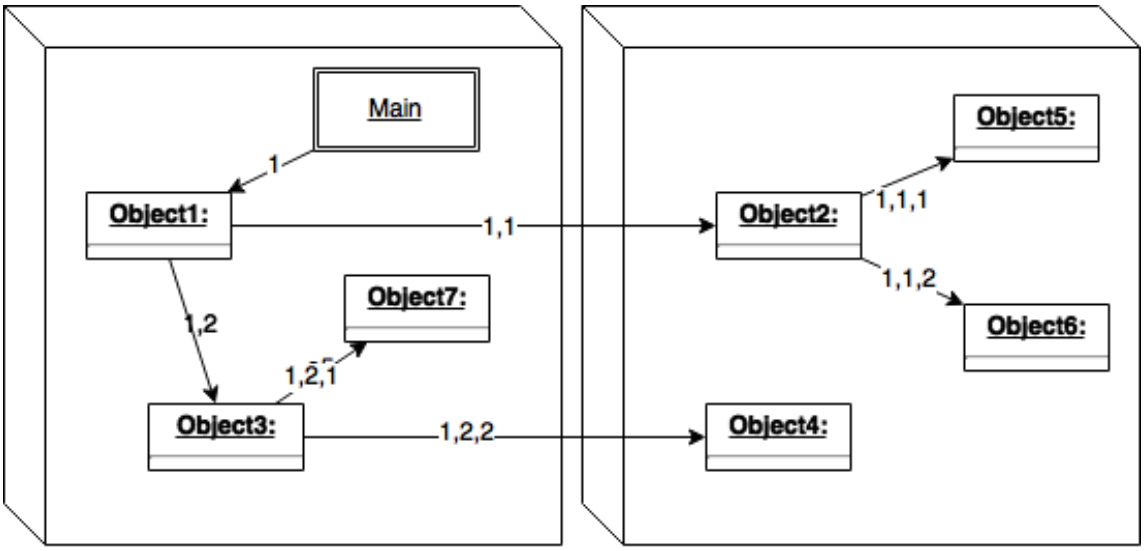

Figure 4: Sequential control flow on a distributed system

A program starts with main method and runs different functions on different objects. If the objects are distributed on different machines, the control flow is distributed. In Figure 3 we see how the control flow of a sequential program is distributed by distributing of objects. In Umple, we can develop a concurrent system using active objects. These objects run in separate threads. Queued and pooled state machines also run in separate threads and can communicate using message passing (asynchronously). In Figure 5 we see a system with active objects. In Figure 6 we see how the control flow of a concurrent system can be distributed by distributing the objects.

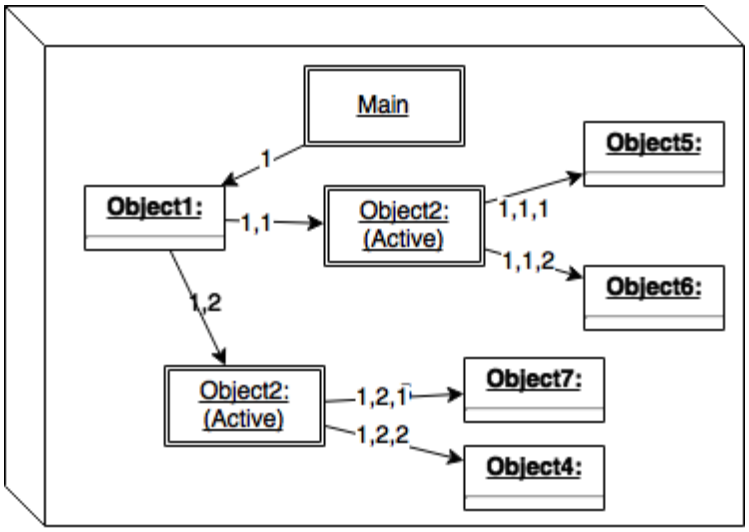

Figure 5: Concurrent control flow of a non-distributed system

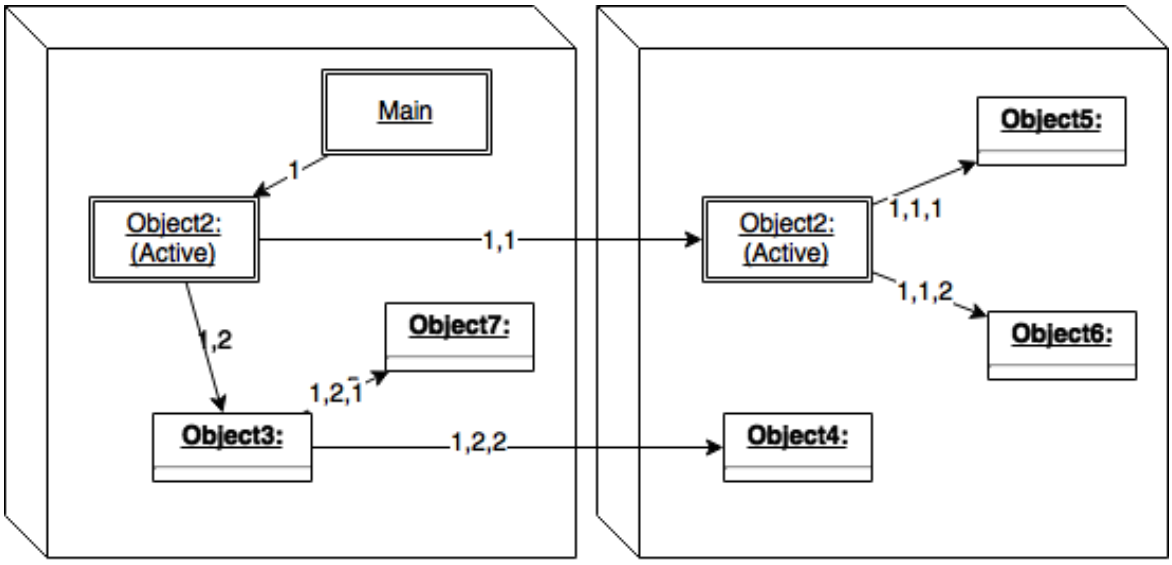

Figure 6: Concurrent control flow of a distributed system

Distributing the functionality of the system results in the flow of control of the system being distributed. In an object-oriented system, each object holds some data and functionality. Therefore, distributing the objects is directly related to the distribution of both data and control flow of the system.

One of the important features of a distributed system is how the data and control flow of the system is distributed. Distributing the control flow of the program is done to improve performance, because of location or type of data being distributed, to take advantage of multiple CPUs or to fulfill some other requirements of the system (e.g. collaboration or scalability). It also depends on the characteristics of each node of the system (e.g. data capacity), and privacy issues (e.g. some data cannot be shared, or must be kept in a secure location).

If performance was the only criteria, the system could be optimized for this. But the user should be able to distribute the objects based on other requirements. This includes explicitly defining which objects should be distributable and how they should be mapped on different nodes. The mapping of the objects on different nodes is called object placement.

In Section 3.1 , we explain the semantics and syntax of creating distributable objects in Umple. The semantics and syntax of placement of the distributable objects are explained in Section 3.2. In Section 3.3, we describe the mechanism of creating the objects on different machines. Section 3.4 explains the model-driven practice of development of distributed systems using Umple.

3.1 Distributable objects

To define which objects should be distributable in Umple, the user must define which classes should be distributable. A distributable object is an instance of a distributable class. Distributable objects are created and exposed so that their methods can be called remotely.

To create a distributable object, the user can create the objects just like any other object using the 'new' keyword. On the other hand, it can be set to be on a certain node as described in Section 3.2.

3.1.1 Distributable Classes

We define a class to be distributable if the objects can be distributed on different nodes and are accessible for remote method calls. Umple generates special code for distributable classes to support remote method calls.

The distributable feature is inherited by the subclasses, meaning the subclasses of a distributable class are also distributable.

There are different options to define which classes are to be distributable classes. One way is to name the distributable classes using a command-line arguments when running Umple. This option does not change the Umple code but cannot be used in the online user interface for Umple (UmpleOnline) since that interface allows for no command-line arguments.

The other approach is to add the keyword "distributable" to each relevant class. This employ's Umple's 'stereotype' mechanism, which is the same as how one can declare a class to be singleton by using 'singleton' keyword. This is illustrated in Snippet 24.

```
1  class Customer{
2
3    distributable;
4  }
5
```

Snippet 24: an example of syntax of a distributable class

When generating Java code, Umple recognizes the distributable classes and creates distributable code for them. This keyword is ignored for other programming languages. Of course, the keyword is reserved and cannot be used as a variable name for string variables.

The distributable keyword can be used along with the mixin feature in Umple by defining the distributable classes all in a separate Umple file. Using this feature, the developer can define the class as distributable as in Snippet 24 and define other elements of the class in separate files. This means that the original (non-distributable) code does not have to be modified to convert an Umple system into a distributed system.

3.1.2 Distributable interfaces

Distributable interfaces are the communication interfaces for the distributed classes that implement them. A communication interface can be used as provided interface of a distributed class. In other words, the interface lists the methods that can be called remotely. When a class implements a distributable interface in Umple using the isA keyword, only the methods defined in this interface can then be used for remote calls. Other methods can only be called locally. Currently, Umple only allows one distributable interface, but having multiple interfaces would allow each client to call specific subsets of methods on the object.

If a distributable class doesn't implement any distributable interface, then Umple creates one by default containing all public methods.

A distributable interface is defined using a simple distributable keyword as shown in Snippet 25.

```
1  interface Customer{
2
3     distributable;
4  }
```

Snippet 25: an example of syntax of interface

3.2 Object Placement

In our approach, we define a two-stage object placement semantics. The first stage is to declare objects on certain 'runtime components' at compile time. Then, the developer needs to map these runtime components on different nodes at execution time using configuration files.

3.2.1 **Runtime components**

Instead of specifying directly which node the object should be created on, we specify containers called runtime components. Objects should be declared on runtime components first in the code. In other words, *a runtime component is a group of objects that always run on the same node*. Each runtime component can run on any node at execution time. There can be more than one runtime components on a given node.

Every object of a distributable class resides in a runtime component with the name of the class unless the object is explicitly defined to be part of a given runtime component. The name of a runtime component can be any arbitrary alphanumeric word. Figure 7 shows an example of distribution of objects based on distribution of runtime components. In Figure 8, the class diagram of the relationship of the concepts node, runtime component, and object can be seen with a 'has a' (composition) relationship between node and runtime component as well as runtime component and object.

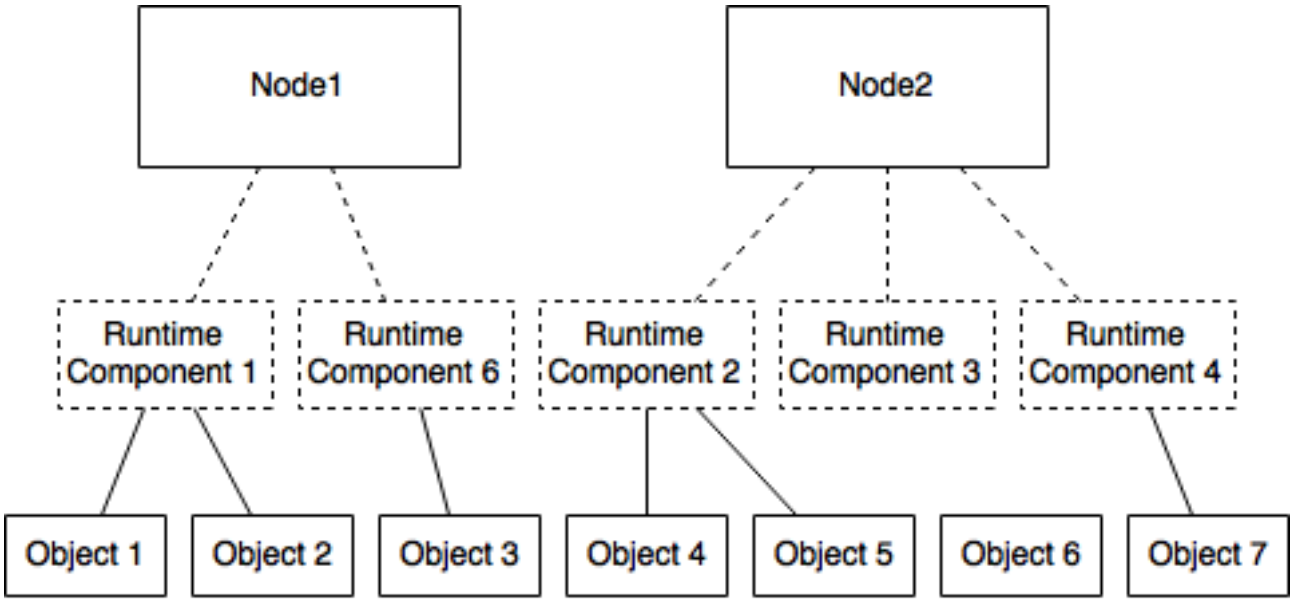

Figure 7 An example of distribution of seven objects over two nodes based on five runtime components

To place an object on a certain runtime component, the name of the runtime component can be defined during object creation. The name of the runtime component is given in the last parameter of the constructor, in a special syntax seen in Snippet 26.

```
1   Vendor v= new Vendor (“Vendor 1”,
        UmpleRuntime.getComponent(“Component1”));
```

Snippet 26: Umple syntax for adding name of runtime component to a constructor

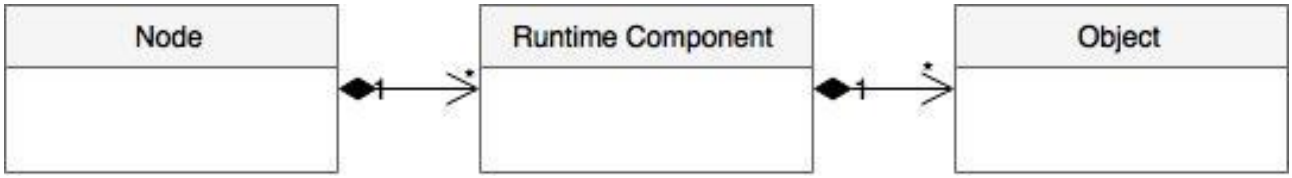

Figure 8 : Class diagram of relationships between node, runtime component, and object.

There is an implicit local runtime component on each node. Therefore using “local” as runtime component will result in the object being created on the same node. (Snippet 27)

```
1   Vendor v2= new Vendor (“Vendor 2”, UmpleRuntime.getComponent(“local”));
```

Snippet 27: Specifying the runtime component to be local

3.2.2 **Configuration File**

Each node needs to know information about other nodes as well as itself. This information consists of the number of nodes in the system, their location, port numbers, and runtime components on that node. This information must be delivered at runtime and is necessary for RMI or Web services to connect to a remote node. This information can be added to the program as a command-line argument. However, as the number of nodes increases, adding this information in the command-line would be very inconvenient. We decided to put this information in a configuration file and the program reads the file when it runs.

The configuration file can be edited by the user before running the system. Therefore, Umple’s runtime system tries to read a file containing this information when a node starts. In the current implementation of Umple, changes to the configuration file have no effect while the system is running. There can be duplicates of this file on different nodes and each node reads their copy of the file. These copies of the configuration files can have different content. Although, currently, the user is responsible for the risk of having different configuration files and some nodes not knowing about

the other nodes. An ability to synchronize the configuration files is a future work, although a technology such as Dropbox or Network File System would work.

In the configuration file; each node's information is wrapped inside brackets. Inside each node the following information is given: an ID, port number, URL or IP address, and a list of runtime components. Runtime components are listed inside a pair of brackets and are separated by commas. Some examples of a configuration file are show in Snippet 28, Snippet 29, and Snippet 30.

```
{id=0 ip=192.168.2.1 port=10541 {rc1}}
{id=1 ip=192.168.2.2 {rc2, rc3}}
{{Vendor} ip=192.168.2.3 id=2 }
{id=3 ip=192.168.2.4 {Warehouse}}
```

Snippet 28: A configuration file with four nodes and five runtime components

```
{id=0 ip=192.168.2.1 port=10541 {rc1}}
{id=1 ip=192.168.2.1 port=10540 {rc2, rc3}}
```

Snippet 29: A configuration file with two nodes on the same machine and different ports

```
{id=0 url=http://localhost  port=10541 {rc1, Vendor, rc2, rc3}}
```

Snippet 30: A configuration file with all runtime components on the same node

Each node must know its own ID so that it knows which of the nodes in the configuration file it is. If we add it to the configuration file, we must edit the configuration file on each node. To be able to use the same configuration file on every node, the ID of the node needs to be added in another way. To enable this, the ID of the node, and the location of configuration file is given to the system either through command-line arguments or a different file. To give this information using a file, it should be added to a properties file named "config.properties". This file should be located in the program directory. An example of the properties file is show in Snippet 31.

```
Name=node1

ConfigFile= c:\ configuration.txt
```

Snippet 31: Example of a config.properties file

If the information is given neither by the properties file nor program arguments, the ID would be zero by default and the configuration file name defaults to “configuration.txt”.

3.3 Remote Creation of objects

We want each distributable object to be created on a specific node which is defined by its runtime component and the configuration file. The object creation process must provide a mechanism to create the object on its target node or create the object and send it to its target node.

Every distributed system generated by Umple has a class called “UmpleRuntime” which is the runtime system of Umple for distributed systems. This class is also an object factory. An instance of this class runs on each node all the time. UmpleRuntime objects are responsible of creating objects on their local nodes based on the object placement policies. Each runtime object initiates communications with other nodes, creates objects on demand, and commands the object factory on the other nodes to create objects.

3.3.1 Initiating the communications among object factories

When an UmpleRuntime object is created on a node, it reads the configuration file to locate other nodes. It then exposes itself as a server and tries to connect to other nodes as client. It can call methods on other object factories after the communication is successfully initiated.

These communications channels must be maintained as long as there is a chance of remote object creation. Detection of when object creation ends for each node would requires each node knowing how many objects it will create. This is not possible without a dynamic code analysis before running the system. Therefore, in the current implementation of Umple, the communications between factories must remain active as long as the system is running.

There are two types of nodes. Active nodes and passive nodes.

Active nodes are those in which the user runs a class with a main method implemented by the user. This main method is an entry point to the control flow and creates other objects. In a traditional system, there is only one of these classes in the system, but there might be more for distributed systems (more details in Section 3.4). These nodes execute any local code that does not require access to remote objects without delay. At some point, when the system wants to create a remote object, the local factory connects to other nodes’ factories as a client. If there are two active nodes in the system

and each one tries to create objects on the other node, whichever node reaches that point first instantiates the UmpleRuntime object and waits for the other node.

On the other hand, if there is an active node that does not create any remote object, but needs to be available to serve other nodes, the user must instantiate the UmpleRuntime object manually (Snippet 32). The user can also set the id and location of the system manually before instantiating the UmpleRuntime object.

```
1  UmpleRuntime.setThisNodeId(1)
2  UmpleRuntime.setFileAddress("c:\configFile.txt")
3  UmpleRuntime.getInstance();
```

Snippet 32: Instantiation of UmpleRuntime

Passive nodes run the main method of the UmpleRuntime class that only instantiates an object of the UmpleRuntime class and waits for remote object creation requests. When a user needs a node that does not start the system, he runs this class on that node. This node accepts the address of the configuration file and the id of the node as command line arguments. (Snippet 33)

```
$ java ecomSystem.UmpleRuntime  c:\configFile.txt  1
```

Snippet 33: An example of running a passive node with node id=1

If the connection to make a remote method call is not initially successful, the client retries indefinitely until the connection is initiated. If there is a time delay for a node to respond (nodes cannot start at the same time), the requesting node waits until the connection is established.

In the current implementation, in case of a failure in the remote call and remote connections, the system waits and retries indefinitely until any network problem is resolved. As future work Umple ought to be enhanced to allow alternative behaviors in case of connection failures on remote method calls.

There can be a situation where node A connects to node B but B cannot connect to A. In this situation; A can create nodes on B but not the reverse. If the system is sequential, this will stop the program at the occurrence of B trying to create an object on A. However, the program will resume whenever the connection is established.

If after initiation one node is disconnected, a remote exception is caught and printed. All the method calls and object creation requests on that node will wait until the connection is established again. If the system is concurrent, the interruptions of the connections would result in a one of the threads of the system waiting. Each thread executes until it needs to create a remote object or to call a method on a remote object.

3.3.2 Object creation

Object creation starts with finding the target node of the distributed object to be created, based on its runtime component given by the user. Then, the object creation command must be given by calling the object creation method of the object factory on the target node. This method is a synchronized and remotely accessible method and is responsible of creation objects locally. The target node, which can be the same node as the initiating node, creates the object when asked. The factory objects together are responsible for distributing the objects on their target nodes. The mechanism of object creation is described in detail in Section 4.1.3 .

3.4 Distributed programming in Umple

The user starts by modeling the system using Umple, without initial regard for distribution. The user then performs an analysis to determine which classes should be distributed by specifying this using the Umple *distributable* keyword. The user indicates which objects should be on which runtime components using the required API of each generated constructor. Runtime components are specified as the last argument of the constructor. Using queued state machines, pooled state machines [30], or active objects in Umple the objects on different nodes can run concurrently.

After compiling the system, the user can place the runtime components inside the nodes using the configuration file. The configuration file can be changed based on desired deployment before running the system.

It is possible to take a non-distributed system and force all the classes to be distributed, using the *forced* keyword as shown in Snippet 34. It means the system will have one active node which starts the system and multiple passive nodes that create objects locally. If the user does not choose runtime components for objects using a configuration file, objects of each class would in this case be grouped

together a separate runtime component (i.e. one per class). This allows distribution of objects without changing the Umple code – in other words the special constructors are not generated or used in this case.

```
1  distributable forced;
```

Snippet 34

The developer must follow best object-oriented practices such as using mutator (set) and accessor (get) methods. In particular, he or she must modify attributes and associations using the Umple-generated API and refrain from creating public attributes. The developer must also refrain from accessing static attributes and methods.

In an object-oriented system, the control flow starts from one of the nodes (running the main class in Java). Objects are created on specified nodes and remote method calls distribute the flow to different nodes. Each object on any node can then create other objects of any class on the same node or on other nodes. While the code is the same on each node, each node runs a different algorithm at runtime due to the different objects having different states.

By default, one of the nodes starts the control flow of the system by running a class with a main method (active node). Alternatively, there is another way of developing a concurrent system by having more than one active node in the system. The user can design the system to have multiple entry points to the control flow by running different or same implemented main classes on the different nodes (more than one active node) as shown in Figure 9. Each node can create different objects within itself and connect to other nodes. This way the developer can add nodes and objects to the system at any time.

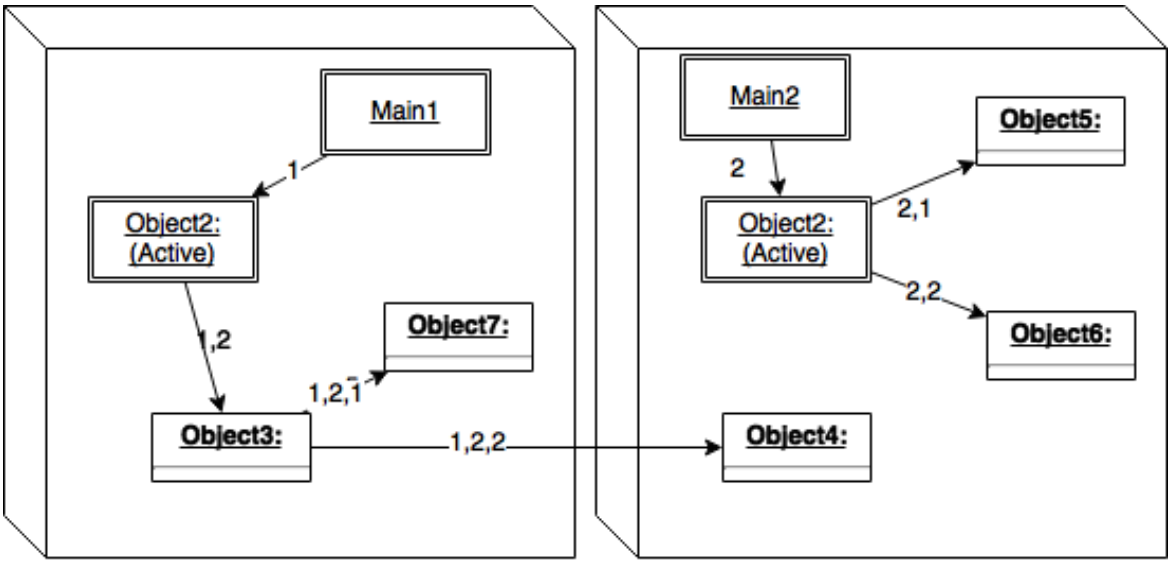

Figure 9: A system with two entry point of control flow

3.4.1 Special cases

3.4.1.1 Static code

The static environment of a class, including static attributes and static methods, is a separate environment from the dynamic part of the class and can run independently. The static methods of a class can be called from anywhere in the program.

In a distributed system, we have two choices for the static environment of each class. It can be defined as one static environment of each class for the whole system, or it can be defined as one static environment on each node. If we choose the first approach, the static environment should be accessed remotely for every class in the distributed system (even non-distributed classes). In good object-oriented design practice, the use of the static environment should be limited anyway, and should only be accessed through specific design patterns such as singleton 3.4.1.2, discussed in the next subsection. Umple does not accept static attributes and shows a warning if the developer defines them. As a result, and for the sake of simplicity, we use the second approach – Umple does not support remote access of static methods and static attributes are duplicated on every node.

3.4.1.2 Singleton classes - a special case

Singleton is a design pattern which enforces that there would be only one instance of the class in the system. This design pattern is supported by Umple and is implemented by making the constructor private and adding a static getInstance() method in the class to access the single instance of the class.

In a distributed system, a singleton class could mean two different semantics. One is that the class has only one instance on each node. The other semantic is to assume that there is only one instance of the object in the whole system. The developer can choose which semantic to be used for the singleton class. If a singleton class is also distributable, it means there should be only one instance of the class in the whole system and remotely accessible by other nodes. Otherwise, there can be one instance on every node.

Since a singleton class has a private constructor and can only be accessed using the getInstance() method, the factory does the object creation differently for such classes. The target runtime component of the singleton object is the name of the class by default. All of the getInstance() calls in the entire system will be redirected to a single object (Snippet 35). However, a specific runtime component can be given as a parameter of the getInstance() method (Snippet 36). This way, there will be a single object on each node. Alternatively, the object on the local node can be accessed using the "local" runtime component (Snippet 37).

```
1 SingletonClass.getInstance().getName();
```

Snippet 35

```
1 SingletonClass.getInstance(UmpleRuntime.getComponent("component1")).getName();
```

Snippet 36

```
1 SingletonClass.getInstance(UmpleRuntime.getComponent("local")).getName();
```

Snippet 37

3.4.2 Checks on the model and warnings

3.4.2.1 Static attributes

Since static attributes are considered global and Umple does not support remote calling of static methods, if any class of the distributed system has a static attribute, there is a chance of unexpected results in the system. Although static attributes are not acceptable in Umple (they are parsed as extra code) Umple warns the user and suggests refraining from designing a distributed system with static variables.

3.4.2.2 Association and object passing between non-distributed and distributed classes

The non-distributable object can call methods on the distributable object regardless of the location of the objects. However, the distributable object can only call the methods of the non-distributable object on the same node where the distributable object has a reference of the non-distributable object. In Umple, non-distributable objects are passed by value when making remote calls. The result is to send a copy of an object over to other node; it is not a reference to the same object. In Figure 10 we see the class diagram of a system with 3 classes. There is an association between class A which is distributable and class B which is not distributable. In Figure 11 the sequence diagram of the system is shown. Object ‘c’ creates an object of type B named ‘b’ with initial value of 1. When ‘a’ calls “getB(0)” on object ‘c’ it returns a reference of b. Then, ‘c’ changes the value of ‘b’ to 2. When ‘a’ calls “getValue()” method on its reference to ‘b’, it returns 2.

When the system is distributed, for example ‘a’ and ‘c’ can be on different nodes as in Figure 12. When ‘a’ calls “getB(0)” on ‘c’ it returns a copy of ‘b’ (pass by value) with serialization. When b changes the value of b to 2, it does not change the value of its copy on the other node. When ‘a’ calls “getValue()” on b, it returns 1 which is not the expected result.

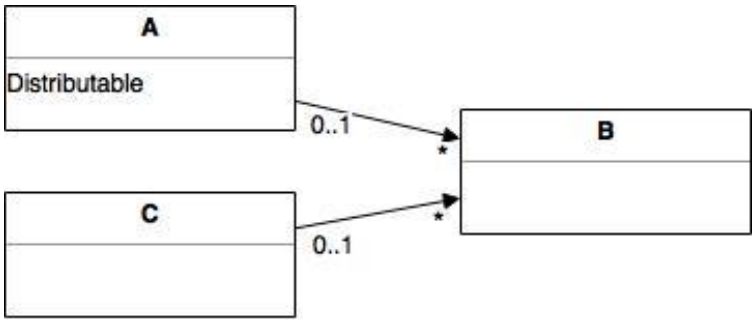

Figure 10: Sequence diagram with association between distributable and non-distributable class

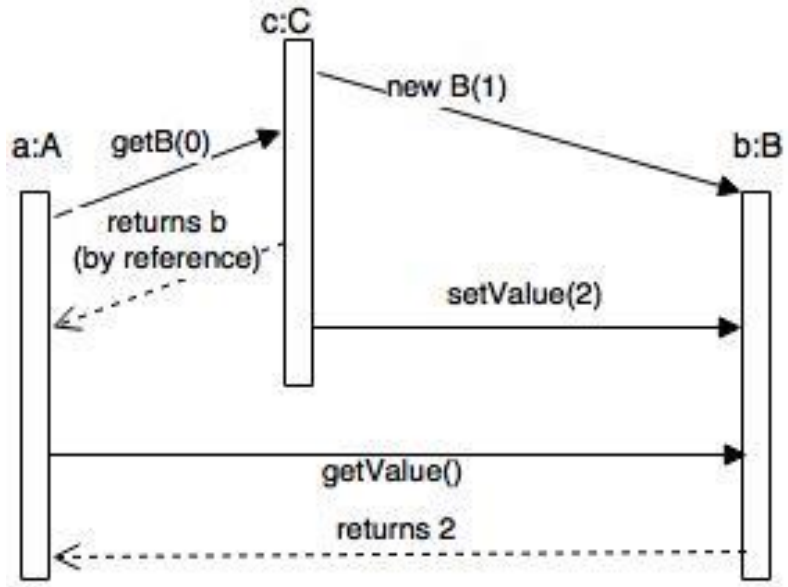

Figure 11: Sequence diagram of the system with a distributable and non-distributable class

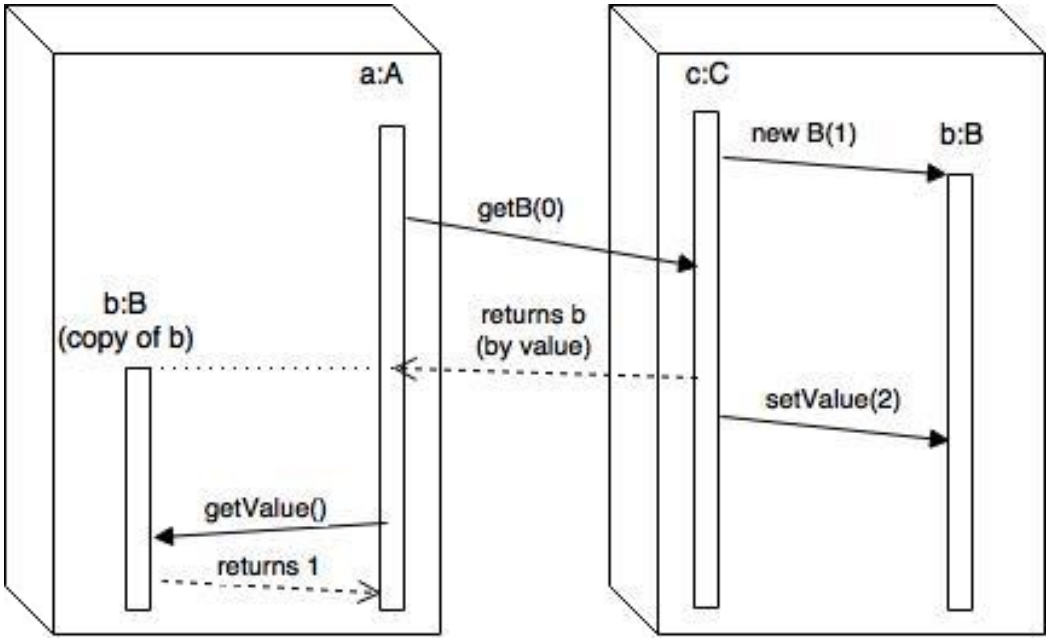

Figure 12: Sequence diagram of a distributed system

Unless an object is immutable, meaning it does not change state after creation, then method calls on the copies of the object on different nodes might return different results.

Because the location of the target non-distributed object might not be the same as the calling distributed one, any association or composition relationship between a distributed class and a non-

immutable and non-distributed class is dangerous if the direction of the association is towards the non-distributed.

Therefore, the developer is notified of such cases. Umple checks the model and warns the user with warning number 7002. The user can then decide either to fix the issue or ignore the warning.

4 Implementation of Distributed Objects in Umple

We implemented three different approaches to generate code for distributed systems. These are described in sections 4.5 (RMI stub as proxy), 4.6 (generating a client-side proxy), and 4.7 (generating a class as a proxy to itself). These approaches focus on generating readable code consistent with the user code which is integrated in the model. The developer can choose which of the patterns to be used as an optional argument of the distributable directive. The default, which can be omitted or given as 0 specifies the third approach (Snippet 38), 1 specifies the second approach and 2 specifies the first approach. The reason why we discuss the default approach last is that it is the most sophisticated.

```
Distributable on 0;
```

Snippet 38: Example of choosing the pattern for the generated distributed code

In the current implementation of the distributed feature, Umple only generates code for distributed systems in Java. The communication technology can be chosen for each class and currently is either Java RMI or web services, although web services only work with the third approach.

In Section 4.1, we explain the implementation of the UmpleRuntime class generated for distributed systems. Section 4.3 describes the generated code to utilize Java RMI for remote calls. Section 4.4 describes the necessary generated code to utilize web services. Sections 4.5, 4.6, and 4.7 describe three different approaches of generating code using different patterns. Each approach has different benefits and shortcomings but the third approach is the recommended pattern.

4.1 UmpleRuntime class

As we discussed starting in Section 3.3, for any distributed system, a special factory class is automatically generated to wrap the functionality needed for each node to create objects. It collaborates with other nodes for this purpose. This class is called ‘UmpleRuntime’. It follows the singleton pattern to only allow one instance of the class. This class includes a main method to start a node that waits for object creation commands, initialize the node by reading the configuration files, and connect to other nodes.

UmpleRutime also includes a method called “getComponent” which is a static public method and can be used to get the runtime component by name in the program. This method is used by the developer as an API to pass the runtime component to the constructor of the distributed objects as shown in Snippet 39.

```
A=new A(UmpleRuntime.getComponent(“componentName”));
```

Snippet 39

UmpleRuntime wraps the UmpleComponent and UmpleNode classes. By this, we mean that these classes are not accessible by the user and are only used by the runtime system. It also includes the factory methods which create objects on their local nodes.

The UmpleRuntime class is generated by Umple from a template and is specific to each application because different applications have different classes and different constructor arguments.

4.1.1 UmpleNode

UmpleNode is a Java class inside the UmpleRuntime class that represents nodes. UmpleNode’s variables includes name, id, ip, port, URL. They are addressed by setter/getter methods. When a node starts, a list of nodes is created in the UmpleRuntime based on the configuration file.

4.1.2 UmpleComponent

UmpleComponent is another class that represents a runtime component and has a name and an UmpleNode. A list of UmpleComponents is created in the UmpleRuntime based on the configuration file when a node starts.

4.1.3 Object Creation in UmpleRuntime

For each distributed class in the system, there are two methods in the UmpleRuntime class: ’new’ and ‘create’.

The ’new' method has a name of the form: “new” + ClassName. The parameters of this method are the same as the parameters of the constructor of the particular class with an optional runtime component as the last parameter. This method finds the target node of the object based on the runtime component and calls the ‘create’ method on that node. It then returns a proxy of the remote object.

The ‘create’ method has a name of the form: “create”+ClassName. It is responsible for creating an object locally. This method should also be accessible for remote calls by other factories and return a reference to the created remote object. It has the same parameters as the constructor of the class. This method creates a real object and a remote object (if separate as in the third approach described below), exposes the remote object, and returns a reference to the remote object. An example of the object creation process is shown in Figure 13.

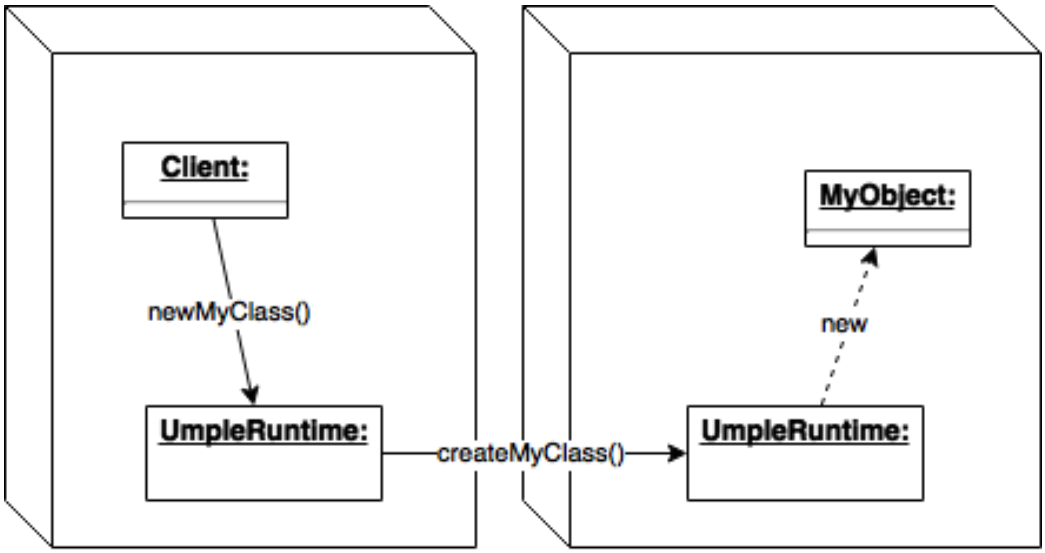

Figure 13: Creation of an object by the factory

There should be one and only one UmpleRuntime instance on each machine. We used the singleton pattern for this class. Therefore, the UmpleRuntime class has a private constructor. This pattern ensures that only one object of the class exists and it can be referenced everywhere in its node.

To simplify the developer’s conceptual model of the code in their system, it is preferred that a client does not see UmpleRuntime and creates the object as though it was a local object. Therefore, the object (or its proxy) is responsible for communicating with the UmpleRuntime object on its node and delegate method calls to the target node. Figure 14 shows how Object1 is created locally and Object2 is created on a remote node. Both objects inform UmpleRuntime after being created. Since Object2 is meant to be created on the remote node, UmpleRuntime asks the other node to create the object and return a reference. The user should not know that Object2 is a proxy. This is one of the goals of this chapter.

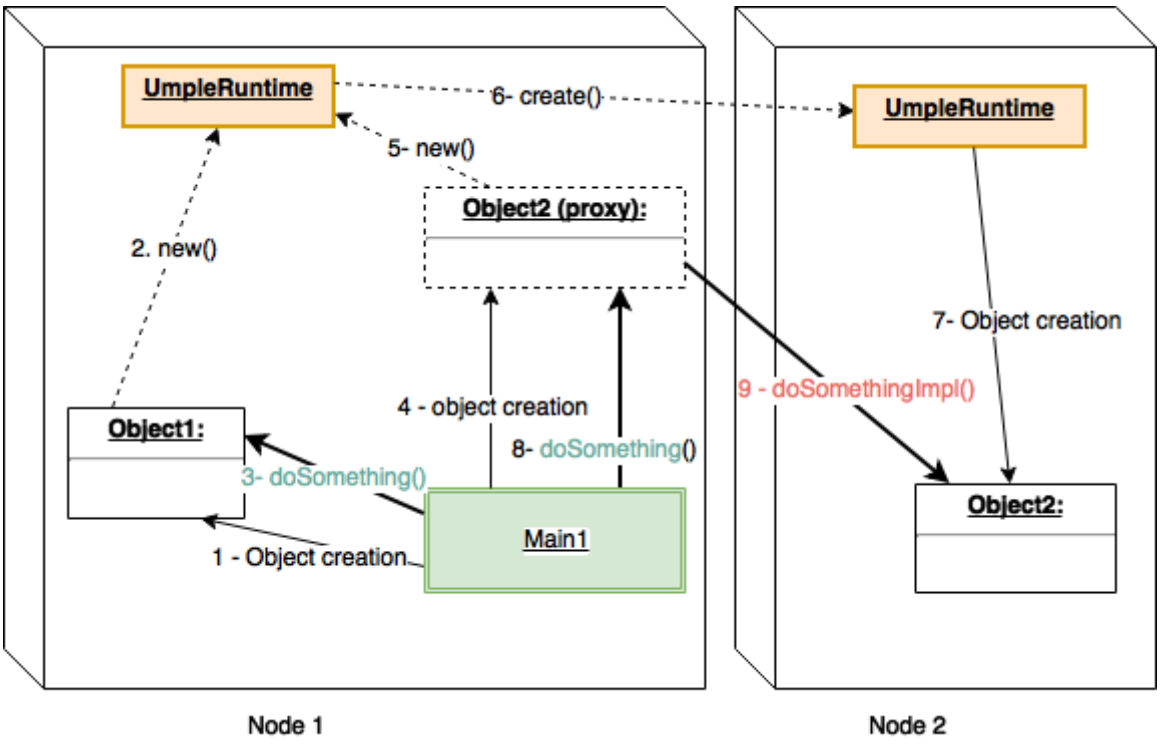

Figure 14: An example of sequence of object creation on two nodes

Explanation of Figure 14:

1. The Main1 object (with main method) creates the object by using the “new” statement; Object1 is created.
2. Object1 calls new() method of UmpleRuntime, UmpleRuntime does not do anything because Object1 is on its target node. The new() method returns and Object1 knows that it is on its target node
3. Main1 calls a function doSomething on Object1 and Object1 returns the result.
4. The Main1 object (with main method) creates the object by using the “new” statement; Object2 is created.
5. Object2 calls new() on UmpleRuntime. UmpleRuntime sees that the target node is on Node 2.
6. UmpleRuntime calls create()on UmpleRuntime on Node2.
7. UmpleRuntime creates an Object2 on Node 2 and returns the reference. The reference goes back to Object2 on Node 1. Object2 on Node 1 becomes the proxy to Object2 on Node 2.
8. Main1 calls doSomething on Object2(proxy).

9. Object2(proxy) calls doSomethingImpl() on Object2. The result of the method call is then returned.

Every object that calls functions on another object needs to have a reference of that object to call functions on. The references should be passed from the creator of the object to other objects. In the distributed system, these references are the proxies. Therefore, for each node that calls a function on a distributed object, the proxy is addressed instead. For example, if there are 1000 nodes in the system, the proxies of Object1 are created only on the nodes that use it.

We also want to be able to have multiple entry points in the system by defining active objects or multiple main methods (and Main objects). Figure 15 shows how Node2 can start independently with Main2 and create an object on the first node. In this figure, the same process of Figure 14 happens. However, the steps 4 to 9 are named 2.1 to 2.5 and do not need to be after step 1.3.

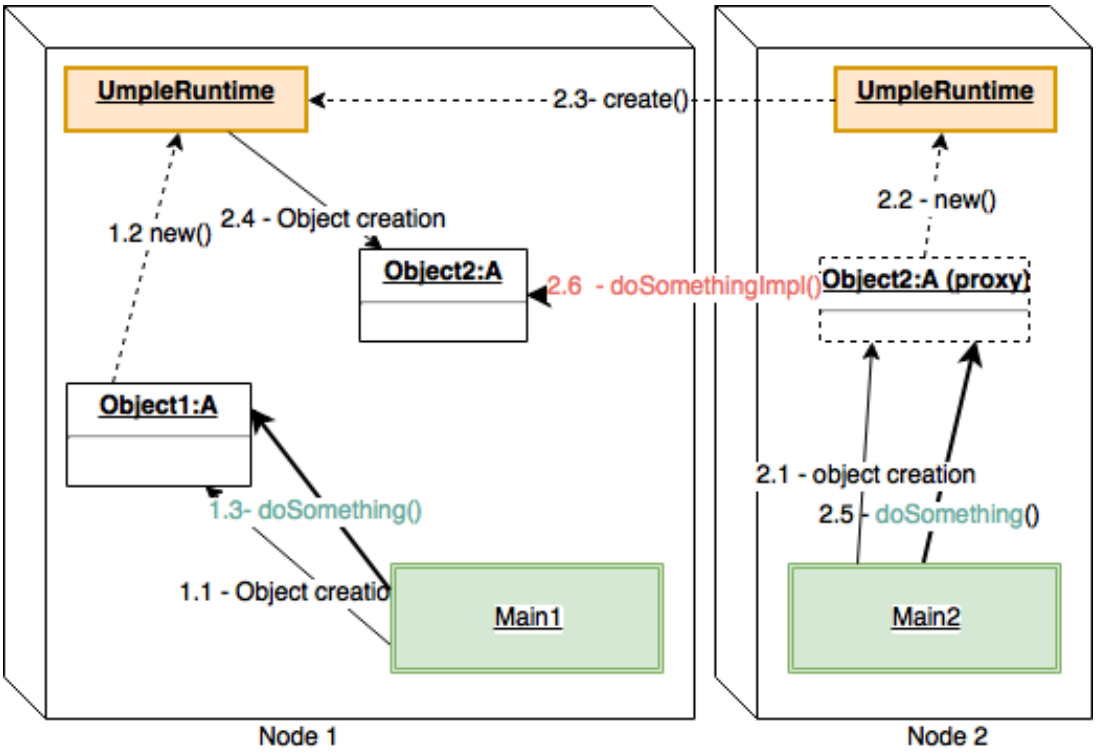

Figure 15: Example of distributed system with multiple entry points

4.1.3.1 Singleton Object Creation

As discussed in Section 3.4.1.2, a singleton class is a class with only one instance in the whole system. In Umple, the user can define a certain class as singleton and Umple generates code that supports the singleton pattern. When the system is distributed and the singleton object is defined as distributed, the factory must make sure to only instantiate the singleton object on a certain node (currently the first node accessing the object) and make it accessible by other nodes by setting up proxies on the other nodes. Since a singleton class has a private constructor and can only be accessed using its getInstance() method, the factory needs to do the object creation differently for singleton classes.

Therefore, after the 'new' method finds the target node and calls the 'create' method for object creation; instead of instantiating an object with the 'new' statement, the 'create' method calls the getInstance() method of the class. Figure 16 shows an example of Node1 creating the singleton object and Node2 accessing it by creating the object.

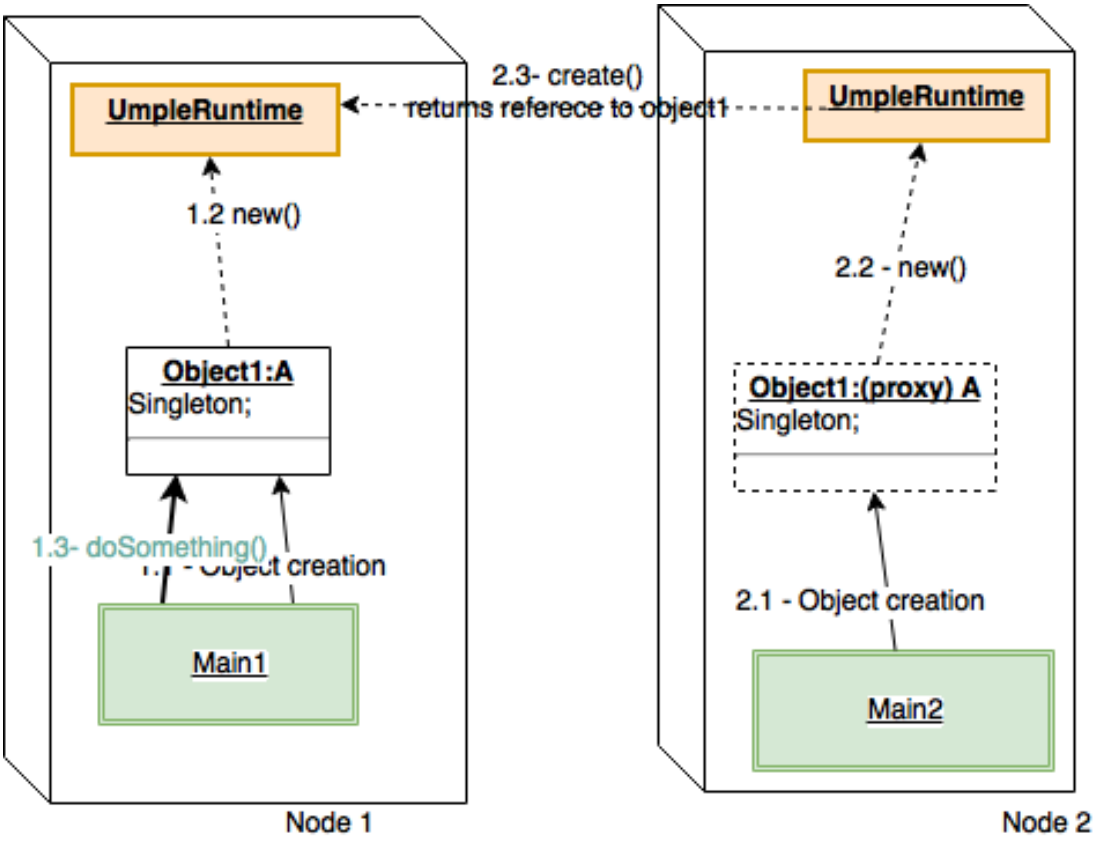

Figure 16: Example of singleton object

Explanation of Figure 16:

1.1 The Main1 object (with main method) creates the object by using the “new” statement; Object1 is created.

1.2 Object1 calls new() method of UmpleRuntime. UmpleRuntime does not do anything because Object1 is on its target node. The new() method returns and Object1 knows that it is on its target node.

1.3 Main1 calls a function doSomething on Object1 and Object1 returns the result.

2.1 The Main2 object (with main method) creates the object by using the “new” statement; Object1(proxy) is created.

2.2 Object1 calls new() method of UmpleRuntime, UmpleRuntime sees that the target node is on Node 2.

2.3 UmpleRuntime calls create() on UmpleRuntime on Node2. It does not create the object1 because it is already created. The reference of Object1 is returned.

4.2 Background: Remote Procedure Call

The technologies that provide remote procedure call (such as RMI and Web services) use the proxy pattern to provide users with a mechanism to call methods on the remote object.

The proxy is an object on the client side that redirects the method calls to the server. The client must first create the proxy object, connect it to the remote node(server), and then the methods can be called on the proxy. The proxy is also called a “stub”. After creation of the stub it can be used to call methods until the connection is stopped. Before the client can create the stub, the server must first be able to be found on the network. This is called exposing the object.

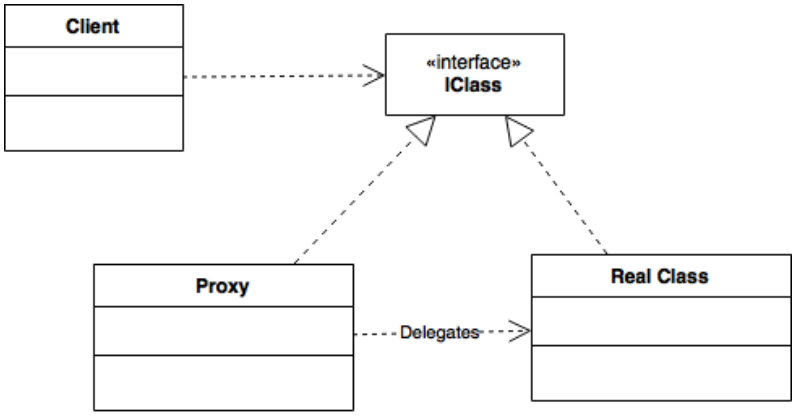

Figure 17: The proxy pattern

The underlying infrastructure of the RPC framework is responsible of packing the method calls and parameters into messages (marshalling), sending and receiving the messages, decoding the message into method name and parameters (unmarshalling), and calling the method on the remote object with parameters. Java RMI is a standard technology for distributed systems when all the nodes are Java based. On the other hand, web services can be used to create distributed systems on different platforms. We created our system first with Java RMI. We then created an additional implemented of it with Web services following the same semantics.

4.3 Background: Java RMI

RMI [8] is a Java-based lightweight framework for development of distributed object models. facilitates the communications between distributed objects on different machines. Java RMI provides remote communications by providing stubs of a remote object to the developer. These stubs implement the same interface as the object and it is used as the reference to the remote object. The client calls the methods on the stub instead of a remote object, then RMI invokes the method on the remote object and returns the result. This way the remote object can be called as if it was local. RMI stubs can be sent to other nodes and client does not have to reinitiate the connection to the server.

The developer uses an interface to indicate which methods are accessed remotely. This interface must extend the "remote" interface, and methods might throw ‘remote exception’. Every remote call on the object is through the stub which implements this interface. Since the methods throw exceptions, the caller needs to catch the exceptions. Any remote object can be exposed by casting it to UnicastRemoteObject.

```
1  public interface IVendor extends Remote{
2    public String getName() throws RemoteException;
3  }
```

Snippet 40: An example of a communication interface for Java RMI

For example, if the class Vendor has a method called getName(), which we want to make remotely accessible, we should first implement an interface (eg. IVendor in Snippet 40). This interface should extend Java's 'Remote' interface, include the signature of the getName() method, and arrange for each method to throw RemoteException.(Snippet 40)

To expose a remote object, there are two different ways: one is by a remote class extending the UnicastRemoteObject class, and one by casting the instances to UnicastRemoteObject:

The distributable class can extend 'UnicastRemoteObject' class: This is not possible if the class has another superclass in the model. An example is shown in

```
1  Person aPerson= new Person();
2  //…. Any code
3  PersonInterface stub=(PersonInterface)UnicastRemoteObject(aPerson);
```

Snippet 42: Snippet 43: Exposing a remote object by casting

```
1  public class Person extends UnicastRemoteObject implements PersonInterface{
2    String name;
3    public Person(){}
4    public String getName(){
5      return name;
6    }
7  }
```

Snippet 41: An example of a remote class exposed by extending UnicastRemoteObject

* Casting the object to UnicastRemote at runtime. The object is exposed after the casting and methods can be called remotely. This way the class implements the interface but does not extend the 'UnicastRemoteObject' class. An example of casting to 'UnicastRemoteObject' at runtime as shown in There should be an RMI registry on each Java machine so that

```
1  Person aPerson= new Person();
2  //…. Any code
3  PersonInterface stub=(PersonInterface)UnicastRemoteObject(aPerson);
```

Snippet 42: Snippet 43: Exposing a remote object by casting

There should be an RMI registry on each Java machine so that Remote objects can also be registered with a name on the RMI registry so that they can be looked up by other machines. The developer must first create a registry and then bind the object to it (Snippet 44).

```
1  Registry rmiRegistry = LocateRegistry.createRegistry(“localIp,port);
2  rmiRegistry.rebind(“Person", stub);
```

Snippet 44: Registering an object on RMI registry

Registered objects can be located by the client. The client must first locate the registry of the server (Snippet 45 line 1). Then it looks up the name of the remote object, and creates a stub (Snippet 45 line 2). If the object is exposed, on the client side, RMI can connect the stub to the remote object. The stub can be used as a proxy to the remote object.

```
1  Registry rmiRegistry = LocateRegistry.createRegistry(remoteIp,remotePort);
2  PersonInterface stub = (PersonInterface) registry.lookup("Person”);
```

Snippet 45: Creation of a stub object on client-side

Stubs can be serialized and sent to other nodes. Java RMI reconnects the stub to the remote object after deserialization. Therefore, there is no need for locating the remote class and initiating the connection after sending the stub to another node.

4.4 Background: JAX-WS (Web Services)

The remote method calls from proxy to the remote object can also be implemented using web service communication. This is implemented in Umple by the Java API for XML Web Services (JAX-WS) library which is a SOAP-based web service library for Java.

The main difference between RMI and web services in the implementation is that the latter does not create a dynamic stub object on the client-side. In other words, the web service’s proxy object is not Serializable. As a result, the proxy does not maintain the communication after being moved to another node. Therefore, creating an object-oriented system with objects sending references to each other is

not possible without creating a proxy that can reconnect to its server even after being sent to another machine.

Similar to RMI, using the Javax.xml library, there is a need for an interface that should be preceded by the annotation “@WebService". All the methods that are going to be remotely accessible must be preceded by “@WebMethod”. Snippet 46 shows an interface with a method annotated for this purpose. It also shows what libraries must be imported by the interface for the annotations.

```
1  import javax.jws.WebMethod;
2  import javax.jws.WebService;
3  import javax.jws.soap.SOAPBinding;
4  import javax.jws.soap.SOAPBinding.Style;
5  @WebService
6  @SOAPBinding(style = Style.RPC)
7  public interface PersonInterface {
8  @WebMethod
9  public String getName ();
10 }
```

Snippet 46: An example of the annotated interface for RPC style web services

In addition, the remote class itself must be annotated as well (Snippet 47) ecomSystem is the package name and PersonInterface is the interface for communications.

```
1  @WebService(endpointInterface = “ecomSystem.PersonInterface”)
2  public class Person implements, PersonInterface {
3  public String getName (){
4  return name;
5  }
6  // remaining parts of the class
```

Snippet 47: An example of the annotated class for RPC style web services

To create a web server, the remote object must be published as a server. The object should be published on a URL decided by the developer. If the object is being created on the same host, the service should be published on 'localhost' using a URL to distinguish the object from other objects. Snippet 48 Shows how a Vendor object is published as an endpoint. The object is exposed for remote calls after being published.

```
1  Endpoint.publish("http://192.168.1.1:8080”+”/Person”, aPerson);
```

Snippet 48: Publishing as an endpoint

The client needs to connect to the service by first requesting the WSDL. It finds the server on the URL with the published name and creates a ‘service’ object of Java's Service library. It then creates a proxy using the service object. Every call to this proxy is redirected to the remote object as a service request.

The example in the Snippet 49 shows the necessary code for creating the service and proxy.

```
1  URL url = new URL("http://192.168.1.1:8080/Person"+"?wsdl”);
2  QName qname = new QName(http://ecomSystem/“, PersonService");
3  Service service = Service.create(url, qname);
4  proxy=service.getPort(PersonInterface.class);
```

Snippet 49: Client connecting to the server using JAX-WS

4.5 Initial approach: Generating Java RMI code

This approach is partially implemented in Umple as our initial approach, but using it is not recommended for reasons we describe at the end of this section. We describe this approach here because its weaknesses motivate the subsequent approaches.

In this approach, we utilize standard Java RMI. Each distributable object is exposed as a remote object using casting. A proxy object (RMI stub) is created on the client-side by RMI. The stub implements the remote interface which is generated by Umple. This interface must include all the methods that are going to be called remotely. The generated class is the ‘implementation class’, which is an RMI remote class and the interface is the ‘communication interface’, which is an RMI remote interface.

Unless the remote interface is defined by the user in the Umple code, each of the public methods of the class should also exist in the interface that is going to be used by RMI. The signatures of the remote interfaces are the same as the signatures of the actual class. In RMI the interface must implement the “Remote” interface and each method should throw “RemoteException”. The package name is the same as the implementation class. Snippet 50 shows an example of the generated communication interface.

For each distributable class, Umple generates the communication interface automatically. This interface includes signatures of all public methods of the class. In addition, it extends Java's "Remote" interface and all the methods throw "RemoteException". The name of the class is used for the interface and the name of the class is changed to the implementation class name, namely ClassName+ "Impl". Therefore, any use of this class is always through the interface and the user does not access it directly. Snippet 51 shows an example of the generated implementation class.

```
1  interface MyClass extends Remote
2  {
3    public String getName () throws RemoteException;
4  }
```

Snippet 50: The communication interface generated for a class named "ClassName"

```
1  class MyClassImpl implements MyClass
2  {
3    public String getName ()
4    {
5      return name;
6    }
7  }
```

Snippet 51: The implementation class generated for a class named "ClassName"

In this approach, the developer asks for object creation by calling the 'new' method of the UmpleRuntime object. and the factory returns a reference to the stub as shown in Snippet 52. All the methods on the interface, which are all the public methods of the class, can be called remotely or locally thereafter. RMI handles the communications and stub creations.

```
1  MyClass X = umpleRuntime.newMyClass(umpleRuntime.getComponent("component1"));
```

Snippet 52: Object creation in the first pattern

Because web service proxies are not serializable, this approach does not work using web services. Due to this fact and the problem with using try-catch, there is a need for different patterns in which there is no need to change user code or force the user to use try-catch.

4.6 Second Approach: Generating an extra Client-Side Proxy

To catch the remote exceptions in a modular way, and to be able to implement the distributed system with other technologies, Umple creates a proxy class on the client side. This proxy handles a reference

to the implementation class by redirecting the method calls to the RMI stub or web service proxy. Therefore, for each distributable class in the model, Umple generates two classes: An implementation class and a proxy class. For each object being created in the system, there is a real object which is an instance of the implementation class and some proxy objects which are instances of the proxy class. The real object of the implementation class resides on the target node and is a remote object. There can be a copy of the proxy object on all other nodes that accesses the object. The proxy objects redirect the method calls to the real object. The sequence diagram of a method call is shown in Figure 18.

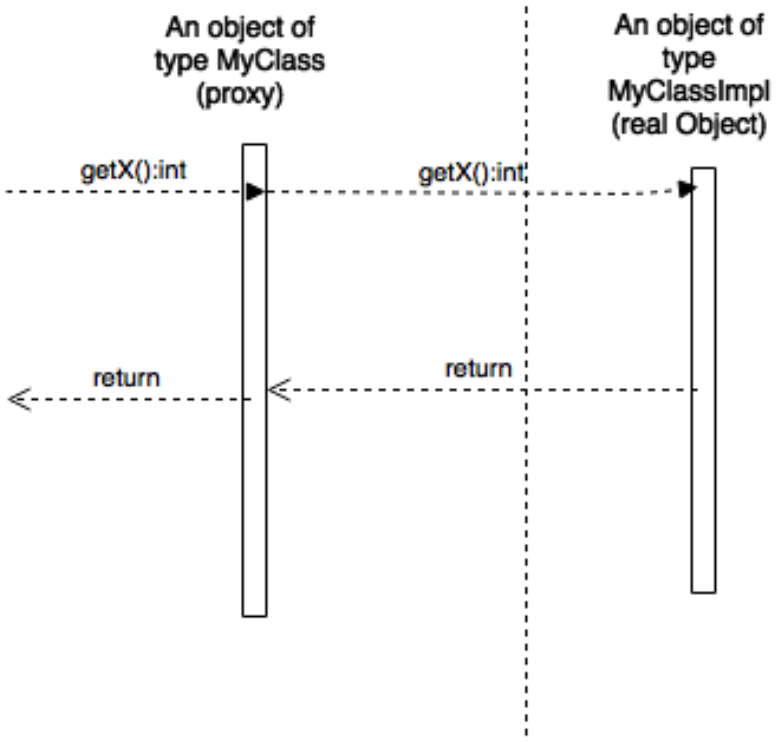

Figure 18: The sequence diagram of a remote call in the client-side only proxy approach

The implementation class contains all the generated code of the source class and has a name of the form ClassName+ "Impl" (like in the first approach). The proxy class has the same name and the same method signatures as the source class, but the implementations of the methods include redirecting every method call to the real object using the proxy (RMI stub) of the remote object. The proxy uses the class's actual name and the name of the automatically generated communication interface is in the form "I"+ClassName+"Impl". The class diagram of the generated system is shown in Figure 19.

Snippet 53 and Snippet 54 show examples of the generated implementation class and communication interface for a class named “MyClass”. More details of the generated code are described in sections 4.6.2, 4.6.3, and 4.6.4. Note that each of the classes and interfaces will be generated as a separate file and is annotated based on the communication technology chosen for distributed class. (currently only RMI).

```
1  interface MyClass extends Remote
2  {
3  public String getName () throws RemoteException;
4  }
```

Snippet 53: Communication interface generated for a class name "ClassName" in second pattern using Java RMI

```
1  class MyClassImple implements MyClass
2  {
3    public String getName ()
4    {
5      return name;
6    }
7  }
```

Snippet 54: Implementation class generated for a class named "ClassName" in the second pattern using Java RMI

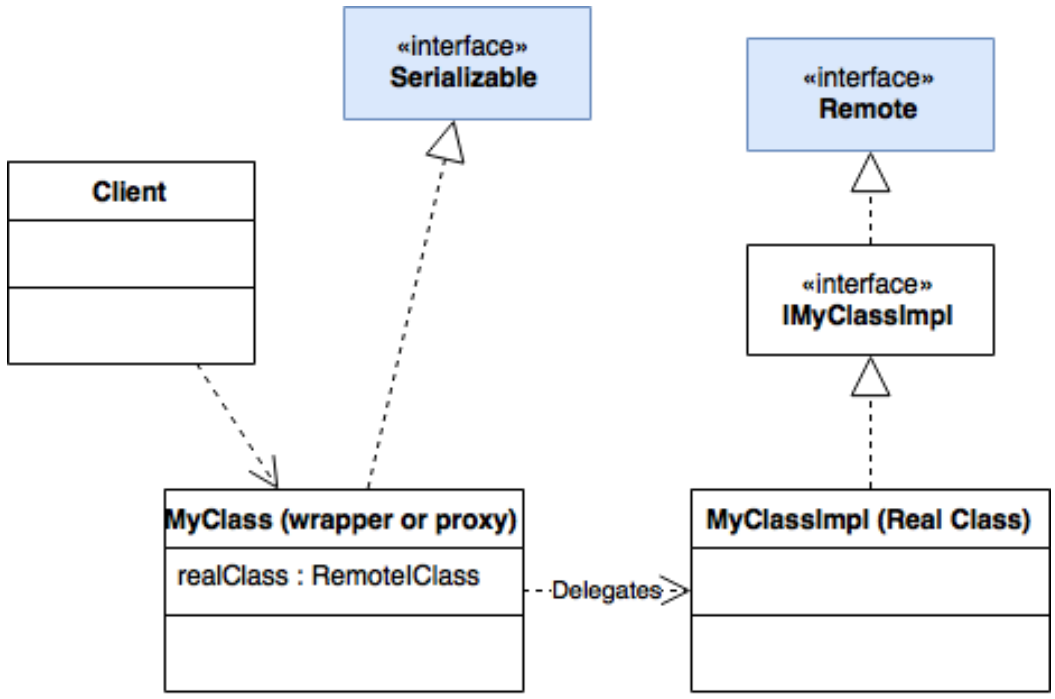

Figure 19: Class diagram of the generated system using the second pattern

4.6.1 Inheritance issues

To support the methods of the superclass of a class, the proxy class must be a subclass of the proxy of the class's superclass. This way if a method, which is not overridden by the class, is called on the proxy object, the call is redirected to the reference by the method on the super class's proxy. Therefore, Umple creates proxy classes for each distributed class and their super classes, so the proxies extend the proxy of the class's superclass. This way the same inheritance tree of the class is followed both by implemented classes and proxy classes.

4.6.2 Generated Code for the Proxy Class

The implementations of the methods of the proxy class includes calling the same method with the same parameters on the stub. The stub calls the method on the implementation object which was created on the designated node by the object factory. Since this call might throw an exception, it needs to be caught using try-catch scopes.

There can be different ways to handle exceptions, but currently Umple creates a code for 'retry' mechanism. The proxy catches the exceptions and if an exception occurs, it waits for a second and retries the method call. This is a basic implementation to handle the exceptions but the developer still has to handle the exception to avoid infinite retry. Snippet 55 is an example of the proxy class with getName() method which delegates the call to the real object which is an RMI proxy (stub) to the real object.

```
1   class ClassName{
2     IClassNameImpl realObject;
3     public String getName (){
4       while(true){
5         try{
6           return realObject.getName();
7         }
8         catch (Exception e){
9           System.err.println(e.toString());
10          Thread.Sleep(1000);
11        }
12      }
13    }
    }
```

Snippet 55: The proxy class generated for a class named "ClassName" in the second pattern

Each proxy has a reference to the real object using the variable called “realObject”. This variable is a reference to the RMI stub. The proxy connects to the real object and delegates the method calls to the real object (see line 6 of Snippet 55). This realObject can be the proxy of any RPC technology. Therefore, any RPC communication technology can be implemented using this pattern.

4.6.2.1 Constructors

Using the proxy class, the ‘new’ factory method can be called inside the constructor of the proxy instead of being called by the user directly. This way, the developer creates a proxy object in the same way they create a local object using the “new” statement. Then, the proxy object calls the new method on the object factory. The object factory returns the reference to the created remote and real object.

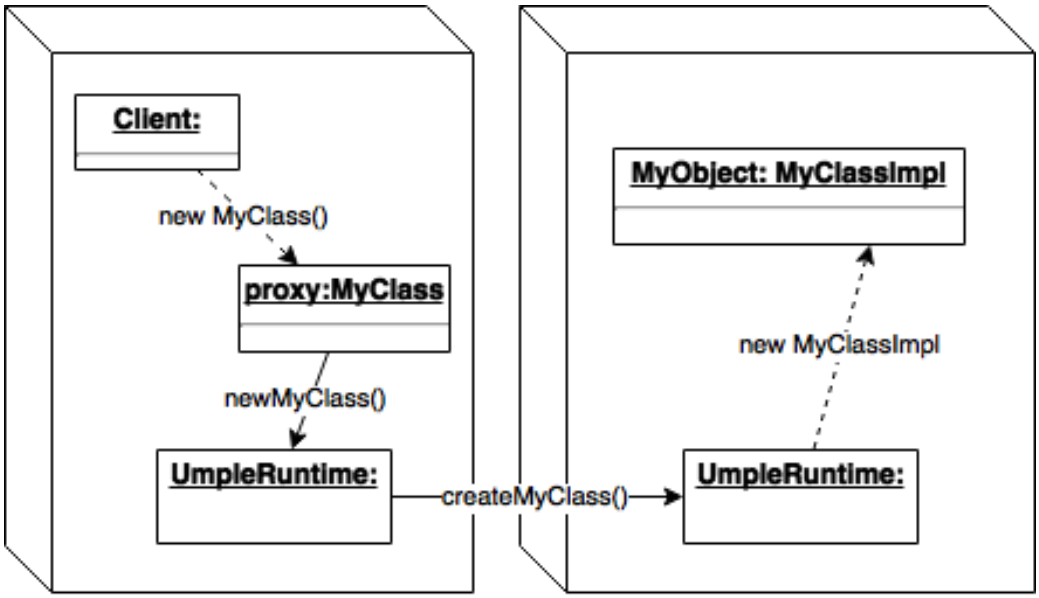

Figure 20: The creation of a remote object by factories

The name of the runtime component can be given to the constructor of the proxy class as an optional parameter. Therefore, there are two constructors in the proxy class for each constructor of the actual class (overloading the constructor). One with the same signature as the actual class, and one with one extra argument of type “UmpleComponent”.

The constructor without the extra argument calls the other constructor with the runtime component of the name of the class. Therefore, when the developer uses the new statement without mention of the runtime component, it is automatically considered to be on the runtime component with the same name of the class. The other constructor is needed so that the user can call the constructor with the

name of a specific runtime component. It calls the ‘new’ method on UmpleRuntime with all its parameters including the runtime component.

```
1  public Vendor()
2  {
3   if(this.getClass()== Warehouse.class)
4   realObject=UmpleRuntime.getInstance().newWarehouse(UmpleRuntime.get(“Vendor”), this);
5  }
```

Snippet 56: An example of the constructor of the proxy class

The job of the constructor is to call the “new” method on the UmpleRuntime to create the real object. UmpleRuntime then returns a reference to the proxy, and the “realObject" variable is set.

Every proxy object will first create its parent, which is an instance of the proxy class of the implementation class’s superclass. When a class has a superclass, the no-argument constructor of the superclass is called first. Since the proxy might have a parent object and the superclass’s constructor is called first, UmpleRuntime’s new method should only be called when in the constructor of the initial object. Using the getClass() method, the constructor can check if the instance is the a direct instance of the class or not. Snippet 56 shows how the no-argument constructor checks if the instance of the object is the same as the class (not child object) before calling the ‘new’ factory method.

4.6.2.2 The ‘equals()’ method

The ‘equals()’ method is generated (overridden) for the proxy class. The ‘equals()’ method by default compares if two variables are references to the same object. But since the proxies are not the real objects, the implemented equals method must compare the real objects of two proxies instead of comparing the proxies.

Since the “equals” method cannot be called remotely, the comparison is done by comparing the hash code of the objects. The ‘hashCode()’ method cannot be called remotely either, therefore, a ‘getHashCode()’ method is generated in the implementation class that returns the hash code of the object. The 'equals’ method uses this method to compare the hash codes of two proxy object (Snippet 57).

```
1  public boolean equals(Object obj)
2    {
3      if (obj.getClass()!=this.getClass())
4        return false;
```

```
5     return (getHashCode()==((Vendor)obj).getHashCode());
    }
```

Snippet 57: equals method for a class called Vendor

Umple already generates hashCode() and equals() methods based on the key if one is defined by the developer. The getHashCode() method uses the hashCode() method generated by Umple. However if the user implements the equals() method manually (without using the key) it is not supported and will not be used.

4.6.3 Changes in the Implementation Class

The implementation class generated is the same as the generated code for a non-distributed class with minor changes. The name of the implementation class is changed from “ClassName” to “ClassNameImpl” for any class name. Of course, the name of the constructors of the class must also be renamed to “ClassNameImpl” to match the new name of the class. The class implements the communication interface.

4.6.4 ‘New’ method of UmpleRuntime

The new method of the UmpleRuntime class has the same set of arguments of the constructor of the class with an extra argument of type UmpleComponent. It calls the create method on the target node to create the real object.

It first finds the node based on the name of the runtime component of the object, then finds the reference to the proxy of the UmpleRuntime system of the target node and calls the create method with the same arguments. The create method then creates the real object, exposes it, and returns a proxy of the object.

4.6.5 Reference to the real object inside the real object (“this” keyword)

One issue with this approach is the following: The user uses the keyword “this” in the body of a method and "this" object is sent to another method as an argument. Because the name of the proxy has been replaced by the name of the class, all other references to the object are then replaced with the reference to the proxy object in the generated code. Sending "this", which is a reference to the real object, does not match the type of the method signature. For example, let us assume that method1

is a method of Class1 and method2 is method of Class2; and c1 is an object of Class1 and c2 is an object of Class2. Suppose method 1 has a signature like in Snippet 58.

```
1  public int method1(Class2 aClass2)
2  {…}
```

Snippet 58: Example method signature to illustrate consequences of the ‘this’ keword

If method2 calls method1 with "this" (Snippet 59), there is no problem if Class2 is not distributed. But if Class2 is distributed, this will result in a type mismatch error. Because “this” is an object of the implementation class, but the parameter of method1 is in type of the proxy class.

```
1  public int method2()
2  {
     return method (this);
   }
```

Snippet 59: Example of referring to ‘this’

One solution to this problem is to set the proxy as superclass of the implementation class. Doing this will prevent the type error but it will send the real object instead of the proxy, therefore the same problem as with the first approach will happen and method calls will have to be wrapped inside a "try" scope (the proxy implements the communication interface). Another solution is replacing "this" with a reference to a proxy object. This can be done either by replacing the "this" keyword in the generated code or by asking the user to use such a reference. Therefore, for every distributed object, a reference to the proxy exists as a variable (Snippet 60). This reference is called "self” and is set to a proxy object by the object factory upon creation of the object.

```
1  ClassName self;
```

Snippet 60: Variable ‘self’ in the implementation class

The “self” variable must be used by the user as a keyword instead of “this”.

```
1  public int method2()
2  {
   return method(self);
   }
```

Snippet 61: Using 'self' instead of 'this' keyword

4.7 Third (best) approach: Generating Proxies on Both Sides

To solve the problems with using the "this" keyword in the second approach Umple needs to generate code so that the real object and the proxy object interchangeably. The real object must work as a proxy when sent to other nodes by serialization. On the other hand, using a server-side proxy also facilitates implementing other technologies such as direct TCP connections.

This pattern, which is the default pattern in Umple, is based on creating a dual-purpose object that can work both as a client-side proxy object and as real object based on its location. If the methods are called by a proxy, it works as the real object. Otherwise it works as the proxy to itself. When the object is sent to another node, the real object and its state stays in the same node. The object that is sent only works as a proxy to the real object.

Since remote objects cannot be serialized in RMI after being exposed (RMI sends the RMI stub of the object instead) and we want to send the object to other nodes; the real object cannot also be the remote object. Therefore, we need a server-side proxy which is a remote object. This remote object will delegate the incoming method calls to the real object. Indeed, the communication interface used in the first approach is also generated for the same purpose. The class diagram of the generated classes for each class is shown in Figure 21.

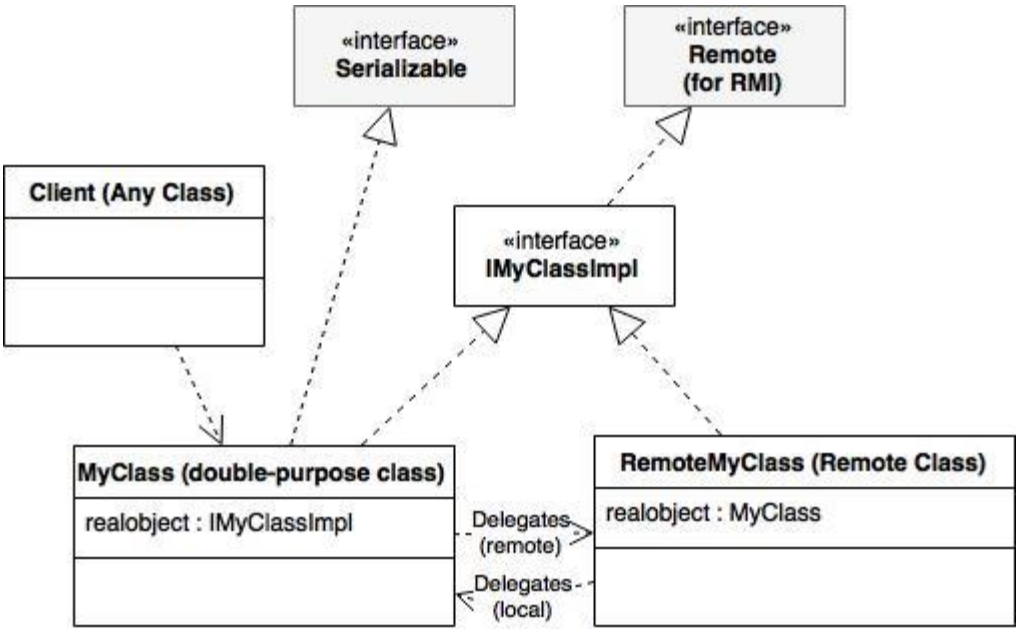

Figure 21: Class diagram of the third pattern

The dual-purpose object works as the real object when it is on its target node, in other words, unlike the second approach in which the real object is a remote object as well, in this pattern they are separate. However, the real object and the remote object always reside on the same node and the remote object redirects the method calls to the real object using local calls.

The remote object, calls the implemented methods on the real object. The remote class has a name in the format of MyClass+"Remote". It implements the communication interface, which is similar to the remote interface in the first and second approach.

The dual-purpose object works as the proxy object on all nodes that need a reference to the object. It redirects the nodes to the remote object. Figure 22 shows the redirections of a remote call from the proxy object to the real object. Note that when the object is on its target node, this object works as a proxy object to itself (without remote object) and does not do a remote call.

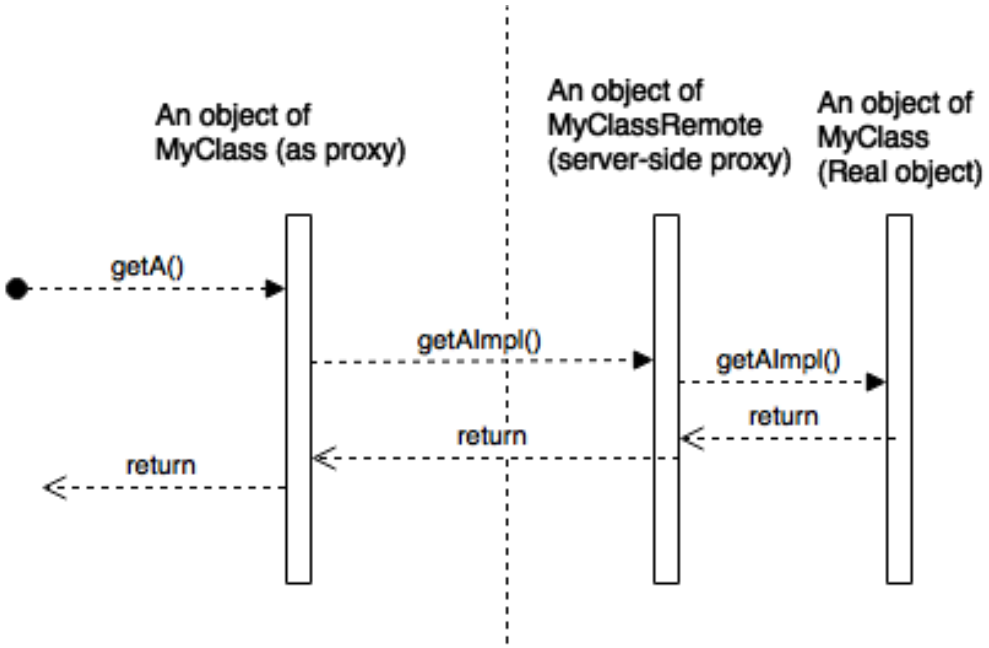

Figure 22 The sequence diagram of a remote call in the server-side proxy approach

4.7.1 **Generated dual-purpose class**

To create objects that can work both as the actual object and as client-side proxy, a dual-purpose class is generated by Umple. This class is named after the name of the source class. It implements the communication interface as well. It has a reference to the remote object named “realObject” as a

variable with the type of the communication interface. This variable is a reference to the RMI or web service proxy object. On the real object, this variable is set to “this”.

For each public method of the class, there are two methods in the generated class. One has the same name and signature of the method which only redirects the call to the realObject (delegating method), as well as another that has a similar signature but differs only in its method name (described below), which includes the implementation of the method (implemented method).

The implemented methods are only called by the remote object or the delegating methods. These methods contain the actual implementation of the methods. They change the state of the implementation class by changing the variables of the object. These methods are only called on the object which resides on the target node.

As shown in Figure 22, when the user calls a method on the object, the delegating method is called. The delegating method calls the same method on the realObject with all the parameters. RealObject can either be a remote object or the object itself. This call is wrapped inside a try-catch scope and the retries the remote call in case of failures (once each second). The developer only uses the delegating methods of the object because the implemented methods’ names are replaced by methodName + ”Impl”. For example; if the method’s name is “getName”, the implemented method would be named “getNameImpl”. This way the methods with the “Impl” postfix satisfy the functionality of the implementation class and the other methods work as a proxy.

Umple modifies the code of the generated implementation class to create a dual-purpose class as explained in following subsections.

4.7.1.1 Variables

The dual-purpose class is an implementation class, which also works as the client-side proxy. Therefore, the variables must be ‘transient’ so that they are not serialized and sent through RMI. The keyword ’transient’ is used to specify that the variable is transient.

All the variables of the client-side proxy in the second approach should also exist on the implementation class. These variables are realObject and the variables necessary for communication which are technology-specific. The variable realObject is a transient variable referring to “this” by default, but it is set to remote objects for proxy objects on other nodes.

The variables necessary for communication are not transient and are serialized so that they can be used by the readObject() method to establish the connection of the proxy with the remote object after moving the object to another node as proxy.

If the selected technology is RMI, there is a need for a variable ‘remoteObject’ which is the RMI stub to the remote object.

4.7.1.2 Methods

As mentioned before, the names of the implemented methods are changed with adding the postfix “Impl”, and the delegating methods of the client-side proxy are also added to the class. Therefore, there are two methods generated for each method of the source class.

```
1   class ClassNameImpl implements ClassName
2   {
3     transient IClassName realObject=this;
4     IClassName remoteObject;
5     public String getName (){
6         while(true){
7           try{
9             return realObject.getName();
10          }
11          catch (Exception e){
12            System.err.println(e.toString());
13          }
14        }
15    }
16  public String getNameImpl ()
17   {
18     return name;
19   }
    }
```

Snippet 62: Dual-purpose class

The readObject() method also exists in the class and reconnects to the remote object after serialization of the object.

If object migration is added in the future, the serialization method writeObject() should be implemented so that the variables of the class are not transient when the object is being serialized.

4.7.1.3 ReadObject() method

When a reference of the object is sent to another node, in fact it is the proxy that is being serialized and sent to the remote object. The readObject() method runs after each deserialization. This method is overridden to allow initialization of the connection between the proxy object and the remote object. In RMI, the proxy object includes the RMI proxy (stub) and the stub can be serialized and sent. Therefore, the readObject() method only includes setting realObject to refer to remoteObject, which a reference to the RMI stub (Snippet 63).

```
1  private void readObject(java.io.ObjectInputStream in) throws Exception
2  {
3    try
4    {
5      in.defaultReadObject();
6    }
7    catch(Exception e)
8    {
9      throw e;
10   }
11   realObject=remoteObject;
12 }
```

Snippet 63: readObject() method with RMI

4.7.1.4 Constructors

The dual-purpose class has two constructors for each constructor of the source class. One constructor has the same signature as Umple's default generated constructor (delegating) and the other has an extra parameter indicating the target runtime component (actual constructor). The actual constructor implements the body of the constructor of the source class.

The actual constructor must have an extra runtime component argument so that the developer can choose the runtime component of the object. This constructor should check if the constructor is of the actual class or of the super class. If it is the constructor of the actual class, it calls the 'new' method of the factory with all the arguments of the constructor plus the runtime component and the reference to the object itself ("this" keyword). The new method of the factory creates the class on the target node and sets the realObject and other variables necessary for communications. It also creates a remote object on the target node. Figure 23 shows the process of creation of objects for a distributable object.

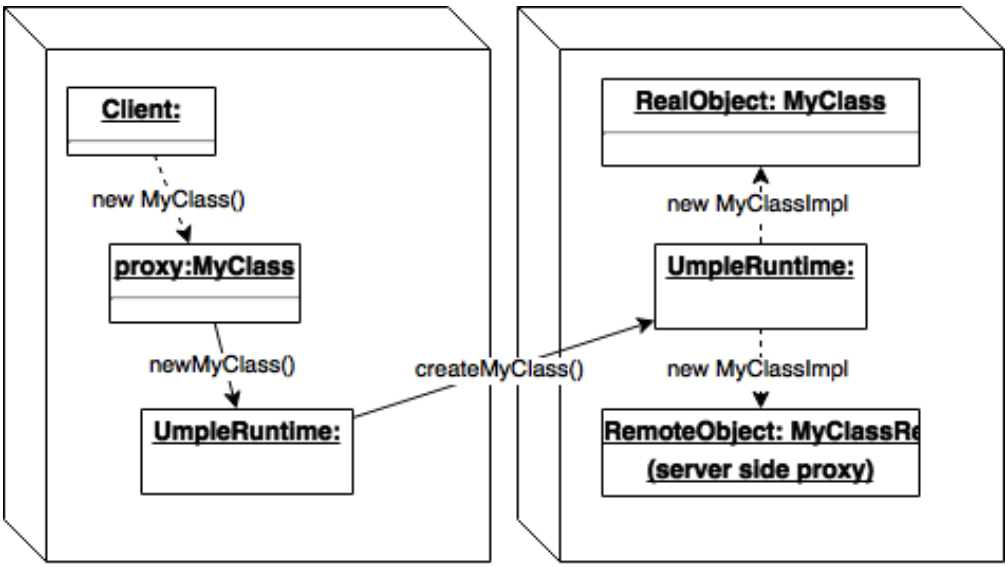

Figure 23: The creation of a remote object by factories in double proxy approach

After calling the ‘new’ method of the factory, it checks if the runtime component is local or not, if it is not local, it executes the remaining code of the constructor.

The actual constructor of the class should also check whether the object is created on the target node or not. If the object is not created on its target node, it means the object is the proxy object and it should not execute the body of the constructor. Instead, it must call the factory to get a reference of the remote object which is located on the target node. The ‘create’ method of UmpleRuntime calls the constructor of the object with “local” runtime component so that the objects created by the factory are always local objects on the target node. It also allows the user to create a distributable object locally using the “local” keyword. Snippet 64 shows the code added to the beginning of the constructor of the constructor.

```
1   public Vendor(String aName, UmpleRuntime.UmpleComponent umpleComponent)
2   {
3     super(aName, umpleComponent);
4     if(umpleComponent.getNode().getId() != UmpleRuntime.getThisNodeId())
5     {
6       if(this.getClass() == Vendor.class)
7         UmpleRuntime.getInstance().newVendor(aName, umpleComponent, this);
8       return;
9     }
10  // the remaining code of the constructor
11
```

Snippet 64: An example of the beginning part of the actual constructor of the dual-purpose class

The delegating constructor is the default constructor of the class. It does not have the runtime component argument and calls the other constructor with an extra argument which is the runtime component. This constructor is generated to group all of the objects of a class in a single runtime component by default. It is done by calling Java's this() method with all of the arguments plus the runtime component which has the name of the class. The this() method calls the constructor of the class within the class. Snippet 65 shows an example of the body of a delegating constructor.

```
1   Public Vendor(){
2     this(UmpleRuntime.getComponent("vendor"))
3   }
```

Snippet 65: Example of a delegating constructor

4.7.2 Communication Interface

The communication interface has a small difference with the interfaces generated in the second approach. Since the names of the implemented methods are replaced by adding the "Impl" postfix, the name of the methods in the communication interface are also replaced to match them so that both the dual-purpose class and the remote class can implement the same interface. Therefore, the only difference between the communication interface in this approach with the communication interface in the second approach is the added "Impl" after the names of the methods.

4.7.3 Remote Class

A server-side proxy class is generated which is the remote class and implements the communication interface. All of its methods delegate the method call to the real object locally. All methods have "Impl" as a postfix to their name and call the same method name with the same parameters on the

“realObject" variable which is a reference of the real object. It is of type of the actual class and is set by the factory to the real object (Snippet 66).

```
1  public class VendorRemote extends AgentRemote implements IVendorImpl
2  {
3    Vendor realObject;
4     public VendorRemote()
5    {}
6     public int getHashCodeImpl()
7     {
8        return realObject.getHashCodeImpl();
     }
   }
```

Snippet 66: Example of remote class

4.7.4 The “new” Method of UmpleRuntime Class

The “new” factory method finds the target node of the object based on its runtime component. If the target node is the same node as the node it is running on, it only creates the remote object and initializes the variables necessary for communications. If it is not on the target node, it means the method has been called by a proxy and the real object should be created on the remote node. It calls the "create” method on the target node.

4.7.5 Using Pure TCP network communications

Using the server-side proxy implementing the system with pure TCP connection is also possible and can be future work.

To implement the system using TCP connections, the client-side proxy must connect to the server-side proxy and request a method call by sending the name of the method and the arguments by serializing their values. The server-side proxy receives the data and deserializes the method name and arguments. It then calls the actual object and returns the return value to the client-side proxy by serializing it. The client-side proxy then deserialize the return value and returns it to the caller.

4.7.6 Implementing with web services

For the proxy to call methods on the remote object using web services, Umple generates code to utilize the Javax.xml library. There are differences in the generated code for the generated interface, the remote class, the dual-purpose class, and in the ‘new’ method of UmpleRuntime class as

explained in section 4.7.6.4. To choose web services as the communication technology of the class. “WS” keyword is used after the “distributable” keyword (Snippet 67). To force all the classes to be distributable using web services the directive in Snippet 68 should be used.

```
1  class A{
2    distributable WS;
3  }
```

Snippet 67: Declaring a specific class to be distributable using Web Services

```
1  distributable WS 0 forced;
```

Snippet 68: Declaring all classes to be distributable using Web Services

Using web services, an object can be both a web service and be serialized and sent to another node. Therefore, there is no need for the remote object to be separate from the real object. Instead of creating a remote object for each object, the proxy calls the implemented methods of the real class directly. Umple makes the dual-purpose object a remote object (a web service endpoint). Figure 24 shows the class diagram for a class called “MyClass”.

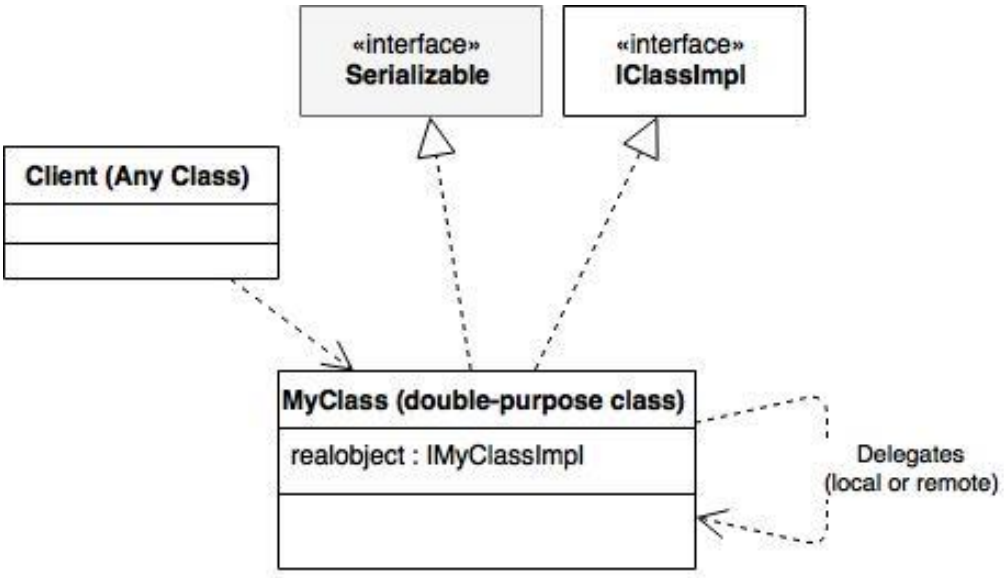

Figure 24: Class diagram of generated system for distributable class with web services

The differences in the generated code for web services is described in the following subsections.

4.7.6.1 Interface

The communication interface is changed to support RPC style Soap binding. All methods are annotated with @WebMethod to be accessible for remote call. Snippet 46 shows an example of the generated interface for a class with web services. The annotations needed for web services are generated as explained in 4.4.

4.7.6.2 Dual-purpose Class

When the communication technology is web services, there are annotations that must be added before the definition of the class. The ‘objectId’(string), ‘remoteUrl’ (string), and ‘remotePort’ (integer) variables are needed to connect to the remote object using web services. An example is shown in Snippet 69 .

```
1   import javax.jws.WebService;
2   import javax.xml.ws.Endpoint;
3   import java.io.Serializable;
4
5   @WebService(endpointInterface = “PackageName.IClassNameImpl”)
6   class ClassName implements java.io.Serializable, IClassNameImpl
7   {
9     String objectId;
10    String remoteURL;
11    String remotePort;
12    IClassName realObject;
13    public String getName (){
14       while(true){
15          try{
16             return realObject.getName();
17          }
18          catch (Exception e){
19             System.err.println(e.toString());
20             Thread.sleep(1000);
21          }
22       }
23    }
24  public String getNameImpl ()
25   {
26      return name;
27   }
28  }
```

Snippet 70: The implementation class generated for a class named "ClassName" in second pattern using web services

The proxy should reconnect to the remote object after moving to another node using the readObject() method or the afterUnmarshal() method. The implementation is described in Section 4.7.6.3.

4.7.6.3 The readObject(), afterUnmarshal, and initializeConnection methods

If the object is sent using RMI, the readObject() method is called after deserialization. If the object is sent using web services, the afterUnmarshal() method is called. These methods are responsible of initializing the connections to the server. They call initializeConnection() method which is generated by Umple.

For web services, the proxy of the web service (realObject) cannot be serialized. Therefore, it is necessary to connect the proxy to the remote object after each deserialization of the proxy. Deserialization of the proxy only happens if an object sends a reference of the remote object to another node.

The proxy holds the URL and port and hash code of the real object as variables and uses them to connect to the server. The proxy connects to the server and creates a web service proxy and sets the real object again. Snippet 71 shows an example of initialization method for a class.

```
1   void afterUnmarshal(Unmarshaller u, Object parent)
2   {
3     initializeConnection();
4   }
5   private void initializeConnection()
6   {
7     if(objectId.contains(Vendor.class.getName()))
8     {
9       boolean success = false;
10      while (!success) {
11        try
12        {
13          URL url = new URL(remoteUrl+":"+remotePort+"/Vendor"+objectId+"?wsdl");
14          QName qname = new QName("http://ecommerceWS/","VendorService");
15          Service service = Service.create(url, qname);
16          setRealObject(service.getPort(IVendorImpl.class));
17          success = true;
18        }
191       catch (Exception e)
20        {
21          System.err.println("Client exception: " + e.toString());
22          e.printStackTrace();
23          try { Thread.sleep(1000); } catch (InterruptedException interruptedException) {};
```

```
24          }
25        }
26      }
27    }
```

Snippet 71: An example of the initializeConnection() method with web services

4.7.6.4 'new' method of UmpleRuntime

The 'new' method calls the create method on the target node and returns the proxy. If the object is on its target node, it sets the objectId, remoteUrl, and remotePort of the object. objectId is the initial hashCode of the object, as shown in Snippet 72.

```
1  object.remoteUrl=nodes.get(getThisNodeId()).getUrl();
2  object.remotePort=String.valueOf(nodes.get(getThisNodeId()).getPort());
3  object.objectId=object.getHashCodeImpl();
4  Endpoint.publish(object.remoteUrl+":"+object.remotePort+"/Warehouse"+String.valueOf(object.objectId),object
   );
```

Snippet 72: Publishing and setting the variables of the server object in 'new' method

4.7.6.5 Issues

There are some issues with web services that should be fixed in the future.

- In the current implementation, since JAX-WS does not support overridden methods, it causes warning when there are two methods with the same name (and different parameters) in the class. Therefore, it cannot transform all systems to distributed systems that use web services.
- Web services do not support polymorphism. When sending an object to another node, such a service only looks at the type of the object in the method signature and does not consider the object being of type of a subclass. Therefore, generating get and set methods for the superclass is not enough and Umple must override association API methods for all of the classes. Currently, inheritance of association causes this problem in Umple.
- JavaX does not support interfaces as return value of the methods. Umple generates getAll() methods in the API for associations that have 'many' as multiplicity. Using associations with * as multiplicity and using the getAll() method currently causes an error. This can be

fixed by changing the Java generation in this case to use a concrete class instead of an interface.

- The JAXB unmarshaller creates the copied object by calling the no-argument constructor. Therefore, all non-distributed classes which have instances that are being sent to other nodes must have no-argument constructor. This can be solved if Umple generates no-argument constructors for all non-distributable classes that are associated with the distributable class.
- The return value of the remote methods cannot be null. This is because JAXB does not support marshalling and unmarshalling of null as return value.
- Umple allows some classes to be distributable with web service in an RMI distributable system. These distributable classes that are implemented with web service cannot serialize the RMI stub. Therefore, the distributable web service classes cannot have association with RMI classes.

5 Case Studies

5.1 Ecommerce system

5.1.1 Description of the system

As a case study, we developed a simple distributed system using Umple. This system consists of products, suppliers, vendors, and customers. There can be different products with different types and serial numbers. Suppliers create these products and put them in different warehouses. Vendors are responsible for selling the products that are in the warehouses. Customers communicate with vendors (and not suppliers) to find the products they want and the vendors sell the products to the customers.

The case study consists of multiple objects with different functionality and behavior. These objects can be placed on one node or multiple nodes without effect on the overall behavior of the system. This shows how Umple distributes a non-distributed system by simply adding extra "distributable" annotations. Umple's object placement mechanism can be tested by applying it on the objects of the system in different ways.

We also run some customer objects on active nodes to show how Umple can generate a distributed parallel system with some nodes starting independently.

We limit the scope of the system to creation of a product when suppliers add products in the warehouses and before the order is confirmed and paid for by the customer. For simplicity, we assume the products are sold one by one, there is no cart to add different product in and orders consist of only one product.

First, we design and develop this system without considering it being distributed. The class diagram of the system is shown in Figure 25. We describe each class and its associations in detail using Umple code. We implement the body of the methods using Java.

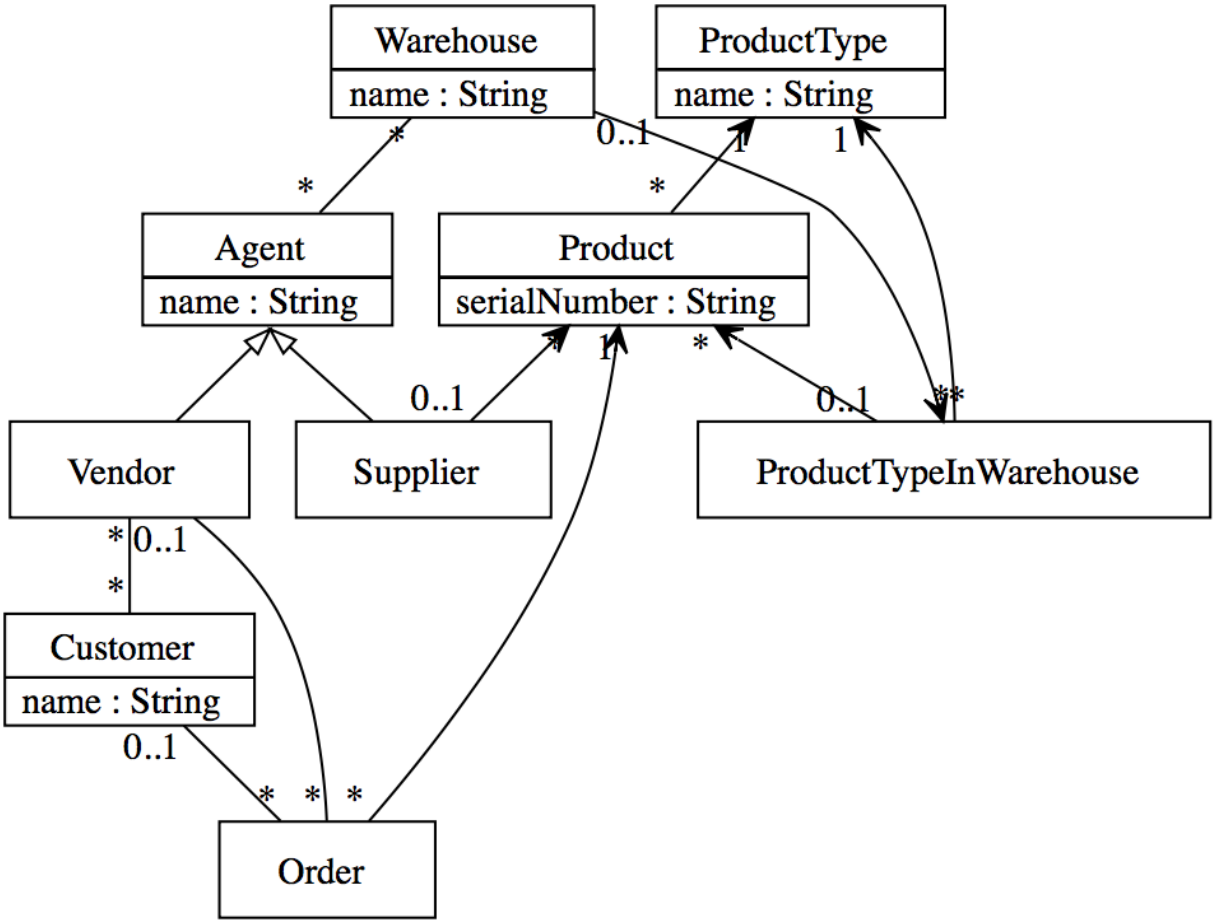

Figure 25: Class diagram of the E-Commerce case study

ProductType is an immutable class with only one attribute name, which is the product type. *Product* has a *ProductType* and a serial number and they must not change. Therefore, we define the *Product* class as immutable to enforce that the instances do not change after creation. Each Warehouse can have several *Products* inside. However, for easier and faster searching of products of a certain type, *warehouses* have *ProductTypeInWarehouse*s which are objects that contain the products of the same type. Therefore, there is an aggregation relationship between *Warehouse class and ProductTypeInWarehouse*s class and between *ProductTypeInWarehouses* class and *Product* class. In Umple, we show aggregation with an association. Warehouse has methods called *findProduct()* which finds a product by its serial number and *addProduct() method* which adds a product to the *ProductType*. The Umple code for *Warehouse* and *Product* class is shown in Snippet 73. The Umple code for Product, ProductType, and ProductTypeInWarehouse is shown in Snippet 74.

```
1  class Warehouse
2  {
3   name;
4   public Product findProduct(ProductType productType)
5   {
6    for(ProductTypeInWarehouse ptiw:getProductTypeInWarehouses())
7    {
8     if(ptiw.getProductType().equals(productType))
9     {
10     for(Product p : ptiw.getProducts())
11      return p;
12    }
13   }
14   return null;
15  }
16  public void addProduct(Product p)
17  {
18   for(ProductTypeInWarehouse ptiw : getProductTypeInWarehouses())
19   {
20    if(ptiw.getProductType().equals(p.getProductType()))
21    {
22     ptiw.addProduct(p);
23     return;
24    }
25   }
26   ProductTypeInWarehouse ptiw = new ProductTypeInWarehouse(p.getProductType());
27   ptiw.addProduct(p);
28   addProductTypeInWarehous(ptiw);
29  }
30 }
31
```

Snippet 73: Umple code of classes: Warehouse and Product

```
1  class Product
2  {
3    immutable;
4    serialNumber;
5    * -> 1 ProductType;
6  }
7  class ProductType
8  {
9    immutable;
10   name;
11   key {name};
12 }
13 class ProductTypeInWarehouse
14 {
15   * -> 1 ProductType;
16   * <- 0..1 Warehouse;
17   0..1 -> * Product;
18
19 }
```

Snippet 74: Umple code for classes: Product, ProductType, and ProductTypeInWarehouse

Supplier and Vendor classes are subclasses of the Agent class. Each Agent uses multiple Warehouses. Therefore, in the UML class diagram of the system, there is an association between Agent and Warehouse (Snippet 75).

```
1  class Agent
2  {
3    name;
4    * -- * Warehouse;
5  }
```

Snippet 75: Umple Code of Agent

Supplier can have *Products* (association with Product class) and can create and put *products* into the *warehouses* (*createProduct()* and *putInWarehouse()* methods). The *Supplier* class is shown in Snippet 76

```
1  class Supplier
2  {
3    isA Agent;
4    0..1 -> * Product;
5    public Product createProduct(String serialNumber, String productType)
6    {
7      Product aProduct=new Product(serialNumber,new ProductType(productType));
8      addProduct(aProduct);
9      return aProduct;
```

```
10    }
11    public void putInWarehouse(Product aProduct,Warehouse warehouse)
12    {
13      warehouse.addProduct(aProduct);
14    }
15  }
```

Snippet 76: Umple code for class: Supplier

Vendor has a many-to-many association with *Customer* and has a *findProduct()* method to return the product based on the serial number as well as a method called *makeOrder()* which creates an order and returns a reference. The Umple code for Vendor class is shown in Snippet 77.

```
1   class Vendor
2   {
3    isA Agent;
4    public Product findProduct(ProductType productType)
5    {
6     for(Warehouse w:getWarehouses())
7     {
8      Product p= w.findProduct(productType);
9      if(p!=null)
10       return p;
11    }
12    return null;
13   }
14   public Order makeOrder(Customer aCustomer, Product aProduct)
15   {  if(aProduct==null)
16      return null;
17    Order aOrder= new Order(aProduct);
18    aOrder.setCustomer(aCustomer);
19    aOrder.setVendor(this);
20    return aOrder;
21   }
22  }
```

Snippet 77: Umple code of class: Vendor

Each order has an Id and a product. It can also include the vendor and customers reference (for simplicity, creating order is the final step and each order includes only one product). Therefore, there is an optional association with Customer class as well as an optional association with Vendor. Snippet 78 shows the Umple code for *Order* class.

```
1  class Order
2  {
3     * -- 0..1 Customer;
4     * -- 0..1 Vendor;
5     * -> 1 Product;
6  }
```

Snippet 78: Umple code for class: Order

The customers buy products only from vendors and the Customer class has an orderProduct() method that buys a product by its serial number from a vendor by its name. There can be other methods to find a product by its type, etc. However, for the purpose of our study of distributed system, one or two methods for the class would suffice.

```
1  class Customer
2  {
3    name;
4    * -- * Vendor;
5    public Order orderProduct(String productType,String vendorName)
6    {
7      for(Vendor v: getVendors())
8      {
9        if(v.getName().equals(vendorName))
10         return v.makeOrder(this,v.findProduct(new ProductType(productType)));
11     }
12     return null;
13   }
14 }
```

Snippet 79: Umple code for class: Customer

5.1.2 **Test cases**

A simple scenario is suppliers adding some products to some warehouses, and then some customers buy (make order) some of those products from some vendors. A simple main method of the system is shown in Snippet 80.

If there are no distributable classes, Umple will generate a local system. In that case, the runtime component should not be given in the *new* statements (e.g. line 3 of Snippet 80). The system will be generated and run normally (without distributed system features). We tested and ran the system as a non-distributed program.

```
1  public static void main (String[] args){
2    Supplier supplier1= new Supplier("supplier1");
3    Supplier supplier2= new Supplier("supplier2");
4    Warehouse warehouse1=new Warehouse("warehouse1");
5    Warehouse warehouse2=new Warehouse("warehouse2");
6    Vendor vendor1= new Vendor("vendor1");
7    Vendor vendor2= new Vendor("vendor2");
8    Customer customer1= new Customer("Customer");
9    Customer customer2= new Customer("Customer");
10   customer1.addVendor(vendor1);
11   customer1.addVendor(vendor2);
12   customer2.addVendor(vendor2);
13   vendor1.addWarehous(warehouse1);
14   vendor2.addWarehous(warehouse1);
15   vendor2.addWarehous(warehouse2);
16   supplier2.putInWarehouse(supplier2.createProduct("0013M2X24","freezer"),warehouse1);
17   supplier2.putInWarehouse(supplier2.createProduct("0013M2X25","freezer"),warehouse2);
18   supplier1.putInWarehouse(supplier1.createProduct("023MM25","microwave"),warehouse2);
19   supplier2.putInWarehouse(supplier2.createProduct("002333","stove"),warehouse1);
20   supplier2.putInWarehouse(supplier2.createProduct("1013M2X25","stove"),warehouse2);
21   supplier1.putInWarehouse(supplier1.createProduct("ddR53434","stove"),warehouse2);
22   supplier2.putInWarehouse(supplier2.createProduct("00232313M2X24","laptop"),warehouse2);
23   supplier2.putInWarehouse(supplier2.createProduct("0013M5672X25","laptop"),warehouse1);
24   supplier1.putInWarehouse(supplier1.createProduct("023MM44424","laptop"),warehouse1);
25   customer1.orderProduct("freezer","vendor1");
26   customer2.orderProduct("freezer","vendor2");
27   customer1.orderProduct("stove","vendor1");
28   customer2.orderProduct("laptop","vendor2");
29   customer1.orderProduct("freezer","vendor1");
30   customer2.orderProduct("freezer","vendor2");
31 }
```

Snippet 80: Main method of the case study

To run the system as a distributed system, each Agent, Customer or Warehouse can reside on any node. Therefore, we make *Agent* (consequently *Vendor* and *Supplier*), *Customer* and *Warehouse* classes distributable (Snippet 81). We could also make all other classes distributable but this will result in more generated code, more communications, and negative effects on performance. The runtime components of the instances of distributable classes can be given as an argument in the *new statement* in Java.

```
1  class Warehouse
2  {
3    distributable;
4  }
5  class Agent
6  {
```

```
7   distributable;
8   }
9   class Customer
10  {
11    distributable;
12  }
```

Snippet 81: Using mixins to make Warehouse, Agent, and Customer classes distributable

In the generated code for distributed system, the *UmpleRuntime* Java class and *IUmpleRuntime* interface are always auto-generated. UmpleRuntime has *newWarehouse*, *newCustomer*, *newAgent*, *newVendor*, and *newSupplier* methods. These methods check whether the object's target runtime component is on the current node and create the remote object, otherwise they call the create method on the UmpleRuntime object target node. The create methods are: *createWarehouse*, *createCustomer*, *createAgent*, *createVendor*, and *createSupplier*. *Therefore, UmpleRuntime and IUmpleruntime include these methods.*

Polymorphism does not work properly in web services. For example, it cannot send *Vendor* as an *Agent*. For that reason, the association between Agent and *Warehouse* is not enough. Instead, there must be separate associations between *Warehouse* and *Vendor* and between *Warehouse* and *Supplier*. Therefore, the Agent class will only have *name* as an attribute. In the generated code for Web services, *IAgentImpl*, *ISupplierImpl*, *ICustomerImpl*, and *IWarehouseImpl* interfaces are generated as described in the previous chapter.

Using the third pattern (default in Umple) the generated code has *AgentRemote*, *SupplierRemote*, *CustomerRemote*, and *WarehouseRemote* as well as the interfaces *IAgentImpl*, *ISupplierImpl*, *ICustomerImpl*, and *IWarehouseImpl*. It also generates the *extra methods in classes Agent*, *Supplier*, *Customer*, and *Warehouse.* For each public method in these classes, there is a generated delegating method that calls the same method on the remote class. For example: the *makeOrder()* method delegates the method to the remote object but *makeOrderImpl()* is now the actual method that creates an order.

In the generated code using the second pattern, we have multiple Java classes called *Warehouse*, *Supplier*, *Agent*, and *Customer* which are all proxy classes. The methods on these classes call the actual methods on the implementation classes which are *SupplierImpl*, *AgentImpl*, and *CustomerImpl*. The implementation classes implement interfaces which are automatically generated based on the public methods of the classes.

As mentioned in Section 4.6, using “this” keyword will cause problem with the second distribution approach. Therefore, we rename them to “self” in the Umple code.

5.1.3 **Tracing**

To trace the system, we use the trace feature of Umple but modify it to support distributed features. For a local system, it generates the trace on one screen. However, when the system is distributed, there is a tracer on each node and the tracing lines appear on different screens based on their node of occurrence. To be able to verify the order of the traces, we create a singleton distributable object called *SystemTracer (code in Appendix* Snippet 94*)*. This singleton object resides on a certain node and is accessible from the whole system. We modify Umple’s tracer to send the trace messages to that object instead of printing it on the screen. This way, all trace messages will be sent to a single node and are printed on one screen to compare different approaches with non-distributed system. Because the system is sequential and the caller is blocked until the trace is printed on the screen and the function returns, we can expect the trace sequence to be the same with the non-distributed run of the program.

5.1.4 **Results**

We tested the system using different number of nodes and machines. We expected the trace of the system to be the same regardless of the number of the nodes and machines.

In Snippet 82 we can see the output of the tracing with main method shown in Snippet 80 which is a non-distributed program.

```
Time,Thread,UmpleFile,LineNumber,Class,Object,Operation,Name,Value
1506211644442,1,ecommerceTracer.ump,11,Supplier,312714112,me_e,createProduct
1506211644445,1,ecommerceTracer.ump,11,Supplier,312714112,me_e,putInWarehouse
1506211644445,1,ecommerceTracer.ump,3,Warehouse,692404036,me_e,addProduct
1506211644446,1,ecommerceTracer.ump,11,Supplier,1554874502,me_e,createProduct
1506211644446,1,ecommerceTracer.ump,11,Supplier,1554874502,me_e,putInWarehouse
1506211644446,1,ecommerceTracer.ump,3,Warehouse,1846274136,me_e,addProduct
1506211644447,1,ecommerceTracer.ump,11,Supplier,1554874502,me_e,createProduct
1506211644447,1,ecommerceTracer.ump,11,Supplier,1554874502,me_e,putInWarehouse
1506211644447,1,ecommerceTracer.ump,3,Warehouse,692404036,me_e,addProduct
1506211644447,1,ecommerceTracer.ump,11,Supplier,312714112,me_e,createProduct
1506211644447,1,ecommerceTracer.ump,11,Supplier,312714112,me_e,putInWarehouse
1506211644447,1,ecommerceTracer.ump,3,Warehouse,692404036,me_e,addProduct
1506211644447,1,ecommerceTracer.ump,11,Supplier,1554874502,me_e,createProduct
1506211644447,1,ecommerceTracer.ump,11,Supplier,1554874502,me_e,putInWarehouse
1506211644448,1,ecommerceTracer.ump,3,Warehouse,1846274136,me_e,addProduct
1506211644448,1,ecommerceTracer.ump,11,Supplier,1554874502,me_e,createProduct
1506211644448,1,ecommerceTracer.ump,11,Supplier,1554874502,me_e,putInWarehouse
1506211644448,1,ecommerceTracer.ump,3,Warehouse,692404036,me_e,addProduct
1506211644450,1,ecommerceTracer.ump,11,Supplier,312714112,me_e,createProduct
1506211644450,1,ecommerceTracer.ump,11,Supplier,312714112,me_e,putInWarehouse
```

```
1506211644451,1,ecommerceTracer.ump,3,Warehouse,692404036,me_e,addProduct
1506211644451,1,ecommerceTracer.ump,11,Supplier,1554874502,me_e,createProduct
1506211644451,1,ecommerceTracer.ump,11,Supplier,1554874502,me_e,putInWarehouse
1506211644452,1,ecommerceTracer.ump,3,Warehouse,692404036,me_e,addProduct
1506211644452,1,ecommerceTracer.ump,11,Supplier,1554874502,me_e,createProduct
1506211644452,1,ecommerceTracer.ump,11,Supplier,1554874502,me_e,putInWarehouse
1506211644452,1,ecommerceTracer.ump,3,Warehouse,1846274136,me_e,addProduct
1506211644452,1,ecommerceTracer.ump,11,Supplier,312714112,me_e,createProduct
1506211644452,1,ecommerceTracer.ump,11,Supplier,312714112,me_e,putInWarehouse
1506211644452,1,ecommerceTracer.ump,3,Warehouse,1846274136,me_e,addProduct
1506211644452,1,ecommerceTracer.ump,20,Customer,1639705018,me_e,orderProduct
1506211644452,1,ecommerceTracer.ump,15,Vendor,1627674070,me_e,findProduct
1506211644453,1,ecommerceTracer.ump,3,Warehouse,1846274136,me_e,findProduct
1506211644453,1,ecommerceTracer.ump,15,Vendor,1627674070,me_e,makeOrder
1506211644454,1,ecommerceTracer.ump,20,Customer,1360875712,me_e,orderProduct
1506211644455,1,ecommerceTracer.ump,15,Vendor,1625635731,me_e,findProduct
1506211644455,1,ecommerceTracer.ump,3,Warehouse,1846274136,me_e,findProduct
1506211644455,1,ecommerceTracer.ump,15,Vendor,1625635731,me_e,makeOrder
1506211644455,1,ecommerceTracer.ump,20,Customer,1639705018,me_e,orderProduct
1506211644456,1,ecommerceTracer.ump,15,Vendor,1627674070,me_e,findProduct
1506211644456,1,ecommerceTracer.ump,3,Warehouse,1846274136,me_e,findProduct
1506211644456,1,ecommerceTracer.ump,15,Vendor,1627674070,me_e,makeOrder
1506211644456,1,ecommerceTracer.ump,20,Customer,1360875712,me_e,orderProduct
1506211644456,1,ecommerceTracer.ump,15,Vendor,1625635731,me_e,findProduct
1506211644456,1,ecommerceTracer.ump,3,Warehouse,1846274136,me_e,findProduct
1506211644456,1,ecommerceTracer.ump,15,Vendor,1625635731,me_e,makeOrder
1506211644456,1,ecommerceTracer.ump,20,Customer,1639705018,me_e,orderProduct
1506211644456,1,ecommerceTracer.ump,15,Vendor,1627674070,me_e,findProduct
1506211644456,1,ecommerceTracer.ump,3,Warehouse,1846274136,me_e,findProduct
1506211644457,1,ecommerceTracer.ump,15,Vendor,1627674070,me_e,makeOrder
1506211644457,1,ecommerceTracer.ump,20,Customer,1360875712,me_e,orderProduct
1506211644457,1,ecommerceTracer.ump,15,Vendor,1625635731,me_e,findProduct
1506211644457,1,ecommerceTracer.ump,3,Warehouse,1846274136,me_e,findProduct
1506211644457,1,ecommerceTracer.ump,15,Vendor,1625635731,me_e,makeOrder
```

Snippet 82: Output of tracing in non-distributed case

We then distribute the system using the provided keyword and change the main method. The Main method used in this experiment is shown in Snippet 83. The changes made to address the objects to runtime components are highlighted. Note that we put each supplier, warehouse, vendor, and customer on separate runtime component.

```
1  public static void main (String[] args){
2    Supplier supplier1= new Supplier("supplier1",UmpleRuntime.getComponent("component1"));
3    Supplier supplier2= new      Supplier("supplier2",UmpleRuntime.getComponent("component2"));
4    Warehouse warehouse1=new   Warehouse("warehouse1",UmpleRuntime.getComponent("component2"));
5    Warehouse warehouse2=new Warehouse("warehouse2",UmpleRuntime.getComponent("component3"));
6    Vendor vendor1= new Vendor("vendor1",UmpleRuntime.getComponent("component3"));
7    Vendor vendor2= new Vendor("vendor2",UmpleRuntime.getComponent("component4"));
8    Customer customer1= new Customer("Customer",UmpleRuntime.getComponent("component5"));
9    Customer customer2= new Customer("Customer",UmpleRuntime.getComponent("component6"));
10   customer1.addVendor(vendor1);
11   customer1.addVendor(vendor2);
12   customer2.addVendor(vendor2);
13   vendor1.addWarehous(warehouse1);
```

```
14    vendor2.addWarehous(warehouse1);
15    vendor2.addWarehous(warehouse2);
16    supplier2.putInWarehouse(supplier2.createProduct("0013M2X24","freezer"),warehouse1);
17    supplier2.putInWarehouse(supplier2.createProduct("0013M2X25","freezer"),warehouse2);
18    supplier1.putInWarehouse(supplier1.createProduct("023MM25","microwave"),warehouse2);
19    supplier2.putInWarehouse(supplier2.createProduct("002333","stove"),warehouse1);
20    supplier2.putInWarehouse(supplier2.createProduct("1013M2X25","stove"),warehouse2);
21    supplier1.putInWarehouse(supplier1.createProduct("ddR53434","stove"),warehouse2);
22    supplier2.putInWarehouse(supplier2.createProduct("00232313M2X24","laptop"),warehouse2);
23    supplier2.putInWarehouse(supplier2.createProduct("0013M5672X25","laptop"),warehouse1);
24    supplier1.putInWarehouse(supplier1.createProduct("023MM44424","laptop"),warehouse1);
25    customer1.orderProduct("freezer","vendor1");
26    customer2.orderProduct("freezer","vendor2");
27    customer1.orderProduct("stove","vendor1");
28    customer2.orderProduct("laptop","vendor2");
29    customer1.orderProduct("freezer","vendor1");
30    customer2.orderProduct("freezer","vendor2");
31    customer1.orderProduct("stove","vendor1");
32    customer2.orderProduct("laptop","vendor2");
33  }
```

Snippet 83: The main method example for tracing (changes after distribution are highlighted)

Next, we trace the distributed version of the system, but run the system only on one node. The trace is shown in Snippet 84. Note the remote object creation trace of UmpleRuntime in the beginning. Except for object creation, the trace of Snippet 82 and Snippet 84 are the same.

```
Time,Thread,UmpleFile,LineNumber,Class,Object,Operation,Name,Value
1506434650605,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newAgent
1506434650624,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newAgent
1506434650624,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newWarehouse
1506434650635,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newWarehouse
1506434650636,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newAgent
1506434650650,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newAgent
1506434650651,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newCustomer
1506434650661,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newCustomer
1506434650662,1,ecommerceTracer.ump,11,Supplier,83954662,me_e,createProduct
1506434650662,1,ecommerceTracer.ump,11,Supplier,83954662,me_e,putInWarehouse
1506434650662,1,ecommerceTracer.ump,3,Warehouse,772777427,me_e,addProduct
1506434650662,1,ecommerceTracer.ump,11,Supplier,1751075886,me_e,createProduct
1506434650663,1,ecommerceTracer.ump,11,Supplier,1751075886,me_e,putInWarehouse
1506434650663,1,ecommerceTracer.ump,3,Warehouse,401625763,me_e,addProduct
1506434650663,1,ecommerceTracer.ump,11,Supplier,1751075886,me_e,createProduct
1506434650663,1,ecommerceTracer.ump,11,Supplier,1751075886,me_e,putInWarehouse
1506434650663,1,ecommerceTracer.ump,3,Warehouse,772777427,me_e,addProduct
1506434650663,1,ecommerceTracer.ump,11,Supplier,83954662,me_e,createProduct
1506434650663,1,ecommerceTracer.ump,11,Supplier,83954662,me_e,putInWarehouse
1506434650663,1,ecommerceTracer.ump,3,Warehouse,772777427,me_e,addProduct
1506434650663,1,ecommerceTracer.ump,11,Supplier,1751075886,me_e,createProduct
1506434650663,1,ecommerceTracer.ump,11,Supplier,1751075886,me_e,putInWarehouse
1506434650663,1,ecommerceTracer.ump,3,Warehouse,401625763,me_e,addProduct
1506434650663,1,ecommerceTracer.ump,11,Supplier,1751075886,me_e,createProduct
1506434650663,1,ecommerceTracer.ump,11,Supplier,1751075886,me_e,putInWarehouse
1506434650663,1,ecommerceTracer.ump,3,Warehouse,772777427,me_e,addProduct
1506434650663,1,ecommerceTracer.ump,11,Supplier,83954662,me_e,createProduct
1506434650663,1,ecommerceTracer.ump,11,Supplier,83954662,me_e,putInWarehouse
1506434650664,1,ecommerceTracer.ump,3,Warehouse,772777427,me_e,addProduct
```

```
1506434650664,1,ecommerceTracer.ump,11,Supplier,1751075886,me_e,createProduct
1506434650664,1,ecommerceTracer.ump,11,Supplier,1751075886,me_e,putInWarehouse
1506434650664,1,ecommerceTracer.ump,3,Warehouse,772777427,me_e,addProduct
1506434650664,1,ecommerceTracer.ump,11,Supplier,1751075886,me_e,createProduct
1506434650664,1,ecommerceTracer.ump,11,Supplier,1751075886,me_e,putInWarehouse
1506434650664,1,ecommerceTracer.ump,3,Warehouse,401625763,me_e,addProduct
1506434650664,1,ecommerceTracer.ump,11,Supplier,83954662,me_e,createProduct
1506434650664,1,ecommerceTracer.ump,11,Supplier,83954662,me_e,putInWarehouse
1506434650664,1,ecommerceTracer.ump,3,Warehouse,401625763,me_e,addProduct
1506434650665,1,ecommerceTracer.ump,20,Customer,1368884364,me_e,orderProduct
1506434650665,1,ecommerceTracer.ump,15,Vendor,835648992,me_e,findProduct
1506434650665,1,ecommerceTracer.ump,3,Warehouse,401625763,me_e,findProduct
1506434650666,1,ecommerceTracer.ump,15,Vendor,835648992,me_e,makeOrder
1506434650666,1,ecommerceTracer.ump,20,Customer,492228202,me_e,orderProduct
1506434650666,1,ecommerceTracer.ump,15,Vendor,1134517053,me_e,findProduct
1506434650666,1,ecommerceTracer.ump,3,Warehouse,401625763,me_e,findProduct
1506434650666,1,ecommerceTracer.ump,15,Vendor,1134517053,me_e,makeOrder
1506434650667,1,ecommerceTracer.ump,20,Customer,1368884364,me_e,orderProduct
1506434650667,1,ecommerceTracer.ump,15,Vendor,835648992,me_e,findProduct
1506434650667,1,ecommerceTracer.ump,3,Warehouse,401625763,me_e,findProduct
1506434650668,1,ecommerceTracer.ump,15,Vendor,835648992,me_e,makeOrder
1506434650668,1,ecommerceTracer.ump,20,Customer,492228202,me_e,orderProduct
1506434650668,1,ecommerceTracer.ump,15,Vendor,1134517053,me_e,findProduct
1506434650668,1,ecommerceTracer.ump,3,Warehouse,401625763,me_e,findProduct
1506434650668,1,ecommerceTracer.ump,15,Vendor,1134517053,me_e,makeOrder
1506434650668,1,ecommerceTracer.ump,20,Customer,1368884364,me_e,orderProduct
1506434650668,1,ecommerceTracer.ump,15,Vendor,835648992,me_e,findProduct
1506434650668,1,ecommerceTracer.ump,3,Warehouse,401625763,me_e,findProduct
1506434650668,1,ecommerceTracer.ump,15,Vendor,835648992,me_e,makeOrder
1506434650669,1,ecommerceTracer.ump,20,Customer,492228202,me_e,orderProduct
1506434650669,1,ecommerceTracer.ump,15,Vendor,1134517053,me_e,findProduct
1506434650669,1,ecommerceTracer.ump,3,Warehouse,401625763,me_e,findProduct
1506434650669,1,ecommerceTracer.ump,15,Vendor,1134517053,me_e,makeOrder
```

Snippet 84: Trace of the distributed program but only on one node

The configuration file to distribute the runtime components on different nodes is shown in Snippet 85. Note that SystemTracer is put on node 2.

```
{id=0; port=8081; http://localhost; {component1,Manager}}
{id=1; port=8082; http://localhost; {component2,component4}}
{id=2; port=8083; http://localhost; {component3,SystemTracer}}
```

Snippet 85: Configuration file with three nodes

In Snippet 86 we can see the system trace distributed on three nodes. This is the trace of the system with each node printing its trace. Note that this time we have extra create methods of UmpleRuntime being called on the target node of the object. These methods are responsible for creating the objects and returning a proxy.

Snippet 87 shows the trace of the program using a SystemTracer. It shows the same trace as of the non-distributed program and also shows on which node the function is being executed. By comparing these traces on different programs and number of nodes, we can verify the distributed system working correctly.

Node 0	Time,Thread,UmpleFile,LineNumber,Class,Object,Operation,Name,Value 1506435097515,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newAgent 1506435097539,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newSupplier 1506435097568,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newWarehouse 1506435097607,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newWarehouse 1506435097648,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newVendor 1506435097732,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newVendor 1506435097762,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newCustomer 1506435097778,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newCustomer 1506435097968,1,ecommerceTracer.ump,11,Supplier,1830908236,me_e,createProduct 1506435097968,1,ecommerceTracer.ump,11,Supplier,1830908236,me_e,putInWarehouse 1506435097987,1,ecommerceTracer.ump,11,Supplier,1830908236,me_e,createProduct 1506435097987,1,ecommerceTracer.ump,11,Supplier,1830908236,me_e,putInWarehouse 1506435097995,1,ecommerceTracer.ump,11,Supplier,1830908236,me_e,createProduct 1506435097995,1,ecommerceTracer.ump,11,Supplier,1830908236,me_e,putInWarehouse 1506435098006,1,ecommerceTracer.ump,11,Supplier,1830908236,me_e,createProduct 1506435098006,1,ecommerceTracer.ump,11,Supplier,1830908236,me_e,putInWarehouse 1506435098009,1,ecommerceTracer.ump,20,Customer,1555093762,me_e,orderProduct 1506435098040,1,ecommerceTracer.ump,20,Customer,1768305536,me_e,orderProduct 1506435098064,1,ecommerceTracer.ump,20,Customer,1555093762,me_e,orderProduct 1506435098073,1,ecommerceTracer.ump,20,Customer,1768305536,me_e,orderProduct 1506435098088,1,ecommerceTracer.ump,20,Customer,1555093762,me_e,orderProduct 1506435098106,1,ecommerceTracer.ump,20,Customer,1768305536,me_e,orderProduct
Node 1	Time,Thread,UmpleFile,LineNumber,Class,Object,Operation,Name,Value 1506435097541,16,ecommerceRMI0,20,UmpleRuntime,1225358173,me_e,createSupplier 1506435097544,16,ecommerceRMI0,20,UmpleRuntime,1225358173,me_e,newAgent 1506435097569,16,ecommerceRMI0,20,UmpleRuntime,1225358173,me_e,createWarehouse 1506435097569,16,ecommerceRMI0,20,UmpleRuntime,1225358173,me_e,newWarehouse 1506435097733,16,ecommerceRMI0,20,UmpleRuntime,1225358173,me_e,createVendor 1506435097733,16,ecommerceRMI0,20,UmpleRuntime,1225358173,me_e,newAgent 1506435097975,16,ecommerceTracer.ump,11,Supplier,550672802,me_e,createProduct 1506435097979,16,ecommerceTracer.ump,11,Supplier,550672802,me_e,putInWarehouse 1506435097980,23,ecommerceTracer.ump,3,Warehouse,2123895056,me_e,addProduct 1506435097980,16,ecommerceTracer.ump,11,Supplier,550672802,me_e,createProduct 1506435097983,16,ecommerceTracer.ump,11,Supplier,550672802,me_e,putInWarehouse 1506435097989,16,ecommerceTracer.ump,11,Supplier,550672802,me_e,createProduct 1506435097990,16,ecommerceTracer.ump,11,Supplier,550672802,me_e,putInWarehouse 1506435097991,23,ecommerceTracer.ump,3,Warehouse,2123895056,me_e,addProduct 1506435097992,16,ecommerceTracer.ump,11,Supplier,550672802,me_e,createProduct 1506435097993,16,ecommerceTracer.ump,11,Supplier,550672802,me_e,putInWarehouse 1506435097996,16,ecommerceTracer.ump,11,Supplier,550672802,me_e,createProduct 1506435097998,16,ecommerceTracer.ump,11,Supplier,550672802,me_e,putInWarehouse 1506435098002,16,ecommerceTracer.ump,11,Supplier,550672802,me_e,createProduct 1506435098005,16,ecommerceTracer.ump,11,Supplier,550672802,me_e,putInWarehouse 1506435098006,23,ecommerceTracer.ump,3,Warehouse,2123895056,me_e,addProduct 1506435098008,16,ecommerceTracer.ump,3,Warehouse,2123895056,me_e,addProduct 1506435098021,18,ecommerceTracer.ump,3,Warehouse,2123895056,me_e,findProduct 1506435098041,16,ecommerceTracer.ump,15,Vendor,3486,me_e,findProduct 1506435098042,23,ecommerceTracer.ump,3,Warehouse,2123895056,me_e,findProduct 1506435098052,16,ecommerceTracer.ump,15,Vendor,3486,me_e,makeOrder 1506435098067,18,ecommerceTracer.ump,3,Warehouse,2123895056,me_e,findProduct 1506435098074,16,ecommerceTracer.ump,15,Vendor,3486,me_e,findProduct 1506435098075,23,ecommerceTracer.ump,3,Warehouse,2123895056,me_e,findProduct 1506435098083,16,ecommerceTracer.ump,15,Vendor,3486,me_e,makeOrder 1506435098094,18,ecommerceTracer.ump,3,Warehouse,2123895056,me_e,findProduct 1506435098111,16,ecommerceTracer.ump,15,Vendor,3486,me_e,findProduct 1506435098112,23,ecommerceTracer.ump,3,Warehouse,2123895056,me_e,findProduct 1506435098114,16,ecommerceTracer.ump,15,Vendor,3486,me_e,makeOrder
Node 2	Time,Thread,UmpleFile,LineNumber,Class,Object,Operation,Name,Value 1506435097608,19,ecommerceRMI0,20,UmpleRuntime,1225358173,me_e,createWarehouse 1506435097613,19,ecommerceRMI0,20,UmpleRuntime,1225358173,me_e,newWarehouse 1506435097649,19,ecommerceRMI0,20,UmpleRuntime,1225358173,me_e,createVendor 1506435097650,19,ecommerceRMI0,20,UmpleRuntime,1225358173,me_e,newAgent 1506435097974,19,ecommerceTracer.ump,3,Warehouse,510256313,me_e,addProduct 1506435097983,17,ecommerceTracer.ump,3,Warehouse,510256313,me_e,addProduct 1506435097988,19,ecommerceTracer.ump,3,Warehouse,510256313,me_e,addProduct 1506435097994,17,ecommerceTracer.ump,3,Warehouse,510256313,me_e,addProduct 1506435097995,19,ecommerceTracer.ump,3,Warehouse,510256313,me_e,addProduct

```
1506435098001,17,ecommerceTracer.ump,3,Warehouse,510256313,me_e,addProduct
1506435098018,19,ecommerceTracer.ump,15,Vendor,1344870762,me_e,findProduct
1506435098028,19,ecommerceTracer.ump,15,Vendor,1344870762,me_e,makeOrder
1506435098066,19,ecommerceTracer.ump,15,Vendor,1344870762,me_e,findProduct
1506435098069,19,ecommerceTracer.ump,15,Vendor,1344870762,me_e,makeOrder
1506435098094,19,ecommerceTracer.ump,15,Vendor,1344870762,me_e,findProduct
1506435098099,19,ecommerceTracer.ump,15,Vendor,1344870762,me_e,makeOrder
```

Snippet 86: Trace of the system on three different nodes

```
             on node number:0;
Time,Thread,UmpleFile,LineNumber,Class,Object,Operation,Name,Value
             on node number:0;
1506811050209,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newAgent
             on node number:0;
1506811050245,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newSupplier
             on node number:1;
1506811050246,18,ecommerceRMI0,20,UmpleRuntime,1225358173,me_e,createSupplier
             on node number:1;
1506811050270,18,ecommerceRMI0,20,UmpleRuntime,845886128,me_e,newAgent
             on node number:0;
1506811050303,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newWarehouse
             on node number:1;
1506811050304,18,ecommerceRMI0,20,UmpleRuntime,1225358173,me_e,createWarehouse
             on node number:1;
1506811050305,18,ecommerceRMI0,20,UmpleRuntime,845886128,me_e,newWarehouse
             on node number:0;
1506811050327,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newWarehouse
             on node number:2;
1506811050328,17,ecommerceRMI0,20,UmpleRuntime,1225358173,me_e,createWarehouse
             on node number:2;
1506811050330,17,ecommerceRMI0,20,UmpleRuntime,1225358173,me_e,newWarehouse
             on node number:0;
1506811050348,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newVendor
             on node number:2;
1506811050350,17,ecommerceRMI0,20,UmpleRuntime,1225358173,me_e,createVendor
             on node number:2;
1506811050351,17,ecommerceRMI0,20,UmpleRuntime,1225358173,me_e,newAgent
             on node number:0;
1506811050377,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newVendor
             on node number:1;
1506811050378,18,ecommerceRMI0,20,UmpleRuntime,1225358173,me_e,createVendor
             on node number:1;
1506811050379,18,ecommerceRMI0,20,UmpleRuntime,845886128,me_e,newAgent
             on node number:0;
1506811050397,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newCustomer
             on node number:0;
1506811050407,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newCustomer
             on node number:0;
1506811050557,1,ecommerceTracer.ump,11,Supplier,517380410,me_e,createProduct
             on node number:0;
1506811050559,1,ecommerceTracer.ump,11,Supplier,517380410,me_e,putInWarehouse
             on node number:2;
1506811050563,17,ecommerceTracer.ump,3,Warehouse,1040996481,me_e,addProduct
             on node number:1;
1506811050565,18,ecommerceTracer.ump,11,Supplier,611771109,me_e,createProduct
             on node number:1;
1506811050572,18,ecommerceTracer.ump,11,Supplier,611771109,me_e,putInWarehouse
             on node number:1;
1506811050574,23,ecommerceTracer.ump,3,Warehouse,675448648,me_e,addProduct
             on node number:1;
1506811050575,18,ecommerceTracer.ump,11,Supplier,611771109,me_e,createProduct
             on node number:1;
1506811050577,18,ecommerceTracer.ump,11,Supplier,611771109,me_e,putInWarehouse
             on node number:2;
1506811050578,19,ecommerceTracer.ump,3,Warehouse,1040996481,me_e,addProduct
```

```
            on node number:0;
1506811050579,1,ecommerceTracer.ump,11,Supplier,517380410,me_e,createProduct
            on node number:0;
1506811050579,1,ecommerceTracer.ump,11,Supplier,517380410,me_e,putInWarehouse
            on node number:2;
1506811050580,17,ecommerceTracer.ump,3,Warehouse,1040996481,me_e,addProduct
            on node number:1;
1506811050581,18,ecommerceTracer.ump,11,Supplier,611771109,me_e,createProduct
            on node number:1;
1506811050583,18,ecommerceTracer.ump,11,Supplier,611771109,me_e,putInWarehouse
            on node number:1;
1506811050585,23,ecommerceTracer.ump,3,Warehouse,675448648,me_e,addProduct
            on node number:1;
1506811050587,18,ecommerceTracer.ump,11,Supplier,611771109,me_e,createProduct
            on node number:1;
1506811050589,18,ecommerceTracer.ump,11,Supplier,611771109,me_e,putInWarehouse
            on node number:2;
1506811050590,19,ecommerceTracer.ump,3,Warehouse,1040996481,me_e,addProduct
            on node number:0;
1506811050590,1,ecommerceTracer.ump,11,Supplier,517380410,me_e,createProduct
            on node number:0;
1506811050591,1,ecommerceTracer.ump,11,Supplier,517380410,me_e,putInWarehouse
            on node number:2;
1506811050592,17,ecommerceTracer.ump,3,Warehouse,1040996481,me_e,addProduct
            on node number:1;
1506811050592,18,ecommerceTracer.ump,11,Supplier,611771109,me_e,createProduct
            on node number:1;
1506811050595,18,ecommerceTracer.ump,11,Supplier,611771109,me_e,putInWarehouse
            on node number:2;
1506811050596,19,ecommerceTracer.ump,3,Warehouse,1040996481,me_e,addProduct
            on node number:1;
1506811050598,18,ecommerceTracer.ump,11,Supplier,611771109,me_e,createProduct
            on node number:1;
1506811050601,18,ecommerceTracer.ump,11,Supplier,611771109,me_e,putInWarehouse
            on node number:1;
1506811050602,23,ecommerceTracer.ump,3,Warehouse,675448648,me_e,addProduct
            on node number:0;
1506811050603,1,ecommerceTracer.ump,11,Supplier,517380410,me_e,createProduct
            on node number:0;
1506811050604,1,ecommerceTracer.ump,11,Supplier,517380410,me_e,putInWarehouse
            on node number:1;
1506811050604,18,ecommerceTracer.ump,3,Warehouse,675448648,me_e,addProduct
            on node number:0;
1506811050605,1,ecommerceTracer.ump,20,Customer,1174361318,me_e,orderProduct
            on node number:2;
1506811050607,17,ecommerceTracer.ump,15,Vendor,1341428977,me_e,findProduct
            on node number:1;
1506811050608,16,ecommerceTracer.ump,3,Warehouse,675448648,me_e,findProduct
            on node number:2;
1506811050610,17,ecommerceTracer.ump,15,Vendor,1341428977,me_e,makeOrder
            on node number:0;
1506811050621,1,ecommerceTracer.ump,20,Customer,2007328737,me_e,orderProduct
            on node number:1;
1506811050623,18,ecommerceTracer.ump,15,Vendor,1700166302,me_e,findProduct
            on node number:1;
1506811050624,23,ecommerceTracer.ump,3,Warehouse,675448648,me_e,findProduct
            on node number:1;
1506811050626,18,ecommerceTracer.ump,15,Vendor,1700166302,me_e,makeOrder
            on node number:0;
1506811050646,1,ecommerceTracer.ump,20,Customer,1174361318,me_e,orderProduct
            on node number:2;
1506811050648,17,ecommerceTracer.ump,15,Vendor,1341428977,me_e,findProduct
            on node number:1;
1506811050648,16,ecommerceTracer.ump,3,Warehouse,675448648,me_e,findProduct
            on node number:2;
1506811050655,17,ecommerceTracer.ump,15,Vendor,1341428977,me_e,makeOrder
            on node number:0;
1506811050660,1,ecommerceTracer.ump,20,Customer,2007328737,me_e,orderProduct
```

```
            on node number:1;
1506811050661,18,ecommerceTracer.ump,15,Vendor,1700166302,me_e,findProduct
            on node number:1;
1506811050663,23,ecommerceTracer.ump,3,Warehouse,675448648,me_e,findProduct
            on node number:1;
1506811050665,18,ecommerceTracer.ump,15,Vendor,1700166302,me_e,makeOrder
            on node number:0;
1506811050670,1,ecommerceTracer.ump,20,Customer,1174361318,me_e,orderProduct
            on node number:2;
1506811050672,17,ecommerceTracer.ump,15,Vendor,1341428977,me_e,findProduct
            on node number:1;
1506811050672,16,ecommerceTracer.ump,3,Warehouse,675448648,me_e,findProduct
            on node number:2;
1506811050675,17,ecommerceTracer.ump,15,Vendor,1341428977,me_e,makeOrder
            on node number:0;
1506811050679,1,ecommerceTracer.ump,20,Customer,2007328737,me_e,orderProduct
            on node number:1;
1506811050680,18,ecommerceTracer.ump,15,Vendor,1700166302,me_e,findProduct
            on node number:1;
1506811050682,23,ecommerceTracer.ump,3,Warehouse,675448648,me_e,findProduct
            on node number:1;
1506811050689,18,ecommerceTracer.ump,15,Vendor,1700166302,me_e,makeOrder
```

Snippet 87: Combined trace of multiple nodes using SystemTracer object

5.1.5 **Performance Testing**

To test the performance of each pattern, we need the system to be able to create multiple *products* and add them to the *warehouses*. Therefore, we implemented *createBulk(int)* method for the *Supplier* class. It creates and adds certain number of product of a certain type to the *warehouse* associated with the *supplier*. To test the performance, we used two Macbook Pro Retina laptops on a network powered by a phone with a Bluetooth hotspot as router.

We tested the example program on one machine as well as on two machines. We also tested different numbers of transactions and recorded the consumed time for each case running on each machine.

The Unix “time” command gives three numbers for each process: Real time, user time, and system time. Real time is the time from the start of the program until it is finished. Real time of the system changes based on the network initialization and handshakes between nodes. It also depends on the network speed and CPU power, CPU load of each machine, and any time taken waiting for a user to respond (in this case starting a second server after starting the first one). User time is the CPU time used outside of the kernel in user-mode. System time is the CPU usage of the program inside the kernel by the process. The total of user time and system time would hence provide us with the actual CPU time used by the process. We compare the CPU time to measure penalty of network usage for each pattern and technology.

For each case, we performed the experiment three times and used the average time of the three cases as the result. In the experiments with more than one node, we used the total time of all nodes for CPU time. The nodes start and end approximately at the same time, we used the average real time of all experiments for each case.

First, we compare the programs with different number of product creation method calls (iterations). The number of iterations are 0, 800, 8000, and 80000. This is to see how the CPU time of method calls increases using our different patterns. Both cases include the initial time overhead of the remote objects exposing themselves in RMI or Web service. All cases include general Java overhead such as garbage collection and initial startup.

Table 3: CPU time of different number of transactions on one node

	Total CPU time			
Iterations	RMI (third pattern)	RMI (second pattern)	Web services	No Pattern
0	0.49	0.45	1.70	0.15
800	0.83	0.86	2.47	0.45
8000	1.25	1.23	2.86	0.84
80000	3.75	3.49	5.34	2.90

The system with 0 products can show us the overhead of starting the system and object creation. We can see that it takes much less CPU time in RMI as compared to web services to start as a server. If we subtract the overhead of initialization of the system from the results, we can compare the increase of CPU time as the result of redirections in the patterns generated by our work. Figure 26 shows how the CPU time increases in each pattern. It shows that the difference of CPU time between different patterns (and no distribution, marked as no pattern) is only modest as the number of iterations increases. While there still exists some overhead, it shows that increasing the number of method calls by 10X only increases the CPU time by 3 times or less.

We can see that the two RMI patterns are very similar for the CPU time, however the second pattern is faster when the number of method calls is increased.

Local Web services are slower than local RMI because of the longer initialization time shown in Table 3 with 0 products. This overhead is about 1.55 seconds for Web services compared to the non-distributed program. However, we can see all three distributed patterns are slightly slower than the non-distributed one due to the redirections created by the proxies.

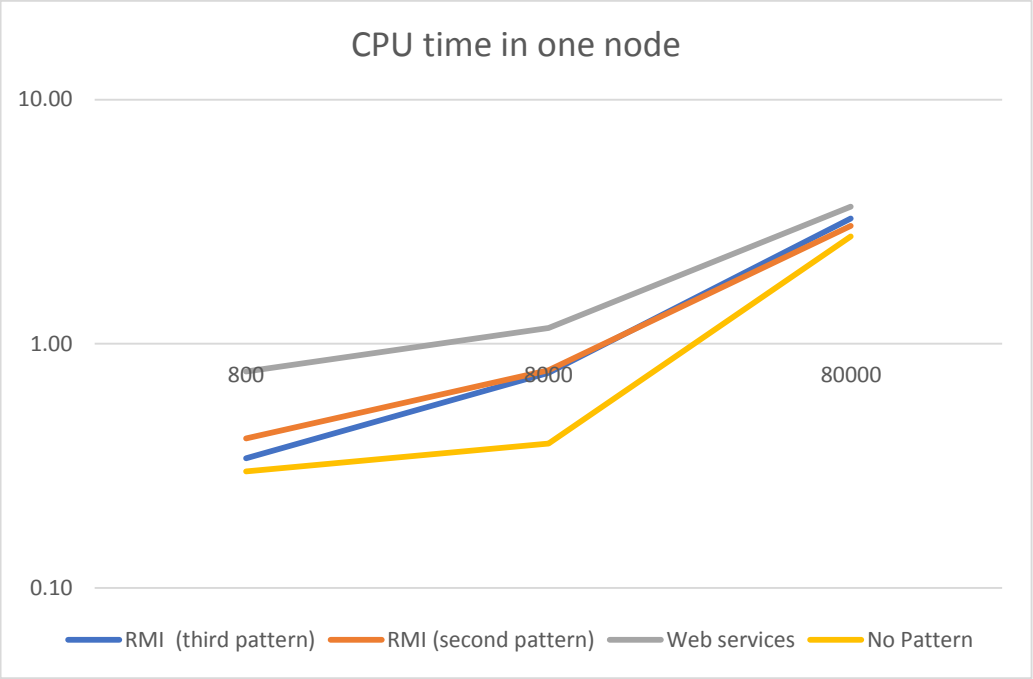

Figure 26: CPU time of one node by number of transactions (remote method calls), in seconds (logarithmic scale in both axes)

To compare the CPU time usage of communications, we ran the same process with the same number of transactions on one node, two nodes on one machine, and two nodes on two machines. The results are in Table 4 for a program with 8000 transactions between nodes. Of course, the transactions are just method calls in the single node case. Table 4 and Figure 27 show the results of this experiment.

Table 4: CPU time of different number of nodes and machines

8000 transactions	Total CPU time		
System Configuration	RMI (third pattern)	RMI (second pattern)	Web services
Single node	1.25	1.23	2.86
2 nodes on one machine	10.99	10.76	48.22
2 nodes on 2 different machines	10.25	10.62	40.16

It can be seen that the two RMI patterns are similar in the consumed time. A network call is about ten times slower than an internal method call using RMI. Web services are 16 times slower between nodes than local method calls.

The reason for better CPU times in the case with two nodes on separate machines compared to the case two nodes on one machine is probably due to the second system having better processing power and hence less CPU time needed for each task. However, real consumed time does grow due to network delay.

The data shows that running the nodes on multiple machines does not make much difference on the CPU time. However, the real time is much greater when nodes are on separate machines, presumably due to network transmission overhead (Figure 28).

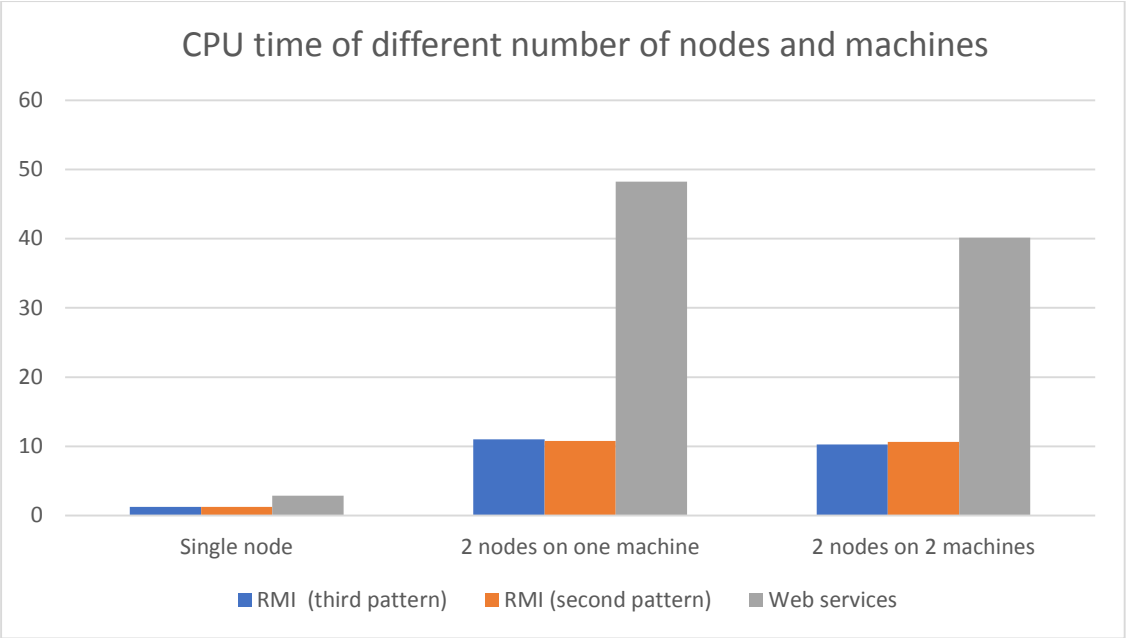

Figure 27: CPU time of different number of nodes and machines in seconds

The comparison of real consumed time is shown in Figure 28. The extra real time for one node is because distributed program with one node reads the configuration file and starts as a server. The real time increases when the system has more nodes due to RMI or WS calls. The consumed time is more when the nodes are on separate machines. This is due to network latency. We can see that RMI is faster than web services in our implementations.

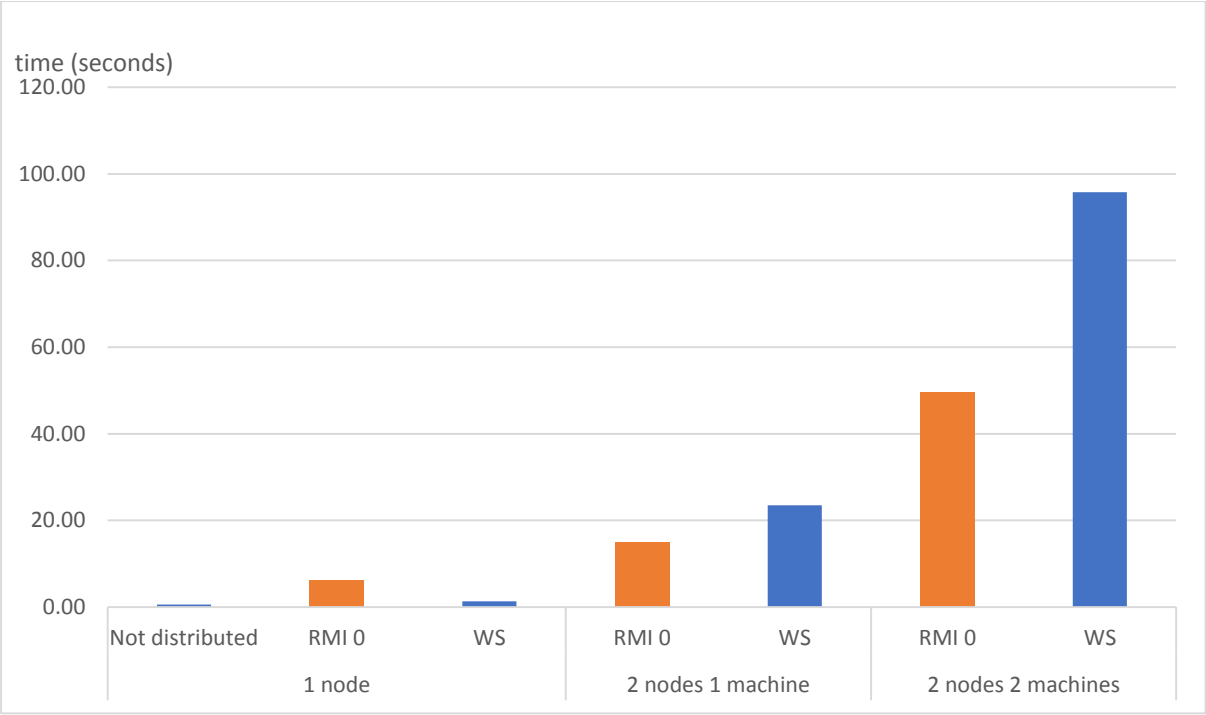

Figure 28: Real time to execute commands in different numbers of nodes and machines

Figure 29 shows the CPU time of different iterations in the case with two nodes on separate machines. It shows RMI is faster than web services. It also shows doubling the number of iterations does not increase the CPU time by as much. Even multiplying the iterations by 10 only increases the CPU time by less than 3 times.

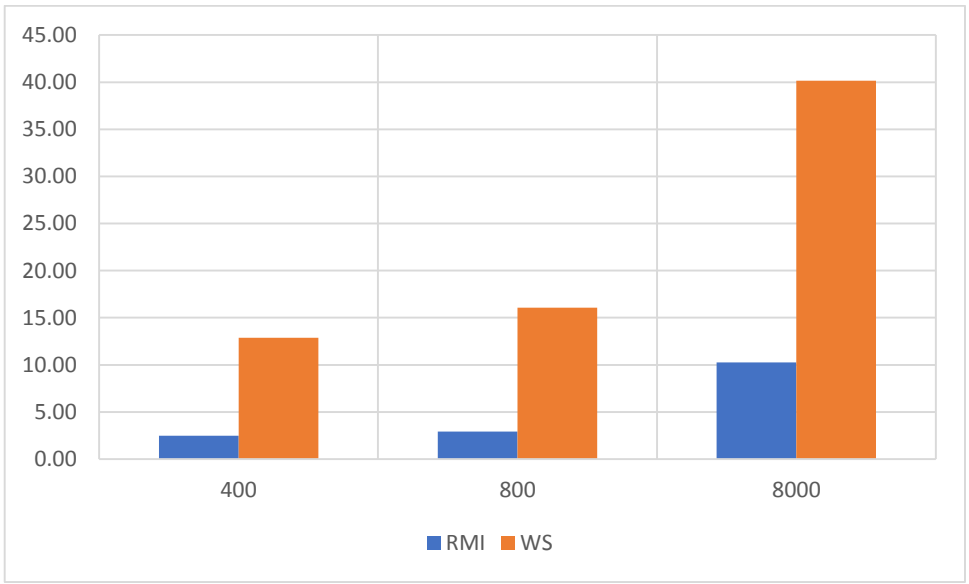

Figure 29: CPU time by number of transactions (remote method calls), for two nodes on separate machines in RMI and web services.

5.1.6 Multiple entry points and parallel nodes

In our design, it is possible to have multiple main methods so that different workflows can start at the same time or at different times. To do that, we implement another Main class and run the system with multiple active nodes.

We implemented a class called “SystemManager” as a singleton class that can have associations with other distributable classes in the system. Using distributable feature for singleton classes in Umple, every node will be able to access this object. This way nodes can connect their objects to other objects of other nodes that are created in parallel. Umple code of the SystemManager is shown in Snippet 88.

```
1   class SystemManager
2   {
3       singleton;
4       distributable;
5       0..1 -- * Agent;
6       0..1 -- * Warehouse;
7       0..1 -- * Customer;
9       0..1 -- * Order;
10  }
```

Snippet 88: Umple code of SystemManager

In this example, we move one of the customer creation lines to another node to show how one node can create a customer in parallel with the rest of the system being run. To do this, we just move line 9 of Snippet 83 to another main method of another main class called *Main2*. We replace the line with the lines shown in Snippet 89. It waits for another node to add a customer to the *systemManager* and then continues the program using the customer of the *systemManager* as ‘*customer2’* and adds a ‘*vendor2’* to it. Without the wait, the system may fail if line 13 executes before the other node creates a customer.

Lines 9 and 10 of Snippet 89 and lines 4 and 5 of Snippet 90 are to keep the time of the execution of that line and sending it to the tracer.

```
9     String checkpointTime=String.valueOf(System.currentTimeMillis());
10    SystemTracer.getInstance().handle(UmpleRuntime.getThisNodeId(),"checkpoint, "+checkpointTime);
11  while(!SystemManager.getInstance().hasCustomers())
12   {
13    try { Thread.sleep(5); } catch (InterruptedException interruptedException) {};
14   }
15   customer2= SystemManager.getInstance().getCustomer(0);
```

Snippet 89: replacing the code in main method to support parallel nodes

Main2 is responsible of creating a customer (‘*customer2’*) and associating it with the *systemManager*. The code can be seen in Snippet 90.

```
1   class Main2
2   {
3    public static void main (String[] args){
4     Customer customer2= new   Customer("Customer",UmpleRuntime.getComponent("component6"));
5     String checkpointTime=String.valueOf(System.currentTimeMillis());
6     SystemTracer.getInstance().handle(UmpleRuntime.getThisNodeId(),"checkpoint, "+checkpointTime);
7     SystemManager.getInstance().addCustomer(customer2);
8
9    }
10  }
```

Snippet 90: Umple code of class *Main2*

The tracing of the system can be seen in Snippet 91. It shows that the trace of the system is the same as for the distributed or non-distributed programs with one main method. (Snippet 84 and Snippet 87).

```
          on node number:0;
Time,Thread,UmpleFile,LineNumber,Class,Object,Operation,Name,Value
          on node number:0;
1506813410932,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newAgent
          on node number:0;
1506813410957,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newAgent
          on node number:0;
1506813410958,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newWarehouse
          on node number:0;
1506813410967,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newWarehouse
          on node number:0;
1506813410969,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newAgent
          on node number:0;
1506813410984,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newAgent
          on node number:0;
1506813410985,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newCustomer
          on node number:0; checkpoint, 1506813410993
          waiting
          on node number:1; checkpoint, 1506813402670
          waiting
          on node number:1;
1506813413019,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newCustomer
          on node number:1;
1506813588894,1,ecommerceTracer.ump,20,Customer,1401420256,me_e,orderProduct
```

```
          on node number:0;
1506813588943,1,ecommerceTracer.ump,11,Supplier,1029991479,me_e,createProduct
          on node number:0;
1506813588943,1,ecommerceTracer.ump,11,Supplier,1029991479,me_e,putInWarehouse
          on node number:0;
1506813588943,1,ecommerceTracer.ump,3,Warehouse,648129364,me_e,addProduct
          on node number:0;
1506813588943,1,ecommerceTracer.ump,11,Supplier,1104106489,me_e,createProduct
          on node number:0;
1506813588943,1,ecommerceTracer.ump,11,Supplier,1104106489,me_e,putInWarehouse
          on node number:0;
1506813588944,1,ecommerceTracer.ump,3,Warehouse,2047329716,me_e,addProduct
          on node number:0;
1506813588944,1,ecommerceTracer.ump,11,Supplier,1104106489,me_e,createProduct
          on node number:0;
1506813588944,1,ecommerceTracer.ump,11,Supplier,1104106489,me_e,putInWarehouse
          on node number:0;
1506813588944,1,ecommerceTracer.ump,3,Warehouse,648129364,me_e,addProduct
          on node number:0;
1506813588944,1,ecommerceTracer.ump,11,Supplier,1029991479,me_e,createProduct
          on node number:0;
1506813588944,1,ecommerceTracer.ump,11,Supplier,1029991479,me_e,putInWarehouse
          on node number:0;
1506813588944,1,ecommerceTracer.ump,3,Warehouse,648129364,me_e,addProduct
          on node number:0;
1506813588944,1,ecommerceTracer.ump,11,Supplier,1104106489,me_e,createProduct
          on node number:0;
1506813588944,1,ecommerceTracer.ump,11,Supplier,1104106489,me_e,putInWarehouse
          on node number:0;
1506813588944,1,ecommerceTracer.ump,3,Warehouse,2047329716,me_e,addProduct
          on node number:0;
1506813588944,1,ecommerceTracer.ump,11,Supplier,1104106489,me_e,createProduct
          on node number:0;
1506813588944,1,ecommerceTracer.ump,11,Supplier,1104106489,me_e,putInWarehouse
          on node number:0;
1506813588944,1,ecommerceTracer.ump,3,Warehouse,648129364,me_e,addProduct
          on node number:0;
1506813588944,1,ecommerceTracer.ump,11,Supplier,1029991479,me_e,createProduct
          on node number:0;
1506813588944,1,ecommerceTracer.ump,11,Supplier,1029991479,me_e,putInWarehouse
          on node number:0;
1506813588944,1,ecommerceTracer.ump,3,Warehouse,648129364,me_e,addProduct
          on node number:0;
1506813588944,1,ecommerceTracer.ump,11,Supplier,1104106489,me_e,createProduct
          on node number:0;
1506813588944,1,ecommerceTracer.ump,11,Supplier,1104106489,me_e,putInWarehouse
          on node number:0;
1506813588944,1,ecommerceTracer.ump,3,Warehouse,648129364,me_e,addProduct
          on node number:0;
1506813588944,1,ecommerceTracer.ump,11,Supplier,1104106489,me_e,createProduct
          on node number:0;
1506813588944,1,ecommerceTracer.ump,11,Supplier,1104106489,me_e,putInWarehouse
          on node number:0;
1506813588944,1,ecommerceTracer.ump,3,Warehouse,2047329716,me_e,addProduct
          on node number:0;
1506813588944,1,ecommerceTracer.ump,11,Supplier,1029991479,me_e,createProduct
          on node number:0;
1506813588944,1,ecommerceTracer.ump,11,Supplier,1029991479,me_e,putInWarehouse
          on node number:0;
1506813588944,1,ecommerceTracer.ump,3,Warehouse,2047329716,me_e,addProduct
          on node number:0;
1506813588944,1,ecommerceTracer.ump,20,Customer,967572623,me_e,orderProduct
          on node number:0;
1506813588944,1,ecommerceTracer.ump,15,Vendor,1368884364,me_e,findProduct
          on node number:0;
1506813588944,1,ecommerceTracer.ump,3,Warehouse,2047329716,me_e,findProduct
          on node number:0;
1506813588944,1,ecommerceTracer.ump,15,Vendor,1368884364,me_e,makeOrder
```

```
          on node number:1;
1506813588946,15,ecommerceTracer.ump,20,Customer,1401420256,me_e,orderProduct
          on node number:0;
1506813588952,15,ecommerceTracer.ump,15,Vendor,401625763,me_e,findProduct
          on node number:0;
1506813588952,15,ecommerceTracer.ump,3,Warehouse,2047329716,me_e,findProduct
          on node number:0;
1506813588957,15,ecommerceTracer.ump,15,Vendor,401625763,me_e,makeOrder
          on node number:0;
1506813588971,1,ecommerceTracer.ump,20,Customer,967572623,me_e,orderProduct
          on node number:0;
1506813588971,1,ecommerceTracer.ump,15,Vendor,1368884364,me_e,findProduct
          on node number:0;
1506813588971,1,ecommerceTracer.ump,3,Warehouse,2047329716,me_e,findProduct
          on node number:0;
1506813588971,1,ecommerceTracer.ump,15,Vendor,1368884364,me_e,makeOrder
          on node number:1;
1506813588972,15,ecommerceTracer.ump,20,Customer,1401420256,me_e,orderProduct
          on node number:0;
1506813588973,15,ecommerceTracer.ump,15,Vendor,401625763,me_e,findProduct
          on node number:0;
1506813588973,15,ecommerceTracer.ump,3,Warehouse,2047329716,me_e,findProduct
          on node number:0;
1506813588975,15,ecommerceTracer.ump,15,Vendor,401625763,me_e,makeOrder
          on node number:0;
1506813588980,1,ecommerceTracer.ump,20,Customer,967572623,me_e,orderProduct
          on node number:0;
1506813588980,1,ecommerceTracer.ump,15,Vendor,1368884364,me_e,findProduct
          on node number:0;
1506813588980,1,ecommerceTracer.ump,3,Warehouse,2047329716,me_e,findProduct
          on node number:0;
1506813588980,1,ecommerceTracer.ump,15,Vendor,1368884364,me_e,makeOrder
```

Snippet 91: Tracing of the system with nodes running in parallel(1)

```
          on node number:0;
Time,Thread,UmpleFile,LineNumber,Class,Object,Operation,Name,Value
          on node number:0;
1506813580719,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newAgent
          on node number:0;
1506813580745,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newAgent
          on node number:0;
1506813580745,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newWarehouse
          on node number:0;
1506813580759,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newWarehouse
          on node number:0;
1506813580762,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newAgent
          on node number:0;
1506813580777,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newAgent
          on node number:0;
1506813580778,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newCustomer
          on node number:0; checkpoint, 1506813580791
          waiting
          on node number:1; checkpoint, 1506813583534
          waiting
          on node number:1;
1506813588812,1,ecommerceRMI0,20,UmpleRuntime,1848402763,me_e,newCustomer
          ….
```

Snippet 92: Tracing of the system with nodes running in parallel (2)

However, running the system multiple times gives us different times for the checkpoints. In Snippet 91, the checkpoint on node 1 happens before checkpoint node 0. In Snippet 92, the checkpoint of

node 0 is reached before node 1. It shows that the nodes are running in parallel before customer2 is created. After that node 1 does nothing.

5.1.7 Threats to Validity

The case studies were performed to validate the distributed object facility of Umple. However, not all possible circumstances were explored. We mention the following threats to validity:

1. The measured time includes all the time taken for running a Java system. Although we did subtract the CPU time of a simple system (load of VM, but no transactions) from our comparisons, there are other factors that can make the measured time unreliable. Inconsistencies in the initial CPU time and real time of starting the JVM, garbage collection, other background processes, and handling caches are examples of such factors.
2. The system might not work in a system with a very large number of nodes. We did not test more than 10 nodes.
3. We did not test the system on more than three physical machines.
4. Network loss at different times, very slow networks, and high network load might affect the results.
5. Different Java versions and platforms were not tested comprehensively.

5.2 Using Active Objects and Queued State Machines

The distributed feature can be used to distribute active objects as well. These active objects include queued and pooled state machines, as well as Umple's active object notation. Umple active objects that run in separate threads can be put on separate nodes. To distribute the active object, the same distributable keyword is used. The runtime components of the active objects should be specified on creation of the object similar to other objects.

Queued and pooled state machines have separate threads to queue the events and to process them. Since each event is implemented as a method in the generated code and event invocation is a method call, the state machine can be distributed making the class that holds the state machine distributable, and triggering events in that state machine by calling the event methods from a different node.

Active objects can all run on a single node or on multiple nodes; the behavior of the system would be same. The difference is when the system runs on a separate node, we have multiple threads on a single machine. By distributing active objects, the same total number of threads (plus the main threads of each node) run on multiple nodes. Therefore, by distributing the active objects, we can distribute the threads. By distributing the threads on multiple machines, we transform a concurrent system into a parallel system and can utilize more CPU power at the same time. For example, the following is an example of a distributed system that improves real time performance.

We created a simple active object called IteratorMachine to show distributed active objects. IteratorMachines start on separate threads when constructed and each set its iterator to a number (start). They then iterate and increment that number 1000000 times. Iterators sleep a certain time on each iteration., Each iterator sends a message to the manager (method call) with the new number (initial number plus 1000000) after finishing the iterations.

The manager is a singleton state machine responsible for receiving messages and summing up the iterated numbers. It is a state machine that has two states: s1 and s2. When in s1, a message event may occur. Message event has a parameter which is a number. When it is in s2, it queues the message events. The state machine is a queued state machine so it can receive e1 events when in state s2.

To show how the parallel program is faster by making the initialization overhead negligible; we use system sleep so that each thread sleeps 5 seconds in each iteration. The Umple code of the system can be seen in Snippet 93.

```
1  class IteratorMachine
2  {
3    Integer iterator;
4      active {
5          for(int i=0;i<1000000;i++)
6          {
7           try { Thread.sleep(5000); } catch (InterruptedException interruptedException) {};
8           iterate();
9          }
10         Manager.getInstance().message(iterator);
11         iterator=0;
12     }
13     void iterate()
14     {
15       iterator+=1;
16     }
17 }
18 class Manager
```

```
19  {
20    singleton;
21    Integer received=0;
22    Integer number=0;
23      queued sm{
24      s1 {
25        message(int i) /{
26          number+=i;
27            } ->s2;
28      }
29      s2 {
30        entry /{
31            received+=1;
32       } ->s1;
33      }
34    }
35  }
```

Snippet 93: Umple code of iterator system

The main thread of the system creates manager and iterator machines and waits in a loop until the job is done by other threads.

We tested the real time by running 4 iterators on a single node versus the case two nodes on two separate machines and 2 iterators on each node. The single node took on average 118,380 milliseconds. The distributed one took 36,458 milliseconds on average.

6 Conclusions and Future Work

In this thesis, we designed and implemented a code generation feature for distributed systems in Umple. We designed a simple syntax for the Umple language to model distribution without concerns about the target platform or the exact number of nodes. The Java code we generate can run on one or more nodes.

The system is implemented in a test-driven environment with more than 50 tests running on each build, testing distributable classes with inheritance, associations, and mixins for different patterns.

The new feature can be used to create distributed programs in Umple and run them as Java programs. The feature can also be used to transform current Umple programs into distributed systems. The transformation of the single-node system into a distributed system is very straightforward and the syntax is integrated into the Umple modelling language.

The goal was to distribute a non-distributed system with the least changes in the code. We allow generation of distributed code with only one switch. The extra keywords only provide more options for the developer to optimize the system. The syntax of the configuration file only consists of brackets and commas to show inclusion of nodes and runtime components.

Umple can generate distributed systems with nodes communicating using different technologies. It currently supports RMI and web services and other technologies can be added as future work.

Using Umple as model-driven development tool for creating distributed system has the following benefits. Providing these benefits constitute the key contributions of this thesis.

- The system can be generated in distributed mode but run only on one node. This allows the system to be tested without object placement and network concerns. If there is a failure after distributing the system, the problem is with the underlying network protocols.
- Umple generates readable and transparent generated code with minimum library dependency instantly, based on the model. The alternative, using third-party distributed system middleware would mean the user must modify and/or annotate the generated code to use it as source for the third-party tool.

- Umple does various kinds of analysis of the model and makes some necessary logical decisions: for example; it distributes the classes that are subclasses of distributed classes, and it warns the user if there is an association between a distributed and a non-distributed class.

- Several patterns in the generated code are supported, that vary according to the different communication technologies. The users can choose their preferred one based on performance and convenience. However, the most convenient pattern is set by default in Umple.

- There can be different choices for distributing objects with different design patterns in the model. For example, singleton objects can be defined as only one object in the whole system or only one object in each node.

We explored different mechanisms and patterns to generate clean code that can be run on multiple machines while keeping the same structure of object oriented systems such as classes, inheritance, and associations. This was a challenge due to differences in the communication technologies.

We showed how to create a parallel program by distributing the system and starting active nodes. Using the concurrency feature and the distributed features, we can create scalable parallel systems.

We verified the correctness of distributed programs by comparing the traces and results with their non-distributed pairs. However, the user must pay attention to parallel programming limitations and methodologies when programming parallel systems.

6.1 Answers to the Research Questions

- How can Umple generate code that can run on different nodes?

  Umple distributes the system by distributing objects. It generates distributable classes and provides API to distribute the instances of the distributable classes. The objects are distributed over different nodes by the UmpleRuntime objects generated by Umple.

- What should the architecture be for a distributed system generated by Umple?

  Several patterns in the generated code are supported, which vary according to different communication technologies. We showed how we improved the patterns to find the best pattern that matches Umple’s code generation architecture.

- What communication technologies should be used to distribute objects in Umple?

  We used RMI and web services as communication technologies. These technologies are widely used for distributed objects and remote method call and provide a level of abstraction over TCP network communications.

- What kind of simple annotations could be used by the developers to define distributed systems.

  We showed how a user can use the simple ‘distributable’ keyword for classes that are going to be distributed. We also introduced runtime components to annotate the placement of the objects during runtime. However, Umple can create a distributed system by distributing all of the objects without annotations.

- How can the system generate distributed and non-distributed executable code out of the same model? How should the generated code differ from generated code for a non-distributed system?

  We designed Umple to create the distributed code out of a simple code with either no difference or simple annotations. Umple generates similar Java code for distributed systems as with non-distributed systems. However, there is some extra code in the distributed code which includes proxy classes and methods, communication interfaces, and UmpleRuntime.

- How can we defer the decision of object placement and number of machines to the run-time instead of during modeling and development?

  Runtime components are used to locate the objects during development. By placing the runtime components on different nodes during runtime, we can defer the object placement; allocation of objects at run time uses a configuration file.

- What should the mechanism of object creation be on a remote machine?

  There is an object on each node to listen for object creation commands and work as an object factory. UmpleRuntime is a singleton class that runs on every node in a distributed system. The UmpleRuntime object on the initiating node asks the UmpleRuntime on the target node to create the object and return the proxy. The mechanism is described in Chapter 4.

- What constraints are added to the model when the system is distributed?

  Remote method calls are slower than local method calls. Therefore, a simple system can be considerably slower when distributed. The developer might need to tweak the code to optimize performance by reducing the remote method calls. The performance can be increased by better object placement.

  Distributing a non-distributed system can be useful when the data is distributed or when there are some processes that need to be executed on a certain machine, but distributing sequential code will reduce performance.

  If the user wishes to increase speed by distributing the system, he or she needs to develop a parallel system with different objects doing heavy processing concurrently. The distributed feature can be used to improve the speed of such a system.

6.2 Future Work

There are some shortcomings in the current implementation that could be improved in the future; additional features related to distributed systems could also be added. These are described in the following subsections.

6.2.1 Code distribution optimization

Not all classes have objects on every node and not all the code executes on every node. For classes that have no objects on a certain node, an optimization would be to remove the code from the program on that node and only keep the code for the proxy or interface. There can also be a mechanism to send the code of the object to be created to the target node that can also be a future work.

6.2.2 Node placement optimization

Automatic optimized object placement: to place objects on the nodes optimized for better performance. This is more of a cluster system challenge than a general distributed application concern, but could be added as a feature in the future.

6.2.3 Network failure recovery

There should be a mechanism to deal with the remote objects in case of network failure. There could be a mechanism to re-create the objects and resume the system or clear the system of objects and recreate them on another node.

6.2.4 Verifying correctness of configuration files

It would be useful to have a simple tool to check the configuration files. While currently the system disregards the recurrence of the same runtime component in different nodes, it would be useful to find redundant runtime components in the configuration files.

6.2.5 Object migration

While Umple can distribute objects on different nodes by creating them on separate nodes, it cannot move an object to another node after it is created. This requires serializing the object and sending it to another node and reconfiguring the proxies to point to the new object.

6.2.6 Smarter Runtime system

It would be useful to add more features to the runtime system to optimize the CPU time based on the configuration file. For example, it becomes a local system when there is only one node. Currently we assume that a node might be added to the system at any time. A future task could be to add syntax to configuration file to control this.

6.2.7 Other platforms

Finally, distribution can be done using other distribution middleware tools and libraries such as pure TCP socket connection or RESTful web services.

The system can also be implemented for other languages that Umple supports. The modelling semantics and Umple syntax can be used for other object-oriented languages with minimum modifications. These languages such as PHP, C++, and Ruby have different libraries to support

distribution and web services. However, most of the methods used in this thesis can be used for code generation of distributed systems in these languages.

The system can be able use different technologies for different features. For example, since queued and pooled state machines need asynchronous method calls to be implemented, message passing technologies can be used for queued and pooled state machines instead of implementing them with extra methods as is currently the case in Umple.

References

[1] “Umple,” 2017. [Online]. Available: http://www.umple.org.

[2] OM Group. "OMG Unified Modeling Language (OMG UML), Superstructure." Open Management Group (2009).

[3] E. Tilevich and Y. Smaragdakis, “J-Orchestra: Enhancing Java Programs with Distribution Capabilities,” *ACM Trans. Softw. Eng. Methodol.*, vol. 19, no. 1, p. 1:1--1:40, 2009.

[4] M. Philippsen and M. Zenger, “JavaParty - transparent remote objects in Java,” *Concurr. Pract. Exp.*, vol. 9, no. 11, pp. 1225–1242, 1997.

[5] C. George, D. Jean, and K. Tim, “Distributed Systems, concepts and design,” *Addison-Wesley*, vol. 2, p. 410, 2005.

[6] R. S. Chin and S. T. Chanson, “Distributed object-based programming systems,” *ACM Comput. Surv.*, vol. 23, no. 1, pp. 90–124, 1991.

[7] “Java.net: Jax-ws,” 2017 .[Online]. Available: https://javaee.github.io/metro-jax-ws/

[8] “Sun Microsystems, Java™ Remote Method Invocation Specification.”

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Software*. Addison-Wesley Longman Publishing Co., 1996.

[10] O. O. Adesina, “Integrating formal methods with model-driven engineering,” *4th Int. Conf. Softw. Eng. Adv. ICSEA 2009, Incl. SEDES 2009 Simp. para Estud. Doutor. em Eng. Softw.*, vol. 1531, pp. 86–92, 2009.

[11] O. O. Adesina, T. C. Lethbridge, and S. S. Somé, “A Fully Automated Approach to Discovering Nondeterminism in State Machine Diagrams,” *10th Int. Conf. Qual. Inf. Commun. Technol.*, no. September, pp. 73–78, 2016.

[12] “UmpleOnline,” 2017. [Online]. Available: http://www.try.umple.org.

[13] CRUiSE, “Umple User Manual,” *Umple User Manual*, 2015. [Online]. Available: http://cruise.eecs.uottawa.ca/umple/GettingStarted.html.

[14] V. Abdelzad and T. C. Lethbridge, “Promoting traits into model-driven development,” *Softw. Syst. Model.*, vol. 16, no. 4, pp. 997–1017, 2017.

[15] M. P. Interface, R. Hempel, T. Hey, and D. W. Walker, “Message Passing Interface,” pp. 1–13, 2015.

[16] E. Curry and P. Grace, “Flexible self-management using the model-view-controller pattern,” *IEEE Softw.*, vol. 25, no. 3, 2008.

[17] S. U. N. Rpc, “remote procedure call protocol specification Version 2; RFC1058,” *Internet Req. Comments*, vol. 1057, p. 15, 1988.

[18] Winer, D. XML-RPC specification, October 1999. URL http://www. xmlrpc. org/spec.

[19] P. Obermeyer and J. Hawkins, “Microsoft .NET Remoting: A technical overview.” 2001.

[20] O. M. G. Portal, “Object Management Group: Common Object Request Broker,” *OMG. org. Retrieved from omg. org August*, vol. 22, p. 2009, 2009.

[21] C. Week, “Middleware Distributed-Object Middleware,” vol. 11, pp. 1–11.

[22] W. Yu and A. Cox, “Java / DSM : A Platform for Heterogeneous Computing 1 Introduction,” *Concurr. Pract. Exp.*, vol. 9, no. 11, pp. 1213–1224, 1997.

[23] E. Freeman, S. Hupfer, and K. Arnold, *JavaSpaces principles, patterns, and practice*. Addison-Wesley Professional, 1999.

[24] M. H. Orabi, A. H. Orabi, and T. Lethbridge, “Umple as a component-based language for the development of real-time and embedded applications,” *4th Int. Conf. Model. Eng. Softw. Dev. Model. 2016*, no. April, pp. 282–291, 2016.

[25] Launay, P., & Pazat, J. L. (1998, September). The Do! project: Distributed Programming Using Java. In First UK Workshop Java for High Performance Network Computing.

[26] O. Holder, I. Ben-Shaul, and H. Gazit, “Dynamic layout of distributed applications in FarGo,” in *Proceedings of the 21st international conference on Software engineering*, 1999, pp. 163–173.

[27] A. Spiegel, “Pangaea: An automatic distribution front-end for java,” *Lect. Notes Comput. Sci.*

(including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), vol. 1586, no. April 1999, pp. 94–99, 1999.

[28] T. Fahringer, "JavaSymphony: A system for development of locality-oriented distributed and parallel Java applications," *Proc. - IEEE Int. Conf. Clust. Comput. ICCC*, vol. 2000–January, pp. 145–152, 2000.

[29] D. Caromel, W. Klauser, and J. Vayssière, "Towards seamless computing and metacomputing in Java," *Concurr. Pract. Exp.*, vol. 10, no. 11–13, pp. 1043–1061, 1998.

[30] A. Alghamdi, "Queued and Pooled Semantics for State Machines in the Umple Model-Oriented Programming Language," University of Ottawa, 2015.

Appendix

```
1   class SystemTracer
2   {
3                       singleton;
4                       distributable;
5                       public void handle(int node, String message)
6                       {
7                           System.out.println("on node number:"+String.valueOf(node)+"; "+message);
8                       }
9                       public void printMessage(String message)
10                      {
11                          System.out.println(message);
12                      }
13  }
```

Snippet 94: SystemTracer.ump

```
/*PLEASE DO NOT EDIT THIS CODE*/
/*This code was generated using the UMPLE 1.26.0-b05b57321 modeling language!*/

package ecommerceRMI0;
import cruise.util.ConsoleTracer;
import java.util.*;
import java.io.Serializable;

// line 25 "../ecommerceDistributableRMI0.ump"
// line 71 "../ecommerce.ump"
// line 13 "../ecommerceTracer.ump"
public class Vendor extends Agent implements java.io.Serializable, IVendorImpl
{

  //------------------------
  // MEMBER VARIABLES
  //------------------------

  //Vendor Associations
  private transient  List<Order> orders;
  private transient  List<Customer> customers;

  //------------------------
  // CONSTRUCTOR
  //------------------------

  public Vendor(String aName, UmpleRuntime.UmpleComponent umpleComponent)
  {
    super(aName,umpleComponent);
    if(umpleComponent.getNode().getId()!=UmpleRuntime.getThisNodeId())
    {
      if(this.getClass()== Vendor.class)
        UmpleRuntime.getInstance().newVendor(aName, umpleComponent, this);

      return;
    }

    orders = new ArrayList<Order>();
    customers = new ArrayList<Customer>();
  }
  //------------------------
  // Returning the Hashcode
  //------------------------
  public int getHashCodeImpl()
  {
    return hashCode();
  }

  //------------------------

  // INTERFACE
```

```
//------------------------

public Order getOrderImpl(int index)
{
  Order aOrder = orders.get(index);
  return aOrder;
}

public List<Order> getOrdersImpl()
{
  List<Order> newOrders = Collections.unmodifiableList(orders);
  return newOrders;
}

public int numberOfOrdersImpl()
{
  int number = orders.size();
  return number;
}

public boolean hasOrdersImpl()
{
  boolean has = orders.size() > 0;
  return has;
}

public int indexOfOrderImpl(Order aOrder)
{
  int index = orders.indexOf(aOrder);
  return index;
}

public Customer getCustomerImpl(int index)
{
  Customer aCustomer = customers.get(index);
  return aCustomer;
}

public List<Customer> getCustomersImpl()
{
  List<Customer> newCustomers = Collections.unmodifiableList(customers);
  return newCustomers;
}

public int numberOfCustomersImpl()
{
  int number = customers.size();
  return number;
}

public boolean hasCustomersImpl()
```

```
{
  boolean has = customers.size() > 0;
  return has;
}

public int indexOfCustomerImpl(Customer aCustomer)
{
  int index = customers.indexOf(aCustomer);
  return index;
}

public static int minimumNumberOfOrders()
{
  return 0;
}

public boolean addOrderImpl(Order aOrder)
{
  boolean wasAdded = false;
  if (orders.contains(aOrder)) { return false; }
  Vendor existingVendor = aOrder.getVendor();
  if (existingVendor == null)
  {
    aOrder.setVendor(this);
  }
  else if (!this.equals(existingVendor))
  {
    existingVendor.removeOrder(aOrder);
    addOrder(aOrder);
  }
  else
  {
    orders.add(aOrder);
  }
  wasAdded = true;
  return wasAdded;
}

public boolean removeOrderImpl(Order aOrder)
{
  boolean wasRemoved = false;
  if (orders.contains(aOrder))
  {
    orders.remove(aOrder);
    aOrder.setVendor(null);
    wasRemoved = true;
  }
  return wasRemoved;
}

public boolean addOrderAtImpl(Order aOrder, int index)
```

```
  {
    boolean wasAdded = false;
    if(addOrder(aOrder))
    {
      if(index < 0 ) { index = 0; }
      if(index > numberOfOrders()) { index = numberOfOrders() - 1; }
      orders.remove(aOrder);
      orders.add(index, aOrder);
      wasAdded = true;
    }
    return wasAdded;
  }

  public boolean addOrMoveOrderAtImpl(Order aOrder, int index)
  {
    boolean wasAdded = false;
    if(orders.contains(aOrder))
    {
      if(index < 0 ) { index = 0; }
      if(index > numberOfOrders()) { index = numberOfOrders() - 1; }
      orders.remove(aOrder);
      orders.add(index, aOrder);
      wasAdded = true;
    }
    else
    {
      wasAdded = addOrderAt(aOrder, index);
    }
    return wasAdded;
  }

  public static int minimumNumberOfCustomers()
  {
    return 0;
  }

  public boolean addCustomerImpl(Customer aCustomer)
  {
    boolean wasAdded = false;
    if (customers.contains(aCustomer)) { return false; }
    customers.add(aCustomer);
    if (aCustomer.indexOfVendor(this) != -1)
    {
      wasAdded = true;
    }
    else
    {
      wasAdded = aCustomer.addVendor(this);
      if (!wasAdded)
      {
        customers.remove(aCustomer);
```

```
    }
  }
  return wasAdded;
}

public boolean removeCustomerImpl(Customer aCustomer)
{
  boolean wasRemoved = false;
  if (!customers.contains(aCustomer))
  {
    return wasRemoved;
  }

  int oldIndex = customers.indexOf(aCustomer);
  customers.remove(oldIndex);
  if (aCustomer.indexOfVendor(this) == -1)
  {
    wasRemoved = true;
  }
  else
  {
    wasRemoved = aCustomer.removeVendor(this);
    if (!wasRemoved)
    {
      customers.add(oldIndex,aCustomer);
    }
  }
  return wasRemoved;
}

public boolean addCustomerAtImpl(Customer aCustomer, int index)
{
  boolean wasAdded = false;
  if(addCustomer(aCustomer))
  {
    if(index < 0 ) { index = 0; }
    if(index > numberOfCustomers()) { index = numberOfCustomers() - 1; }
    customers.remove(aCustomer);
    customers.add(index, aCustomer);
    wasAdded = true;
  }
  return wasAdded;
}

public boolean addOrMoveCustomerAtImpl(Customer aCustomer, int index)
{
  boolean wasAdded = false;
  if(customers.contains(aCustomer))
  {
    if(index < 0 ) { index = 0; }
    if(index > numberOfCustomers()) { index = numberOfCustomers() - 1; }
```

```
    customers.remove(aCustomer);
    customers.add(index, aCustomer);
    wasAdded = true;
   }
   else
   {
    wasAdded = addCustomerAt(aCustomer, index);
   }
   return wasAdded;
  }

  public void deleteImpl()
  {
   while( !orders.isEmpty() )
   {
    orders.get(0).setVendor(null);
   }
   ArrayList<Customer> copyOfCustomers = new ArrayList<Customer>(customers);
   customers.clear();
   for(Customer aCustomer : copyOfCustomers)
   {
    aCustomer.removeVendor(this);
   }
   super.delete();
  }

  // line 28 "../ecommerceDistributableRMI0.ump"
   public Order makeOrderImpl(Customer aCustomer, Product aProduct){
   ConsoleTracer.handle(
System.currentTimeMillis()+","+Thread.currentThread().getId()+",ecommerceTracer.ump,15,Vendor,"+System.iden
tityHashCode(this)+",me_e,makeOrder" );
   if(aProduct==null)
                                        return null;
                                Order aOrder= new Order(aProduct);
                                aOrder.setCustomer(aCustomer);
                                aOrder.setVendor(this);
                                return aOrder;
  }

  // line 75 "../ecommerce.ump"
   public Product findProductImpl(ProductType productType){
   ConsoleTracer.handle(
System.currentTimeMillis()+","+Thread.currentThread().getId()+",ecommerceTracer.ump,15,Vendor,"+System.iden
tityHashCode(this)+",me_e,findProduct" );
   for(Warehouse w:getWarehouses())
                                {
                                        Product p= w.findProduct(productType);
                                        if(p!=null)
                                                return p;
                                }
                                return null;
```

```
  }

  public void setRealObject(IVendorImpl aObject)
  {
  super.setRealObject(aObject);
    realObject=aObject;
  }

  transient IVendorImpl realObject=this;
  public Vendor(String aName)
  {

this(aName,UmpleRuntime.getComponent("Vendor"));
  }
  public int getHashCode()
  {
    while(true)
      try{
        return realObject.getHashCodeImpl();
        }
    catch(Exception e) {System.err.println(e.toString());}
  }
  public Order getOrder(int index)
  {
    while(true)
      try{
        return realObject.getOrderImpl(index);
        }
    catch(Exception e) {System.err.println(e.toString());}
  }
  public List<Order> getOrders()
  {
    while(true)
      try{
        return realObject.getOrdersImpl();
        }
    catch(Exception e) {System.err.println(e.toString());}
  }
  public int numberOfOrders()
  {
    while(true)
      try{
        return realObject.numberOfOrdersImpl();
        }
    catch(Exception e) {System.err.println(e.toString());}
  }
  public boolean hasOrders()
  {
    while(true)
      try{
        return realObject.hasOrdersImpl();
```

```
      }
  catch(Exception e) {System.err.println(e.toString());}
}
public int indexOfOrder(Order aOrder)
{
  while(true)
    try{
      return realObject.indexOfOrderImpl(aOrder);
      }
  catch(Exception e) {System.err.println(e.toString());}
}
public Customer getCustomer(int index)
{
  while(true)
    try{
      return realObject.getCustomerImpl(index);
      }
  catch(Exception e) {System.err.println(e.toString());}
}
public List<Customer> getCustomers()
{
  while(true)
    try{
      return realObject.getCustomersImpl();
      }
  catch(Exception e) {System.err.println(e.toString());}
}
public int numberOfCustomers()
{
  while(true)
    try{
      return realObject.numberOfCustomersImpl();
      }
  catch(Exception e) {System.err.println(e.toString());}
}
public boolean hasCustomers()
{
  while(true)
    try{
      return realObject.hasCustomersImpl();
      }
  catch(Exception e) {System.err.println(e.toString());}
}
public int indexOfCustomer(Customer aCustomer)
{
  while(true)
    try{
      return realObject.indexOfCustomerImpl(aCustomer);
      }
  catch(Exception e) {System.err.println(e.toString());}
}
```

```
public boolean addOrder(Order aOrder)
{
  while(true)
    try{
      return realObject.addOrderImpl(aOrder);
      }
  catch(Exception e) {System.err.println(e.toString());}
}
public boolean removeOrder(Order aOrder)
{
  while(true)
    try{
      return realObject.removeOrderImpl(aOrder);
      }
  catch(Exception e) {System.err.println(e.toString());}
}
public boolean addOrderAt(Order aOrder, int index)
{
  while(true)
    try{
      return realObject.addOrderAtImpl(aOrder,index);
      }
  catch(Exception e) {System.err.println(e.toString());}
}
public boolean addOrMoveOrderAt(Order aOrder, int index)
{
  while(true)
    try{
      return realObject.addOrMoveOrderAtImpl(aOrder,index);
      }
  catch(Exception e) {System.err.println(e.toString());}
}
public boolean addCustomer(Customer aCustomer)
{
  while(true)
    try{
      return realObject.addCustomerImpl(aCustomer);
      }
  catch(Exception e) {System.err.println(e.toString());}
}
public boolean removeCustomer(Customer aCustomer)
{
  while(true)
    try{
      return realObject.removeCustomerImpl(aCustomer);
      }
  catch(Exception e) {System.err.println(e.toString());}
}
public boolean addCustomerAt(Customer aCustomer, int index)
{
  while(true)
```

```
    try{
      return realObject.addCustomerAtImpl(aCustomer,index);
      }
  catch(Exception e) {System.err.println(e.toString());}
}
public boolean addOrMoveCustomerAt(Customer aCustomer, int index)
{
  while(true)
    try{
      return realObject.addOrMoveCustomerAtImpl(aCustomer,index);
      }
  catch(Exception e) {System.err.println(e.toString());}
}
public void delete()
{
  while(true)
    try{
      realObject.deleteImpl();
      break;
      }
  catch(Exception e) {System.err.println(e.toString());}
}
 public Order makeOrder(Customer aCustomer, Product aProduct)
{
  while(true)
    try{
      return realObject.makeOrderImpl(aCustomer,aProduct);
      }
  catch(Exception e) {System.err.println(e.toString());}
}
 public Product findProduct(ProductType productType)
{
  while(true)
    try{
      return realObject.findProductImpl(productType);
      }
  catch(Exception e) {System.err.println(e.toString());}
}

public void setRemoteObject(IVendorImpl aRemoteObject)
{
  remoteObject=aRemoteObject;
}
public IVendorImpl getRemoteObject()
{
  return (IVendorImpl)remoteObject;
}
private void readObject(java.io.ObjectInputStream in) throws Exception
{
  try
  {
```

```
      in.defaultReadObject();
      realObject=(IVendorImpl)remoteObject;
    }
    catch(Exception e)
    {
      throw e;
    }

  }

  public boolean equals(Object obj)
  {    if(obj.getClass()!=this.getClass())
      return false;
    return (getHashCode()==((Vendor)obj).getHashCode());
  }
}
```

Snippet 95: Generated Java code for Vendor: Vendor.java (RMI default pattern)

```
/*PLEASE DO NOT EDIT THIS CODE*/
/*This code was generated using the UMPLE 1.26.0-b05b57321 modeling language!*/

package ecommerceRMI0;
import cruise.util.ConsoleTracer;
import java.util.*;
import java.io.Serializable;
import java.rmi.RemoteException;
public interface IVendorImpl extends java.rmi.Remote, IAgentImpl
{
  public int getHashCodeImpl() throws RemoteException;
  public Order getOrderImpl(int index) throws RemoteException;
  public List<Order> getOrdersImpl() throws RemoteException;
  public int numberOfOrdersImpl() throws RemoteException;
  public boolean hasOrdersImpl() throws RemoteException;
  public int indexOfOrderImpl(Order aOrder) throws RemoteException;
  public Customer getCustomerImpl(int index) throws RemoteException;
  public List<Customer> getCustomersImpl() throws RemoteException;
  public int numberOfCustomersImpl() throws RemoteException;
  public boolean hasCustomersImpl() throws RemoteException;
  public int indexOfCustomerImpl(Customer aCustomer) throws RemoteException;
  public boolean addOrderImpl(Order aOrder) throws RemoteException;
  public boolean removeOrderImpl(Order aOrder) throws RemoteException;
  public boolean addOrderAtImpl(Order aOrder, int index) throws RemoteException;
  public boolean addOrMoveOrderAtImpl(Order aOrder, int index) throws RemoteException;
  public boolean addCustomerImpl(Customer aCustomer) throws RemoteException;
  public boolean removeCustomerImpl(Customer aCustomer) throws RemoteException;
  public boolean addCustomerAtImpl(Customer aCustomer, int index) throws RemoteException;
  public boolean addOrMoveCustomerAtImpl(Customer aCustomer, int index) throws RemoteException;
  public void deleteImpl() throws RemoteException;
   public Order makeOrderImpl(Customer aCustomer, Product aProduct) throws RemoteException;
   public Product findProductImpl(ProductType productType) throws RemoteException;
}
```

Snippet 96: Generated code for Vendor: IVendor.java (RMI default pattern)