View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Saskatchewan's Research Archive

DESIGNING A FRAMEWORK FOR RESTFUL MULTI-AGENT SYSTEMS

A Thesis Submitted to the College of

Graduate Studies and Research

In Partial Fulfillment of the Requirements

For the Degree of Masters of Science

In the Department of Computer Science

University of Saskatchewan

Saskatoon

By

ABDULLAH ALTHAGAFI

© Copyright Abdullah Althagafi, October, 2012. All rights reserved.

PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science

176 Thorvaldson Building

110 Science Place

University of Saskatchewan

Saskatoon, Saskatchewan

Canada

S7N 5C9

ABSTRACT

Nowadays there are many systems that require some degree of automation. To attain this automation, agent technology has generally been found to be a promising approach. An agent is a piece of software that does activities on behalf of a user or another program. However, designing and deploying an agent infrastructure that achieves scalability is still a major challenge.

In this thesis, a pattern for designing agents following RESTful principles is proposed in an effort to address the aforementioned challenges. In addition, the pattern will follow the FIPA Abstract Architecture; which is aimed at developing intelligent agents and supporting interoperability among agents and agent-based systems. Furthermore, an evaluation is done to investigate the scalability of the deployment of a RESTful multi-agent system.

ACKNOWLEDGEMENTS

First of all I would like to express my genuine thanks to my supervisor, Dr. Ralph Deters for his advice, intuitive criticisms, and patience throughout my entire duration of study. Thanks to the members of my advisory committee.

Thanks to all students of the Multi-Agent Distributed Mobile and Ubiquitous Computing (MADMUC) Lab for their friendship.

Finally, I would like to thank my wife, my daughter, and my entire family for the unconditional love and generosity.

TABLE OF CONTENTS

Page

PERMISSION TO USE i
ABSTRACT ii
ACKNOWLEDGEMENTS iii
TABLE OF CONTENTS iv
LIST OF TABLES vi
LIST OF FIGURES vii
LIST OF ABBREVIATIONS ix
Chapter 1 INTRODUCTION 1
Chapter 2 PROBLEM STATEMENT 4
Chapter 3 LITERATURE REVIEW 7
3.1 Introduction 7
3.2 Factors affecting Multi Agents Systems 7
3.3 Distributed Multi Agent Systems 10
3.4 Scalable Hierarchical Coordination of Multi-Agent Resource Usage 12
3.5 Agents and web services (SOAP) 13
3.6 REST 14
3.7 HATEOAS and RESTful design 14
3.8 REST and Distributed Systems 15
3.9 REST and SOAP Web Services 18
3.10 REST vs SOAP web services design approaches 21
3.11 Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services 22
3.12 Conclusion 26
3.12.1 Key Findings 26
3.12.2 RESTful distributed agent communication 29
3.12.3 Development of runtime environment 29
Chapter 4 THE PROPOSED APPROACH 31
4.1 The Programming Environment 31
4.2 RESTful Architectural Style 33
4.3 ACL Abstraction 34
4.4 Conclusion 41
Chapter 5 THE IMPLEMENTATION 43
5.1 The Architecture 43
5.1.1 Agent communication 46
5.1.2 Service Directory 46
5.1.3 ACL Representation 47
5.1.4 Envelope Representation 48
5.1.5 Message Transport 48
5.1.6 Transport Protocol 49
5.2 Implementation Code Samples & Examples 50
5.2.1 Starting a communication 51
5.2.2 Message Structure (envelope) 52

5.2.3 Agent Message Transport & Transport 54
5.2.4 Agent Management System 56
5.2.5 Speech Act /ACL Representation 58
5.3 Conclusion 61
Chapter 6 SYSTEM PERFORMANCE EVALUATION 63
6.1 Experiment Goals 64
6.2 Parameters 65
6.3 Experiments setup 66
6.4 Results and Evaluations 69
6.5 Conclusion 74
Chapter 7 CONCLUSION AND CONTRIBUTION 77
Chapter 8 future work 82
8.1 Introduction 82
8.2 Future directions 82
Chapter 9 LIST OF REFERENCES 84

LIST OF TABLES

Table page

Table 3-1 Comparing REST vs SOAP......22

Table 3-2. Literature review summary......26

Table 4-1. FIPA Communicative Act Specifications and their implemented in RESTful approach......35

Table 6-1. Experiments and Goals......65

Table 6-2 Experiment 2 parameters......67

Table 6-3 Experiment 3 parameters......68

Table 6-4 Experiment 4 parameters......68

Table 6-5 Experiment 5 parameters......69

LIST OF FIGURES

Figure page

Figure 2.1: FIPA Abstract Architecture Mapped to Various Concrete Realizations............4

Figure 3.1. Multi-agent model in an hierarchical structure 8

Figure 3.2. Steps involved in iterative contracting 9

Figure 4.1. Cooperation between two Agents..32

Figure 4.2. Agents in one environment..37

Figure 4.3. Finding Agent..37

Figure 4.4. Agent communication via Service BUS..38

Figure 4.5. Agent creates a call for proposal 39

Figure 4.6. Agent informing other agents 39

Figure 4.7. Agent monitors others and has call back 40

Figure 4.8. Agent C accepts agent A’s proposal...40

Figure 5.1. FIPA Abstract Architecture 44

Figure 5.2. A Message Represented as an Envelope 48

Figure 5.3. Message Transport Reference Model...49

Figure 5.4 A ping pong agent example where two agents send messages to each other......51

Figure 5.5 Creating an agent in Erlang 51

Figure 5.6 Creating an agent using spawn function..52

Figure 5.7 Sending message to the created agent 52

Figure 5.8 Message receiving snippet...52

Figure 5.9 Message structure 53

Figure 5.10 Format of the message..54

Figure 5.11 How to send message 54

Figure 5.12 Sending message to a targeted agent 54

Figure 5.13 How an agent sends message to itself 55

Figure 5.14 The receiving block 55

Figure 5.15 Decreasing an argument by 1 55

Figure 5.16 How to terminate a communication 55

Figure 5.17 Sending message operation 56

Figure 5.18. Agent registration process 57

Figure 5.19 Creating an agent - agentb 57

Figure 5.20 Registering agents in the AMS 58

Figure 5.21. Agent create call for proposal 58

Figure 5.22 Posting the proposal 59

Figure 5.23. Agent uses the service Bus to broadcast information 59

Figure 5.24 How to inform agents about a proposal 59

Figure 5.25 How to check the proposal 60

Figure 5.26. Agent uses PATCH routine to receive information 60

Figure 5.27. Agent B accepts the proposal 61

Figure 5.28 How to check if proposal is accepted 61

Figure 6.1 Agent creation times 70

Figure 6.2 Parallel message sending times with both numbers of agents and messages constant 71

Figure 6.3 Messages sending times with constant numbers of agents and varying number of messages 72

Figure 6.4 Messages sending times with constant numbers of messages and varying number of agents 73

LIST OF ABBREVIATIONS

Abbreviation	
ACID	Atomic, Consistent, Isolated, Durable
ACL	Agent Communication Language
AMS	Agent Management System
AVEB	Audio Video Entertainment Broadcasting
BOINC	Berkeley Open Infrastructure For Network Computing
CAP	Consistency, Availability, Partition-Tolerance
CPU	Central Processing Unit
FIPA	Foundation For Intelligent Physical Agents
GB	Gigabyte
HATEOAS	Hypermedia As The Engine Of Application State
HTTP	Hypertext Transfer Protocol
JADE	Java Agent Development Framework
JPEG	Joint Photographic Experts Group
JRE	Java Runtime Environment
MAS	Multi-Agent System
OS	Operating System
OTP	Open Telecom Platform
PID	Process Identifier
RAM	Random Access Memory
REST	Representational State Transfer
SICS	Swedish Institute Of Computer Science
SMP	Symmetric Multi-Processing
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WS*	Web Services
WSDL	Web Services Description Language
WWW	World Wide Web
XML	Extensible Markup Language

CHAPTER 1 INTRODUCTION

In 1973, Carl Hewitt et al. [25] described the “Actor Model” as a mathematical model of concurrent computation that treats "actors" as the universal primitives of concurrent digital computation; in response to a message that it receives, an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message received. By the end of the twentieth century, intelligent software agents, intersecting with the then implementations of distributed computing, were defined and described by Stan Franklin and Art Graesser (1997) [18], and distinguished from programs in their autonomy, goal-orientation, ability to react to the environment and persistence.

Many practical applications were predicted as suitable areas where agents were to be in solving problems from data-mining, military, surveillance, e-mail to personal-shopper agents and more; but it quickly became evident that scalability was to be a major issue, as large numbers of agents were required to address these practical application problems. With large numbers of agents, additional design issues stemmed including hierarchies, messaging, collaboration, prioritization and scheduling. Different systems that are currently in use need some degree of automation to perform their designated task. To attain this automation, agents could be created and used in the systems.

An agent is a piece of software that does activities for a user or another program [31]. Agents make it easier to program complex application [2]. Much progress has been made in the study and use of multi-agent systems. This progress has allowed the identification of some features about multi-agents. Some of these features [17] are their autonomy and their ability to react to situations. Autonomy means that the agent has the capability to choose, prioritize tasks, and make decision without human intervention. There are two ways an agent could be designed to

react to a given situation either proactively or reactively. Proactive agents anticipate a situation and take charge of it. Reactive agents interpret the environment they are in and react to it appropriately. This thesis research will focus on reactive multi-agent systems. Most multi-agent systems are designed following FIPA ACTS [14]. There has been success for this design approach. However, there are some limitations to this approach of designing multi-agent systems. This research identifies scalability and dependability, compliance to web standards, and following RESTful Speech Acts as issues that that need to be addressed. To design scalable multi-agent systems that satisfy the identified issues, web services approaches such as REST or SOAP could be used. This research chooses to implement and examine REST approach. Much of the research in the last fifteen years, focused on the scalability issue, was constrained by attempting to solve the problem within a relatively bounded environment. This naturally came up against the absoluteness of finite resources. And unfortunately, the previously mentioned issues (hierarchies, messaging, communication, collaboration, prioritization and scheduling) did not lend themselves to distribution solutions.

The nature of the larger problem demands a new strategy. This research will show that two existing mechanisms can successfully address these issues to achieve stability and scalability. The first is a relatively new (1986) [3] programming language, Erlang, which is more suited for managing large numbers of small programs, as it was designed to manage telephone-switching, concurrent programming and messaging. To this, the problems of hierarchies, messaging, prioritization and scheduling will be relegated.

The next is Representational state transfer [29, 30], defined in 2000 by Roy Fielding, which has demonstrated its prowess by becoming the ubiquitous model for Internet hypermedia distribution. This architecture will address distributed agent communication; while collaboration

and intelligence will remain in the program realm and be tailored by programmers and designers, as intelligent agent systems continue to evolve.

The aim of this research is to examine ways of taking the agents designed in ways that follow FIPA ACTS agent's communication protocols and redesign those following RESTful principles. To illustrate this idea, there is need to design some multi-agent system following the most scalable system i.e. the web. This will require the designing of agents following REST principles and architecture.

The rest of this thesis is organised as follows: chapter 2 introduces the problem statement while chapter 3 covers literature review which includes review on multi-agent systems, review on Web Services architectures, and the approaches used while designing agent based systems. Chapter 4 presents the approach. The details of the architecture and implementation are presented in chapter 5. The proposed experiments are presented in chapter 6; experiments that provide the evaluation where issues such as finding out if the performances of the multi-agents built following RESTful principles are also examined. Finally, chapter 7 provides the conclusion and the contribution of this thesis. The future works are presented in Chapter 8. Finally, chapter 9 provides all list of references.

CHAPTER 2 PROBLEM STATEMENT

This research focuses on dealing with scalability of multi-agent systems. Lots of real life complex applications supported by multi-agent systems have been built using the FIPA principles, e.g. the AVEB (Audio Video Entertainment Broadcasting) system [5]. Since 1995, FIPA has been developing architectural structures that govern the design and communication on multi-agents systems [12]. This architecture is based on an abstract architecture as shown by Figure 2.1.

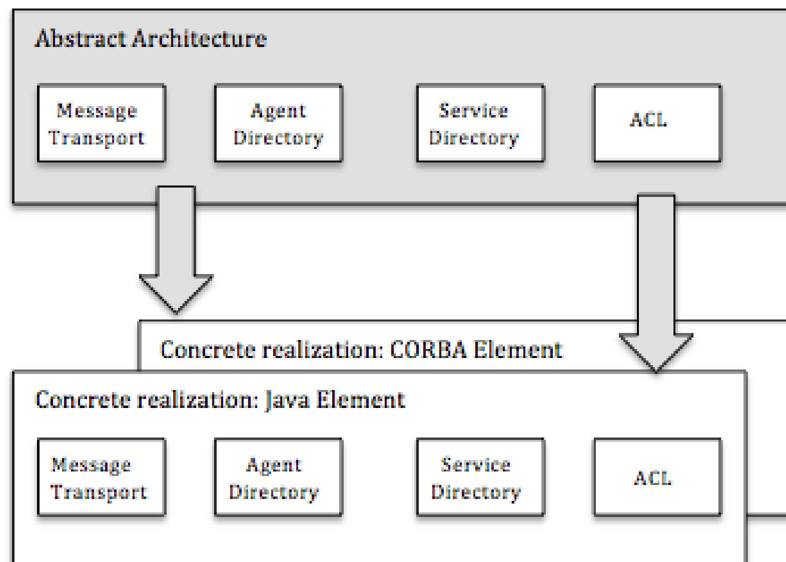

Figure 2.1: **FIPA Abstract Architecture Mapped to Various Concrete Realizations [12]**

Despite the great strides made in the area of multi-agent systems using the FIPA design approach, systems do not scale in a manner that they could in terms of the number of agents in a system. The key questions that need to be addressed are;

1. What are the reasons causing these agents not to scale?
2. Is it possible to follow the architecture of the web to create scalable agents?
3. Can agents be designed to use a RESTful Speech Act in their communication?

One of the reasons for the multi-agent systems not to scale is that these systems do not follow the principles of the web. The web is an example of a complete system that can be seen which has great scalability. The web is the largest, most scalable system used in most working environments. Web application can be implemented following Web Service design principles. Such web based design principles include Restful, and SOAP based approaches [34]. REST is a software architecture style for distributed hypermedia systems such as the Web. SOAP however is a protocol for exchanging structured data in the implementation of Web Services in a network.

The World Wide Web implements REST architecture hence is highly scalable. In order to achieve high scalability in multi-agents system, it is this research's assumption that it is important to implement them following REST architecture. The main question that this research wants to answer is how to build a framework that supports the development of RESTful agents. This research wants to have RESTful agents because RESTful services scale very well, they have clear notion of state, and they are relatively easy to manage. There is no work that has looked into support for RESTful agents, so this research wants to develop a framework to support RESTful agents.

The approach in this research is to investigate JADE [39] and implement its components using Erlang Language. JADE is an agent middle-ware framework used to develop multi-agent systems following the FIPA specifications with the goal of simplifying development at the same time ensuring standards are followed in the services and agents [39]. The main goal of this research is to build a RESTful agent framework and based on this framework investigate and answer the following questions:

1. What are the primary environmental factors that affect implementation of a scalable multi-agent system?

    a. From a scalability and stability perspective, how are environmental factors affected by sustained increases in demand on bounded resources?

2. Is it possible to follow the architecture of the web to create scalable multi-agent systems?
    a. Can agents be designed following the web design patterns?
    b. Can a web-centric agent design follow FIPA ACTS agent's communication protocols?

CHAPTER 3 LITERATURE REVIEW

3.1 Introduction

The literature reviewed in this chapter focuses on solutions, which address aspects of scalability and stability in the field of multi-agent systems. Although most previous work was tied to propagation of agents and distributed programs within bounded non-RESTful environments, their findings are still relevant and may offer valuable direction within bounded environments. Other documents, such as "The Role of Hypermedia in Distributed System Development", reinforce the direction this research has undertaken.

3.2 Factors affecting Multi Agents Systems

Lee et al. [29] discusses the factors affecting the Multi Agents System. Agents are the elements of a software program or an environment that interacts with another agent. These agents solve a global goal or an individual goal of a system. Agents in a system need to coordinate for allocating resources; they eliminate conflicts to achieve a common goal, to improve efficiency. Functional properties like co-ordination, knowledge, and rationality were focused in most Agent research. Non-functional properties like performance, scalability, and stability received less attention because these issues depend on underlying design and implementations of the systems. A term for naming multi-agent models called Multi-Agent System (MAS) is introduced, this term refers to a set of organized agents in either a hierarchical or meshes structure, see figure 3.1

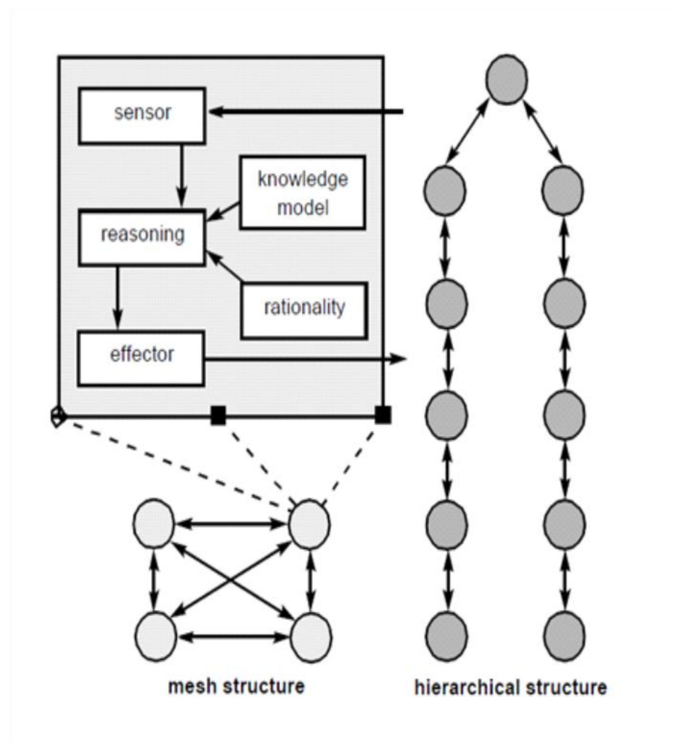

Figure 3.1. Multi-agent model in an hierarchical structure [12]

Due to the increased use of Multi Agent Systems in building real time distributed applications, the researchers are interested in performance, scalability, and stability properties. Lee et al. [29] states that coordination, agent's knowledge, and agent's rationality are the factors that affect the performance. Co-ordination generally involves computation costs and the communication overheads. For co-ordination to be successful, the agents involved in the computation tasks need to have the required knowledge. This knowledge is passed to agents through some iteration of task information passing. Figure 3.2 shows an example of iterative contracting.

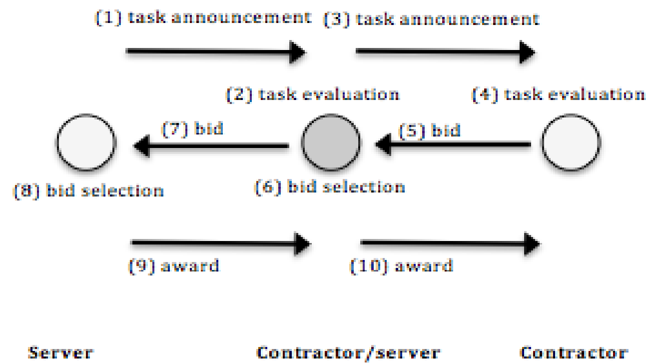

Figure 3.2. Steps involved in iterative contracting [12]

An agent's knowledge can be defined as storage costs and computation costs of the model. Agent's rationality model focuses on computation costs. Therefore the performance of a Multi Agent System can be measured by throughput, response time, concurrent tasks, computational time, and communication overhead.

The second important property is scalability. Scalability, in computing terms, is defined as: "the ability of a system to be used or produced in a range of capabilities: it is scalable across a range of systems" [32]. But, within the scope of this research, it refers to the deployment of agents and agent systems on the scale expected and required, as the number of agents and users increase. Where these agents are automatically created by user demands, as well as other intelligent programs, such as Internet based intelligent and autonomous programs, designed to query, search and or modify the environment, toward a specified goal, on behalf of said users or entities. The underlying scale of this environment, the World Wide Web, is literally the world; where millions of people want to "check-out" the latest offering by Apple, a movie or world event. We see this type of irruption regularly, in the situation where a popular video "goes viral", and is viewed by millions of people in a short period of time, as an example. Stability, of such a scalable Multi

Agent System, is the property of maintaining an equilibrium state, despite the sudden increase in popularity and demand. This type and level of stable scalability would not otherwise be achievable, without the RESTful nature of a HATEOAS deployment.

Lee et al. [29] tested one of their hypotheses, through a mesh agent structure experiment, to explain the importance of non-functional properties in distributed system. In the mesh model, the agents are connected to each other in a hierarchical structure. The Multi Agent System is given a task that requires all the agents to communicate to complete it. The task is divided into various subtasks by the server agent and each agent is allocated with a sub task and they in turn co-ordinate with each other.

In their paper, Gaku et al. [19] discusses an approached they have developed for performing simulation of objects whose mathematical modeling is not easy. Their approach is based on Massive Agent-Based Simulation. They suggest Agent based simulation is an efficient way of dealing with such objects. In addition, they also mention that there are frameworks that can enable agent-based simulation using large number of agents. Their intention was to develop a system that could be used in the PCs that are connected in a high performance network. They then test their system in an auction simulation environment where the numbers of agents were in the range of millions. Their evaluation show that the number of agents affect the final bid prices and the way these prices are distributed. Another result showed good scalability ratio.

3.3 Distributed Multi Agent Systems

From Lee et al. [29] descriptions of Multi Agent Systems are mostly used in practical applications in distributed applications. For such applications performance and scalability are important factors that most software engineering applications are facing in spite of their successful deployment. Deters [6] presents the scalability problem of the multi agent system by

executing large concurrent threads. Also, in Deters [6] a transparent JAVA/CORBA layer that can access the resources of different physical machines is presented. Scalability is an important issue in successful deployment of the software and the performance requirements can change with time, therefore the system should be able to scale up and down depending on requirements. An application could be scaled up by adding certain number of resources therefore increasing the performance. The application should be able to manage and coordinate between the agents when the resources are reduced. This is mostly used in resource constrained environments with mobile computing ability.

In Multi Agent Systems, there are two kinds of agents: reactive and proactive. A reactive agent will be executed if it receives a message. And thereby it takes processor time if it computes responses to the incoming messages. In large multi agent systems importance is given to move out the agents stored in the memory. The proactive agents do not need messages to start actions. But proactive agents need one or more threads execution thereby increasing concurrent threads. To improve the scalability of multi agents Deters [6] proposed usage of component distribution, component replication, and agent scheduling mechanisms. All the three approaches enhance the scalability of the systems but fall short to scale beyond the limitations of underlying physical machines.

Based on the principles discussed by Deters [6] two patterns were developed. One way is to improve the performance is by distributing the load on multiple cells. Another area is to increase the fault tolerance of the systems by replicating agents.

From the literature it is evident that scalability is an important aspect when developing practical software applications. Rana et al. [35] considers scalability from a performance engineering perspective based on Pertri nets used for modeling data flow mechanisms within distributed

networks. Scalability is important in multi agent communities to predict the performance of a system operating concurrently with a large group of agents from a small group of known agents. At present the emphasis is on software engineering approaches with little importance on performance modeling. Rana et al. [35] intends to incorporate performance model into design methodologies to enable automatic construction of large scale multi agent communities.

3.4 Scalable Hierarchical Coordination of Multi-Agent Resource Usage

In the context of bounded peer-owned resources, a distribution of functionally semi-autonomous sub-computations, inevitably introduces a degree of uncertainty [28], with respect to resource usage efficiency. Jamali and Zhao [27] practically address this issue by abstracting one vain of resource usage, through implementing a CPU usage hierarchical coordination scheduling model called CyberOrgs.

CyberOrgs create a market of CPU usage needs, as a resource, where semi-autonomous agents may buy and sell resources among peers using eCash, where the computational resources are defined by time and space. Sale of resources is hierarchically owned and contracted between parties, to be used within a certain period of time, CPU “ticks”.

The abstraction, and resulting limitations set by and controls used by CyberOrgs, separates concern of the function of agents' computing from their resource requirements; thereby eliminating the real possibility of the proliferation of runaway, erroneous or malicious agents, which can have exponential effects, specifically wherein a reactive management system of such inevitability is inadequate, as calculated by Jamali and Zhao [27].

In CyberOrgs, agents and Actors, represent computing elements which are responsible for processing independently and transact eCash with each other through messages. The life-cycle and resource needs of such objects, managed by CyberOrgs, uses primitives such as “Isolate”,

used to create a new CyberOrgs, Actors or facilitators/helpers etc.; and “Assimilate”, used to absorb unused resources from completed process owners; as well as “Migrate”, taking advantage of the extension of the Quantum formulation [28], and making use of distributed peer-owned resources .

In this paper, Jamali and Zhao proposed and tested a prototype implementation of CyberOrgs using Actor Foundry, a library of Java classes supporting Actor functionality. They also introduced programming constructs for implementing systems of CyberOrgs as well as described scheduling techniques for efficient distribution of processors resources. Their experiment showed that overhead used by CyberOrgs' scheduler was proportional to the native Java-compiler’s performance, despite the additional management overhead, and thereby acquired benefits, irrespective of the number of CyberOrgs and threads.

Potentially, a more comprehensive system, based on this work, including management of threads/memory as well as possibly compiler and firmware modifications, could produce a faster, more stable and scalable environment, within the context of the aforementioned bounded peer-owned resources.

3.5 Agents and web services (SOAP)

Multi-agent based systems can be developed following different techniques just like most web applications. In developing these systems, technologies that could be employed and used include REST, SOAP, WSDL, XML Schema just to name a few among many other WS-* specific technologies. The presence of so many technologies could be overwhelming when deciding what technological approach to follow while designing a multi-agent system.

According to Brennan [42], there exist two main ways of developing web services: either using SOAP approach or using REST approach. SOAP has been there for a while and has been used as

the default and as a standard-based approach. REST is a newer approach and it provides newer trends. Each approach has its own advantages and disadvantages.

SOAP was developed in 1998; it was designed to be platform independent, to be extensible, to integrate easily with other technologies, and to be compatible with older technologies. SOAP is used with WSDL and XML as the "standard" method of exchanging XML based messages. The need for SOAP to integrate with many other technologies, made SOAP seem heavy and complex.

3.6 REST

According to Brennan [42], the emergence of RESTful approach to designing web applications was more of reaction to the perceived heavy-based SOAP approach to systems design. REST emphases on simple point-to-point communication over the backbone of the web infrastructure; the HTTP protocol. REST was coined by Roy Fielding in his PhD thesis: Representative State Transfer (REST). REST utilizes four verbs; GET, POST, PUT, and DELETE from HTTP 1.1 protocol. Web services designed following the REST approach are simpler and more concise than those designed following the SOAP approach.

3.7 HATEOAS and RESTful design

Richardson [37] talks about developer's notion towards "hypermedia as the engine of application state". HATEOAS is the basis for web browsers. According to this principle, the application is like a model that changes its state time to time. The importance of using such a principle is to evolve new systems, loose coupling between clients and servers. But they fail to apply this principle to Web Services. The developers showed resistance to the Launchpad protocol that was developed by author with few other developers to show the world a RESTful design. Few

developers used their own interface to interact with the application. The concept "Hypermedia as the engine of application state" differentiates REST from client server systems. In a client-server system, there is a rapid change in application state due to the hypermedia links formed in each server response. These dynamic changes should be adapted by the client. But in REST, the client needs to know a single application URL to access it and all future actions are discovered dynamically from the hypermedia links of the resources returned by the URLs.

Hadley et al. [23] describes Roy Fielding architectural style REST. As mentioned by Fielding in his thesis, REST has four constraints. The most important one is HATEOAS "Hyperlinks as the Engine of Application State" which refers to the use of hyperlinks in resource representations as a way of navigating the state for an application. This paper proposes the use of action resources and applies it to the Jersey system. Action resources are sub-resources defined for the purpose of exposing workflow related operations on parent resources [34]. The action resources define the links a contract with the client that has potential to evolve depending on the applications state. Using action resources client-servers can be developed with different degrees of coupling and enables the servers to evolve independently.

3.8 REST and Distributed Systems

Parastatidis et al. [33] discuss the role of REST in developing distributed applications. REST is governed by HTTP which requires applications state, the business process that affect state, distributed data structures that hold it, the contracts, and protocols that establish interactions between the constituent parts of the system. The important part of REST is the use of hypermedia that enables developing robust, scalable, and adaptable systems by taking the advantages of the underlying web infrastructure.

An application protocol is a set of legal instructions to realize its behaviour. The state of the application itself is a snapshot of such application protocol. The protocol defines the interaction rules between the participants in the system. Therefore an applications state is a snapshot of the system at any instance in time. Thus HATEOAS means hypermedia drives systems to transform application states.

By using hypermedia, the business protocols are exposed on the web because of loose coupling, scalability, maintainability aspects conferred by the REST architecture.

Fernandez et al. [10] discuss the utilization of the REST architectural pattern to design software solutions or use parts of it to find solutions to the problems before the existence of World Wide Web. The World Wide Web was greatly successful in solving problems with simplicity, flexibility, and scalability. The basis for WWW is REST protocols which can be applied to other domains. Few researchers and practitioners tried to apply the principles of REST to solve problems before WWW and also compared with web services and SOA. But it is not compared with any other architectural styles, but to specific technologies and protocols. The author created a blog to clear the confusions of other practitioners and encourages them to use it properly. But they need new names to describe the names of the software architectures built on top of it, need alternate software architecture style. In this work Roy Fielding proposed the utilization of extended influence diagrams to visualize the structure and rational of an architectural style. The researchers can study easily reutilization of one architectural style or parts of it to design a new architectural style to address problems with different requirements.

Roy Fielding mentions that they needed a model of REST design rationale that could be manipulated to visualize changes. REST is an architectural style that has a set of constraints that can be added to an architectural style. The result of adding an architectural decision to an

architectural style is another architectural style. Requirements for such systems are to visualize and understand how each one of the architectural decisions of REST impacts the goals, visualizes the alternative approaches, and visualizes the changes caused in adding new properties. Some of the important insights that the author obtained after looking at the styles are simplicity, scalability, modifiability are the important features of REST. The goals of simplicity and scalability are to receive positive effects from REST. Reliability is partially affected in REST. Another important reason for applying REST to the problem domains is to interpret the software qualities used in REST. Some problems could be that the solution might be too verbose and is hard to reuse them in future. One way to solve this problem is to define architectural decisions which are used to model an architectural style.

Traditional distributed computing problems have been solved by breaking it down into chunks to handle it easily. Such solutions cannot be applied to problems where there are greater dependencies. Jacobi et al. [26] says that by portioning the algorithm instead of the data, this research can achieve general application of distributed computing. Partitioning the algorithm requires tight communication between the participants of the network. In the recent times distributed systems like BOINC platform, Google's Map Reduce have been used to solve several problems. Many of these partitioned the domain problem into smaller, more tractable chunks. Individual hosts are then processed to obtain partial results that are later merged to form a final result. This solution may not apply to all domains.

Selonen et al. [38] discusses the application of RESTful web services in a mixed reality content application at Nokia Research Center. Mixed reality refers to fusion of real and virtual worlds for creating environments in which physical and digital objects co exist. In this application there are geo-spacial relations that mean there is a special attention to searching and storing content with

geographical location and special arrangement of information. The author says that REST and Resource Oriented Architecture were chosen as the architectural style because they are used for storing, retrieving, and managing content. The client fetches building outlines, panoramas, and points of interest based on location. The application enables users to annotate particular buildings and other landmarks through touch and share with other users. The approach used to RESTify during the development is to use software development approaches for web services development in real life.

3.9 REST and SOAP Web Services

Pautasso et al. [34] compares REST and SOAP web services based on their architectural principles and guidelines. The two approaches differ in architectural decisions, number of decisions to be made, complexity, and the development and maintenance costs associated with each of them. Enterprise integration could be achieved with shared databases, remote procedure calls, message bus, and file transfers. Among these SOAP web services follow remote procedure calls and message integration style. These web services include SOAP, WSDL, WS-Security, WS-Addressing, WS-Reliable messaging, and the new solution to provide for remote procedure calls across the web i.e. REST. The popularity of the web services is due to its usage, simplicity in design, and publishing them. Pautasso et al. describes the comparison of web services and REST in a conceptual, technological, and decision stages. According to conceptual view, services are software components that can be accessed through a network accessible end point. The service consumer and service provider exchange messages to request and respond. Based on the technology view, SOAP is an XML based protocol used for exchange of information and the SOAP document represents an envelope with a header and body. The header information is used for routing purposes with security, reliability. The body consists of the actual message described

using XML so the SOAP engines at endpoints can marshall and unmarshall the content and route it to the actual implementation. The web services description language (WSDL) is an XML language for defining interfaces. The WSDL port contains multiple operations that are associated with some incoming and outgoing messages.

The strengths of SOAP and WSDL format have increased their adoption, making it possible to achieve interoperability between middleware systems. The same message in the same format can be transported across multiple middleware systems. WSDL provides a machine process able description with syntax and structure of the corresponding request and response message for a service. If the business needs change, the same service interface can be bound to different transport protocols and endpoints. A major weakness of web services is the interoperability of services.

REST, on the other hand, is an architectural style introduced to build large scale distributed hypermedia systems. REST is often used with HTTP because of its excellent scalability with HTTP1.0 and HTTP1.1. REST architecture identifies 4 important principles.

(i) A RESTful web service exposes its resources that interact with its clients. Resources are identified by URIs.

(ii) Resources are manipulated with a fixed set of four verbs. GET, PUT, POST, and DELETE.

(iii) Resources can be decoupled from their representations so that they can be accessed in any format (i.e HTML, XML, PDF, JPEG etc).

(iv) Every interaction is a stateless interaction.

The strengths of REST are its lightweight infrastructure and ability to build with minimum tools. The effort required to build a client for REST is very low and can be tested with normal web

browsers. With the use of REST it is possible to build and discover resources on the web without a centralized repository. On the operational side, it is known for its scalability for large number of clients with its stateless behavior. The confusion between hi-REST users is encouraging usage of all 4 verbs and lo-REST users using POST and GET to perform all transactions causes' confusion. For requests having large input data over 4KB the server will reject the malformed URIs.

The comparison between SOAP web services and REST are organized in to three levels. (1) Comparison of architectural principles: These principles determine how an architectural style addresses its requirements and designs goals. Protocol layering in the context of REST is seen as a medium to publish and access information. From SOAP web services perspective, the web is seen as a medium for transport of messages between the endpoints of published applications. In defining loosely coupled nature of the services RESTful services appear more loosely coupled than web services as each web service interface publishes a different set of operations. (2) Conceptual comparison: the complexity of interface design, REST appears to be simpler as it is completely dependent on the set of operations. With web services there is a trade-off between modularity, reusability and performance. (3) Technology comparison: web services use a single format for messages whereas REST does not have a format for representing resources. This can complicate the interoperability of REST. Web services lack the ability to express its services in URIs but REST has an advantage in expressing the information in URIs without the need for centralized registry.

The conclusion of the SOAP vs REST discussion is that, REST can be seen as a favorable solution for simple integration scenarios. From the comparison, two styles are similar with

respect to technology decisions. REST is better with respect to flexibility and control, but requires a lot of tools support and introduces dependency on vendors and open source projects.

In the fast changing world there is an increasing demand for dynamic and large scale systems to solve complex problems. Turner et al. [44] says there is a need for huge Multi Agent Systems composed of thousands of autonomous agents. These Multi Agent Systems are composed of multiple interacting agents that can be used to solve problems that are difficult. In the dynamic environments the number of agents in a system may change.

There is a need to be able to determine the organizational structure of the system (i.e) self-building, and able to change their structure with environment changes (i.e) adaptive. The primary reason in making the multi agent systems adaptive and self- building is to deal with a variable number of agents. In multi agents systems, agents should be allowed to build and maintain their own organizational structure that allows them to decide what tasks should be shared, delivered, or pursued individually. This enables the agents to know about their own internal efficiencies, goals, give tasks and information to other agents and to build sufficiently manageable small size systems that could scale. Turner et al. [44] describe these principles with an ecommerce example where scalability is measured, and presents the organizational structures in different forms, and talks about organizational adaptability.

3.10 REST vs SOAP web services design approaches

Table 3-2 provides a table showing the difference in the approaches used for designing and implementing web services. These findings are based on the information provided by Pautasso et al. [34] and Brennan Spies [42]. It is not easy to provide a fair comparison between the two approaches because they have very different philosophical approaches of accomplishing tasks.

Table 3-1 Comparing REST vs SOAP

REST	SOAP
Language and platform independent	Language and platform independent
REST has lightweight infrastructure	Is heavy when used with other WS*
Ability to build with minimum tools	Harder to develop, requires tools
Testing is really easy (use normal browsers)	Testing is harder
Supports scalability really well	Designed for distributed environments
Small learning curve, less reliance on tools	A lot of support from vendor tools exist
Employ the principles of the web	Very extensible
Tied to HTTP transport protocol	Transport protocol independent
Lack support for WS*(security, policy..)	Better support for WS* (policy, security..)

3.11 Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services

According to CAP theorem there are three main requirements to be considered while designing and using distributed systems, these are: Consistency, Availability, and Partition-Tolerance. Systems that use and incorporate agents could be considered as examples of distributed systems. Seth Gilbert and Nancy Lynch [21] in their paper mention that it is impossible for a web application to provide these three guarantees that are desirable and expected from the real-world: Consistency, Availability, and Partition-Tolerance.

The GenieDB White Paper [20] also examines Brewer's CAP Theorem for the builder and users of fault-tolerant database systems and shows how to overcome the restrictions of providing

databases that are consistent, available and partition-tolerant. The CAP theorem states that a distributed system can achieve any two of the three properties but not all three at the same time. Using the multi-agents context to define the CAP theorem terms, Consistency is the ability for all the agents to use the same data at the same time. Any operation executed on the data should be applied to all data at once. For example, the operations that update the system should appear as if they happened in an instant. Availability is the ability to guarantee availability of data to the agents whenever the any agent asks for it, for example, the functioning of the system should continue running as normal, should a server fail due to network failures. Partition tolerance is the ability for the system to work well even when some part of the system is broken down and is not working. For example, should the network randomly fail or the system is chopped in half, the system should continue running until things are back to normal and resume in the same consistency. The CAP theorem states that it is impossible to design distributed systems that provide services and fulfill all the three requirements of Consistency, Availability and Partition tolerance satisfactory. It is only possible to pick two of the requirements.

According to GenieDB White Paper [20], earlier distributed databases chose to build duplicated systems that used quorum algorithms instead of implementing partition tolerance. If the network partition happened, the smaller servers became read-only and all writes rejected and occur in one group of connected servers. A system is no longer “available” if there is no partition larger than half of the system, because it will not be able to handle the writes. Current databases have chosen to not implement consistency, where servers are all in contact with each other sharing idempotent data, such that if one was inaccessible and reappeared, all the missed data is sent to it. According to White Paper, updates should be idempotent to provide excellent tolerance of both server and network failures.

In their work, Gilbert and Lynch mention and discuss the availability of significant research in designing ACID databases that are necessary for the new framework for building web services. Interactions are expected to behave in a transitional matter, Atomic, Consistent, Isolated, and Durable. Systems designed in such a manner are good for billing and commercial transactions. Web Services are expected to be highly available;" every request should succeed and receive a response". Services go down when most needed and significant real-world problems arise, e.g. E-Trade website goes down. The goal for most web services is to be as available as the network they run. In a highly distributed network, fault-tolerance should be provided, should the nodes crash or a link fails, service should still perform as expected, e.g. ability to survive a network partitioning into multiple components.

Most web applications expect atomic or linearizable consistency nowadays. There must be total alignment of operations so that it looks like it happened in an instant. It is the easiest model for users to understand and for those attempting to design a client application that uses the distributed service.

"For a distributed system to be continuously available, every request received by a non-failing node must result in a response" [20]. Any algorithm used must eventually terminate, showing weak definition of availability as it puts no bound on how long the algorithm will run before it terminates, unless when qualified by need for partial tolerance.

To recreate partition tolerance, the network will be allowed to lose many messages from one node to another. When partitioned all messages sent from nodes in one component to another component are lost. Atomicity requirement implies every response will be atomic and messages sent might not be delivered. Availability implies that every node receiving a request must respond even though the arbitrary messages may be lost.

In order to prove the impossibility result, Gilbert and Lynch used the asynchronous network model as formalized earlier by Lynch. Using this model, they tested four theorems.

In their conclusion, Gilbert and Lynch manage to show that it is impossible to provide atomic consistent data when there are partitions in the network; however, it is possible to achieve any two of the three properties consistency, availability and partition tolerance. In Asynchronous model, it is impossible to provide consistent data when there are no clocks available. The impossibility test is fairly strong. In Partially Synchronous models, it is possible to achieve a practical compromise between consistency and availability.

In conclusion, the CAP theorem allows us to realize that, we cannot have total consistency in designing agents that fit perfectly to a given scenario task. It is important to realize in a complex large system, it is not that there is a hard state that is shared by all, it not that when one agent knows something all the agents knows about it, the reality is that, it always takes time to propagate data to the various agents in a complex system. Using an agent-based system is always a good way to accomplished such activities because of the possibility of automating the functionalities of the agents.

The CAP theorem also shows us that we have to have some mechanisms in large scale distributed systems to deal with the absences of a hard state because we have a soft state. We therefore need to have events that could be informed of messages that could be utilized in complex systems. Agents are very useful in doing this task.

Agents have been widely used in combination with SOAP based web services. However, there is no work that has combined multi-agent systems with REST. We are therefore looking at designing a framework that combines REST and multi-agents systems. The advantage of using REST is that, it provides a framework that scales well following the web infrastructure.

3.12 Conclusion

Table 3-2. Literature review summary

Agents Scalability Management and Load Distribution	[6] Deters [29] Lee, L., Nwana, H. S., Ndumu, D. T., & De Wilde, P. [35] Rana, O. F., & Stout, K. [44] Turner, P. J., & Jennings, N. R. [31] Nwana, H.S. [2] http://www.agentbuilder.com/Documentation/whyAgents.html [17] Stan Franklin and Art Graesser [5] P. Charlton R. Cattoni, A. Potrich and E. Mamdani [39] Nadeem Jamali and Xinghui Zhao
REST REST vs. SOAP HATEOAS and Heterogeneous Environments	[10] Fernandez, F., & Navón, J. [23] Hadley, M., Pericas-Geertsen, S., & Sandoz, P. [26] Jacobi, I., & Radul, A. [33] Parastatidis, S., Webber, J., Silveira, G., & Robinson, I. S. [34] Pautasso, C., Zimmermann, O., & Leymann, F. [37] Richardson, L. [38] Selonen, P., Belimpasakis, P., & You, Y. [42] Brennan Spies
CAP Theorem	[20] White Paper Beating the CAP Theorem [21] S. Gilbert N. Lynch
Standards References	[16] http://www.fipa.org/specs/fipa00037/SC00037J.html [14] http://www.fipa.org/repository/aclspecs.html [11] http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm [12] FIPA - Foundation for Intelligent Physical Agents

3.12.1 Key Findings

Current implementations and scalability

From all the submitted works and examples, that use traditional methodologies, i.e. SOAP, where agent numbers need to scale and accommodate millions of simultaneous users; the number of thread, agents, memory and processing requirements would eventually overwhelm any system, unless the greatest majority of agents are distributed on a per-user basses; very much like current successful RESTful implementations, such as HATEOAS.

Agents have been widely used in combination with SOAP based web services. However, there is no work that has combined multi-agent systems with the RESTful architecture style. We are therefore looking at designing a framework that combines REST and multi-agents systems. The advantage of using REST boasts the proven provision of a framework that scales well, following the web infrastructure.

Functionality and Scalability

Designing agents that are compliant with and following web infrastructure, require a delineation of duties and functions of such agents. The static number of agents is required, at the server-side. The functions of these agents include creating client-side agents, in response to client demands. These agents, or groups of agents, must be designed stateless and tailored to client requests and specific queries. The functions also include the management, evolution and direction (direction toward required resources, i.e. URIs, in the fulfillment of their targeted duties) of the client-side agents and server-side environments.

On the server-side, management agents' numbers should be stable by design, notwithstanding the number of end users, and methods proven effective in the above literature review, such as resource management using eCash, by Jamali and Zhao, may be sufficient within the server-side bounded environment, while mitigating and "beating" [20] CAP limitations. This web compliant distribution approach, should allow for client-side agents to scale with demand, without increasing server-side resource requirements.

One of the problems in dealing with simulation in large-scale multi-agent applications with continuously changing communicational patterns is distributing the agents among the different system nodes extended to take advantage of distributed computing. Load balancing algorithm

can be used to assign tasks on distributed computers nodes. This method has limitations as it requires computation and communicational cost to be known in advance. Furthermore, when there is continuous communicational pattern, it is difficult to estimate the communicational cost. In their paper, Myeong-Wuk and Agha [30] propose a solution of using two adaptive agent allocation methods. The first method aims at reducing the agent communication cost. The second method attempts to prevent overloaded nodes from affecting the performance of the system. They tested their methodologies on a multi-agent framework. This framework allowed for the monitoring of workload and the communication pattern on each node. This made it possible to periodically change the location of the agents based on their communicational patterns and node status on overload.

In their paper, Gaku at el. [42] discusses an approached they have developed for performing simulation of objects whose mathematical modeling is not easy. Their approach is based on Massive Agent-Based Simulation. They suggest Agent based simulation is an efficient way of dealing with such objects. In addition, they also mention that there are frameworks that can enable agent-based simulation using large number of agents. Their intention was to develop a system that could be used in the PCs that are connected in a high performance network. They then tested their system in an auction simulation environment where the numbers of agents were in the range of millions. Their evaluation show that the number of agents affect the final bid prices and the way these prices are distributed. Another result showed good scalability ratio.

Zhuge Hai [24] suggests that the modern society requires an interconnected environment that goes beyond the scope of automated machine intelligence. The reasons for this argument are: personal computers have evolved to networks, then to human-computer environments resulting in a major concern of the society. The Agent Grid Environment is not only a scalable but also a

sustainable and intelligent networking environment. It is possible to have harmonious co-existence between humans, agents, machines, and nature in such an environment. The environment gathers useful data from the society and the nature, the data should fulfill certain requirements that will allow the transformation of these data into resources. The resources will be processed and the result will be used to affect the same environment. Based on the set rules of the environment, the elements found within the environment can intelligently cooperate to perform tasks, generate knowledge and solve problems. Zhuge Hai, in this paper [24] explores different rules of flow to determine the flow of information, knowledge, and service flow to support the cooperation of the elements. Peer-to-Peer that has been used to support resource sharing among massive agents could be used in such an environment.

3.12.2 RESTful distributed agent communication

In order for distributed web compliant agents to communicate, across Internet expanses, there needs to be an abstraction of FIPA ACTS protocols, adhering to the "common-interface" specifications set by Fielding [11]. Such an abstraction facilitates the communication and evolution of either locale, thereby allowing the locales to independently scale.

3.12.3 Development of runtime environment

RESTful development is traditionally done in Java and JavaScript etc., supported by the ubiquitous proliferation of the Java Runtime Environment (JRE). Unfortunately, Java and other high-level mutable programming languages, suffer from scalability issues related to shared-memory and distributed thread management [1].

For this research, we have chosen Erlang, specifically because it is defined around the use of immutable variables, which directly addresses the issue of shared memory errors; furthermore, it is inherently designed for environments requiring processes-based-concurrency, distributed-computing, asynchronous-messaging, reliability and fault-tolerance. As far as the proliferation of a supportive runtime environment, there are currently several functional pursuits toward that end for Erlang, including continued development of BEAM (beam.lib), JErlang (SourceForge [41]) and ErJang (now at GitHub [22]).

CHAPTER 4 THE PROPOSED APPROACH

There are two interesting aspects of the approach followed in this research. First, there is the development of the agents based systems using Erlang. The details of the architecture and the implementation of this agent based system are described in Chapter 5. In this Chapter, the focus is more on describing the general approach on how a JADE like agent based system is implemented following specific features supported by Erlang and describing the architecture of the agent system. The key features parallel the three issues (key questions), as described in chapter 2, are concerned with:

- Web-centric, i.e. RESTful, and therefore more scalable architectural style.
- A functional programming environment, to deal with issues of concurrency, asynchronous-messaging and fault-tolerance.
- The RESTful abstraction of the Agent Communication Language (ACL).

4.1 The Programming Environment

The first characteristic of the system is that the agents themselves must be programmed in a way that allows for large scalability. A question then arises, “How can agents be made more scalable?” The first step is to choose a platform where the agents themselves are programmed in a lightweight manner. The use of a functional language allows us to create very lightweight agents as entities. The second step is to design the agents in a manner that is very robust and scalable. Since the entities as the agents can scale, there is need for fault tolerance that allows the restart of entities that have stopped working.

It is important to use an environment that supports the creation of very large numbers of concurrently executing entities. These characteristics require the use of a concurrency oriented language in implementing such entities. Erlang is a functional and concurrency oriented programming language that provides and supports the qualities this research suggests for creating agents.

The second characteristic of the system is that you cannot have agents in a very high level programming language environment as an object. An agent cannot be an object; because objects involve or need memory sharing, hence there will be fundamental scalability issues to deal with. The agents cannot represent locally managed threads, because threads are tied too close to implementation.

When designing agents for systems, the accepted approach is Agent Communication Language (FIPA ACT). ACT is a "message exchange interaction protocols, Speech Act theory-based communicative acts and content language representations" [14]. FIPA ACT [16] contains several communicative acts. The diagram below shows two examples of these acts that enable communication between two agents (Say agent A and agent B).

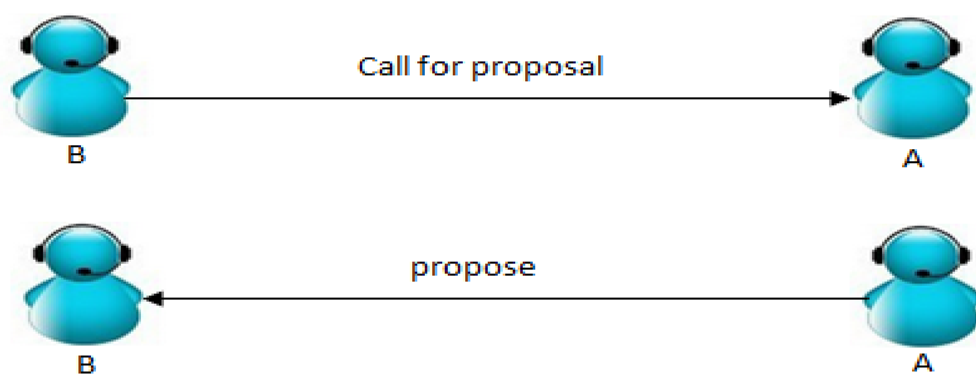

Figure 4.1. Cooperation between two Agents

As a basis for this work, this research explores and uses the Actor Model [18] based approaches on top of which the system will be built. The actor based approach is a message based environment. An actor is an entity that receives messages and responds to messages. The actor knows how to deal with the incoming and outgoing of messages.

4.2 RESTful Architectural Style

Designing agents using FIPA architecture, means encouraging scalability issues within the agent systems. A way to overcome scalability issues for agents is to implement a RESTful design of the agents. REST is an architectural style for designing distributed and scalable systems [11]. The architecture uses HTTP for communication Pautasso et al. [34]. The main concern to be handled in this stage of the architecture is not to create agents (processes) that use HTTP but to create agents (processes) that receive tuples (messages) and send tuples.

To design these agents following REST principles means following the RESTful constraints and using HTTP methods for data manipulation. Messages (resources) are manipulated with a fixed set of four verbs. GET, PUT, POST, and DELETE. Pautasso et al. [34] another verb is PATCH. It also means, agents maintain statelessness, and their interaction does not assume they have knowledge about one another. The agents, when they communicate, have to be stateless. This means all the transitive information has to be maintained by the sender and then sent to the receiver. The receiver cannot be expected to remember information about senders, or the nature of the larger task at-hand. This is because they simply respond to the specific task, as defined by the agent making the request.

When agents engage in talking to each other, objects (i.e. new resources) are created. These created resources hold the state of that communication because they are obligated to not limit the

agent to one to one communication; the agent should still be able to respond to other requests by creating new resources, independent of its current obligations.

The expected contribution of this research is toward agents being designed following RESTful constraints; similarly, the system itself must be designed following RESTful principles. This research proposes exploring and extending the Actor Model approach further and have agents communicating using RESTful Speech Acts. This means for every agent, there is GET, POST, PUT, PATCH, and DELETE requests, as required. The agents is made to follow very clear operations with very clear semantics. This means when there is a message sent e.g. a READ request; it has to be sent as a GET method. If it is a CHANGE request message, PUT or PATCH is used, as required. If it is a CREATE request message for a new entity, POST is used. What is novel about this approach is that most people have not radically looked into working with more than one method. They typically use one method which usually is accepted or process message.

4.3 ACL Abstraction

In agent based systems, communication between agents follows a Communicative Act protocol called the FIPA Speech Act or performatives. In most of the FIPA Speech Acts, agents' communication will be assertive, directive, expressive or declarative. This means this research will define a message transport component, integral to the system. To test the system and the required FIPA Speech Act actions (mainly GET & POST), messages will be sent to the server (Gen Server defined using Erlang). Second level tests will dilate the performatives as shown in Table 4-1.; these commands will be relayed as HTTP requests. The message layer (message API), will be given instructions as process methods, e.g. process read message, process create message, process message, and process delete message. The agents themselves do not need to know which of the HTTP methods they have to call; the message API abstracts the method based

on FIPA Speech Acts definitions. As such, this research will define the separate methods for GET, POST, PUT and DELETE, among other FIPA Speech Act methods and operations, the agents will be undertaking.

Table 4-1. FIPA Communicative Act Specifications and their implemented in RESTful approach.

FIPA ACT	Meaning	Corresponding RESTful Verb
Accept-proposal	The action of accepting a previously submitted proposes to perform an action.	POST/PUT/PATCH
Agree	The action of agreeing to perform a requested action made by another agent. Agent will carry it out.	POST/PUT(update)
Cancel	Agent wants to cancel a previous request.	DELETE/PATCH
Cfp	Agent issues a call for proposals. It contains the actions to be carried out and any other terms of the agreement.	GET/POST
Confirm	The sender confirms to the receiver the truth of the content. The sender initially believed that the receiver was unsure about it.	POST
Disconfirm	The sender confirms to the receiver the falsity of the content.	POST
Failure	Tell the other agent that a previously requested action failed.	PATCH then POST
Inform	Tell another agent something. The sender must believe in the truth of the statement. Most used performative.	POST/PATCH
Inform-if	Used as content of request to ask another agent to tell us is a statement is true or false.	POST
Inform-ref	Like inform-if but asks for the value of the expression.	GET/POST
Not-understood	Sent when the agent did not understand the message.	GET/POST
Propagate	Asks another agent so forward this same propagate message to others.	GET/POST
Propose	Used as a response to a cfp. Agent proposes a deal.	POST/PATCH
Proxy	The sender wants the receiver to select target agents denoted by a given description and to send an embedded message to them.	POST
Query-if	The action of asking another agent whether or not a given proposition is true.	POST

Query-ref	The action of asking another agent for the object referred to by a referential expression.	POST
Refuse	The action of refusing to perform a given action, and explaining the reason for the refusal.	PATCH
Reject-proposal	The action of rejecting a proposal to perform some action during a negotiation.	PATCH
Request	The sender requests the receiver to perform some action. Usually to request the receiver to perform another communicative act.	GET
Request-when	The sender wants the receiver to perform some action when some given proposition becomes true.	POST
Request-whenever	The sender wants the receiver to perform some action as soon as some proposition becomes true and thereafter each time the proposition becomes true again.	POST
Subscribe	The act of requesting a persistent intention to notify the sender of the value of a reference, and to notify again whenever the object identified by the reference changes.	POST/PATCH

Here is a description of an example showing the call for proposal: to communicate, an agent, say agent A, creates a proposal and sends it to the environment where it will be accessible to all targeted agents, see Figure 4.2. Agent A can request and get detailed information about each agent in that environment, then send each desired agent a request and point it to the URI of the proposal resource. Each of the agents communicating with agent A would either create response to the URL or not. These bits represent entities (resources).

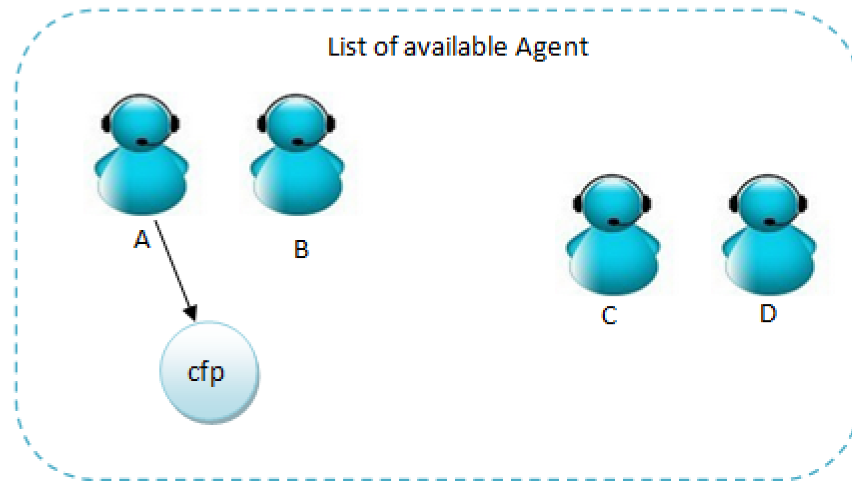

Figure 4.2. Agents in one environment

To develop this, there is need for at least one routing component, or service bus, so when agent A wants to communicate with agent B, agent A first sends a special search request (see Figure 3) to the service BUS, to determine if agent B is in the environment and is available to receive the call for proposal (cfp) messages.

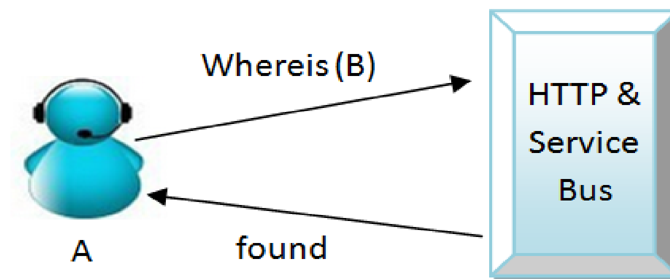

Figure 4.3. Finding Agent

To find out if agent B exists, the function "Whereis(B)" is called, and if there is no response to agent A's search request, then agent B must be outside that environment and another action may be required to achieve the desired goal. HTTP provides a conduit for agents (which could be implemented as Erlang nodes) to exploit the flexibility of REST; these agents could be designed to make best use of communication executed concurrently or sequentially.

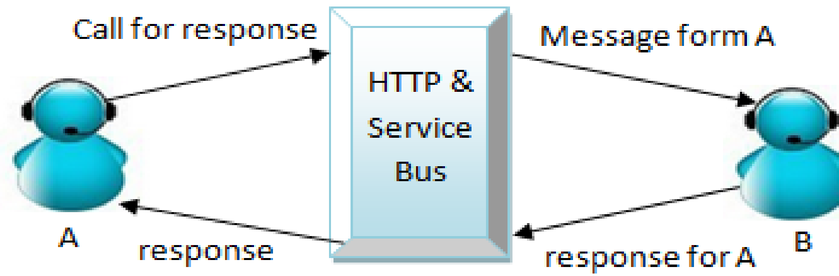

Figure 4.4. Agent communication via Service BUS

If agent B is found (see Figure 4.3), then agent A sends a message to agent B via the Service BUS (see Figure 4.4). In Figure 4.4 above, there is an HTTP module; it accepts the HTTP request which is converted into messages which are then sent to individual agents, in this case agent B. Using the same example as in FIPA here are two agents: agent A and agent B. When agent A wants to talk to agent B, it will route its message through the HTTP or a service BUS. Agent A sends a message to the service BUS, the service BUS, because it knows of the presence of all agents in its area, sends the message to agent B. The message passing will be done through the HTTP from agent A to agent B, vice versa, and between any other agents.

In the above example, creating a request for a proposal will require creating a RESTful proposal. This proposal can be put out as a RESTful resource which can be cached. This caching allows the system to have some advantages which could be applied to a proposal, a request or a bid that can be easily accessible to agents. The above communication scenario between agent A and agent B can be created in a RESTful manner following the next steps:

1. Agent A creates a call for proposal. This step includes the following;

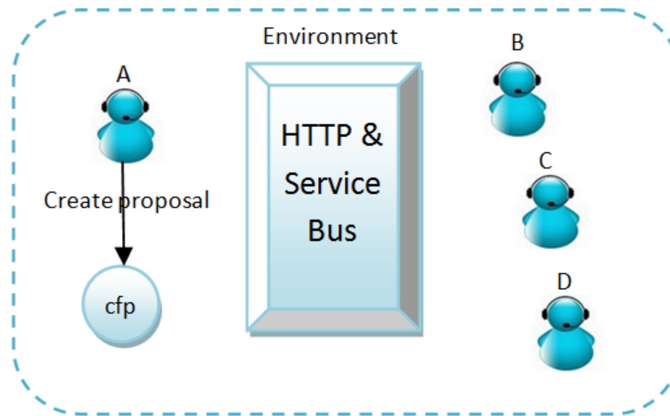

Figure 4.5. Agent creates a call for proposal

a. Create the proposal using POST
b. Inform other agents of the presences of a proposal using Service BUS using PATCH. It is not known if all agents will receive the information about the proposal.

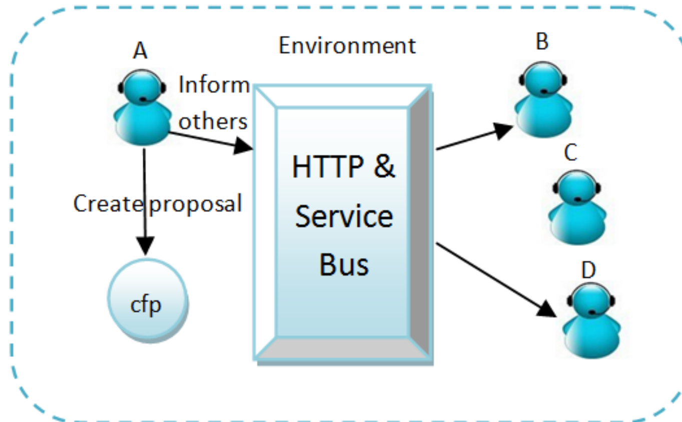

Figure 4.6. Agent informing other agents

2. The call for proposal is taken or represented as an entity (resource).
3. Agent A could be informed whether the proposal has been accepted by either of the following:
   a. Agent A monitors the proposal using the GET request routine.

b. The proposal has knowledge of a call back routine that when it receives something it will inform agent A of the change using the PATCH verb.

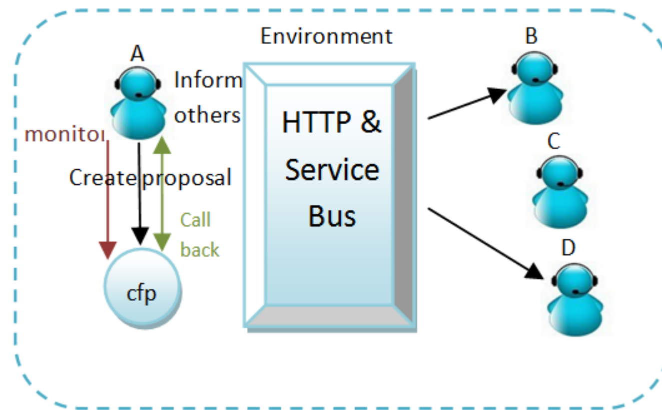

Figure 4.7. Agent monitors others and has call back

4. Agent B then accepts the proposal by first going to the proposal and POSTing 'I accept this proposal'. There is a chance other agents could have either accepted or rejected this proposal.

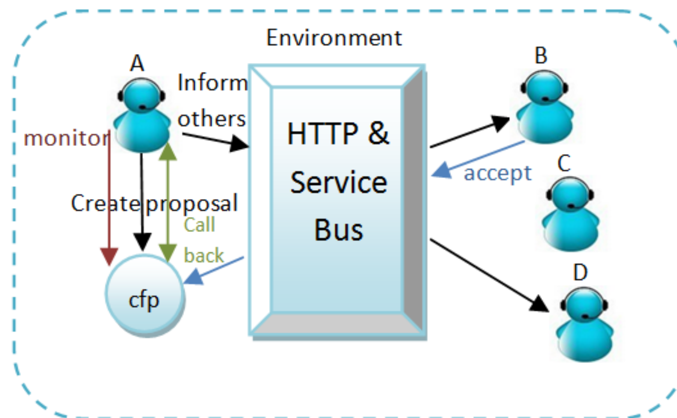

Figure 4.8. Agent C accepts agent A's proposal

Implementing separate methods is important because, when there is an agent system which is distributed over the web and the HTTP request method is used e.g. the GET method to enable communication between two or more agents and there is a cache in between, the cache allows us to harvest the advantages of caching. There is recognition that in this approach first, the agents

will have the four REST commands (the GET, POST, PUT, and DELETE) and the headers implemented. In Erlang, a generic server process as called gen_server is used for implementing behaviour of the agents [7]. Therefore, the gen_server will be changed so that it can act on each of the HTTP commands. The message API always examines the messages it receives to determine which of the HTTP commands is being requested and should be executed. Second, this will require putting HTTP layer around the agents because there is no longer need to directly use the FIPA Speech Acts itself, as the response/request will be made into a GET message, a PUT, a DELETE, a POST message or a HEAD message. When two agents are communicating, the infrastructure of the web can be used for better scalability of the agents. Third, not only the agents can be RESTful, but also their interactions can be made RESTful. For example, a request for proposal can be done by creating a proposal using the CREATE command; then it is made known to all the agents using the PUT or the PATCH method that there is a new proposal. Alternatively, a request for proposal can be created as an entity (resource), which can be sent as a URL, as a request for PID method which is a POST message to an agent.

4.4 Conclusion

In conclusion we have three elements highlighted in this chapter as contribution to this research.

This work will first examine what happens if we implement a JADE like system in an environment that is optimised for concurrency. To accomplish that, we will use Erlang, as we want to use a lightweight implementation approach, as well as dealing with other scalability issues, such as asynchronous-messaging and fault-tolerance.

Secondly, development of agents must follow RESTful principles; this will allow the system to scale. As the number of clients increases, the proportional increase in the number of agents will

be distributed in a web-centric manner, which addresses the known issues of increased resource needs as described in chapter three.

Last but not least, concerning the RESTful abstraction of the ACL, these operations must have clear read/write semantics. The agents will be wrapped in HTTP protocol, so that we can use the infrastructure of the web, e.g. caching, and the ability to scale easily to support the system's expansion; furthermore, communication between the agents will be made more web-compliant. This means our agent based system consists of autonomous resources.

CHAPTER 5 THE IMPLEMENTATION

This chapter is dedicated to the description of the architecture and experimental demonstration of implementation and justification for the use of Erlang, in solutions involving high numbers of threads requiring proportionally high numbers of message exchange, between said processes. In the sections to follow, you will see how Erlang is used to create examples of agent following FIPA Abstract Architecture, distribution, message communication (through the Agent Message System - AMS) and the implementation of an "HTTP & Service Bus", that will relay agent proposals and responses in a RESTful manner, as per Table 4-1, in chapter 4. It is interesting to note that the term "Light-Weight" is used to refer to many things in the computing world; but when comparing thread/ process heap and stack memory allocation, between Erlang and other popular programming languages and environments, the meaning becomes literal where values such as 309 words/process, in Erlang, are compared to Java's minimal allocation of 512K/process, (SICS testing [39]). This is over and above all other benefits of Erlang, in this type of application, including:

- dynamically allocated stacks;
- concurrency;
- event-driven asynchronous messaging;
- fault-tolerance;
- Hot-code-swapping (truly a requirement of RESTful distributed scalability).

5.1 The Architecture

The FIPA architecture defines six core components for implementing agent based systems, namely; Agent Communication component, Agent Management component, Agent Message Transport component, ACL Representation component, Envelope Representation component,

and Transport Protocol component. The architecture presented in this research is based on this FIPA Abstract Architecture.

FIPA's created agent standards is meant to encourage inter-operable between agent applications and agent systems; this process also included a specifying agent infrastructure. Agent communication language, agent services, and supporting management are some of the features that were included in the infrastructure standards. The following features are useful in supporting agent communication, message transfer, and message representation and authentication mechanism. The need to deploy agents in different environments lead to the creation of an architecture that could meet many commonly used mechanisms such as various message transports, directory services and other commonly.

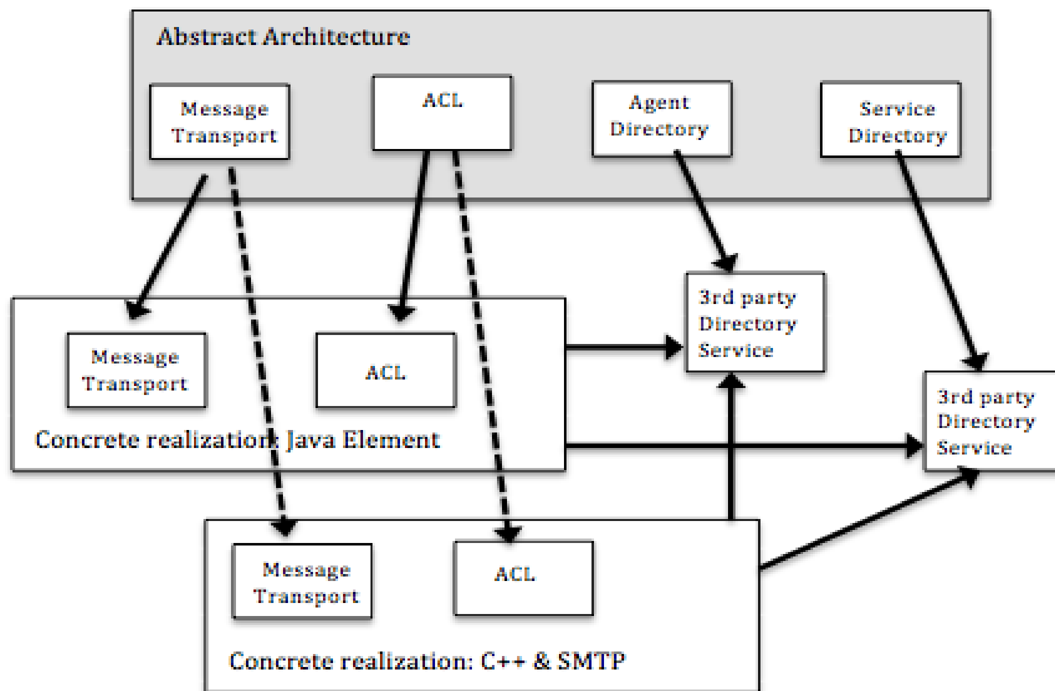

Figure 5.1. FIPA Abstract Architecture [13]

The above architecture is based on FIPA Abstract Architecture. The purpose of this architecture is to promote understanding, reusability and interoperability of the components that are implemented in this thesis system. This FIPA Abstract Architecture allows the creation of multiple concrete realizations of agents; it also allows agents to interoperate including transformation of transport, encoding and integrating with environmental elements. FIPA Abstract Architecture aims at creating semantically meaningful message exchange between agents which may be using different content languages and messaging transports. These needs require that the architecture include a model of services and discovery of services available to agents, message transport interoperability, support of different forms of ACL representations, support different content languages, and support for different of directory services representations.

It is important to note that the architecture forms the basis for the development of concrete architectural specifications. These specifications describe the precise details of how to create an agent system. The architecture must have ways for agent registration and agent discovery and inter-agent message transfer. The designer of the concrete architecture has options to choose from when developing an agent system based on the abstract elements. To simplify the programmatic view of the system, one needs to follow an architecture that has one encoding for messages, or only one transport protocol. The FIPA abstract does not however prevent adding new elements to the agent system.

The tract level of the FIPA Abstract Architecture defines how two agents can register themselves a mechanism that help them in locating and communicating with each other via exchanging messages; this is done by establishing a relationship of a set of architecture elements.

5.1.1 Agent communication

The Agent Communication component provides a way in which agents can exchange and pass information. Agents use the agent communication language to communicate by exchanging messages which are encoded and represented as speech acts.

The Agent Communication components contain agent directory services and message transport services that provide communication support services for the agents; these maybe implemented either as agents or as software that is accessed via method calling. It is better to implement services as a method because implementing them as an agent creates opportunity for errors in discovering and communications between agents. The agent is provided with a service identity to start; a service identity will have enough information to either describe all of the services available within the environment. The message structure, message representation and message transport are aspects that enable the communication between agents.

In the architecture presented in this research, the Agent Communication component is defined and implemented as an Erlang module. The behaviour of this component is explained in section 5.1.7

5.1.2 Service Directory

The service directory component also known as agent directory service is used to control the operations of the agents by allowing them to be registered. The agent directory service main role is to provide a location where agents register their descriptions which enables other agents to search and find when they want. Therefore an agent directory entry contains descriptive attributes associated with the agent. Agent management might include agent registry and its discovery.

Registering an Agent

An agent that wishes to advertise itself to some service first binds it to one or more transports offered by the agent message transport component; it then advertises itself by creating an agent directory entry which includes agent-name, agent locator and other attributes that describe the services in offer.

Discovering an Agent

Agent directory service is used by other agents to locate other agents with which to communicate; the searching process involves this searching for an agent directory entry that includes the key-value-pair that is relevant to what the agent wants.

In the architecture presented in this research, the Agent Management component is defined and implemented as an Erlang module. The behaviour of this component is explained in section 5.1.10

5.1.3 ACL Representation

The ACL Representation component's main duty is to provide consistent ways by which agents and services can discover services offered by other agents. Agents can search the ACL Representation to find services that meet their requirements. Attribute entries in an ACL Representation include service description consisting of a service name, service type, a service locator and a set of optional service attributes.

In the architecture presented in this research, the ACL Representation component is defined and implemented as an Erlang module. The behaviour of this component is explained in section 5.1.11

5.1.4 Envelope Representation

The Envelope Representation component is used to put the message into a structure. Written in an agent communication language, the structure of messages in agent based systems contains key-value attributes. The messages contain the one sender and zero or more receivers' names as shown in figure 5.2.

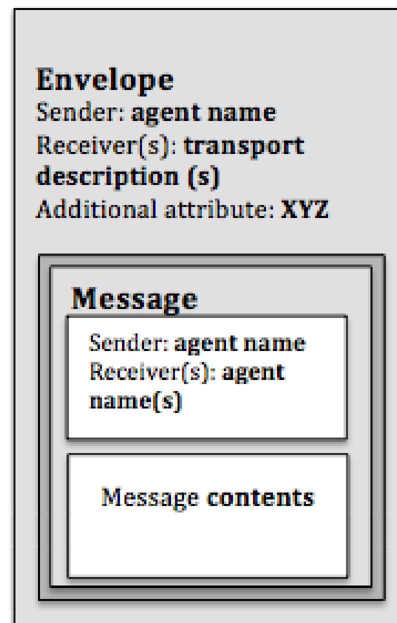

Figure 5.2. A Message Represented as an Envelope [17]

In the architecture presented in this research, the Envelope Representation component is defined and implemented as an Erlang module. The behaviour of this component is explained in section 5.1.8

5.1.5 Message Transport

The Message Transport component helps in carrying messages between agents. When sending a message agents encode it into a payload, and include in a transport message. The transport

message is the payload plus the envelope which include sender and receiver transport description which contains information of how to send the message as shown in Figure 5.3.

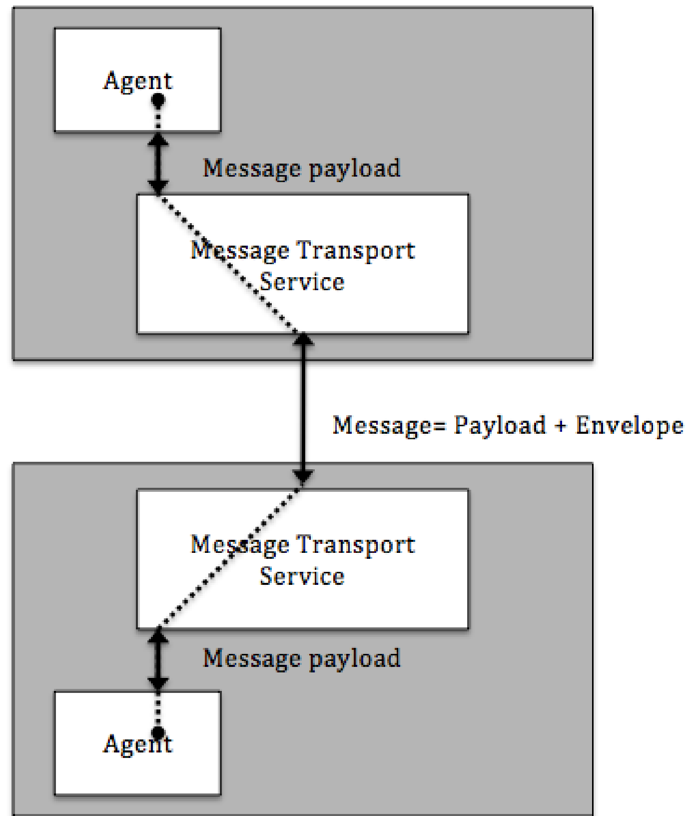

Figure 5.3. Message Transport Reference Model [15]

In the architecture presented in this research, the Agent Message Transport component is defined and implemented as an Erlang module. The behaviour of this component is explained in section 5.1.9

5.1.6 Transport Protocol

The Transport Protocol component defines the set of rules that should be followed in the transfer of messages using the architecture. In the architecture presented in this research, the Transport Protocol component is defined and implemented as an Erlang module. Erlang module is a small

library that is implemented in Erlang as module that have massaging services. The behaviour of this component is explained in section 5.1.9

5.2 Implementation Code Samples & Examples

To help understand the implementation process of a multi-agent system, an example is used. The example shows how agents can communicate with each other. The communication involves the sending and receiving of messages between agents. To explain how the send message works, the example below shows how an agent who receives a message and performs a side effect of the message is described next. The description is focusing on the logic that handles the messaging. The following is a ping pong agent example where two agents send messages to each other.

```
% description to ping_pong
% module attribute defines a certain property of a module
-module(ping_pong).

%%export tells Erlang which functions in this module to export
-export([start/0, ping/1, pong/0]).
ping(0) ->
%% The atom finished is sent to "pong"
    pong ! finished,
    io:format("ping finished~n", []);
ping(N) ->
     %%self() returns the pid of the agent which executes
self(),
     %%in this case the pid of "ping".
    pong ! {ping, self()},
    %%the receive allows agents to wait for messages from other
agents
     receive
        pong ->
           %% print that out
              io:format("Ping received pong~n", [])
    end,
     %%call the ping function again, (N_!)N-1 causes the first
argument to be decremented until it becomes 0 .
    ping(N - 1).
pong() ->
   %%the receive allows agents to wait for messages from other
agents
```

```
      receive
        %% if the agent finished
           finished ->
                %% print that out
             io:format("Pong finished~n", []);
         %% and if he recieves this format
           {ping, Ping_PID} ->
                %% do the following which is printing.

             io:format("Pong received ping~n", []),

                %sends the atom pong to the agent "ping":
                Ping_PID ! pong,
             pong()
     end.
%%the function start is for creating an agent
start() ->
    %%Pong_PID is the agent identity of the "pong" agent
     %%and The function start now creates another agent "ping"
     Pong_PID = spawn(ping_pong, pong, []),
     %%Erlang thus provides a mechanism for processes to be
given names so that these names
     %%can be used as identities instead of pids
     register(pong, spawn(ping_pong, pong, [])),
    spawn(ping_pong, ping, [23]),
     %% executes the function shown in the code below
     ping_pong:ping(3, Pong_PID).
```

Figure 5.4 A ping pong agent example where two agents send messages to each other.

5.2.1 Starting a communication

The function start first creates an agent; let's call it "pong":

```
start() ->
     Pong_PID = spawn(ping_pong, pong, []),
```

Figure 5.5 Creating an agent in Erlang

This agent executes ping_pong(). Pong_PID is the agent identity of the "pong" agent. The function start now creates another agent "ping" as shown in the code below.

```
spawn(ping_pong, ping, [23]),
```

Figure 5.6 Creating an agent using spawn function

This agent executes the function shown in the code below

```
ping_pong:ping(3, Pong_PID).
```

Figure 5.7 Sending message to the created agent

An id value (e.g. <0.36.0>) is the returned from the start function.

The agent "pong" now does what is shown in the code below:

```
receive
       %% if the agent finished
          finished ->
              %% print that out
            io:format("Pong finished~n", []);
        %% and if he recieves this format
          {ping, Ping_PID} ->
              %% do the following which is printing.

            io:format("Pong received ping~n", []),

              %sends the atom pong to the agent "ping":
              Ping_PID ! pong,
```

Figure 5.8 Message receiving snippet

5.2.2 Message Structure (envelope)

The receive construct is used to allow agents to wait for messages from other agents. It has the following format:

```
    receive
  pattern1 ->
      actions1;
  pattern2 ->
      actions2;
  ....
  patternN
      actionsN
end.
```

Figure 5.9 Message structure

Each agent has its own input inbox for messages it receives. Messages are routed to wherever the destination agent resides and are placed in its message inbox. New messages received are put at the end of the inbox queue data. When an agent executes a receive operation, the first message in the inbox is matched against the first pattern in the receive operation. If this matches, the message is removed from the inbox and the actions corresponding to that pattern are executed. However, if the first pattern does not match, the second pattern is tested. If this matches the message is removed from the inbox and the actions corresponding to the second pattern are executed. If the second pattern does not match the third is tried and so on until there are no more patterns to be tested. If there are no more patterns to test, the first message is kept in the inbox and the second message is tested instead. If this matches any pattern, the appropriate actions are executed and the second message is removed from the inbox (keeping the first message and any other messages in the inbox). If the second message does not match the third message is tested and so on until the end of all the messages in the inbox. If the test reaches the end of the inbox, the agent stops execution and waits until a new message is received and this procedure is repeated.

5.2.3 *Agent Message Transport & Transport*

Erlang allows for “clever” implementation that minimizes the number of times each message is tested against the patterns in each receive.

Using the ping pong example, "Pong" is waiting for messages. If the message “finished” is received, "pong" writes "Pong finished" to the output and as it has nothing more to do, so it terminates. If it receives a message with the format:

```
            {ping, Ping_PID} ->
```

Figure 5.10 Format of the message

It writes "Pong received ping" to the output and sends the atom pong to the agent "ping":

```
Ping_PID ! pong,
```

Figure 5.11 How to send message

Erlang uses the operator "!" to send messages between processes. Since Erlang is used in programming the agents, the operator "!" is therefore used to send messages between agents.

After sending the message pong, to the agent "ping", "pong" calls the pong function again, which causes it to get back to the receive again and wait for another message. The agent "ping" was started by executing:

```
ping_pong:ping(3, Pong_PID).
```

Figure 5.12 Sending message to a targeted agent

In the function ping/2 the second clause of ping/2 is executed since the value of the first argument is 3 (not 0) (first clause head is ping(0,Pong_PID), second clause head is ping(N,Pong_PID), so N becomes 3).

The second clause sends a message to "pong":

```
pong ! {ping, self()},
```

Figure 5.13 How an agent sends message to itself

self() returns the pid of the agent which executes self(), in this case the pid of "ping".

"Ping" now waits for a reply from "pong":

```
receive
        pong ->
            %% print that out
               io:format("Ping received pong~n", [])
    end,
```

Figure 5.14 The receiving block

and writes "Ping received pong" when this reply arrives, after which "ping" calls the ping function again.

```
ping(N - 1).
```

Figure 5.15 Decreasing an argument by 1

N-1 causes the first argument to be decremented until it becomes 0. When this occurs, the first clause of ping/2 will be executed:

```
ping(0) ->
    pong ! finished,
    io:format("ping finished~n", []);
```

Figure 5.16 How to terminate a communication

The atom finished is sent to "pong" (causing it to terminate as described above) and "ping finished" is written to the output. "Ping" then itself terminates as it has nothing left to do.

In order to allow communication between agents like in the ping-pong example above, agents have different behaviours; behaviour is the action or reaction of the agent under specified circumstances e.g. sending or receiving message. The message sending agent has the following behaviours:

- Send message
- Wait for response

While the receiving agent has the following behaviors:

- Receive message
- Respond to message

During the communication period, the message sent usually has additional information to its content. The message has content, the intended recipients, the sender and the message pattern. Assuming an agent has id information for the recipient, sending a message is fairly easy. Below is the code example that is executed when the command to send message from agent A to agent B.

```
Agent A ! agentb
```

Figure 5.17 Sending message operation

5.2.4 Agent Management System

In multi-agent systems, it is important that individual agents communicate and interact using the behaviours described above. This is achieved through the exchange of messages and, to understand each other, it is important that agents agree on the format and the way these messages are sent. AMS is Agent Management System; it provides a way to register agents and provides a mechanism for agent message transportation. AMS routes the messages in different ways, depending on where the agents are, i.e. in the same machine, in a different machine. The AMS requires every agent to register itself in a global registry upon creation. They are registered in a way that they are also stored so that the AMS can resolve them. The registration will give each

agent an identity which is used during communication with other agents. The figure below shows how two agents say agent A can be created and registered.

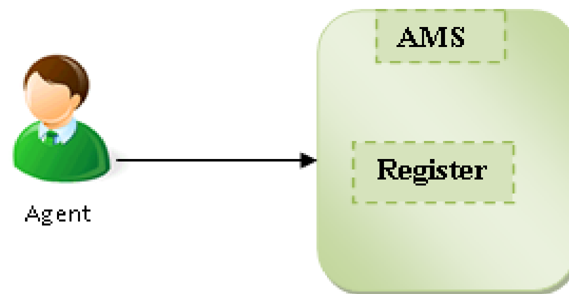

Figure 5.18. Agent registration process

The agents in this thesis are implemented using Erlang programming language; the code below shows how the two agents, agent A and agent B are created.

```
start() ->
     %spawn creates a new process
     %and The new process will start executing in agents:agentb
     % [] the arguments
     Agent A spawn(agents, agentb,[]),
```

Figure 5.19 Creating an agent - agentb

And the code below shows how the two agents, agent A and agent B are registered in the AMS

```
start_link([]) ->
     %global is the ServerName which can be local too
     %the name of the database which is (ETS)
     % the function to register agent

     gen_server:start_link({global, ?DATABASE}, ?ETS,
{"locations"}, []).

% the function register is to register the name and the job of
the agent
register_agent(Name,Job) ->

     gen_server:call({global, ?DATABASE},
{register_agent,Name,Job}).
```

```
%%Creates a table and returns a table identifier
init({File}) ->
%the parameter Options is a list of atoms which specifies table
type, access rights, key position and the location
     ets:new(?ETS, [named_table,{keypos,#directory.name}]),
    {ok, #state{}}.
```

Figure 5.20 Registering agents in the AMS

5.2.5 Speech Act /ACL Representation

When an agent wants to create a request for proposal, this will require creating a RESTful proposal. This proposal can be put out as RESTful resource which can be cached. This caching allows the system to have some advantages which could be applied to a proposal, a request or a bid that can be easily accessible to agents. The communication scenario between agent A and agent B can be created in a RESTful manner following these steps:

5. Agent A creates a call for proposal; this step includes the following:

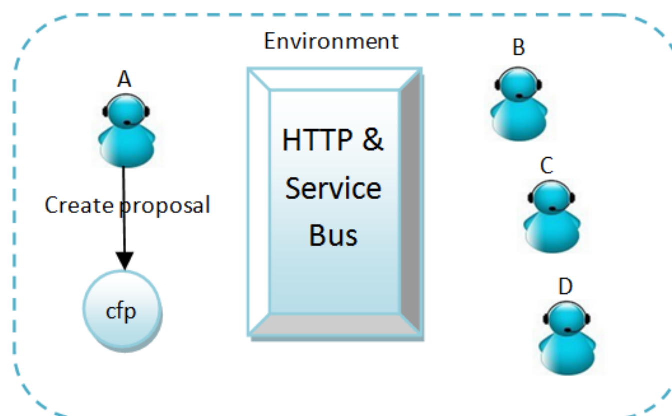

Figure 5.21. Agent create call for proposal

a. Create the proposal using POST

```
start() ->
  %spawn creates a new process
  %and The new process will start executing in agents:agentb
```

```
  % [] is the arguments
  Agent A = spawn(agents, agentb,[]),

  gen_server:start({global,proposal_generator}, prop_gen, [],
[]).

%post the proposal
post_prop(Text) ->
  gen_server:call({global,proposal_generator}, {post, Text}).
```

Figure 5.22 Posting the proposal

b. Inform other agents of the presences of a proposal using Service BUS using PATCH. It is not known if all agents will receive the information about the proposal.

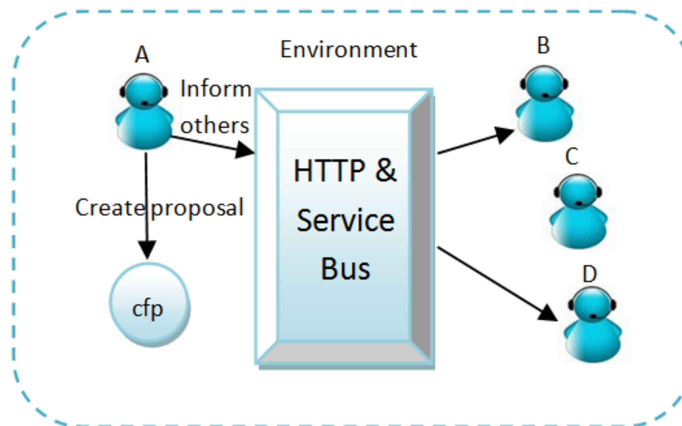

Figure 5.23. Agent uses the service Bus to broadcast information

```
%inform other agents about the proposal
    patch(inform,Agent_ID) ->
    gen_server:call({global,cc}, {inform,Agent_ID}).
```

Figure 5.24 How to inform agents about a proposal

6. The call for proposal is taken or represented as an entity (resource).
7. Agent A could be informed whether the proposal has been accepted by either of the following:

a. Agent A monitors the proposal using the GET request routine.

```
%% using get to check on the proposal
    get_proposal(ID) ->

    gen_server:call({global,ID}, {get}).
```

Figure 5.25 How to check the proposal

b. The proposal has knowledge of a call back routine that when it receives something it will inform agent A of the change using PATCH routine.

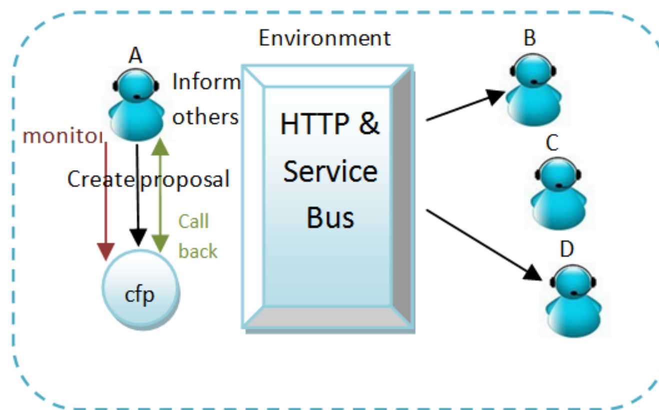

Figure 5.26. Agent uses PATCH routine to receive information

8. Agent B then accepts the proposal by first going to the proposal and POSTing ‘I accept this proposal’. There is a chance other agents could either have accepted or rejected this proposal.

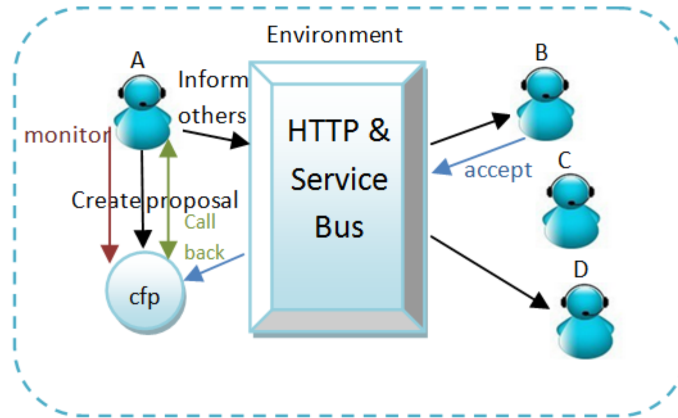

Figure 5.27. Agent B accepts the proposal

```
%% if any one of the agents accept the proposal
post(accept, Proposal_id) ->
         gen_server:call({global,proposal_generator}, {accept,
                                   Proposal_id}).
```

Figure 5.28 How to check if proposal is accepted

5.3 Conclusion

The code examples above show how agents can be created, using Erlang and how they can communicate using an environment designed with asynchronous parallelism as a basis. This relieves the programmer from complicated coding required for keeping track of asynchronous thread distribution and message passing. The scalability of an implementation using an Erlang based environment becomes evident by the benchmark testing [39], compared to Java and like environments, whether one is looking at sheer numbers of threads, spawn times, message passing, memory allocation requirements and/or code generation. Additional benefits of dynamically allocated stacks, concurrency, event-driven asynchronous messaging, fault-tolerance and/or Hot-code-swapping, add to the benefits and to the requirements of future implementations to accommodate such demands and to truly utilize new hardware (such as

multi-core processors and distributed processing) and the increasing number of users/clients on popular applications. Adding a RESTful approach to inter-agent communication promotes further scalability to the robustness of the Erlang environment, in compliance with REST's client-server, uniform-interface and statelessness constraints.

CHAPTER 6 SYSTEM PERFORMANCE EVALUATION

In order to evaluate the performance of the system, a series of simulation experiments were conducted. For the experimental setup, a Microsoft Windows server running Windows 7 (Version 2009, Service Pack 1, with an Intel ® – Xeon® CPU, 2.33 GHz and 16.0 GB RAM) with Erlang was the primary platform. The experiments attempts to generate large numbers of agents and initiate messages between the system and agents.

In Erlang there is mechanism to prevent processes being overwhelmed by messages. If the process is accumulating too many messages in its queue and thus has problems processing these messages, those processes that send additional messages to this process will be penalized (reduced CPU allotment). The scheduler will punish processes that send additional messages by reducing their time slice. Each process in Erlang maintains and uses its own heap. This heap is divided into two halves and only one half is used at anytime. When the process needs more memory and has no more space in the heap, the garbage collection is activated. When this happens, the process copies elements from one half to the other and removes items that are no longer needed (not referenced). When there is free space that fulfills the need, the process utilizes the available memory and continues with its task. If the free memory is not available, the process requests an increased heap from the Erlang Virtual Machine. The process will therefore be allocated a larger heap following the Fibonacci number sequence.

The main interest of this research was in the scalability of systems, specifically, the communication process. For this reason, the system that is tested must have lots of agents handling and sending many messages. For example, in a scenario where we have ***a*** number of ***agents*** and ***m*** number of ***messages***, it is possible to split these agents over ***h*** number of ***host***

servers. In this case the agents reside over ***n*** number of ***nodes***. We could also have the number of concurrent messages/ parallel messages as a parameter ***p***.

Following the above scenario we have the parameters for an experiment as ***a, m, n (h)*** and ***p***. This allows us to know how many agents are sending at the same time. This scenario shows that there is need for four experimental parameters. This thesis examines the impact of each of these parameters by making one parameter values flexible while maintain the other three as fixed values.

There are four sets of experiments performed. The first set measures how long it takes to create agents in our system environment. Create a large number of agents, and then initiate a set number of messages. The number of messages will be increased, linearly, and performance will be measured. Similarly, the experiment will be repeated with a linearly increasing number of agents (from sample experiments, the number of simultaneous agents can reach, possibly, more than one million).

6.1 Experiment Goals

Goal 1 – To determine the relationship between time and the number of agents being created

Goal 2 – To examine the impact of increasing parallelism provided a constant number of agents each sending a constant number of messages.

Goal 3 – To examine the impact of increasing the number of messages sent by a constant number of agents

Goal 4 – To examine the impact of increasing the number of agents each sending a constant number of messages

Goal 5 – To examine the impact of increasing the number of hosts having a constant number of messages and agents

Table 6-1. Experiments and Goals

Goal	Experiment
Determine the relationship between time and the number of agents being created	# of Agents Scalability & Stability
Examine the impact of increasing the number of agents sending messages in parallel, each sending a constant number of messages.	# Agents & Messages Scalability & Stability &
Examine the impact of increasing the number of agents each sending a constant number of messages.	# of Agents Scalability & Stability
Examine the impact of increasing the number of messages sent by a constant number of agents.	# of Messages Scalability & Stability
Examine the impact of increasing the number of hosts and having a constant number of agents and messages.	# of Messages Scalability & Stability

6.2 Parameters

The total amount of Random Access Memory (RAM) in an environment, available to a program, is the primary limiting factor for that program's creation of multitudes of agents (independent threads or processes).

And, while system-specific memory-leaks, background processes and services may affect performance and other experimental results, they will be kept to a minimum by rebooting the server before every run, and allowing time for instantiation and boot processes to complete. Additionally, the systems "Task Manager" was used, in conjunction with Erlang structures (memsup and wall-clock), to document system states and resource usage. Each run will be

repeated, the results will be averaged but anomalous numbers will be dismissed, which may be due to the aforementioned OS irregularities.

As of OTP R12B, Erlang has Symmetric Multi-Processing (SMP) support on all the major platforms, and it's enabled by default [9]. The Erlang environment will be augmented, on initiation, with parameters to achieve best results. Theses augmentations will include "+P" to increases the maximum number of threads, from the default 32768, as well as "+hms" to decrease the default thread size allocation, from the default 309 byte [8]. Erlang garbage collection was not being addressed in these experiments as each program terminated before any process gets "old".

6.3 Experiments setup

The experiments used eight (8) Erlang programs to achieve the results required to address the above described goals and problem statement-questions, as stated in chapter two.

Experiments 1: Scalability and Stability.

The main goal of this experiment is to determine the time it takes to create an increasing number of agents, as well as finding, at least, the primary reason(s) for the breakdown. Program alpha "A", creates ***n*** agents (starting at 1,000 agents increasing to 1 million). Processing time is recorded as well as RAM state before and after each instantiation. "N" is increased until the system becomes unstable.

Experiments 2: Scalability, Stability, and performance.

Table 6-2 Experiment 2 parameters

Agents	Messages	Nodes/host	Parallel
100000	10000	8	1
100000	10000	8	10
100000	10000	8	100
100000	10000	8	200
100000	10000	8	400
100000	10000	8	600
100000	10000	8	1000
100000	10000	8	2000

From the table above, we have 100000 agents, sending 10000 messages split over 8 nodes and we want to measure the impact of parallelism. We therefore have the forth parameter as flexible. In the case we want to test the impact when parallel value is at 1, 10, 100, .. up until 2000. It is important to remember that the number of agents actually relates to the memory size of a computer hosting the agents. The issue of memory will also affect the number of messages being sent. The host is the indicator of the inter-process communication so that the larger this number is the higher the inter process communication there is, this is bound to the CPU of the system. The challenge is to know how much is happening on the system. In the above table, one agent sends 10000 messages, 10 agents sends 1000 messages, 100 agents sends 100 messages and 1000 agents sends 10 messages. The setup of this experiment would be able to give us an indication of the impact that is how fast these messages are processed in the system.

After running the second experiment on parameter p, the next step is the experiment on the number of messages, which is the parameter m. The purpose of running this third experiment is to allow us see the difference and note which one is the slowest. In this second experiment the value of p is kept at 1000 and the rest of the parameters also remain constant except m changes as shown in the table below.

Experiments 3: Scalability and Stability.

Table 6-3 Experiment 3 parameters

Agents	Messages	Nodes/host	Parallel
100000	1000	8	1000
100000	2000	8	1000
100000	5000	8	1000
100000	10000	8	1000
100000	25000	8	1000
100000	50000	8	1000
100000	80000	8	1000
100000	100000	8	1000

In this case, 1000 agents send 10 messages, 1000 agents sends 100 messages, 1000 agents sends 1000 messages and 1000 agents sends 10000 messages. The numbers of messages being send increases by a varying factor. This second experiment seeks to find out is the system takes more time by the same factor or is it merely indistinguishable.

In the fourth experiment, the parameter whose impact is to be tested is the number of agents, *a*. The number of message is initialized to the 1000 as the beginning of the experiment. The nodes and number of parallelism remains at 8 and 1000 respectively. The setting of the experiment is shown in the table below.

Experiments 4: Scalability and Stability.

Table 6-4 Experiment 4 parameters

Agents	Messages	Nodes/host	Parallel
2000	10000	8	1000
5000	10000	8	1000
10000	10000	8	1000
25000	10000	8	1000
50000	10000	8	1000
80000	10000	8	1000
100000	10000	8	1000

The number of agents will be increased from 2000 to 100000. Running each of these number of agents against a contestant number of messages, host and parallelism, we can see which of these test has the strongest impact on the system. It would then be possible to tell what would be a problem when using agent based system, that knowing if it is the number of agents, number of messages, and number of hosts or is it the number of parallel running agents.

Experiments 5: Scalability and Stability.

Table 6-5 Experiment 5 parameters

Agents	Messages	Nodes/host	Parallel
100000	10000	8	1000
100000	10000	12	1000
100000	10000	16	1000
100000	10000	20	1000

From the table above, we have 100000 agents, sending 10000 messages and we want to measure the impact of changing the number of nodes. We therefore have the third (host) parameter being change. In this case we want to test the impact when the nodes are increased by a factor of 4, i.e. start from 8, then 12, then 16 and finally 20.

These five set of experiments should tell which factor is the most important one. Based on the results of the experiments, we should be able to say increasing the number of agents do not affect the system but increasing the number of messages makes the system run slower and slower.

6.4 Results and Evaluations

Results for Experiments 1:

In Figure 6.1, the x-axes represent the number of agents created and the y-axis represents the time in seconds it takes to create the agents. The aim of this figure is to show the relationship that exists between the numbers of agents being created with the total time it takes to create them.

From the graphs it can be observed that there is a linear relationship between these two parameters at the beginning, there is then a constant relationship before finally showing a linear relationship. This behaviour displayed by the graph was not expected. The beginning of the graph shows that there is a small numbers of agents to be created its clear there is a linear relationship between the number of agents and the time it takes to create them. The presence of garbage collection may affect the system and a possible explanation for the constant relationship between the number of agents and time. However, as the number of the agents' increases, the presence of garbage collection is minimal hence the linear relationship between agents and time is restored because the agents are created and removed from the schedule. This shows that if there is need to create a larger population of the agents, the time that will be required to create them will also increase. According to this graph, an increase demand for higher number of agents to be created will have a linear impact on the system scalability.

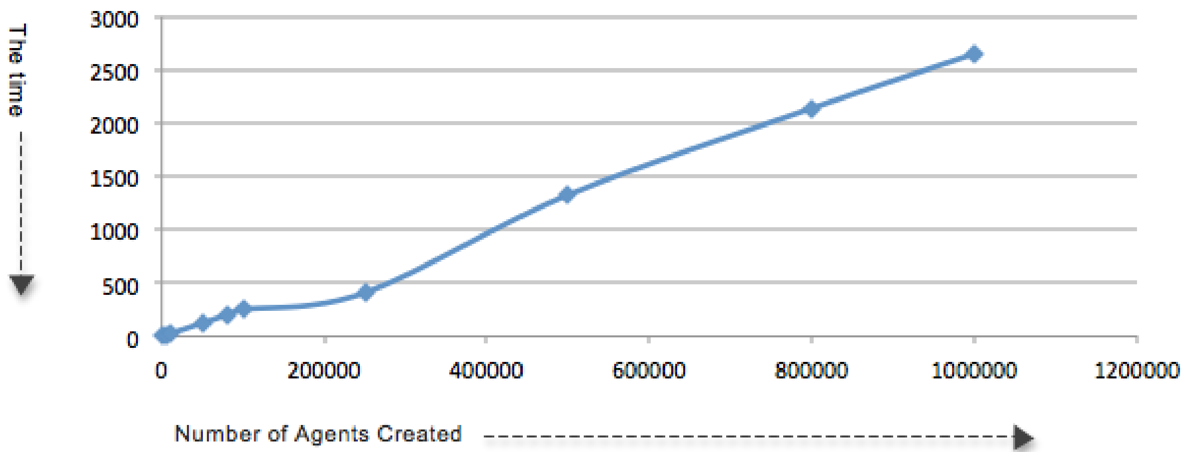

Figure 6.1 Agent creation times

Results for Experiments 2:

For the second experiment as shown by Figure 6.2, the aim is to examine the behaviour of the system in handling an increased parallelism in the number of agents each sending a constant

number of messages. In the graph, the x-axes represents the number of parallel agents sending messages concurrently and the y-axis represents how long it takes to send all of these messages. The number of messages is 10000. This graphs shows some interesting relationship that exist between the number of agents that can each send 10000 message with the total time it takes to send all the messages. From the graph it can be observed that there is an interesting relationship between these two parameters. This shows that if the population of the agents is increased within the range of about 200-800, the time that will be required for each of them to send a single message will also increase. The increase is possibly increased because of the presence of garbage collection. However, for small number of parallel agents, below 200 and a number higher than 800, it is possible to make a case for a logarithmic relationship. This behaviour displayed by the graph was not expected. According to this graph, an increase in the number of agents will have minimal impact on the system scalability.

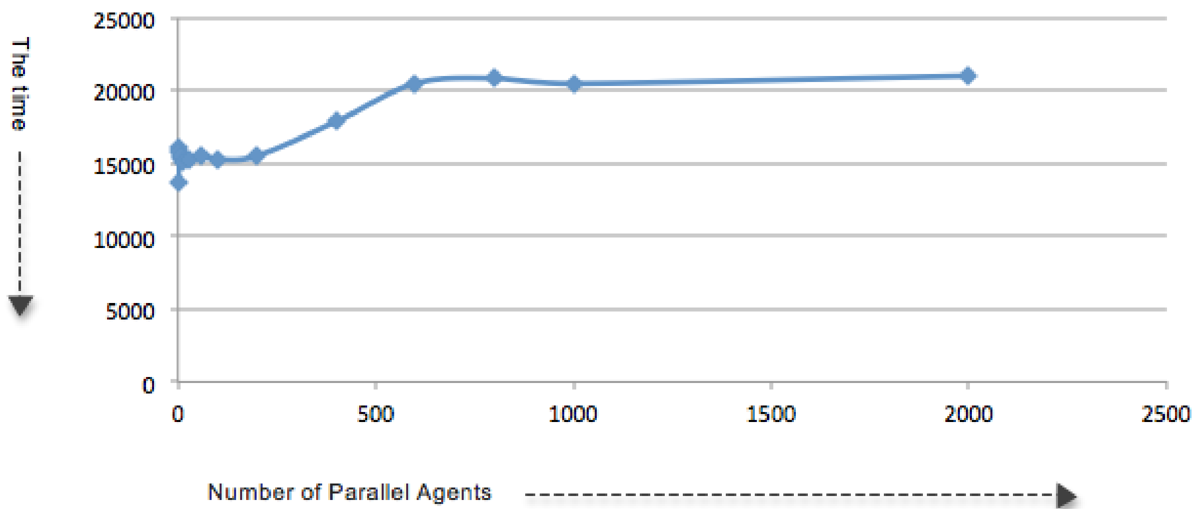

Figure 6.2 Parallel message sending times with both numbers of agents and messages constant

Results for Experiments 3:

For the third experiment as shown by Figure 6.3, the x-axes represents the number of messages being send by 100000 agents and the y-axis represents how long it takes for all of these messages to be send. The aim of this experiment is to examine the relationship that exists between the numbers of messages that can be sent in relation to the number of agents. From the graph, an interesting observation is shown, it takes way longer for the same number of agents to send fewer message. It is also observed that, after a certain number of messages, most likely 20000, the time becomes constant. This shows that if the number of messages is increased beyond 20000, the average time that will be required for them to be sent will be constant. According to this graph, this increase will have no impact on the system scalability.

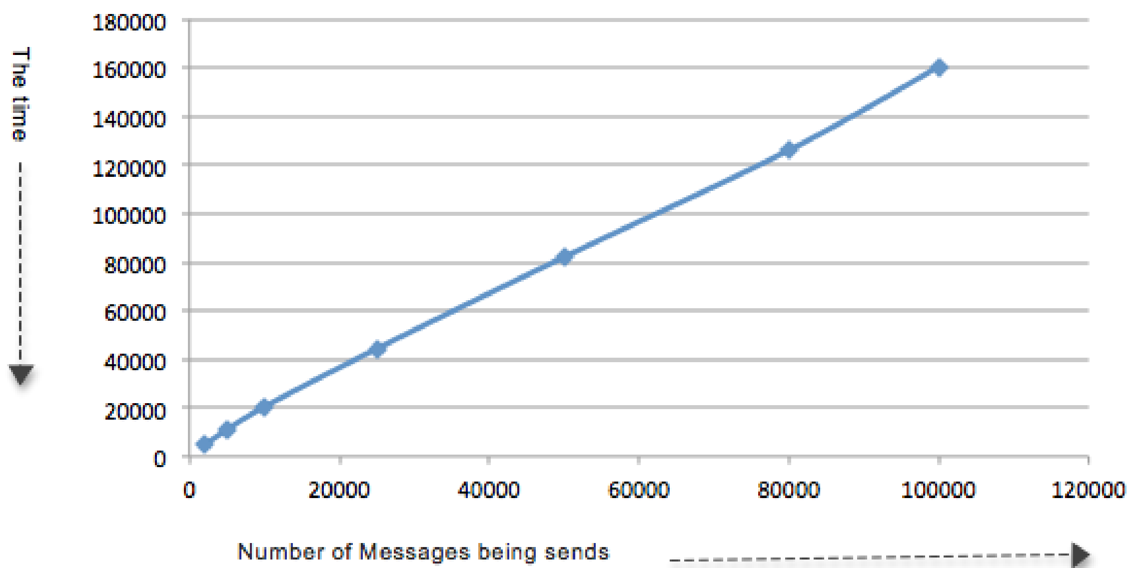

Figure 6.3 Messages sending times with constant numbers of agents and varying number of messages

Results for Experiments 4:

For the forth experiment as shown by Figures 6.4, the x-axes represents the number of agents in the system with 1000 of them sending messages concurrently and the y-axis represents how long it takes for these concurrent agents to send these messages. The number of message is 10000. The aim of these two graphs is to show the relationship that exist between the number of agents

in the systems with the total time it takes 1000 agents to send 10000 messages. From the graphs it can be observed that there is a linear relationship between these two parameters. This shows that if the number of messages is increased, the time that will be required for them to communicate by sending them will also increase. According to these graphs, an increase in the number of messages will have a linear impact on the system scalability.

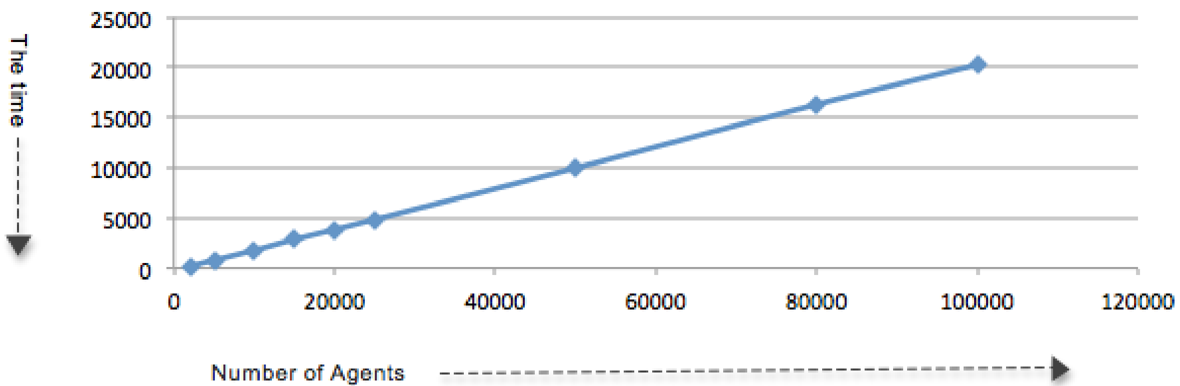

Figure 6.4 Messages sending times with constant numbers of messages and varying number of agents

Results for Experiments 5:

For the fifth experiment as shown by Figure 6.5, the aim is to examine the behaviour of the system in handling an increase in the number of hosts in the system. In the graph, the x-axes represent the number of parallel hosts and the y-axis represents how long it takes to send 10000 messages. This graphs shows some interesting relationship that exist between the number of hosts used with the total time it takes to send all the messages. From the graph it can be observed that as the numbers of hosts are increased, the time taken to perform the same operation is reduced. This is the case because the sending operations are distributed among the hosts and so the work load for each host is reduced meaning it will all its resources can be directed at performing the sending of messages operation quickly. According to this graph, an increase in

the number of hosts will decrease the time needed to perform tasks and have positive impact on the system scalability.

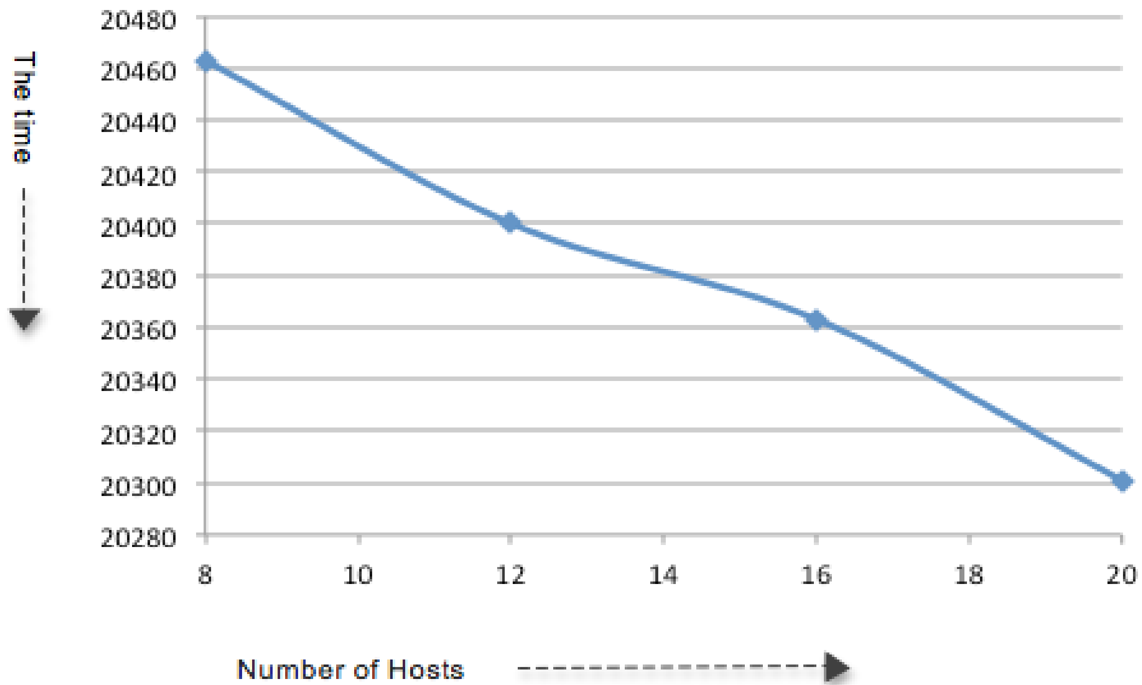

Figure 6.5 Messages sending times with increased number of hosts and constant numbers of messages & agents

6.5 Conclusion

This chapter focused on various experiments that were conducted in order to evaluate the proposed architecture based on FIPA Abstract Architecture and Erlang. The experiments focused on the following factors: scalability, stability and performance. This chapter starts off by providing an overview of the experimental setup and gives details about machine specifications. Section 6.1 describes the list of experimental goals that will be tested in the proposed architecture. A total of four experiments were performed.

The first experiment focused on testing the number of agents that the system could handle creating given time constraints. This test was performed by instructing the system to create a

target number of agents. The time it took to create that number of agents was recorded. The same process was repeated as the number of target agents was increased. A large number of agents were created, and a set number of messages were initiated. The number of messages was increased, linearly, and performance was measured. A similar experiment was repeated with a linearly increasing number of agents. Experiment 2, 3, & 4 focused on testing the scalability aspects of the system by finding the maximum number of agents and messages they send that the system could handle, as the number of messages is increased.

The summary of the experiments are listed below based on the goals.

Goal 1 - Determining the relationship between time and the number of agents being created. The results shows that overall there is a linear relationship between the time it takes and the number of agents to be created. In essence, if there is need to create a larger number of agents, then enough time is required and should be allocated.

Goal 2 - Examining the impact of increasing the number of agents sending messages in parallel.The results shows that when the number of agents sending messages parallely is in the range of 200 – 800, a linear relationship between increase in the number of agents and the time it took for each agent to process all the incoming messages is expected. However, the results show a logarithmic relationship between the numbers of agents and time it takes to process the messages.

Goal 3 - Examining the impact of increasing the number of agents each sending a constant number of messages.The results shows a linear relationship between an increase in the number of agents and the time it took for each agent to process the incoming messages.

Goal 4 - Examining the impact of increasing the number of messages sent by a constant number of agents.The results shows that there is a linear relationship between the numbers of messages

and the time it took to send the messages. Therefore, if the messages being sent are increased, the time required for these messages also increases.

Goal 5 - Examining the impact of increasing the number of hosts but maintaining a constant number of agents and messages.The results shows that if there is an increase in the number of hosts, the time required to send the same number of messages is reduced.

CHAPTER 7 CONCLUSION AND CONTRIBUTION

Before expounding on the detailed explanation of the experimental results and their conclusion, here is a quick reminder of the main questions this research aimed to answer and a look at the contributions. The questions raised in chapter 2 are as follows:

> **Question 1:** What are the primary environmental factors that affect implementation of a scalable multi-agent system?
>
> **Question 2:** Is it possible to follow the architecture of the web to create scalable multi-agent systems?

As a contribution, this thesis presented an approach of converting FIPA ACL into REST. It aimed at applying RESTful principles into multi-agents. This approach allows for building more scalable systems. As shown in Chapter 4, all standard FIPA Communicative Acts Specifications can be converted to REST. The second contribution on use of RESTful architecture is the evaluation of agent-based system based on Erlang. This thesis examined the number of agents and some factors (number of agents, number of messages, number of hosts and number of messages sent in parallel) that affect the design and implementation of such a system.

The first question aims at identifying the environmental factors that affect scalability of multi-agent systems specifically from a scalability and stability perspective. It further aims to answer concerns on how environmental factors are affected by sustained increases in demand on bounded resources. In other words, what are the factors that need to be considered in order to improve the scalability of a multi-agent system?

To answer this first question, this thesis takes into account experiment performed and factors mentioned in other literature. Scalability concerns the ability of a system to solve a problem even when the size of the problem increases [45]. A system is considered scalable if when resources

such as memory and processor are added, the performance is increased [40]. Scalability is one of the important factors to consider when implementing multi-agent systems. According to [40], the main factors that impact scalability of multi-agent system are load and complexity. The load factor can be said to include memory load, CPU load, and communication load. Memory load referring to how much memory is used and occupied by the agents. CPU load refers to how much the CPU is used by the agents for processing and computation. Communication load refers to how the messages are routed as a major form of communication between the agents. Computation complexity contributes to the scalability of the system depending on the algorithms used.

This thesis focused on the communication load as a factor that affects scalability of a multi-agent system. It focused on factors that affect a multi-agent system needing to scale in a number of different directions pertaining communication as follows: increasing the number of agents within a system, increasing the number of communication happening in parallel, increasing the number of messages to be routed, and increasing the number of hosts. For these reasons, the system that is tested must have lots of agents handling and sending many messages. The experiments are performed considering the number of agents and the number of messages. The number of host servers is also considered. Finally, the number of concurrent messages/parallel messages being sent by the system is tested.

This research aimed to evaluate four factors affecting scalability of multi-agent systems specifically pertaining to communication load. To examine the impact of these factors, four sets of experiments were performed. The first set measured how long it takes to create agents in our system environment. A large number of agents are created, and then initiated a set number of messages. The number of messages is gradually increased, linearly, and performance of the

system is noted. The specific aim was to evaluate the impact creating agents had on the system. The results show it takes more time to a high number of agents. Therefore, it can be concluded that within a similar context (platform specification) used in this research, number of agents being created is a factor that affects scalability of a multi-agent system.

The second experiment evaluated the behaviour of the system in handling constant number of agents each sending a constant number of messages in parallel. This experiment gave an indication of how fast the messages were being processed. The results shows that there is treshold of when the number of parallel agents is no longer relevant and does affect the behaviour of the system. The number is around 800 agents. For a high number of agents, the performance does not change. This shows increasing the number of agents acting in parallel above a given treshold in a system could have litle impact on scalability.

The third experiment evaluated the impact of increasing the number of messages the agents have to process by increasing the number of agents. This experiment gave an indication of how fast the messages were being processed. The results shows that increasing the number of agents, also the time it takes for each agent to reply to an incoming messages. Therefore, it can be concluded that the number of messages being handle in the system is a factor that affects scalability of a multi-agent system.

The fourth experiment evaluated the behaviour of the system in handling an increase in the number of agents each sending a constant number of messages. This experiment gave an indication of how fast the messages were being processed. The results show that if the numbers of messages being sent are increased, the time required for these messages to be transmitted also increases. Therefore, it can be concluded that the number of busy agents in the system is a factor that affects scalability of a multi-agent system.

Other factors that influence scalability are the size of memory and processor [40]. When agents interact with each other to perform their intended purpose, the performance of the systems could lower due to optimization, and autonomy [43].

The second question aims to determine if the agents could be designed in a RESTful manner i.e. following the web design patterns. It also aims to determine if a web-centric agent design following FIPA ACTS agent's communication protocol is possible.

The answer to this second question is yes, agents and multi-agent systems can be created following RESTful design principles. It is important to note that based on literature review we can infer that it is possible to follow the architecture of the web to create scalable multi-agent systems. By reviewing the work done by Richardson [37] "HATEOAS and RESTful design", Brennan [42] "REST" and Parastatidis et al. [33] "REST and Distributed Systems", we can logically argue that agents for multi-agent based systems can be created following the web-centric approach. By taking advantage of the verbs provided by the REST architecture for communication, the web-centric agent design can allow communication between agents by following the FIPA ACTS agent's communication protocols. More details on what these researchers are provided in Chapter 3 sections 3.6, 3.5 and 3.7 respectively.

According to Brannan [42], it is possible to use RESTful approaches to design and create web applications which emphasize a simple point-to-point communication over the backbone of the web infrastructure; the HTTP protocol. She also argues that such system (designed following the REST approach) was simpler and more concise. Richardson [37] focuses on the development of system that changes its model and state from time to time. More like agents reacting to receiving and sending messages. Such approach enables building of systems that evolve, and provides loose coupling between clients and servers. Parastatidis et al. [33] discusses the role of REST in

developing distributed systems. He mentions that REST is governed by HTTP which requires applications state, the business process that affect state, distributed data structures that hold it, and the contracts and protocols that establish communication between the different components of the system.

The primary factor that will allow the scaling solutions, using agents, into multiple millions of agents, will be the separation of server side agents and client side agents (and defining new standards for describing each environment's needs, limitations, functions, etc.), as per REST. Applying this will utilize the resources of the clients' machines (each client having a small number of agents, using local resources), rather than the bounded and limited resources of a server, or even a bank of servers, as the number of clients "goes viral".

CHAPTER 8 FUTURE WORK

8.1 Introduction

The first version of a system based on FIPA Abstract Architecture and Erlang implementation has been developed, and it can be used to examine various factors affecting scalability in a multi-agent system. At present our system implementation aims at examining four factors including number of agents, number of messages, number of hosts, and number of messages sent in parallel.

In this Chapter, some future directions are presented.

8.2 Future directions

This work in the experiment and the evaluation sections focused on the following factors: scalability, stability and performance. However, it did not focus on a set of experiments that use a static number of agents and compare performance changes as a RESTful FIPA ACL compliant mechanism adds to the messaging overhead. As part of the future work experiments intended to test the REST aspect of the implementation, would be of interest.

Part of the future work would be to identify what resource overhead would be incurred if agent communications complied with RESTful constraints? Two programs might be used that differ in the protocol used for message exchanged, RESTful and not. Both programs would otherwise be the same, using the same number of agents, content and size of messages exchanged.

Considering that multi-agent systems depend on network for communication between nodes, it will be important to consider the impact of network bandwidth as a resource that is useful to the functioning of the system. When an agent is to send messages to other agents located on different nodes network bandwidth would be an important resource.

It would be interesting to perform a comparison between the results of RESTful and Non-RESTful experiments, to determine the difference, in performance, between Erlang created agent communication with the only differentiation being the native Erlang inter-messaging RESTful and non-RESTful communication, thereby allowing the assessment of any additional overhead incurred.

In this thesis, the agents built for the system that was evaluated were not complex in their tasks and operations. It would be interesting to study the effect of smarter or more complex agents being used in the system and see what effects such agents will have on the scalability and performance of the system.

The programming platform used for programming the system in this work is Erlang. As part of a future work, it would be interesting to use another functional language in programming the system and compare the performances of the two systems.

During the performance analysis of the system that was built, a comparative study against other larger FIPA systems. As part of future work it would be interesting to perform a comparative study between a large FIPA system built using Java against the system that has been built for the purposes of this thesis.

Injection of fault and failures was not done during the evaluation of the system. It would be interesting to perform a study where simulation of injected faults and failures are introduced. Such simulated faults include network failures and deliberately killing processes.

CHAPTER 9 LIST OF REFERENCES

[1] Agarwal, S., Lakhina, P. Erlang - Programming the Parallel World, http://cs.ucsb.edu/~puneet/reports/erlang.pdf (last accessed 20/10/2012)

[2] AgentBuilder. Why, When, and Where to Use Software Agents, http://www.agentbuilder.com/Documentation/whyAgents.html (last accessed 20/10/2012)

[3] Armstrong, J. (2007) "History of Erlang", in HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages, 2007, ISBN 978-1-59593-766-X

[4] Benchmarking CLIPS/R2, http://www.pst.com/benchcr2.htm(last accessed 20/10/2012)

[5] Charlton, P., Cattoni, R., Potrich, A., and Mamdani, E. (2000), Evaluating the FIPA Standards and Its Role in Achieving Cooperation in Multi-agent Systems. Proceedings of the 33rd Hawaii International Conference on System Sciences – 2000

[6] Deters, R. (2001). Scalability & multi-agent systems. 2nd International Workshop Infrastructure for Agents MAS and Scalable MAS 5th int conference on Autonomous Agents (pp. 1-8). Retrieved from http://users.cs.cf.ac.uk/O.F.Rana/agents2001/papers/02_deters.pdf.

[7] ERLANG How do I…, http://www.erlang.org/faq/how_do_i.html(last accessed 20/10/2012)

[8] Erlang Run-Time System Application (ERTS) Reference Manual, http://www.erlang.org/doc/man/erl.html(last accessed 20/10/2012)

[9] ERLANG STDLID Reference Manual, http://www.erlang.org/doc/man/gen_server.html(last accessed 20/10/2012)

[10] Fernandez, F., & Navón, J. (2010). Towards a practical model to facilitate reasoning about REST extensions and reuse. In C. Pautasso, E. Wilde, & A. Marinos (Eds.), Proceedings of the First International Workshop on RESTful Design WSREST 10 (p. 31). ACM Press. doi: 10.1145/1798354.1798383.

[11] Fielding R. (2000). Chapter 5 of Fielding's dissertation is "Representational State Transfer (REST)"

[12] FIPA - Foundation for Intelligent Physical Agents, http://www.fipa.org/.

[13] FIPA Abstract Architecture Specification, http://www.fipa.org/specs/fipa00001/SC00001L.html(last accessed 20/10/2012)

[14] FIPA Agent Communication Language Specifications, http://www.fipa.org/repository/aclspecs.html(last accessed 20/10/2012)

[15] FIPA Agent Message Transport Service Specification, http://www.fipa.org/specs/fipa00067/SC00067F.pdf(last accessed 20/10/2012)

[16] FIPA Communicative Act Library Specification, http://www.fipa.org/specs/fipa00037/SC00037J.html(last accessed 20/10/2012)

[17] FIPA infrastructure, http://modelbased.net/soi/agent/agent_fipa.html(last accessed 20/10/2012)

[18] Franklin, S., and Graesser, A. (1996). "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents". Proceedings of the Third International Workshop on Agent Theories,

Architectures, and Languages, Springer-Verlag, 1996.
http://www.msci.memphis.edu/~franklin/AgentProg.html

[19] Gaku, Y., Hideki, T., & Hideyuki, M. (2007). A Platform for Massive Agent-Based Simulation and Its Evaluation. *AAMAS Workshop.* Japan: Springer-Berlin.

[20] GenieDB, White Paper Beating the CAP Theorem

[21] Gilbert, S., Lynch, N.,, Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services

[22] GitHub, https://github.com/trifork/erjang/wiki

[23] Hadley, M., Pericas-Geertsen, S., & Sandoz, P. (2010). Exploring hypermedia support in Jersey. In C. Pautasso, E. Wilde, & A. Marinos (Eds.), Proceedings of the First International Workshop on RESTful Design WSREST 10 (p. 10). ACM Press. doi: 10.1145/1798354.1798378.

[24] Hai, Z. (2004). Exploring Flows in the Intelligent Agent Grid Environment. *Massively multi-agent systems I: first international workshop, MMAS 2004,* (pp. 13-25). Kyoto: Springer.

[25] Hewitt, C., Bishop, P., and Steiger, R., (1973). A Universal Modular Actor Formalism for Artificial Intelligence. IJCAI.

[26] Jacobi, I., & Radul, A. (2010). A RESTful messaging system for asynchronous distributed processing. Proceedings of the First International Workshop on RESTful Design WSREST 10, 46. ACM Press. doi: 10.1145/1798354.1798385.

[27] Jamali, N., and Zhao, X. (2006). "Distributed Coordination of Massively Multi-Agent Systems". Massively multi-agent technology: AAMAS workshops, MMAS 2006, LSMAS 2006

[28] Jamali, N., and Zhao, X. (2005). "Scalable Hierarchical Coordination of Multi-Agent Resource Usage". Proceedings of the Seventeenth Annual Conference of the Cognitive Science

[29] Lee, L., Nwana, H. S., Ndumu, D. T., & De Wilde, P. (1998). The stability, scalability and performance of multi-agent systems. BT Technology Journal, 16(3), 94–103. Springer. Retrieved from http://www.springerlink.com/index/JQ3237WQ7V41W413.pdf.

[30] Myeong-Wuk, J., & Agha, G. (2004). Adaptive Agent Allocation for Massively Multi-agent Applications. *Massively multi-agent systems I: first international workshop* (pp. 25-40). Kyoto: Springer.

[31] Nwana, H.S. (1996). Software Agents: An Overview. Knowledge Engineering Review, Vol.11, No.3, 205-244, Cambridge University Press

[32] Oxford Dictionary, http://oxforddictionaries.com/definition/scalable

[33] Parastatidis, S., Webber, J., Silveira, G., & Robinson, I. S. (2010). The role of hypermedia in distributed system development. In C. Pautasso, E. Wilde, & A. Marinos (Eds.), Proceedings of the First International Workshop on RESTful Design WSREST 10 (pp. 16-22). ACM Press. doi: 10.1145/1798354.1798379.

[34] Pautasso, C., Zimmermann, O., & Leymann, F. (2008). Restful web services vs. "big" web services. Proceeding of the 17th international conference on World Wide Web WWW 08, 08, 805. ACM Press. doi: 10.1145/1367497.1367606.

[35] Rana, O. and Stout, K. What is Scalability in Multi-Agent Systems. *Proceedings Autonomous Agents* 2000, Barcelona, pages 56-63, 2000.

[36] Rana, O. F., & Stout, K. (2000). What is scalability in multi-agent systems? In C. Sierra, M. Gini, & J. S. Rosenschein (Eds.), AGENTS 00 Proceedings of the fourth international conference on Autonomous agents (pp. 56-63). ACM. doi: 10.1145/336595.337033.

[37] Richardson, L. (2010). Developers enjoy hypermedia, but may resist browser-based OAuth authorization. Retrieved from http://portal.acm.org/citation.cfm?id=1798377.

[38] Selonen, P., Belimpasakis, P., & You, Y. (2010). Developing a ReSTful mixed reality web service platform. Proceedings of the First International Workshop on RESTful Design WSREST 10, 54. ACM Press. doi: 10.1145/1798354.1798387.

[39] SICS, http://www.sics.se/~joe/ericsson/du98024.html(last accessed 20/10/2012)

[40] Song, R. and Korba, L. The Scalability of a Multi-agent System in Security Services. *Published in NRC/ERB-1098* August 2002, 23 Pages. NRC 44952.

[41] SourceForge, https://github.com/trifork/erjang/wiki(last accessed 20/10/2012)

[42] Spies, B., Web Services, Part 1: SOAP vs. REST , http://ajaxonomy.com/2008/xml/web-services-part-1-soap-vs-rest

[43] Sridhar, P. and Madni, A. M. Scalability and Performance Issues in Deeply Embedded Sensor Systems. *International journal on smart sensing and intelligent systems*, vol. 2, no. 1, march 2009.

[44] Turner, P. J., & Jennings, N. R. (2000). Improving the Scalability of Multi-Agent Systems. Infrastructure for Agents MultiAgent Systems and Scalable MultiAgent Systems, 246–262. Springer Verlag. Retrieved from http://eprints.ecs.soton.ac.uk/6035/.