University of Windsor
Scholarship at UWindsor

Electronic Theses and Dissertations | Theses, Dissertations, and Major Papers

2011

A Distributed Task Management Solution for Peer-To-Peer and Cloud Environments

Lichun Zhu
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation
Zhu, Lichun, "A Distributed Task Management Solution for Peer-To-Peer and Cloud Environments" (2011). *Electronic Theses and Dissertations*. 106.
https://scholar.uwindsor.ca/etd/106

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

A Distributed Task Management Solution for Peer-To-Peer and Cloud Environments

by

Lichun Zhu

A Thesis
Submitted to the Faculty of Graduate Studies
through Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2011

© 2011 Lichun Zhu

A Distributed Task Management Solution

for Peer-To-Peer and Cloud Environments

by

Lichun Zhu

APPROVED BY:

Dr. Gokul Bhandari
Odette School of Business

Dr. Jianguo Lu
School of Computer Science

Dr. Christie Ezeife, Co-supervisor
School of Computer Science

Dr. Robert Kent, Advisor
School of Computer Science

Dr. Joan Morrissey, Chair of Defense
School of Computer Science

September 28, 2011

DECLARATION OF ORIGINALITY

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

ABSTRACT

In this thesis we introduced the Lightweight Coordination Calculus based logic programming approach to the programming models of the Platform-as-a-Service cloud. By using this approach, PaaS based cloud systems will enable cloud application developers to have more options to implement various kinds of programming models for their distributed tasks. We built a prototype framework based on OpenKnowledge middleware because the OpenKnowledge currently is the only framework that fully supports the LCC based programming model. By adding task control and administrative features such as automated task initiation, task status querying, task termination and input/output message channel, we extended the original usage of the OpenKnowledge framework and made it capable of being used to construct PaaS cloud systems. The automation level of the transformed OpenKnowledge framework is improved and its original advantages are retained simultaneously. All of our work reveals the underlying mechanism of the next generation Platform-as-a-Service cloud system which supports logic programming.

DEDICATION

The thesis is dedicated to

those who enlightened me,

and to those who supported me.

ACKNOWLEDGEMENTS

First, I owe my deepest gratitude to my supervisor, Dr. Robert Kent, who guided me as I stepped into the fantastic domain of distributed computation, and provided great patience, encouragement, guidance and support of my exploration. Without his help during a time of handicap, it would have been impossible for me to find a proper thesis topic and proceed in the correct direction.

Besides my supervisor, I would like to thank the rest of my thesis committee: my co-supervisor, Dr. Christie Ezeife, Dr. Jianguo Lu, and Dr. Gokul Bhandari, for their encouragement, insightful comments, and hard questions.

I thank my fellow classmates of the School of Computer Science department: Paul Preney, Lihua Duan, and Xin Wu, for the broad and stimulating discussions of emerging state of art technologies and theories. In particular, I am grateful to Xin Wu for enlightening me with the first glance of a feasible path.

I would like to thank my family: my mother-in-law Yuguang Zhong, my wife Lingru Li and other parents on both sides, for the encouragement and support both spiritually and physically during these busy days and nights. Last, I would send my deep regret to my daughter, Caroline, for my losing a lot of time to fulfill my responsibility as a father.

TABLE OF CONTENTS

DECLARATION OF ORIGINALITY iii

ABSTRACT iv

DEDICATION v

ACKNOWLEDGEMENTS vi

LIST OF TABLES x

LIST OF FIGURES xi

1. INTRODUCTION **1**

2. BACKGROUND STUDY **4**

2.1 Contemporary PaaS Cloud 4

2.1.1 INTRODUCTION OF PAAS CLOUD 4

2.1.2 LIMITATION STATEMENT 5

2.2 LCC and OpenKnowledge Framework 5

2.2.1 INTRODUCTION OF LIGHT WEIGHT COORDINATION CALCULUS BASED MODELING TECHNOLOGY 5

2.2.2 TASK MANAGEMENT AND COORDINATION SUPPORTED BY OK FRAMEWORK 7

2.2.3 LIMITATION STATEMENT OF EXISTING OK FRAMEWORK 8

2.3 Summary and Statement of Research Objectives 9

2.4 Related Work 10

2.4.1 IN THE DOMAIN OF CLOUD COMPUTATION 10

2.4.2 IN THE DOMAIN OF AGENT SYSTEMS 11

2.4.3 IN THE DOMAIN OF GRID AND P2P 12

2.4.4 SUMMARY 13

2.5 Contribution 13

3. OPENKNOWLEDGE SYSTEM ARCHITECTURE AND COORDINATION PROTOCOL ANALYSIS **16**

3.1 Lifecycle of an Interaction 18

3.2 Issues for Enhancing and Extending 25

4. FORMALIZATION 28

5. DESIGN AND IMPLEMENTAION 30

5.1 System Architecture 30
5.2 Design of Task Description Data Structure 31
5.3 Design of GP 33
5.4 Extension made to the OKManager 40
5.5 Enhancements to the Role Allocation Procedure 41
5.6 Enhancements to the Interaction Complete Procedure 41

6. EXPERIMENTAL APPROACH AND RESULTS 43

6.1 Experimental Environment Usage 43

6.1.1 START THE ENVIRONMENT 43

6.1.2 SUBMIT A TASK 47

6.1.3 AUTOMATED TASK ENROLMENTS AND SHOW TASK STATUS 48

6.1.4 TERMINATE A TASK 51

6.1.5 USER I/O MESSAGE CHANNEL VIA THE MESSAGECLIENT API 51

6.1.6 INVOKE A CHILD TASK WITHIN A RUNNING TASK 52

6.2 Performance Analysis 54

6.2.1 PERFORMANCE ANALYSIS VIA REAL TESTING ENVIRONMENT ... 55

6.2.2 PERFORMANCE ANALYSIS VIA SIMULATION 57

6.2.3 SIMULATION OF SEQUENTIAL TASK PROCESSING 58

6.2.4 SIMULATION OF CONCURRENT TASK PROCESSING 60

6.3 Concerns about Dead Locks 65

7. CONCLUSIONS AND FUTURE WORK 66

7.1 Conclusions 66
7.2 Future Work 68

REFERENCES 70

APPENDICES 74

A1. MAJOR ADMINISTRATIVE RELATED METHODS OF EXTENDED OKMANAGER AND OKMANAGERIMPL 74

A2. NEWLY ADDED MESSAGE TYPES 78

A3. TEST DATA COLLECTED FROM REAL EXPERIMENTS 80

A4. SOURCE CODE AND EXPERIMENT DATA DOWNLOAD 81

A5. LCC SPECIFICATION AND EXAMPLE 82

A6. OPENKNOWLEDGE COMPONENT EXAMPLE 85

VITA AUCTORIS 88

LIST OF TABLES

Table 1. Comparison of task management mechanisms. 13

Table 2. TaskDescription related operations 33

Table 3. Major methods of TaskManagerHelper 35

Table 4. Major methods of MessageClient and MessageClientImpl 38

Table 5. Testing environment 43

Table 6. Source code tree of the extended OK framework 44

Table 7. Major administrative related methods of *OKManager* and *OKManagerImpl* 77

Table 8. Newly added message types for task management purpose 79

Table 9. Test data collected from real ex experiments 80

Table 10. Description of files and transcripts 81

LIST OF FIGURES

FIGURE 1. OK SYSTEM COMPONENTS AND THEIR RELATIONSHIPS 16

FIGURE 2. LIFE CYCLE OF OK SYSTEM 19

FIGURE 3. [UML] SEQUENCE DIAGRAM FOR PUBLISHING AN INTERACTION MODEL........... 20

FIGURE 4. UML SEQUENCE DIAGRAM FOR SEARCHING IM AND ROLE SUBSCRIPTION 21

FIGURE 5. UML SEQUENCE DIAGRAM FOR INITIATING AN INTERACTION 22

FIGURE 6. UML SEQUENCE DIAGRAM FOR CHOOSING PARTNERS AND ALLOCATING ROLES23

FIGURE 7. UML SEQUENCE DIAGRAM FOR STARTING AND TERMINATION OF AN INTERACTION 25

FIGURE 8. LOGICAL ARCHITECTURE OF THE PROTOTYPE TASK MANAGER 31

FIGURE 9. UML CLASS DIAGRAM OF TASKDESCRIPTION 32

FIGURE 10. UML CLASS DIAGRAM OF GP 33

FIGURE 11. UML SEQUENCE DIAGRAM FOR I/O REQUEST BETWEEN OKC INSTANCE AND TASK MANAGER 39

FIGURE 12. UML SEQUENCE DIAGRAM FOR I/O REQUEST RELAY BETWEEN OKC INSTANCES OF PARENT TASK AND CHILD TASK 40

FIGURE 13. UPDATED UML SEQUENCE DIAGRAM FOR CHOOSING PARTNERS AND ALLOCATING ROLES 41

FIGURE 14. UPDATED UML SEQUENCE DIAGRAM FOR THE TASK COMPLETING PROCESS ... 42

FIGURE 15. INITIAL RUNNING ENVIRONMENT OF GP_1 45

FIGURE 16. IP AND PORT ALLOCATION OF INITIAL RUNNING ENVIRONMENT 46

FIGURE 17. INITIAL RUNNING ENVIRONMENT OF GP_2 46

FIGURE 18. SCREEN SHOT AFTER TASK “HELLO WORLD” IS SUBMITTED (GP_2)................. 48

Figure 19. Screen shot after task "Hello World" is submitted (GP_1)................. 49

Figure 20. Screen shot of how original OK works with the "dining philosopher" example.. 50

Figure 21. Screen Shot after task "Hello World" invoked child task "Dining Philosophers" (GP_1) 53

Figure 22. Screen Shot after task "Hello World" invoked child task "Dining Philosophers" (GP_2) 54

Figure 23. Response time with fixed number of peers and changing number of roles. 58

Figure 24. Response time with fixed number of roles and changing number of peers. 59

Figure 25. $\frac{M \cdot N_p}{L \cdot N_r}$ ratio vs. the throughput.. 61

Figure 26. Expanded view of figure 25, which shows that the throughput has an upper limit 62

Figure 27. Task submission speed/throughput ratio *V/TP* vs. average response time *T* of all data series with upper bound and lower bound envelope. 64

Figure 28. Exponential regression function of average response time *T* based on Figure 27 65

1. INTRODUCTION

In recent years, a new service model called Cloud computing [Buyya *et al* 2009][Zhang *et al* 2010] has gained considerable interest and undergone rapid development. Within this service model, resources such as CPU and storage capacity are provided as general utilities that can be leased and released by users through the internet in an on-demand fashion. From the perspective of users, the cloud is a kind of virtual sandbox that hides the complexity of management details of internal distributed resources and provide services at different levels: from the infrastructure level, which offers virtual machine service; to the platform level, which offers operating systems and application framework service; and to the application level, which offers specific software utility to end users. In this thesis, we focus on the platform layer service, also called Platform-as-a-Service (PaaS), which offers the service of a computation platform that enables application developers to submit and manage their own distributed tasks to the cloud.

There are many tools and frameworks at this level that have emerged to support distributed data storage and access and software programming. Through a literature review we found that, compared to the advances made in data storage and access measures, progress towards more effective support for programming methodologies offered by existing PaaS frameworks are relatively limited. Due to the distributed nature that cloud computing has, we focus our attention to the concurrent system modeling techniques.

In 2005, Robertson *et al* presented a new modeling technique that is called the Light Weight Coordination Calculus (LCC) [Robertson 2005]. They also presented a middleware framework called OpenKnowledge [PA *et al* 2007] that provides application

developers a flexible way to define complex programming models using LCC and also provides basic support to deploy and execute such applications in a peer-to-peer based overlay network.

We conducted research on exploring how to introduce LCC based concurrent system modeling techniques to the domain of PaaS cloud computation by studying the underlying working mechanism of OK, and found that one major obstacle is its lack of a sophisticated management infrastructure that automates the task deployment, launch of interactions as well as the monitor and task control functionality after the distributed task is launched.

In this thesis, we constructed a task manager prototype framework by providing two extensions to the OK framework to make it cloud ready. First, we extended the OK framework's computation model from the "submit-manual select–subscribe–allocation-run" model to the "submit-proactive select–subscribe–allocation-run" model by introducing a new type of peer (GP) that has the intelligence of detecting and participating newly submitted tasks proactively. Second, we enhanced the task management functionality of the OK framework by adding a task control console to the peer so that user can monitor the execution status of the distributed task and provide intervention to the execution process.

The significance of our research lies on:

1. To the best of our knowledge, we believe that we are the first to introduce formal concurrent system based modeling techniques (specifically LCC) to the domain of cloud computation. It is expected that cloud application developers will

benefit from more selections enabled to design their applications and have more controls on the distributed resources.

2. We provided partial solutions to management and coordination challenges encountered during the construction of the prototype framework. The method we used to solve the challenges can be contributed to the design of the future generation cloud infrastructure that supports above computation models.
3. The open source OK framework coupled with the extensions and modifications presented in our work can be used to support and conduct further research, and we provided a benchmark for comparison against future improvements.

The impact of this work is expected to change how the applications are constructed to utilize clouds. This will be achieved using the new features developed in this thesis that support more complex coordination and negotiation protocols.

The rest of the thesis is organized in this way: Section 2 presents the background study. Section 3 analyzes lifecycle of OK framework and proposes our enhancements and extensions to the framework. Section 4 presents formalization of the task management model we extended. Section 5 presents the detailed design and implementation of our approach. Section 6 demonstrates our experimental approaches and result analysis. Finally, in section 7, we present our conclusions and some opportunities for future work.

2. BACKGROUND STUDY

2.1 Contemporary PaaS Cloud

2.1.1 Introduction of PaaS Cloud

According to [Zhang *et al* 2010], *Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.* From above definition and the study of how existing cloud system works, it demonstrates that two important requirements that cloud management system should solve are the dynamically provision of resources for tasks and how Service-Level-Agreement established via negotiation [Buyya *et al* 2009]. Cloud computation works at three different layers, infrastructure, platform and application. At the level of platform, the computation model delivers a computing platform and/or solution stack as a service, i.e. providing platform layer resources, including operating system support and software development frameworks, which is called Platform-as-a-Service (PaaS). Typical PaaS providers include Google App Engine [Google App Engine], Aneka system [Chu *et al* 2007], Apache HaDoop [HADOOP Project] and Microsoft Windows Azure [Windows Azure].

From the aspect of software development, two factors of major concerns are how data is stored and accessed, and how to design and express a distributed computation within cloud. At the data storage layer, PaaS clouds provide technologies such as distributed file system [Google GFS][HDFS] to provide persistent and durable storage for applications in the cloud. As to how to express a distributed computation within cloud,

the Aneka system [Chu *et al* 2007] SDK enlisted three programming models that are adopted as standards in all other PaaS cloud SDKs:

Task Programming Model: In this model, a distributed application is a collection of independent tasks. The system does not enforce any execution order or sequencing but these operations have to be completely managed by the developer on the client application if needed.

Thread Programming Model: This model provides fundamental component for building distributed applications based on the concept of distributed thread. It allows developers to have finer controls to a single thread.

MapReduce Programming Model: MapReduce [Dean and Ghemawat 2008] is a widely used programming model in PaaS cloud. It provides a standard mechanism to split task into partitions, map them to the worker nodes in the cloud, and then aggregate or reduce the computation result and present it to the end user.

2.1.2 Limitation Statement

Compared to the data storage service methods, the programming models offered by PaaS that developer can choose are still limited. For applications that have complex interacting role relationships, currently there is little way of defining such interaction model at abstract level. This situation gave us the motivation to introduce LCC based interaction model into PaaS.

2.2 LCC and OpenKnowledge Framework

2.2.1 Introduction of Light Weight Coordination Calculus based modeling technology

[Robertson 2005] defines the notion and syntax of the Light Weight Coordination Calculus (LCC), and explains how to use LCC to define the message exchange protocol

among different roles scattered over a p2p network. One can refer to Appendix A5 for detailed specification and example. In the rest of the thesis, we use term "protocol" to represent the interaction model defined using LCC, and use "OK" to represent the OpenKnowledge framework.

The LCC based modeling originates from process calculi [Milner *et al* 1992], Actor model [Agha 1986] and the study of role & social norm based multiple-agent systems [Robertson 2005]. Although LCC is used to describe behaviour of agents in multiple-agent systems, it has been proven through our work to also possess significant power to deal with other domain of applications.

From the aspect of business process standards, there exist two methods of automated arrangement, coordination, and management of complex computer systems, one is orchestration, and the other one is choreography [Peltz 2003].

Characterized by workflow specifications like BPEL [OASIS-BPEL 2007], orchestration is a kind of collaboration, which focuses on a common goal and has a central coordinator that controls the involved participants and coordinates the execution of their different operations. The involved participants do not need to have the knowledge about their position in a higher business process. Only the central coordinator of the orchestration knows this, so the orchestration is centralized with explicit definitions of operations and the order of invocation of the participants.

On the other hand, choreography such as [W3C-WS-CDL 2004] is a collaborative effort focused on a common goal, but there is no central coordinator (at least logically). Each participant involved in the collaboration effort knows exactly when to execute its operations and whom to interact with.

The LCC automatically falls to the choreography category based on its specification. Compared with orchestration based collaboration, choreography possesses the advantages like:

- Fully decentralized nature make it suitable for distributed environment with p2p based fabric,
- Easier to achieve load balance and avoid single point of failure.

However, the decentralized nature of choreography also adds the difficulty to implement this collaboration pattern.

2.2.2 Task management and coordination supported by OK Framework

The OK framework is a middleware that is designed to support deploy, launching and interpreting distributed tasks with interaction model defined in LCC. Taking the advantage of a modeling language LCC that can be used to define the interaction model at abstract level, OK hides the underlying message relay operation to the application developer and provides a nice and neat way for developers to focus their effort on defining: the role of peers, the business logic of each role (implemented as class library called OKC or OpenKnowledge Components) and the way peer interact each other. The standard routine of running a task is (For detailed task lifecycle explanation at API level please refer section 3.1):

1. User publish the interaction model defined in LCC and necessary supportive code defined as OKC package (see Appendix A6 for example),
2. Distributed participants subscribe to the roles of the published interaction model and download the code needed,
3. The OK middleware selects from the subscribers and initiates the task runtime

environment, then handles the control of the task to a selected component called Coordinator,

4. The Coordinator interprets the LCC and controls the message[1] exchange with all selected role participants.

2.2.3 Limitation Statement of existing OK framework

By comparing OK's standard routine of running a task to the internal requirements of PaaS as mentioned in Section 2.1.1, which are dynamically provision of resources for tasks and Service-Level-Agreements established via negotiation. One can see that the OK framework already provides the provision and negotiation mechanism to some extent, which established a base for integrating OK middleware into the infrastructure of PaaS cloud. However, existing OK framework still have limitations that hinders this integration:

Limitation one: limited management supports.

Existing OK framework only offers a small management interface. However, it can only satisfy the management requirements on a single peer. Other than this, existing OK framework does not offer task management supports for users to control and monitor the status of their submitted tasks as well the communication mechanisms between the task manager and its running task, or between one task and another.

Limitation two: the role subscription (step 2 in section 3.1) of current implementation of OK is not automated.

---

[1] One thing need to be noted is that the term "message" mentioned from here on is different to the message mentioned in section 2.2.1. Due of the implementation method as described in later section 3.1, the LCC "message" defined in section 2.2.1 is virtual message that is semantically meaningful within the scope of coordinator's LCC interpreter itself. The "message" mentioned from here on is actual structured data packages that are relayed between peer endpoints.

Limitation three: the coordination method (step 4 in section 3.1) of current implementation of OK is based on centralized model. The distributed coordination mechanisms depicted in [Robertson 2005] are not realized in existing implementation. The centralized coordination model will increase the network traffic and make the coordinator component itself a single point of failure.

2.3 Summary and Statement of Research Objectives

In section 2.1, we analyzed the limitations of the programming models that current PaaS cloud supports, which reveals the significance of enriching cloud programming models using other programming methodologies such as logic programming in the concurrent systems realm. Based on our further analysis in Section 2.2, we found that LCC based logic programming approach is an ideal candidate for PaaS clouds because of its choreography based nature. The underlying design of the OK framework that supports the deployment and runtime management of LCC based tasks already provides some extent of provision and negotiation mechanism that PaaS cloud system requires. This also makes it worthwhile to integrate OK into the PaaS cloud infrastructure. However, major obstacles exist to achieve this integration. We summarized three limitations in section 2.2.3 that reveals our objective in this integration.

Statement of Research Objectives:

The objective of our work is to demonstrate that the LCC based logic programming approach, and its entire set of supportive mechanisms provided by OK framework, can be integrated into the infrastructure of the cloud platform, through enhancing the provision and negotiation mechanism of existing OK framework and extending its task management functionality.

In this thesis research, our practical goal is to construct a prototype system that provides for proving the concepts and establishing benchmarks of behaviour, while also serving as a foundation platform for future research. We focused on solving the first two limitations in section 2.2.3, namely enhancement of the management capabilities and interface, and automation of role subscription. For limitation three, that deals with the low efficiency problem of the centralized coordination mechanism, since it does not affect the runnability of the system we leave it as one major problem to be solved in future optimization research.

2.4 Related Work

We surveyed task management solutions provided by distributed computation systems from different domains. Our goal is to evaluate their pros and cons, and examine if they have unique characteristics that can be referred to and provide comparison in the future evolution of our solution.

2.4.1 In the domain of Cloud computation

Related work includes the Aneka system [Chu *et al* 2007]. As the producer Manjrasoft Ltd. mentioned [Chu *et al* 2007], Aneka is a platform for deploying clouds developing applications on top of it. It provides a runtime environment and a set of APIs that allow developers to build .NET applications that leverage their computation on either public or private clouds. Like OK, it is a middleware that provides a set of APIs that support developers to build their own applications. We find that several of its components have correspondences in OK system. The Aneka Scheduler is actually performing the role that the OK Coordinator does, and the Aneka Executor is doing OKManager functionality (which will be further explained in section 3). The major advantage of

Aneka is that it provides a complete mechanism for security, management functions like task monitoring and accounting, while the major weak point of Aneka is its lack of methods to describe complex coordination between work units running on different Executors at abstract level like LCC to OK.

Another related work is the Apache HaDoop [HADOOP Project], which is an open source software library that allows for the distributed processing of large data sets across clusters of computation and data storage nodes using a simple programming model. This project is widely used in PaaS based Google applications. The Cloudera Enterprise offers a collection of administrative tools to enhance the HaDoop's functionality from the aspect of Authorization Management & Provisioning, Resource Management and Integration Configuration & Monitoring. The major limitation of HaDoop is it only supports the MapReduce [Dean and Ghemawat 2008] model for its distributed computation.

2.4.2 In the domain of Agent systems

A related work in the domain of Agent systems is the JADE (Java Agent DEvelopment Framework) [Bellifemine *et al* 2001]. The JADE framework is running on top of one or many containers (including one main container, additional containers are registered to the main container). In each container can register one or many agents and each agent has a collection of behaviours that defines the agent's task. The main container has two special agents, one is Agent Management System that provides administration and monitor service, another one is Directory Facilitator that provides search and index service. Although JADE provides a management mechanism, like Aneka, one major weak point is that it still does not provide a way to enable user define

the interaction model in an abstract manner. User will have to implement their interaction model through implementing different behaviours and message relay protocols at low level. A later work based on JADE is WADE (Workflows and Agents Development Environment) [Caire *et al* 2008], which provides the support of defining, deploying, executing and fault management of workflow tasks over a network composted of JADE nodes. However, the workflow is expressed at Java class level and it is user's responsibility to deploy the activities to different agents.

2.4.3 In the domain of Grid and p2p

Related work on distributed task management on peer-to-peer network can be seen on [Yan *et al* 2005] and [Yan *et al* 2006], which introduced the p2p-based decentralized workflow management system, known as SwinDeW. This system combines grid (based on GT4) and p2p (based on Sun JXTA) technologies and simulates the enactment of business processes in a decentralized manner. Similar to the way that OK allocates roles to subscribed peers, SwinDeW assigns activities called processes to suitable peers. As workflow execution is coordinated by distributed peers, management and monitoring of workflow execution becomes more difficult. To handle the management task, SwinDeW implements a special, but centralized management peer that communicates with ordinary peers directly to obtain the related information. Compared with other related works, the stated workflow management mechanism provided by SwinDeW is comparatively more complete. What we want to explore is to construct a management mechanism over a fully decentralized p2p network, while it still retains all the management mechanism provided by SwinDeW, and provide support for LCC based tasks.

2.4.4 Summary

In Table 1, we compare the aforementioned three related works with the current OK framework and our proposed OK framework with task management extension. From the table, we state once again that our goal is to enable the OK based collaborate network with enhanced process management, automated process deployment and process enactment along with the full advantage of LCC based interaction model definition. The features listed under the proposed OK with task manager column can be viewed as a wish list that needs to be implemented in our work.

	Current OK	**Aneka PaaS Cloud / HaDoop**	**JADE**	**SwinDeW**	**Proposed OK with Task Manager**
Scope of manageability	Local	Whole network	Whole network	Whole network	Whole network
Formal definition of Interaction Model	LCC at abstract level	No	Through WADE – workflow builder at instance level	XML process definition language	LCC at abstract level
Role distribution	Manual	Work units distributed automatically	Agents distributed manually	Activities distributed automatically	Automatic
Peer selection and process enactment	Automatic	Manual	Manual	Automatic	Automatic
Show task running status and Intervene the task process	No	Yes, through Management Studio / web	Yes, through Remote Management Agent	Yes, through monitoring and administration service	Yes
Decentralized management	Yes (weak)	No	No	No	Yes

Table 1. Comparison of task management mechanisms.

2.5 Contribution

We list two contributions of our work:

1. Proposed a new concept of introducing LCC based logic programming approach into the programming models of the PaaS cloud in order to enable cloud

application developers to have more options to implement various kinds of programming models for their distributed tasks,

2. Built a prototype framework that proves this concept is feasible and serve as a platform to support future research.

Based on the objective stated in section 2.3, in the prototype of the extended OK framework, the working scenario is a pool of peers with computation resources for generic purposes (GP) cooperate each other. To ensure the prototype is runnable and meets the PaaS cloud requirement of integration, we summarized a minimal set of features need to be implemented listed as follows:

- Dynamic join of peers: new GP can join the network dynamically.
- Fully decentralized management pattern. Each GP can act as a management console, which greatly lowers the management burden of the whole system.
- User interface: each GP has a management console that can accept and execute users' input, including task submit, show task status, and terminate a task.
- Task detection: each GP routinely checks from the Discovery Service for pending tasks, decides if it has enough resources to participate the role of the task, and subscribe to the selected role.
- Input/output channel: redirect the input/output requests of roles on distributed peers back to the management console or the parent task.

In [Robertson 2004], the author proposed a broker model as shown as below (5), (6) that act as an extension of LCC. In this model, a client role can request the broker role to send the whole protocol of a task and then initiate and continue as the received protocol. In this way, a client peer can acquire knowledge from a broker peer and then act to

complete a specific task. No further achievements have been found be done in this direction. In our experimental work, we realize a scenario that can be seen as one step toward this direction. Instead of transfer a whole protocol, a task managed by our prototype task manager can invoke a child task that uses a newly published or existing interaction model. The child task can communicate with is parent task and further interact with the end user at the management console via the above mentioned input/output channel.

$$
\begin{array}{l}
a(client(B), A) :: \\
\quad ask(send_protocol(T)) \ \Rightarrow \ a(broker, B) \ \leftarrow \ have_task(T) \ \ then \\
\quad inform(protocol(T, \mathcal{P})) \ \Leftarrow \ a(broker, B) \ \ then \\
\quad \mathcal{P}
\end{array} \tag{5}
$$

$$
\begin{array}{l}
a(broker, B) :: \\
\quad ask(send_protocol(T)) \ \Leftarrow \ a(client(B), A) \ \ then \\
\quad inform(protocol(T, \mathcal{P})) \ \Rightarrow \ a(client(B), A) \ \leftarrow \ protocol_for_task(A, T, \mathcal{P})
\end{array} \tag{6}
$$

In the following sections we will analyze how the OK framework works at underlying level and we will demonstrate how to extend the framework to make it capable to manage distributed collaborating partners.

3. OPENKNOWLEDGE SYSTEM ARCHITECTURE AND COORDINATION PROTOCOL ANALYSIS

As shown in Figure 1, the architecture of the OK system includes many modules which we discuss individually in the following [PA *et al* 2007]:

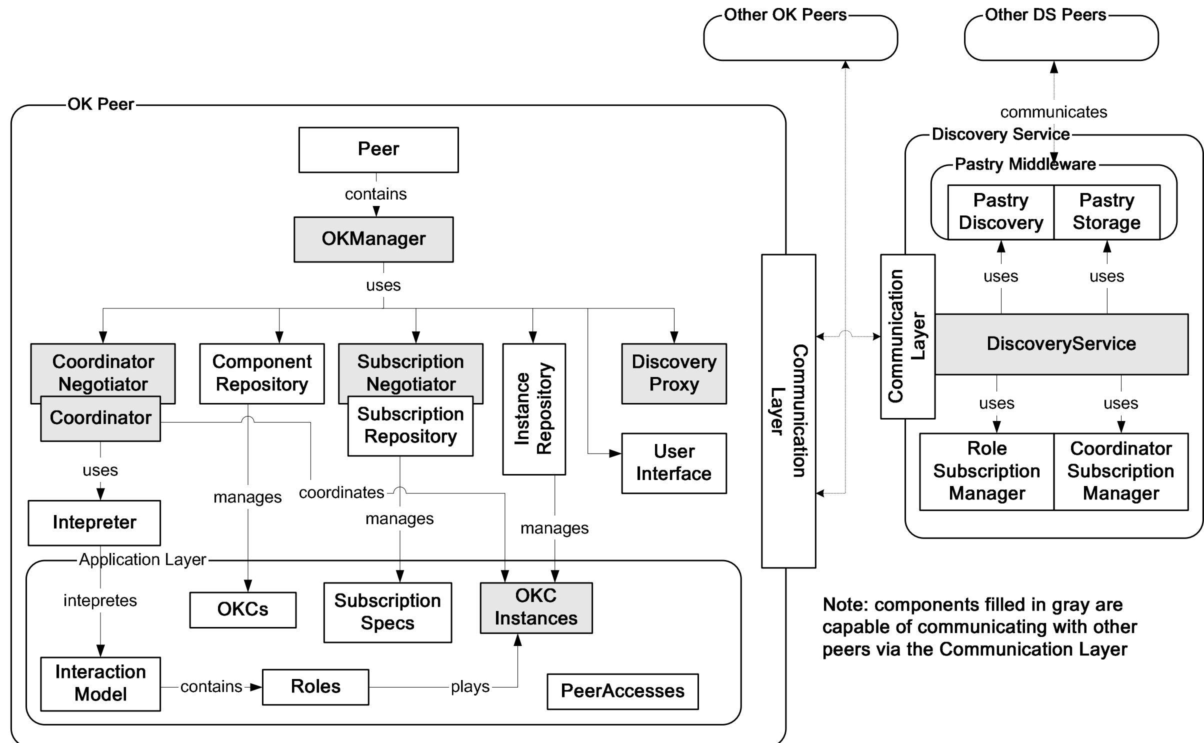

Figure 1. OK system components and their relationships

- Interaction Model (IM)

The IM is a piece of script written in the LCC language which defines how multiple roles collaborate between each other to complete a task. It can be published to the p2p network and can be found by OK peers.

- OpenKnowledge Component (OKC)

The OKC is a class library that implements the service provided by the roles. From the perspective of LCC language, it implements the functionality of

constraints. It is mobile and can be published to the Discovery Service (DS). Peers that want to act as specific roles can find and download proper OKC libraries the DS, and use it to support its business functionality upon interaction initiated.

- OK Manager - *OKManager*

The *OKManager* is the class module that controls all other OK peer modules. Its functions include creating OKC instance from an OKC, delegating constraints received from coordinators to appropriate OKC instance, delegating all the publishing, subscription and search actions.

- Coordinator

The Coordinator is the component dynamically allocated to a peer that interprets the IMs and coordinates the communication with each OKC Instances.

- OKC Instance – *InteractionRunContext*

The component generated after the peer is accepted to play as a specific role in an IM, which contains the pointers to the OKCs needed for solving all the constraints in a specific run of an interaction. It interacts with Coordinator to complete the interaction.

- Discovery & Storage Service (DS)

The DS provides persistent storage for published IMs and OKCs and dynamic storage to their descriptive information. It also stores other information such as available coordinators, roles for published IMs, subscribed candidates for roles etc. Currently it is constructed based on Pastry [Rowstron and Druschel 2001] based p2p framework.

- Interpreter

The interpreter is a LCC parser that interprets the IM by transforming it into a parse tree. It determines which role is acting, which message is to be sent by expanding, traversing and closing branches of the tree. It also interacts with OKC instances to collect results of constraints to determine how to traverse the tree.

Like other peer-to-peer or agent based systems, the OK interaction is completed based on message exchange. The basic component of an OK network that can listen to and handle the received message must implement the *Endpoint* interface. Each *Endpoint* has a unique URI called *EndpointID*. Many above mentioned components including OK Manager, Coordinator, OKC Instance and DS etc are derived from the *Endpoint* that are designated to handle specifically kinds of messages based on their functionality. The module at transport layer that support the message exchange is called Communication Layer.

From Figure 1 one can also see that the manager (*OKManager*) is the class module that controls all the other modular components in a peer.

3.1 Lifecycle of an Interaction

Figure 2 depicts our in-depth analysis about how the whole OK framework works based on the life cycle of an interaction. The lifecycle of an interaction contains eight steps, plus one initial step when a peer joins the network.

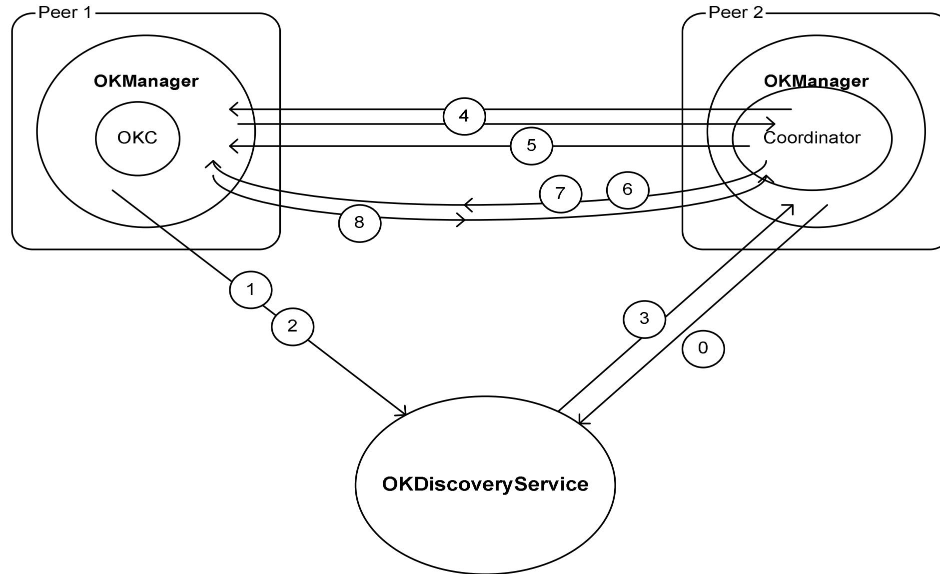

Figure 2. Life cycle of OK system

- **Step 0: Peer Joins To The Network**

This action is taken place between *OKManager* and DS. Whenever a peer joins the OK network, it can choose whether it can be dynamically selected to act as the Coordinator. If it chooses to behave like a Coordinator, it will send the *RequestSubscribAsCoordinator* message to the DS to register itself as a candidate of the Coordinator.

- **Step 1: Publish An Interaction Model**

As shown in Figure 3, the IM publishing action takes place between *OKManager* and DS when a peer in the network decides to publish a new Interaction Model to the network. The *OKManager* of the peer that wants to publish the IM sends a *RequestPublishIMMessage* to DS. The DS publishes this IM to the p2p network, and provides persistent storage to the published IM. Each published IM will be assigned with a unique Interaction Model ID.

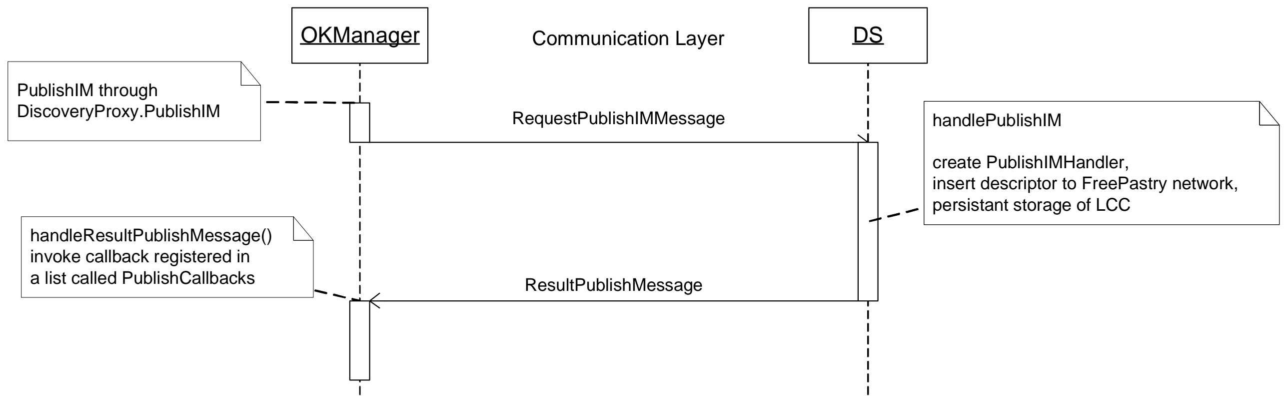

Figure 3. [UML] Sequence diagram for publishing an interaction model

- **Step 2: Search IM and Subscribe To Interaction**

After an IM is published, it is discoverable to all the peers in the network. The peer can inquiry a published IM by search its name. The DS will return the Interaction Model ID, descriptive information and all the roles it has. Then the peer can decide which role it can participate. Upon it decides which role to participate; it uses the sequence described in Figure 4 to subscribe to the role. This interaction is taken place between *OKManager* and DS.

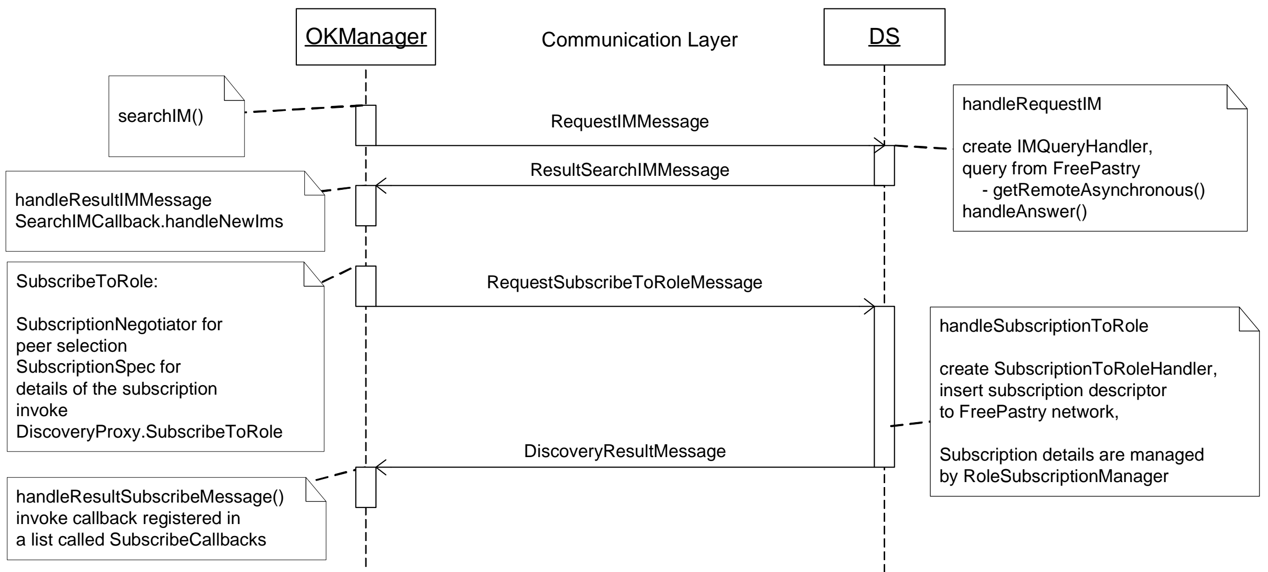

Figure 4. UML Sequence diagram for searching IM and role subscription

When the *OKManager* of a peer decides to subscribe itself to a role of an IM, it invokes its *SubscribeToRole* member function. Inside this function, an instance of *SubscriptionSpec* class is created along with a *SubscriptNegotiator* instance, which is later used for peer selection. The *SubscriptionSpec* instance contains all the subscription information and is sent via the *RequestSubscribeToRoleMessage* to the DS. The subscription information is then made discoverable and is managed by the *RoleSubscriptionManager*.

- **Step 3: Initiate The Interaction – Choose the Coordinator**

As mentioned above, the roles of an IM can be subscribed by different peers over the network. All the subscription information is maintained by the *RoleSubscriptionManager* in the DS. The *RoleSubscriptionManager* checks if all roles of an IM is subscribed and the interaction is ready to start. When the interaction is ready to start, it sends a

StartInteractionMessage to a selected peer that is registered to be a candidate of the Coordinator. After the peer that is selected as the Coordinator received this message, it goes into the bootstrap process (OK uses *BootStrapCoordinator* to handle the Coordinator's bootstrap process).

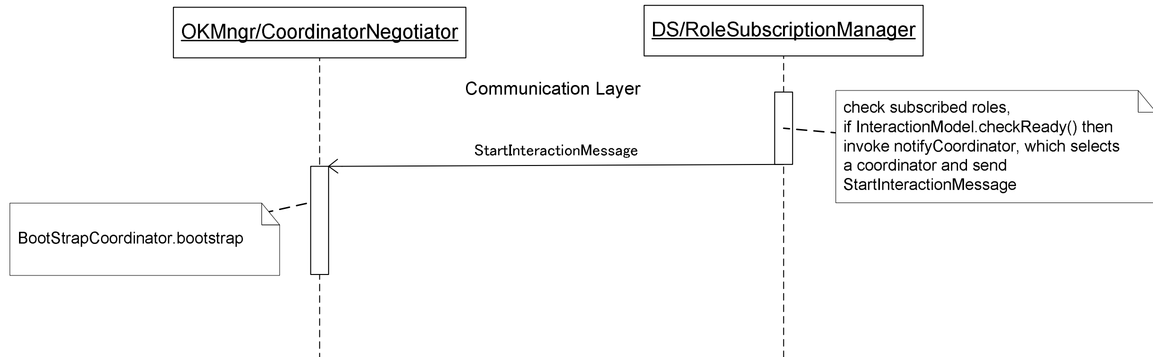

Figure 5. UML Sequence diagram for initiating an interaction

- **Step 4: Choose Partners**

This interaction happens between the *BootStrapCoordinator* and the *SubscriptionNegotiator* of subscribed peers via message *SelectPeersMessage*. It belongs to the peer election process. The *SelectPeersMessage* contains the subscription information of all proposed peers. It is first sent from *BootStrapCoordinator* to each *SubscriptionNegotiator*. The *SubscriptionNegotiator* of each peer uses its own experience to select peers it is comfort to interact with using OK provided trust model interface. The subscription information of selected peers is also packed into a *SelectPeersMessage* and is sent back to the *BootStrapCoordinator.* Each time upon received the *SelectPeersMessage*, the *BootStrapCoordinator* uses *haveAllSelectRequestsReplied* function to check if all subscribed peers are replied. If all subscribed peers are replied, it

uses the list of agreed peers to find out a mutually compatible team of peers to run the interaction. And then starts to allocate the roles to the team of peers.

- **Step 5: Allocating Roles**

The *BootStrapCoordinator* sends *CommittedRequestMessage* to selected peers. When the *SubscriptionNegotiator* of a selected peer receives the message, it can either choose to accept the request, which means that it will join the interaction, or choose to reject the request. If the negotiator selects to accept the request, it will create the runtime context of the role on the peer as well as the diagnostic module that is used for monitoring and auditing purpose. The runtime context is also called the OKC instance that has a new endpoint id. It will be used to interact with the Coordinator during the

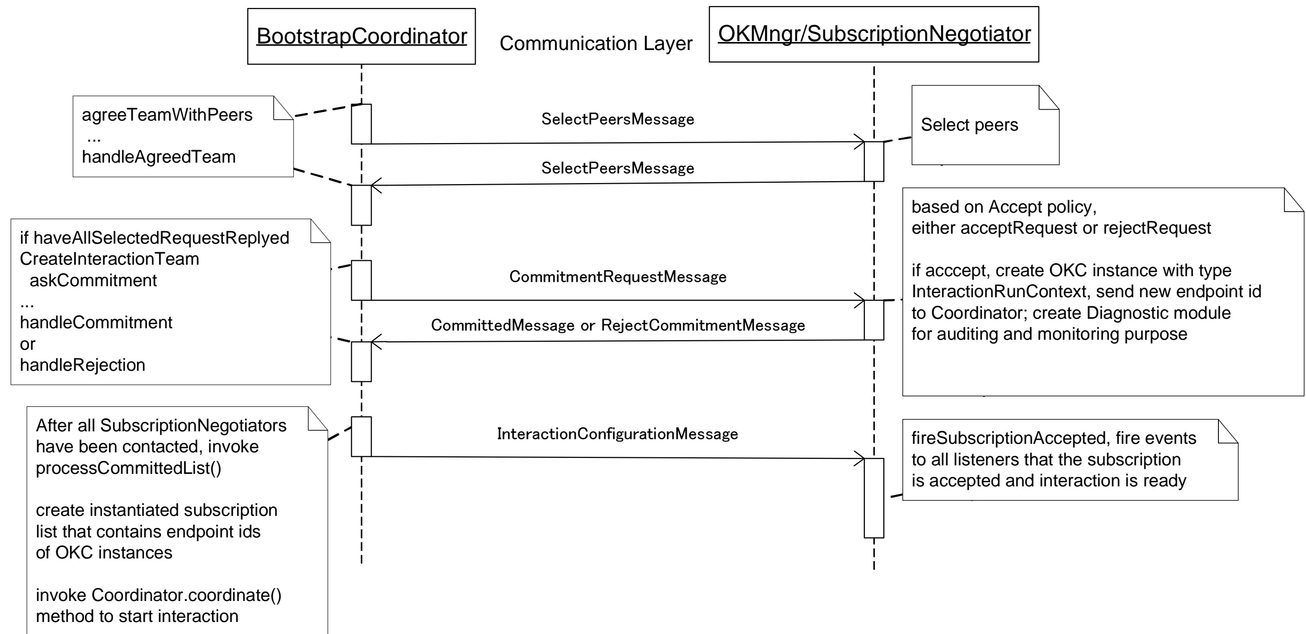

Figure 6. UML Sequence diagram for choosing partners and allocating roles

interaction (solving constraints requested by Coordinator). A *ComittedMessage* or a *RejectCommitmentMessage* will be sent back to the *BootStrapCoordinator* after the peer

accepts or reject the commitment request. As to the *BootStrapCoordinator,* after it has accepted committed message from all the selected roles, it will invoke the *Coordinator's coordinate* member function, which starts the interaction.

- **Step 6: Start Interaction**

The *Coordinator* uses *LCCIntegreter* that is a LL parser generated by [JavaCC] to interpret the LCC interaction model. By executing the IM, the Coordinator determines which role is switched to the current role and which constraint is going to be solved. When the *Coordinator* needs to resolve a constraint, it sends *SolveConstraintMessage* to the *InteractionRunContext* instance of the peer that is allocated with the specific role. The constraint is solved remotely and the result is sent back to Coordinator via the *SolveConstraintResponseMessage*. From here we can see that the current OK kernel uses orchestration to handle the interaction at fundamental level. The Coordinator is the one that actually owns the conversation. Solving the constraint remotely is similar of invoking web services from service providers. The choreography only happens at abstract level. A difference of OK based orchestration versus BPEL based orchestration is that the OK Coordinator is dynamically allocated, which provides room for future improvements on fault tolerance and load balancing optimization etc.

- **Step 7: Interaction Terminate**

The *Interpreter* on the *Coordinator* determines which role is completed. When all roles are completed and there is no next role to execute, the interaction goes into the terminate state and gets into the shutdown process.

- **Step 8: Interaction Feedback**

The shutdown process fires *interactionEnded* event to all its listeners, which causes sending *InteractionCompletedMessage* to *InteractionRunContext* instances of all participated peers and fires up the cleanup process on each peer. The *InteractionCompletedMessage* also contains the status information on the *Coordinator,* hence peers can determine how things going on during the execution of the IM.

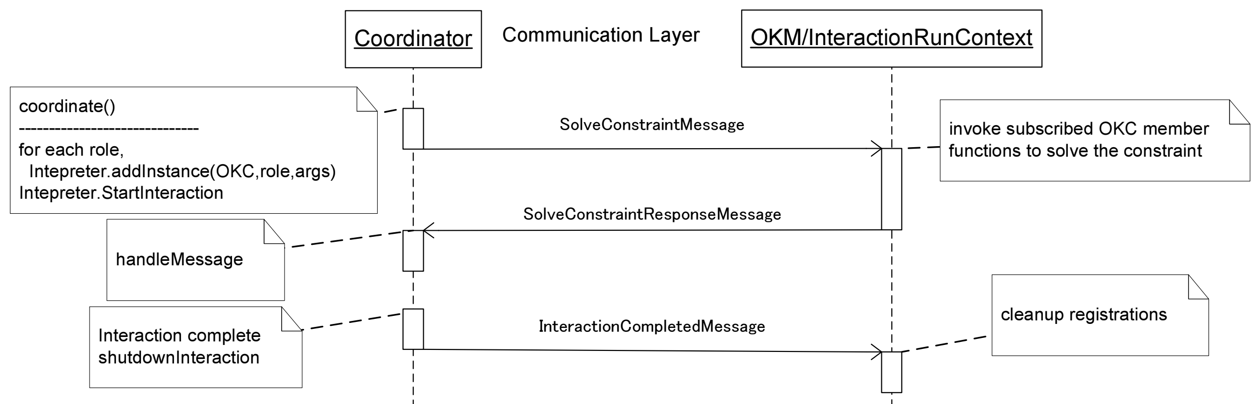

Figure 7. UML Sequence diagram for starting and termination of an interaction

- **Step 9: Learning From Interactions**

This is an optional step. As participated peers can receive abundant information from Coordinator about the execution of the IM, they can use this information to adjust their future behaviour autonomously.

3.2 Issues for Enhancing and Extending

Based on the lifecycle analysis of the existing OK framework and the proposed features mentioned in section 2.5, we proposed two extensions and three enhancements to the OK framework:

- Introduce the concept of task coupled with a new data structure called

TaskDescription to the extended OK framework

A task represents one execution of an interaction model. It uses a *TaskDescription* instance to record all of the information of its internal state. Each *TaskDescription* has a unique task id. It contains the reference to the interaction model and the descriptive information of the required collection of OKCs. The *TaskDescription* instance retains all the runtime status of the lifecycle of a task from pending, running to termination. It is publishable and is discoverable by participant peers.

- Introduce a new component to the framework called Generic Peer (GP)

The GP component acts at the top most layer of a peer in the p2p environment. At one hand, it provides a management console to handle the end user's input/output requirements, which includes a command parser to interpret task submit, task status query and terminate task commands, and also provides interface for user to provide input for a participant of running task and display output information of a remote task participant to the management console. On the other hand, the GP routinely inspects the DS and tries to find pending tasks in which it can participate potentially. Upon finding a matching task, the GP selects a role based on its own resources and subscribes the role to the DS, and let the OK framework to select and execute the interaction.

- Enhance the task management mechanism to the existing control manager (*OKManager*)

The management functionalities of above mentioned GP is supported by an extended *OKManager*, which contains an enhanced message relay interface. Certain

types of messages for the administrative purpose are added. Detailed message types are defined in the following categories: task publish, task status query, updating task status, task termination and inter-task communication.

- Enhancements added to the coordinator allocation procedure (Step 3)

Upon the coordinator of an interaction is selected, The *TaskDescription* need to be updated to reflect the allocation of the Coordinator.

- Enhancements added to the role allocation procedure (Step 5)

Upon a role is allocated to the subscriber of the interaction of a task and the OKC instance is created, the *TaskDescription* need to be updated to reflect the creation of the OKC instance.

4. FORMALIZATION

We define the distributed task as

$$T = \{T_{id}, p_m, R_s, P_s, RP\} \tag{7}$$

Where T_{id} is the unique identifier of the task, p_m is the peer from which the task is submitted; R_s is the set of roles that the task defined, $R_s = \{r_1, r_2, \ldots, r_n\}$; P_s is the set of peer variables that represent the peer for the role to run, $P_s = \{P_1, P_2, \ldots, P_n\}$; RP is a subset of $R_s \times P_s$, which contains a set of tuples like $(r_1, P_1), (r_2, P_2),$ that we call agents.

We use T' to represent the instantiated task, i.e. the task after deployed to the network that is under running state.

$$T' = \{T_{id}, p_m, C, R_s, P_s', RP'\} \tag{8}$$

Where the T_{id} and p_m are same as above, C is the id of the allocated coordinator for the interaction, R_s is the same set of roles as defined above (We presume each role contains all the information at implementation level, which is defined by OKC class library. When a role is deployed to a peer, the OKC is deployed to the same peer accordingly), P_s' is the set of peer ids that are allocated to the task. $P_s' = \{p_1, p_2, \ldots, p_n\}$, we use lower case characters to represent id of actual peers that are constants. RP' is a subset of $R_s \times P_s'$, which contains a set of tuples like $(r_1, p_1), (r_2, p_2),$ that we call instantiated agents.

After a task is submitted from GP running at peer p_m, its roles will be automatically subscribed by a group of listening GPs. The task enrollment is the process of instantiating $P_i \in P_s$ to $p_i \in P_s'$ and instantiating $(r_i, P_i) \in RP$ to $(r_i, p_i) \in RP'$. The difference between the original OK framework and the extended OK framework is that to the latter, the instantiating process is automatic.

A role r_i of task T can decide whether to redirect user input/output request to the task's management console p_m or handle the I/O request on local p_i, depending on if it uses provided API to handle the user I/O request. If the role decides to redirect user's I/O to the task's management console, the user I/O request that generated at $(r_i, p_i) \in RP'$ will be relayed to p_m via the extended message interface of *OKManager* and the management console at p_m will act as an I/O broker to collect input or display output to the end user. If the task $T' = \{T_{id}, p_m, C, R_s, P_s', RP'\}$ is the child task of another task $T'_0 = \{T_{id0}, p_{m0}, C_0, R_{s0}, P_{s0}', RP'_0\}$, when the message of I/O request is relayed to the management console of T' at p_m, it will be cached in a message queue instead of being processed. The running instantiated agent $(r_k, p_k) \in RP'_0$ that belongs to the parent task can inspect the cached I/O request and either process the request and send the input back to $(r_i, p_i) \in RP'$, or it can relay the I/O request to the parent task's own management console at p_{m0} hence a chain of message relay channel is formed and user at the management console of the parent task can control the execution processes of both the task and its child tasks.

5. DESIGN AND IMPLEMENTAION

5.1 System Architecture

The proposed system architecture of the task management system is shown in Figure 8. The entire system contains a collection of Generic Peers. Each peer itself is a Java application running on the same or different machines.

After a GP is launched, it automatically joins the network. All GPs are running based on the same code base that enables them either to be a management console or a role subscription listener. Therefore an end user can submit new tasks and query task runtime information at the management console of any GP. A GP can subscribe to one or many roles of one or more tasks, depending on its own capability such as computation power, resources etc. After it subscribes to the roles, it participates in the interaction lifecycle depicted in section 3.1. From here on, we call the GP from which task *T* is submitted as the task manager of task *T*, and other GPs that participate with task *T* as participant GPs of *T*.

The whole system is implemented using Java programming language (JDK 1.5 or above).

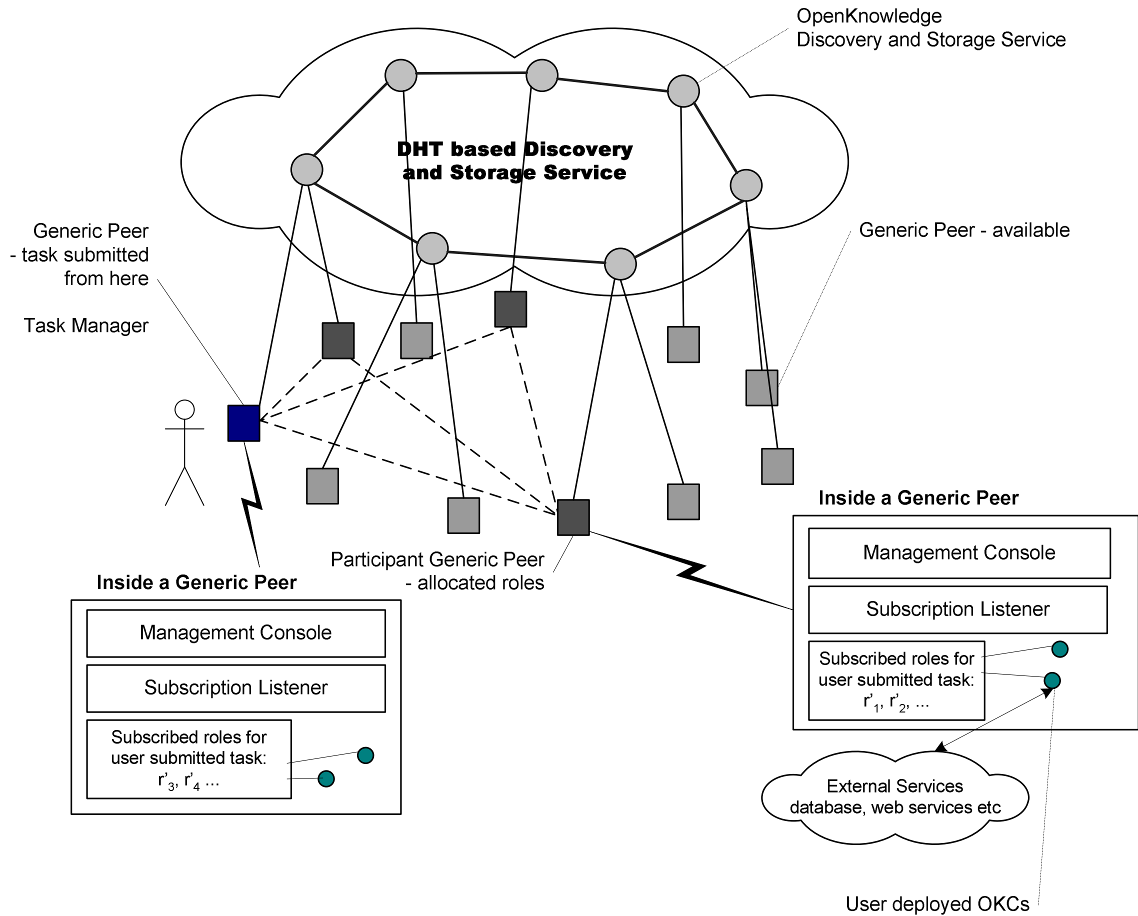

Figure 8. Logical architecture of the prototype task manager

Below is the detailed design of the task management system.

5.2 Design of Task Description Data Structure

The *TaskDescription* data structure is used to record the status of a submitted task. Each *TaskDescription* instance retains all runtime status of the lifecycle of the task from pending, running to termination. It is publishable and is discoverable by participant peers.

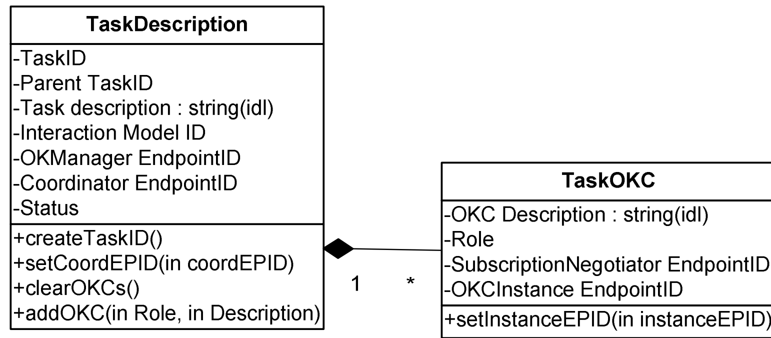

Figure 9. UML Class diagram of TaskDescription

TaskDescription related operations are defined as follows:

Operation	Description
Create task	The *TaskDescription* instance is created upon a task is submitted. Upon creation, the status is initiated as *PENDING*, and the set of *TaskOKC* reflects the minimal set of required OKCs to support the running of the task. At this time, because the interaction is not started, the Coordinator *EndpointID* is null, the OKC Instance *EndpointID* of each *TaskOKC* is also null.
Publish task	Upon task submission, its search criteria and the *EndpointID* of the *OKManager* are published to DS and are discoverable. The effective *TaskDescription* instance is stored at the *OKManager* from which the task is submitted.
Search task	Query the DS by using criteria strings, obtain the *OKManager* Endpoint from which the task is submitted, and then query from the *OKManager*
Update task status	During the lifecycle of a task, the *TaskDescription* instance is always updated to reflect the current task status: ● The Coordinator *EndpointID* will be updated when the interaction is ready to start, with the selected Coordinator's *EndpointID*; ● The corresponding *TaskOKC*'s OKC Instance *EndpointID* will be updated when the OKC instance is created. ● The status of the *TaskDescription* will be changed to *RUNNING* when interaction is started.

	● The status of the *TaskDescription* will be changed to *COMPLETE* when interaction is completed After the *TaskDescription* instance is updated, it will be written back to the list of running task of the *OKManager* from which the task is submitted. When the task is completed, the instance will be taken away from the list of running tasks of the *OKManager*.

Table 2. TaskDescription related operations

5.3 Design of GP

The Generic Peer is the top level program of the peer application that runs as an autonomous peer. It contains a group of classes either added to the OK kernel or extended from the existing classes of the OK kernel. The class diagram of the Generic Peer can be referred to as Figure 10.

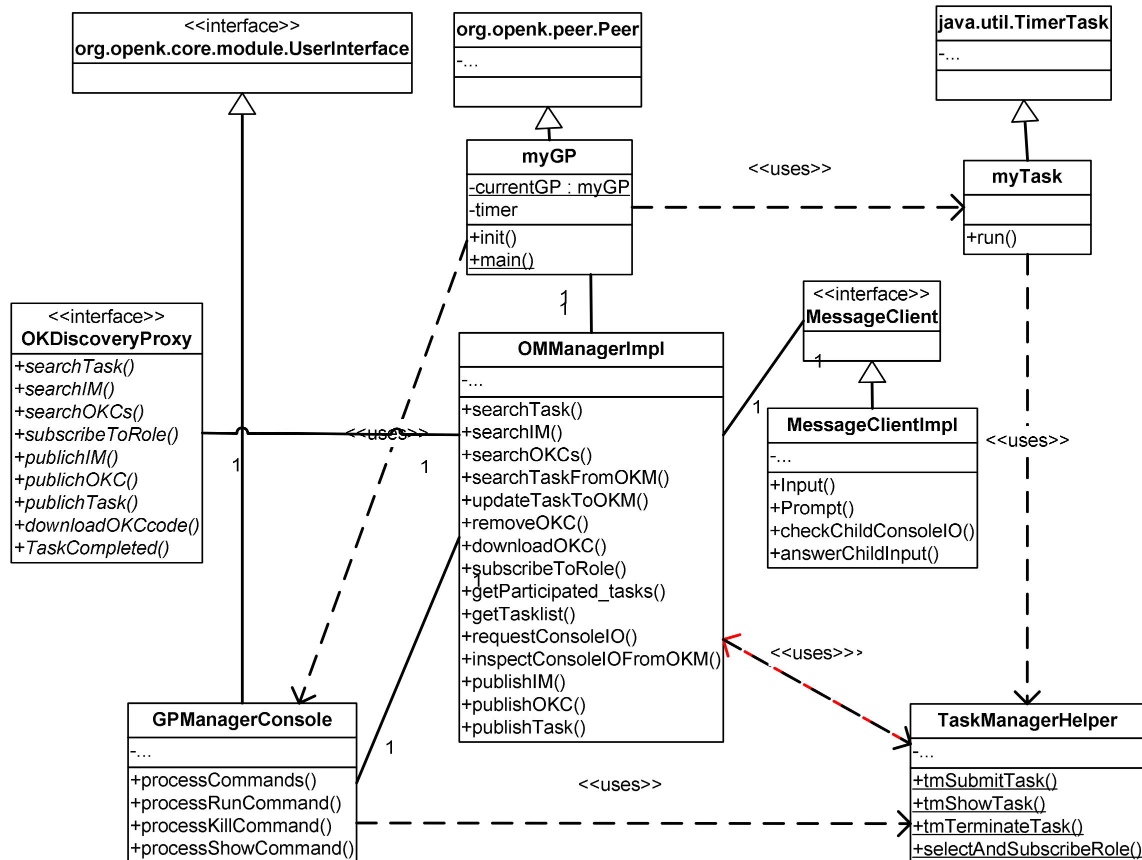

Figure 10. UML Class diagram of GP

The *myGP* class is a new class introduced as the main entrance of the peer application.

It has two functionalities. The first function is to use a timer to schedule a timer task that checks the DS regularly in order to find and attempt to participate newly submitted tasks, by invoking method *TaskManagerHelper.selectAndSubscribeRole*. The second function is to use class *GPManagerConsole* to construct a user interface, interpreting user submitted administration commands.

In order to make the extended code more manageable, we introduced the *TaskManagerHelper* class that provides a group of static functions that are used for task managements. All these functions provide synchronized interface to their callers. Major methods include:

Method	Description
tmSubmitTask	Submit a new task. Parameters: mgr – reference to *OKManagerImpl* instance taskname – string of task name, im – string of the interaction model defined in LCC, okcs[] – array of *OKCDescription*, ptid – task id of parent task if has one Returns: A *TaskDescription* instance
tmShowTask	Query the task running status and print out. Parameters: mgr – reference to *OKManagerImpl* instance taskname – string of task name
tmTerminateTask	Terminate a task. Parameters: mgr – reference to *OKManagerImpl* instance tasked – id of the task to be killed

	force – Boolean value to specify if the kill is a forced kill
selectAndSubscribeRole	Inspect a newly submitted task, select proper role and subscribe to the role. Parameters: mgr – reference to *OKManagerImpl* instance td – the *TaskDescription* of the task to be inspected Returns: A *SubscriptionSpec* instance – the data structure that records the subscription of a role

Table 3. Major methods of TaskManagerHelper

The pseudo code for task submission is as follows:

```
Procedure tmSubmitTask (ManagerPeer, TaskName, IM, OKC[], parentTid)
   returns TaskDescription
Begin
   T := new TaskDescription(generateTaskID(), TaskName);
   T.okmanagerEpid := ManagerPeer.EndPointID;
   Publish IM to DS if IM not published, set T.imid: = id of published IM or existing IM;
   For each okc in OKC[]
   Begin
            Publish okc to DS if okc not published; register okc to T's TaskOKC list;
   End;
   ManagerPeer.TaskList.add(T); // Register T to local list of submitted tasks
   Publish T to DS;
   Return T;
End
```

The procedure for display task status is fairly simple, the pseudo code is:

```
Procedure tmShowTask (ManagerPeer, taskname)
Begin
      tset[] := searchTaskFromDS;   // get list of published tasks
      For each t in tset[]
      Begin
            TaskDescription tdescr := searchTaskFromOKM(t);  // get task detail from task
                                                              // manager
            Print(tdescr);
```

```
        End;
    End
```

The pseudo code for task termination is as follows:

```
    Procedure tmTerminateTask (ManagerPeer, taskid, isforce)
    Begin
        tset[] := searchTaskFromDS;   // get list of published tasks
        For each t in tset[]
        Begin
            If t.taskid = tasked Then
            Begin
              M := createTaskCompletedMessage(t, force);
              Send M to t's Coordinator;
              /* Upon received M, the coordinator will perform all the resource release works
                and send InteractionComplete messages to all the participants of the task */
            End;
        End;
    End
```

The pseudo code for task enrollment is as follows:

```
    Function selectAndSubscribeRole (ManagerPeer, T) returns SubscriptionSpec
    Begin
      IM := searchIMFromDS(T.taskname);
        Select role to subscribe based on IM's role semantics and subscription status of
        T.TaskOKC[];
        OKCDescription okcdes := searchOKCFromDS(selected T.taskOKC);
        Download OKC code from DS and save it to local OKC storage;
        // Subscribe to the selected role from DS
        SubscriptionSpec s := subscribeToRole(selected role, okcdes);
        // register endpoint id of the SubscriptionNegotiator of selected role
        T.taskOKC.subscriberEPID := s.subscriberEPID;
      Return s;
    End;

    tset[] := searchTaskFromDS;   // get list of published tasks
    For each t in tset[]
```

```
Begin
    TaskDescription tdescr := searchTaskFromOKM(t);  // get task detail from task manager
    SubscriptionSpec s := selectAndSubscribeRole(self.mamager, tdescr);
    If sub <> null then
    Begin
      // update task subscription information back to task manager
      updateTaskDescriptionToOKM(tdescr, s);
      Register tdescr to local list of participated tasks;
    End;
End;
```

It is possible to consider several algorithms to deal with the role select and subscription problem. In this thesis we have selected simple algorithm to let participant GP decide which task and role to subscribe. The GP only considers two factors to decide the role subscription to ensure that a task can be initiated upon minimal running criteria has been reached.

- If the maximum number of subscription has reached;
- If the minimal number of requested subscriptions of a role has reached.

More sophisticated selection algorithms that consider load balance and performance optimization will be introduced in the future versions.

The *MessageClient* interface and its implementation *MessageClientImpl* is the client API that provides synchronized interface for inter-task and task/manager communication. It contains the following 4 methods:

Method	Description
Input	Redirect user input request to the task's management console in order to get user's input. Parameters: prompt – string to be displayed to the user

	defaultval – string of default value to be displayed to the user Returns: Input string provided by end user
Prompt	Redirect output request to the task's management console. Parameters: prompt – string to be displayed to the user
checkChildConsoleIO	Called by parent task to contact the management console of child task to get the next cached I/O request message. Parameters: childtsk – *TaskDescription* of child task Returns: Cached *RequestConsoleIOMessage* message
answerChildInput	Send response to the role of child task peer who sent the I/O request message. Parameters: origReqMsg – original *RequestConsoleIOMessage* message send by role of child task ret – string to be returned

Table 4. Major methods of MessageClient and MessageClientImpl

Figure 11 displays the time sequence of how the *Input* and *Prompt* methods work between an OKC instance of a role and the task's management console. The Task manager's *OKManager* acts as a server by responding I/O requests sent from OKC instance of participant GPs.

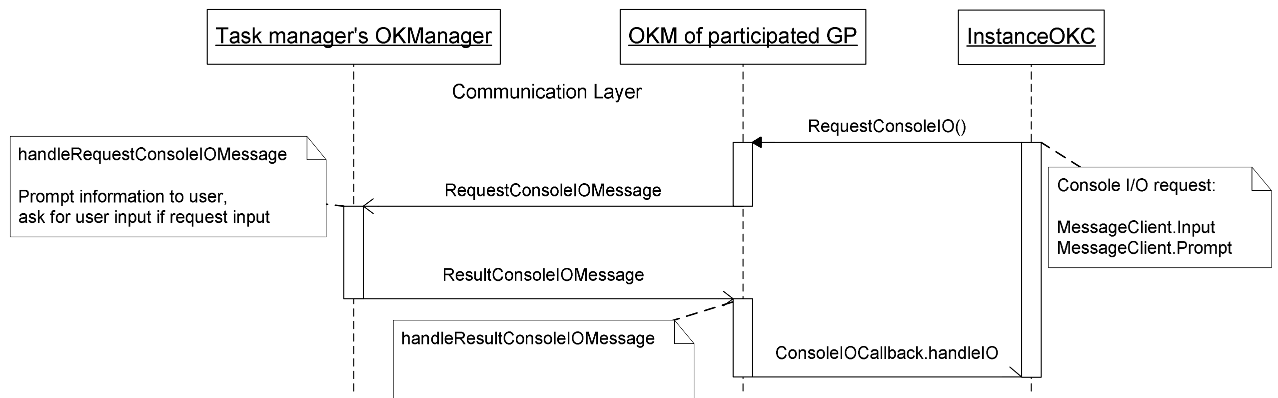

Figure 11. UML Sequence diagram for I/O request between OKC instance and task manager

Figure 12 displays the time sequence of how the *checkChildConsoleIO* and *answerChildInput* methods work between OKC instances of parent task and child task. The inter-task I/O request process uses simplified producer/consumer design pattern. Like Figure 11, the participant GP of child task send I/O request via its *OKManager* to its task manager's *OKManager*. Instead of generating user interface and process the I/O request, the task manager of child task noticed that the request is sent by a child task and simply caches the request in its local queue, hence the request will be hold and wait for the inspection & process request sent from parent task. The OKC instance of a participant GP that belongs to the parent task can initiate a request to inspect its child task's I/O requests. The inspection request is sent to the task manager of child task. The I/O request is then de-queued and returned to the OKC instance of parent task. One thing to be noted here is that the result message of the I/O request is sent back directly from the participant GP of parent task to the participant GP of child task, and the response message *ResultColsoleIOMessage* must retain the original request handler information so that the *OKManager* of child task's participant GP can find the matching callback function to

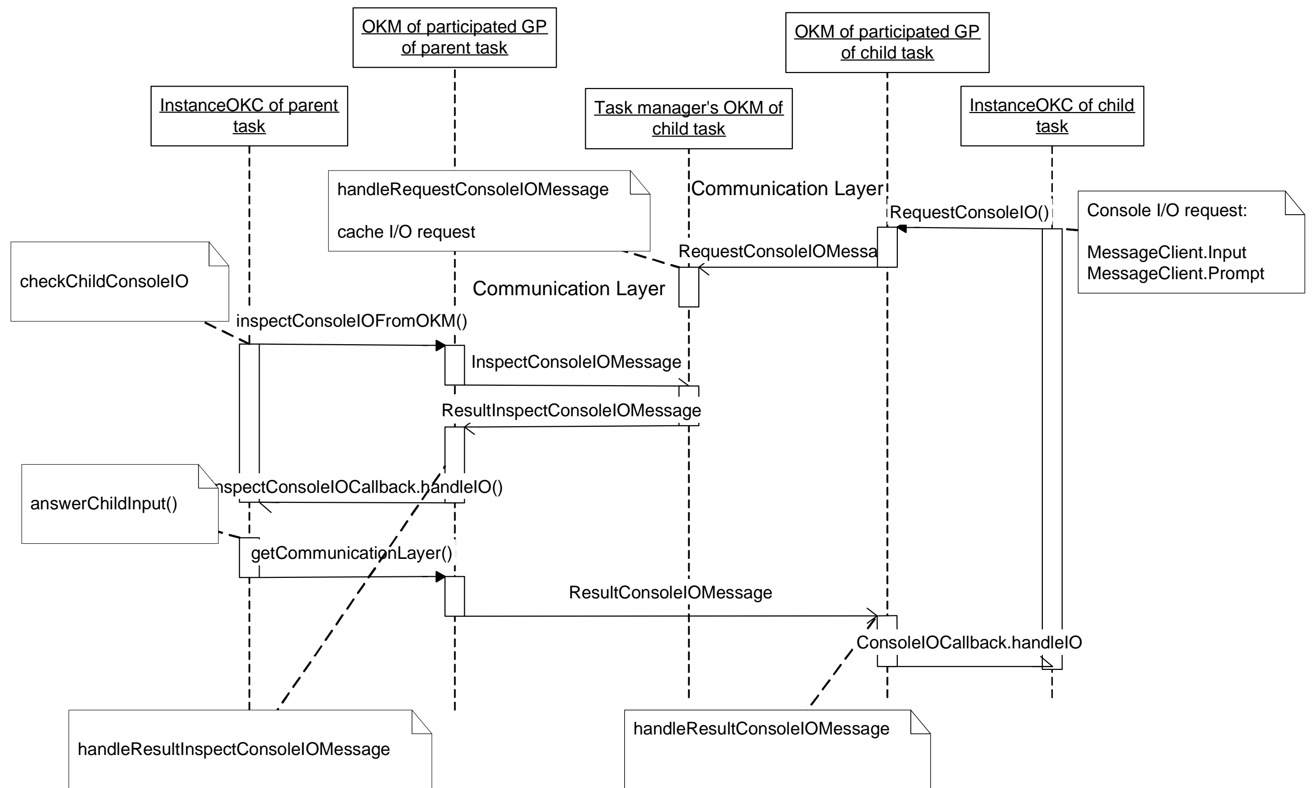

Figure 12. UML Sequence diagram for I/O request relay between OKC instances of parent task and child task

handle the returned message. From Figure 12 one can also notice that the top-level API that directly called by OKC instance uses synchronized pattern and the underlying communication between different *OKManagers* are working under the asynchronous mode.

5.4 Extension made to the OKManager

Both *TaskManagerHelper* and *MessageClientImpl* class uses the extended *OKManager* interface to complete their functions. We extend the management interface and its implementation to handle the task management and I/O redirect functionalities. The extended functionalities include a collection of methods that are used asynchronously based on listener design pattern and message relay between peers. Major

extended methods can be referred to from Appendix A1, and the detailed description of messages used by these methods can be referred to from Appendix A2.

5.5 Enhancements to the Role Allocation Procedure

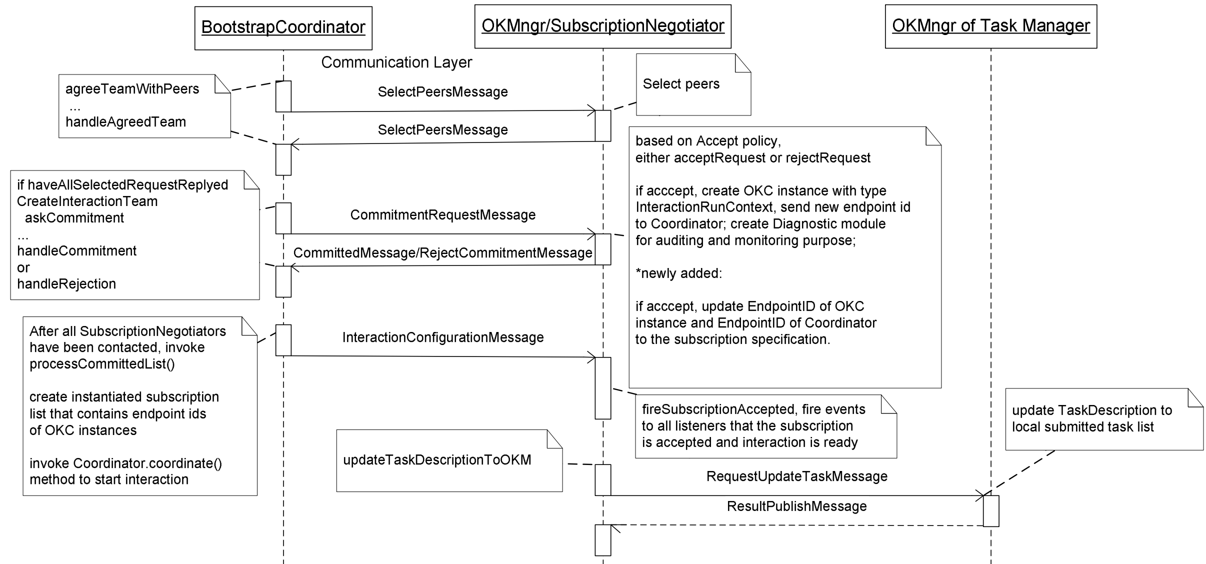

Figure 13. Updated UML sequence diagram for choosing partners and allocating roles

In order to keep the task status updated, we modified the role allocation procedure by introducing an update task description message relay operation to the time sequence. Updated time sequence diagram is shown as Figure 13.

5.6 Enhancements to the Interaction Complete Procedure

Figure 14 demonstrates how message propagates from the GP that sends the *TaskCompleteMessage* to the task's *Coordinator* and then sends to the task manager and all participated GPs. This sequence is added to the diagram of starting and termination of an interaction described in Figure 7.

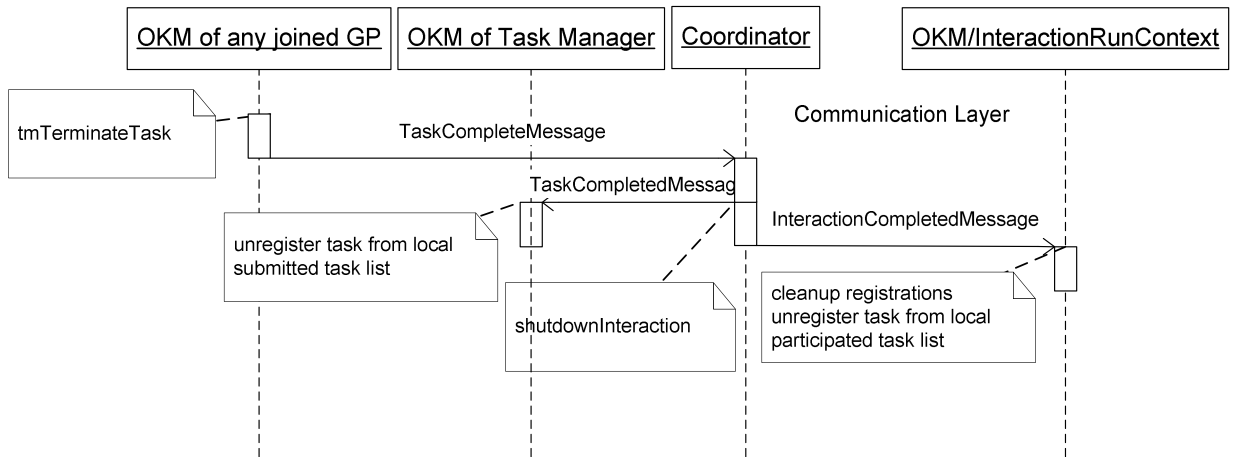

Figure 14. Updated UML sequence diagram for the task completing process

6. EXPERIMENTAL APPROACH AND RESULTS

We discussed all aspects of the experimental work involved in this thesis. In the following subsections we first demonstrate the use of the prototype framework in the order of: the experimental environment, task submission, task enrollment, task termination, and message channel and parent/child task interaction, then discuss the performance analysis based on the experiment conducted on real environment and experiment conducted via simulation.

6.1 Experimental Environment Usage

6.1.1 Start the Environment

We construct the testing environment on two machines as shown in Table 5:

	Machine 1	**Machine 2**
Configuration	CPU: Intel T5670 Duo CPU 1.80GHz MEM: 3GB OS: Windows Vista Ultimate Java: JDK 1.60	CPU: MEM: 768MB OS: Windows XP SP2 Java: JDK 1.60
	LAN: 100Mbps LAN	
Usage	Discovery Service-1st node, GP_1	Discovery Service-2nd node, GP_2

Table 5. Testing environment

The source code of the prototype can be downloaded from the SVN server described in Appendix A4 or be requested via the author's email. Table 6 displays the source code tree of the extended OK framework:

./startDiscovery.cmd or ./startDiscovery.sh	File for launching the Discovery Service. Files with extension ".cmd" are for WINDOWS platform. Files with the ".sh" extension are for Linux platform. Class *org.openk.service.discovery.StartDiscoveryAndStorage* is the main entry.
./startGP.cmd or ./startGP.sh	File for launching the GP application. Class *org.openk.core.tm.impl.myGP* is the main entry.

./startOK.cmd or startOK.sh	File for launching the original OK Manager. We still use this application to build OKC packages or do some testing work. Class *org.openk.core.management.impl.OKManagerImpl* is the main entry.
./build/	Folder for the destination of the compiled files
./config/	Folder for configuration files, frequently used files include: *defaults.properties*: main resource file for OK framework. *logging.properties*: resource file for log4j configuration, used to set the logging preference.
./FreePastry-Storage-Root/	FreePastry generated folder for cached files, used by DS.
./lib/	Third party library (jar) files that should be added in the Java classpath.
./log/	Directory of log files.
./res/	User interface related resources for testing application.
./src/	Folder for all source code files of OK framework. We made changes to the following three sub folders.
./src/discovery/	Source code for Discovery Service. Changes are made on files under this folder for new publishable resource types.
./src/src/	Source code for the OKManager and GP client application. Most of the extensions are added to under the *org.openk.core.tm* namespace.
./src/storage/	Source code for persistent storage of published LCC, OKCcode used by DS. Changes are for the purpose of improving the system stability by upgrading the version of FreePastry based p2p communication layer from 2.0b to 2.1.
./gettingstarted/	Folder of applications for demonstration and testing purposes.

Table 6. Source code tree of the extended OK framework

Figure 15 displays the initial running environment of the first testing machine. We can launch the Discovery Service by running the batch command *startDiscovery.cmd*. After the DS is running, it listens at port 6678 for requests sent from other DS nodes and listens at port 7000 for requests sent from underlying GPs. After the *DS* application is launched, we launch the GP application by executing command *startGP.cmd*, which uses

the IP address of the pre-launched DS as its bootstrap host, 7000 as its bootstrap port and listens at port 4000 (configurable in resource file *config/defaults.properties*) for incoming request sent from DS or other GPs.

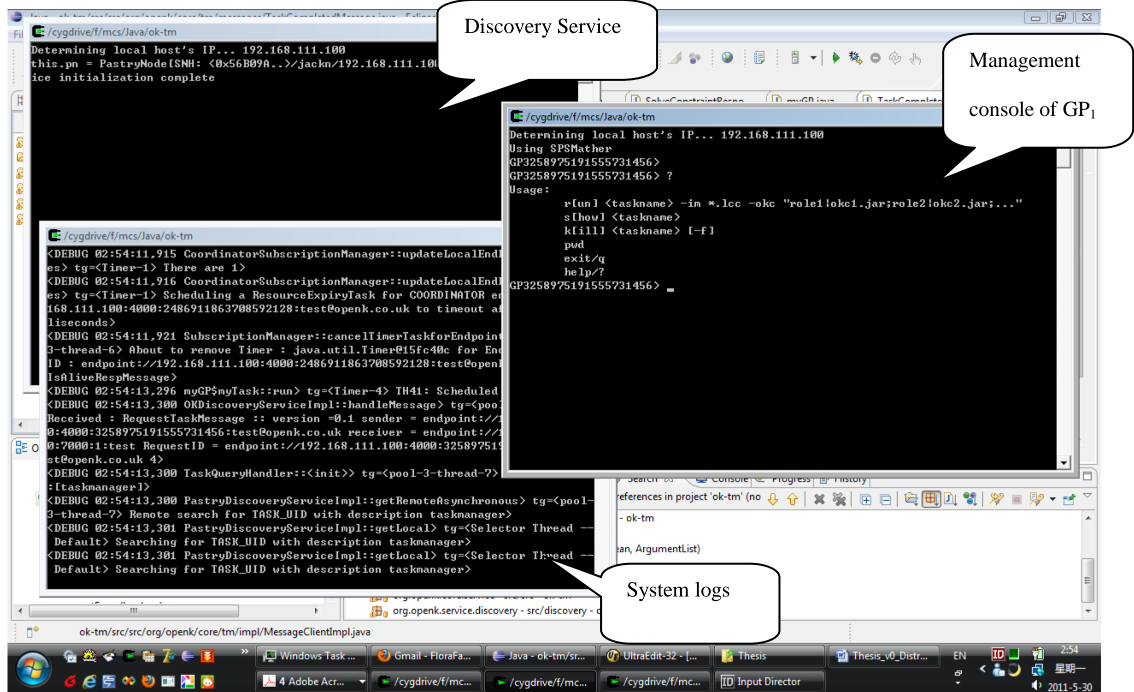

Figure 15. Initial running environment of GP_1

We can use the same steps to launch the DS and GP on the second machine. One difference is that in order to construct a single Discovery Service ring, the second DS should use the first DS as its bootstrap node. The complete runtime configuration is shown in Figure 16.

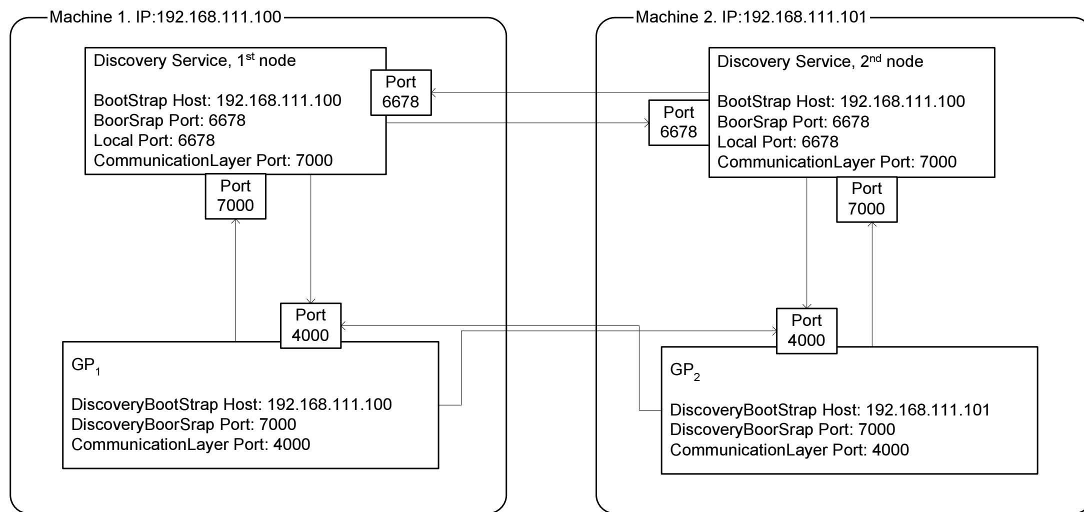

Figure 16. IP and port allocation of initial running environment

The second GP can also register itself to the first DS directly, which saves one DS node and demonstrates that one DS can accept the registration request from multiple GPs. The initial running environment of the second testing machine is shown in Figure 17.

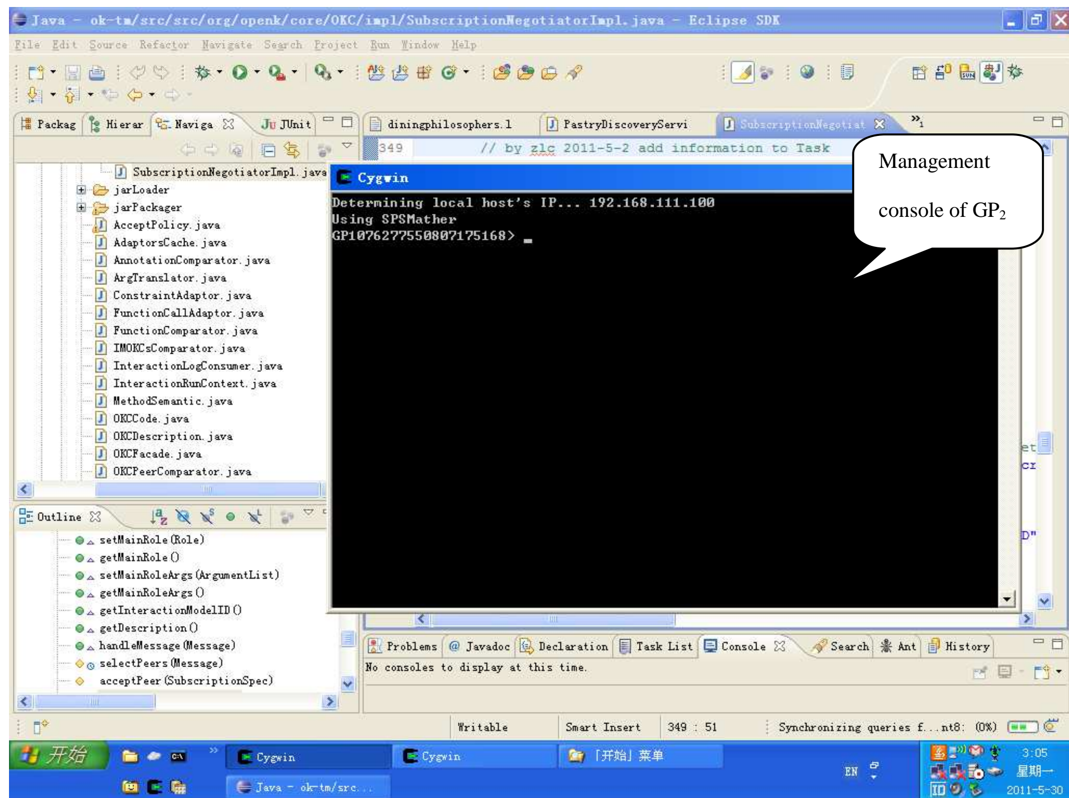

Figure 17. Initial running environment of GP_2

6.1.2 Submit a Task

Figure 18 displays what happened after one submits a testing application "Hello World" from GP_2. The "Hello World" application is provided by the original OK framework as an example. The example command is:

```
run hello -im ./gettingstarted/lcc/helloworld.lcc  \
        -okc "peerResponder | ./gettingstarted/bin/ResponderOKC.jar;  \
            peerGreeter | ./gettingstarted/bin/GreeterOKC.jar"
```

In which "*hello*" is the name of the task and will be used as the name of the published interaction model as well. The file "*./gettingstarted/lcc/helloworld.lcc*" after "*–im*" option contains the specification of the interaction model defined in LCC. Items specified after the "*-okc*" option are the OKC packages developed to support the interaction. Items are delimited by semicolon. For each item, the string before the "|" is the name of the role that the OKC is designed for, and the path after "|" is the path to access the OKC package in specialized format. One can use the OK Manager tool to construct the OKC package by referring to the "Creating and Publishing OKCs" section in [OpenKnowledge Manual].

Figure 18. Screen shot after task "Hello World" is submitted (GP_2)

After the command is submitted, it will pass through a serial of publishing steps depend on whether the IM or OKCs' availability in DS. A task ID will be displayed to user in the format of "<taskname>/<unique serial number>".

6.1.3 Automated Task Enrolments and Show Task Status

After the task is submitted, its pending state will be captured by the registered GPs within a short time interval. We use the "*show*" or "*s*" command to display status of all

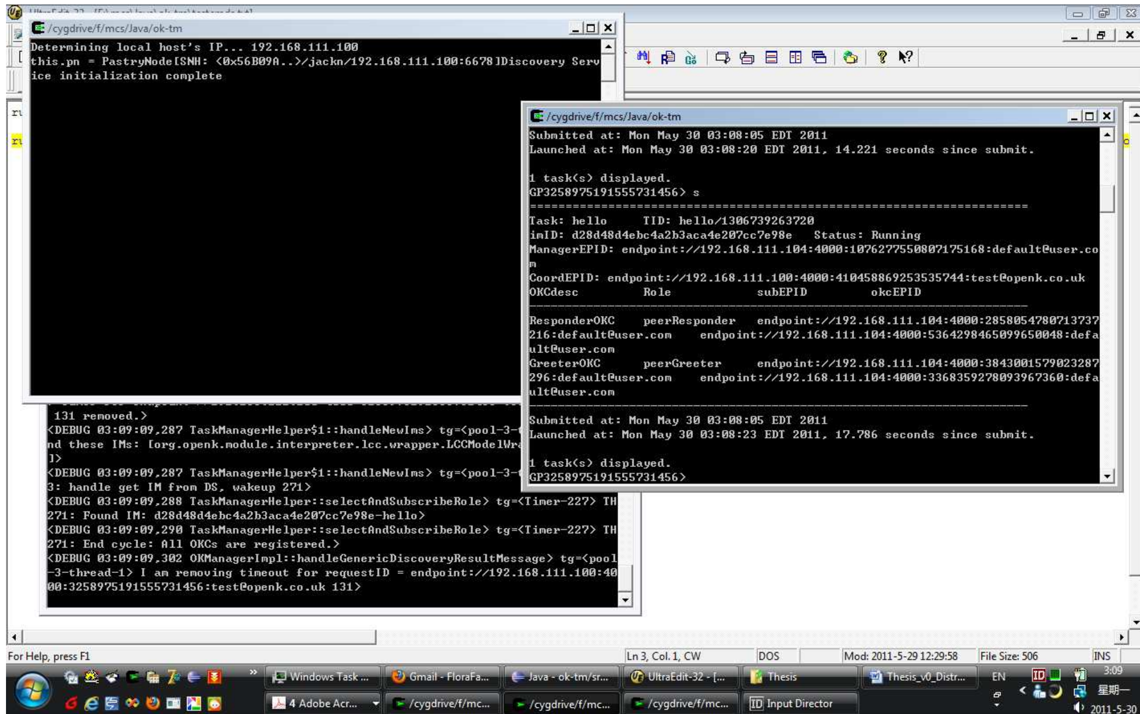

Figure 19. Screen shot after task "Hello World" is submitted (GP_1)

submitted tasks. From Figure 19 one can see the task status after the "*Hello World*" task is fully launched. The allocation of roles to different GPs is non-deterministic due to situation of each GP and the time point of subscription of each GP. The "*show*" command displays the endpoint id of all the requested peer components: the *OKManager*, allocated coordinator, the *SubscriptionNegotiator* of each role and the OKC instance of each role. Because the task is submitted from GP_2, GP_2 now acts as the task manager of this task, and the user input dialog is displayed on GP_2 only, as shown in Figure 18.

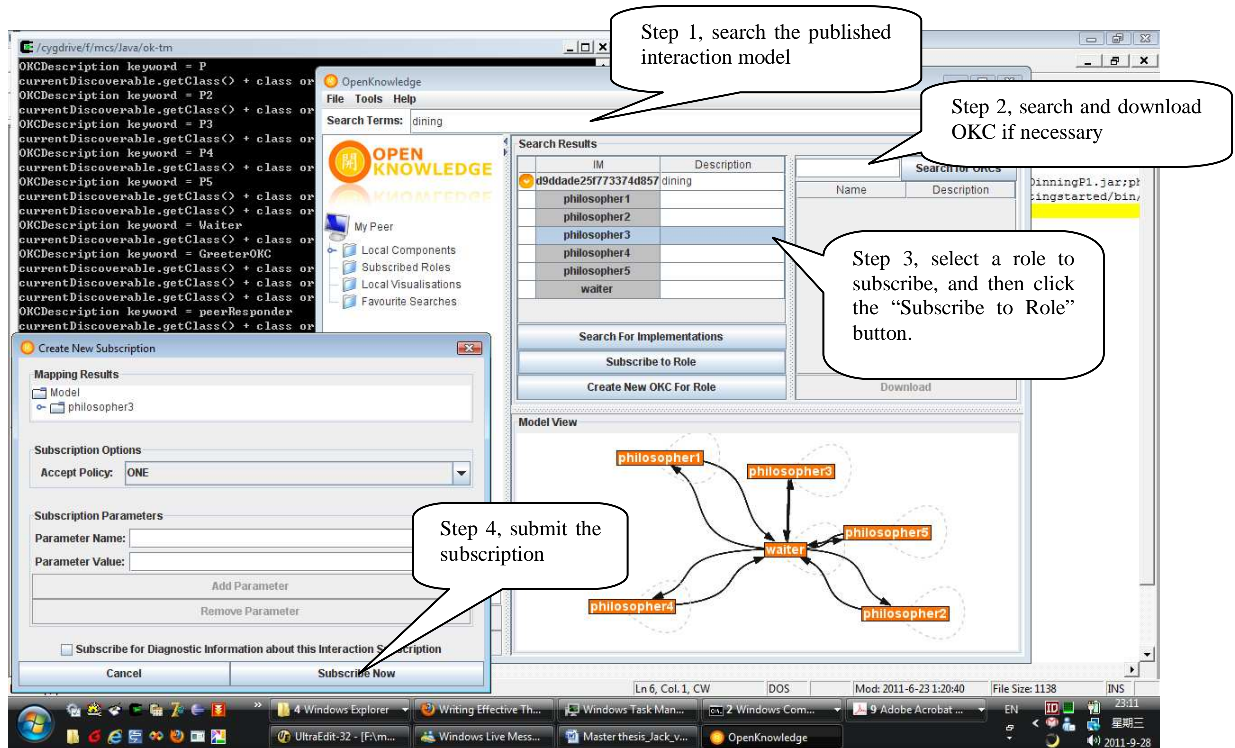

Figure 20. Screen shot of how original OK works with the "dining philosopher" example.

The automated task enrolment represents one of the major adaptions added to make OK cloud ready. For comparison, we use Figure 20 to display the user interface of the original OK manager. From its interface one can find that the original OK manager provides basic management user interface for users to:

- Publish and search an IM,
- Create, publish, search and download an OKC package,
- Import, remove OKC packages from local repository.

From above one can see that compared to the extended OK framework, the original OK manager only provides limited management functions. Because the original OK does not support the concept of task explicitly, user will have no way either to find out the global status of a running Interaction Model, or force control to the course of the interaction from outside. The steps of selecting a role and participating in an interaction is

also annotated in Figure 20, from which one can see that with original OK, the role selection process has to be completed manually. This makes the original task manager not applicable to the cloud platform, in which computation resources or work units should be distributed dynamically via negotiation.

6.1.4 Terminate a Task

Currently, one can submit a "*kill <taskid>*" command at any registered GP as long as one knows the task's identifier. The "*kill*" command has an "*-f*" option. If this option is not set, it performs a mild termination, i.e. the coordinator only informs its LCC interpreter to set the status of all the roles to "*Completed*", and let the interpreter to finish the task in its succeeding operations. Otherwise, if the force option is set, it performs a forced termination, i.e. in addition to notify the LCC interpreter to set the complete status of each role, the coordinator actively send *InteractionCompletedMessage* message to the OKC instances of all participant GPs and send *TaskCompletedMessage* message to the task manager.

6.1.5 User I/O Message Channel via the MessageClient API

The user I/O message channel functionality is provided as a client *MessageClient* API to application developers. It is the decision of the developer about whether to use the *MessageClient* API to redirect the I/O request to the manage console or let the I/O request be processed at local peer without using the *MessageClient* API. Figure 18 displays the input dialog displayed by the manage console of the task manager, in which one can see the endpoint id from which input request sent as well as the prompt message "*Please enter a greeting to send to the other agent*" and default value "*Hello*". The user input will be send back to the GP who invoked the *MessageClient* method.

The user I/O message channel is useful when the agent is running at a remote node of the cloud. In this situation nobody will handle the user input requested at an unattended node, the only way to get the request processed is to redirect it to the management console.

6.1.6 Invoke a Child Task within a Running Task

To test invoking a child task within a running parent task, we rewrite the "*Hello World*" application by adding the interpreting to the user input. When the *peerResponder* receives user input "r" returned from the task manager, it will submit a new task which is an extended version of the "*Dining Philosopher*" example using *MessageClient* API. The child task will use the same GP that behave as the *peerResponder* role of the parent task as its task manager, and its I/O requests will be cached to the I/O queue of the task manager of the child task. When the *peerResponder* receives the user input "c" returned from the task manager of the parent task, it will invoke the de-queue operation on the cached I/O queue of the child task's task manager, and process the user I/O locally based on the fetched I/O request, from which user provide selection about whether a philosopher should eat or think. The selection will then be send back to the role of the child task that had sent the original I/O request.

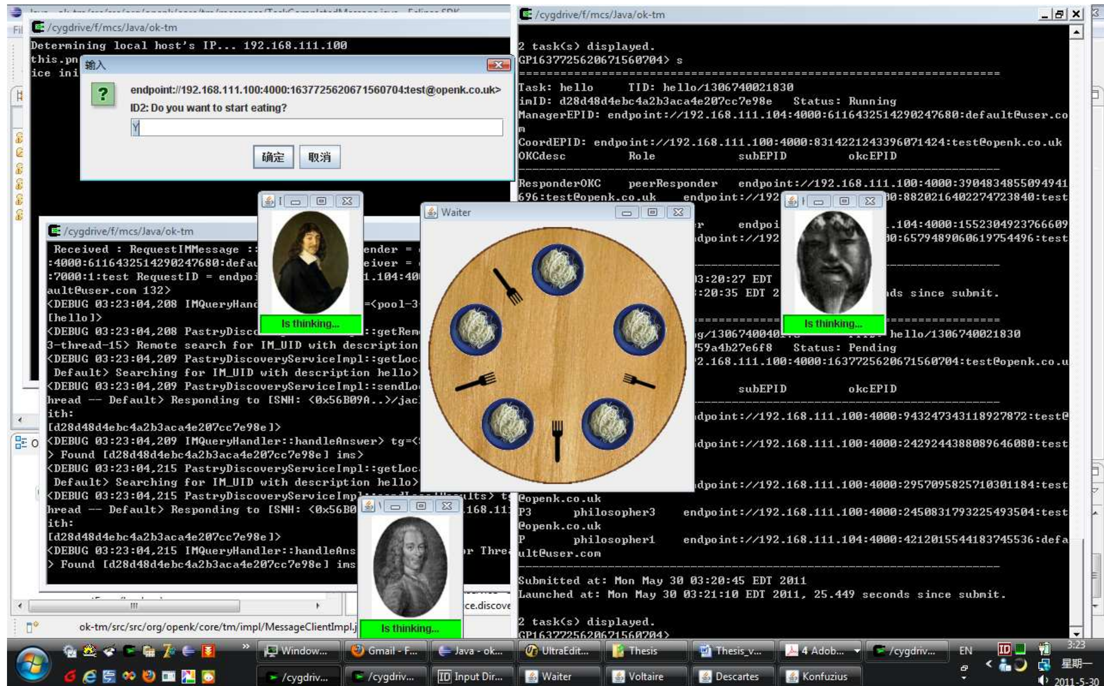

Figure 21. Screen Shot after task "Hello World" invoked child task "Dining Philosophers" (GP_1)

Figure 21 and 22 demonstrates that the roles of the child task are distributed on different GP's. Because the *peerResponder* role of the parent task, which initiates the child task, is allocated to GP_1, the input dialog for child task is displayed at GP_1. However the input dialog of parent task is displayed at GP_2, which is because the parent task is submitted from GP_2. All above phenomena demonstrate that the I/O redirection between parent task and child task works as the design of the message holding mechanism described in Figure 12 of Section 5.3 expected.

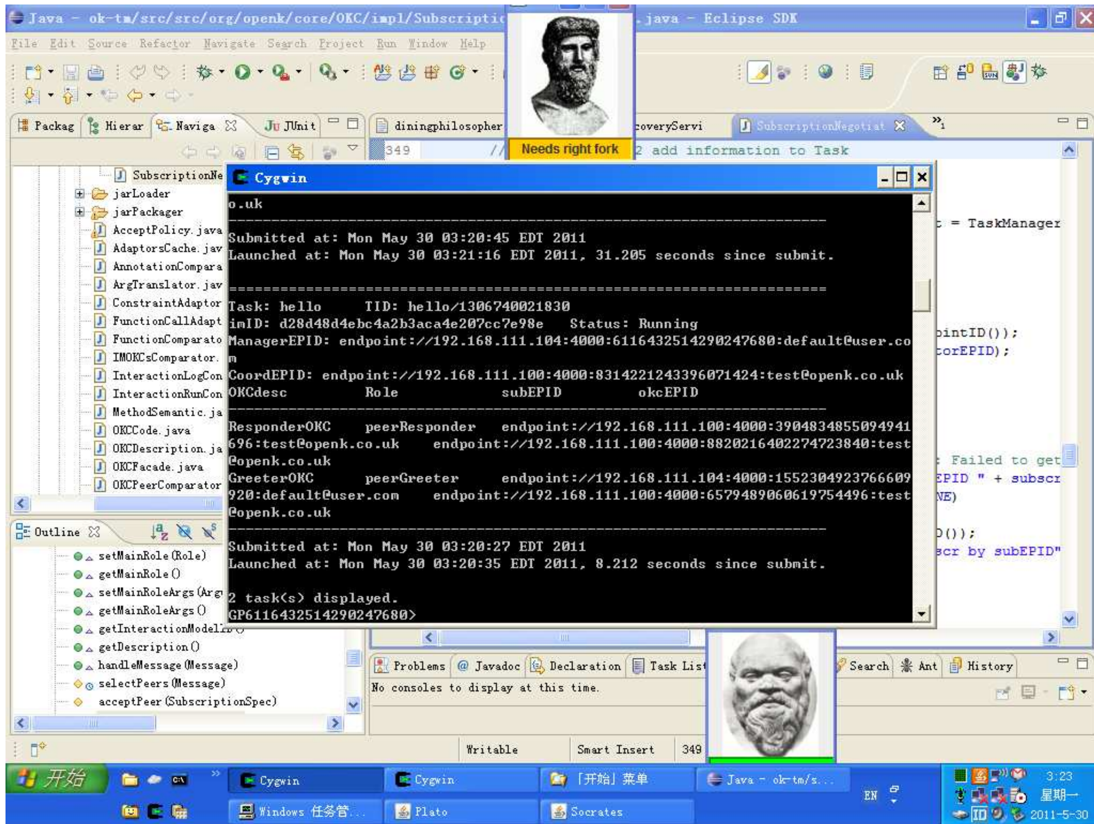

Figure 22. Screen Shot after task “Hello World” invoked child task “Dining Philosophers” (GP_2)

6.2 Performance Analysis

We measure the performance of the prototype task manager by using two metrics: response time and throughput. Because our focus is not to study the performance of the application but to study the performance of the task manager itself, we focus on how much time the task manager used to launch a task and how many tasks can be launched within certain unit of time. We define “task launch” as the action that task manager takes to subscribe all roles of a task and switch its status from *PENDING* to *RUNNING*, and define:

- Task launch response time (or response time *T*): average time for the task manager to launch a task;

- Task launch throughput (or throughput *TP*): average number of tasks can be launched in a given amount of time.

The method we used to analyze the task launch performance is:

First we conduct experiments on real test environment. The goal is to exam if the collected experimental results conform to the calculated results based on the formula for sequential processing of tasks (only allow one task to go through the subscription/launch procedure each time).

Second, we conduct sequential task processing experiments on the simulator to examine if the collected results conform to the calculated results based on the formula. In this way, we can verify that both the prototype system and the simulator behave in the same pattern.

Finally we conduct heavy loaded concurrent task processing experiment on the simulator, and reveal how different factors affect the response time T and throughput TP.

6.2.1 Performance Analysis via Real Testing Environment

Based on the system design, for the sequential submission of tasks, the response time depends on the number of peers in the system and the number of roles to be subscribed. In an ideal scenario, we assume the time used for role subscription and the time used for the interaction to launch to be constant. We define:

t_d time interval that the GP checks pending tasks

t_s average time for a GP to subscribe a role

t_l average time for the coordinator to launch an interaction

N_r number of roles to be subscribed (in the rest of the thesis, for ease of analysis, we treat single role with n instances the same as n roles with single

instance. This engagement does affect the analysis result because the subscription procedure does not make difference between subscribing to one of the multiple instances of a single role and subscribing to one role from a group of roles each requires only one instance),

N_p number of GPs in the system

Assuming the roles are evenly distributed to participated GPs, each GP will subscribe *ceil*(N_r/N_p) roles. The task launch response time of the system is:

$$T = t_d/2 + [ceil(N_r/N_p) - 1]\, t_d + t_s\, ceil(N_r/N_p) + t_l \quad (9)$$

$$= ceil(N_r/N_p)\,(t_d + t_s) - t_d/2 + t_l$$

Where T consists of four parts:

$t_d/2$ Average wait time for GP to check and subscribe the first role

$[ceil(N_r/N_p) - 1]\, t_d$ GP's poll/select interval for the rest $[ceil(N_r/N_p) - 1]$ roles

$t_s\, N_r/N_p$ Total role subscription time for $ceil(N_r/N_p)$ roles

t_l Rest of the interaction initiation time

Table 9 in Appendix A3 shows the test data gathered from running the tasks in the real testing environment. As noted, we executed the "*Hello world*" (containing 2 roles) and the "*Dining Philosophers*" (containing 6 roles) example separately on single GP, double GP single DS, double GP and double DS configurations.

By comparing the collected response time with the calculated response time based on formula (9), we found that for single task scenario, the response time meets with formula (9), which is proportional to N_r and inverse to N_p.

From Table 9 we also found that the number of peers in Discovery Service does not affect the response time significantly. This is because the DS is an independent

subsystem that provides discovery and storage service to other parts of the system. Its query time is constant and only depends on the scale of the underlying Pastry network.

6.2.2 Performance Analysis via Simulation

To study the response time in a larger scale, we used [PeerSim Project] to construct a simulator that simulates the role selection and subscription behaviour in an environment with more peers and number of roles. The simulator works on event based mode. It contains:

- One DS component which represents the whole Discovery Service;
- A group of nodes that represents the GP notes. It uses the same algorithm to select and subscribe roles of submitted tasks. The number of GP nodes is configurable as the *Network Size* or N_p;
- A traffic generator that generate tasks at a specific rate (Task Generation Speed v), and the number of roles of a task (N_r) and its lifespan L can be configured either as fixed values or be assigned randomly from a range;
- A message observer that monitors the running status of the system at configurable time interval and serves the functionality to gather and aggregate data for analysis;
- Other configurable parameters include the total number of tasks to be submitted (*TOTALNUMTASKS*), maximum number of roles a GP can subscribe at one time (M), t_d, t_s and t_l of the system.

Because the current design of the task management framework prototype is based on ideal lab environment at this stage, for simplicity, the simulator is constructed based on two assumptions:

- All peers are running on computers with the same configuration (CPU, memory), they have equal chance of being selected,
- The system is running under ideal state, i.e. no failure of nodes, no transport failure, all messages can arrive at destination.

6.2.3 Simulation of Sequential Task Processing

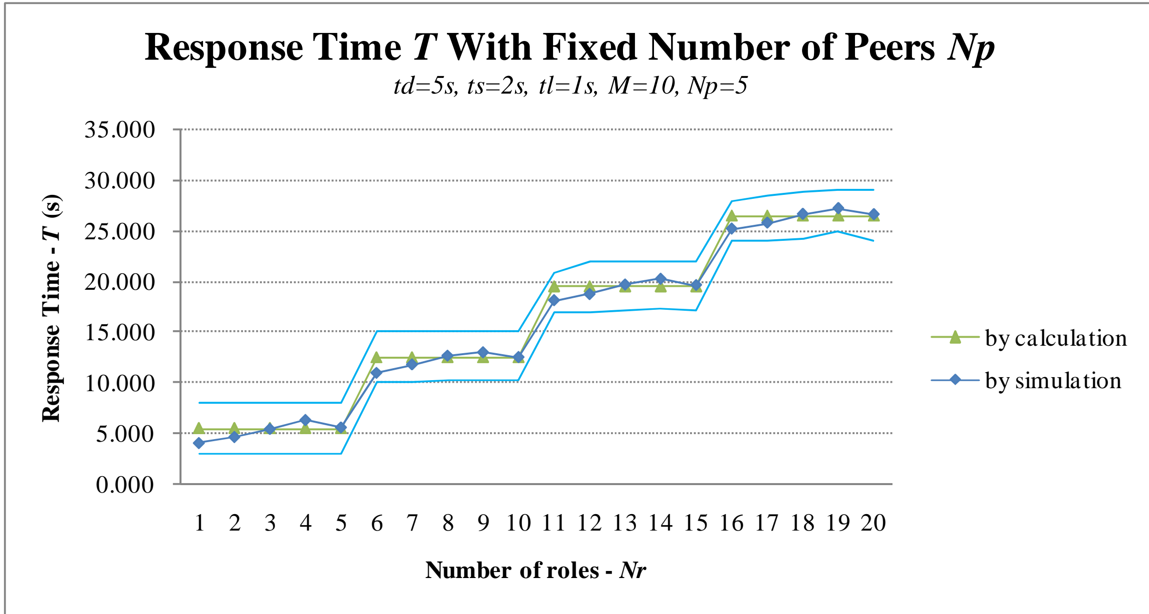

Figure 23. Response time with fixed number of peers and changing number of roles. Tasks are submitted sequentially. One observes that results gathered via simulation are close to the calculated values based on formula 9, where *R-square=0.990*. The width of the upper bound/lower bound envelop falls within *5s*, which matches to t_d, the time interval that a GP examines pending tasks.

Figure 23 demonstrates the comparison between calculated response time (green line) and simulation results (blue line, average response time of 100 tasks per group) in the situation of sequential task processing when the number of peers is fixed, from which we can see the response time collected via simulation are quite close to the calculated values. The response time increases linearly with the increase of number of roles, and the slope matches to $(t_s+t_d)/N_p=1.2$. The upper boundary and lower boundary lines are drawn based on the maximum response time and the minimum response time collected on each round

of experiment. The height of the region falls within t_d = *5s*, which is the interval that the GP checks pending tasks.

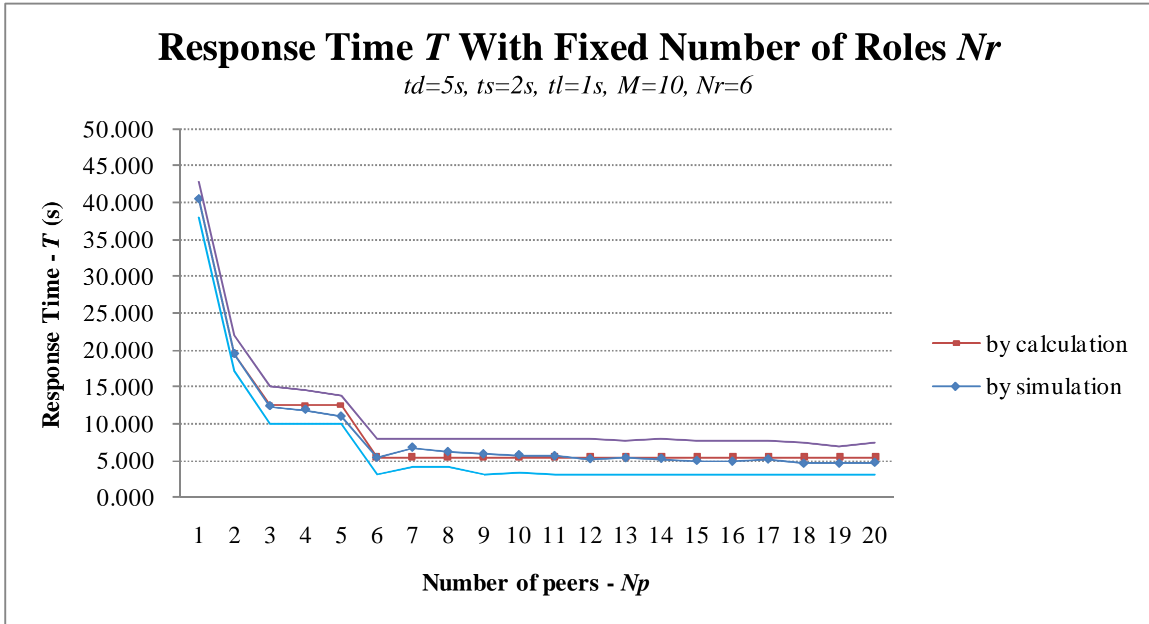

Figure 24. Response time with fixed number of roles and changing number of peers. Tasks are submitted sequentially. As in Figure 23, one observes that results gathered via simulation are close to the calculated values based on formula 9, where *R-square=0.993*. The width of the upper bound/lower bound envelop falls within *5s*, which matches to t_d, the time interval that a GP examine pending tasks.

Figure 24 demonstrates the comparison between calculated response time (green line) and simulation results (blue line, average response time of 100 tasks per group) in the situation of sequential task processing when the number of roles is fixed, from which we can see the task load time values collected via simulation are also close to the calculated values. The task load time decreases inversely with the increase of number of peers.

From Figure 23 and Figure 24 we can see that curve *T* by simulation and curve *T* calculated by using formula (9) are closely fitted each other, which suggests that the simulator works the same way as what formula (9) predicts.

6.2.4 Simulation of Concurrent Task Processing

Next we studied the cases that tasks are submitted at a steady rate without having to wait until the previously submitted task is launched. We submit groups of auto generated tasks (1000 tasks per group) to the simulator in order to reveal how five predictors affect the task launch response time T and task launch throughput TP. The predictors are:

M maximum number of roles a GP can subscribe at one time;

L average life span of tasks;

N_r average number of roles of all tasks during an experiment;

N_p number of peers;

v speed of task submission.

We find that the throughput TP actually depends on combined predictor $\frac{M \cdot N_p}{L \cdot N_r}$.

Figure 25 reveals the linear relationship between $\frac{M \cdot N_p}{L \cdot N_r}$ and TP.

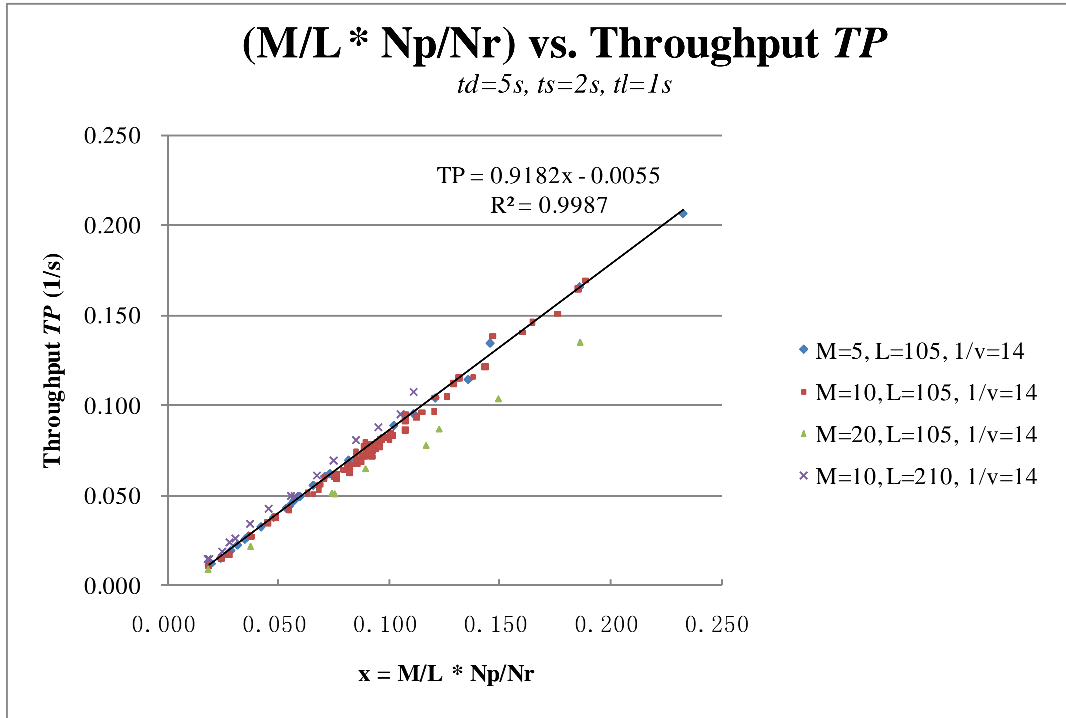

Figure 25. $\frac{M \cdot N_p}{L \cdot N_r}$ **ratio vs. the throughput. Tasks are submitted concurrently. One can observe that the task launch throughput of all series clustered together and is linearly dependent on the ratio** $\frac{M \cdot N_p}{L \cdot N_r}$. **The regression equation is shown in the graph, where *R-square=0.9987*.**

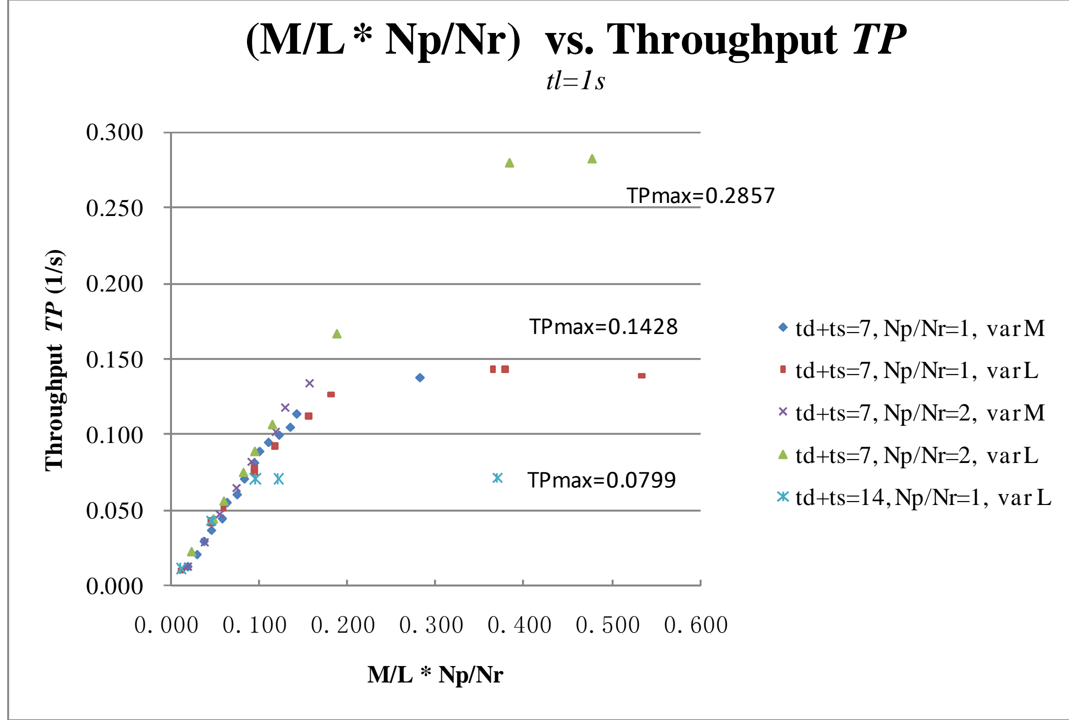

Figure 26. Expanded view of figure 25, which shows that the throughput has an upper limit

Further experimental results show that the *TP* has an upper limit as shown in Figure 26. The upper limit depends on t_d+t_s and N_p/N_r, which is $TP_{max} = 1/(t_d+t_r) * N_p/N_r.$ In summary, the throughput obeys the following empirical formula:

$$TP \approx 0.9182 \cdot \frac{M \cdot N_p}{L \cdot N_r} - 0.006$$
$$TP_{\max} = \frac{1}{t_d + t_s} \cdot \frac{N_p}{N_r} \tag{10}$$

Where the slope 0.9182 is constant and does not depend on any of L, M, N_r, N_p, t_d, t_s *and v*.

The reason why *TP* has an upper limit can be explained as follows:

When $\frac{M \cdot N_p}{L \cdot N_r}$ reaches to certain level, which means that *M* becomes large enough, *L* reduced to certain extent, and the system always has enough peers to host all roles, the

system will be able to launch all tasks in time within the time span of t_d+t_s. At this stage, the only factor that affects the throughput will be reduced to the schedule interval time t_d, role subscription time t_s, and N_p/N_r ratio.

Next we looked at the task launch response time *T*. When the speed of submitting tasks exceeds the system's throughput, more and more tasks will not be processed in time and will be queued to be processed. The longer the queue is the longer the response time will be for those tasks waiting at the tail of the queue. Therefore, the average task launch response time in the situation of infinite task feed will be emanative and is not measurable under this overloading situation. As a result, the average task launch response time *T* should only be measured under the condition that the task submission speed does not exceed the throughput *TP*.

We observe *T's* distribution along with the combined predictor *v/TP* as shown in Figure 27, in which *T* is only measureable within *v/TP*'s region [0, 1]. The *T* upper bound and lower bound envelops of all series of data overlaps each other. Based on experimental results, the lower bound lines of the envelops of all data series stay as a horizontal line $T_{min} = 3s$, where 3 seconds is the sum of $t_s + t_l$, which is the most optimistic situation that all roles of a submitted task are subscribed instantly and the task launched without any delay within the t_d period. The trend of *T* arises along with the increase of *v/TP* and roughly obeys formula

$$T \approx 0.5767 \cdot e^{4.3595 \cdot \frac{v}{TP}} \qquad \text{where } \frac{v}{TP} \in [0,1] \text{ and } R^2 = 0.6344 \tag{11}$$

The regression function is shown as Figure 28. Because the R-square of formula (11) is not very high, the calculated *T* is just a rough estimate. Future work will include more in-depth research on how *T* is affected by each of the predicates M, L, N_r/N_p and v.

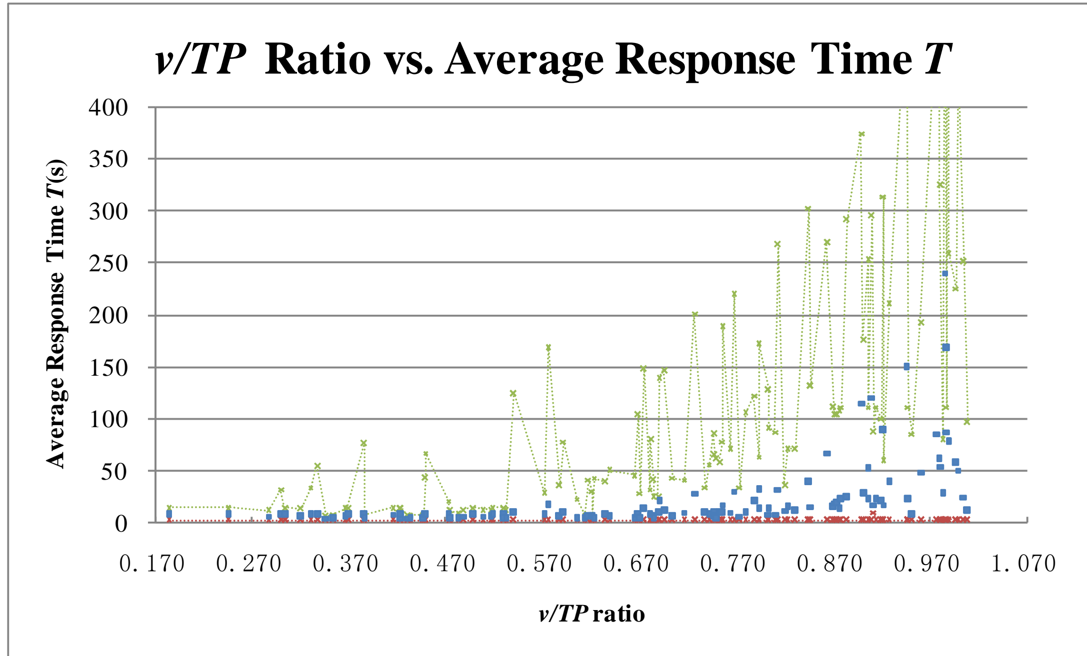

Figure 27. Task submission speed/throughput ratio *v/TP* vs. average response time *T* of all data series with upper bound and lower bound envelope. The quantity *v/TP* is the ratio of task submission speed divided by the throughput. One can observe that the closer the *v/TP* ratio approaches to 1, the variation of either the upper bound or the average response time becomes more dramatic. This can be explained using the nature of the producer/consumer model: The task launch throughput represents the system's maximum consuming speed of submitted tasks. When the task submission speed, i.e. the producing speed, approaches to the consuming speed, where tasks are generated with random lifespan and number of roles, the system will more likely to reach into a temporary overload state, although this overloading state will get relieved in the long run, it will make some queued tasks' response time become extra long. The closer the producing speed approaches to the consuming speed, the harder these overload state will get relieved. Therefore the upper bound/ lower bound envelop will becomes wider. The behaviour of individual random generated tasks that are blocked in the waiting queue will have more impact to the calculation of average response time. Until the task producing speed overtakes the consuming speed, the overload state will not be able to get relieved in the end, and the average task response time becomes emanative and not measurable.

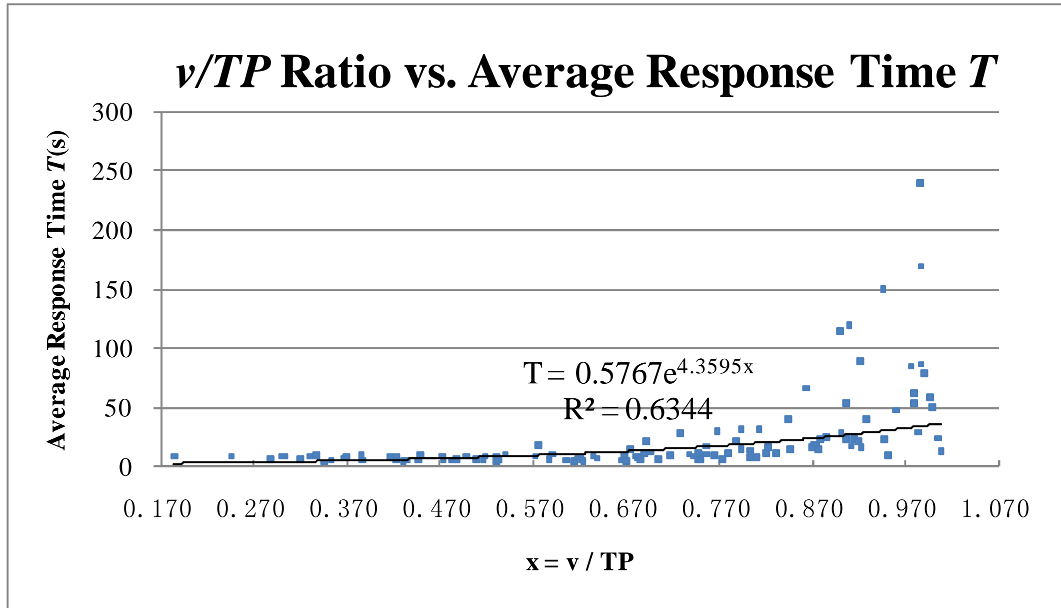

Figure 28. Exponential regression function of average response time *T* based on Figure 27

6.3 Concerns about Dead Locks

For the scenario of sequential task submission, the system will not able to execute or accept new tasks if $Nr > Np \cdot M$. For the concurrent task submission scenarios, dead lock could happen when the N_r/N_p *Ratio* approaches to M. Currently the prototype does not take deadlock into considerations. The deadlock detection and handling mechanisms will be added to the future improvements. We could use time-out based deadlock detection mechanism and algorithms to select and release exclusively occupied resources forcibly.

7. CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

In this thesis, we presented the design and implementation of a software platform that realizes a prototype task management framework to support the running and managing of the LCC based collaboration model under PaaS cloud environment. The framework is constructed through enhancing and extending the OK framework. It improves the provision and negotiation mechanism of existing OK framework and also its manageability. The contributions of our work are:

1. We first proposed the concept of introducing role and social norm based logic programming approach to enrich the programming models of PaaS clouds and used this prototype framework to prove our concept,
2. The framework provides partial solutions to the fully decentralized management challenges of a choreography based distributed collaborating network. The method we used to solve the challenges can be contributed to the design of the future generation cloud infrastructure that supports PaaS based computation models.
3. We performed performance measurements of task launch time behaviours and thereby provided a benchmark for comparison against future improvements.
4. The prototype framework itself can serve as a research platform to support future research. .

The detailed work includes

- Extending existing task management functionality with a set of fully distributed task submission/termination control and task monitoring functionalities,

- Enhancing the underlying task management mechanism of OK framework from the "submit-manual select–subscribe–allocation-run" model to the "submit-proactive select–subscribe–allocation-run" model, which improved the automation level of the task management and make it satisfies two basic requirements of cloud systems, i.e. dynamic provision of resources for tasks and SLA achieved via negotiation.

Although our work is still preliminary, the prototype framework can be used to support and conduct further research, and provide benchmarks and new research hot spots. In the end, our work will impact the way applications are constructed to utilize clouds, and provide cloud application developers with more options to designing and manage their applications.

By analyzing the experimental results, we revealed the underlying mathematical formulas that reflect the performance of the prototype task manager by using different methods, including both real environment experiment and simulation, and under different scenarios, including sequential task processing and concurrent task processing. We focused on analyzing how task launch throughput is influenced by different predictors and in turn how the task launch response time is influenced by task launch throughput and task submission speed. The experimental data collected both from testing and simulation supports the view that the task launch response time is linearly dependent on the number of roles for subscription and inversely dependent on the number of peers in the system in sequential task processing scenario. For concurrent task processing scenarios, we found that the average task launch throughput *TP* is closely related to the combined ratio of $\frac{L \cdot N_r}{M \cdot N_p}$ and obey the formulas described as (10). We realized that the

task launch response time *T* is only measurable when the task submission speed does not exceed *TP*. Its relationship with task submission speed and *TP* can be roughly depicted using formula (11). The collected performance data will be used as the benchmark for the future system optimization.

7.2 Future Work

Currently, the prototype system is only a proof of concept system with less concern for performance, robustness, security and completeness of functionalities. To produce a production system, future work will need to be fulfilled from the following aspects:

1) Performance and robustness:

   Replace the existing centralized coordination mechanism with a distributed coordination mechanism depicted in [Robertson 2005] to improve the performance and robustness of the OK framework.

   More sophisticated role selection algorithms, deadlock detection and handling mechanisms, auditing, post-run analyzing mechanisms, automated distribution of 3rd party libraries, and version control of published IMs, OKCs and 3rd party libraries.

2) User level security and transport level security:

   - Introducing domain based authentication and authorization mechanism to the task management framework. Trust model of p2p system will be fully studied.
   - Introducing message level security to the communication layer.

3) Transport level improvement:

   - Extend the communication layer to support message relay across

NAT/firewall features.

- Optimize the publishing and discovery algorithm that is based on Pastry overlay network for OK Discovery Service.

From the aspect of research, the problem to be solved in the future with the highest priority is the optimization of the existing coordination model. As LCC is originated from concurrent system models like Actor model and process calculi, it inherits the indeterminacy in concurrent computation [Agha 1986], (indeterminacy caused by the arrival order of messages does not necessarily corresponds to the sending order of messages). Although the collaboration model of current OK framework appears to be fully distributed and choreography based, it actually uses centralized coordination and sequential computation to solve the indeterminacy problem which sacrifices the performance and increases network traffic. Future research will focus on breaking down the coordinator into distributed mode. Due to feasibility concern, current consideration tends to adopt the hybrid coordination model which is partial centralized and partial distributed. To weigh to which extent the distribution should be requires further study.

REFERENCES

1. [Robertson 2005] Robertson, David. 2005. A Lightweight Coordination Calculus for Agent Systems. *Springer Berlin / Heidelberg, Declarative Agent Languages and Technologies II, Lecture Notes in Computer Science*, vol. 3476, 183-197.

2. [Robertson 2004] Robertson, David. 2004. Multi-agent Coordination as Distributed Logic Programming. *Springer Berlin / Heidelberg, Logic Programming, Lecture Notes in Computer Science*, vol. 3132, 77-96. Doi: 10.1007/978-3-540-27775-0_29, Url: http://dx.doi.org/10.1007/978-3-540-27775-0_29

3. [PA *et al* 2007] PA, de Pinninck, D, Dupplaw, S, Kotoulas, R, Siebes. 2007. The OpenKnowledge Kernel. *Proceedings of the XXI International Conference on Computer, Information and Systems Science.* Available at: http://www.cisa.informatics.ed.ac.uk/OK/Publications/The%20OpenKnowledge%20Kernel.pdf

4. [Quan *et al* 2007] Xueping Quan, Chris Walton, Dietlind L. Gerloff, Joanna L. Sharman, and Dave Robertson. 2007. Peer-to-peer experimentation in protein structure prediction: an architecture, experiment and initial results. *In Proceedings of the 2006 international conference on Distributed, high-performance and grid computing in computational biology (GCCB'06), Werner Dubitzky, Mathilde Romberg, Assaf Schuster, Peter M. A. Sloot, and Michael Schroeder (Eds.). Springer-Verlag, Berlin, Heidelberg*, 75-98.

5. [Dillon *et al* 2009] Dillon, Tharam, Chang, Elizabeth, Meersman, Robert, Sycara, Katia, Robertson, David, *et al*. 2009. Models of Interaction as a Grounding for Peer to Peer Knowledge Sharing. *Advances in Web Semantics I. Lecture Notes in Computer Science, Springer Berlin / Heidelberg.* vol. 4891, 81-129

6. [Milner *et al* 1992] Milner, Robin, Parrow, Joachim, and Walker, David. 1992. A calculus of mobile processes, I. *Information and Computation*, Volume 100, Issue 1, September 1992, Pages 1-40, ISSN 0890-5401, 10.1016/0890-5401(92)90008-4.

7. [Milner *et al* 1992] Milner, Robin, Parrow, Joachim, and Walker, David. 1992. A calculus of mobile processes, II. *Information and Computation*, Volume 100, Issue 1, September 1992, Pages 41-77, ISSN 0890-5401, 10.1016/0890-5401(92)90009-5.

8. [Agha 1986] Agha, Gul. 1986. Actors: a model of concurrent computation in distributed systems. *MIT Press*, Cambridge, MA.

9. [Peltz 2003] Peltz, Chris. 2003. Web Services Orchestration and Choreography. *Computer*, pp. 46-52, October, 2003.

10. [Trecarichi *et al* 2009] Trecarichi, Gaia and Rizzi, Veronica and Vaccari, Lorenzino and Marchese, Maurizio and Besana, Paolo 2009. OpenKnowledge at work: exploring centralized and decentralized information gathering in emergency contexts. *Technical Report DISI-09-011, Ingegneria e Scienza dell'Informazione*, University of Trento.

11. [OASIS-BPEL 2007] Web Services Business Process Execution Language Version 2.0. *OASIS Standard, April 2007*. Available at: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

12. [W3C-WS-CDL 2004] Web Services Choreography Description Language Version 1.0. *W3C Working Draft,* December 2004. Available at: http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/

13. [Yan *et al* 2005] Yan, J., Yang, Y., Kowalczyk, R., Nguyen, X.T. 2005. A service workflow management framework based on peer-to-peer and agent technologies. *Quality Software, 2005. (QSIC 2005)*. Fifth International Conference on Quality Software (QSIC'05), pp. 373-382, doi: 10.1109/QSIC.2005.8.

14. [Yan *et al* 2006] Yan, Jun, Yang, Yun, and Raikundalia, G.K. 2006. SwinDeW-a p2p-based decentralized workflow management system. *Systems, Man and Cybernetics, Part A: Systems and Humans*, IEEE Transactions on , vol.36, no.5, pp.922-935, doi: 10.1109/TSMCA.2005.855789.

15. [Besana and Barker 2009] Besana, Paolo and Barker, Adam. 2009. An Executable Calculus for Service Choreography. *In Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on the Move to Meaningful Internet Systems: Part I (OTM '09). Springer-Verlag, Berlin, Heidelberg*, 373-380. DOI=10.1007/978-3-642-05148-7_26.

16. [Buyya *et al* 2009] Buyya, Rajkumar, Yeo, Chee Shin, Venugopal, Srikumar, Broberg, James, and Brandic, Ivona, 2009, Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility, *Future Generation Computer Systems*, Volume 25, Issue 6, June 2009, Pages 599-616, ISSN 0167-739X, DOI: 10.1016/j.future.2008.12.001.

17. [Zhang *et al* 2010] Zhang, Qi, Cheng, Lu, and Boutaba, Raouf, 2010, Cloud computing: state-of-the-art and research challenges, *Journal of Internet Services and Applications*, Volume 1, Issue 1, May 2010, Pages 7-18, Springer London, DOI: 10.1007/s13174-010-0007-6.

18. [Barros *et al* 2006] Barros, Alistair, Decker, Gero, and Dumas, Marlon. 2006. Multi-staged and Multi-viewpoint Service Choreography Modelling. TECH REPORT. Available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.90.420.

19. [Chu *et al* 2007] Chu, Xingchen, Nadiminti, K., Jin, Chao, Venugopal, S., and Buyya, R.. 2007. Aneka: Next-Generation Enterprise Grid Platform for e-Science and e-Business Applications. *e-*

Science and Grid Computing, *IEEE International Conference on*, vol., no., pp.151-159, 10-13 Dec. 2007 doi: 10.1109/E-SCIENCE.2007.12.

20. [Bellifemine *et al* 2001] Bellifemine, Fabio, Poggi, Agostino, and Rimassa, Giovanni. 2001. JADE: a FIPA2000 compliant agent development environment. *In Proceedings of the fifth international conference on Autonomous agents (AGENTS '01)*. ACM, New York, NY, USA, 216-217. DOI=10.1145/375735.376120. Available at: http://doi.acm.org/10.1145/375735.376120

21. [Caire *et al* 2008] Caire, Giovanni, Gotta, Danilo, and Banzi, Massimo. 2008. WADE: a software platform to develop mission critical applications exploiting agents and workflows. *In Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: industrial track (AAMAS '08). International Foundation for Autonomous Agents and Multiagent Systems,* Richland, SC, 29-36.

22. [HADOOP Project] The Apache HaDoop Open Source Project. Available at: http://hadoop.apache.org/

23. [Rowstron and Druschel 2001] Rowstron, Antony and Druschel, Peter. 2001. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Springer Berlin / Heidelberg, Middleware, Lecture Notes in Computer Science*, vol. 2218, 329-350. Doi: 10.1007/3-540-45518-3_18, Url: http://dx.doi.org/ 10.1007/3-540-45518-3_18.

24. [OpenKnowledge Manual] OpenKnowledge Manual.
Available at: http://www.cisa.inf.ed.ac.uk/OK/download/manual.pdf

25. [OASIS-BPEL 2007] Web Services Business Process Execution Language Version 2.0. *OASIS Standard*, April 2007. Available at: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.

26. [W3C-WS-CDL 2004] Web Services Choreography Description Language Version 1.0. *W3C Working Draft*, December 2004. Available at: http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217.

27. [Dean and Ghemawat 2008] Dean, Jeffrey and Ghemawat, Sanjay. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM 51*, 1 (January 2008), 107-113. DOI=10.1145/1327452.1327492.

28. [Google App Engine] Google App Engine, Available at: http://code.google.com/appengine.

29. [Google GFS] Ghemawat, S, Gobioff, H and Leung, S-T. 2003. The Google file system. *In proceeding of SOSP*, October 2003.

30. [HDFS] Hadoop Distributed File System. Available at: http://hadoop.apache.org/hdfs

31. [Windows Azure] Windows Azure, Available at: www.microsoft.com/azure

32. [UML] The Unified Modeling Language. Object Management Group. Available at: http://www.uml.org/

33. [JavaCC] Java Compiler Compiler, The Java Parser Generator, Available at: http://javacc.java.net

34. [PeerSim Project] PeerSim: A Peer-to-Peer Simulator. Available at: http://peersim.sourceforge.net/

APPENDICES

A1. Major administrative related methods of extended OKManager and OKManagerImpl

Method	Description
searchTask *(new)*	Search task information from DS based on query terms by invoking *OKDiscoveryProxy.searchTask* method. Its purpose is to get task id and task manager's endpoint id. Parameters: query – terms for query criteria limit – max number of returned items callback –*SearchTaskCallback* typed callback function Message sent: *RequestTaskMessage* Message received: *ResultSearchTaskMessage*
searchTaskFromOKM *(new)*	Query task's *TaskDescripition* information from a task's manager endpoint. The reason we store *TaskDescripition* detail at the manager side instead of DS side is that information published to DS is not changeable due to the current underlying p2p layer limitations based on FreePastry. Parameters: query – terms for query criteria receiver – receiver's manager end point id callback – *SearchTaskCallback* typed callback function Message sent: *RequestTaskMessage* Message received: *ResultSearchTaskMessage*
searchIM	Search published interaction model from DS based on query terms by invoking *OKDiscoveryProxy.searchIM* method. Parameters: query – terms for query criteria limit – max number of returned items receiver – receiver EndPointID callback –*SearchIMCallback* typed callback function Message sent:

	RequestIMMessage Message received: *ResultIMMessage*
searchOKCs	Search published OKC information from DS based on query terms by invoking *OKDiscoveryProxy.searchOKCs* method. Parameters: query – terms for query criteria limit – max number of returned items receiver – receiver EndPointID callback –*SearchOKCCallback* typed callback function Message sent: *RequestOKCMessage* Message received: *ResultOKCMessage*
updateTaskToOKM *(new)*	Update changes of a subscription or instantiation status back to the OKManager from which the task is submitted, all changes are updated to the task's *TaskDescripition* instance stored in the OKManager's task list. Parameters: t – id of the task sub - SubscriptionSpec data structure of the subscription information receiver – receiver EndPointID callback – *PublishCallback* typed callback function Message sent: *RequestUpdateTaskMessage* Message received: *ResultPublishMessage*
removeOKC	Remove OKC from local OKC repository. Parameters: okc – OKCDescription information of the OKC
downloadOKC	Download published OKC code from DS and add it to local OKC repository by invoking *OKDiscoveryProxy.downloadOKCcode* method. Parameters: okc – OKCDescription information of the OKC

	callback – *DownloadOKCCodeCallback* typed callback function Message sent: *RequestOKCMessage* Message received: *ResultDownloadOKCCodeMessage*
subscribeToRole	Subscribe to specific role of a published interaction to DS by invoking *OKDiscoveryProxy.subscribeToRole* method. Parameters: adapt – *SubscriptionAdaptor* instance that provides maping between role and OKC policy – AcceptPolicy, available values: ONE/ALL/NONE participant – string of the role participantArgs – ArgumentList type, not used currently modelID – id of the interaction model subscriptionDescription - Description of the subscription that can be searched subscriptionParams - Map<String,Object>, not used currently expireInterval – number of millisecond of expiration diagnostics – Boolean value of enable diagnostics, additional listeners for the incoming messages callback – *SubscribeCallback* typed callback function EOIListeners - List<InteractionLogConsumer>, listeners that monitors end of interaction askForPeerSelection – Boolean value indicates whether the bootstrap coordinator should ask the peer to select the peers it wants to interact with Returns: A *SubscriptionSpec* instance for subscription information Message sent: *RequestSubscribeToRoleMessage* Message received: *DiscoveryResultMessage*
getParticipated_tasks *(new)*	The Participated_tasks is a collection that stores all pending or running tasks' information that are participated by local GP.
getTasklist	The Tasklist is a collection that stores all pending or running tasks'

(new)	information that are submitted by local GP.
requestConsoleIO *(new)*	Send task role's input/output request to the manager of the task. Used by MessageClient's Input and Prompt method. Message sent: *RequestConsoleIOMessage* Message received: *ResultConsoleIOMessage*
inspectConsoleIOFromOKM *(new)*	Inspect I/O request on the task manager of a child task and get the next I/O request message. Used by *MessageClient.checkChildConsoleIO* method Message sent: *InspectConsoleIOMessage* Message received: *ResultInspectConsoleIOMessage*
publishTask	Publish the *TaskDescription* to DS to make the task searchable by other GPs through invoking *OKDiscoveryProxy.publishTask* method Message sent: *RequestPublishTaskMessage* Message received: *ResultPublishMessage*
publishIM	Publish an IM to DS to make it searchable by other OKManagers through invoking *OKDiscoveryProxy.publishIM* method Message sent: *RequestPublishIMMessage* Message received: *ResultPublishMessage*
publishOKC	Publish the *OKCDescription* of an OKC to DS to make it searchable by other OKManagers through invoking *OKDiscoveryProxy.publishOKC* method. Message sent: *RequestPublishOKCMessage* Message received: *ResultPublishMessage*

Table 7. Major administrative related methods of *OKManager* and *OKManagerImpl*

A2. Newly added message types

Message Type	From	To	Description	Content	Response Message
RequestTaskMessage	*OKDiscoveryProxy* of participant GP	Discovery Service	Search task from DS	*RequestID* – handler for callback matching, Query terms of task description	*ResultSearchTaskMessage*
	OKM of participant GP	OKM of task manager	Get task information from task manager		
ResultSearchTaskMessage	Discovery Service or OKM of task manager	OKM of participant GP	Response of above message	*RequestID*, A *TaskDescription* instance	
RequestPublishTaskMessage	OKM of task manager	Discovery Service	Publish task information to DS	*RequestID*, A *TaskDescription* instance	*ResultPublishMessage* (existing)
RequestUpdateTaskMessage	OKM of participant GP	OKM of task manager	Update task description to task manager	*RequestID, TaskID*, *SubscriptionSpec*	*ResultPublishMessage*
TaskCompletedMessage	OKM of any GP in the network	Coordinator	Inform Coordinator or task manager to terminate task	*TaskDescription,* force flag	
	Coordinator	OKM of task manager			
RequestConsoleIOMessage	OKM of	OKM of task	Relay user I/O	*RequestID, TaskID,* string to	*ResultConsoleIOMessage*

	participant GP	manager	request from task manager	be displayed, operation type – INPUT or OUTPUT, default value to be displayed prior to input	
ResultConsoleIOMessage	OKM of task manager or OKM of participant GP of parent task	OKM of participant GP	Response of above message	*RequestID,* string of result	
InspectConsoleIOMessage	OKM of participant GP of parent task	OKM of child task manager	Get next I/O request from child task	*RequestID, TaskID*	*ResultInspectConsoleIOMessage*
ResultInspectConsoleIOMessage	OKM of child task manager	OKM of participant GP of parent task	Response of above message	*RequestID,* original *RequestConsoleIOMessage* sent by child task	

Table 8. Newly added message types for task management purpose

A3. Test data collected from real experiments

Test Data																
			GP check interval:		5	s	max # of roles per peer can participate:			10						
			response time(s)		**response time(s)**		**response time(s)**		**response time(s)**		**response time(s)**		**avg response time(s)**		**response time(s)**	**relative error**
# DS	**# GP**	**# roles**	**GP1**	**GP2**	**GP1**	**GP2**	**GP1**	**GP2**	**GP1**	**GP2**	**GP1**	**GP2**	**GP1**	**GP2**	**avg**	
1	1	2	9.142	14.130	9.473	14.160	9.900	14.171	10.944	14.291	9.420	14.191	9.776	14.189	11.982	
			avg ts #1:	0.854	avg ts #2:	3.277	avg tl #1	0.109	avgtl #2	0.500		Calculated:	9.317	14.554	11.936	
														error(s):	(0.047)	-0.39%
1	1	6	37.605	57.653	39.897	61.418	38.087	57.152	39.857	62.560	38.429	56.651	38.775	59.087	48.931	
			avg ts #1:	1.709	avg ts #2:	5.604	avg tl #1	1.134	avgtl #2	0.451		Calculated:	38.887	61.576	50.231	
														error(s):	1.300	2.66%
1	2	2	9.477	10.566	8.369	8.082	7.641	8.930	8.662	8.101	8.011	8.382	8.432	8.812	*8.812*	
			avg ts #1:	1.384	avg ts #2:	5.917	avg tl #1	0.118	avgtl #2	0.581		Calculated:	4.002	8.998	*8.998*	
														error(s):	0.186	2.11%
1	2	6	27.076	29.515	27.357	26.078	27.587	30.684	31.826	27.786	31.015	34.810	28.972	29.775	*29.775*	
			avg ts #1:	1.618	avg ts #2:	5.868	avg tl #1	0.680	avgtl #2	0.537		Calculated:	18.034	30.640	*30.640*	
														error(s):	0.866	2.91%
2	2	2	8.744	7.942	8.071	7.311	8.423	8.171	8.404	9.234	7.631	11.246	8.255	8.781	*8.781*	
			avg ts #1:	1.404	avg ts #2:	5.687	avg tl #1	0.119	avgtl #2	0.469		Calculated:	4.023	8.656	*8.656*	
														error(s):	(0.124)	-1.42%
2	2	6	30.374	30.785	29.032	27.670	31.746	34.730	29.795	28.811	28.385	32.657	29.866	30.931	*30.931*	
			avg ts #1:	1.603	avg ts #2:	5.664	avg tl #1	0.646	avgtl #2	0.467		Calculated:	17.954	29.959	*29.959*	
														error(s):	(0.972)	-3.14%

Table 9. Test data collected from real ex experiments

Note: above data are collected in groups of different number of DS, GPs and roles to subscribe. In each group we collect five pairs of data from GP_1 and GP_2 with single task running on machine #1 and #2 respectively. For ease of comparison, under each group of collected data, we provide calculated task response time based on formula (9). The average task subscription time t_s and average interaction launch time t_l are also based on collected data. For cases with # of GP greater than 1, roles are evenly distributed to each peer. The response times for these cases in grey area are calculated using *max* aggregation function rather than the *avg* function because the final response time depends on the time used on the slower node.

A4. Source code and experiment data download

All source code for the Task Manager, the simulator and the experiment data can be downloaded from the SVN server at:

svn+ssh://safetysurvey.ca/export/vhosts/sites/safetysurvey.ca/svn/repos/projects/SurveyProjects/v2.0_or_older/jack

or upon request at zhu19@uwindsor.ca.

File Path	Filename	Description
ok-tm	Refer to Table 6	Source code of enhanced OK kernel and task manager.
peersim	src/peersim/taskmanager/*	Source code of the implemented simulator for task manager.
	taskmanager.cfg	Configurations file for implemented task manager simulator.
	Run.cmd	Batch command to start the task manager simulator.
	Mui.m	Matlab script that visualizes the time series of a task simulation.
	TMObserverlog.dat	Input for mui.m generated by peersim.
	Mynlinfit.m	Matlab script that generate the empirical formula via nonlinear least-squares regression
	t.dat	T, v, N_r/N_p * L/M data extracted from testdata.xls, used as input for mynlinfit.m
thesis	testdata.xls	Excel spreadsheet of raw and derived experimental data

Table 10. Description of files and transcripts

A5. LCC Specification and Example

The BNF definition of LCC [Robertson 2005] is:

$Framework := \{Clause, \ldots\}$

$Clause := Agent :: Def$

$Agent := a(Type, Id)$

$Def := Agent \mid Message \mid Def\ then\ Def \mid Def\ or\ Def \mid Def\ par\ Def \mid null \leftarrow C$

$Message := M \Rightarrow Agent \mid M \Rightarrow Agent \leftarrow C \mid M \Leftarrow Agent \mid M \Leftarrow Agent \leftarrow C$

$C := Term \mid C \wedge C \mid C \vee C$

$Type := Term$

$M := Term$

The LCC is a set of clauses; each clause defines how a role in the interaction be performed. Roles are described as *a(Role, Identifier)*, which contains the name of the role and an identifier for the individual peer undertaking that role. The definition of performance of a role is constructed using combinations of the sequence operator *'then'* or choice operator *'or'* to connect messages and changes of role. Messages are either outgoing to another peer in a given role ('⇒') or incoming from another peer in a given role ('⇐'). Message input/output or change of role can be governed by a constraint defined using the normal logical operators for conjunction, disjunction and negation. A constraint acts as a function or service that returns a Boolean value to indicate if it is satisfied. There are two kinds of constraints: proaction constraints and reaction constraints. Proaction constraints define the circumstances under which a message allowed by the dialogue framework is allowed to be sent. Each constraint is of the form:

$$A : (M \Rightarrow Ar) \leftarrow Cp \qquad (12)$$

Where *A* and *Ar* are peer descriptors (of the form *a*(*Role, Id*)); *M* is a message sent by *A* addressed to *Ar*; and *Cp* is the condition for sending the message (either empty or a conjunction of sub-conditions which should hold in *A*). If *Cp* returns *true* value, which means the constraint is satisfied, message *M* will be sent from *A* to *Ar*. Reaction constraints define what should be true in a peer following receipt of a message allowed by the dialogue framework. It usually returns *true* and is used to define the post action after *A* received message *M* from *Ar*. Each constraint is of the form:

$$A : (M \Leftarrow As) \leftarrow Cr \qquad (13)$$

Below is a piece of LCC script which describes the interaction model of dining philosophers (Full length source code can be found at "*gettingstarted/lcc/ diningphilosophers1.lcc*" of the source tree):

```
1.  r(waiter, initial)
2.  r(philosopher, necessary, 5)
3.
4.  a(waiter, W) ::
5.   // Initialise
6.   null <- getPeers("philosopher", Peers) and initialise(Peers, NumP)  then
7.   a(waiter(Peers, NumP), W)  then
8.   a(waiter, W)
9.
10. a(waiter(Peers, NumP), W) ::
11.        null <- Peers = []
12.        or    // choice
13.        (    null <- Peers = [Peer | PeerR] and getID(Peer, ID, PID) then
14.             init(ID, NumP) => a(philosopher, Peer) then
15.             (
16.               (
17.                  requestLeft(ID) <= a(philosopher, Peer) then
18.                  (
19.                     left(ID) => a(philosopher, Peer) <- giveFork(ID)
20.                     or
21.                     waitLeft(ID) => a(philosopher, Peer)
22.                  )
23.                 or
                  …
50.           )
51.          then
52.          a(waiternew(PeerR, NumP), W)
```

Infinite recursion

Head and tail list operation

Proactive constraint for sending a message

```
53.       )
54.
55. a(philosopher, P) ::
56.   init(Temp, NumP) <= a(waiter, W) <- initialise1(Temp, NumP) then
57.   (
58.      (
59.         requestLeft(Temp) => a(waiter, W) <- wantsLeft(Temp)
60.         then
61.         (
62.                 left(Temp) <= a(waiter, W) <- gotLeft(Temp)
63.            or
64.            waitLeft(Temp) <= a(waiter, W) <- gotWaitLeft(Temp)
65.         )
66.      )
67.      or
         …
86.   )
87.   then
88.    a(philosophernew, P)
```

Reactive constraint after receiving a message

The first two lines of above script specify that there are two roles in the interaction, the *waiter* and the *philosopher*. This interaction needs one *waiter* and five *philosophers*. The interaction starts from the *waiter* role. The interpretation process of the LCC script is a series of clause expansion and closing similar to the way other logical programming languages are executed [Robertson 2005].

The *getPeers("philosopher", Peers)* constraint at line 6 is an OK predefined constraint that provides a list of participant peers that act as the specific role, which is "*philosopher*" in this case. All the arguments for constraints are reference arguments that can pass information in or out. The *initialise* constraint at line 6 uses argument *Peers* to initialize the *waiter*'s user interface, and returns a number via output parameter *NumP* to represent the number of participant philosophers.

The *a(waiter(Peers, NumP), W)* statement at line 7 and its clause definition starting from line 10 demonstrates a scenario that a role can retain its state at LCC level. The clause *a(waiter(Peers, NumP), W)* at line 10 can be explained as: the agent act as role

waiter running at peer *W*, which retains a list of peers (which is the list of philosopher OKC instances) and number of philosophers. The body of the clause *a(waiter(Peers, NumP), W)* is a standard design pattern of a finite recursion in logic programming, which is achieved through splitting a set into its header element and the tail set (line 13), taking the header element and passing the tail set to the next level of recursion (line 52). At last, the recursion stops until the set (*Peers*) becomes empty (line 11).

A6. OpenKnowledge Component Example

The LCC script only defines how different roles interact through role state change or message exchange, and uses constraints to define the pre-condition of whether the action will happen or the post-condition about the consequences of the action. The internal logic of these constraints is implemented as OpenKnowledge Components (OKC).

An OKC is a class library that contains descriptive information about what the OKC is about and a class that contains the implementation of all the constraints of a role as member functions. The following sample code is the OKC source code for the *waiter* role of above "*diningphilosophers*" LCC.

```
1.  public class PeerWaiterOKC extends OKCFacadeImpl
2.  {
3.          private static final int WIDTH = 320;
4.          private static final int HEIGHT = 340;
5.          ...
21.         private List peerList = new ArrayList();
22.         public boolean[] forks = new boolean[] { true, true, true, true, true };
23.
24.         public boolean initialise(Argument Peers, Argument NumP)
25.         {
26.                 List ps = (List)Peers.getValue();
27.                 NumP.setValue(new Integer(ps.size()));
                    ...
43.                 if (frame == null)
44.                 {
45.                         //Initialize the UI
                            ...
92.                         frame.setVisible(true);
```

All OKC classes inherit the *OKCFacadeImpl* base class.

Corresponds to constraint initialize. Returns true or false to indicate if the constraint is satisfied...

Set and get the argument value

```
93.             }
94.             return true;
95.     }
96.
97.     public boolean giveFork(Argument ForkIndex)
98.     {
99.             // Update the state of the dining table, set result based on the fork's availability
                ...
118.            updateGUI();
119.            return result;
120.    }
121.
122.    public boolean forkReturned(Argument ForkIndex)
123.    {
124.            // Update the state of the dining table
                ...
128.            updateGUI();
129.            return true;
130.    }
        ...
164. }
```

Above source code can be compiled and built into an OKC package (which is a jar file) using OK Management Tool. The OKC package can published to the DS and can be found and downloaded by the peer that is allocated with the specific role. After an interaction is launched, the OKC package will be loaded into the memory as a part of the OKC Instance to provide the constraint solving service upon requested by the Coordinator.

The above example demonstrates three advantages of using LCC and OKC based programming model to design and implement distributed applications. First, by using LCC, one can easily grasp the essential characteristics of the interaction through role identification, message exchange and reasoning about social norms. Second, the definition of the interaction model is modular due to the role-based nature of LCC. Third, the introducing of OKC helps developers to organize the implementation details in an

elegant manner. Therefore, we see great prospect in introducing LCC based modeling techniques to cloud application development.

VITA AUCTORIS

Name: Lichun (Jack) Zhu

Place of birth: Xining, Qinghai, P.R.China

Education:

Bachelor of Engineering, Computer Science,

University of Science and Technology of China, Hefei, China

1989-1994

Master of Science, Computer Science Department,

University of Windsor, Windsor, Canada

2006-2008, 2011