

國立交通大學

資訊科學與工程研究所

碩 士 論 文

點對點環境下的通用發佈／訂閱框架

A Generic Publish/Subscribe Framework for Peer-to-Peer
Environment

研 究 生：簡士強

指 導 教 授：袁賢銘 教授

中 華 民 國 九 十 七 年 六 月

點對點環境下的通用發佈/訂閱框架
A Generic Publish/Subscribe Framework for Peer-to-Peer Environment


研究生：簡士強

Student : Shih-Chiang Chien

指導教授：袁賢銘

Advisor : Shyan-Ming Yuan

國立交通大學
資訊科學與工程研究所
碩士論文



A Thesis
Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in
Computer Science

June 2008

Hsinchu, Taiwan, Republic of China

中華民國九十七年六月

點對點環境下的通用發佈/訂閱框架

學生：簡士強

指導教授：袁賢銘

國立交通大學資訊科學與工程研究所

摘要

直至今日，許多新的點對點網路演算法仍不斷的被提出。應用程式開發者便需要學習各種相異 API 的用法，因而增加了開發者在轉換使用不同點對點網路時額外的負擔。同時也使得開發者難以針對特定應用領域評量各點對點網路的優劣。而從點對點網路開發者的角度來看，如果能夠提供一組完備且可重用的網路傳輸組件，將能大幅的提昇開發者將點對點網路部署到各種實體網路的進程。

本研究提出一個全新的開發框架，協助開發者使用各種點對點網路拓撲與發佈訂閱演算法來開發點對點網路應用程式。在我們所提出的系統架構中，包含了各個開發點對點網路相關應用所需的功能，其中涵蓋了「網路傳輸」、「點對點網路演算法」、「網路啟動」與「可抽換發布訂閱服務」四項功能群。為了展示此開發框架的通用性與優點，我們提供了環狀網路與 Viceroy 點對點網路的示範實作，並且提供一個簡單的發布訂閱演算法實作。此外，我們將透過一個範例應用程式來展現此框架在開發點對點發布訂閱應用時，藉由抽換功能組件達成部署應用於不同環境上的便利性。

A Generic Publish/Subscribe Framework for Peer-to-Peer

Environment

Student: Shih-Chiang Chien

Advisor: Shyan-Ming Yuan

Institutes of Computer Science and Engineering

National Chiao-Tung University

Abstract

At present, the structured P2P algorithms have been proposed frequently. Consequently, the P2P application developers need to learn different API semantics. It generates additional efforts of switching to different P2P topologies. Moreover, it is difficult for the developers to evaluate the performance of an application based on a particular underneath P2P APIs. On the other hand, if the P2P framework can provide reusable and comprehensive network communication components, it can expedite developing progress. Therefore, the P2P protocols can easily accommodate to different network environments.

In this research, a novel P2P developing framework is proposed to assist in developing P2P applications by using various structured P2P protocols and P2P pub/sub algorithms. We design an architecture to construct the structured P2P functional blocks, including network communication components, P2P topology maintenance and routing, network bootstrapping, as well as pluggable pub/sub services. In order to demonstrate the genuineness and generality of the framework, we provide a ring protocol, the Viceroy DHT implementation, and a simple pub/sub algorithm. Furthermore, we generate a client application to indicate the convenience of exchanging among different underlying networks, P2P protocols, and pub/sub services.

Acknowledgement

兩年的碩士生涯匆匆的過去了，很慶幸能在學生生活結束之際完成了令自己滿意的研究成果。爲此，首先要感謝我的指導教授袁賢銘老師，在尋找研究題材時給予我相當大的自由度，並且適時的給予建議，幫助我抓緊研究主軸。同時也感謝三位口試委員，蔡清欉教授、謝筱齡教授、以及林獻堂教授，百忙之中仍抽空給予我許多有用的批評與建議。在此要特別感謝謝筱齡老師在論文寫作方面給了我相當大的協助。此外，感謝葉秉哲學長與高子漢學長，在數次的合作中分享了許多研究以及實作上的經驗，同時也感謝吳瑞祥學長、鄭明俊學長、邱繼弘學長、高永威學長、與林家鋒學長在兩年的研究室生活中提供的諸多協助。

在此也感謝分散式系統實驗室夥伴們：周鴻仁、林辰璞、林宜豐、以及謝明志，在課業上與研究上相互砥礪，也一起營造了實驗室歡樂的氣氛。此外，我也要感謝田晏、盧奕丞，以及所有交大資科 95 級的同學，在大學與研究所六年的期間留下了許多美好的回憶，也祝福大家不管在研究上或是工作上都能一切順利。

最後要感謝生養我的父母親：簡天成先生與吳麗雪女士，賜與我一個靈活的腦袋，並且在我求學的過程中提供了生活上所需的一切，讓我能專心致力於學習以及研究。同樣的，也要感謝我的兄長簡大鈞，在日常生活中不斷加強我邏輯與思考辯證的能力，僅以這篇研究來回報你們無私的付出。

Table of Contents

ACKNOWLEDGEMENT	I
TABLE OF CONTENTS	II
LIST OF FIGURES.....	V
LIST OF TABLES	VI
1. INTRODUCTION	1
1.1. MOTIVATION.....	2
1.2. CONTRIBUTIONS OF THIS THESIS.....	4
1.3. THESIS OUTLINE	4
2. BACKGROUND AND RELATED WORK	5
2.1. BACKGROUND.....	5
2.1.1. Structured P2P Network.....	5
2.1.2. P2P Publish/Subscribe Algorithm.....	7
2.2. P2P COMMON API.....	8
2.3. PUBLISH/SUBSCRIBE COMMON API	9
2.4. P2P PUB/SUB LIBRARY	10
3. SYSTEM ARCHITECTURE	11
3.1. OVERVIEW.....	11
3.2. P2P PROTOCOL LAYER	12
3.2.1. Peer interface.....	14
3.2.2. PeerFactory class	15
3.2.3. Resource, Id, and PeerId interface	15
3.2.4. IdFactory and PeerIdFactory interface	16
3.2.5. CancellableTask interface.....	16
3.2.6. Service interface	16
3.3. PUB/SUB SERVICE AND API.....	16
3.3.1. PubSubService interface.....	17
3.3.2. Publisher and Subscriber.....	18
3.3.3. EventHandler interface.....	19
3.3.4. Event interface	19
3.4. TRANSPORT LAYER	20
3.4.1. CommunicationManager interface	21

3.4.2.	<i>Address interface</i>	21
3.4.3.	<i>RouteMessage interface</i>	21
3.4.4.	<i>Message interface</i>	21
3.5.	BOOTSTRAP SERVICE.....	22
3.5.1.	<i>BootstrapService interface</i>	22
4.	IMPLEMENTATION DETAILS	23
4.1.	CONTROL FLOW.....	23
4.1.1.	<i>Peer Bootstrapping</i>	23
4.1.2.	<i>Message Transmission</i>	24
4.1.3.	<i>Pub/Sub Actions</i>	26
4.2.	ADDITIONAL CLASS USAGE.....	27
4.2.1.	<i>Environment</i>	27
4.2.2.	<i>AbstractPeer</i>	28
4.2.3.	<i>AbstractEvent</i>	28
4.2.4.	<i>NodeHandle</i>	28
4.2.5.	<i>Topic</i>	29
4.2.6.	<i>LocalBootstrapService</i>	29
4.2.7.	<i>HttpBootstrapService</i>	29
4.2.8.	<i>LocalCommunicationManager</i>	30
4.2.9.	<i>TCPCommunicationManager</i>	30
5.	EVALUATIONS	31
5.1.	SCENARIO DEMONSTRATION.....	31
5.1.1.	<i>Implementing P2P Protocol</i>	31
5.1.2.	<i>Implementing Pub/Sub Service</i>	36
5.1.3.	<i>Develop P2P Pub/Sub Application</i>	38
5.2.	COMPARISONS.....	43
6.	CONCLUSION AND FUTURE WORK	46
6.1.	CONCLUSION.....	46
6.2.	FUTURE WORK.....	47
7.	APPENDIX	50
A.	THE RING PROTOCOL.....	50
B.	ENHANCED VICEROY PROTOCOL.....	52
C.	THE SIMPLE PUB/SUB PROTOCOL.....	57
D.	EXAMPLE PROGRAM.....	58
I.	<i>Publish Client</i>	58
II.	<i>Subscribe Client</i>	60

8. REFERENCES63



List of Figures

Figure 2-1 – Common Structured P2P Topologies.	6
Figure 3-1 – System Architecture Overview.....	11
Figure 3-2 – Class Diagram of P2P Protocol Layer.....	13
Figure 3-3 – Class Diagram of Pub/Sub Service and API.	17
Figure 3-4 – Class Diagram of Transport Layer.	20
Figure 3-5 – Class Diagram of Bootstrap Service	22
Figure 4-1 – Sequence Diagram of Peer Bootstrapping	23
Figure 4-2 – Sequence Diagram of Message Routing	24
Figure 4-3 – Sequence Diagram of Service Callbacks	25
Figure 4-4 – Sequence Diagram of Publishing	26
Figure 4-5 – Sequence Diagram of Subscribing	27
Figure 5-1 – Relationships between Ring Protocol implementation and P2P Protocol Layer	33
Figure 5-2 – Relationship between Viceroy DHT implementation and P2P Protocol Layer	35
Figure 5-3 – Setup P2P and network environment.	38
Figure 5-4 – Using Pub/Sub API for Publishing.....	40
Figure 5-5 – Using Pub/Sub API for Subscribing.....	41
Figure 5-6 – P2P and Network Environment Options.	42
Figure 6-1 – Integrated with OSGi platform.....	49
Figure 7-1 – Concept of Simple Pub/Sub Algorithm.....	57

List of Tables

Table 2-1 – List of famous DHT schemes.	6
Table 5-1 – Task Descriptions of Developing P2P Protocols	32
Table 5-2 – Task Descriptions of Developing Pub/Sub Services.....	36
Table 6-1 – Possible Usage of Send Policy	47
Table 7-1 – Pseudo code for the node join and lookup operation.....	50
Table 7-2 – Pseudo code of stabilization and neighbor update operation.....	51



1. Introduction

Nowadays, with the computing power of PC and network bandwidth increasing, people are willing to dispense their computing power and share information with each others. In *pure* P2P network, each participant shares their resources in order to gain benefits from other peers. By the natural of sharing in P2P networks, the more users joining the network, the more capacity this P2P network obtained. The scalability is based on the performance of P2P protocols, not determined by the server capacity in traditional centralized architecture. The P2P networks are proved to be an alternative technique in distributed information processing [14]. In addition, the ownership of shared resources and the right to distribute are possessed by the user in the P2P network as opposed to typical central server system where the user grant the service provider the rights of using and distributing resources.

In order to construct an efficient and scalable P2P network, many structured P2P network have been proposed these days and have been verified as efficient and fault-tolerated in large distributed environment. Most of them, e.g. Chord [33], Pastry [31], Viceroy [16], etc., are able to route message between two peers in $O(\log N)$ hops where there are N peers within the network. With the feature of self-organize and failover, structured P2P networks have been widely used in file sharing [13][25], network data storage [6], and distributed indexing [32]. There are several research works on deploying distributed personal information portal [20] and online auction systems [10] onto the P2P networks.

Publish/Subscribe paradigm is effective in disseminating information to peers who are interested in. In order to apply this mechanism on the P2P network,

P2P pub/sub algorithms are designed with the consideration of both time efficiency and transmission overhead. Efficient pub/sub algorithms are able to alleviate the communication burden when dealing with the burst of information on a large scale P2P network.

As investigated the research topic of structured P2P networks, however, each P2P network was implemented under different approaches, providing various application interfaces. A standardized development and deployment framework can reduce the overheads of implementing P2P protocols and applications. Therefore, developers can focus on the applications' unique functionalities.

1.1. Motivation

In the application domain of content management system, e.g., personal blog system, large amount of information are created and requested over the entire user community. With the search capability, users can retrieve information which has particular contents according to given query. As the P2P community keeps advancing, however, the number of updating events will soon overwhelm the size of events that human can handle. By introducing pub/sub mechanism, applications can automatically disseminate information to the interested peers in P2P network. Like the RSS supported on many website, the pub/sub paradigm provides the functionality for users focusing on only the interested events. Therefore, pub/sub mechanism is an essential feature while designing a platform for developing P2P applications.

There are three aspects of developing a P2P Pub/Sub-related program: application developers, P2P protocol developers, and P2P pub/sub protocol developers. From the aspect of developing pub/sub applications, programmers usually need to learn new APIs when changing the underlying overlay network.

The difference of semantics can reside in peer initialization, network construction, and even communication mechanism; that is, implementing the same functionality on different P2P APIs could cause code rewriting. The same situation happens in changing pub/sub APIs. This means application would be strong coupled with P2P and pub/sub implementations. Application developers have no chance to compare the performance of their systems on different overlays.

- ***Issue 1a: Application developer need to learn different semantics from numerous P2P APIs.***
- ***Issue 1b: The cost of rewriting code is huge for testing performance of particular application on different P2P network.***

For p2p pub/sub algorithm developers, the lack of a common platform for evaluating performance makes it hard to compare between algorithms. First, preparing identical test case on two different p2p pub/sub systems is cumbersome. Second, the delay of event dissemination needs to normalize due to the different implementation of internet communication.

- ***Issue 2: P2P pub/sub algorithm developers need a common platform to compare with other algorithms.***

While developing a P2P algorithm, developers writing their own code communicating with other peer through physical network connection. Each P2P API introduces redundant code on network programming. Developers take additional time on debugging network-related code. Without network-related code reusing, the effort for extending deployment environment is huge.

- ***Issue 3a: P2P network developer write redundant code for network communication, make it hard to deploy P2P on different physical network environment.***

- *Issue 3b: A common process is needed for overlay network initialization.*

Our goal is to solve these issues mentioned above. Thus, a standardized API and communication mechanism for P2P application development is need to be defined.

1.2. Contributions of this thesis

In this thesis, a generic development framework for P2P applications is proposed. With the design of multi-layers abstraction, P2P application developers can deploy their application to different kinds of P2P overlays and physical network environments. In additional, this framework can help developers realizing the P2P protocols and creating value added pub/sub services. We define a general pub/sub service SPIs which focused on deploying pub/sub mechanism over entire P2P network. The layered design of this framework encourages that developers create reusable components. Moreover, an execution configuration module is provided which can externalize parameters to customize for different environment constraints without recompiling programs.

1.3. Thesis Outline

Chapter two introduces the background knowledge and shows the previous researches in defining common API for P2P programming and pub/sub application. In Chapter three, a layered architecture and primary interfaces are described in detail. The interaction between modules and the usage of components are further described in chapter four. Chapter five demonstrates the usability of this framework and shows the pros and cons by comparing with existing solutions. In the end, future work and conclusion are given in section six.

2. Background and Related Work

This chapter covers the definition of structured P2P network and introduces the types of pub/sub algorithm. We also describe previous research on defining common API for P2P network and pub/sub system. The comparison between previous research and our work will be briefly described in this chapter.

2.1. Background

2.1.1. Structured P2P Network

P2P network is a virtual network that consist an amount of peers. Each peer links to a subset of peers on the network and communicates with each other through a specific routing algorithm. In structured P2P network, each peer is mapped to a peer id within a large identifier space. This identifier space defines the distance metrics between ids. Links between peers are determined by the distance. Each resource, such as files, is assigned a unique key from the same identifier space. With the mapping from key and resource, the structured P2P network naturally organizes as a distributed hash table (DHT).

In order to accommodate with scalability, structured P2P networks are usually designed with four criteria: low degree, low diameter, greedy routing, and robustness [28]. Many different topologies are used in structured P2P protocols and keep peers knows only local information. These common P2P topologies are depicted in Figure 2-1.

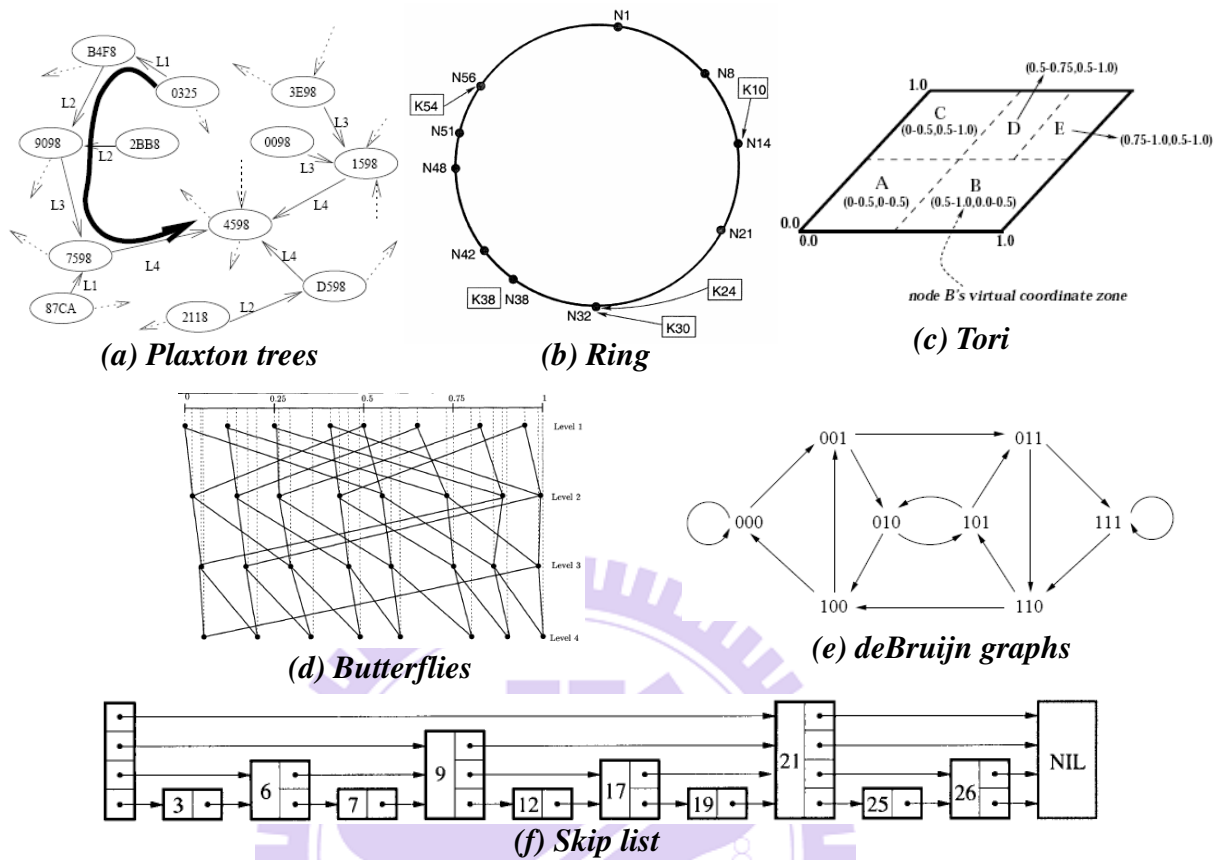


Figure 2-1 – Common Structured P2P Topologies.

The capability of DHT protocols are affect by the underlying topology.

Table 2-1 lists the famous DHT schemes with their capability and corresponding topology.

<i>DHT scheme</i>	<i>Topology</i>	<i>Degree</i>	<i>Diameter</i>
Pastry, Tapstry[36], Kademia[19]	Plaxton tree[24]	$O(\log N)$	$O(\log N)$
Chord	Ring	$O(\log N)$	$O(\log N)$
CAN[27]	Tori	$O(d)$	$O(dN^{1/d})$
Viceroy	Butterflies	$O(1)$	$O(\log N)$
D2B[9], Koorde[12]	de Bruijn graphs	$O(1)$	$O(\log N)$
Skip Graph[2]	Skip list[26]	$O(\log N)$	$O(\log N)$

Table 2-1 – List of famous DHT schemes.

2.1.2. P2P Publish/Subscribe Algorithm

Publish/Subscribe mechanism involves two roles of actor, publisher and subscriber. Publishers generate events associated with topic or tagged with properties. Subscribers register with interested topics or properties. Subscribers receive only matched events. A message routing protocol determines how to disseminate events to interested subscribers. In pub/sub messaging system, publisher and subscribers are generally anonymous and can dynamically publish and subscribe with given topic or properties. Publishers and subscribers are loosely coupled, without addressing message receiver directly. The pub/sub algorithm can be categorized into two major models [29]:

A. *Topic-based model*

In topic based model, publishers and subscribers are associated with a channel by given topic name. Subscriber can only filtering events by topic name, which are considered less expressive. Therefore, a hierarchical topic space is used to provide more expressiveness. Topics are organized in a hierarchical structure, i.e., a topic can be defined as a sub-topic. An event associated with a particular topic is conceptually associated with its super-topic. A subscriber can receive both events associate with interested topics and its sub-topics.

B. *Content-based model*

In this model, an event is published with several properties denoted. Subscribers register their interested values of certain properties. When an event is disseminated over P2P network, a distributed filtering mechanism is used to limit transmission to the set of interested subscribers.

Establishing pub/sub mechanism on P2P network provides additional scalability and load balancing. Moreover, the single point failure in traditional

server-based pub/sub system is reduced by the fail-over mechanism provided in the self-organized P2P network. In previous study, two design patterns are identified to implement P2P pub/sub algorithms [3]. In the *Store-Sub* paradigm, the subscribers store their subscriptions in the DHT network. While publishers publish an event, all subscribers are retrieved and the event can then be disseminated to each of them. On the other hand, the *Store-Pub* paradigm aggregates publisher into the distributed directory based on the previous publishing events. Publishers announce their existence and the DHT network maintains the statistical profile of publishing history. These two design patterns accommodate to different pub/sub scenarios.

2.2. P2P Common API

To facilitate independent innovations in P2P protocols, services, and applications, Debak et al. [7] propose a common API for structured overlays. Following research revises this API with the request-response communication pattern [5]. Moreover, a conceptual model for structured P2P network is proposed by Aberer et al. [1] to provide interoperability between decentralized overlay networks. These researches focus on providing a standardizing P2P network API to application developers.

JXTA [34] is a platform for peer-to-peer computing, proposed by open source community. The JXTA protocols are a set of six protocols that standardize the behaviors between peers. In order to provide interoperability in different language and network environment, JXTA protocol uses XML messages and the super-peer architecture. The index information is also stored within the super-peers, providing reliability and supporting heterogeneous nodes which installed different set of services. JXTA achieves a great success as a P2P application platform, but offers no high level support for structured P2P topology.

2.3. Publish/Subscribe Common API

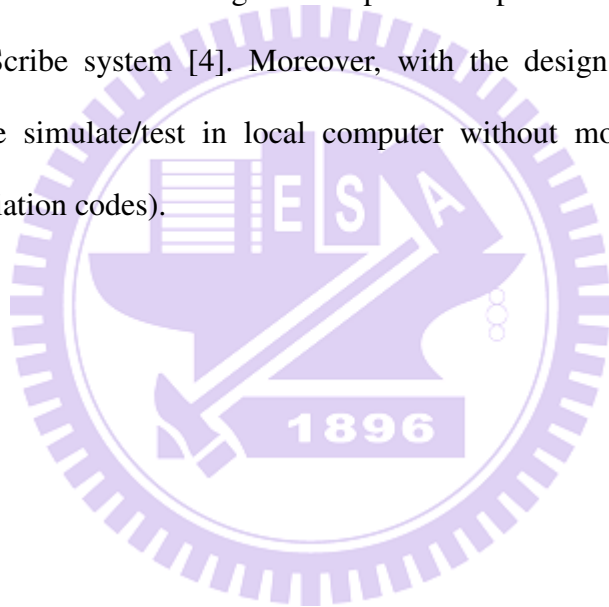
Java Message Service (JMS) [35] is a part of standard service that included in Java EE platform. JMS defines the common set of interface and associated semantics. By JMS provider implementing the standard API, developers can easily deploy programs with different messaging server. JMS provide two messaging domain:

- A. *Point-to-Point Domain*: This messaging domain is built on the concept of message queue. Each message has only one consumer. The point-to-point messaging is used when every message must be processed successfully by one consumer.
- B. *Publish/Subscribe Domain*: This domain is defined with topic-based model. In addition, JMS API defines an SQL-like selection language and provides a built-in facility for supporting application-defined property values.

However, the JMS API is a proprietary specification for Java to intercommunication with messaging server. In order to provide a lightweight API, Pietzuch et al. [23] define a simplified abstraction for pub/sub system. This common API uses XML-RPC to describe the interaction, preserving the interoperability with other languages and platforms. With little efforts, this API shows that many pub/sub systems can be brought to compliance. These pub/sub APIs assumed that both publisher and subscriber are clients to a messaging service. Therefore, an auxiliary server is required for delivering messages.

2.4. P2P Pub/Sub Library

Developing the P2P routing protocols and pub/sub systems is cumbersome task requiring sophisticated testing on scalability and reliability. P2P application developers tend to implement their system using a P2P library. FreePastry [22] is an open source P2P library which provides pub/sub functionality. The FreePastry implements the Pastry network routing protocol intended for deployment in the Internet. Based on the Pastry network, additional functionalities are built, such as pub/sub system and distributed storage. The topic-based pub/sub system supported in FreePastry is Scribe system [4]. Moreover, with the design of peer factory, application can be simulate/test in local computer without modifying program (other than the initiation codes).



3. System Architecture

In this chapter, we briefly introduce our system architecture. According to the issues described in chapter one, our system needs to provide sufficient abstraction to cover the common functionalities in P2P routing protocol and pub/sub protocol. The following sections will point out how we achieve the goal of design a general and flexible development framework.

3.1. Overview

Previous research of common P2P API shows the common functionality of structured P2P networks. Inspired by FreePastry and PeerSim [18], we further extend the P2P API by abstracting the physical network communication from P2P protocols and introduce additional bootstrapping facility. A standard pub/sub API is designed to accommodate with heterogeneous pub/sub model in pure P2P networks.

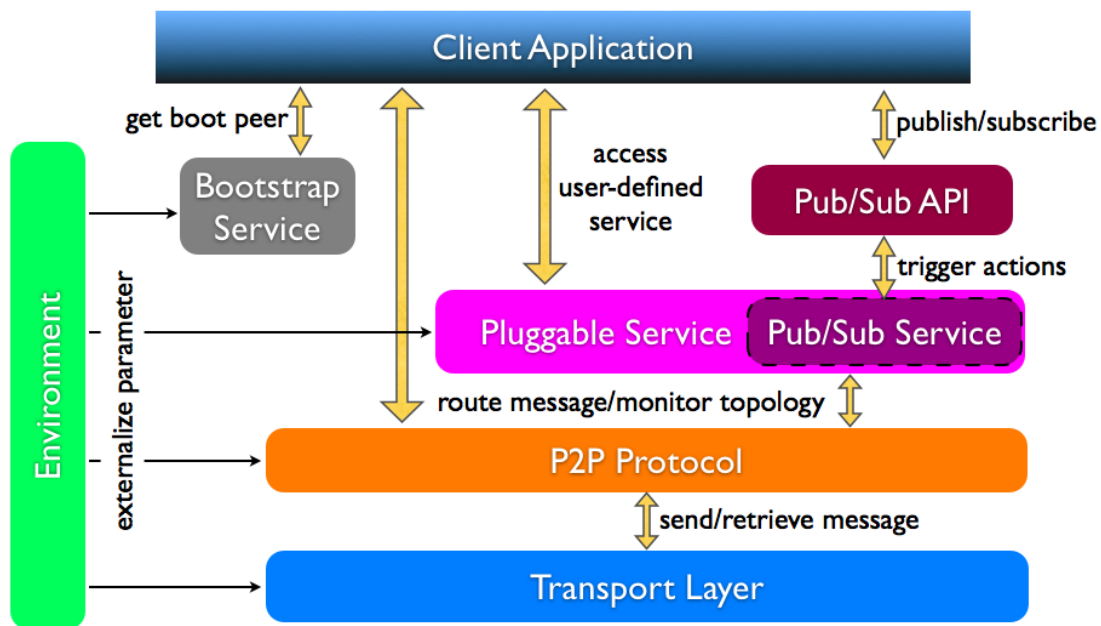


Figure 3-1 – System Architecture Overview.

P2P Applications retrieve a live peer in the P2P network through *Bootstrap Service*. This live peer is used to initiate the join operation. Application can directly access the *P2P Protocol Layer* for message routing and performing lookup operation. By registering *Pub/Sub Service* to local peer, applications use *Pub/Sub API* to do event publication and subscription. P2P Protocol Layer delegates the physical network transmission to *Transport Layer*. *Environment* module loads external parameters from configuration file.

3.2. P2P Protocol Layer

This is the core layer of performing structured P2P functionalities. In P2P Protocol Layer, we propose an object model to describe the relationships within structured P2P network components. This object model consists of the interfaces of common P2P functionalities, peer initialization, and constraints of generating topology. By implementing these interfaces, P2P network library developers are able to create a particular routing protocol.

`Peer` exposes the common API for general purpose P2P network accessing. Each `Peer` associates with a `PeerId` mapping to identifier space and a `CommunicationManager` for network accessing. The identifier space contains `Id` for the general key to any `Resource` and subclass `PeerId` for identifying peers. `NodeHandle` is a peer reference to be used for remote peer communication and topology maintenance.

We use Abstract Factory pattern to standardize the process of id creation and peer initialization. `PeerFactory` and `PeerIdFactory` define the interface for create peer instance and assign a unique peer identifier. `IdFactory` consists of the methods generating the key for resources.

The `Service` interface is defined for create user-defined application that can monitor the activities of P2P network. In order to achieve the goal of define pluggable pub/sub service, we introduce the `Service` interface that can receive certain events while a message arrived and topology changed. With the service registration mechanism, developers are free to implement additional functionalities without polluting the code of P2P protocol.

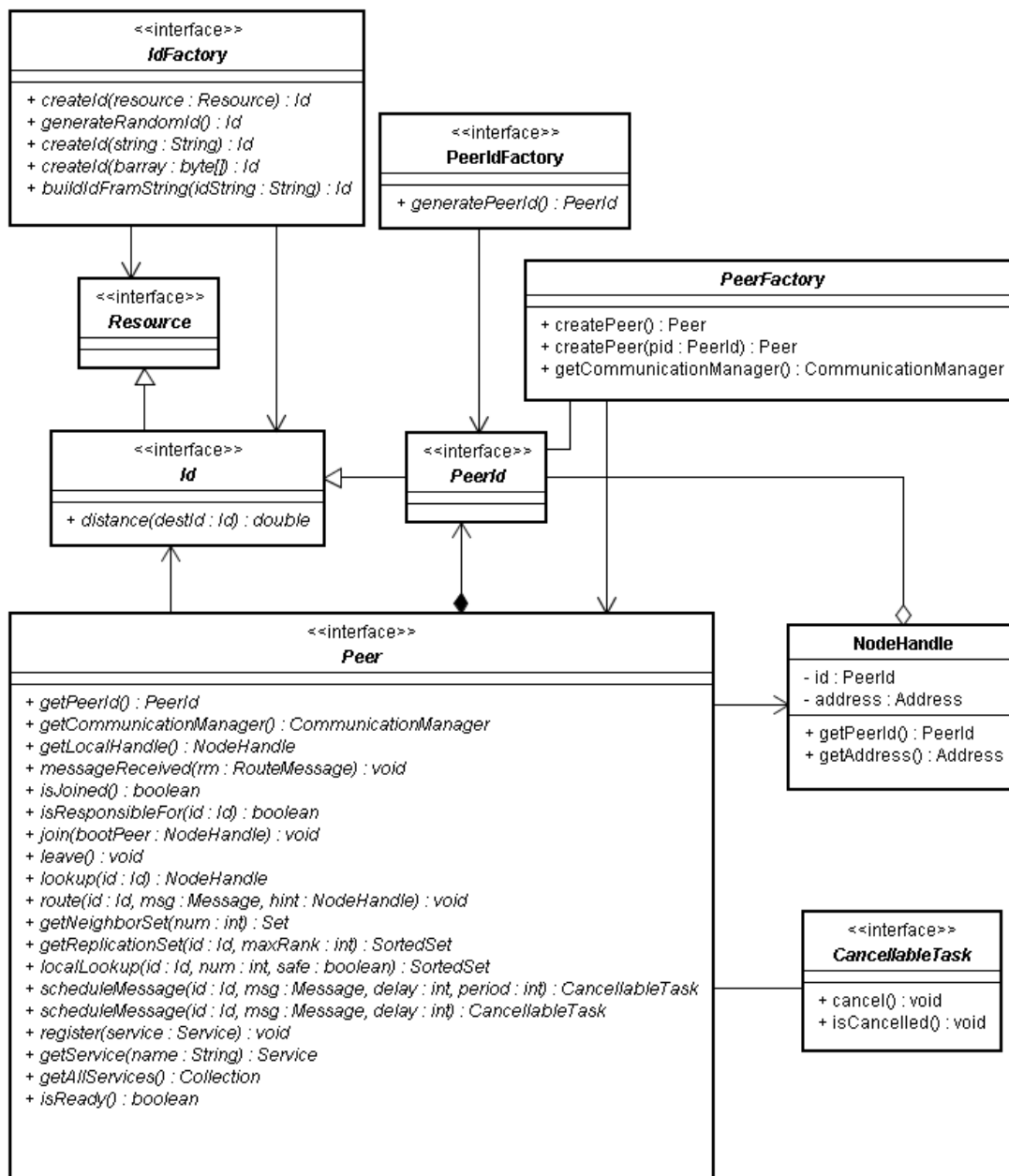


Figure 3-2 – Class Diagram of P2P Protocol Layer

3.2.1. Peer interface

The method defined in `Peer` interface can be categorized in three groups of operations. The first group accesses basic attributes and status.

```
public PeerId getPeerId();
public CommunicationManager getCommunicationManager();
public NodeHandle getLocalHandle();
public boolean isJoined();
public boolean isReady();
```

The `getLocalHandle()` operation are used to create a transferable peer reference that represents current node. The method `isJoined` is used for determining if the peer is in a correct state to perform P2P functions. The `isReady` method determines if the peer is ready to process message received from transport layer.

The second group defines the common P2P API. Developers use these methods to perform message routing and to explorer P2P topology.

```
public void join(NodeHandle bootPeer);
public void leave();
public NodeHandle lookup(Id id);
public void route(Id id, Message msg, NodeHandle hint);
public boolean isResponsibleFor(Id id);
public CancellableTask scheduleMessage(Id id, Message msg, long delay);
public CancellableTask scheduleMessage(Id id, Message msg, long delay,
    long period);
```

With the `join` and `leave` operations, peers can either participate or leave the network. After successfully joining a P2P network, the return value of `isJoined` should be always true. The `scheduleMessage` methods are used to perform asynchronous messaging. Developer can use these functions to establish network stabilization procedure.

The third group is used for registering user-defined services.

```
public void register(Service service);  
public Service getService(String serviceName);  
public Collection<Service> getAllServices();
```

Arbitrary number of services can be registered in one peer. Each service is identified with given service name. The two getter functions, `getService` and `getAllServices`, are primarily used in routing service specific message and notify the event of topology changing.

3.2.2. PeerFactory class

`PeerFactory` provides the standard API for creating a new peer and for restoring peer with existing peer id. Peer initialization is hidden behind the implementation of `PeerFactory`. The physical network module must be determined while constructing an instance of `PeerFactory`. All peers created from this factory are registered to the same network module for further communication.

3.2.3. Resource, Id, and PeerId interface

`Resource` is a marker interface, denotes a class of object that can be distributed across the P2P network. By extending `java.io.Serializable` interface, resources can naturally be transferred over the network. `Id` represents the identifier space, with a distance function defined. The peer identifiers can be a sub space of identifier space. Thus, `PeerId` is a subclass of `Id`, represents the identifier of each peer.

3.2.4. IdFactory and PeerIdFactory interface

These two factory interface are in control of generating and computing id for distributable resources. Despite of invoking constructor directly, using factory methods provides standard interfaces for creating id.

3.2.5. CancellableTask interface

This interface represents an asynchronous message routing task and isolates the implementation of job scheduling. While using the functionality of message scheduling, `CancellableTask` provides the functionality to peek the status of asynchronous task and to interrupt it.

3.2.6. Service interface

A *service* is a plug-in for establishing additional protocols on top of existing P2P networks. This interface defines callback functions for processing message routing and handling. In addition, the topology changing events of peers are propagated to installed services. With topology awareness, services are able to implement fail-over mechanism without periodically polling the information of neighborhoods.

3.3. Pub/Sub Service and API

This module provides a light-weight API for executing pub/sub related task. `Publisher` and `Subscriber` define the common pub/sub API that can connect with arbitrary pub/sub service. Each `Publisher` and `Subscriber` is associated with one topic.

The PubSubService is a subclass of Service that defines the SPI needed for implementing P2P pub/sub algorithms. PubSubService receives the actions from pub/sub applications via Publisher and Subscriber. In order to accommodate to both topic-based model and content-based mode, the pub/sub API is designed with topic-based model and additional selector language like the one used in JMS for attribute filtering. The content-based model is also supported by introducing a wildcard topic. Pub/sub client program receives interested event via registering EventHandler.

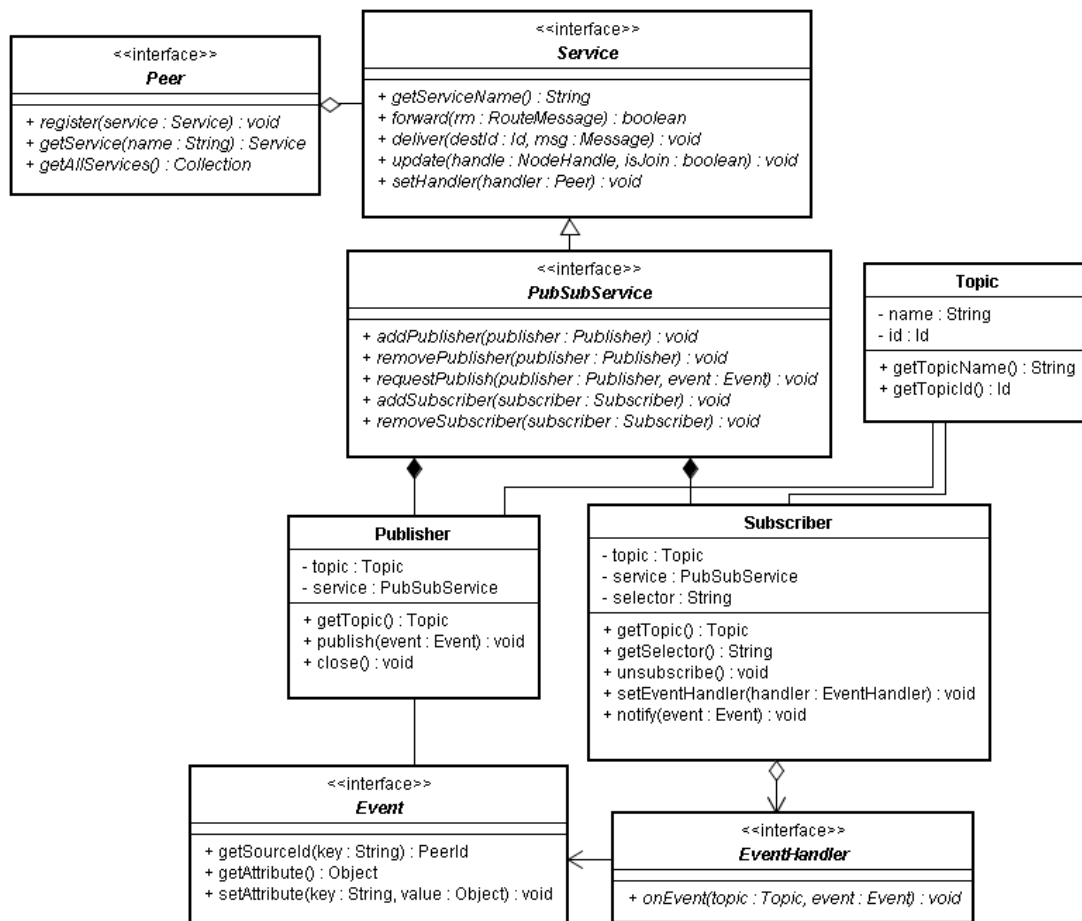


Figure 3-3 – Class Diagram of Pub/Sub Service and API.

3.3.1. PubSubService interface

The PubSubService interface extends the Service interface with special pub/sub related functions. According to Bender’s work [3], both Store-Pub

and Store-Sub patterns can be used in implementing pub/sub services. Therefore, we define five common operations for accommodating these two approaches:

```
public void addPublihser(Publisher publisher);  
public void removePublisher(Publisher publisher);  
public void addSubscriber(Subscriber subscriber);  
public void removeSubscriber(Subscriber subscriber);  
public void requestPublish(Publisher publisher, Event event);
```

These add/remove methods are invoked while publishers and subscribers are joined or left. The P2P pub/sub service providers implement these methods to maintain the information of publishing and subscribing in the P2P network. The `requestPublish` method is invoked while a publisher requests for event disseminating.

3.3.2. Publisher and Subscriber

Developers use `Publisher` and `Subscriber` to access pub/sub systems. `Publisher` and `Subscriber` are both bind with an instance of `PubSubService` at runtime. While publishing an event, `Publisher` delegates this operation to the binding pub/sub service. For adapting to various pub/sub system implementations, `Publishers` are required invoking `close` method to explicitly terminate the publishing session.

In order to adapt both topic-based model and content-based model, we provide two methods to describe users' subscription: *topic* and *attribute selector*. Pub/sub services can provide their own selector string format for further attribute filtering or supporting content-based subscribing.

3.3.3. EventHandler interface

EventHandler defines a callback function that applications can be notify of the arrival of events:

```
public void onEvent(Topic topic, Evnet event);
```

The onEvent method receives the coming event and the topic belongs to as parameters. Application-specified tasks are defined within this function. With topic information and user-defined event properties, developers can aggregate the event processing in single callback.

3.3.4. Event interface

This interface defines the basic operations which can retrieve general information, such as source id and attributed. The attributes of event can be used in content-based pub/sub algorithm and event filtering mechanism. The following methods are defined in this interface:

```
public PeerId getSourceId();
public Object getAttribute(String key);
public int getIntAttribute(String key);
public long getLongAttribute(String key);
public float getFloatAttribute(String key);
public double getDoubleAttribute(String key);
public void setAttribute(String key, Object obj);
public void setIntAttribute(String key, int i);
public void setLongAttribute(String key, long l);
public void setFloatAttribute(String key, float f);
public void setDoubleAttribute(String key, double d);
```

3.4. Transport Layer

The transport layer encapsulates the detail of resolving physical address and establishing connection. The `CommunicationManager` is the representative of physical network infrastructure. Through the abstraction of network communication, P2P protocol can easily deploy on different network environment. In our design, peers can register to one single instance of `CommunicationManager`, reducing the overhead of activating multiple P2P networks. `CommunicationManager` uses `Address` to establish network connection in order to perform message transmission. Peers communicate with each other by sending message. `Message` interface defines the essential attributes for determine the source peer and the message handler. Figure 3-4 shows the relationship within transport layer.

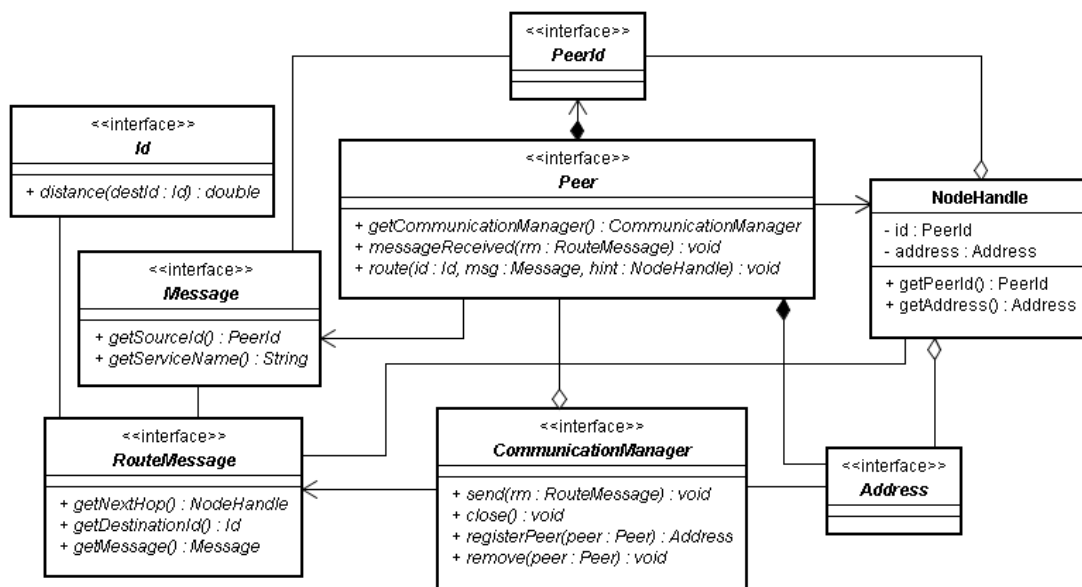


Figure 3-4 – Class Diagram of Transport Layer.

3.4.1. CommunicationManager interface

The `CommunicationManager` contains methods handling message transmission. Peers retrieve the physical address by registering themselves to `CommunicationManager` and detach from network by remove itself from `CommunicationManager`. The `close` operation is used for cleaning up resources, such as network connection and listening port. Through the `send` operation, messages can be sent with destination id and next hop information. The implementers of the `CommunicationManager` interface are responsible for resolving network address and establishing connections.

3.4.2. Address interface

This interface denotes the real address that one can use to communicate through underlying network environment. Each implementation of `CommunicationManager` is responsible for providing the physical address, which implements the `Address` interface.

3.4.3. RouteMessage interface

`RouteMessage` encapsulates the required information for transport message between sites. The *message* stores the content that needs to be transferred. The message receiver is designated as the peer responsible for *destination id*. P2P protocol and service specified the *next hop* for message routing.

3.4.4. Message interface

In our framework, peers are communicated with each other by messaging. The `Message` interface represents the information that exchanged over P2P

network. The peer id of message source is able to retrieve from message. With service name specified, a user-defined service can be designated as the handler of arriving message.

3.5. Bootstrap Service

In order to join an existing overlay network, peers must know a live peer on that network. The bootstrap service provides a general interface that can adapt to different service implementations.

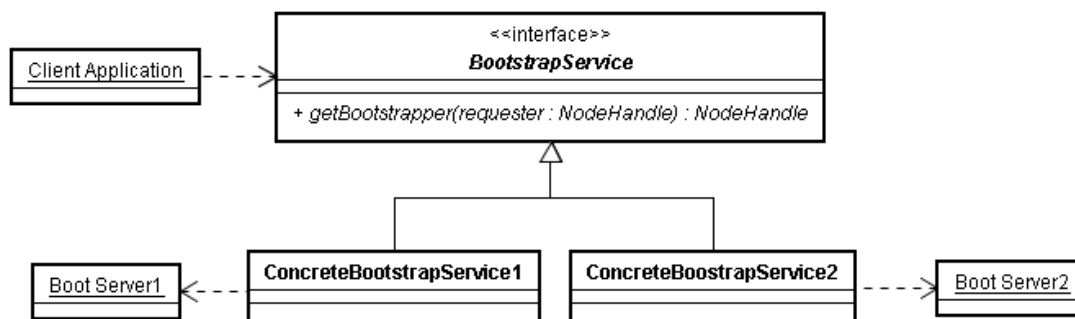


Figure 3-5 – Class Diagram of Bootstrap Service

3.5.1. BootstrapService interface

The class `BootstrapService` is a façade to external bootstrap service instance. The `getBootstrapper` method encapsulates the communication protocol between client and bootstrap server. The parameter of `getBootstrapper` method is the self reference of requesting peer. Bootstrap server can then take the request information to select a nearest boot peer for better join performance, or to promote requester as a boot peer while deploying a novel P2P network.

4. Implementation Details

This section describes the interaction between components that mentioned in chapter 4, including: peer bootstrapping, pub/sub actions, and message processing. The detail control flow of this framework and the usage of each class are covered in following sections.

4.1. Control Flow

4.1.1. Peer Bootstrapping

While performing network bootstrapping, an external boot server is required to retrieve the P2P network information for the nodes outside. Application initiates a BootstrapService instance that implements particular communication protocol for boot server. An active peer is returned by invoking getBootstrapper method. Through the active node, join request can be dispersed among peers in the network.

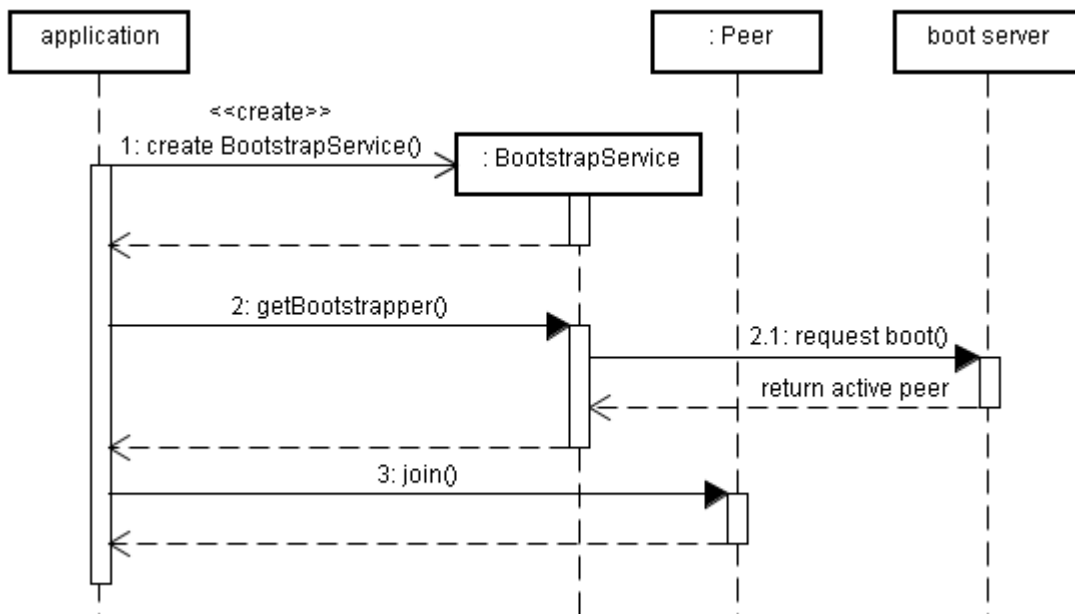


Figure 4-1 – Sequence Diagram of Peer Bootstrapping

4.1.2. Message Transmission

When the route operation has been invoked, the route message is prepared and is sent by `CommunicationManager`. According to the next hop address, route message can be transferred to the `CommunicationManager` at remote site. `CommunicationManager` receives the route message and then notify the handle peer using the `messageReceived` callback. The destination id will be checked first, see if current peer is responsible for the arriving message. If this arriving message doesn't specify any service handler, peer will directly forward this message to next hop, using `route` function. Otherwise, the `handleMessage` method will take over this message.

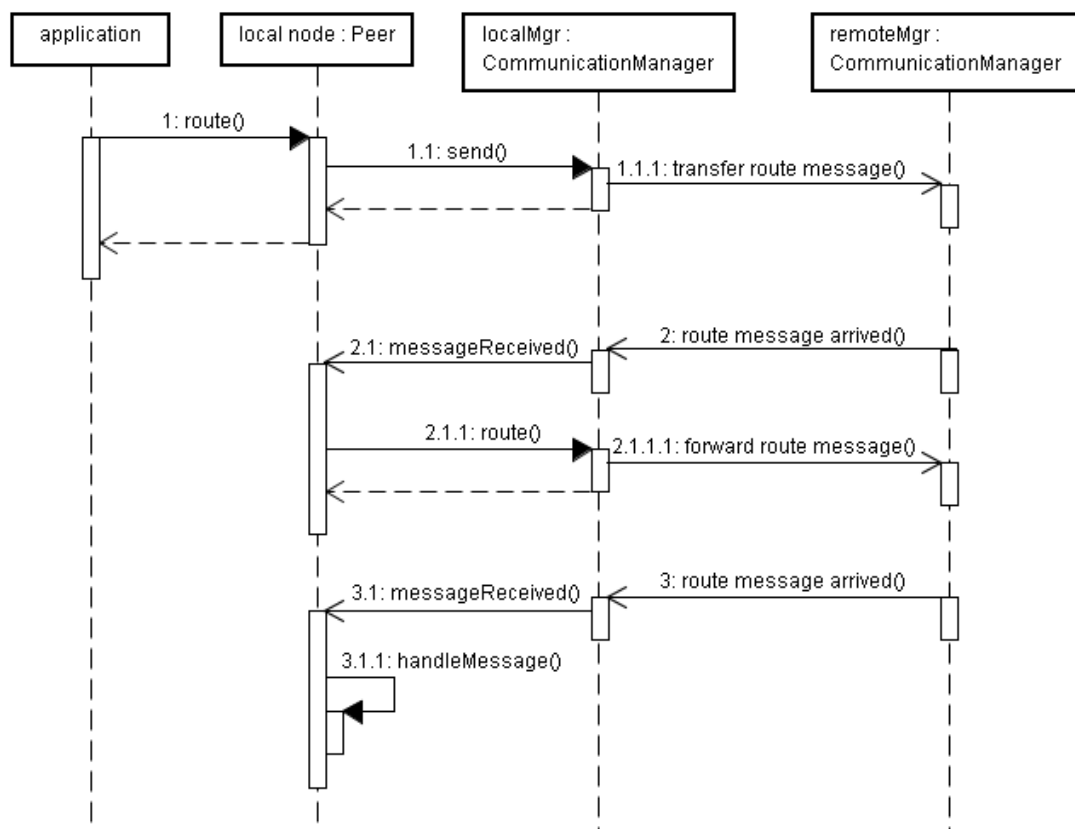


Figure 4-2 – Sequence Diagram of Message Routing

If the message is intended for processing by designate service, the control will hand over to the specified service. The following action is decided whether

current peer is the destination or not. The `forward` function is invoked to determine the next hop. The permission of forwarding current message can be decided by the return value of `forward` method. While message is delivered to its destination, peer will invoke the `deliver` method of corresponding service instance.

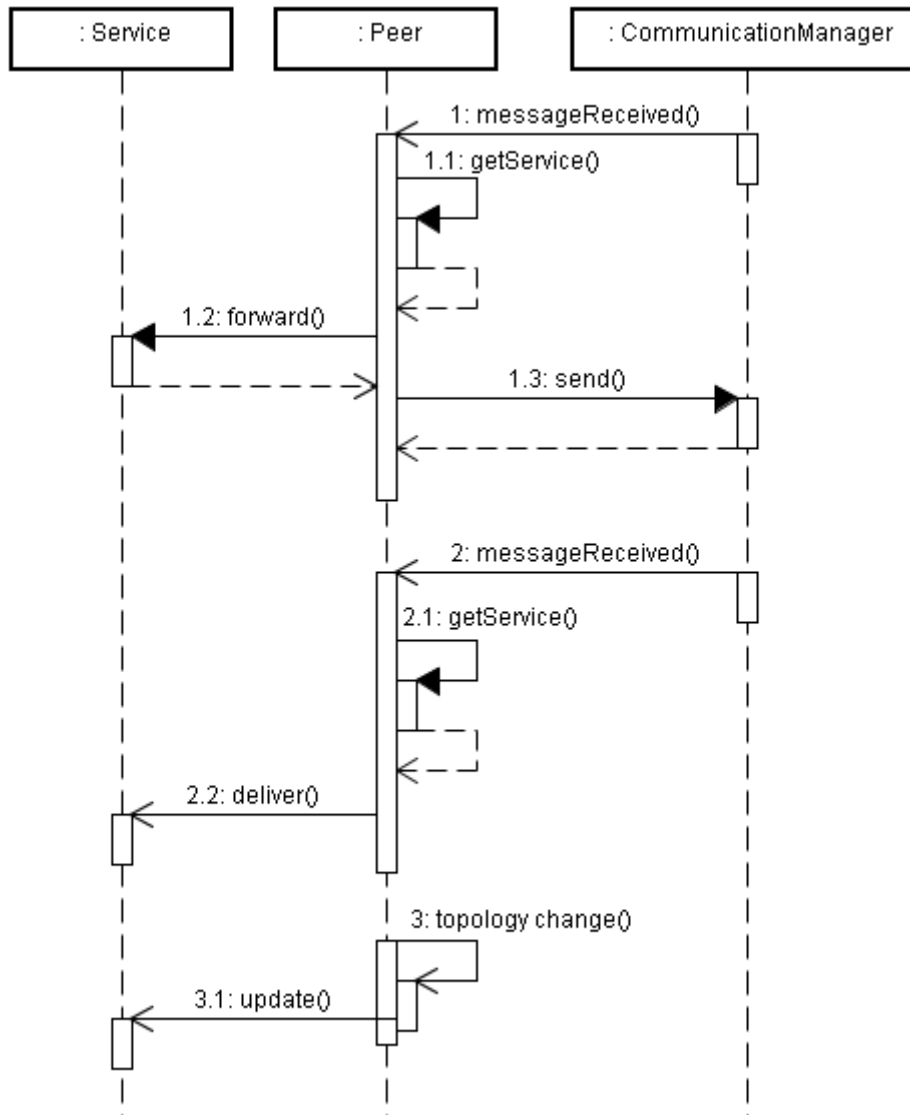


Figure 4-3 – Sequence Diagram of Service Callbacks

The `update` method will be invoked by the P2P protocol developer when the topology is changed. Services can perform data migration or backup whether a node is joined or not.

4.1.3. Pub/Sub Actions

When an instance of `Publisher` is created, the `addPublisher` function will be triggered and pub/sub service can then perform publisher join operation. While an event is published through associated `Publisher`, the pub/sub service will invoke the `requestPublish` method. This event will then be disseminated on the network according to the pub/sub algorithm implemented in the associated pub/sub service. The `Publisher` class also provides a `close` function that explicitly notifies the leave of a publisher.

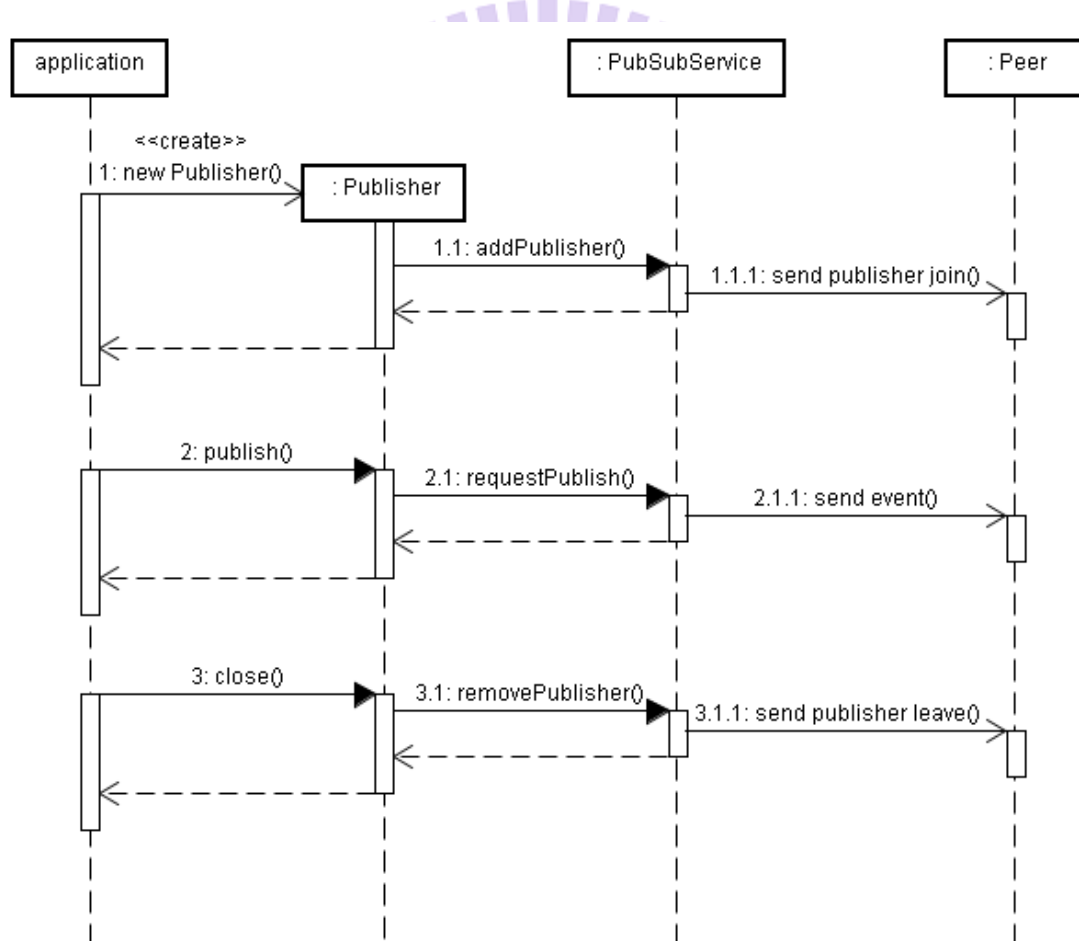


Figure 4-4 – Sequence Diagram of Publishing

The subscription is sent while application creates a new instance of `Subscriber` via the `addSubscriber` method. In this framework, the event retrieving is implemented in event-driven fashion. With event handler being

specified, application will be notified when an event is arrived. To revoke subscription, the unsubscribe method is invoked and pub/sub service will send out the request of unsubscribing.

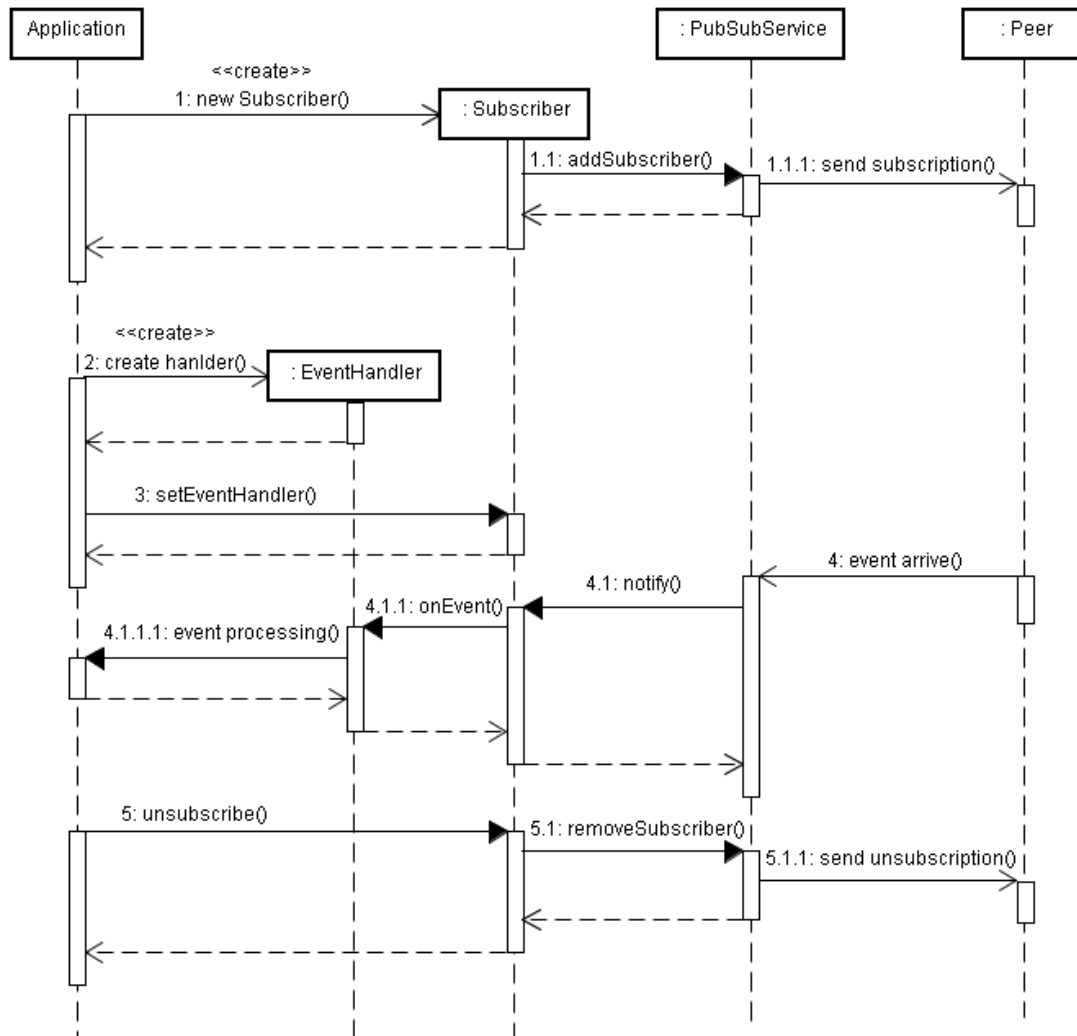


Figure 4-5 – Sequence Diagram of Subscribing

4.2. Additional Class Usage

4.2.1. Environment

This class setups the execution context for initializing components. It also provides utilities for externalizing parameters from configuration files and plain

code. The configuration file format is the same as of Java properties file, contains simply lines of name-value pairs.

4.2.2. AbstractPeer

This abstract class implements the `Peer` interface, provides default work flow of message routing as that is listed in section 4.1.2. By using JavaSE 5.0 concurrency package, we provide the implementation of the `scheduleMessage` methods. The `AbstractPeer` also provide additional `scheduleTask` methods that can register tasks which run asynchronously or periodically. The `join` and `leave` operations are override in this class for deferring scheduled message and task until node joined, and provide a `joinImpl` method for protocol developers implementing the core join algorithm. The P2P protocol developers should extend this class to create their own protocol implementations.

4.2.3. AbstractEvent

We provide the implementation of the `Event` interface for accessing basic attributes such as event source and custom properties. Application developers can extend this abstract class and create their application specific event type.

4.2.4. NodeHandle

`NodeHandle` represents the peer reference and contains information that can uniquely identify and connect to certain peer, including peer id and physical address. This class is a distributable resource, which implements the `Resource` interface.

4.2.5. Topic

The Topic object is used to represent the associating channel of publishers and subscribers. Each topic is identified with a topic name. In order to isolate the event topic from particular identifier space, this class provides two constructors for properly generating corresponding topic key.

```
public Topic(IdFactory factory, String topicName);  
public Topic(Id id, String topicName);
```

The application developers formally pass an instance of `IdFactory` as a parameter in order to generate a valid hash key from given topic name in target identifier space. For debugging convenience, an additional constructor is provided which directly specifies the corresponding topic id.

4.2.6. LocalBootstrapService

The `LocalBootstrapService` is used in bootstrapping P2P nodes on single VM. This bootstrap service requires no external boot server. This class is a utility for testing applications without deploy additional service.

4.2.7. HttpBootstrapService

We provide a HTTP-based boot server and the `HttpBootstrapService` cooperated with that allow peers retrieve bootstrapping information from network. The URL of boot server can be specified in Environment configurations. The peer reference of requesting peer is serialized and is encoded in Base64 code format, which can be append in the HTTP Post request body. The boot server accepts additional domain parameter that can differentiate boot request from different P2P

network. By the nature of HTTP protocol, developers can adapt to different boot server implementation, as long as the server obey the request message format.

4.2.8. LocalCommunicationManager

Peers that setup in the same VM environment can be addressed using LocalCommunicationManager. This communication manager forces messages sent in sequential order, it can be used in protocol simulation and debugging. Since both message source and destination are resident in the same virtual machine, the I/O exception will not occur during message transmission. This component is useful in early stage of development process.

4.2.9. TCPCommunicationManager

The TCPCommunicationManager allows message transmission through TCP/IP network. The address create by TCPCommunicationManager contains the information of IP address and port are listened, which allows peer startup on different port. This component has a configurable size of thread pool used for consuming incoming message.

5. Evaluations

This chapter starts with how to develop reusable components for P2P network and flexible pub/sub application through the framework we proposed. Two P2P protocols and A P2P pub/sub algorithm are used in order to present the design flow from pseudo codes to correctly executable programs. Section 5.2 gives a detailed comparison with previous researches and describes the pros and cons of this framework.

5.1. Scenario Demonstration

5.1.1. Implementing P2P Protocol

The fundamental elements of a structured P2P protocol consist in defining overall network structure, basic P2P operations, and fail-over mechanisms. The following table lists the tasks for implementing P2P protocol using our framework.

<i>Task</i>	<i>Remarks</i>
define id space	Create corresponding <code>Id</code> and <code>PeerId</code> class, which define the distance function. Implement <code>IdFactory</code> and <code>PeerIdFactory</code> for creating identifier.
define topology	Implement <code>Peer</code> interface with routing table information. Determine the neighbor set and replication set.
peer initialization	Define externalized parameter name and type for environment configuration. Implement <code>PeerFactory</code> for creating peers.
join operation	Define message format of join request and response. Implement associated action in <code>joinImpl</code> method and provide corresponding message handling procedures in <code>handleMessage</code> method. Change peer status after join operation finished.

lookup operation	Define message format of lookup request and response. Implement <code>route</code> and <code>localLookup</code> method for determine routing path. Implement associated action in <code>handleMessage</code> method. Define request time out mechanism to prevent thread locking.
leave operation	Define leave request format, carrying the information of topology correction.
stabilization	Managing periodical probing task using <code>scheduleMessage</code> method, <code>scheduleTask</code> method and <code>CancellableTask</code> class. Should be tolerated on every possible exception.

Table 5-1 – Task Descriptions of Developing P2P Protocols

Here we present two reference implementations that show how to implement a P2P protocol. First, we implement *Ring Protocol*, which is a simplified version of Chord. Like Chord, Ring Protocol maps both peers and resources in to an m -bits, ordered identifier circle. Each peer in Ring Protocol maintains the link to its predecessor, successor, and K random peers on the network. The details of Ring Protocol are described in Appendix A.

`RingIdFactory` and `RingPeerIdFactory` are created using MD5/SHA-1 hash algorithm to produce corresponding `RingId` and `RingPeerId`. Because both type of identifier mapping to the same id space, `RingPeerId` is simply a subclass of the `RingId` class. A `NeighborTable` is introduced to manage the information of random neighbor table, which provides corresponding methods to refresh/retrieve the neighbors' status.

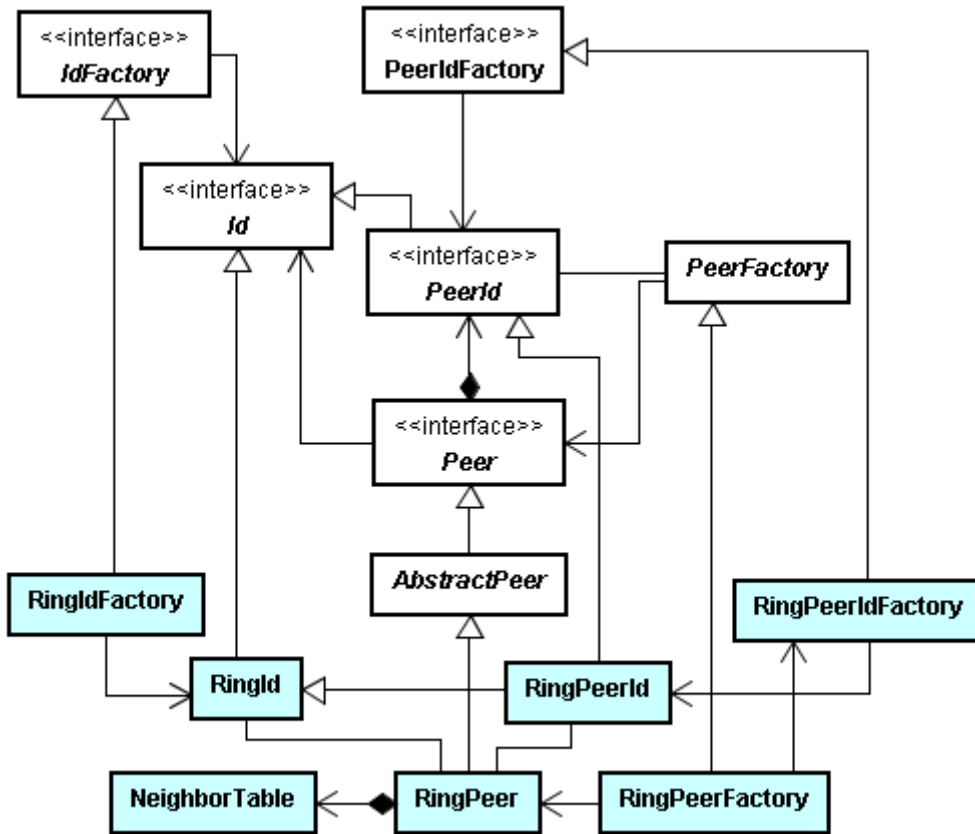


Figure 5-1 – Relationships between Ring Protocol implementation and P2P Protocol Layer

In order to implement the join operation, we directly send out a join request to boot peer that search for the successor. When the lookup response is arrived, the joining peer can setup successor and predecessor. The random neighbor table can also be filled up according to the response message. While peer leaving, the predecessor is notified with successor correction information. The predecessor of leaving peer can then inform its new successor for further topology repairing.

The lookup operation is implemented in recursive way. The lookup request is forward to the closest neighbor until the successor of current peer is the possible handler for the certain id. The entire message flow involves lookup, route, localLookup, and handleMessage method. The localLookup method determines the closest neighbor with the information about current neighborhoods. The route method compacts lookup request and routing information into a

RouteMessage, delegate the message transmission to communication manager. In handleMessage method, the lookup request is examined if the request is reach the correct destination. A lookup response will be sent to the requester, or a fail occurs while the request is time out.

The stabilization process is established by using `scheduleTask` methods. A periodical message sending task is registered that send notification to its successor (if existing), and the successor reports its predecessor as response. Any inconsistency of ring topology will be correct during the request-response cycle. The neighbor table updating task also utilizes message scheduling to ping each peer, the table entry is removed if exception occurred in message transmission.

The second protocol we used is the Viceroy DHT (details present in Appendix B) with topology stabilization enhanced. In the design of Viceroy DHT, both peers and resources are mapping to the interval of real numbers between the interval of $[0, 1)$. The `ViceroyId` class represents a valid identifier and defines the distance function. A `ViceroyPeer` maintains additional peer status, such as level, seven out-bound links and all remote links. By defining the `ViceroyIdFactory` and `ViceroyPeerFactory`, developers can adapt their application to Viceroy DHT.

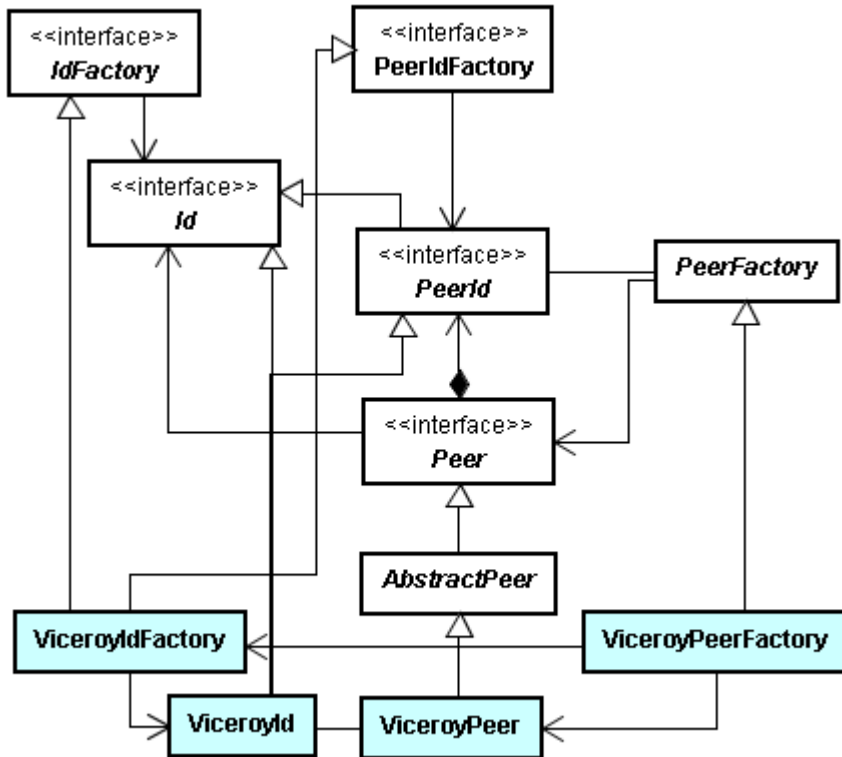


Figure 5-2 – Relationship between Viceroy DHT implementation and P2P Protocol Layer

Within the `handleMessage` method, each arriving message is delegated to individual processing functions. The join operation sends a `JoinMessage` out to find the correct joining position on the P2P network through the boot peer. Once the join operation fails to complete within the timeout, peer will start a new network and join as the first peer in this network. Peers send a `LeaveMessage` to its successor to notify the change of topology.

The `route` method implements the greedy routing algorithm via the `localLookup` method determining the next hop. The lookup operation involves two messages, `LookupMessage` and `LookupAckMessage`, to discover the handle peer and notify the result. By using the lookup operation, each Viceroy peer can then forward the `LevelLookupMessage` to complete the level lookup operation. If these two lookup operation do not finish before timeout, the lookup method will be interrupted and will throw an exception.

By using `sheduleTask` method, the periodical stabilization and probing tasks are implemented. If the peer status is incorrect, a series of actions for topology reconstruction will be triggered. For deterring the correctness of inbound connections, a random remote peer will be chosen and a `InboundValidateMessage` will be sent to see if the remote peer still holds the link.

5.1.2. Implementing Pub/Sub Service

The P2P pub/sub service providers need to implement the `PubSubService` interface to deploy their pub/sub algorithm in P2P network using our framework. Two desgin patterns of developing a P2P pub/sub system are identified. The detail of implementing P2P pub/sub services is lists the following table:

	<i>Store-Sub</i>	<i>Store-Pub</i>
addPublisher	N/A	Maintain Publisher
removePublisher	N/A	Structure
requestPublish	Event Dissemination	
addSubscriber	Maintain Subscriber	N/A
removeSubscriber	Structure	N/A

Table 5-2 – Task Descriptions of Developing Pub/Sub Services

A simple topic-based pub/sub algorithm is used to demonstrate the flexible design of the pluggable pub/sub service. The simple pub/sub protocol maps each topic to a hash key. The *topic handler* is dynamically assigned to the peer that is responsible for the hash key. Topic handler receives the event from publisher and notifies every subscriber currently interested in. Appendix C shows the protocol in details.

We create a class named *SimplePubSubService* that implement this simple pub/sub protocol. Since only subscribers are need to stored, *SimplePubSubService* simply ignores the action of publisher joining/leave by leaving both `addPublisher` and `removePublisher` method empty. Subscribers are stored in local service instance and a subscription message is sent to topic handler while a new subscriber initiated. The handling peer stores the subscriptions with corresponding subscribing peer and interested topic for later notification process.

When publishers send out events, the *SimplePubSubService* send out an event-publishing message to the topic handler. Once the topic handler received the publish request, it lookups all subscribers that interested in the same topic and send out the event-notification message to each one of them. Each event notification will be delivered to subscribers and corresponding event handler will then be triggered.

Our framework not only supports topic-based model, but also accommodates content-based model. There are two approaches to implementing content-based pub/sub algorithms. First, by introducing a wildcard topic, publishers and subscribers discard the topic information. The subscription is described only using selector string, which represents the user's interests. Second, the topic can be treated as a special attribute. Events which published by the publisher associated with specific topic are all associate with the same attribute value. By using these two approaches, applications can access content-based pub/sub system via our Pub/Sub API.

5.1.3. Develop P2P Pub/Sub Application

We introduce how to create an application as a client accessing P2P pub/sub service via our framework. There are five steps to initialize the P2P network environment: (1) load configurations, (2) setup network environment, (3) setup P2P protocol, (4) setup services, (5) join to P2P network. Tasks in each step are described in Figure 5-3.

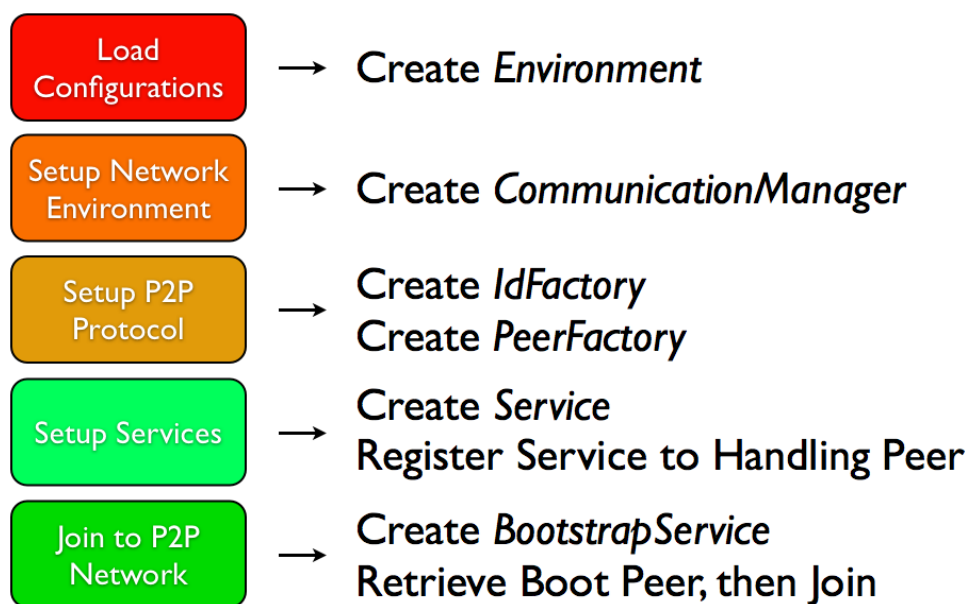


Figure 5-3 – Setup P2P and network environment.

In order to participate in an existed network, we need to prepare the execution environment and peer initializer first. The following code snippet demonstrates how to setup Ring Protocol on TCP/IP network environment:

```
//determine underlying transportation mechanism
Environment env = new Environment(new File("example.cfg"));
CommunicationManager layer = new TCPCommunicationManager(env);

//determine p2p network type and id type
IdFactory idFactory = new RingIdFactory(env);
PeerFactory factory = new RingPeerFactory(idFactory, layer);

//peer initialization
```



```
Peer peer = factory.createPeer();
PubSubService service = new SimplePubSubService();
peer.register(service);
```

The `Environment` object can create or load external parameters from properties file. Here we use TCP connection and Ring Protocol as our underlying network transmission and P2P network topology. By using `RingIdFactory` and `RingPeerFactory`, we are able to create a new peer. The pub/sub service plug-in is also register to the new created peer at this step. After peer successfully initialized, we use `BootstrapService` to connect to arbitrary boot server and retrieve a valid boot peer:

```
//connect to http boot server
BootstrapService bootService = new HttpBootstrapService(env);

try { //perform join operation
    NodeHandle localhandle = peer.getLocalHandle();
    NodeHandle booter = bootService.getBootstrapper(localhandle);
    peer.join(booter);
} catch (PeerJoinException ex) { //if join failed on exception
    ex.printStackTrace();
}

if (peer.isJoined()) {
    //following p2p operating goes here
}
```

We use HTTP protocol connect to boot server. The server URL is defined in external properties loaded by `Environment`. The peer we created uses the value returned from `BootstrapService.getBootstrapper()` to perform join operation.

If the peer successfully joins the network and registers pub/sub service, we can use Pub/Sub API to create a publishing session. Figure 5-4 lists the corresponding actions for publishing events.

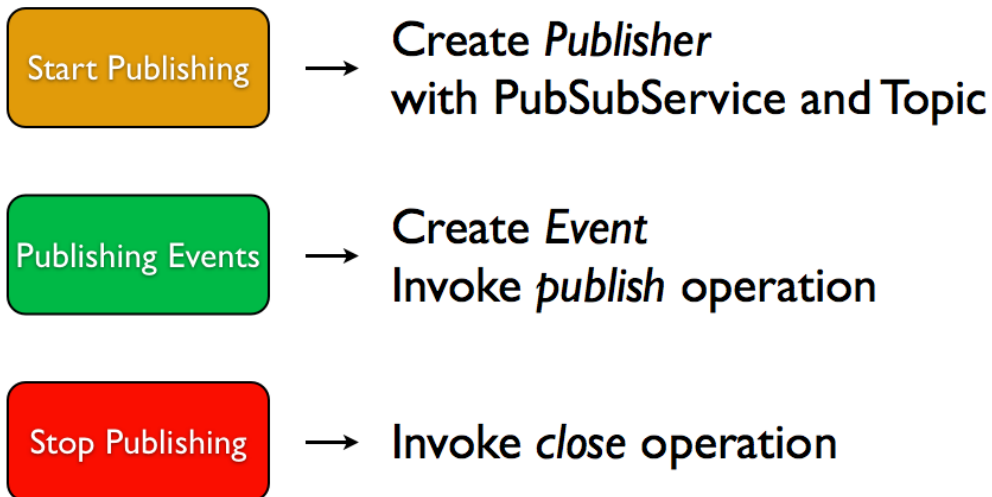


Figure 5-4 – Using Pub/Sub API for Publishing.

The following code demonstrate how to perform pub/sub tasks:

```
//create publisher associated with specific topic and peer
Topic topic = new Topic(idFactory, "hello topic");
Publisher publisher = null;
try {
    publisher = new Publisher(service, topic);
    // publish message
    Event event = new TextEvent(peer.getPeerId(), "hello pubsub");
    publisher.publish(event);
} catch (PubSubException ex) { //publish failed
    ex.printStackTrace();
    System.exit(1);
} finally { //clean up publisher and peer
    try {
        if (publisher != null) { publisher.close(); }
    } catch (PubSubException ex) {
        ex.printStackTrace();
    }
    try {
        peer.leave();
    } catch (PeerLeaveException ex) {
        //peer may not leave network correctly,
        //stabilization will handle the error
        ex.printStackTrace();
    }
}
}
```

We create a topic named *hello topic* and associate *Publisher* with that topic and the pub/sub service previously created. Then, events can be published via the publisher.

Like performing publishing, applications create corresponding subscription to receive interested information. Figure 5-5 describes how to add/remove subscriptions.

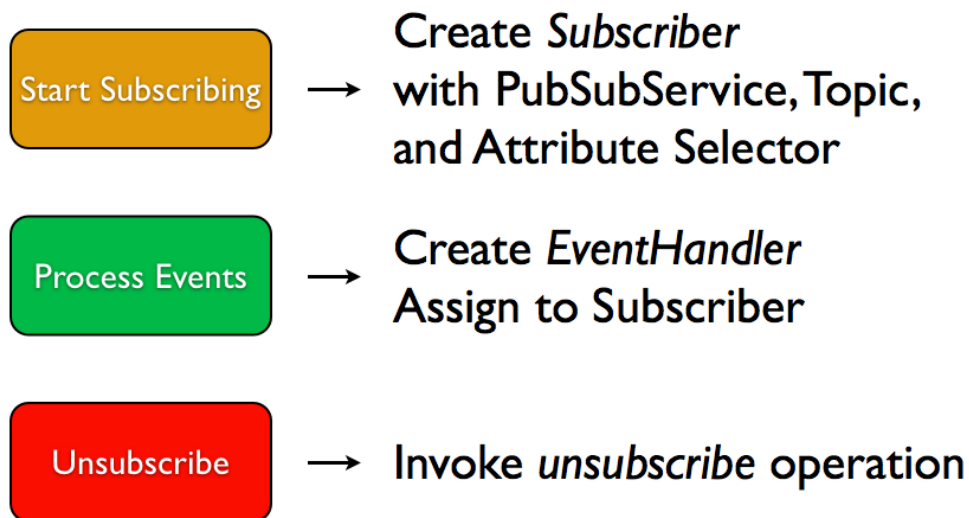


Figure 5-5 – Using Pub/Sub API for Subscribing.

The following code is used for creating subscription on a specific topic:

```
//create subscriber associated with specific topic and peer
Topic topic = new Topic(idFactory, "hello topic");
Subscriber subscriber = null;
try {
    subscriber = new Subscriber(service, topic);
    //set message listener to catch message event
    subscriber.setEventHandler(new EventHandler() {
        public void onEvent(Event event) {
            if (event instanceof TextEvent) {
                System.out.println(((TextEvent) event).getText());
            }
        }
    });
    try { //wait until user input 'q'
        for (int c = 0; c != 'q'; c = System.in.read()){
        } catch (IOException ex) {}
    }
}
```

```

} catch (PubSubException ex) { //subscribe fail
    ex.printStackTrace()
} finally { //clean up subscriber and peer
    try {
        if (subscriber != null) { subscriber.unsubscribe();}
    } catch (PubSubException ex) {}
    try { peer.leave(); } catch (PeerLeaveException ex) {}
}

```

The Subscriber also associates with given topic and pub/sub service. In addition, an EventHandler is specified to receive the event notification. The onEvent() method will be invoked while an event arrived. We simply print all the content of arriving text event.

Both physical network and P2P routing protocol components are exchangeable. With a small amount of modification, the applications can deploy on different operation environments. Currently, our framework supports two network environments and two P2P networks. Figure 5-6 shows these environment options and the corresponding steps during initialization.



Figure 5-6 – P2P and Network Environment Options.

The following code snippet demonstrates the adaptation by changing P2P protocol to Viceroy DHT and limits the communication within local computer:

```
        :
//deploy p2p network on local machine
CommunicationManager layer = new LocalCommunicationManager(env);
//using viceroy id and peer factory
IdFactory idFactory = new ViceroyIdFactory(env);
PeerFactory factory = new ViceroyPeerFactory(idFactory, layer);
        :
```

Only three object instances are changed and leaving other application code unmodified. Through these examples above, we can see how agile this framework is to develop pub/sub application on different environment.

5.2. Comparisons

The framework we proposed is based upon object-oriented architecture and event-driven methodology. According to the structured P2P specification defined in [5][7], we enhance the functionalities into object models that fully describe the relationships between the identifier space and the routing protocol. Moreover, the framework proposed by Aberer et al. [1], the additional service modules, e.g., P2P Storage Interface, and the P2P Basic Interface, i.e., P2P protocol, are objects directly inherited from the same parent class. However, in our architecture, we introduce pluggable modules, e.g., the Pub/Sub Service, those decouple from the P2P protocol implementation. The features are achieved by invoking services as events arrived. The events contain communication messages as well as topology modifications. The event of state transition of handling peer, i.e., peer joins to a network and peer is ready to receive message, is not propagated to the services. Developers can only perform stabilization and replication in proactive style while generating persistence services. Nevertheless, this pluggable approach makes a lightweight peer implementation. Therefore, the P2P Protocol Layer only needs to

handle routing protocols. The additional pluggable services are independent modules not included in the layer.

Our design has been focusing on *pure* P2P networks. In other words, each peer in the architecture shares information and collaborates with other peers without a centralized server. In previous researches [23][35], publishers and subscribers are both clients of a message server. In our platform, each peer involves message dispersing and propagating via the pub/sub mechanism without an additional message server. Instead, each peer is involved in the information dispersal of the pub/sub mechanism in our framework, without establishing additional message server. The benefit of pure P2P is that applications do not depend on a pre-constructed server infrastructure. The index information is connoted in the P2P network topology and routing protocol, compared to the super-peer indexing mechanism used in JXTA. However, this statement assumes the computation power of each peer is about equal. According to the assumption, this framework does not grant developer the advantage deploying P2P applications on the environment of heterogeneous devices.

In previous research of P2P protocol, network bootstrapping is usually omitted. By considering the practicality of creating P2P applications, we define the bootstrap service interface and provide two boot server implementations. By externalizing the network communication, the framework allows different protocols transmitting messages through one single network port. With the evolving of `CommunicationManager`, the performance of all P2P protocols can be boosted.

The FreePastry library is an open source implementation of Pastry. With Scribe system implemented as an additional service, developers can create group communication system with efficient pub/sub capability. In the design of FreePastry, the factory methods are used for testing/simulating applications without

modification to the source code. However, applications developed using FreePastry are limited to the functionalities that this P2P library offered, i.e., only Pastry network and Scribe system. Our framework provides a flexible architecture that application can easily deploy to any P2P network and any network environment. With the lightweight pub/sub APIs, application developers can adopt any P2P pub/sub service to meet their system requirements.

JXTA is a general P2P platform that allows heterogeneous applications be deployed on top of a virtual JXTA network. JXTA can provide additional structured P2P network functionality based on *Peer Resolver Protocol*. An open source project named *Meteor* [17] implements Chord and CAN on top of the JXTA platform. This approach deploys the DHT overlays upon the virtual JXTA network, which causes the performance downgrade because of the communication overheads among peers that introduced by JXTA. The JXTA platform provides a propagating pipe which can simulate pub/sub mechanism via the one-to-many message transmission. The message might be lost without noticed during the process of propagation. The performance degrading and reliability issue make this propagation mechanism not scaled to a large group communication system. In our framework, the message transmission among peers is directly delegated to physical network transportation, which does not incur the overheads of additional node discovery. Without message propagating, our pub/sub service can maintain a distributed multicast structure and support many-to-many message transmission. Therefore, disseminating information among peers will not cause unnecessary bandwidth dissipation.

6. Conclusion and Future Work

6.1. Conclusion

In this thesis, we identify major issues from three aspects of developing P2P pub/sub applications. These issues result from the lack of standardize P2P API, common P2P pub/sub API, and network abstraction. Therefore, we synthesis standardized P2P API, common pub/sub API a generic P2P pub/sub framework. Our framework provides a standard P2P API for application develops to interact with various structured P2P networks. Furthermore, a P2P pub/sub API and SPI are introduced for using/creating P2P pub/sub algorithm in pure P2P networks. In our design, a layered architecture is created with common P2P API, common pub/sub API/SPI, network transportation, and bootstrapping service. This framework allows P2P application developer to switch the underlying overlay with a little bit code to modify. We standardize the control flow between each module. The following benefits are brought out by this framework:

- A. *Easily develop/deploy application on different P2P networks and different pub/sub systems.*
- B. *Support both topic-based and content-based pub/sub models.*
- C. *Deploy the P2P applications on various network environments.*

This framework is designed for developing P2P pub/sub applications in pure P2P network. It provides an adaptive architecture for developing applications on any overlays without incurring performance degradation. On the other hand, this framework does not accommodate to an overlay network containing more than one role of peers, e.g., the super-peers architecture used in JXTA. By comparing to

other P2P pub/sub library and P2P platform, this framework provides generality of adapting to any P2P routing protocol and P2P pub/sub algorithm and preserves the performance and reliability of P2P networks.

In conclusion, with the realization of common API, this framework not only standardizes the semantic of using structured P2P network, but also creates a general control flow of develop a P2P pub/sub application. By adopting our framework, developers can generate full-fledged pub/sub applications on top of every structured P2P networks.

6.2. Future Work

For further extension, the transport layer can provide predefined retransmission policy for P2P network developer to implement routing protocol in a robust way. The transmission policy can be implemented in two ways, used as a parameter of send operation or declared as class scope/method scope annotation. The function parameter solution provides a fine-grain control on every single send operation.

function parameter	annotation
<pre>RouteMessage rm = ... SendPolicy policy = SendPolicy.TryTwice; commMgr.send(rm, policy);</pre>	<pre>@Policy(retry=2) public class RingPeer extends AbstractPeer { ... }</pre>

Table 6-1 – Possible Usage of Send Policy

The policy annotation approach provides a coarse-grain control that defines the send behavior over entire class or method. Using annotation keeps protocol implementations away from physical transport problems. However, this approach requires additional code preprocessing. Table 6- lists the possible usage of these two approaches. In addition, providing response-waiting utilities helps P2P protocol

developers implementing request-response operations, e.g. lookup operation, without establishing their own lock-notify mechanism.

We omit the security issue from the framework. In order to build a secure P2P application, the ultimate solution is applying a secure mechanism, e.g. Byzantine fault tolerance [30], to P2P protocols. We leave the implementation to P2P protocol developer. However, this framework can provide connection security and data encryption. Implementing a `TLPCommunicationManager` offers a secure connection or constructing a `MessageEncryptor` provides data integrity.

Although this framework is design in the way of adapting P2P application to different structured P2P network, fully implementing these P2P network components requires excessive works. A protocol adaptor can be introduced to reduce the overhead of implementing P2P components using legacy libraries.

This framework can be further extended into a P2P service middleware, integrated with OSGi platform [21]. Our framework can be employed as the communication infrastructure for other OSGi services. With runtime deployment and activation, applications can easily deploy on an existing P2P topology. Under this service-oriented architecture, P2P components are not only reused in development process, but also in runtime. Moreover, the monitoring services can be dynamically introduced based on the architecture of OSGi platform.

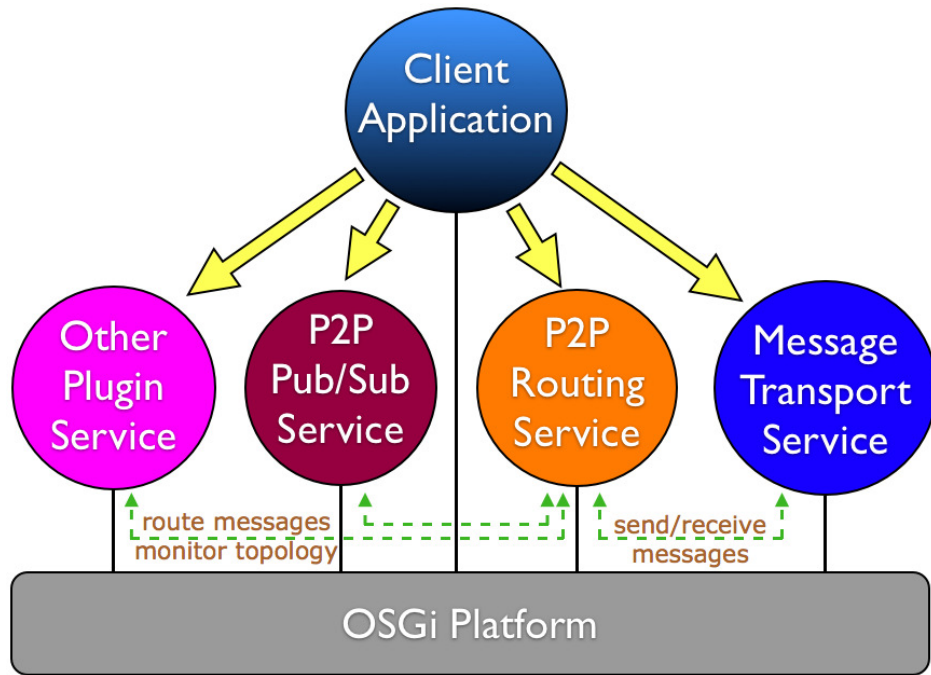


Figure 6-1 – Integrated with OSGi platform.

Currently, our framework supports only message-based communication. With application level socket, application can manage interaction between peers with stream-based communication. The socket API should have ability for P2P application and service to establish long-live connections between peers. This long-live connection reduces the effort on waiting message acknowledgement in a frequent interaction scenario.

A real-world P2P pub/sub application should be implemented on top of our framework for further examining the usability. A P2P Blog system, based on Scribe system and Pastry network, has been implemented in our laboratory and has been proved as an efficient approach to disseminate articles to massive readers. By adapting this blog system to our framework, a complementary performance evaluation on different overlay environment will be taken and we will have opportunity to optimize the performance for both full-featured PCs and constrained mobile devices.

7. Appendix

A. The Ring Protocol

The ring protocol is a simple routing protocol built on top of the ring topology and is a simplified version of Chord protocol. The identifier is a 128 or 160 bits integer and the distance metrics between peers is determined by the clockwise distance. Each peer connects to the closest id in both clockwise and counter clockwise directions. For improving the resilience on network partition, each peer links to K random peer. The detail of this protocol is shown in Table 7-.

<pre>// node n joins the network // n' is an arbitrary node in the network n.join(n') if(n' is not null) successor = n'.lookup(n.id); predecessor = successor.predecessor; for i = 1 to K neighbor[i] = successor.neighbor[i]; else successor = n; predecessor = n; // lookup the responsible node for given id n.lookup(id) x = n; while(id ∉ (x.id, x.successor.id)) x = x.closest_neighbor(id); return x.successor;</pre>	<pre>// return closest neighbor to given id n.closest_neighbor(id) min = successor.dist(id); x = successor; if(predecessor.dist(id) < min) min = predecessor.dist(id) x = predecessor; for i = 1 to K if(neighbor[i] is not null) if(neighbor[i].dist(id) < min) min = neighbor[i].dist(id); x = neighbor[i]; return x;</pre>
---	---

Table 7-1 – Pseudo code for the node join and lookup operation

The ring peer periodically checks the existence of all the neighbors. By proactive stabilization, peers are guaranteed to have a correct successor at some

time after the last join/leave operation occurred. The random neighbors are also updated every time a message delivered and the oldest random peer is been replaced by the new coming request peer. Table 7-2 shows the pseudo code of the stabilization operation.

```

// periodically check the consistency of successor
n.stabilize()
    x = successor.predecessor;
    if(x.id ∈ (n.id, successor.id))
        successor = x;
    successor.notify(n);

// n' might be the predecessor of n
n.notify(n')
    if(predecessor is null or n' ∈ (predecessor.id, n.id))
        predecessor = n';

// periodically check the neighbor table entries
n.check_neighbors()
    i = random number in (1, K);
    if(neighbor[i] is not null and neighbor[i] is not alive)
        for j = i to K - 1
            neighbor[j] = neighbor[j + 1];
        neighbor[K] = null;

// update neighbor table with latest visitor
n.update_neighbors(n')
    if(n' already exists in neighbor table)
        move n' to neighbor[K]
    else
        for i = 1 to K - 1
            neighbor[i] = neighbor[i + 1]
        neighbor[K] = n'

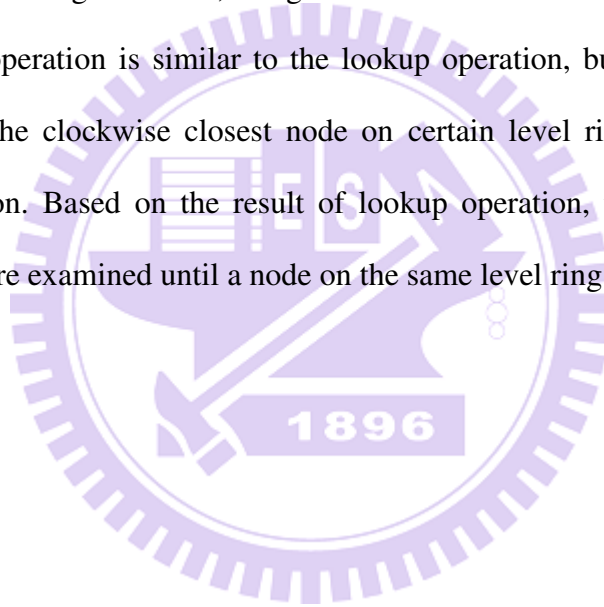
```

Table 7-2 – Pseudo code of stabilization and neighbor update operation

B.Enhanced Viceroy Protocol

The design of Viceroy Protocol is based on both ring topology and butterfly topology. In the previous research, the issues of concurrent join/leave and unexpected peer failure are omitted. We slightly modify the algorithm to improve the reliability.

We use the greedy *FindFast* algorithm (mentioned in the technical report) as our default lookup algorithm. The FindFast algorithm simply forwards the lookup request to the closest neighbor node, using both inbound and outbound connections. The **nextonlevel** operation is similar to the lookup operation, but with additional *level* parameter. The clockwise closest node on certain level ring will be found using this operation. Based on the result of lookup operation, the successors of responsible node are examined until a node on the same level ring is found.



```

// node n joins the network
// n' is an arbitrary node in the network
n.join(n')
  if(n' is not null)
    successor = n'.lookup(n.id);
    predecessor = successor.predecessor;
    n.update_level();
    ring_successor = n.nextonlevel(n.id, level);
    ring_predecessor = ring_successor.ring_predecessor;
    n.update_butterfly();
  else
    successor = n;
    predecessor = n;
    level = 1;
    ring_successor = n;
    ring_predecessor = n;
    up = null;
    left = null;
    right = null;

// lookup the responsible node for given id
n.lookup(id)
  x = n;
  while(id ∉ (x.id, x.successor.id))
    x = x.closest_neighbor(id);
  return x.successor;

// select level via the estimation of total node numbers
n.select_level()
   $\tilde{n} = \lceil 1/n.dist(successor) \rceil$ ;
  level = random number in (1,  $\lfloor \log(\tilde{n}) \rfloor$ );

// lookup closest successor for given id on level i
n.nextonlevel(id, i)
  x = n.lookup(id);
  while(x.level ≠ i)
    x = x.successor;
  return x;

```

```

// init or update the butterfly links
n.update_butterfly()
  if(level > 1)
    up = n.nextonlevel(n.id, level - 1);
    up.notify_inbound(n);
    left = n.nextonlevel(n.id, level + 1);
    left.notify_inbound(n);
    right = n.nextonlevel(n.id + 2-level, level + 1);
    right.notify_inbound(n);

// return closest neighbor to given id
n.closest_neighbor(id)
  min = successor.dist(id);
  x = successor;
  if(predecessor.dist(id) < min)
    min = predecessor.dist(id);
    x = predecessor;
  if(ring_successor.dist(id) < min)
    min = ring_successor.dist(id);
    x = ring_successor;
  if(ring_predecessor.dist(id) < min)
    min = ring_predecessor.dist(id);
    x = ring_predecessor;
  if(up.dist(id) < min)
    min = up.dist(id);
    x = up;
  if(left.dist(id) < min)
    min = left.dist(id);
    x = left;
  if(right.dist(id) < min)
    min = right.dist(id);
    x = right;
  for each x' in inbounds
    if(x'.dist(id) < min)
      min = x'.dist(id);
      x = x';
  return x;

```


The additional stabilization protocol is used for maintain both ring and level ring structure. By periodically checking its successor, peers has chance to modify its level duo to the total number of peers might changed significantly. Because the level of each peer is determined by the distance of its successor, the ring stabilization needs to reselect level while the successor is changed. The level ring stabilization checks if the level ring is changed or if the peer needs to join another level ring. The butterfly links will be fixed while the peer detects itself join a different level ring. Without actively inform the change of butterfly links, the periodically inbound checking process will remove the inbound link if peer is left or no longer establish a butterfly link to it.

```

// periodically check the consistency of successor
n.stabilize()
    x = successor.predecessor;
    if(x.id ∈ (n.id, successor.id))
        successor = x;
        n.select_level();
        successor.notify(n);

// periodically check the consistency of ring _ successor
n.level_stabilize()
    x = ring_successor.ring_predecessor;
    if(level ≠ x.level)
        ring_successor = n.nextonlevel(n.id, level);
        n.update_butterfly();
    else if(x.id ∈ (n.id, ring_successor.id))
        ring_successor = x;
        ring_successor.level_notify(n);

// n' might be the predecessor of n
n.notify(n')
    if(predecessor is null or n' ∈ (predecessor.id, n.id))
        predecessor = n';

```

```

// n' might be the ring _ predecessor of n
n.level_notify(n')
    if(ring _ predecessor is null or n' ∈ (ring _ predecessor.id, n.id))
        ring _ predecessor = n';

// periodically check the inbound connections
n.check_inbounds()
    x = random node in inbounds;
    if(x.validate_inbound(n) is not true)
        inbounds.remove(x);

// validate if n still hold a butterfly link to n'
n.validate_inbound(n')
    return (up ≡ n' or left ≡ n' or right ≡ n');

// periodically check the butterfly links
n.check_butterfly()
    if(up is null or up is not alive)
        if (level > 1)
            up = n.nextonlevel(id, level - 1);
            up.notify_inbound(n);
    if(left is null or left is not alive)
        left = n.nextonlevel(id, level + 1);
        left.notify_inbound(n);
    if(right is null or right is not alive)
        right = n.nextonlevel(id + 2-level, level + 1);
        right.notify_inbound(n);

// notify node n that a butterfly link
// established from node n'
n.notify_inbound(n')
    if(n' is not in inbounds)
        inbound.add(n')

```

C.The Simple Pub/Sub Protocol

Simple pub/sub protocol is based on topic-based model. Each topic is mapping to a hash id. The peer that handles the given key is called topic handler. Subscriber sends topic subscription message topic handler for registering interest on specified topic channel. Publisher sends the event publish message to the topic handler. Once an event publish message were received, topic handler relay the event notification to all subscribers that register to the certain topic. When an event notification arrives, the event handlers to corresponding topic are awake and the onEvent method is invoked. The concept of this algorithm is illustrated in Figure 7-1.

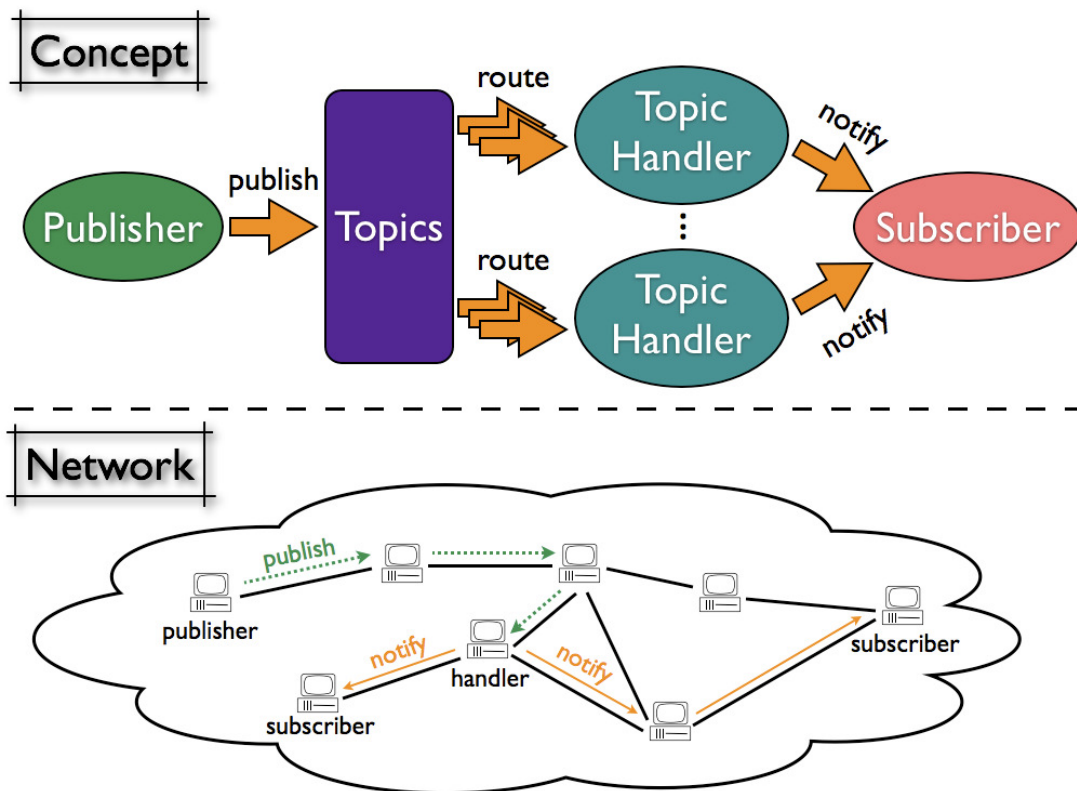


Figure 7-1 – Concept of Simple Pub/Sub Algorithm

Whenever a peer joined and the update operation is invoked, topics that should handled by the newly joined peer will migrate to the certain peer. On

receiving migration message, the pub/sub service that registered on that peer stores all the migrated topics.

For data persistence, a stabilizing message is sent periodically for dispersing topics that peer handles to all peers in replication set. Once the peer is left, another peer will take over these topics.

This pub/sub algorithm does not support attribute filtering. The selector expression of subscriber is simply ignored. Event handler is notified whatever attributes arriving event contains.

D.Example Program

I. Publish Client

```
package pubsub.app;

import dcslab.p2p.IdFactory;
import dcslab.p2p.Peer;
import dcslab.p2p.PeerFactory;
import dcslab.p2p.PeerJoinException;
import dcslab.p2p.PeerLeaveException;
import dcslab.p2p.boot.BootstrapService;
import dcslab.p2p.boot.impl.LocalBootstrapService;
import dcslab.p2p.environment.Environment;
import dcslab.p2p.pubsub.PubSubService;
import dcslab.p2p.pubsub.PubSubException;
import dcslab.p2p.pubsub.Publisher;
import dcslab.p2p.pubsub.Topic;
import dcslab.p2p.pubsub.event.Event;
import dcslab.p2p.pubsub.event.TextEvent;
import dcslab.p2p.pubsub.impl.SimplePubSubService;
import dcslab.p2p.transport.CommunicationManager;
import dcslab.p2p.transport.impl.TCPCCommunicationManager;
import dcslab.p2p.impl.RingIdFactory;
```

```

import dcslab.p2p.impl.RingPeerFactory;
import java.io.File;

public class PublishClient {
    public static void main(String[] args) {
        //determine underlying transportation mechanism
        Environment env = new Environment(new File("example.cfg"));
        buildTCPEnvironment(env);
        CommunicationManager layer = new TCPCommunicationManager(env);

        //determine p2p network type and id type
        IdFactory idFactory = new RingIdFactory(env);
        PeerFactory factory = new RingPeerFactory(idFactory, layer);

        //peer initialization
        Peer peer = factory.createPeer();
        PubSubService service = new SimplePubSubService();
        peer.register(service);

        //connect to http boot server
        BootstrapService bootService = new LocalBootstrapService();

        try { //perform join operation
            NodeHandle localhandle = peer.getLocalHandle();
            NodeHandle booter = bootService.getBootstrapper(localhandle);
            peer.join(booter);
        } catch (PeerJoinException ex) { //if join failed on exception
            ex.printStackTrace();
        }

        if (peer.isJoined()) {
            //create publisher associated with specific topic and peer
            Topic topic = new Topic(idFactory, "hello topic");
            Publisher publisher = null;
            try {
                publisher = new Publisher(service, topic);
                // publish message
                Event event =

```

```

        new TextEvent(peer.getPeerId(), "hello pubsub");
        publisher.publish(event);
    } catch (PubSubException ex) { //publish failed
        ex.printStackTrace();
        System.exit(1);
    } finally {
        try {
            if (publisher != null) { publisher.close(); }
        } catch (PubSubException ex) {
            ex.printStackTrace();
        }
        try {
            peer.leave();
        } catch (PeerLeaveException ex) {
            //peer may not leave network correctly,
            //stabilization will handle the error
            ex.printStackTrace();
        }
    }
} else {
    // join failed, exit program
    System.exit(1);
}
}
}
}

```

II. Subscribe Client

```

package pubsub.app;

import dcslab.p2p.IdFactory;
import dcslab.p2p.Peer;
import dcslab.p2p.PeerFactory;
import dcslab.p2p.PeerJoinException;
import dcslab.p2p.PeerLeaveException;
import dcslab.p2p.boot.BootstrapService;
import dcslab.p2p.boot.impl.LocalBootstrapService;

```

```

import dcslab.p2p.environment.Environment;
import dcslab.p2p.pubsub.PubSubException;
import dcslab.p2p.pubsub.PubSubService;
import dcslab.p2p.pubsub.Subscriber;
import dcslab.p2p.pubsub.Topic;
import dcslab.p2p.pubsub.event.Event;
import dcslab.p2p.pubsub.event.EventHandler;
import dcslab.p2p.pubsub.event.TextEvent;
import dcslab.p2p.pubsub.impl.SimplePubSubService;
import dcslab.p2p.transport.CommunicationManager;
import dcslab.p2p.transport.impl.TCPCommunicationManager;
import dcslab.p2p.impl.RingIdFactory;
import dcslab.p2p.impl.RingPeerFactory;
import java.io.File;
import java.io.IOException;

public class SubscribeClient {
    public static void main(String[] args) {
        //determine underlying transportation mechanism
        Environment env = new Environment(new File("example.cfg"));
        CommunicationManager mgr = new TCPCommunicationManager(env);

        //determine p2p network type and id type
        IdFactory idFactory = new ViceroyIdFactory(env);
        PeerFactory factory = new ViceroyPeerFactory(idFactory, mgr);

        //peer initialization
        Peer peer = factory.createPeer();
        PubSubService service = new SimplePubSubService();
        peer.register(service);

        //connect to http boot server
        BootstrapService bootService = new HttpBootstrapService(env);

        try { //perform join operation
            NodeHandle localhandle = peer.getLocalHandle();
            NodeHandle booter=bootService.getBootstrapper(localhandle);
            peer.join(booter);
        }
    }
}

```

```

} catch (PeerJoinException ex) { //if join failed on exception
    ex.printStackTrace();
}
if (peer.isJoined()) {
    //create subscriber associated with specific topic and peer
    Topic topic = new Topic(idFactory, "hello topic");
    Subscriber subscriber = null;
    try {
        subscriber = new Subscriber(service, topic);
        //set message listener to catch message event
        subscriber.setEventHandler(new EventHandler() {
            public void onEvent(Topic topic, Event event) {
                if (event instanceof TextEvent) {
                    System.out.println(
                        topic.getTopicName() +
                        ((TextEvent) event).getText());
                }
            }
        });
        try { //wait until user input 'q'
            for (int c = 0; c != 'q'; c = System.in.read()){}
        } catch (IOException ex) {}
    } catch (PubSubException ex) { //subscribe fail
        ex.printStackTrace();
    } finally { //clean up subscriber and peer
        try {
            if (subscriber != null) { subscriber.unsubscribe();}
        } catch (PubSubException ex) {}
        try { peer.leave(); } catch (PeerLeaveException ex) {}
    }
} else {
    // join failed, exit program
    System.exit(1);
}
}
}

```


8. References

- [1] K. Aberer, L. O. Alima, A. Ghodsi, S. Girdzijauskas, S. Haridi and M. Hauswirth, "The Essence of P2P: A Reference Architecture for Overlay Networks," *P2P*, vol. 0, pp. 11-20, 2005.
- [2] J. Aspnes and G. Shah, "Skip graphs," *ACM Trans. Algorithms*, vol. 3, pp. 37, 2007.
- [3] M. Bender, S. Michel, S. Parkitny and G. Weikum, "A Comparative Study of Pub/Sub Methods in Structured P2P Networks," *Databases, Information Systems, and Peer-to-Peer Computing*, pp. 385-396, 2007.
- [4] M. Castro, P. Druschel, A. -M. Kermarrec and A. I. T. Rowstron, "Scribe: a large-scale and decentralized application-level multicast infrastructure," *Selected Areas in Communications, IEEE Journal on*, vol. 20, pp. 1489-1499, 2002.
- [5] G. Ciaccio, "A Pretty Flexible API for Generic Peer-to-Peer Programming," *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1-8, 26-30 March 2007.
- [6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris and I. Stoica, "Wide-area cooperative storage with CFS," *SIGOPS Oper. Syst. Rev.*, vol. 35, pp. 202-215, 2001.
- [7] F. Dabek, B. Zhao, P. Druschel, J. Kubiawicz and I. Stoica, "Towards a Common API for Structured Peer-to-Peer Overlays," *Peer-to-Peer Systems II*, pp. 33-44, 2003.
- [8] F. Delmastro, M. Conti and E. Gregori, "P2P common API for structured overlay networks: A cross-layer extension," in *WOWMOM '06: Proceedings of the 2006 International Symposium on on World of Wireless, Mobile and*

- Multimedia Networks*, 2006, pp. 593-597.
- [9] P. Fraigniaud and P. Gauron, "D2B: A de Bruijn based content-addressable network," *Theoretical Computer Science*, vol. 355, pp. 65-79, 4/6. 2006.
- [10] D. Hausheer, "Decentralized auction-based pricing with PeerMart," *Integrated Network Management, 2005. IM 2005. 2005 9th IFIP/IEEE International Symposium on*, pp. 381-394, 2005.
- [11] M. O. Junginger, "A self-organizing publish/subscribe middleware for dynamic peer-to-peer networks," *Network, IEEE*, vol. 18, pp. 38-43, 2004.
- [12] M. Kaashoek and D. Karger, "Koorde: A Simple Degree-Optimal Distributed Hash Table," *Peer-to-Peer Systems II*, pp. 98-107, 2003.
- [13] Y. Kulbak and D. Bickson, "The emule protocol specification," 2005.
- [14] A. Loo, "The future of peer-to-peer computing," *Communications of the ACM*, vol. 46, issue 9, pp. 57, 2003.
- [15] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma and S. Lim, "A Survey and Comparison of Peer-to-Peer Overlay Network Schemes," *Communications Surveys & Tutorials, IEEE*, vol. 7, pp. 72-93, 2005.
- [16] D. Malkhi, M. Naor and D. Ratajczak, "Viceroy: a scalable and dynamic emulation of the butterfly," pp. 183-192, 2002.
- [17] P. Manish, J. Nanyan, S. Cristina and M. Vincent. (2007, Feb. 21). Meteor. 2.4.1 Available: <https://jxta-meteor.dev.java.net/>
- [18] J. Márk, M. Alberto, P. Gian Jesi and V. Spyros. (2007, Dec. 23). PeerSim: A peer-to-peer simulator. 1.0.3 Available: <http://peersim.sourceforge.net/>
- [19] P. Maymounkov and D. Mazieres, "Kademlia: A Peer-to-peer Information System Based on the XOR Metric," 2002.
- [20] MONKIA Info., "NUWeb," 2007

Available: <http://tw.nuweb.cc/>

- [21] OSGi Alliance, "OSGi Service Platform Core Specification Release 4.1," October. 2007.
- [22] D. Peter, E. Eric, G. Romer, H. Andreas, H. Jeff, C. Y. Hu, I. Sitaram, L. Andrew, M. Alan, N. Animesh, P. Ansley, R. Charlie, S. Dan, S. Jim, S. Atul and Z. RongMei. (2007, Nov. 2). FreePastry. 2.0_03
Available: <http://freepastry.rice.edu/FreePastry/>
- [23] P. Pietzuch, D. Eysers, S. Kounev and B. Shand, "Towards a common API for Publish/Subscribe," in *DEBS '07: Proceedings of the 2007 Inaugural International Conference on Distributed Event-Based Systems*, 2007, pp. 152-157.
- [24] C. G. Plaxton, R. Rajaraman and A. W. Richa, "Accessing Nearby Copies of Replicated Objects in a Distributed Environment," *Theory of Computing Systems*, vol. 32, pp. 241-280, 02/24. 1999.
- [25] J. Pouwelse, P. Garbacki, D. Epema and H. Sips, "The Bittorrent P2P File-Sharing System: Measurements and Analysis," *Peer-to-Peer Systems IV*, pp. 205-216, 2005.
- [26] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Commun ACM*, vol. 33, pp. 668-676, 1990.
- [27] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Schenker, "A scalable content-addressable network," in *SIGCOMM '01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2001, pp. 161-172.
- [28] J. Risson and T. Moors, "Survey of Research Towards Robust Peer-to-Peer Networks: Search Methods," *Computer Networks*, vol. 50, pp. 3485-3521, 12/5. 2006.

- [29] B. Roberto, Q. Leonardo and V. Antonino, "Distributed event routing in Publish/Subscribe communication systems: A survey," In: Technical Report TR-1/06, Dipartimento di Informatica e Sistemistica, Università di Roma 'La Sapienza' (2005).
- [30] R. Rodrigues, B. Liskov and L. Shriram, "The design of a robust peer-to-peer system," in *EW10: Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, 2002, pp. 117-124.
- [31] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, 2001, pp. 329-350.
- [32] I. Stoica, D. Adkins, S. Zhuang, S. Shenker and S. Surana, "Internet indirection infrastructure," in *SIGCOMM '02: Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2002, pp. 73-86.
- [33] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *SIGCOMM Comput. Commun. Rev.*, vol. 31, pp. 149-160, 2001.
- [34] Sun Microsystems Inc. (2007, Oct 16th). JXTA v2.0 protocols specification.
Available: <https://jxta-spec.dev.java.net/nonav/JXTAProtocols.html>
- [35] Sun Microsystems Inc. (2003, Dec 2nd). Java message service API.
Available: <http://www.jcp.org/en/jsr/detail?id=914>
- [36] B. Zhao, J. Kubiatowicz and A. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," Computer Science Division, U. C. Berkeley, apr, 2001.