SPECIFICATION AND RAPID DEPLOYMENT OF COMMUNICATION

PROTOCOLS FOR PEER-TO-PEER SYSTEMS

by

GOPINATHAN KANNAN

(Under the direction of Krzysztof J. Kochut)

ABSTRACT

Protocol Specification techniques have been used traditionally for verification of communication protocols. With the fast growing Internet and increasing uses of mobile devices, ubiquitous computing has come to the limelight of commercial and academic research. More and more technologies such as peer-to-peer systems are emerging that enable mobile devices to surf between networks with minimum effort and have access to a plethora of services. However, in order to use a particular service, the user needs to know the communication protocol(s), specifying the exchange of information needed to use the service successfully.

In this thesis, we address this problem in a peer-to-peer system (JXTA) by developing a protocol specification and implementation mechanism using Petri nets, and a protocol management infrastructure to upload the protocol dynamically and access the service associated with that protocol as and when requested by a peer.

INDEX WORDS: Protocol Specification, Protocol Verification, Communication Protocols, Ubiquitous computing, Peer-to-peer systems, Services, Petri Nets, Protocol Server, JXTA, PUMPS.

SPECIFICATION AND RAPID DEPLOYMENT OF COMMUNICATION PROTOCOLS FOR PEER-TO-PEER SYSTEMS

by

GOPINATHAN KANNAN

B.E., Anna University, India, 1999.

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2002

SPECIFICATION AND RAPID DEPLOYMENT OF COMMUNICATION PROTOCOLS FOR PEER-TO-PEER SYSTEMS

by

GOPINATHAN KANNAN

Approved:

Major Professor: Krzysztof J. Kochut

Committee: John A. Miller
Daniel M. Everett

Electronic Version Approved:

Gordhan L.Patel
Dean of the Graduate School
The University of Georgia
August 2002

DEDICATION

To my beloved parents and my wonderful wife for their love, support and encouragement.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Kochut for his continuous guidance and encouragement. I would also like to thank Dr. Miller and Dr. Everett for consenting to serve in my committee. Finally, I thank my wife, Gayathri, for her patience and support throughout this degree program.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS v

LIST OF FIGURES ix

CHAPTER

1 INTRODUCTION 1

1.1 Motivation 1

1.2 A Typical Example 2

1.3 Thesis Organization 3

2 BACKGROUND 4

2.1 Introduction to networking 4

2.2 Protocols and Network Layering 5

2.3 Application Layer Protocols 6

2.4 Peer-To-Peer (P2P) Computing 6

2.5 P2P Implications on Protocol Management 8

3 REVIEW OF PROTOCOL SPECFICATION METHODS 10

3.1 Introduction 10

3.2 Operating-System Based Methods 11

3.3 Programming Annotations Based Methods 11

3.4 Language Based Methods 12

3.5 Formal Specification Methods 13

3.6 Web Services Conversation Language (WSCL) 16

3.7 Active Networks 16
3.8 Petri Nets....17
4 PROTOCOL SPECFICATION USING COLORED PETRI NETS 19
4.1 Introduction....19
4.2 Behavioral Properties 22
4.3 Petri Net Variants....23
4.4 Colored Petri Nets (CPN) 24
4.6 Examples of Systems Implemented Using CPN....26
5 PEER-TO-PEER PROTOCOL MANAGEMENT ARCHITECTURE....31
5.1 General Overview 31
5.2 Design and Framework 32
5.3 Protocol Specification....33
5.4 Protocol Implementation Classes....38
5.5 JXTA: A P2P Computing Platform 39
5.6 The Protocol Management Service (PMS) 45
6 IMPLEMENTATION....47
6.1 Protocol Advertisement 47
6.2 Protocol Service Definition....49
6.3 Channel Request Messages 52
6.4 Channel Response Messages 53
7 EXAMPLES 55
7.1 LOGIN Protocol: A Simple Login Protocol....55
7.2 Internet Message Access Protocol (IMAPv4rev1: Abridged) 58

7.3 Distributed File Search Protocol (GNUTELLite) 68

8 CONCLUSION AND FUTURE WORK 72

8.1 Conclusion 72

8.2 Future Work 73

REFERENCES 75

APPENDICES 81

A Contents of the Jar file for LOGIN protocol 81

B Contents of the Jar file for IMAP protocol 83

C Contents of the Jar file for GNUTELLite protocol 91

LIST OF FIGURES

Page

Figure 2.1: The 4-layer TCP/IP model and the 7-layer OSI model 6

Figure 4.1: Either transition may fire simultaneously (concurrency). 20

Figure 4.2: Firing of one transition disables the other (conflict). 20

Figure 4.3: Hierarchical Modeling using Petri nets. 21

Figure 4.4: Illustration of Transitions Firing Rule in a High-Level Net 25

Figure 4.5: The Unfolded Net of the High-Level Net shown in Figure 4.4 26

Figure 4.6: Working of PUMPS 27

Figure 5.1: Specification and Implementation of Protocols 32

Figure 5.2: Interactions of a simple LOGIN protocol 33

Figure 5.3: Interpretation of CP-nets in PUMPS 34

Figure 5.4: Modeling of a priority system 36

Figure 5.5: DTD for PUMPSpec2 38

Figure 5.6: JXTA P2P Architecture 39

Figure 6.1: Channel Negotiation and Protocol Loading Sequence 49

Figure 7.1: CPN-Based Protocol Model for LOGIN 55

Figure 7.2: CPN-Based Client-side Protocol Model for IMAP 59

Figure 7.3: CPN-Based Server-side Protocol Model for IMAP 60

Figure 7.4: CPN-Based Protocol Model for GNUTELLite 69

CHAPTER 1

INTRODUCTION

1.1 Motivation

With the proliferating use of networking in day-to-day computing activities, more and more applications are networked as compared to their stand-alone counterparts. An important area of interest during the design and development of such applications is the design of the protocol(s) that would be used for communication among the interacting entities across the network, be it another instance of the same program or a service-providing application. The efforts involved in designing and implementing such application level protocols are made somewhat easier when formal methods and other specification techniques are used. Further, the use of software tools for verification and translation of the protocol specification prove to be invaluable in rapidly and correctly developing of network applications.

In today's networking world, given the mammoth presence of the Internet, peer-to-peer (P2P) networking applications are becoming popular. This technology offers an increased utilization of information, bandwidth, and computing resources in the Internet [LIG01]. However, given the heterogeneous composition of P2P systems, a peer may not know the protocols used by all services. Hence, providing users or peers with the ability to search as well as use services becomes a challenging issue. The issue becomes even more challenging as the P2P systems become more generic where no underlying assumptions can be made about the peers on the network.

In our work, we focus on

1. improving an existing mechanism, PUMPS [P.J00], for specifying application level protocols, and
2. demonstrating its applicability within a P2P infrastructure.

The protocol specifications are published within the peer network in a form of advertisements along with the peer groups offering the services associated with the protocol. A peer that needs access to a particular service searches for the protocol used by service. With the help of a protocol management framework, the peer then loads the protocol and creates a channel to that service. In arriving at the above architecture, the following were the main contributions of this thesis:

- Development of a protocol specification method for peer-to-peer systems based on PUMPSpec [P.J00].
- Development of P2P framework for deployment of protocol specification and its implementation including protocol advertisements, channel initiation sequence and dynamic-protocol-loading services (in JXTA). The Protocol Management Service (PMS) forms the crux of this framework.
- Implementation of the above framework using JXTA
- Validation of new framework by specification and implementation of popular protocols such as IMAP and their deployment using the P2P framework. Peers were also allowed to provide their own implementation apart from the default implementation provided along with the specification.

We discuss the above contributions in more detailed in Chapter 5. We have used a java reference implementation of JXTA Protocols as the P2P infrastructure and Petri nets as the specification mechanism.

1.2 A Motivating Example

Consider a traveler carrying a hand-held Internet-enabled device entering an airport. In order to board the correct flight, the traveler has to check the schedule of departing flights to his destination, information on whether the flight is on-time, any connecting flight updates, gate number and similar information. He might choose to go to various so-called service access points such as a ticket counter or he could use his hand held to connect to the airport's network and find the services he needs. Better still, the airport could upload a client application with a GUI onto the traveler's handheld through which the traveler can access customized information.

For any such "open system" in which entities can join and leave at will, the ability to access information or services without prior knowledge of the protocols involved, cannot be achieved unless the protocols can be dynamically loaded and used. Technologies such as the Bluetooth [Jaap98] provide protocols that manage connectivity between devices in a ad hoc networks such as the one described above but do not support the management of protocols used by the network services. This is what we have tried to achieve.

1.3 Thesis Organization

The rest of the thesis is structured in the following manner. After briefly presenting background information on the issues involved in Chapter 2, we give an overview of available process specification techniques and other formal methods in Chapter 3. In Chapter 4, we justify the use of Petri nets for protocol specification and the mechanism hence used. In Chapters 5 and 6, we discuss the design and architecture of our mechanism and the implementation specific details. Chapter 7 illustrates the working of the system with a few examples. Finally, in chapter 8, we conclude the work and discuss possible future work.

CHAPTER 2

BACKGROUND

2.1 Introduction to Networking

Computer networks are composed of a collection of computing devices (nodes) wherein data are transferred between the devices through the network Many networking technologies exist, such as Ethernet, that are mainly characterized by the underlying transmission mode and number of nodes they can handle. There are two main modes of transmission, which are

- Point-to-Point: Messages are transmitted from one point (or node) to another
- Multipoint/Broadcast: Messages are transmitted to multiple/all nodes on the network.

There are other special modes of communication, which are not relevant for our study. Depending upon the scale of a network, the network may be classified as LAN (Local Area Network) or MAN (Metropolitan Area Network) or WAN (Wide Area Network), in the increasing order of scalability. It is often possible to combine existing smaller networks into a single large network by using specialized computers such as hubs, bridges and routers, or more generally known as the Interface Message Processors (IMPs) [ASH99]. The Internet is a network of networks that virtually spans the globe [RIC94]. Applications that are designed for such large networks like the Internet need to be independent of knowledge about the underlying connections so that they might run on any node without changes.

2.2 Protocols and Network Layering

A typical protocol defines the order [ORG84] in which messages of different types should be exchanged for an overall successful communication. A protocol is a set of rules [SHA94] that governs the exchange of data. In addition, a protocol addresses the issues of how the sending and receiving nodes should indicate successful transmission or receiving of a message and how the integrity of the message is ensured. Networking protocols are normally developed in layers, with each layer responsible for a different facet of the communications. The set of layers and the protocols in each layer is collectively termed as the network architecture and the list of protocols used by one particular system or application, one protocol per layer, is called a protocol stack. Two of the widely accepted network architectures are the TCP/IP model (4 Layers) and the Open Systems Interconnection (OSI - 7 layers) model, with the former being more popular due to its use in the Internet. Figure 2.1 shows the TCP/IP reference model and the OSI reference model with mappings between them.

Each layer has protocols that perform different functionalities. For example, in TCP/IP, the link layer at the lowest level of the stack handles the details of physical interfacing of the transmission medium such as cables with the computer. The network layer manages the movement of packets around the network as it has protocol that can manage routing and management of packets. Similarly, the transport layer provides flow control between two communicating devices connected to the network. It is easy to understand that the decomposition of the stack into layers is a natural choice since it provides a way to enhance the above-mentioned functionality independently. Hence, a layer provides functionality while using the layer below for further communication and abstracting its functionality to the layers above. The application layer consists of applications that use the transport layer to send actual application data across to another application or a group of applications and hence is the actual initiator of any communication from the user's point of view. As discussed before, the rules of initiation

of communication and carrying out message exchanges is defined by protocols,. Each layer has a collection of its own protocols.

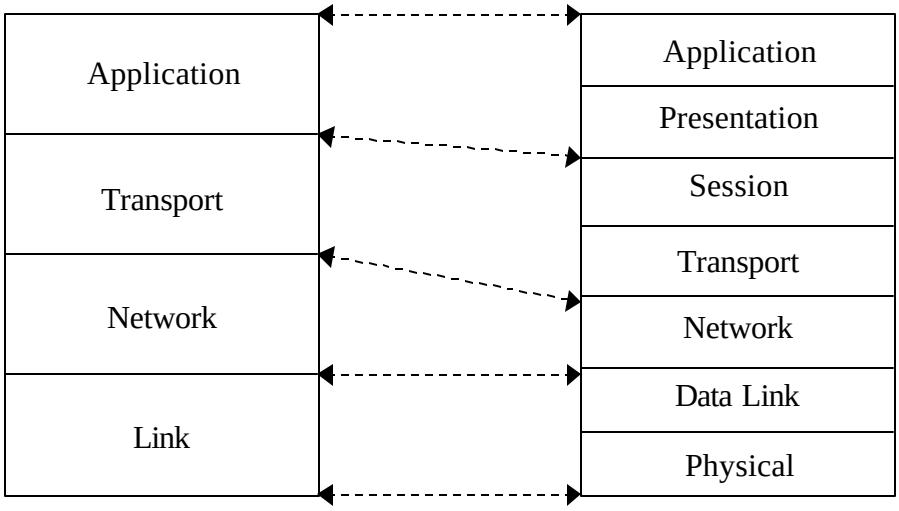

Figure 2.1: The 4-layer TCP/IP model and the 7-layer OSI model

2.3 Application Layer Protocols

In our work, we will focus on application-level protocols only. Some of the earlier protocols that are now standards include Telnet, FTP (File Transfer Protocol) and SMTP (Simple Mail Transfer Protocol). The Telnet protocol allows users to remotely access other host machines in the network. The FTP protocol is used to transfer files to and from remote machines and SMTP is an application-level protocol used to send and receive "electronic mail" over the network. Other popular protocols are SNMP (Simple Network Management Protocol), HTTP (Hypertext Transmission Protocol), NNTP (News Network Transport Protocol), NTP (Network Time Protocol), and IMAP (Internet Message Access Protocol).

2.4 Peer-To-Peer (P2P) Networking

Depending on the nature of the protocol, one might distinguish applications based on those protocols as belonging to the peer-to-peer (P2P) paradigm or the client-server

paradigm. Experts might argue that P2P should not be characterized [DAN02] strictly by the degree of centralization versus decentralization. This is true since centralization in P2P can exist in various degrees, which can be categorized as:

- *Fully Centralized*: Such as the client-server, where the service is available on one host and client requiring that service need to contact that particular host or server, e.g. Telnet [RFC854].
- *Brokered*: Wherein only certain but not all functionality is centralized such as *Napster* [OP.01]. The centralized functionalities are often bookkeeping operations such as registration information, service monitoring, active servers lookup, etc.
- *Fully Decentralized*: Wherein no two peers are different in their functionality, but only differ in the data or content they carry. For example, *Gnutella* [GNU] is a distributed file-sharing network, where no one peer is different from another except in the content that it shares.

However, in our references to P2P network we will refer to fully decentralized systems, unless and otherwise stated explicitly.

P2P applications are typically flexible and fault-tolerant as the chances of a large number of peers functioning correctly is more than a single peer functioning as a server. P2P applications can replicate data as needed and broadcast to multiple destinations thus increasing the availability of data. The overall performance of the P2P application tends to increase as the number of peers increases, as opposed to a strict client-server application where the performance degrades with increasing number of clients. This is typically because increasing the number of peers in a P2P application increases the resources available to the application but in a client-server system, increasing the number client decreases the resources available at the server. Note must be made that the

performance is also dependent on the application, the P2P protocol, and the network topology.

2.5 P2P Implications on Protocol Management

[DAN02] lists some of the common characteristics of today's typical P2P systems to include the following:

- Peer nodes have awareness of other peer nodes.
- Peers create a virtual network that abstracts the complexity of interconnecting peers despite firewalls, subnets, and lack of specific network services.
- Each node can act as both a client and a server.
- Peers form working communities of data and application (or services) that can be described as peer groups.

The characteristics prove to be very powerful and attractive to service providers and developers of network-centric applications who would like to deploy their service without the expense of setting up servers and maintaining them at their cost. However, in order to deploy their services or applications, the service provider needs some way to ensure that the client peers know which protocol to "speak" while using their services. There is currently no direct way and the service provider ends up developing an entirely new P2P system catering to peers which can do nothing more than only access the service for which the P2P system was originally developed (e.g., Gnutella [GNU] for file-sharing). This leads to two negative impacts:

1. The increasing number of services (or service providers) the number of P2P systems will increase, resulting in P2P systems losing their generality in practice due to absence of compatibility between systems, and
2. Users who wish to access more than one service have to actually switch networks to access the different services.

However, if we are able to adapt our mechanism by dynamically loading protocols to a P2P system and offer service providers or application developers a framework for publishing their service. Then, upon uploading the necessary protocol, any peer wishing to use the service, can download the protocol and simply use or invoke it. Hence, P2P systems not only provide us with a test bed to test and demonstrate our mechanism, but also are a typical infrastructure where our mechanism can be put to its best use.

CHAPTER 3

REVIEW OF PROTOCOL SPECIFICATION METHODS

3.1 Introduction

Even though there are not many methods developed purely for protocol specification, the areas of formalism such as Graph theory, Workflow Management and System Modeling have been influential in developing such methods. Protocol specification methods can be broadly categorized as follows:

- OS-Based: These methods provide high performance implementation of a protocol but typically, lack in ability to represent large complex protocols, e.g., the x-Kernel [Hut91].
- Programming (Annotations/Directives/Constructs) Based: These methods extend existing general-purpose languages to provide support for protocol specification. These methods inherit efficient compilers but are limited in expressiveness, e.g., Cicero [HUA93].
- Language-Based: These methods describe an entirely new language design specifically with protocol specification in mind. They have a higher level of expressiveness compared to the above methods but lack efficient compilers, e.g., Morpheus [MBA92].
- Formal Specification: These methods are the most generic and powerful in terms of expressiveness. They can be used in modeling very complex systems and hence prove to be a very powerful tool in representing protocol designs. However, they are not easily realizable and very few compilers exist, e.g., Estelle [SBP87], SDL [Z.100], and LOTOS [BOL87].

We will now discuss some of these specification methods in detail.

3.2 Operating System Based Methods

3.2.1 Kanga Framework

Kanga [GAR96] provides a framework for building protocols from basic blocks. The approach is based on the argument that for many applications like multimedia and databases, a typical TCP or UDP protocol does not fit their requirements and the functionality provided by these extremes is either an overkill or insufficient in most situations. So by using the basic building blocks of a protocol in an object oriented manner, one can build applications with protocols specific to their needs. For example, one might develop an application that uses a connectionless transaction-oriented protocol for querying a local server and a reliable, connection-oriented and rate-controlled transaction-oriented protocol while querying across the Internet. There are two categories of building blocks or classes in kanga: protocol classes and transport classes. While the protocol classes are used to specify functions specific to the protocol such as reliability and fragmentation, the transport classes are used as an abstraction to the actual communication across the network. The abstraction also includes an abstract base class that defines the standard interface to all protocol and transport classes and a wrapper class that joins multiple protocol class objects and a single transport class object into a protocol graph, by specifying which object is above and below each of the other objects in the graph.

3.3 Programming Annotations Based Methods

3.3.1 Cicero Linguistic Support

Cicero [HUA93] is a set of language constructs that is used to extend existing languages to support protocol development. Its aim is to provide constructs that deal specifically with synchrony, asynchrony, and concurrency in protocol execution. Cicero has a strong

foothold in the assumption that in the future, more of application-level functionality will be moved to protocol implementation to save application development time and with a increasing number of distributed applications, correct and efficient implementation and verification of protocols will require support of concurrency within the language itself.

Cicero is also based on another assumption that event-driven paradigm is a natural extension to protocol abstraction, where events are synonymous to messages. It borrows the notion of event patterns by which one can efficiently describe events and event combinations and relationships within events. Cicero advocates the use of event patterns [CVR89] for protocol design. More information on event patterns and the constructs implemented in Cicero with examples can be found in [YEN94].

3.4 Language-Based Methods

3.4.1 Data-Stream Language for Protocols

The philosophy behind use of stream based languages for protocol design is that they provide both structure in design and performance in implementation. It uses a data-stream architecture to provide the much-needed abstraction necessary for designing complex protocols. It also maps easily to protocol data processing problems, which leads to efficient implementations. The current research [Clay95] is restricted to applying data-stream architecture to protocol functions such as encryption, decompression, etc. It also supports pipelining of data stream functions by which smaller protocol functions can be combined to get functions that are more powerful. Another useful aspect of this language is the existence of a fast and efficient compiler. The compiler converts a data-stream program into a set of routines and a scheduler. The routines are executed till some event such as an empty buffer or buffer overflow occurs, upon which control is transferred to the scheduler. The scheduler has the task of picking the next routine appropriate for the current state. The system imitates the working of a finite state machine, which makes it easier to learn.

3.4.2 Application Level Protocol Specification Language

The ALPS [Ash99] Language is an attempt to provide a language that is efficient in specification and can be compiled into executable code easily and efficiently. The author envisions the communication model used by application level protocols to be similar to a finite state machine, where a state represents a discrete position reached by the client or server during communication. The client or server can go from one state to another by sending or receiving data, which can be considered a transition. In a finite state machine, only a finite number of previous inputs can affect the current output. Hence, they use an extension to the Deterministic Finite Automata (DFA). The state of the DFA is a complete description of the status of the client/server and transitions enable state changes by accepting data or sending data. Hence, each state is encapsulated into a function that executes in three stages: pre-process, action and post-process. A transition which links two states will have the starting state, the send or receive primitive and the ending state. The entire DFA is then put together in a specification file for both client and server. The resulting specification is compiled and code is generated. The protocol is then installed using a protocol handler. ALPS has an interesting feature of being able to modify the protocol at runtime that has the obvious advantages while developing or testing networked applications. However, the language is limited in its expressiveness, especially in its ability to represent concurrency or synchronization.

3.5 Formal Specification Methods

3.5.1 Specification and Description Language (SDL)

The purpose of SDL [Z.100] [IEC01] is to provide unambiguous specification and description of the behavior of telecommunications systems. It is a recommendation of ITU-T and can be used for specification of not only telecommunication systems, but also any complex, event-driven, real-time, and interactive application involving concurrent activities that communicate using discrete signals. The specifications and descriptions

using SDL are intended to be formal in the sense that it is possible to analyze and interpret them unambiguously. The terms specification and description are used with the following meaning:

- Specification of a system is the description of its required behavior.
- Description of a system is the description of its actual behavior.

A system specification in SDL is the specification of both the behavior and a set of general parameters describing properties of a system. SDL is an object-oriented approach to system design and analysis. SDL adapts from the FSM model with extensions and the capability for parallel execution. Systems, blocks, processes and procedures form the basic structure of SDL specifications. SDL uses standardized and generalized methods for representing data: ADT (Abstract Data Types) and ASN.1 (Abstract Syntax Notation). This enables sharing of data between languages and reuse of existing data structures. Current versions of SDL allow use of gates, which are points of interaction for channels within a process or block. It also introduces the concept of agents (blocks and processes bundled together) where an agent is characterized by having variables, procedures, and a state machine (given by an explicit or implicit composite state type). A system is considered the outermost agent containing all other agents in the design. The state machine of a block is interpreted concurrently with its contained agents, while the state machine of a process is interpreted alternating with its contained agents.

3.5.2 Estelle: A Specification Language for Distributed Systems

Estelle [SBP87] [ISO97] [SBP88], like SDL, is a standard formal description technique (FDT), used for specification of concurrent and distributed information processing systems (esp. communication protocols and services). It is based on an extended state transition (FSA) model. Estelle separates (optionally) the communication interfaces of the system components from their internal behavior defined as a module. A module can

thus be any process or activity and can be dynamically created or deleted. The internal behavior of a module is characterized by a non-deterministic state transition system that is defined by a set of states, an initial state and a next-state relation. A state is, in general, a complex structure composed of many components such as: value of the control state, values of variables, contents of FIFO queues associated with interaction points and a status of the module's internal structure (sub-module instances, bindings between interaction points, etc.). The global situation, i.e., the collective states of the subsystems,, is expressed by

- a hierarchical structure of module instances within the specified system. The structure includes bindings between module interaction points and the local state of each module.
- the transitions that are in parallel execution within the system. The transition from/to global situations are defined using next-situation-relation that is obtained by union-set operation of next-state of each module in the system.

Hence, the overall behavior of the system is characterized by the set of all sequences of global situations that can be generated from a given initial sequence.

3.5.3 Language of Temporal Ordering Specification (LOTOS)

LOTOS [BOL87] [LMH92] is another FDT developed within ISO for formal specification of the open distributed systems and in particular for those related to the OSI (Open Systems Interconnection) architecture. The language is based on an extension of Calculus of Communication Systems (CCS, which is based on process algebra) [MIL89] and Hoare's Communicating Sequential Process (CSP) [HOA85]. The primary design criteria for the language were high expressiveness, formal definition, abstraction and structure. In LOTOS, a distributed concurrent system is viewed as a process consisting of possibly many sub-processes. LOTOS hence describes a system by defining a hierarchy of processes, where each process is capable of internal actions and interaction with other

processes or its environment in general. Much of the power of LOTOS is seen in the way complex interactions between processes can be built out of elementary units of synchronization called events or actions. Interested readers are directed to [LMH92] for more details on system definition using LOTOS.

3.6 Web Services Conversation Language (WSCL)

WSCL [WCN01] is a specification language, which allows definition of abstract interfaces for Web services [WWS02]. It specifies the XML documents being exchanged and the protocol for the exchange. WSCL conversations are themselves XML documents and hence can be interpreted by the Web services infrastructures and development tools. In the Web services model, the messages sent to and from a Web service are in the form of XML documents. By specifying the order of exchange of these XML documents, we are defining the external behavior of the service and hence, its abstract interface. Allowing the concepts of abstract interface, communication protocol, and concrete service, we allow services to interact with each other in a dynamic and loosely coupled way. WSCL can be used to create service frameworks that enable service implementers to offload the responsibility for conversation related tasks to the infrastructure. From this perspective, the objective of WSCL is similar to ours. However, WSCL is a new model and specific to Web services, hence its applicability is not clear in general applications.

3.7 Active Networks

With the increase in the applications that have varied service requirements over a network, the one-size-fits-all single-service model of Internet Protocol (IP) is fast becoming out-dated. Active networks [K.P99] seek to avoid the IP model by making the network components (router, switches, messages) programmable, hence adapting to the needs of the service, on the fly. Traditionally, network components have had limited processing capabilities of routing, congestion control and quality of service schemes due

to reasons such as: difficulty of integrating new technologies, poor performance due to redundant operations at several protocol layers, and the difficulty in accommodating new services due to architecture constraints. Active networks address these issues by making the network components programmable (as well as messages) by services and users of the network. Active networks are realized by an active packets approach, where the packets carry active code or by an active nodes approach, where packets carry some identifiers or references to predefined functions that resides in a node. SNAP [JTM01], Smart Packets [Bev98], and M0 [A.B97] are some example of active packets network and ANTS [DJW98], and DAN [D.D98] describes some active node architectures. A combination of the two approaches can also be used.

Active networks provide a neat platform for deployment of new protocols. Protocol specification or code can be sent to various active nodes and nodes can be configured to use the new protocol. This will be very useful for applications that need customized communication requirements. However, as far as application level protocols are concerned the focus is still on the efficient design and implementation.

3.8 Petri Nets

Petri Nets [Pet62] are used as a mathematical modeling tool [DaT77]. They are designed with notions of concurrency, non-determinism, communication and synchronization that are often seen in distributed systems. A Petri net can be viewed as a directed graph [J.L81] [DaT80] with nodes and arcs. The nodes are of two types – places and transition. Each place holds a certain number of tokens, which are analogous to messages, data elements, or event-occurrence records. An arc connects a place with a transition or vice-versa. We can have multiple places connected to a transition or multiple transitions connected to a place. The arcs can be weighted with some positive number (default=1), which represents the number of tokens to be seen at the incoming place for the transition

to become active. When all the places have tokens as per their outgoing weighted arcs to a particular transition, the transition is said to be active or enabled and can be fired at any given time. When an enabled transition fires, tokens equal to the weight of the outgoing arcs are generated and put in their respective places. The distribution of tokens among the places at any given time is called the "marking" of the Petri net and the initial distribution is called the "initial marking". Firing a transition also removes tokens that enable the transition from each input place. Each transition can have an optional guard function, which can be used to impose extra constraints for firing of an active transition. Hence, a transition is enabled only when it is active and its guard evaluates to true. Mathematically [AgT79], a Petri Net is a bipartite, directed graph N=(T,P,A) where

$T = \{t1,t2,\ldots,tn\}$ is a set of transitions

$P=\{p1,p2,\ldots,pm\}$ is a set of places ($T \cup P$ form the nodes of N)

$A \subseteq \{ T \times P \} \cup \{ P \times T \}$ is a set of directed arcs.

A marking M of a Petri net is a mapping:

$M: P \rightarrow I$ where $I = \{0,1,2,\ldots..\}$.

M assigns tokens to each place in the net. Hence a Petri net N = (T, P, A) with marking M is a *Marked Petri Net* C = (T, P, A, M).

Colored Petri nets (CPN) (CP-nets) [JEN81] [P.J00] extend the concept of simple tokens to colored tokens and simple places to places with a color set. Hence, for a transition to become active, certain colored tokens of certain color-set needs to be present in the input places. By using CP-nets, one can achieve a high amount of compaction since many of the sub-systems in a real-time system are similar (but not identical) and associating data values to tokens (in terms of color constants) ease defining behavior variations. Other than CP-nets, certain other variants also give powerful capability to a regular Petri net. Timed-Petri nets allow delays in transaction and stochastic Petri nets associate delays as with places (seen in buffering).

CHAPTER 4

PROTOCOL SPECIFICATION USING COLORED PETRI NETS

1.1 Introduction

As discussed in the previous chapter, A Petri net is a graphical and mathematical modeling notation applicable to a wide range of applications. [J.L81] describes that a Petri net, like the system which it models, can be viewed as a sequence of discrete events whose order of occurrence is one of the possibly many allowed by its basic structure. This leads to *nondeterminism* in Petri net execution. If at any one time, more than one transition is enabled, then any of the transition may fire in a nondeterministic fashion, i.e. randomly or by forces not modeled in the system. While nondeterminism is advantageous from a modeling point of view, it does introduce considerable amount of complexity into the analysis of Petri net models. Petri nets reduce the complexity of analysis by enforcing the rule that firing of events should be considered as instantaneous. This implies that probability of any two or more events happening simultaneously is zero i.e. two transitions cannot fire simultaneously, since time is a continuous function.

The nondeterministic and non-simultaneous nature of firing of transitions in the modeling of concurrent systems takes two forms. One of these, shown in Figure 4.1, represent simultaneous events that may occur in any order and the second, shown in Figure 4.2, represents two enabled transitions that are in *conflict*, where only one transition will fire, since in doing so the other transition becomes disabled. The advantage [DIAZ84] of Petri nets over many other graphical modeling tools is that it has a mathematical formalism that makes the dynamic behavior of the underlying system

well defined, and amenable to theoretical analysis using results from e.g. linear algebra and graph theory.

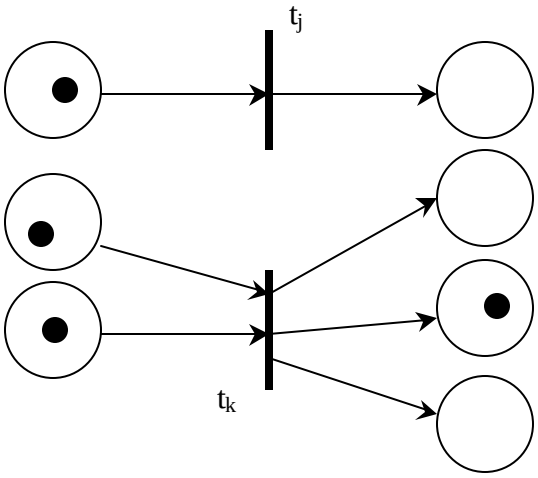

Figure 4.1**:** Either transition may fire simultaneously (concurrency).

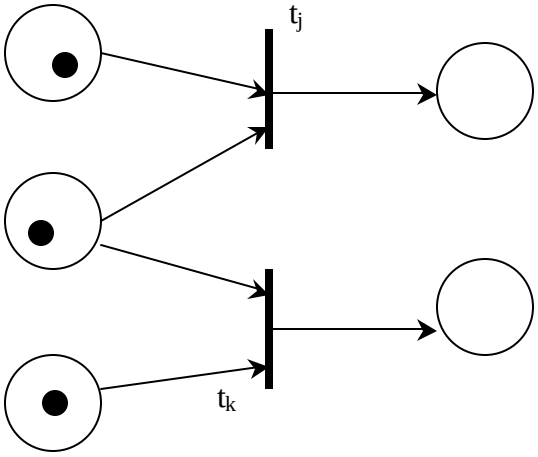

Figure 4.2**:** Firing of one transition disables the other (conflict).

The concepts of concurrency and conflict form the basis of using Petri nets for protocol specification. With token being analogous to messages, it is obvious to deduce that places represent a particular state or collection of states of a protocol whereas the transitions represent the associated action.

Another valuable feature of Petri nets is their ability to be modeled hierarchically, analogous to the procedural abstraction in programming languages. An entire net may be replaced by a single place or transition while modeling at a more abstract level; *abstraction* or places and transitions may be replaced by subnets to provide more details; *refinement*. This feature is particularly used in protocol specification when a large protocol is divided into smaller parts or one protocol uses another within itself. Figure 4.3 illustrates the same.

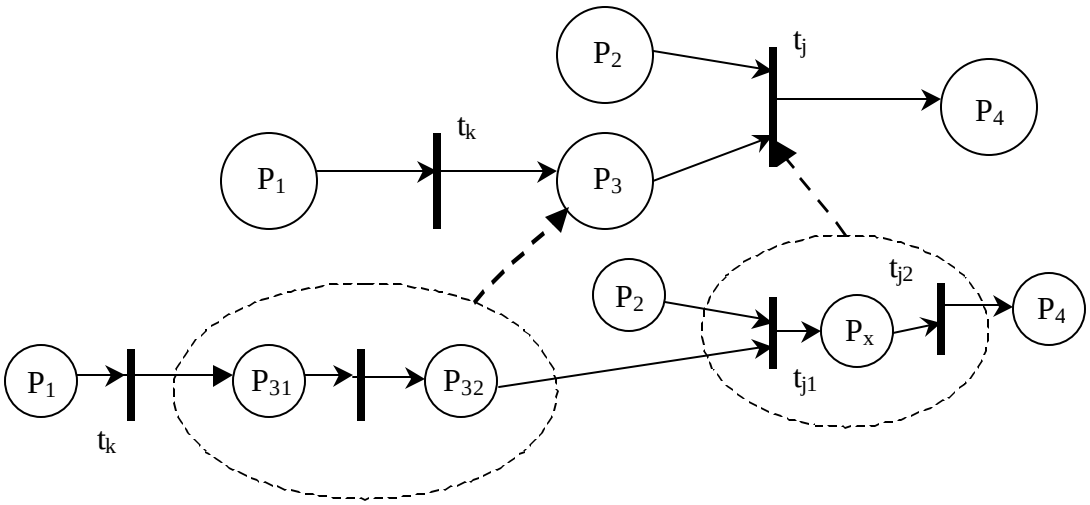

Figure 4.3**:** Hierarchical Modeling using Petri nets.

The meaning of markings in a Petri net and the execution rules for a marked Petri net has been described in the previous chapter. We now define the state of a Petri net. The state of a Petri net is defined by its marking. Thus, the firing of a transition represents a change in the state of the net. The *state space of* a Petri net with n places is the set of all markings. Given a marking and a set of possible transitions, the set of markings that is reachable by firing of transitions in the transition set, are called the *immediately reachable* markings. The *reachability set* of a marked Petri net is the set of all states into which a Petri net can enter by any possible execution. Hence, many analysis questions deal mainly with the properties of the reachability set of a Petri net.

4.2 Behavioral Properties

[TAD89], in his tutorial-review paper on Petri nets, discusses the support for analysis in Petri nets. Interested readers are recommended to this reference for more details. Two types of behavioral properties can be studied with a Petri net model: those, which depend on the initial marking, and those that are independent of the initial marking. The first type is called a behavioral property and the second is called a structural property. In this chapter, we will refer only to the *liveness* and *safeness* behavioral properties as they are often used as correctness criteria in protocol specification. Reachability is also an important behavioral property and it has been discussed in the previous section.

4.2.1 Liveness

Liveness is closely related to the complete absence of deadlocks. A Petri net is said to be *live* if it is possible to fire any transition in the net by going some firing sequence, no matter what the current marking is. This guarantees a deadlock-free Petri net. However, a non-live Petri net doesn't always imply that the Petri net is not deadlock free. Liveness is considered an ideal property for systems and often is impractical. It is also costly to verify this property for many complex systems. However, liveness ensures deadlock free operation of the Petri net, which is often essential in protocol specification.

4.2.2 Safeness

Safeness is a property that is often related to the *boundedness* of a Petri net. A Petri net is k-bounded or simply bounded if the number of tokens does not exceed a finite number k for any marking reachable from the initial marking. In addition, a Petri net said to be safe, if it is 1-bounded. Places in the Petri net are often used to represent buffers and registers for storing intermediate data (in form of tokens). By verifying that the net is bounded or safe, it is guaranteed that there will be no overflows in the buffer or registers, no matter what firing sequence takes place.

Other behavioral properties include reversibility, cover-ability, persistence, synchronic distance, and fairness. The properties of Petri nets have been well analyzed and mathematically formalized [Jav90]. There are also many tools and techniques available that can be used for verification of these properties.

4.3 Petri Net Variations

The theory of Petri nets has been developed from foundation work by Carl Adam Petri. It came to the attention when a group of researchers led by Anatol Holt, developed the theory of "systemics" [HAW70], which was concerned with representation and analysis of systems and their behavior. Since then many variations of Petri nets are available that the can be used according to a system requirements for complex and analysis. [L.B92] makes an attempt to classify Petri net but it is only of historic interest. However, we list them here to get a quick overview of the main differences between the various forms of Petri nets. The first form of Petri nets are characterized by places that are represented by boolean values. Some of the nets that fall under this category are the CE-nets or Condition-Event nets. This net allows each place to contain at most one token representing a boolean value. Other examples are Elementary (EN) nets, 1-safe systems and state machines. The second form of nets is characterized by places that can be represented by integer values i.e., a place is marked by a number of unstructured tokens. PT nets or place-transition nets discussed in earlier sections is an example of this form of nets. Other examples are free-choice nets and marked graphs. The third and final form of nets is high-level nets and is characterized by places that can represent high-level values, i.e. a place is marked by a multi-set of structured tokens. Environment-Relationship Nets, Predicate-Transition Nets, and Colored Petri Nets are good examples of this category. Most practical applications of Petri nets use either PrT-nets or CP-nets. There is very little difference between the two. CP-nets have two different representations: expression representation that uses arcs with expressions and guards; and the function representation

that uses linear functions between multi-sets. Throughout our work and this thesis, we will mean the expression representation of CPNs when we refer to CPNs in general, unless and otherwise specified.

4.4 Colored Petri Nets

With specification of very large systems, Petri nets tend to grow graphically unreadable. The main reason for this drawback is that in a PT net, we have to represent two similar but non-identical processes by two separate subnets. Unfortunately, this problem is very typical in real world systems, impeding the practical use of Petri nets. CP-nets belong to the class of high-level nets that can achieved a high level of compaction by equipping each token with an attached data value called the *token color*. The data value may be an arbitrarily complex type such as a structure with many fields of varying types. For a given place all tokens mush then have token colors that belong to a specified type. This type is called as the *color set* of a place. The use of color sets for colors is analogous to use of types for variables in programming languages. Attaching a color to each token and a color set to each place allows us to use fewer places and transitions that PT nets, however this also means that token colors can be inspected by the transitions, which means that enabling of a transition may depend upon the color of the input tokens. Further, the color of the output tokens can also depend on the color of the input tokens. For such a situation, CP-nets provide more complex arc expressions and firing rules than PT-nets. In addition, in addition to the arc expression, a transition is allowed to have a *guard* expression or simply a guard, which is a boolean expression with purpose of providing an additional constraint which must be fulfilled before the transition is enabled. [TAD89] has an illustration, reproduced in Figure 4.4 for convenience, that shows the transition firing rule of high-level nets which can be applied to CP-nets. The illustration doesn't show guards but it is not difficult to construct one with it.

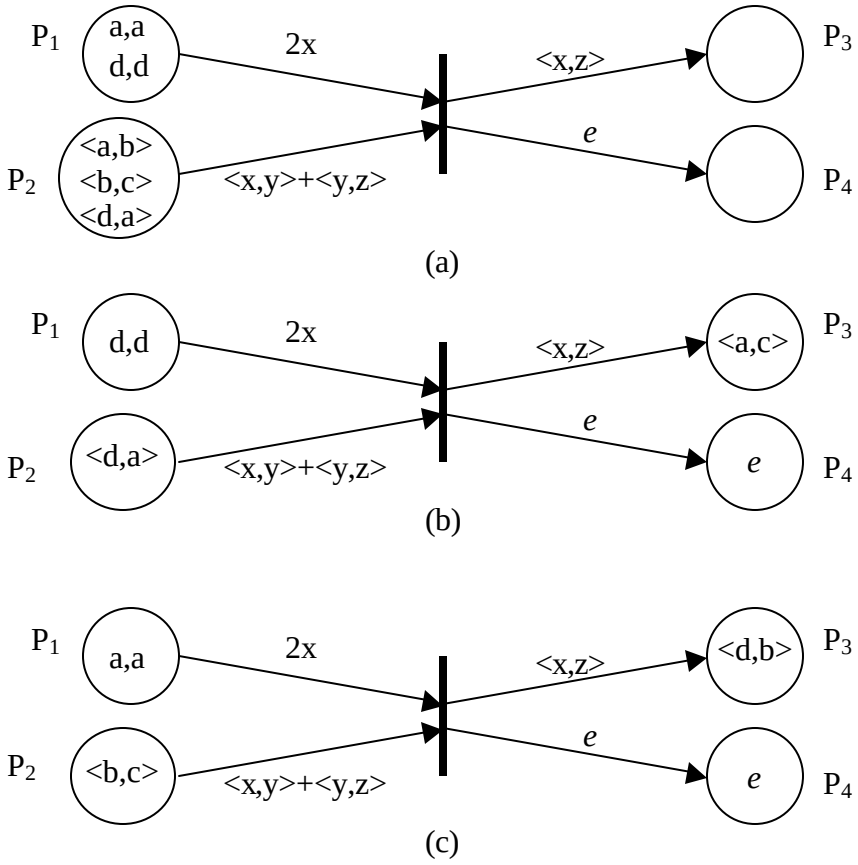

Figure 4.4**:** Illustration of transitions firing rule in a high-level net

The net consists of one transition t and four places (two input places p1 and p2, and two output places p3 and p4) The four arcs are labeled with expressions and each arc label dictates how many and which kinds of colored tokens will be removed from or added to the places. Here, when transition t is fired, the following will occur: p1 loses two tokens of the same color <x>, p2 loses two tokens of different colors <x,y> and <y,z>, p3 gets one token of the color <x,z>, and p4 gets one token of color, e (a constant). With the initial marking as shown in Figure 4.4(a), the nets shown in 4.4(b) and 4.4(c) show the markings after firing t with substitutions (a|x, b|y, c|z} and {d|x, a|y, b|z} respectively. It is interesting to note that a high-level net can be considered as a structurally folded version of a regular Petri net if the number of colors is finite. For example, Figure 4.4(a) can be unfolded into the regular Petri net as shown in Figure 4.5.

The reason we selected CP-nets for protocol specification is mainly that protocols involved in large systems consist of sub systems, which are usually similar if not identical. Moreover, graphical nature of CP-nets appeals to human users. They also have well-defined semantics that form the basis of formal analysis. It is also relatively easy to learn as they have very few primitives. The definition of CP-nets is short and builds upon concepts that are already known to system designers. Moreover, one could include or exclude features like timed nets or inhibitor arcs in a given Petri net to get their own customized variation, which suits their system better.

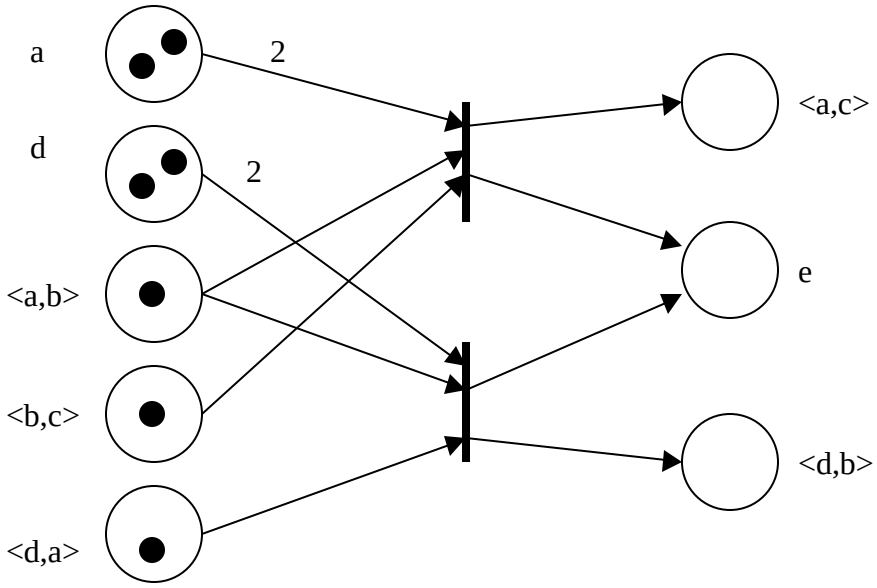

Figure 4.5: The unfolded net of the high-level net shown in Figure 4.4.

4.5 Examples of Systems Implemented Using CPN

Protocol specification is mentioned in literature [DaT80] [DIAZ84], as one of the main applications of Petri nets. Current research seems to be focused on design and verification of communication systems using Petri nets. In this section, we list some of these recent technologies.

4.5.1 PUMPS: Protocol Uploading and Management Protocol Server

It is important to understand PUMPS [P.J00] since our work is based heavily on this project. This project is an attempt at developing a universal protocol-server architecture that will enable the clients to upload their protocols and manage the same. One has to note that the client for PUMPS could be a service-server or a service-client forming the client-server architecture that needs to talk a particular protocol supported by PUMPS. We will hence call it a protocol-client for clarity. To design the protocol, the authors propose the use of Colored Petri Nets. The "places" in CPN are considered equivalent to the "synchronization points" called SYNCPOINT and the transitions are equivalent to some "action" called PSTATION. Actions include sending and receiving of data. Tokens are analogous to messages in the communication system. Each PSTATION has an associated input expression, guard and output expression along with its action procedure. During the implementation phase, the implementers will need to supplement code to do the actual evaluation of input expressions; guard and output expressions for every incoming and outgoing arc associated with a PSTATION and write the action code for that PSTATION. This design specification and its implementation classes are then sent to the universal server that uses the specification to initialize a protocol data object that simulates the Petri net-like state machine. Figure 4.6 illustrates the high level working of PUMPS.

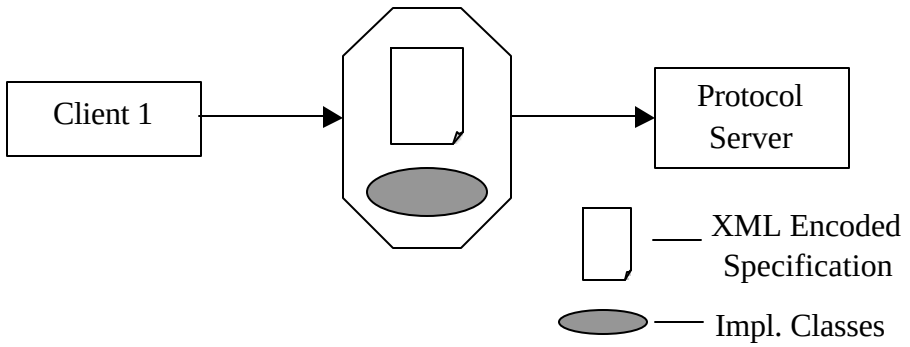

Figure 4.6: Working of PUMPS.

The system has the capability of having many clients share the same protocol by using a unique id assigned to each protocol. This architecture is especially useful in servers that provide customized service to their clients. The client can choose from a given set of protocols or upload its own protocol (conforming to the server specification for message formats, etc). However, even through it is easy to conceive a server side behavior or client-side behavior using this architecture, PUMPS would not be truly universal unless both client and server behavior can be conceived in a unified view. This work doesn't address the issues from this perspective. Security is another open issue in this work.

4.5.2 CP-Nets for Conversation Modeling

Protolingua [Cost99] discusses structuring communicative interactions among agents, by organizing messages into relevant contexts, called conversations. A conversation is thus defined as a pattern of message exchange that two (or more) agents agree to follow in communicating with each other. In effect, a conversation is a communication protocol though it may be initiated through negotiation and may be short-lived compared to protocols. A conversation specification hence would be a specification shared among agents and would contain information about the conversation and about the agents that will use it. To specify the pattern of message exchanges in a conversation specification, Cost et. al.[Cost99] investigate the use of CPNs as an alternative mechanism to finite automata. Agents use a common language named Protolingua, for manipulating CPN-based conversations. Protolingua uses IDL for the association of well-known functions and data types with a CPN framework. Protolingua was kept simple to facilitate the porting of interpreting across platforms.

To understand the relationship between the above introduced system components, let us consider a situation where a Java-based agent would like to converse with another agent, and that it has determined (by some means) that protocol A is needed. It then obtains the declarative specification for A, if it doesn't already have it, from the other

agent or from a third party (probably a broker). Description A contains the specification using Petri net, a schema, and functions given in IDL. The agent can then obtain the executable attachments and type specifications appropriate for its interpreter (Java classes) and use the protocol to engage the other agent. Note that the purpose of IDL is only for identification and retrieval of executable modules and not the interaction of distributed components. The core purpose of developing such an architecture is to facilitate the analysis and verification of conversations. Cost et. al. have used a CPN analysis tool, Design/CPN, which uses CPN-ML (modeling language), to which the specification is translated. The interpreter and a demonstration of the technology have not yet been realized.

4.5.3 Trellis Hypermedia Model

Trellis [Nav96] is a Human-Computer Interaction initiative to investigate the application of automata for structuring hyperdocuments and group collaboration protocols. Trellis has a client server architecture where the central server process (engine) has no user-visible interface of its own and implements a Petri net object that contains both the structure (and state) and behavior of the document. One or more client processes communicate with the server (via RPC) to give users some visible representation of information annotated on the net engine. The net and its annotations are interpreted by interface clients as links and node contents of a hypermedia document. Server processes (open documents) represent instances of some hyper textual document description. This is also used for specifying collaboration protocols (specifications of how group members interact in collaboration), which is why this interests us.

The Trellis model is an annotated net where places are annotated with node content (such as file names); transitions are annotated with link anchor names and timing information; and arcs are annotated with display controls and color expressions. Timing is used on transitions to support non-user involved interactions. Most of the activity is

seen and controlled in a browser client, which provides a visual interface to give the user a tangible interpretation of the active net and its annotations. Each place in the net that is marked with a token is said to have active content; for each such place, a client can take several actions. First, the client will query the net engine to get the content annotations on the marked place; the file will be rendered appropriately. Then, the client will further query the engine to get names of all transitions that are enabled leading out of the marked place (usually the links to other content nodes). When the user selects a particular link, the client requests the net engine to fire the associated transition. Tokens are moved and the client again renders the active contents. By introducing colors, we obtain a "color-specific browsing model". The colored net model and collaboration protocols in Trellis were developed to support coordinated efforts of multiple users. In most collaboration protocols, colors are used to distinguish individual members or groups of members. The applications are also implemented with multiple clients, where each user invokes separate instances from a set of relevant clients.

One advantage of the model is that it allows dynamic modifications to the net (Trellis protocols are interpreted and not compiled). This provides the programmer/moderator of a protocol the ability to change the specification 'on-the-fly'. Though it is an interesting architecture, its main purpose is in managing hypermedia documents and not designing communication protocols, which makes a straightforward adaptation difficult.

4.5.4 Other Systems

[H.V98] describes an approach for modeling and implementing Active Objects [Kie96] with Generative Communication [Gel85] using CP-nets. A protocol design for detecting mobile agent clones has also been developed by using CP-nets [JuB98]. Interested readers are directed to the references.

CHAPTER 5

PEER-TO-PEER PROTOCOL MANAGEMENT ARCHITECTURE

5.1 General Overview

In the previous chapter, we discussed the Colored Petri Nets and their application to protocol specification. We also discussed the Protocol Uploading and Management Protocol Server [P.J00] PUMPS that envisions a central universal *protocol server* through which clients can upload protocols and management. As of now, the system allows clients to install a protocol that other clients may access using a unique identifier and run or stop running protocol. All these management functions use Protocol Management Protocol (PMP) for communication with the server, which all clients must know. In fact, PMP by itself can be installed into the server by the server and be identified by a special and reserved identifier.

As introduced in Chapter 1, such a system can be put to its best use in a peer-to-peer (P2P) infrastructure. The open architecture of a generalized P2P networks raises the interesting question: how does a peer learn to use a service with which it has not interacted before and has no knowledge about the communication protocol. We try to address this question by adapting PUMPS into a P2P system, namely JXTA [SLi01]. We continue to use the protocol specification technique that uses CP-nets, with some enhancements, as done in PUMPS. The CP-net specification named as PUMPSpec2 is converted into an XML [XML00] document and accessed through a Universal Resource Indicator (URI).

5.2 Design and Framework

Shown in Figure 5.1 illustrates the specification and implementation of protocols using the protocol management service in JXTA. Initially, the protocol is designed using CP-nets, which is then encoded into an XML document named PUMPSpec2 and stored at an URI. After verification, the implementation classes are archived and stored at another URI. The location of the implementation is added also to the PUMPSpec2. In the JXTA world, for every service, a peer group is created. According to JXTA protocols, every peer group inherits the services and applications that belong to its parent. The Protocol Management Service (PMS) can thus be assumed as loaded into a group that is at the top of the parent tree. Hence, PMS is available to all sub-groups. JXTA-related terms and concepts are explained in detail in the next section.

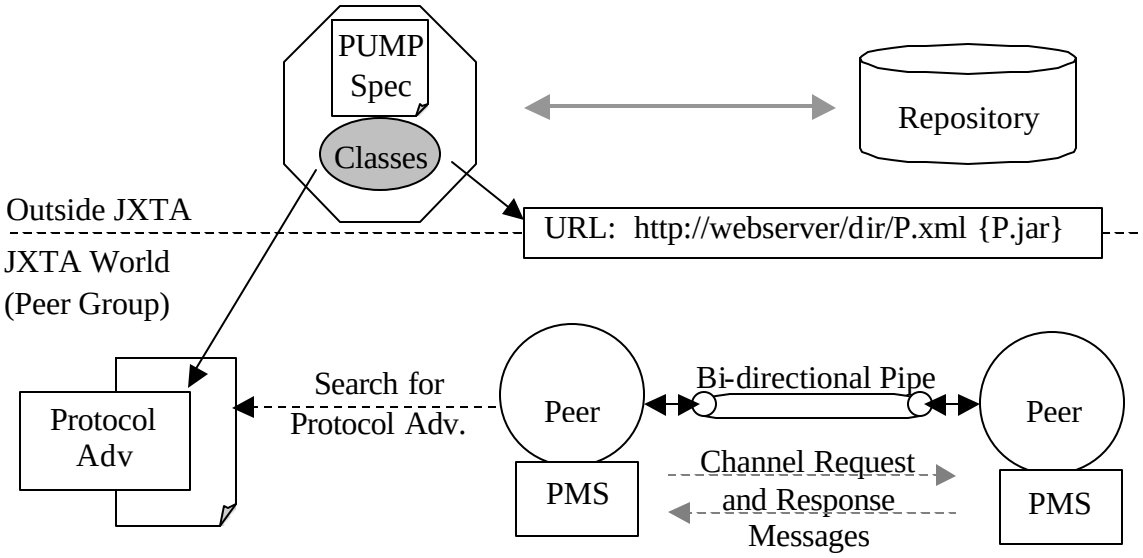

Figure 5.1**:** Specification and Implementation of Protocols

5.3 Protocol Specification

5.3.1 PUMPSpec

As stated earlier, we use the protocol specification technique PUMPSpec used in PUMPS [P.J00] to create our own specification named PUMPSpec2. A brief overview of the techniques and the enhancements applied to that technique is described in this section. In any application level protocol the interaction are in the form of queries and responses. Depending upon the stateful-ness or statelessness of protocol, session information may or may not be maintained as in a stateful protocol. Sessions are typically composed of multiple query-response interactions and may require storing intermediate data at either side. Figure 5.2 shows the interaction of a simple login protocol where username and password are sent to the server and the server responds with a success or login failure.

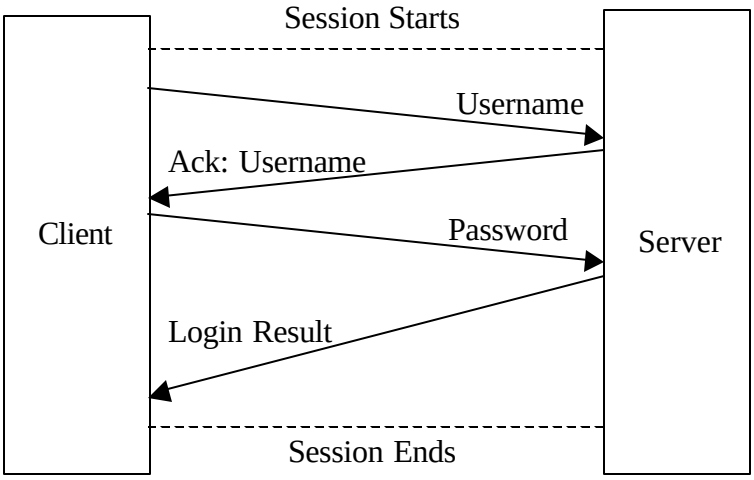

Figure 5.2: Interactions of a simple LOGIN protocol

In PUMPS, the protocol represented by a CP-Net is interpreted in the following way. Transitions correspond to actions and are called processing stations (PSTATION) and places correspond to synchronization and are called sync points (SYNCPOINT). The tokens are analogous to messages in the protocol, which is collection of name-value pairs of a certain type. The overall state of the protocol is maintained by the markings in the

places or synchronization points of the CP-net. Finally, each processing station has its own set of incoming and outgoing arcs (CONNECTOR) associated with their corresponding input and output sync points. In addition to the above, each arc expression and guard expression (associated with a PSTATION) is implemented as a method that returns true or false. Figure 5.3 below shows the interpretation of CP-nets in PUMPS.

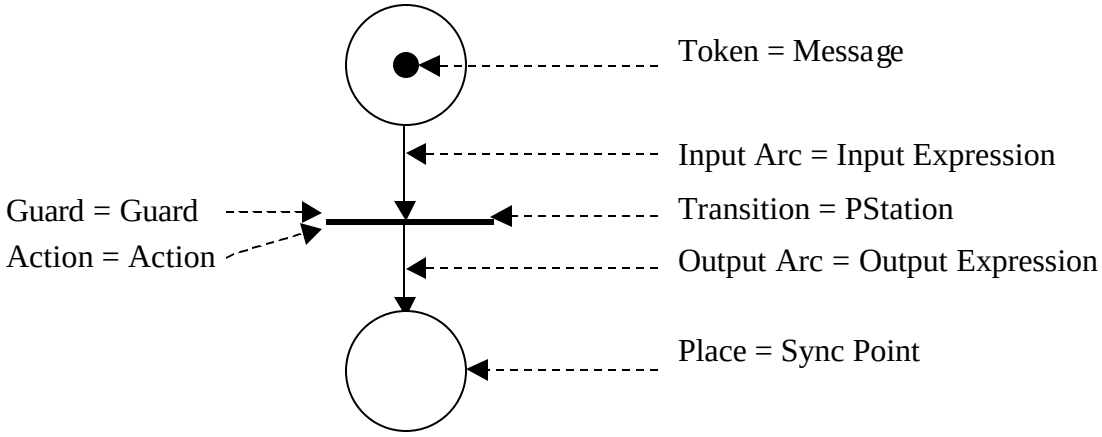

Figure 5.3: Interpretation of CP-nets in PUMPS

Before we see the encoded XML document (PUMPSpec) for a protocol, let us enlist the improvement made to PUMPS in this project.

5.3.2 PUMPSpec2: PUMPSpec Enhancements

5.3.2.1 Unified View of Protocol

The original design of PUMPS involved modeling the server side only. The absence of client side modeling made it impossible to show the synchronization between a client and a server, which is most essential while designing protocols. In PUMPSpec2, users have the capability to specify both sides of communication. The places that synchronize the client with the server (or any two communicating entities), typically representing the underlying medium for communication, are replaced with appropriate SEND and RECV processing stations and data buffers.

5.3.2.2 Hierarchical Modeling Capability

Petri nets by their very nature can model system hierarchy efficiently [J.L77]. Figure 4.3 in the previous chapter illustrated the hierarchical modeling capability. An entire net may be replaced by a single transition or place for abstraction or places and transitions may be replaced by a subnet for more detailed modeling or refinement. It is important to understand this ability is not the same as substitution, where a net can be substituted for a place or transition more than once by supplying the associated arc expressions. Substitution is difficult to achieve, especially in high-level nets and increases the complexity of the net that is not desired. In PUMPSpec, a processing station or a sync point may optionally specify an URI, a location, where the specification for the subnet may be found. Recursive refinement is not allowed but nested subnets can be conceived.

5.3.2.3 Inhibitor arcs

[J.L77] suggests a fundamental extension of Petri nets (taken from various sources) in response to the difficulties in modeling of priority systems. The extension is also called as the "zero-testing" [KRM74] or introduction of "inhibitor" arcs. In Figure 5.4, the introduction of inhibitor arcs from a place b_1 to a transition c_2, allows the transition to fire only if the place b_2 has zero tokens in it. Hence, when b_1 and b_2 have tokens, c_2 is inhibited from firing and thus c_1 gets a priority over c_2. [J.L77] describes the addition of inhibitor arc as a major extension to the concept of Petri nets. Petri nets extended in this manner have the modeling power of a Turing machine [AgT74]. Other extensions like the introduction of priorities between transitions, time-boundedness on transition firings and constraint sets [PSS70] are equivalent to Petri nets with inhibitor arcs.

5.3.2.4 Definitions of User and System places

PUMPSpec2 allows the creator of the specification to specify SYNCPOINTs with "user" or "system" permission. Often a user executing a protocol finds it necessary to place

messages or tokens at SYNCPOINTS. These messages usually represent client data or trap messages. SYNCPOINTs with “user” permission allow users to put and remove messages from them and SYNCPOINTs with “system” do not allow the same, hence are used only for storing internal tokens or messages.

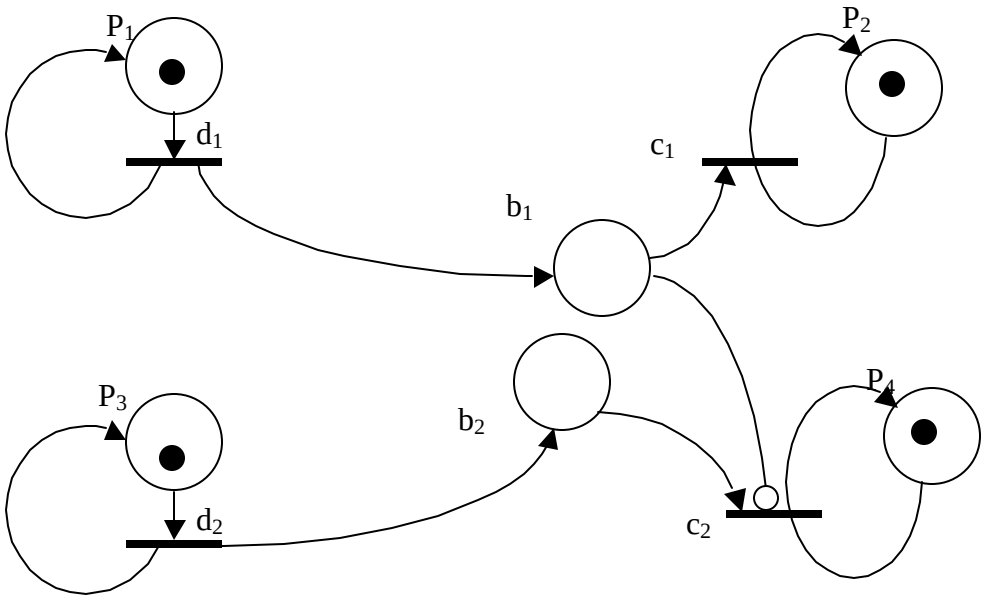

Figure 5.4: Modeling of a priority system

5.3.3 XML-Encoded PUMPSpec2

The protocol specification PUMPSpec2 is encoded into an XML (eXtended Markup Language) document. For a primer on XML, readers are directed to [BRA98]. Figure 5.5 shows the Document Type Definition (DTD) for the PUMPSpec documents. Within the protocol management service, this DTD is used to check the well-formed ness of the PUMPSpec. A well-formed document is parsed to form a protocol object that contains all of the needed information to run the protocol.

The following DTD is for PUMPSpec2, which includes the enhancements to PUMPSpec discussed earlier.

```
<!DOCTYPE ProtocolSpecification [
        <!ELEMENT ProtocolSpecification (ROLE+)>
        <!ATTLIST ProtocolSpecification
              name CDATA #REQUIRED
              version CDATA #REQUIRED
              defaultrole CDATA #REQUIRED>

        <!ELEMENT ROLE (SYNCPOINT+, PSTATION+, GNF)>
        <!ATTLIST ROLE
              name CDATA #REQUIRED
              type (PSTATION|SYNCPOINT) #IMPLIED
              source CDATA #IMPLIED
              sink CDATA #IMPLIED>

        <!ELEMENT SYNCPOINT (MESSAGE*)>
        <!ATTLIST SYNCPOINT
              name  CDATA #REQUIRED
              subnet CDATA #IMPLIED
              permission (user|system) “system”
              subnetrole CDATA #IMPLIED>

        <!ELEMENT PSTATION (CONNECTOR*, INHIBITOR*)>
        <!ATTLIST PSTATION
              name CDATA #REQUIRED
              action CDATA #IMPLIED
              guard CDATA #IMPLIED
              subnet CDATA #IMPLIED
              subnetrole CDATA #IMPLIED>

        <!ELEMENT CONNECTOR (#PCDATA)>
        <!ATTLIST CONNECTOR
              kind CDATA #REQUIRED
              syncpoint CDATA #REQUIRED
              expression CDATA #IMPLIED>
        <!ELEMENT INHIBITOR (#PCDATA)>
```

```
<!ATTLIST INHIBITOR
      syncpoint CDATA #REQUIRED>

<!ELEMENT MESSAGE (ATTRIBUTE*)>
<!ATTLIST MESSAGE
      name CDATA #REQUIRED
      type CDATA #REQUIRED>

<!ELEMENT ATTRIBUTE (#PCDATA)>
<!ATTLIST ATTRIBUTE
      name CDATA #REQUIRED
      value CDATA #REQUIRED>

<!ELEMENT GNF (#PCDATA)>
<!ATTLIST GNF
      uri CDATA #REQUIRED>

]>
```

Figure 5.5: DTD for PUMPSpec2

5.4 Protocol Implementation Classes

As discussed earlier in this chapter, the implementation classes are archived and the URI location where the archive is stored is added to the PUMPSpec. The archive contains the main implementation class, which has the following: input and output expressions, guard expression, and action procedures. The input and output expression evaluation functions have an input and an output vector as parameters. The guard expression receives a hash table of message vector from each incoming sync point. The action function simply carries out some processing (using a hash table of message vectors from all incoming sync points) and stores the output message(s) in a vector. For detailed information on implementation classes, readers are referred to [P.J00].

5.5 JXTA: A Peer-to-peer computing platform

[LiG01] [J.ORG] describes JXTA technology as a network programming and computing platform. It is aimed to alleviate the shortcomings of client-server distributed programming platforms such as unused bandwidth that were described in Chapter 1. Originally conceived by Sun Microsystems, Inc., JXTA is now a open-source and its development is contributed by a community of developers. Figure 5.6 shows a common layering structure of JXTA software architecture.

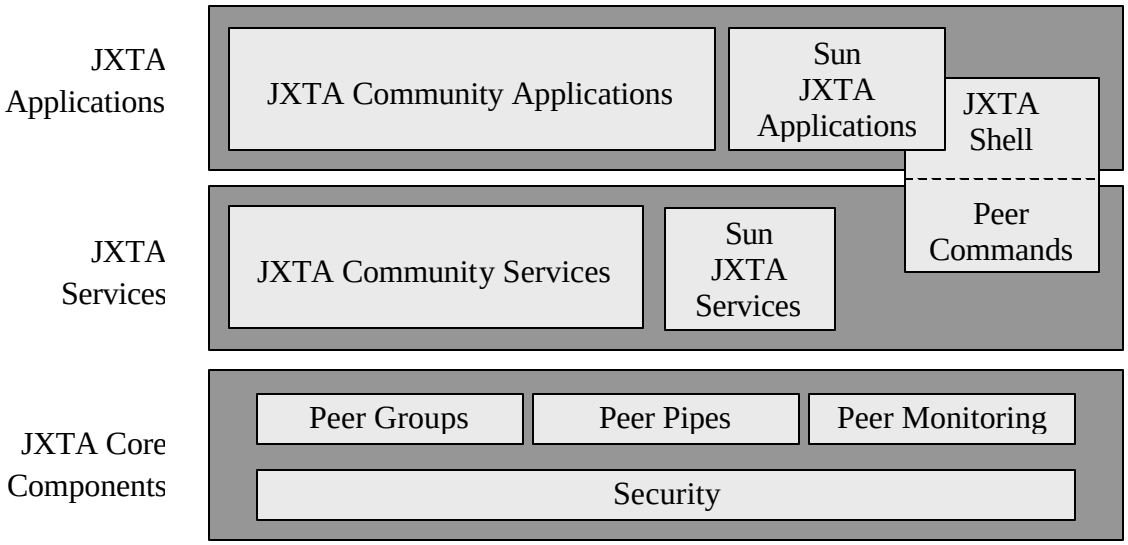

Figure 5.6: JXTA P2P Architecture

The typical P2P software stack is divided into three layers. At the bottom is the core layer that deals with peer establishment, communication management such as routing, and other low-level functions. The middle layer is a service layer that deals with services such as indexing, searching, and file sharing. This layer is typically included as components in an overall P2P system. The top layer consists of applications such as emailing, auctioning, and storage systems. JXTA implementation is designed to provide a framework in which services and applications can be built.

5.5.1 Concepts

At the highest abstraction level, JXTA technology is a set of protocols. Each protocol is defined by one or more messages exchanged among participants of the protocol. Each message has a pre-defined format, and may include various data fields. It is similar to TCP/IP in the sense that TCP/IP links nodes of the Internet together and is platform-independent (set of protocols), while JXTA technology connects peer nodes with each other and is also platform-independent. JXTA technology is transport independent and can utilize TCP/IP as well as other transport standards. JXTA technology defines a number of concepts including:

5.5.1.1 Identifiers

JXTA uses a 128-bit identifier to refer to an entity, be it a peer, an advertisement, a service, etc. It is guaranteed that each entity has a unique ID within a local runtime environment. It is securely bound to other information such as a name and a network address.

5.5.1.2 Advertisements

An advertisement is an XML structured document that names, describes, and publishes the existence of a resource, such as a peer, a peer group, a pipe, or a service. JXTA technology defines a basic set of advertisements. And, more advertisements can be formed from these basic types.

5.5.1.3 Peers

A peer is any entity that can speak the protocols required of a peer. Such a peer could manifest in the form of a processor, a process, a machine, or a user. Importantly, a peer may choose to implement only those protocols that it needs most (eg. Peer Discovery) and still function at a reduced level.

5.5.1.4 Messages

Messages are designed to be usable on top of asynchronous, unreliable, and unidirectional transport. A message is designed as a datagram, containing an envelope and a stack of protocol headers with bodies. The envelope contains a header, a message digest, the source endpoint, and the destination endpoint. An endpoint is a logical destination, given in the form of a URI, on any networking transport capable of sending and receiving datagram-style messages. Endpoints are typically mapped to physical addresses by a messaging layer. The protocol body within the message contains a variable number of bytes, and one or more credentials that is used to identify the sender.

5.5.1.5 Peer Groups

A peer group is a collection of cooperating peers that speak the set of peer group protocols and typically provide a common set of services. The relationship between a peer and a peer group can be somewhat meta-physical. JXTA specification does not limit the number of groups a peer can belong to. There is a special group, called the World Peer Group, which includes all JXTA peers. Participation in the World Peer Group is by default.

5.5.1.6 Pipes

Pipes are communication channels for sending and receiving messages, and are asynchronous. They are also unidirectional. Pipes are virtual; hence, a pipe's endpoint can be bound to one or more peer endpoints. A pipe is typically bound to a peer at runtime via the Pipe Binding Protocol and can moved around, bounding to different peers at different times. JXTA define kinds of message passing through pipe that are:

- A point-to-point pipe connects exactly two peer endpoints. The pipe is an output pipe to the sender and input pipe to the receiver, with traffic going in one direction only.

- A propagate pipe connects multiple peer endpoints together, from one output pipe to one or more input pipes. The result is that any message sent into the output pipe is sent to all input pipes.

5.5.2 Protocols

Project JXTA has defined the following six protocols. More protocols are currently under development by the developer community. A brief description of the protocols is given in this section. Interested readers are directed to [JPr02] for a detailed overview.

5.5.2.1 Peer Discovery Protocol

This protocol enables a peer to find advertisements of peers, peer groups, or any other resource on other peers. This protocol is the default discovery protocol for all peer groups, including the World Peer Group. Peer discovery can be done with or without specifying a name for either the peer to be located or the group to which peers belong. When no name is specified, all advertisements are returned.

5.5.2.2 Peer Resolver Protocol

This protocol enables a peer to send and receive generic queries to find or search for peers, peer groups, pipes, and other information. Typically, this protocol is implemented only by those peers that have access to data repositories and offer advanced search capabilities.

5.5.2.3 Peer Information Protocol

This protocol allows a peer to learn about other peers' capabilities and status. For example, one can send a ping message to see if a peer is alive. One can also query a peer's properties where each property has a name and a value string. It is mostly used for monitoring and presence services.

5.5.2.4 Peer Membership Protocol

This protocol allows a peer to obtain group membership requirements (such as an understanding of the necessary credential for a successful application to join the group), to apply for membership and receive a membership credential along with a full group advertisement, to update an existing membership or application credential, and finally, to cancel a membership or an application credential. Authenticators and security credentials are used to provide the desired level of protection.

5.5.2.5 Pipe Binding Protocol

This protocol allows a peer to bind a pipe advertisement to a pipe endpoint, thus indicating where messages actually go over the pipe. In some sense, a pipe can be viewed as an abstract, named message queue that supports a number of abstract operations such as create, open, close, delete, send, and receive. Bind occurs during the open operation, whereas unbind occurs during the close operation.

5.5.2.6 Peer Endpoint Protocol

This protocol allows a peer to ask a peer router for available routes for sending a message to a destination peer. Often, two communicating peers may not be directly connected to each other. Examples of this might include two peers that are not using the same network transport protocol, or peers separated by firewalls or NAT. NAT or Network Address Translation is a technique for translating one set of IP address, often private, to another set, often public. Peers across firewalls and NAT devices cannot see each other directly. Peer routers respond to queries with available route information, which is a list of gateways along the route. Any peer can decide to become a peer router by implementing the Peer Endpoint Protocol.

5.5.3 Java Reference Implementation

The current Project JXTA J2SE platform binding version 1.0 requires a platform that supports the Java Run-Time Environment (JRE) or Software Development Kit (SDK) 1.3.1 release or later. This environment is currently available on the Solaris Operating Environment, Microsoft Windows 95/98/2000/ME/NT 4.0, Linux, and the Macintosh. It needs to be mentioned that version 1.0 is just a starting point. The system has undergone lots of changes since its initial release and is still undergoing many refinements.

5.5.4 Services in JXTA

A service denotes a set of functions that a provider offers. A peer can offer a service by itself or in cooperation with other peers, as in a peer group. A service provider peer publicizes the service by publishing a service advertisement. Other peers can then discover this service and make use of it. Each service has a unique ID and name that consists of a canonical name string and a series of descriptive keywords that uniquely identifies the service. Sometimes, a service is well defined and widely available such that a peer can just use it. Other times, special code may be needed in order to actually access a service. For example, the way to interface with the service provider may be encoded in a piece of software. In this case, it is most convenient if a peer can locate an implementation that is suitable for the peer's specific runtime environment. Of course, if multiple implementations of the same service are available, then peers hosted on systems with Java runtime environments can use Java programming language implementations while native peers to use native code implementations. Service implementations can be pre-installed into a peer node or loaded from the network.

The process of finding, downloading, and installing a service from the network is similar to performing a search on the Internet for a web page, retrieving the content of the page, and then installing the required plug-in to work with the page. Once a service is installed and activated, pipes may be used to communicate with the service. We refer to a

service that executes only on a single peer as a peer service. We call a service that is composed of a collection of cooperating instances of the service running on multiple peers, a peer group service. The first JXTA implementation has built in a set of default peer group services such as peer discovery, as well as a set of configurable services such as routing.

5.6 The Protocol Management Service (PMS)

The PMS is the core of the whole protocol deployment and access process. It is analogous to the protocol server. In fact, it carries a protocol service engine, which loads and executes a protocol like a protocol server. When a peer enters a network, it may choose to search for a service by peer group or a protocol advertisement. As described earlier in this chapter, a protocol advertisement carries information about the protocol or service and its PUMPSpec location. It would then choose a peer on which it would like to run the protocol. As we noted a PUMPSpec may describe one or more sides of a communication like the "client" and "server". Hence, the peer would also need to select the role, so that the other interacting peer, henceforth called as the "buddy peer", might run that side of communication. When the peer is ready with all this information (i.e., the protocol advertisement, the buddy peer, and his role), it would make a call to the Protocol Management Service to initiate a new channel in form of a query. The Protocol Management Service then negotiates a channel with the buddy peer. On successful negotiation, PMS creates a bi-directional pipe and starts a protocol daemon on the buddy peer, which listens to that bi-directional pipe. It then responds to the initiating peer with a response, which contains the bi-directional pipe advertisement. It is then left upon the initiating peer to bind to the pipe and start the communication.

This method helps peers find and load service protocols at their will while preserving the generality of the system. However, there are many assumptions to this method. These are:

- the peer knows which other peer can act as a buddy peer and is responsible to check its availability.
- the role to be played by the buddy peer is known to the initiating peer (else the default role specified in PUMPSpec2 is chosen).
- the initiating peer knows to parse PUMPSpec2 for its role.

It is obvious that in many generic P2P systems, these assumptions may not hold, unless the user is an advanced user having prior knowledge on the working of PMS and JXTA concepts. In such situations, the developer of a service may choose to develop his own default client application and load it into a group with the same name as the service. Hence, a novice user (peer) would then search for an available service using peer group names and upon joining the peer group the default client will automatically perform the negotiation and start the communication. The client may also provide a user interface, if the peer/user needs to interact with the service, provided the peer platform supports the same.

CHAPTER 6

IMPLEMENTATION

6.1 Protocol Advertisement

The Protocol Advertisement will be responsible for describing the protocol information of a particular service or protocol. In order to uniquely identify a protocol, the Protocol Advertisement will need a name and a version. To represent the protocol information, the Protocol Advertisement uses the XML format shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ProtocolAdvertisement>
<ProtocolAdvertisement>
   <Name>...</Name>
   <Version>...</Version>
   <Desc>...</Desc>
   <SURL>...</SURL>
   <PUMPSpec>...</PUMPSpec>
</ProtocolAdvertisement>
```

The content of the Protocol Advertisement describes all of the elements related to a protocol on the P2P network, and includes

- *Name*: Name of the protocol and is a required field.
- *Version*: Protocol version is optional but is often necessary, as many versions of the same protocol may exist.
- *Desc*: Optional description of the protocol.

- *SURL*: is a required field and specifies the URL for PUMPSpec of the protocol.
- *PUMPSpec:* Optional inclusion of the protocol specification.

To implement the Protocol Advertisement we define an abstract class derived from the `net.jxta.document.Advertisement` class. This class defines basic accessors to set and retrieve the advertisement's various parameters. In addition, the class defines the static `getAdvertisementType` method to return the root element tag used by the Protocol Advertisement. `ProtocolAdvertisement` also defines the `getID` method, which is used by the Cache Manager to index the advertisement in the cache. The ID returned by `getID` should uniquely identify the advertisement. To avoid having to implement our own ID implementation, `ProtocolAdvertisement` returns `ID.nullID`. This null ID will prompt the Cache Manager to use a hash of the advertisement to index the advertisement in the cache, and is sufficient for our purposes.

The `Advertisement.getDocument` method is not defined by `ProtocolAdvertisement` to allow the implementation of `ProtocolAdvertisement` to define logic for parsing and formatting a Protocol Advertisement. This method is implemented by the `ProtocolAdv` subclass. In addition to providing a `getDocument` implementation, the `ProtocolAdv` class provides several constructors that provide advertisement-parsing functionality. All of the parsing and formatting functionality is built using the `net.jxta.document` classes to handle manipulating the XML object tree. In order for the protocol advertisement through the `AdvertisementFactory` like other basic advertisements, the implementation class must be registered with the `AdvertisementFactory` by calling the `registerAdvertisement` method. This call needs to be executed when the application starts, before any other class attempts to use the `AdvertisementFactory` to instantiate a `ProtocolAdvertisement`.

6.2 Protocol Management Service Definition

The protocol service abstracts the creation of channel initiation request and response messages and provides a simple interface that a developer can use to send these messages. It also provides a mechanism for developers to register and un-register listener objects that can be used to handle the requests and responses.

The task of determining if a channel request should be approved is left to the service developer. When the `ProtocolService` (implementation of the PMS) receives an initiate channel request message, it notifies each of the registered listeners instance's `channelRequested` method. It is then the responsibility of a listener to approve a request. When the `ProtocolService` receives an initiate channel response message, the registered listener instance's `channelApproved` method is notified, which then handles creation of a new protocol session. The following time-line diagram shows the request and response sequences between two peers. It is assumed that the initiating peer has already gotten the protocol advertisement.

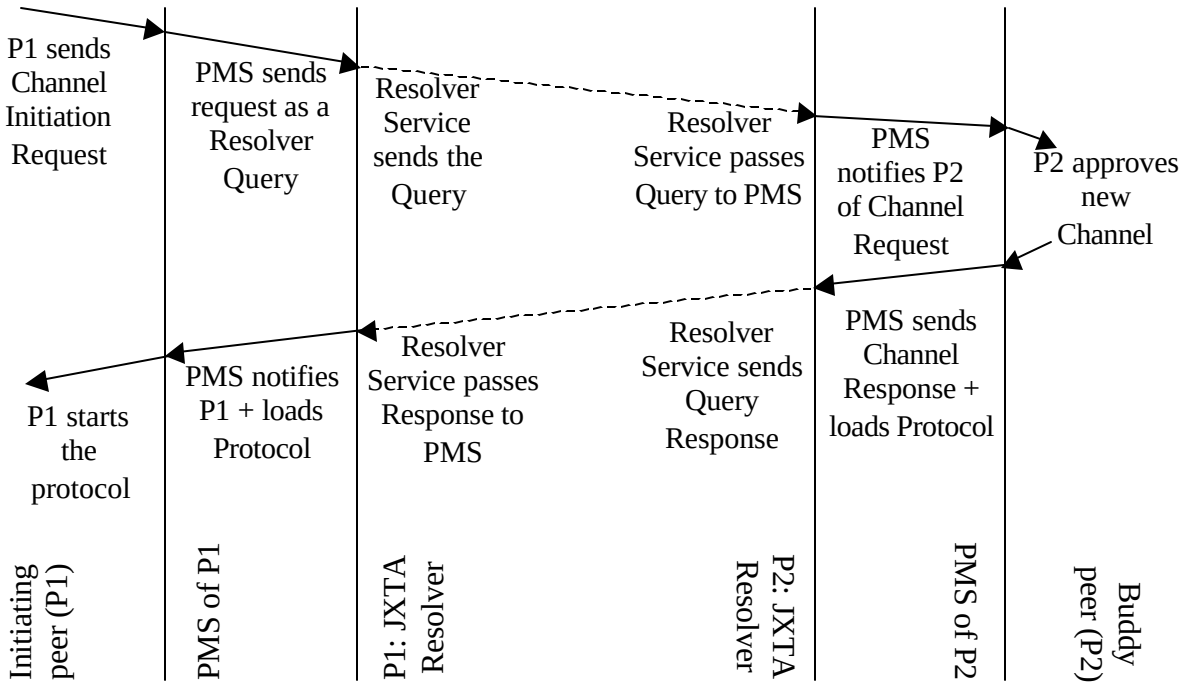

Figure 6.1: Channel Negotiation and Protocol Loading Sequence

As seen in Figure 6.1, the `ProtocolService` implementation does not directly deal with sending and receiving Initiate-Channel Request and Response messages; it uses the Resolver service instead. The Resolver service provides an implementation of the Peer Resolver Protocol (PRP), which defines how peers can exchange query and response messages. The Resolver service is responsible for wrapping a query string in a more generic message format, and sending it to a specific handler on a remote peer. On the remote peer, a Resolver service instance is responsible for taking an incoming message, passing it to the appropriate handler, and sending any response generated by the handler. Usually, a query is sent to known peers or is propagated via known rendezvous peers. When the query is propagated, any peer's Resolver service that receives a Resolver Query Message attempts to find a registered handler for the query. If a matching handler is found, the Resolver passes it the message, and then manages sending the response message generated by the handler back to the source peer.

The Protocol Service provides a simple interface that developers can use to send and receive Channel Request and Response messages through the following interfaces. It also provides a convenient way to register and remove listener objects as shown below:

```
//import needed packages
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.service.Service;
public interface PRunnerService extends Service
{
    // The module class ID for the PMS class of service.
    public static final String refModuleClassID =
       "urn:jxta:uuid-128E938121DD4957B74B90EE27FDC61F05";

    // Add a listener object to the service.
    public void addListener(PRunnerListener listener);
```

```
        // Approve a protocol session.
        public void approveChannel(PipeAdvertisement
                    pipeAdvertisement, String displayName,
                    int queryID);

        // Remove active listener
        public boolean removeListener(PrunnerListener
                                      listener);

        //Send a channel request to the peer specified.
        public void requestChannel(String peerID,
                   ProtocolAdvertisement protocolAdvertisement,
                   String displayName, String Role,
                   PRunnerListener listener);

    }
```

In addition to channel negotiation and protocol loading, the PMS also provides the following features:

6.2.1 Protocol Event Notification Service

The user of a protocol (the initiating peer or buddy peer), on loading the protocol, will be able to specify markings upon which a protocol event may be triggered by the PMS. A protocol event returns the actual messages in the marking that caused the event. This allows users to be notified when certain markings are reached. For example, one may choose to trigger a protocol event when a message of name “SERVER” and type “RESPONSE” arrives at a SYNCPOINT named “INPUT_BUFFER”, symbolizing the arrival of a server response to a client query in the input buffer.

6.2.2 Loading User-Provided Implementation Classes

Upon successful negotiation of a channel (bi-directional pipe) , the users (both the initiating peer and the buddy peer) may choose to provide their own implementation classes for the protocol. If they fail to do so, the implementation classes located at the URI specified in the protocol specification are used. This is the default implementation provided by the protocol creator and often user need refinements due to optimization concerns or hardware requirements.

6.2.3 Ability to add and remove messages at SYNCPOINTs

As discussed in earlier chapter, the PUMPSpec2 provides a means to specify those SYNCPOINTs on which a user may add or remove messages safely. Such messages usually comprise of user data such as login information, or server responses like downtime alerts. Such messages are often kept in SYNCPOINTs that act as buffers for user to put or remove that needs to be communicated or have been a result of communication.

6.3 Channel Request Message

Before being able to communicate using the protocol with a remote peer, a peer will need to request a Pipe Advertisement that it can use to establish the protocol session with the remote peer. The Initiate-Channel-Request Message shown below is used for this purpose.

```
<?xml version="1.0" encoding="UTF-8"?>
<InitiatePRunnerRequest>
<Name> . . . </Name>
<Role> . . . </Role>
<ProtocolAdvertisement> . . . </ProtocolAdvertisement>
</InitiatePRunnerRequest>
```

The fields in this message are:

- *Name*: Name of the protocol and though optional, it is typically used for clarity.
- *Role*: A recommended field indicating the role the remote peer must play during the protocol session. If missing, the default role in protocol specification is used.
- *ProtocolAdvertisement*: The protocol advertisement itself and is a required field. We use the entire advertisement since it saves some time of the remote peer, which may be otherwise wasted searching for the advertisement.

When a peer receives an Initiate Channel Request Message, the peer can extract the name of the protocol and the role and all the needed protocol information (if needed, determine whether it wishes to start the session). An Initiate Channel Response Message will be returned containing a Pipe Advertisement that can be used by the requesting peer to establish the protocol session.

6.4 Channel Response Message

To allow a requesting peer to establish a protocol session, a peer needs to generate a Pipe Advertisement and send it as part of an Initiate Channel Response Message to the requesting peer. The XML for the Initiate Channel Response Message is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<InitiatePRunnerResponse>
<Name> . . . </Name>
<PipeAdvertisement> . . . </PipeAdvertisement>
</InitiatePRunnerResponse>
```

The fields in this message are:

- *Name*: An optional element containing the protocol name.

- *jxta:PipeAdvertisement*: A required element that contains the Pipe Advertisement to use to establish the protocol session. Note that this element is actually the root of the Pipe Advertisement XML tree.

Once a peer receives an Initiate Channel Response Message, they can proceed to use the Pipe Advertisement with the `BidirectionalPipeService` class to establish two-way communication and begin the session.

CHAPTER 7

EXAMPLES

7.1 LOGIN: A simple login protocol

The LOGIN protocol is a simple stateful protocol designed to demonstrate the working of protocol management in a P2P infrastructure. The server side starts by receiving the username and then the password. Using the username and password stored, it authenticates the user and based upon the result of such an authentication, an OK message or ERROR message is sent, if the authentication succeeded or failed respectively. Figure 7.1 shows the CP-Net representation of the protocol model.

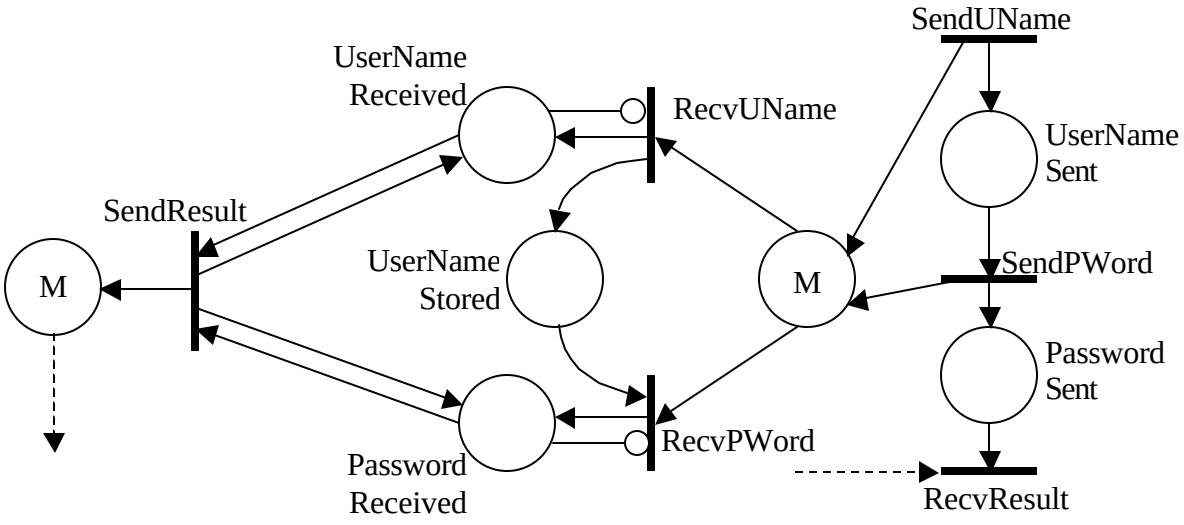

Figure 7.1: CPN-Based Protocol Model for LOGIN

Here, inhibitor arcs are used to block the receiving stations, once they have received the need data. This is one more use of an inhibitor arc i.e., zero testing. Following is the

PUMPSpec2 specification for the same and the implementation classes are attached to appendix A.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE ProtocolSpecification SYSTEM
         "http://www.cs.uga.edu/~kannan/data/pspec.dtd">
<ProtocolSpecification name="LOGIN" version="1.0"
         defaultrole="server">

  <ROLE name="server" type="PSTATION" source="RECVLogin"
            sink="SENDLoginResult">
    <!-- Specify synchronization points -->
    <SYNCPOINT name="BlockReceive"/>
    <SYNCPOINT name="LoginReceived"/>
    <SYNCPOINT name="SendLoginResult"/>

    <!-- Specify processing stations -->
    <PSTATION name="RECVLogin">
      <INHIBITOR syncpoint="BlockReceive"/>
      <CONNECTOR kind="output" syncpoint="LoginReceived"
                expression="any"/>
   </PSTATION>
   <PSTATION name="Authenticate" action="aAuthenticate"
             guard="gAuthenticate">
      <CONNECTOR kind="input" syncpoint="LoginReceived"
            expression="any"/>
      <CONNECTOR kind="output" syncpoint="SendLoginResult"
            expression="any"/>
   </PSTATION>
   <PSTATION name="SENDLoginResult">
      <CONNECTOR kind="input" syncpoint="SendLoginResult"
           expression="any"/>
     <CONNECTOR kind="output" syncpoint="BlockReceive"
            expression="any"/>
```

```
 </PSTATION>

  <!-- The jar file with guards and functions -->
  <GNF uri=
    "http://www.cs.uga.edu/~kannan/data/LOGIN.jar"/>
</ROLE>

<ROLE name="client" type="SYNCPOINT"
      source="ClientLoginInfo" sink="LoginInfoReceived">
  <SYNCPOINT name="ClientLoginInfo">
  <!--A message of the following type should be
      generated and placed at this SYNCPOINT by the
   client program-->
  </SYNCPOINT>
  <SYNCPOINT name="ClientLoginInfoSent"/>
  <SYNCPOINT name="LoginResultReceived" />
  <!-- Specify processing stations -->
  <PSTATION name="SENDLoginInfo">
      <CONNECTOR kind="input" syncpoint="ClientLoginInfo"
            expression="any"/>
     <CONNECTOR kind="output"
            syncpoint="ClientLoginInfoSent"
            expression="any"/>
  </PSTATION>
  <PSTATION name="RECVLoginResult">
     <CONNECTOR kind="input"
             syncpoint="ClientLoginInfoSent"
             expression="any"/>
     <CONNECTOR kind="output"
             syncpoint="LoginResultReceived"
             expression="any" />
  </PSTATION>
     <!-- The jar file with guards and functions -->
   <GNF uri=
      "http://www.cs.uga.edu/~kannan/data/LOGIN.jar"/>
```

```
    </ROLE>
</ProtocolSpecification>
```

Note that the synchronization points of type "medium" (marked as M in Figure 7.1) are replaced by a buffer followed by a sending pstation and a receiving pstation followed by a buffer in sides that treat the synchronization points as output and input buffers, respectively.

7.2 Internet Message Access Protocol (IMAP4rev1: Abridged)

In this section, we present the Internet Message Access Protocol [RFC2060], which is fast becoming popular with email clients. IMAP4rev1 allows a client to access and manipulate electronic mail messages on a server. IMAP4rev1 permits manipulation of remote message folders, called "mailboxes", in a way that is functionally equivalent to local mailboxes. IMAP4rev1 also provides the capability for an offline client to resynchronize with the server.

IMAP4rev1 includes many operations but we will deal with only a few of them to keep the protocol interesting and readable. Our IMAP4rev1 (Abridged) will allow the following commands: logging in, checking for new messages, selecting and examining of mailboxes, and closing the connection. Messages in IMAP4rev1 are accessed by the use of numbers. Following are additional information about the abridged version. These numbers are either message sequence numbers or some unique identifiers. IMAP4rev1 supports a single server. All interactions transmitted by client and server is in the form of lines. The client command begins an operation. We assume that a client always sends its command in full, which may not be always true in the original version. Hence, a server reads a command line from the client, parses the command and its arguments, and transmits server data and a server command completion result response. We also assume that clients send a single command at a time, hence we will not be dealing with handling

of multiple commands. Messages tagged with "*" are termed as untagged messages and represent status information from an unfinished command. Figure 7.2 shows the CP-Net model for IMAPv4rev1 (Abridged).

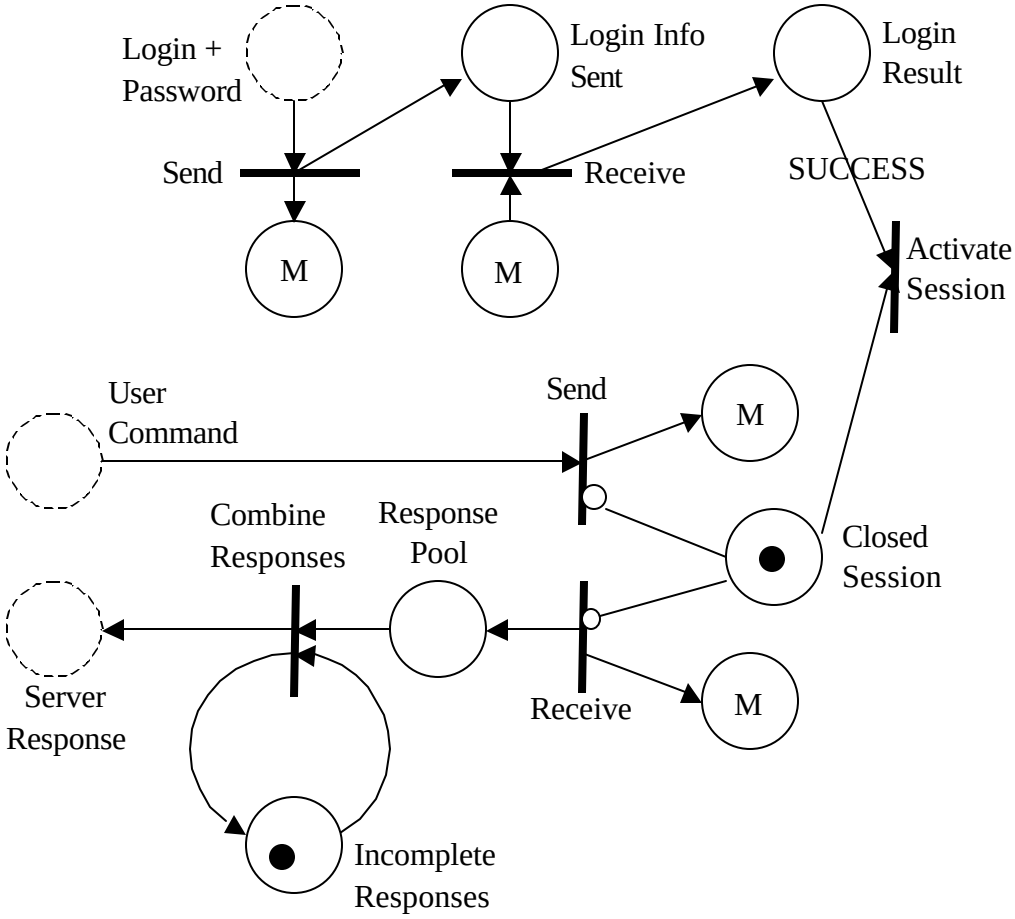

Figure 7.2: CPN-Based Client-side Protocol Model for IMAP

Figure 7.2, represents the client side of the communication, the SYNCPOINTs in dotted lines represent places in which clients can add or remove messages. A typical client first starts by placing the user name and password information, the information is sent while enabling the receiving of the result. On a successful login, the session is activated. The client may simultaneously send user command and receive server responses. The sending of user command is simple and straightforward. When a server response is

received, it is first checked to see if it is an untagged response. If it is the message is concatenated with previous untagged responses and placed temporarily in “Incomplete Responses”. Upon receiving a tagged response, the incomplete responses are taken and put in “Server Responses” synchronization point, from which user can remove the message.

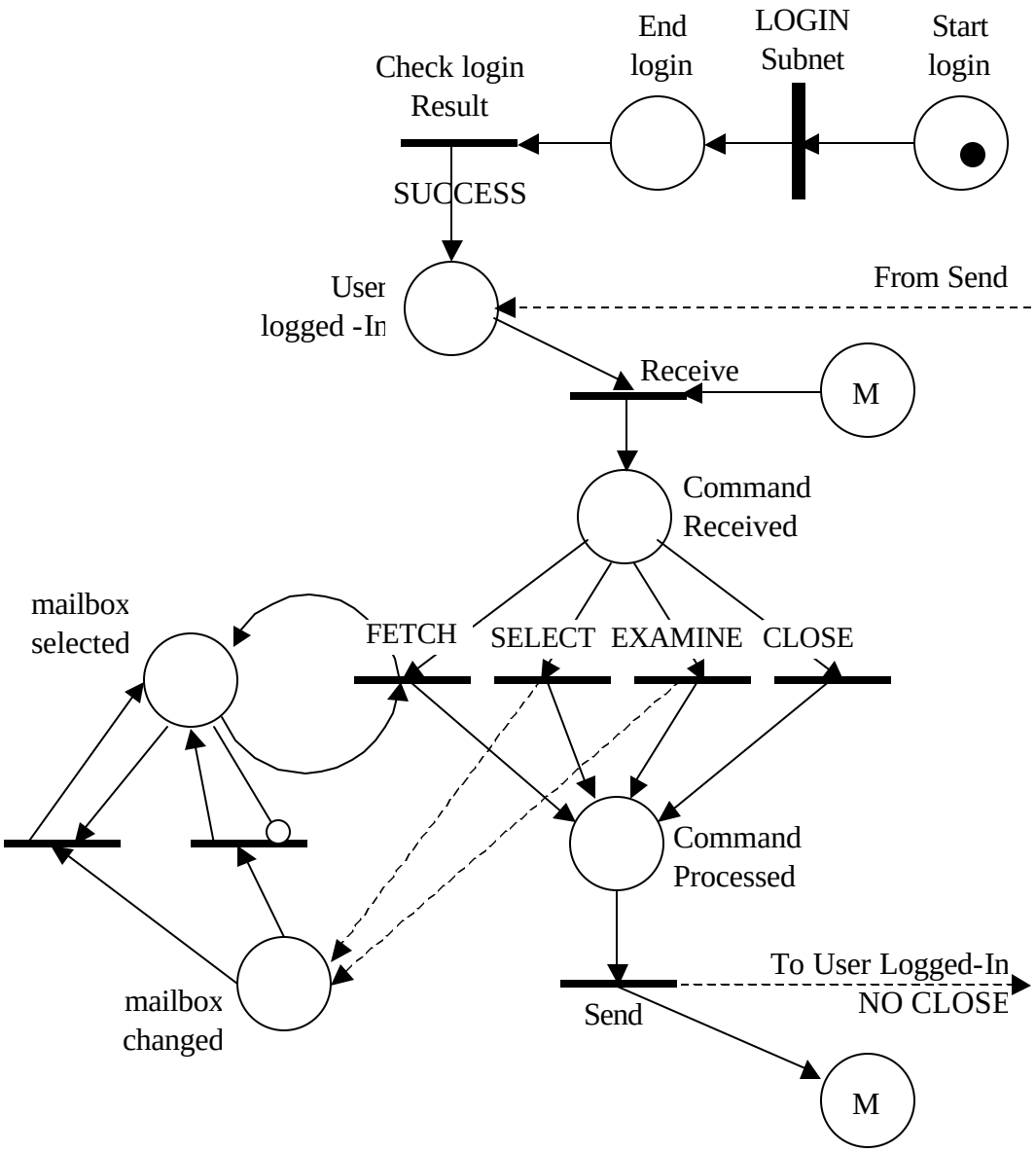

Figure 7.3: CPN-Based Server-side Protocol Model for IMAP

In Figure 7.2, we can see that we use the PUMPSpec for LOGIN protocol described earlier as a subnet represent the PSTATION “LOGIN Subnet”. Upon successful user login, the server is ready to receive user command. Upon receiving a user command, the server chooses one of the four PSTATIONS that would process the command and the processed result is sent back to the user. If the most recently processed command is not a CLOSE command, then a token is put back onto the “User Logged-In’ SYCNPOINT allowing the server to receive further commands. The SELECT command lets the user to select a mailbox and the mailbox name is retained for mailbox-commands such as FETCH. In addition, the EXAMINE used in examining mailboxes other than what is currently selected leads to deselecting of any previously selected mailbox. The conditions are handled as shown in Figure7.2. Following is the PUMPSpec2 specification for the protocol model shown above and the corresponding implementation classes in appendix B.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE ProtocolSpecification SYSTEM
"http://www.cs.uga.edu/~kannan/data/pspec.dtd">

<ProtocolSpecification name="IMAP" version="1.0"
 defaultrole="server">

    <ROLE name="server" type="PSTATION"
     source="RECVCommand" sink="SENDCommand">

     <!-- Specify synchronization points -->
      <SYNCPOINT name="StartLogin">
          <MESSAGE name="SYNC" type="OK"/>
      </SYNCPOINT>
      <SYNCPOINT name="EndLogin"/>
      <SYNCPOINT name="UserLoggedIn"/>
      <SYNCPOINT name="UserCommandReceived"/>
      <SYNCPOINT name="UserCommandProcessed"/>
```

```
<SYNCPOINT name="MBoxCommandProcessed"/>
<SYNCPOINT name="MBoxCurrentlySelected"/>

<!-- Specify processing stations -->
<PSTATION name="LoginSubNet"
      subnet=
      "http://www.cs.uga.edu/~kannan/data/LOGIN.xml">
      <CONNECTOR kind="input" syncpoint="StartLogin"
            expression="any"/>
      <CONNECTOR kind="output" syncpoint="EndLogin"
            expression="any"/>
</PSTATION>

<PSTATION name="LoginCheck" action="aLoginCheck"
      guard="noGuard">
      <CONNECTOR kind="input" syncpoint="EndLogin"
            expression="any"/>
      <CONNECTOR kind="output"
            syncpoint="UserLoggedIn"
            expression="OnNoError"/>
</PSTATION>

<PSTATION name="LoginCheck" action="aLoginCheck"
      guard="noGuard">
      <CONNECTOR kind="input" syncpoint="EndLogin"
            expression="any"/>
      <CONNECTOR kind="output"
            syncpoint="UserLoggedIn"
            expression="OnNoError"/>
</PSTATION>

<PSTATION name="RECVUserCommand">
      <CONNECTOR kind="input"
            syncpoint="UserLoggedIn"
            expression="any"/>
```

```
      <CONNECTOR kind="output"
            syncpoint="UserCommandReceived"
            expression="any"/>
</PSTATION>

<PSTATION name="ProcessClose" action="aProcessClose"
      guard="gProcessClose">
<CONNECTOR kind="input"
      syncpoint="UserCommandReceived"
      expression="any"/>
<CONNECTOR kind="output"
      syncpoint="UserCommandProcessed"
      expression="any"/>
</PSTATION>

<PSTATION name="ProcessSelect"
      action="aProcessSelect" guard="gProcessSelect">
      <CONNECTOR kind="input"
            syncpoint="UserCommandReceived"
            expression="any"/>
      <CONNECTOR kind="output"
            syncpoint="UserCommandProcessed"
            expression="any"/>
      <CONNECTOR kind="output"
            syncpoint="MBoxCommandProcessed"
            expression="OnResponseNotBad"/>
</PSTATION>

<PSTATION name="ProcessExamine"
      action="aProcessExamine"
      guard="gProcessExamine">
      <CONNECTOR kind="input"
            syncpoint="UserCommandReceived"
            expression="any"/>
      <CONNECTOR kind="output"
```

```
            syncpoint="UserCommandProcessed"
            expression="any"/>
      <CONNECTOR kind="output"
            syncpoint="MBoxCommandProcessed"
            expression="OnResponseNotBad"/>
</PSTATION>

<PSTATION name="ProcessFetchWithMBox"
      action="aProcessFetchWithMBox"
      guard="gProcessFetch">
      <CONNECTOR kind="input"
            syncpoint="MBoxCurrentlySelected"
            expression="any"/>

      <CONNECTOR kind="input"
            syncpoint="UserCommandReceived"
            expression="any"/>
      <CONNECTOR kind="output"
            syncpoint="UserCommandProcessed"
            expression="OfTypeServer"/>
      <CONNECTOR kind="output"
            syncpoint="MBoxCurrentlySelected"
            expression="OfTypeMBox"/>
</PSTATION>

<PSTATION name="ProcessFetchWithNoMBox"
      action="aProcessFetchWithNoMBox"
      guard="gProcessFetch">
      <INHIBITOR syncpoint="MBoxCurrentlySelected"/>
      <CONNECTOR kind="input"
            syncpoint="UserCommandReceived"
            expression="any"/>
      <CONNECTOR kind="output"
            syncpoint="UserCommandProcessed"
            expression="OfTypeServer"/>
```

```
</PSTATION>

<PSTATION name="SENDServerResponse">
      <CONNECTOR kind="input"
            syncpoint="UserCommandProcessed"
            expression="any"/>
      <CONNECTOR kind="output"
            syncpoint="UserLoggedIn"
            expression="OnNoClose"/>
</PSTATION>

<PSTATION name="ChooseNewMBox"
      action="aChooseNewMBox" guard="noGuard">
      <INHIBITOR syncpoint="MBoxCurrentlySelected"/>
      <CONNECTOR kind="input"
            syncpoint="MBoxCommandProcessed"
            expression="any"/>
      <CONNECTOR kind="output"
            syncpoint="MBoxCurrentlySelected"
            expression="any"/>
</PSTATION>

<PSTATION name="ReplaceOldMBox"
      action="aReplaceOldMBox" guard="noGuard">
      <CONNECTOR kind="input"
            syncpoint="MBoxCurrentlySelected"
            expression="any"/>
      <CONNECTOR kind="input"
            syncpoint="MBoxCommandProcessed"
            expression="any"/>
      <CONNECTOR kind="output"
            syncpoint="MBoxCurrentlySelected"
            expression="any"/>
</PSTATION>
```

```
        <!-- The jar file with guards and functions -->
        <GNF uri=
        "http://www.cs.uga.edu/~kannan/data/IMAP.jar"/>
</ROLE>

<ROLE name="client" type="SYNCPOINT"
        source="ClientLoginInfo" sink="LoginInfoReceived">
        <SYNCPOINT name="ClientLoginInfo"/>
        <SYNCPOINT name="ClientLoginInfoSent"/>
        <SYNCPOINT name="LoginResultReceived" />
        <SYNCPOINT name="UserCommand"/>
        <SYNCPOINT name="ConnectionClosed"/>
        <SYNCPOINT name="ResponsePool"/>
        <SYNCPOINT name="TempResponsePool">
                <MESSAGE name="SYNC" type="OK"/>
        </SYNCPOINT>
        <SYNCPOINT name="ServerResponse"/>

        <PSTATION name="SENDLoginInfo">
                <CONNECTOR kind="input"
                        syncpoint="ClientLoginInfo"
                        expression="any"/>
                <CONNECTOR kind="output"
                        syncpoint="ClientLoginInfoSent"
                        expression="any"/>
        </PSTATION>

        <PSTATION name="RECVLoginResult">
                <CONNECTOR kind="input"
                        syncpoint="ClientLoginInfoSent"
                        expression="any"/>
                <CONNECTOR kind="output"
                        syncpoint="LoginResultReceived"
                        expression="any" />
        </PSTATION>
```

```
<PSTATION name="ActivateSession"
      action="aActivateSession" guard="noGuard">
<CONNECTOR kind="input"
            syncpoint="LoginResultReceived"
            expression="OnNoError"/>
      <CONNECTOR kind="input"
            syncpoint="ConnectionClosed"
            expression="any"/>
</PSTATION>

<PSTATION name="SENDUserCommand">
      <INHIBITOR syncpoint="ConnectionClosed"/>
      <CONNECTOR kind="input" syncpoint="UserCommand"
            expression="OnlyValidCommands"/>
</PSTATION>

<PSTATION name="RECVServerResponse">
      <INHIBITOR syncpoint="ConnectionClosed"/>
      <INHIBITOR syncpoint="ResponsePool"/>
      <CONNECTOR kind="output"
            syncpoint="ResponsePool" expression="any"
      />
      <CONNECTOR kind="output"
            syncpoint="ConnectionClosed"
            expression="OnCloseCommand" />
 </PSTATION>

<PSTATION name="PassTheResponse"
      action="aPassTheResponse" guard="noGuard">
      <CONNECTOR kind="input"
            syncpoint="ResponsePool" expression="any"
      />
      <CONNECTOR kind="input"
            syncpoint="TempResponsePool"
            expression="any" />
```

```
            <CONNECTOR kind="output"
                  syncpoint="TempResponsePool"
                  expression="OnlyContinuingResponses" />
            <CONNECTOR kind="output"
                  syncpoint="ServerResponse"
                  expression="OnlyFinalResponses" />
      </PSTATION>

      <!-- The jar file with guards and functions -->
      <GNF uri=
      "http://www.cs.uga.edu/~kannan/data/IMAP.jar"/>
</ROLE>
</ProtocolSpecification>
```

7.3 Distributed File Search Protocol (GNUTELLite)

In this section, we discuss the specification and implementation of a distributed file search protocol. The protocol is derived from the popular distributed file sharing protocol for peer-to-peer systems called Gnutella [GNU]. GNUTELLite deals with the distributed file search aspect s of the parent protocol. A Gnutella network consists of numerous numbers of peers, which are equal in functionality, called servants. There is no concept of centralized servers. Each peer functions as both client and server. This makes the system more tolerance to shutdown and regulation.

In GNUTELLite, when a servant queries for a particular filename, a new message is generated and broadcasted to all other servants that are directly connected to itself. And when a servant receives a message that is not a response to one of its own queries, the servant looks up a hash table to messages to see if it has any routing information for the message. All messages have an Message ID that is kept unique within a servant. If routing information is found, then the message is sent directly else it is re-broadcasted

through all of the servant's connections. Figure 7.3 shows the Petri net representation of the protocol described above.

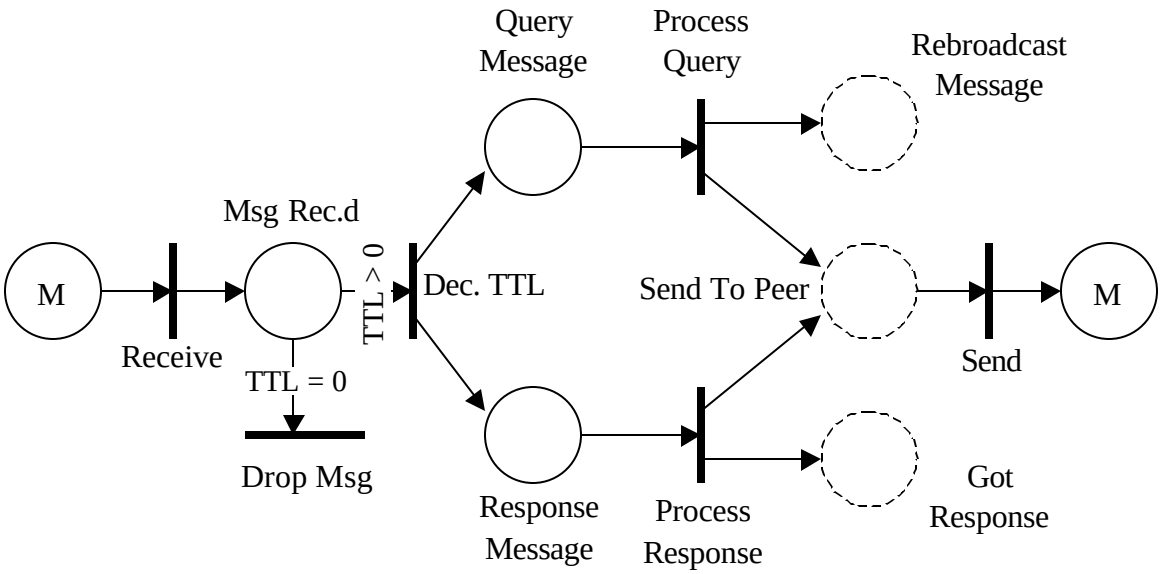

Figure 7.4: CPN-Based Protocol Model for GNUTELLite

The lack of hierarchy in such a P2P system means the network cannot be easily described, and is usually cyclic. In order to avoid cyclic shuffling of messages indefinitely, each packet has a numeric time-to-live (TTL) value. A peer must decrement this value before passing on the message to another peer. Any peer that receives a message with TTL value set to 0, simply drops the message. This protocol specification is encoded into PUMPSpec2 shown below. Note that peers need to play the role of servant only.

```
<?xml version="1.0" standalone="no"?>

<!DOCTYPE ProtocolSpecification SYSTEM
"http://www.cs.uga.edu/~kannan/data/pspec.dtd">
<ProtocolSpecification name="GNUTELLite" version="1.0"
defaultrole="servent">
<ROLE name="servent">
    <!-- Specify synchronization points -->
    <SYNCPOINT name="UserQuery"/>
    <SYNCPOINT name="MessageReceived"/>
    <SYNCPOINT name="QueryMessage"/>
    <SYNCPOINT name="ResponseMessage"/>
    <SYNCPOINT name="PeerIDLookup"/>
    <SYNCPOINT name="RebroadcastMessage"/>
    <SYNCPOINT name="SentToPeer"/>
    <SYNCPOINT name="UserResponse"/>
    <!-- Specify processing stations -->
    <PSTATION name="SENDUserQuery">
    <CONNECTOR kind="input" syncpoint="UserQuery"
        expression="any"/>
    </PSTATION>
    <PSTATION name="RECVMessage">
        <CONNECTOR kind="output"
            syncpoint="MessageReceived" expression="any"/>
    </PSTATION>
    <PSTATION name="DropMessage" action="aDropMessage"
        guard="noGuard">
        <CONNECTOR kind="input" syncpoint="MessageReceived"
             expression="InvalidTTL"/>
    </PSTATION>
    <PSTATION name="DecrementTTL" action="aDecrementTTL"
        guard="noGuard">
        <CONNECTOR kind="input" syncpoint="MessageReceived"
            expression="IsValidTTL"/>
        <CONNECTOR kind="output" syncpoint="QueryMessage"
```

```
            expression="IsQuery"/>
        <CONNECTOR kind="output"
            syncpoint="ResponseMessage"
            expression="IsResponse"/>
    </PSTATION>
    <PSTATION name="ProcessQuery" action="aProcessQuery"
        guard="noGuard">
        <CONNECTOR kind="input" syncpoint="QueryMessage"
            expression="any"/>
        <CONNECTOR kind="output" syncpoint="UserQuery"
            expression="IsResponse"/>
        <CONNECTOR kind="output"
            syncpoint="RebroadcastMessage"
            expression="IsQuery"/>
    </PSTATION>
    <PSTATION name="ProcessResponse"
        action="aProcessResponse" guard="noGuard">
        <CONNECTOR kind="input" syncpoint="ResponseMessage"
            expression="any"/>
        <CONNECTOR kind="output" syncpoint="UserResponse"
            expression="IsMyMessage"/>
        <CONNECTOR kind="output" syncpoint="SendToPeer"
            expression="PeerMessage"/>
    </PSTATION>
    <!-- The jar file with guards and functions -->
    <GNF uri=
    "http://www.cs.uga.edu/~kannan/data/GNUTELLite.jar"/>
</ROLE>
</ProtocolSpecification>
```

CHAPTER 8

CONCLUSION AND FUTURE WORK

8.1 Conclusion

The main motivation behind this work is the ability of any computing device to use an available service without much (or absolutely no) prior knowledge about the service. A key issue in such a system is to be able to specify the communication protocol in an abstract manner. There also needs to be a method for clients of the service to dynamically load the protocol so that they may communicate with the service. We have noted that the growth of Internet and distributed computing paradigm has made peer-to-peer computing popular. However, the growth of peer-to-peer computing is somewhat hindered due to the lack of the above described system and the lack of compatibility between the peer-to-peer systems themselves. With systems such as the one described above, service developer may confidently develop services that can be used by any peer without the trouble of building altogether a new peer-to-peer network. In our work, we have shown that such a system is viable. We have developed a framework for JXTA, a community-owned generic peer-to-peer system, which uses Petri nets for protocol specification and a Protocol Management Service that enables peers within JXTA to dynamically load protocols and communicate with a buddy peer. The peers can load the protocol both locally as well as remotely and start a communication channel using this service. We have also enhanced the PUMPS specification technique PUMPSpec to PUMPSpec2, by addition of powerful constructs like inhibitor arc, hierarchical modeling capability, and run-time management of tokens.

8.2 Future Work

JXTA is one of the first generic peer-to-peer communication systems. It is open and community-owned. It is a new project and is still in the research and development stage. The use of JXTA has given the scope of improving our work in terms better service management capabilities and like, as JXTA becomes more powerful (i.e., more and better protocols get added). As discussed in section 5.6, in developing the framework for our project we have made certain assumptions about a peer, which initiates a protocol session. Specifically, the initiating peer is assumed to have prior knowledge of the buddy peer that it wants to have a protocol session with, complementing its role. It would be better if the peer could get some help in deciding this (and other aspects like role), through a service or a set of pre-define policies known to the PMS.

Suitable GUI tools may also be developed for the design and verification of Petri nets associated with PUMPSpec2. Such tools could also have the option of automatically setting up PUMPSpec2 files in XML and publishing them within the P2P system along with the protocol implementation. In addition, to improve the performance of our framework, tools and techniques for compilation the protocol specification and implementation may be developed. Currently, the PMS loads the protocol specification and interprets the same. In section 5.3.2, when we described the enhancements made to PUMPSpec, we discussed the run-time management of tokens with the protocol specification. We have currently restricted the capability adding and removing tokens only to places define with “user” permission. As a future work, one might develop the concept of a protocol administrator or a protocol owner, who has the privilege of manipulating the tokens at runtime, thus giving a regulatory control over the protocols.

In section 7.3, while specifying the protocol for distributed file search, we develop a test program that was responsible for managing the connections with other peers. In order to alleviate the peers of the burden of connection management, one could develop support for the same within the protocol specification. This feature would prove to be valuable to

many peer-to-peer applications, which are usually connected to more than one peer at any given time. Finally, an extended study of the security considerations of such a system should be done. This would give a great deal of information on the right security policies and procedures to be followed while using this system. Currently, JXTA provides the means to tag the resources with credentials. These credentials can be attached to protocol advertisements, session-initiating peers, communication pipes, and peer groups to ensure a secure execution of protocols between peers.

REFERENCES

[A.B97] Albert Banchs et. al., *Multicasting Multimedia Streams with Active Networks,* ICSI Technical Report 97-050, 1997

[AgT74] Agerwala T., *A complete model for representing the coordination of asynchronous processes,* Technical report 32, John Hopkins Univ., Baltimore, Md., July 1974

[AgT79] Agerwala T., *Putting Petri Nets to Work.* Computer, Vol. 12(12), pp. 85-94, 1979

[ASH99] Ashish Swarup, *ALPS: Application Level Protocol Specification Language and its Implementation*, Thesis Report, December 1999

[Basu97] Anindya Basu, Mark Hayden, Greg Morrisett, Eicken T., *A Language-Based Approach to Protocol Construction*, ACM-SIGPLAN, Workshop on Domain Specific Languages, January 1997

[Basu98] Anindya Basu, Greg Morrisett, Eicken T., *Promela++: A Language for Constructing Correct and Efficient Protocols*, INFOCOM, pages 445-462, 1998

[Bel89] F. Belina, D. Hogrefe, *The CCITT Specification and Description Language SDL*, Computer Networks and ISDN Systems, Vol 16, pp. 311-314, 1989

[Bev98] Beverly S., Zhou W., Jackson A.W., *Smart Packets for Active Networks,* BBN Technologies, Jan 1998.

[BOL87] Bolognesi B., Brinksma E., *Introduction to the ISO Specification Language LOTOS,* Computer Networks and ISDN Systems, Vol.14, pp. 25-29, 1987

[Clay95] Clayton R., Calvert K., *A Data-Stream Language for Protocols,* Report, Georgia Institute of Technology, Georgia, October 1995

[Cost99] Cost R.S., Ye Chen, Finn T. Labrou Y., Peng Y., *Using Colored Petri Nets for Conversation Modeling*, Working notes, IJCAI'99, Sweden, August 1999

[CVR89] C. V. Ravishankar, R. Finkel, *Linguistic Support for Dataflow*, Technical Report CSE-Tr-14-89, The University of Michigan, Ann Arbor, Michigan, 1989

[DaT77] Danthine A., *Petri nets for protocol modeling and verification,* COMNET Budapest, Working Papers Vol. II, pp. 663-685, 1977.

[DaT80] Danthine A., *Protocol Representation with finite state models,* IEEE Transactions on communications, COM-28 (4), pp. 632-643, April 1980.

[D.D98] Decasper D., Plattner B., *DAN: Distributed Code Cashing for Active Networks,* IEEE INFOCOM, April 1998

[DAN02] Dan Brookshier, Dareen, Navaneeth , *JXTA: Java P2P Programming*, Chapter 1, Sams Publication, March 2002

[DHa88] D. Harel, *On Visual Formalisms,* Communications of the ACM, Vol.31, pp. 514-530, 1988

[DIAZ84] Diaz M., Azema P., *Petri net based models for the specification and validation of protocols,* SIG-Petri Net and Related System Models, Newsletter No. 17, pp. 21-39, June 1984

[DJW98] D.J. Wetherall, J.V. Guttag, D.L.Tennenhouse, *ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols,* IEEE OPENARCH'98, April 1998

[EM85] Ehrig H., Mahr B., *Fundamentals of Algebraic Specification 1*, Springer-Verlag, Berlin 1985.

[GAR96] Garrett Burke, *KANGA: A framework for building application specific communication protocols*, Thesis, Trinity College, Dublin, Sept 1996

[GCN97] Necula G.C., *Proof-Carrying Code,* 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, 1997

[Gel85] D. Gelernter, *Generative Communication in Linda,* ACM Transactions on Programming Languages and Systems, Vol. 7, No. 1, pp. 80-112, 1985

[GNU] *The Gnutella Protocol,* http://www.gnutelladev.com

[H.V98] Holvoet T. and Verbaeten P., *Using Petri Nets for Specifying Active Objects and Generative Communications*, K.U.Leuven, Belgium, 1998

[HAW70] Holt A.W., Commoner F., *Events and condition,* Applied Data Research N.Y., 1970

[HOA85] Hoare C.R., *Communicating Sequential Processes*, Prentice-Hall, 1985

[HOL91] G.J Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, N, 1991

[HUA93] Yen-Min Huang, Chinya Ravishankar, *Cicero: A Protocol Construction Language*, Technical Report CSE-TR-171-93, University of Michigan, Ann Arbor, Michigan, 1993

[Hutc91] N. C. Hutchinson and L. L. Peterson. *The x-Kernel: An architecture for implementing network protocols.* IEEE Transactions on Software Engineering, 17(1): 64-76, Jan. 1991.

[IEC01] Telelogic., *Specification and Description Language,* WebProForum Tutorials, International Engineering Consortium, 2001

[ISO97] ISO/TC97/SC21: *Estelle - A Formal Description Technique Based on an Extended State Transition Model*, ISO, IS 9074, 1997

[J.L77] Peterson J.L. *Petri Nets.* Computer Surveys, Vol. 9, No. 3, September 1977

[J.L81] Peterson J.L. *Petri Net Theory and the Modeling of Systems.* Prentice Hall, Englewood Cliffs, N.J., 1981

[J.ORG] *http://www.jxta.org*

[J.T98] Thees J. *Protocol Implementation with Estelle – from prototypes to efficient implementations,* Estelle'98, November 1998

[Jaap98] Jaap H., Warren A., Inouye J, Joeressen O., Nagshineh M., *Bluetooth: Vision, Goals and Architecture,* Mobile Computing and Communications Review, October 1998, 2(4): 38-45

[Jav90] Esparza J., Silva M., *On the Analysis and Synthesis of Free Choice Systems*, In: Rozenberg, G. (ed.) Lecture Notes in Computer Science, Vol. 483, Advances in Petri Nets 1990, Springer-Verlag, 1991, pp. 243-286.

[JEN81] Jensen K., *Coloured Petri Nets and the Invariant Method,* Theoretical Computer Science, Vol. 14, pp. 317-336, 1981

[JPr02] Bondolo, *Jxta v1.0 Protocols Specification,* spec.jxta.org, Project JXTA, latest revision - June 2002

[JTM01] Moore J.T., Hicks M., Nettles S., *Practical Programmable Packets,* INFOCOMM, IEEE, Alaska, April 2001

[JuBaek98] Jusung Baek, *A design of a protocol for detecting a mobile agent clone and its correctness proof using Colored Petri Nets*, University of South Korea, Technical Report, 1998

[K.P99] Psounis K., *Active Networks: Applications, Security, Safety, and Architectures*, IEEE Communications Surveys, First Quarter 1999

[KGr95] Kari Granö, Harju, Tapani J., Tapani L., Jukka, *Kannel: A Language for Tuning Protocols*, PLST, June 1995

[Kie96] T. Kielmann, *Designing a Coordination Model for Open Systems.* In Proceedings of Coordination'96, Cesena, Italy, April 1996

[KRM74] Keller R. M., *Vector replacement systems: a formalism for modeling asynchronous systems,* Technical Report 117, Princeton University, Princeton, N.J., December 1974

[L.B92] L. Bernardinello, F. De Cindio, *A survey of Basic Net Models and Modular Net Classes*, LNCS Vol. 609, Springer Verlag, 1992

[LiG01] Li Gong, *Project JXTA: A Technology Overview*, Sun Microsystems Inc. April 2001.

[LMH92] Logrippo L., Faci M., Haj H.M., *An Introduction to LOTOS: Learning by Examples.* Computer Networks and ISDN Systems, Vol. 23, pp 325-342, 1992

[MBA92] Mark Abbott, Larry Peterson, *A Language-Based Approach to Protocol Implementation,* Technical Report TR 92-2, The University of Arizona, Tucson, Arizona, July 1992

[MIL89] Milner R., *Calculus of Communicating Systems*, Lecture Notes in Computer Science No.92, Springer-Verlag, 1980.

[Nav96] Navon J., Furuta R., *Collaborative Hyperdocuments and Prototyping Groupware,* Computers in Industry, Vol. 29, pp. 91-104, Elsevier, 1996

[OP.01] *OpenNap: Open Source Napster Server*, http://opennap.sourceforge.net

[ORG84] International Standards Organization, *International standards iso7498: Information processing systems-open systems interconnection-basic reference model*, 1984.

[P.J00] Johri P., *PUMPS: Protocol Uploading and Management Protocol Server*, Technical Report, The University of Georgia, Georgia, 2000

[Pet62] Petri C. A., *Kommunikation mit automaten,* Technical Report RADC-TR-65-377, 1962

[PSS70] Patil S. S., *Coordination of asynchronous events,* PhD. Thesis, MIT, MA, May 1970 (also MAC TR-72, Project MAC, June 1970)

[RFC2060] Crispin M., *Internet Message Access Protocol v4 rev1*, Network working group, RFC 2060, December 1996

[RFC854] Postel J., Reynolds J., *Telnet Protocol Specification*, Network working group, RFC 854, May 1983

[RIC94] W. Richard Stevens, *TCP/IP Illustrated, Vol. 1*, Addison-Wesley, 1994

[SBP87] S. Budkowshi, P. Dembinski, *An Introduction to ESTELLE: A Specification Language for Distributed Systems*, Computer Networks and ISDN Systems, Vol.14, pp. 3-23, 1987

[SBP88] S. Budkowshi, P. Dembinski, M.Diaz, *ISO Standardized Description Technique ESTELLE,* Technical Report, Distributed System Architecture and Standards, France, 1988

[SHA94] Robin Sharp, *Principles of Protocol Design*, Prentice Hall, 1994

[SLi01] S. Li, *Early Adopter JXTA: Peer-to-Peer Computing with Java*, Wrox Press Inc, December 2001.

[TAD89] Tadao M., *Petri nets: Properties, Analysis and Applications,* Proceedings of IEEE, Vol. 77, No. 4, 1989

[TBE87] T. Bolognesi, E. Brinksma, *Introduction to the ISO Specification Language LOTOS,* Computer Networks and ISDN Systems, Vol. 14, pp. 25-59, 1987

[Von92] Von Eicken, David Culler, Seth, Klaus, *Active Messages: A Mechanism for Integrated Communication and Computation*, ISCA, pages 256-266, 1992

[WCN01] Banerji et.al., *Web Services Conversation Language (WSCL) 1.0*, W3C Note 14, March 2002

[WWS02] Web Services Activity, *http://www.w3.org/2001/ws*, W3C Architecture Domain

[XML00] World wide web Consortium Recommendation, *Extensible Markup Language Version 1.0*, October 2000

[YEN94] Yen-Min Huang, Chinya Ravishankar, *Linguistic Support for Controlling Protocol Execution*, ICDCS 1994, pages 581-588

[Z.100] *Specification and Description Language (SDL) – Recommendation Z.100*, The International Telecommunications Union. Latest Revision: Nov. 1999

APPENDIX

A. Contents of the JAR file for LOGIN protocol

```
/*
 * LOGIN.java, holds expressions, guards and actions for the
 * clear text LOGIN protocol.
 */
import java.io.*;
import java.util.*;

public class LOGIN {

  public LOGIN() {
    System.out.println(" <<LOGIN instantiated>>");
  }
  //----------- RECVCommand, Authenticate (any)-------------
  public boolean any(Vector in, Vector out) {
    boolean status = false;
    Enumeration e = in.elements();
    while(e.hasMoreElements())
    {
      ProtocolMessage m = (ProtocolMessage) e.nextElement();
      out.add(m);
      status = true;
    }
    return status;
  }
  //---------------------- Authenticate --------------------
  public boolean No_Sync_Mesgs(Vector in, Vector out) {
    Enumeration e = in.elements();
    while(e.hasMoreElements())
    {
      ProtocolMessage m = (ProtocolMessage) e.nextElement();
      if (!m.getName().equals("SYNC"))  out.addElement(m);
    }
    return (out.size()>0);
  }

  //-------------- Authenticate (gAuthenticate)-------------
  public boolean gAuthenticate(Hashtable in) {
    return true;
  }
```

```
  //------------- Authenticate (aAuthenticate)---------------
  public void aAuthenticate(Hashtable in, Vector out) {
    System.out.print("Inside aAuthenticate");
    System.out.println(in.toString());

    // get login and password
    Vector v = (Vector)in.get("LOGIN.LoginReceived");
    ProtocolMessage loginInfo = (ProtocolMessage)v.elementAt(0);

    ProtocolMessage result = new ProtocolMessage();

    result.setName("LOGIN");
    result.setType("RESULT");

    // authenticate
    if (("kannan".equals(loginInfo.getAttribute("PARAM1"))) &&
        ("test".equals(loginInfo.getAttribute("PARAM2")))) {
        result.setAttribute("VALUE","SUCCESS");
    }
    else {
        result.setAttribute("VALUE","ERROR");
    }
    result.setAttribute("USER",loginInfo.getAttribute("PARAM1"));
    out.addElement(result);
  }
}
```

B. Contents of the JAR file for IMAP protocol

```
import java.io.*;
import java.util.*;
import javax.swing.JOptionPane;

public class IMAP {
  public IMAP() {
}

  //---------- RECVCommand, Authenticate (any)------------
  public boolean any(Vector in, Vector out) {
    boolean status = false;
    Enumeration e = in.elements();
    while(e.hasMoreElements())
    {
      ProtocolMessage m = (ProtocolMessage) e.nextElement();
      out.add(m);
      status = true;
    }
    return status;
  }

  //----------- LoginCheck (output) ----------------------
  public boolean OnNoError(Vector in, Vector out) {
    Enumeration e = in.elements();
    while(e.hasMoreElements())
    {
      ProtocolMessage m = (ProtocolMessage) e.nextElement();
      if (!m.getAttribute("VALUE").equals("ERROR"))
         out.addElement(m);
    }
    return (out.size()>0);
  }

  //----------- UserCommand (input) ----------------------
  public boolean OnCloseCommand(Vector in, Vector out) {
     Enumeration e = in.elements();
     while(e.hasMoreElements()) {
       ProtocolMessage m = (ProtocolMessage) e.nextElement();
       if ((m.getAttribute("COMMAND").equals("CLOSE")) &&
            (m.getAttribute("RESPONSE").equals("OK"))      ) {
          out.addElement(m);
       }
     }
     return (out.size()>0);
  }

  //----------- UserCommand (input) ----------------------
  public boolean OnlyValidCommands(Vector in, Vector out) {
    boolean retValue = true;
```

```
  Enumeration e = in.elements();
  while(e.hasMoreElements()) {
    ProtocolMessage m = (ProtocolMessage) e.nextElement();
    if (!(m.getAttribute("COMMAND").equals("SELECT")) &&
        !(m.getAttribute("COMMAND").equals("EXAMINE")) &&
     !(m.getAttribute("COMMAND").equals("FETCH")) &&
     !(m.getAttribute("COMMAND").equals("CLOSE"))   ) {
        retValue = false;
    }
    else
       out.addElement(m);
  }
  return retValue;
}

//----------- UserLoggedIn (input) ----------------------
public boolean OnNoClose(Vector in, Vector out) {
  boolean retValue = true;
  Enumeration e = in.elements();
  while(e.hasMoreElements()) {
    ProtocolMessage m = (ProtocolMessage) e.nextElement();
    if (((m.getAttribute("COMMAND").equals("CLOSE")) &&
         (m.getAttribute("RESPONSE").equals("OK")))) {
       retValue = false;
    }
    else {
       if (!m.getType().equals("."))
          out.addElement(m);
    }
  }
  return retValue;
}

public boolean noGuard(Hashtable in) {
  return true;
}

public boolean OnResponseNotBad(Vector in, Vector out) {
  boolean retValue = true;
  Enumeration e = in.elements();
  while(e.hasMoreElements()) {
     ProtocolMessage m = (ProtocolMessage) e.nextElement();
     if (m.getAttribute("RESPONSE").equals("BAD")) {
         retValue = false;
     }
     else {
        if (!m.getType().equals("*"))
            out.addElement(m);
     }
  }
  return retValue;
}
```

```
//------------- LoginCheck (action)-----------------------
public void aLoginCheck(Hashtable in, Vector out) {
  System.out.print("Inside aLoginCheck");
  System.out.println(in.toString());

  // get first number
  Vector v = (Vector)in.get("IMAP.EndLogin");
  ProtocolMessage loginResult =
       (ProtocolMessage)v.elementAt(0);

  ProtocolMessage output = new ProtocolMessage();

  output.setName("IMAP");
  output.setType("USER");
  output.setAttribute("VALUE",
         loginResult.getAttribute("USER"));

  // Do some book keeping if necessary.
  out.addElement(output);
}


//------------ ActivateSession (guard)------------------------
public boolean gActivateSession(Hashtable in) {
  System.out.print("Inside gActivateSession");
  System.out.println(in.toString());

  // get first number
  Vector v1 = (Vector)in.get("IMAP.LoginResultReceived");
  ProtocolMessage pm = (ProtocolMessage) v1.elementAt(0);

  return (pm.getAttribute("VALUE").equals("ERROR")) ;
}

//---- --- ActivateSession (action)--------------------------
public void aActivateSession(Hashtable in, Vector out) {
  System.out.print("Inside aActivateSession");
  System.out.println(in.toString());

  // get first number
  Vector v1 = (Vector)in.get("IMAP.LoginResultReceived");
  Vector v2 = (Vector)in.get("IMAP.ConnectionClosed");
}

public void aChooseNewMBox(Hashtable in, Vector out) {
  System.out.print("Inside aChooseNewMBox");
  System.out.println(in.toString());

  // get first number
  Vector v1 = (Vector)in.get("IMAP.MBoxCommandProcessed");

  ProtocolMessage output = null;
```

```
  Enumeration enum = v1.elements();
  while (enum.hasMoreElements()) {
     ProtocolMessage pm = (ProtocolMessage) enum.nextElement();

     if (pm.getAttribute("COMMAND").equals("SELECT")) {
       output = new ProtocolMessage("IMAP","MBOX");
       output.setAttribute("MBOX",pm.getAttribute("MBOX"));
     }
  }

  if (null != output) out.addElement(output);
}

public void aReplaceOldMBox(Hashtable in, Vector out) {
  System.out.print("Inside aReplaceOldMBox");
  System.out.println(in.toString());

  // get first number
  Vector v1 = (Vector)in.get("IMAP.MBoxCommandProcessed");
  Vector v2 = (Vector)in.get("IMAP.MBoxCurrentlySelected");

  ProtocolMessage output = null;

  Enumeration enum = v1.elements();
  while (enum.hasMoreElements()) {
      ProtocolMessage pm = (ProtocolMessage)
                enum.nextElement();
      if (pm.getAttribute("COMMAND").equals("SELECT")) {
         output = new ProtocolMessage("IMAP","MBOX");
         output.setAttribute("MBOX",pm.getAttribute("MBOX"));
      }
  }
  if (null != output) out.addElement(output);
}

public boolean gProcessClose(Hashtable in) {
  Vector v = (Vector) in.get("IMAP.UserCommandReceived");
  ProtocolMessage command = (ProtocolMessage) v.elementAt(0);
  if (null==command.getAttribute("COMMAND")) return false;
  return (command.getAttribute("COMMAND").equals("CLOSE"));
}

public void aProcessClose(Hashtable in, Vector out) {
  System.out.print("Inside aProcessClose");
  System.out.println(in.toString());

  Vector v = (Vector) in.get("IMAP.UserCommandReceived");
  ProtocolMessage command = (ProtocolMessage) v.elementAt(0);

  ProtocolMessage output1 = new ProtocolMessage();

  output1.setName("IMAP");
```

```
  output1.setType("SERVER");
  output1.setAttribute("COMMAND","*");
  output1.setAttribute("RESPONSE","BYE");
  output1.setAttribute("BYE","BYE");

  // Do some book keeping if necessary.
  out.addElement(output1);

  ProtocolMessage output = new ProtocolMessage();

  output.setName("IMAP");
  output.setType("SERVER");
  output.setAttribute("COMMAND","CLOSE");
  output.setAttribute("RESPONSE","OK");

  // Do some book keeping if necessary.
  out.addElement(output);
}

public boolean gProcessSelect(Hashtable in) {
  Vector v = (Vector) in.get("IMAP.UserCommandReceived");
  ProtocolMessage command = (ProtocolMessage) v.elementAt(0);
  if (null==command.getAttribute("COMMAND")) return false;
  return (command.getAttribute("COMMAND").equals("SELECT"));
}

public void aProcessSelect(Hashtable in, Vector out) {
  String mbox = "";
  System.out.print("Inside aProcessSelect");
  System.out.println(in.toString());

  Vector v = (Vector) in.get("IMAP.UserCommandReceived");
  ProtocolMessage command = (ProtocolMessage) v.elementAt(0);

  if (null != command.getAttribute("PARAM1")) {
     mbox = command.getAttribute("PARAM1");
  }

  ProtocolMessage output1 = new
       ProtocolMessage("IMAP","SERVER");
  ProtocolMessage output2 = new
       ProtocolMessage("IMAP","SERVER");
  ProtocolMessage output3 = new
       ProtocolMessage("IMAP","SERVER");

  output1.setAttribute("MBOX",mbox);
  output2.setAttribute("MBOX",mbox);
  output3.setAttribute("MBOX",mbox);

  if (mbox.equalsIgnoreCase("INBOX")) {
      output1.setAttribute("COMMAND","*");
      output1.setAttribute("RESPONSE","RECENT");
```

```
      output1.setAttribute("RECENT","4");
      out.addElement(output1);

      output2.setAttribute("COMMAND","*");
      output2.setAttribute("RESPONSE","EXISTS");
      output2.setAttribute("EXISTS","11");
      out.addElement(output2);

      output3.setAttribute("COMMAND","SELECT");
      output3.setAttribute("RESPONSE","OK");
      out.addElement(output3);
 }
 else if (mbox.equals("gemini")) {
      output1.setAttribute("COMMAND","*");
      output1.setAttribute("RESPONSE","RECENT");
      output1.setAttribute("RECENT","5");
      out.addElement(output1);

      output2.setAttribute("COMMAND","*");
      output2.setAttribute("RESPONSE","EXISTS");
      output2.setAttribute("EXISTS","7");
      out.addElement(output2);

      output3.setAttribute("COMMAND","SELECT");
      output3.setAttribute("RESPONSE","OK");
      out.addElement(output3);
 }
 else {
      output3.setAttribute("COMMAND","SELECT");
      output3.setAttribute("RESPONSE","BAD");
      output3.setAttribute("BAD","invalid mailbox");
      out.addElement(output3);
   }
}

public boolean gProcessExamine(Hashtable in) {
   Vector v = (Vector) in.get("IMAP.UserCommandReceived");
   ProtocolMessage command = (ProtocolMessage) v.elementAt(0);
   if (null==command.getAttribute("COMMAND")) return false;
   return (command.getAttribute("COMMAND").equals("EXAMINE"));
}

public void aProcessExamine(Hashtable in, Vector out) {
    String mbox = "";
    System.out.print("Inside aProcessExamine");
    System.out.println(in.toString());

    Vector v = (Vector) in.get("IMAP.UserCommandReceived");
    ProtocolMessage command = (ProtocolMessage) v.elementAt(0);


   if (null != command.getAttribute("PARAM1")) {
```

```
        mbox = command.getAttribute("PARAM1");
     }

    ProtocolMessage output1 = new
         ProtocolMessage("IMAP","SERVER");
    ProtocolMessage output2 = new
         ProtocolMessage("IMAP","SERVER");
    ProtocolMessage output3 = new
         ProtocolMessage("IMAP","SERVER");

    if (mbox.equalsIgnoreCase("INBOX")) {
        output1.setAttribute("COMMAND","*");
        output1.setAttribute("RESPONSE","RECENT");
        output1.setAttribute("RECENT","4");
        out.addElement(output1);

        output2.setAttribute("COMMAND","*");
        output2.setAttribute("RESPONSE","EXISTS");
        output2.setAttribute("EXISTS","11");
        out.addElement(output2);

        output3.setAttribute("COMMAND","EXAMINE");
        output3.setAttribute("RESPONSE","OK");
        out.addElement(output3);
    }
    else if (mbox.equals("gemini")) {
        output1.setAttribute("COMMAND","*");
        output1.setAttribute("RESPONSE","RECENT");
        output1.setAttribute("RECENT","5");
        out.addElement(output1);

        output2.setAttribute("COMMAND","*");
        output2.setAttribute("RESPONSE","EXISTS");
        output2.setAttribute("EXISTS","7");
        out.addElement(output2);

        output3.setAttribute("COMMAND","EXAMINE");
        output3.setAttribute("RESPONSE","OK");
        out.addElement(output3);
    }
    else {
        output3.setAttribute("COMMAND","EXAMINE");
        output3.setAttribute("RESPONSE","BAD");
        output3.setAttribute("BAD","invalid mailbox");
        out.addElement(output3);
    }
}


public boolean OfTypeServer (Vector in, Vector out) {
    Enumeration e = in.elements();
    while(e.hasMoreElements()) {
```

```
        ProtocolMessage m = (ProtocolMessage) e.nextElement();
        if (m.getType().equals("SERVER")) {
           out.addElement(m);
        }
      }
    return (out.size()>0);
  }

  public boolean OfTypeMBox (Vector in, Vector out) {
     Enumeration e = in.elements();
     while(e.hasMoreElements()) {
        ProtocolMessage m = (ProtocolMessage) e.nextElement();
           if  (m.getType().equals("MBOX")) {
              out.addElement(m);
        }
     }
     return (out.size()>0);
  }

  public boolean gProcessFetch(Hashtable in) {
      Vector v = (Vector) in.get("IMAP.UserCommandReceived");
      ProtocolMessage command = (ProtocolMessage) v.elementAt(0);
      if (null==command.getAttribute("COMMAND")) return false;
     return (command.getAttribute("COMMAND").equals("FETCH"));
  }


  public void aPassTheResponse(Hashtable in,Vector out) {
    Vector v1 = (Vector) in.get("IMAP.ResponsePool");

    Enumeration enum = v1.elements();
    while (enum.hasMoreElements()) {
   ProtocolMessage mesg = (ProtocolMessage) enum.nextElement();
      out.addElement(mesg);
    }
  }

  public static void main(String[] args) {
    System.out.println("Main of IMAP, for testing");
  }
}
```

C. Contents of the JAR file for GNUTELLite protocol

```
import java.io.*;
import java.util.*;
import javax.swing.JOptionPane;

public class GNUTELLite1 {

  public static final String MyPeerID="1";
  public GNUTELLite1() {
  }

  //------------ RECVCommand, Authenticate (any)-------------
  public boolean any(Vector in, Vector out) {
    boolean status = false;
    Enumeration e = in.elements();
    while(e.hasMoreElements())
    {
      ProtocolMessage m = (ProtocolMessage) e.nextElement();
      out.add(m);
      status = true;
    }
    return status;
  }


  public boolean noGuard(Hashtable in) {
    return true;
  }

  public boolean InvalidTTL(Vector in, Vector out) {
    boolean status = false;
    Enumeration e = in.elements();
    while (e.hasMoreElements()) {
      ProtocolMessage m = (ProtocolMessage) e.nextElement();
      if ((Integer.valueOf(m.getAttribute("TTL")).intValue()))
        out.add(m);
        status = true;
      }
    }
    return status;
  }

  public boolean IsValidTTL(Vector in, Vector out) {
    boolean status = false;
    Enumeration e = in.elements();
    while (e.hasMoreElements()) {
      ProtocolMessage m = (ProtocolMessage) e.nextElement();
      if ((Integer.valueOf(m.getAttribute("TTL")).intValue()) {
        out.add(m);
        status = true;
```

```
    }
  }
  return status;
}

public boolean IsResponse (Vector in, Vector out) {
  boolean status = true;
  Enumeration e = in.elements();
  while (e.hasMoreElements()) {
    ProtocolMessage m = (ProtocolMessage) e.nextElement();
    if (m.getType().equalsIgnoreCase("HITS")) {
      out.add(m);
      status = true;
    }
  }
  return status;
}

public boolean IsQuery (Vector in, Vector out) {
  boolean status = true;
  Enumeration e = in.elements();
  while (e.hasMoreElements()) {
    ProtocolMessage m = (ProtocolMessage) e.nextElement();
    if (m.getType().equalsIgnoreCase("QUERY")) {
      out.add(m);
      status = true;
    }
  }
  return status;
}

public boolean IsMyMessage (Vector in, Vector out) {
  boolean status = true;
  Enumeration e = in.elements();
  while (e.hasMoreElements()) {
    ProtocolMessage m = (ProtocolMessage) e.nextElement();
    if (m.getAttribute("MESSAGEID").equals(MyPeerID)) {
      out.add(m);
      status = true;
    }
  }
  return status;
}

public boolean PeerMessage (Vector in, Vector out) {
  boolean status = true;
  Enumeration e = in.elements();
  while (e.hasMoreElements()) {
    ProtocolMessage m = (ProtocolMessage) e.nextElement();
    if (!m.getAttribute("MESSAGEID").equals(MyPeerID)) {
      out.add(m);
      status = true;
```

```
    }
  }
  return status;
}

public void aDropMessage(Hashtable in, Vector out) {
  System.out.print("Inside aDropMessage with");
  System.out.println(in.toString());
  out.clear();
}
public void aDecrementTTL(Hashtable in, Vector out) {
  System.out.print("Inside aDecrementTTL with");
  System.out.println(in.toString());

  //Get the message
  ProtocolMessage pm = (ProtocolMessage) ((Vector)
    in.get("GNUTELLite.MessageReceived")).elementAt(0);
  pm.setAttribute("TTL",String.valueOf(Integer.valueOf(
                  pm.getAttribute("TTL")).intValue()-1));
  out.add(pm);
}

public void aProcessQuery(Hashtable in, Vector out){
  System.out.print("Inside aProcessQuery");
  System.out.println(in.toString());
  boolean fileExists = true;

  //Get the message
  ProtocolMessage pm = (ProtocolMessage) ((Vector)
    in.get("GNUTELLite.QueryMessage")).elementAt(0);

  String filename = pm.getAttribute("FILENAME");;
  String response = new String();
  // open input filestream and create response message
  File file = new File(".\\shared\\".concat(filename));
  if (!file.exists()) fileExists = false;

  if (fileExists) {
    ProtocolMessage pm1 = new
       ProtocolMessage("GNUTELLite","HITS");
    pm1.setAttribute("MESSAGEID",pm.getAttribute("MESSAGEID"));
    pm1.setAttribute("PEERID",MyPeerID);
    pm1.setAttribute("TTL","2");
    out.add(pm1);
  }
  else {
    out.add(pm);
  }
}

public void aProcessResponse(Hashtable in, Vector out){
  System.out.print("Inside aProcessResponse");
```

```
    System.out.println(in.toString());
    boolean fileExists = true;

    //Get the message
    ProtocolMessage pm = (ProtocolMessage) ((Vector)
      in.get("GNUTELLite.ResponseMessage")).elementAt(0);

    pm.printMessage();
    out.add(pm);
  }

  public static void main(String[] args) {
    System.out.println("Main of IMAP, for testing");
  }
}
```