Resilient and Efficient Delivery over Message Oriented Middleware.

Jia, Yue

For additional information about this publication click this link.
http://qmro.qmul.ac.uk/jspui/handle/123456789/8950

Resilient and Efficient Delivery over Message Oriented Middleware

by

Yue Jia

A thesis submitted to the University of London for the degree of
Doctor of Philosophy

School of Electronic Engineering and Computer Science
Queen Mary, University of London
29 September 2014

Acknowledgements

I would like to gratefully and sincerely thank Dr. Chris I Phillips for his guidance, understanding, patience, and most importantly, his friendship during my graduate studies at Queen Mary. His mentorship was paramount in providing a well rounded experience consistent my PhD research. I deeply appreciate all his contributions of time and work to make my research productive and stimulating. For everything you've done for me, Dr. Phillips, I thank you.

I would also like to thank Dr. John Bigham and Dr. Eliane L Bodanese for their guidance and support that have given me. Their knowledge and enthusiasm towards research and life have inspired me all the time. Additionally, I am very grateful for the friendship of all of the members of Network Research group, especially Ran Tao, with whom I worked closely and puzzled over many of the same problems.

I would like to thank Kejiong Li and Xinyue Wang, especially those members in my office for their support, solicitude, and valuable discussions.

Finally, and most importantly, I would like to thank my parents. Their support, encouragement, quiet patience and unwavering love were undeniably the bedrock upon which the past years of my life have been built. Their tolerance of my occasional vulgar moods is a testament in itself of her unyielding devotion and love. It was under their watchful eye that I gained so much drive and an ability to tackle challenges head on.

Abstract

The publish/subscribe paradigm is used to support a many-to-many model that allows an efficient dissemination of messages across a distributed system. Message Oriented Middleware (MOM) is a middleware that provides an asynchronous method of passing information between networked applications. MOMs can be based on a publish/subscribe model, which offers a robust paradigm for message delivery. This research is concerned with this specific type of MOM. Recently, systems using MOMs have been used to integrate enterprise systems over geographically distributed areas, like the ones used in financial services, telecommunication applications, transportation and health-care systems. However, the reliability of a MOM system must be verified and consideration given to reachability to all intended destinations typically with to guarantees of delivery.

The research in this thesis provides an automated means of checking the (re)configuration of a publish/subscribe MOM system by building a model and using Linear-time Temporal Logic and Computation Tree Logic rules to verify certain constraints. The verification includes the checking of the reachability of different topics, the rules for regulating the working of the system, and checking the configuration and reconfiguration after a failure. The novelty of this work is the creation and the optimization of a symbolic model checker that abstracts the end-to-end network configuration and reconfiguration behaviour and using it to verify reachability and loop detection. In addition a GUI interface, a code generator and a sub-paths detector are implemented to make the system checking more user-friendly and efficient.

The research then explores another aspect of reliability. The requirements of mission critical service delivery over a MOM infrastructure is considered and we propose a new way of supporting rapid recovery from failures using pre-calculated routing

tables and coloured flows that can operate across multiple Autonomous System domains. The approach is critically appraised in relation to other published schemes.

Table of Content

List of Figures

List of Tables

List of Abbreviations

AMQP	Advanced Message Queue Protocol
API	Application Programming Interface
ATL	Alternating-time Logic
CAM	Content-addressable Memory
CCM	CORBA Communication Model
CORBA	Common Object Request Broker Architecture
CTL	Computation Tree Logic
DCOM	Distributed Component Object Model
DDS	Data Distribution Service
EJB	Enterprise JavaBeans
EWMA	Exponentially Weighted Moving Averaging
FIB	Forwarding Information Base
FIFO	First-In-First-Out
GUI	Graphical User Interface
ID	Identification
IT	Information Technology
JMS	Java Message Service
LIFO	Last-In-First-Out
LSA	Link-state Advertisement
LSD	Link State Database

LTL	Linear Time Logic
MOM	Message Oriented Middleware
MPLS	Multi-Protocol Label Switching
MPLS	Multiprotocol Label Switching
NuSMV	New Symbolic Model Checker
OBDD	Ordered Binary Decision Diagram
OSPF	Open Shortest Path First
P/S	Publish/Subscribe
PCRT	Pre-Calculated Routing Table
QoS	Quality of Service
Qpid	Open Source AMQP Messaging
RIB	Routing Information Base
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RRL	Resilient Routing Layers
SLSD	Super Link State Database
SPF	Shorted Path First
TCP	Transmission Control Protocol
XML	Extensible Markup Language

1 Introduction

1.1 Introduction and Motivation

A telephone communication system is a typical synchronous system. A synchronous system [51] involves tight coupling which requires both the caller and the callee to be available at the same time. This kind of systems causes considerable problems when trying to implement certain forms of communication between end-systems or managing the interaction between clients and servers in regard to fault tolerance, availability, and so forth). Therefore, asynchronous systems have become a popular alternative. Like an email system, an asynchronous system [52] has queues to store messages transitionally, guaranteeing that messages are retained even after failures occur. A key advantage over traditional synchronous solutions in terms of fault tolerance and overall system flexibility is that asynchronous system components do not need to be operational at the time a request is made. Queues provide a way to communicate across diverse networks and systems while still being able to make some assumptions about the behaviour of the messages. Message Oriented Middleware (MOM) is a loosely coupling asynchronous infrastructure [3, 4, 5, 45]. A MOM is an infrastructure focused on sending and receiving messages that allows application modules to be distributed over heterogeneous platforms [2].

MOM provides messaging services between the transport layer and application layer [1, 6]. Processes and/or applications do not need to know the existence of each other, so it allows communication in an asynchronous, decoupled manner. In a topic-based publish/subscribe MOM, each message is classified as belonging to one of a fixed set of topics. There can be an arbitrary number of topics in the system. A publisher labels each message it produces with a particular topic. Similarly, the subscriber has the ability to express their interest in a topic or a pattern of topics to a broker in a suitable

data structure. When a component publishes a message, the broker matches this against existing subscriptions, and delivers the message to all those application components that issued matching subscriptions that are local or at neighbouring brokers, and optionally performs message mediation. Each component can publish and subscribe to one or many topics. Due of its nature, a MOM system is widely recognized as a promising solution for communication between dissimilar systems. During the communication phase, brokers located at different geographical areas may experience unexpected failure of a broker or an interconnection.

As one of the most popular applications in MOM, mission critical enterprise-computing systems, such as banking systems and stock markets, are the focus of this research. A mission critical system is a system that is essential to the survival of a business or organization. When a mission critical system fails or is interrupted, business operations are significantly impacted. A mission critical message oriented middleware application requires reliable and efficiency message delivery [46, 47, 103, 104, 106]. Messages are assumed to be time sensitive so it is critical to employ an overlay network fast recovery scheme. Given the growing size and complexity of MOM-based mission critical overlay networks, the presence of component failures is expected to be an everyday occurrence. Hence, considerable attention has been devoted to the problem of fast recovery from link failures [53, 54, 55].

Recent studies show that network access control configuration is one of the most complex and error prone network management tasks [7]. For this reason, network misconfiguration has become the main source of network unreachability and vulnerability problems. To achieve resilience, an efficient way to (re)configure the system is necessary. The main idea of configuring and reconfiguring systems is to make sure that the systems works normally under various conditions. However, if the protocols which determine the configuration and reconfiguration have flaws in

themselves, the (re)configuration will be unreliable. Therefore, it is important to find a means of verifying that the configuration protocol or mechanism is correct for a self-healing MOM system.

Moreover, one of the advantages of using a publish/subscribe structure in a MOM system is that it allows the MOM system to be more flexible [56]. This is because an additional logical layer, called the dispatcher, mediates communications; the brokers, publishers and subscribers' information is dynamically controlled by the dispatcher. As a consequence, new components can join a federation, become immediately active, and cooperate with the others without requiring any reconfiguration of the architecture. Furthermore, any addition or deletion of publishers or subscribers will not affect other devices. However, this flexibility of adding or removing devices impedes the validation of a publish/subscribe system. The verification of applications developed using this paradigm is a challenging task. It is easy to configure an individual device like a broker or a firewall, but it is extreme complex to attempt the configuration of a large network with many connections between devices [7]. Components are often written independently of the way they are federated and their interactions can change dynamically. Manual analysis of the interaction among the policies of the many devices in the network with different syntax and semantics is unfeasible [8]. A reduced / abstracted model of a system can enable complex problems to be represented in a simpler form. Verifying all the rules based on this model is thus tractable. Hence, this research focuses on building a MOM system model and verifying the reachability and reconfiguration rules of the modelled system.

Protocols for the real-time restoration of mesh transport networks have been studied for several years [105, 106, 107, 108, 109]. Investigators generally agree that it is feasible for such protocols to compute a set of replacement paths for span restoration in under 2 seconds [109], which is fast enough to avoid large scale consequences

within the network. However, there is still a significant motivation to increase the speed of mesh-based restoration schemes [109].

Though MOM is an overlay system, it depends on the underlay network layer. If the network layer fails, it will affect the resilience of the MOM system. With larger network sizes of MOM systems to meet increasing demands, there is a higher chance of broker failure. As a resilient system, it is important for a MOM system to have an effective fast recovery mechanism for link or broker failures.

1.2 Research Contributions

This research proposes two means of promoting the resilience in a mission critical MOM system. This research proposes a model-checking tool to verify the reachability and reconfiguration rules of MOM systems. In addition, in order to recover rapidly from failure, this thesis proposes a Pre-Calculated Routing Table (PCRT) algorithm that is simpler, faster, loop free and more robust than several state-of-the-art network fast recovery mechanisms [105]. More precisely, this thesis provides several contributions to better achieve resilience in a realistic MOM system as follows:

1. From the year 1996 to 2013, there are over 500 papers published by IEEE, ACM and Springer regarding publish/subscribe paradigms and systems. However, less than 60 combined model-checking tools are proposed to verify the software and protocol correctness belonging to a publish/subscribe infrastructure. None of them used model-checking tools to validate the (re)configuration of an overlay network, characteristic of a publish/subscribe-based Message Orientated Middleware (MOM) system. To our knowledge this research is the first to propose the use of a formal verification model checker tool to validate the (re)configuration of a MOM overlay network.

2. Publish/subscribe systems are hard to validate. In particular, given the inherent

non-determinism in the order of received events, delays in event delivery, and variability in the timing of event announcements, consequently, the number of possible system states becomes combinatorial large.

This research automatically generates a finite-state MOM model in a model checking language, called New Symbolic Model Checker (NuSMV). Using a model checking method combined with Computation Tree Logic (CTL) or Linear Time Logic (LTL) [39] or possibly other types of logic statement can ensure that a MOM overlay functions as expected upon reconfiguration in accordance with the requirements of the business model (end user or application). A novel NuSMV Code Generator is designed in Java that reads a Hash Map file created by a Graphical User Interface (GUI) front-end process from which a full NuSMV MOM model is generated with all the specifications necessary for verification within a NuSMV verification environment.

3. However, verifying the reachability of every topic can take a considerable amount of time. Therefore, a novel method of identifying sub-paths in a network is developed. This research uses these sub-paths to eliminate duplicate paths by verifying the reachability of 'super paths'. In this way considerable time can be saved in respect to the verification of the reachability of topics.

4. As a MOM is an overlay system, it depends on the underlay network layer. If the network layer fails, it will affect the resilience of the MOM system. With increases in the network size of MOM systems to support additional distributed customer end-systems and to meet the increasing traffic demands, there is an associated higher chance of virtual link failures between brokers, or indeed broker failures. As a resilient system, it is important for a MOM system to have methods in place to handle link/links and broker failures.

This research presents a novel algorithm, with a 'colour' representation of the network state as a factor, for pre-calculating routing tables and condensing them into a single super routing table per node to save space. This algorithm guarantees 100% fast recovery from single link failures or single node failures. In addition, shortest path first (SPF) [57, 58] routing is maintained even after a failure has happened. Since all the calculation of routing tables is performed in advance, the online time needed to run the SPF calculation is avoided. This Pre-Calculated Routing Table algorithm is compared with two other popular fast recovery schemes [101, 102] (i.e. IP Fast Recovery and the Resilient Routing Layers algorithm) and provides favourable results.

5. As part of the proactive PCRT algorithm, a novel idea of super condensed routing table is provided. Rather than storing routing tables for every failure situation, this research proposed to merge those routing tables into one super condensed routing table avoiding redundant entries as we note that quite a lot of routing tables share the same routing entries for different failure situations. Using the novel 'colour' concept and a two-dimensional array, a corresponding routing entry in a super condensed routing table can be located directly, avoiding the need to perform searches and so provide low latency access to the appropriate next hop data and other routing information.

1.3 Author's Publications

As part of this research the following publications have been produced by the author.

Y. Jia, E. Bodanese, J. Bigham, 'Model Checking of the Reliability of Publish/Subscribe Structure Based System', First IEEE International Conference on Communications in China, Beijing, August 2012, pp 173-178.

(This paper provides a novel and scalable design for abstracting a MOM overlay

network into a model for formal verification)

Y. Jia, E. Bodanese, J. Bigham, 'Checking the Robustness of a Publish/Subscribe Based Message Oriented System', IV International Congress on Ultra Modern Telecommunications and Control Systems, St. Petersburg, October 2012, pp 291-296.

(This paper proposes a sub-path detection algorithm to reduce verification time. The paper also describes our Code Generator for translating MOM overlay network into a model checker model)

Y. Jia, E. Bodanese, C. Phillips, J. Bigham, and R. Tao. 'Improved Reliability of Large Scale Publish/Subscribe based MOMs using Model Checking', in Proc. of IEEE/IFIP Network Operations and Management Symposium (NOMS'14), Poland, 2014.

(This paper models a realistically large-scale MOM overlay network and specifically shows the data structures and procedures employed by the Code Generator)

Y. Jia, C. Phillips. 'Fast and Reliable IP Recovery for Overlay Routing in Mission Critical Message Oriented Middleware', 8th International Conference on Frontier of Computer Science and Technology (FCST2014), Chendu, China, December 19-21, 2014. Submitted

(This paper presents a novel algorithm for pre-calculating routing tables and condensing them to save space. The algorithm is compared with two other popular fast recovery schemes and provides favourable results)

1.4 Thesis Organization

The remainder of this thesis is organised as follows. Chapter 2 presents the main

features underpinning MOM systems. Firstly, it describes the publish/subscribe concept followed by the message queuing paradigm. It presents Java Message Service (JMS) [24, 25] as the first implementation of a message oriented middleware system. The Advanced Message Queue Protocol (AMQP) [21, 22. 23] is then introduced as one of the protocols used to implement publish/subscribe MOMs together with a broker implementation using Open Source AMQP Message (Qpid) [21] from Apache. IBM WebSphere [98, 139,140] is described as a real implementation of a message queuing system. The chapter then concludes with an analysis of the similarities and differences between both approaches, including a critical appraisal of their major advantages and disadvantages by comparing the presented MOM systems. In addition, several state-of-the-art measures of fast recovery from a network failure are introduced in this chapter.

Chapter 3 then presents Temporal Logic [39] concepts that are used in the proposed model checker, called New Symbolic Model Checker (NuSMV) [38], and how they are expressed. Following this, Chapter 4 provides a review of related literature and shows the importance and usefulness of model checking large-scale systems like MOM networks. The chapter also presents and critically appraises published work concerned with modelling publish/subscribe systems and how they relate to the original work subsequently presented in this thesis. The chapter concludes with an analysis of the main differences between the state-of-the-art and the contributions proposed in this thesis.

Chapter 5 describes two abstracted MOM models together with an innovative way for generating NuSMV models and specifications. The time taken to generate models of different MOM sizes is examined. Moreover, to avoid the state 'explosion' problem, a sub-path detection method is developed in this research and the improvements it provides are evaluated.

Chapter 6 considers a MOM system of realistic size and number of topics. A user-friendly interface and a routing table generation program are designed to provide a much easier way for users to model a large-scale MOM system. Sub-path detection is also implemented into this model. This thesis also presents a novel method to verify the link table when the routing table is not available in advance.

In addition to ensuring state information is valid, a further means of improving the robustness of a MOM is to investigate resilience measures so that inevitable failures can be accommodated in an appropriate way. To this end, Chapter 7 introduces a novel approach of pre-calculating routing table algorithm to allow for speedy recovers from various forms of failure. Detailed explanations, examples and simulations of this algorithm are provided. Moreover, an assessment of the scheme compared with state-of-the-art alternatives is also provided in this chapter and demonstrates its superior performance.

Finally, Chapter 8 provides a summary of the key conclusions of this research and also considers possibilities for future work.

2 Message Oriented Middleware Systems and Overlay Networks for Fast Recovery

2.1 Message Oriented Middleware Systems

The scale of distributed systems [59] has increased considerably. Distributed systems may involve thousands of entities potentially distributed all over the world. There is a need for a linking infrastructure to bind large distributed systems together in a reliable manner. This infrastructure is supported by dedicated middleware, which can handle large messaging and security requirements while retaining the decoupled nature of the application.

The communication paradigms employed in distributed systems are mainly built on transmission control protocol (TCP) [124], and include remote procedure call (RPC) [125], used for example in some applications of Enterprise JavaBeans (EJBs) [126], but mainly use asynchronous messaging, and middleware functionalities such as message queuing and publish subscribe support, with push and pull capabilities.

Message passing represents a low-level form of distributed communication. In this structure, participants communicate by simply sending and receiving messages. The producer sends messages asynchronously through a communication channel (previously set up for communication). The consumer receives messages by listening synchronously on that channel. The producer and consumer are coupled in both time and space. Message passing is nowadays viewed as primitive to build complex interaction schemes and it is rarely used directly [16].

RPC is a widely used form of distributed interaction proposed in [10, 11, 126] for procedural languages, and in object-oriented contexts for remote method invocations.

It is, for example, used in Java Remote Method Invocation (RMI) [12, 126], Common Object Request Broker Architecture (CORBA) [13, 14], Microsoft Distributed Component Object Model (DCOM) [67]. In RPC and its derivatives, the producer performs a synchronous call, which is processed asynchronously by the consumer. RPC makes remote interactions appear almost the same way as local interactions.

In contrast to the RPC method that is invocation centric, **message queuing** [15] and **publisher/subscribe** are message centric communication schemes. Message queuing and publish/subscribe are tightly intertwined paradigms. **Message queuing** middleware buffers the produced messages in a queue while supporting, when required, reliable delivery, transactional and ordering guarantees [16]. In practice, based upon the messaging order, the queue can be specified as First-In-First-Out (FIFO), Last-In-First-Out (LIFO), limited FIFO, and limited LIFO. As the name suggests, FIFO/LIFO means the first/last arrived message will be served first while the limited ones introduce the priority concept, which means that the message with higher priority will be severed earlier than those with lower priority. Typically, the limited FIFO message queue is the most widely used queue model in MOM applications. The message queuing paradigm is used to support a point-to-point model where messages are addressed to recipients. This message queuing paradigm is suitable for the request/reply message exchange pattern. A message queue has various properties: private or shared, durable or transient, permanent or temporary. If a client application creates a message queue, it can select some important properties: name (if applications want to share a message queue, they need to agree on a queue name beforehand); durable (if the queue is declared as non-durable, the message will be lost when the server restarts); auto-delete (if specified, the server will delete the message queue when all clients have finished using it).

When a user application wants to transfer data to another application, it puts data into messages, and then puts the messages onto a queue (or publishes them with a topic).

Then there are four main ways the messages can be retrieved: Another application on this computer (or another part of the same application) retrieves the messages from the same queue where they were put. A queue manager is configured to send the messages through a channel over the network to a remote queue on a queue manager on another computer. An application on that computer retrieves the messages from the remote queue. An application on another computer pulls the messages across the network when it needs them. A queue manager sends the published message to a subscribe application.
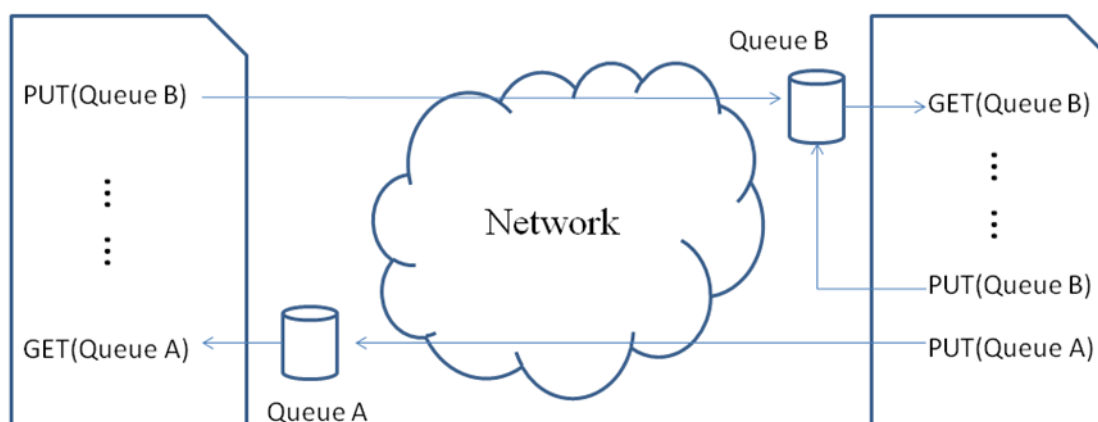
Figure 2-1Example of a Message Queuing system [66]

Figure 2-1 is an example of a Message Queue system. In a Message Queue system, users can have many queues and topics on one queue manager and they can also have more than one queue manager on one computer. A computer with a client installation with no queue manager is allowed as well. The client uses the queue manager on a server installation on another computer for messaging.

The **publish/subscribe** paradigm is used to support a many-to-many model that permits an efficient dissemination of messages across a distributed system [17]. Application clients in a publish/subscribe structure can be a publisher (the message source) or a subscriber (the message receiver). In publish/subscribe, the middleware keeps the registered interests of consumers according to the topic (for topic-based publisher/subscriber) or matching conditions defined by consumers (content-based

publisher/subscriber); and delivers a message to all consumers that match with their registered interests. The topic-based publish/subscribe variant represents a static schema which offers only limited expressiveness since the pre-defined external criterion the topic name cannot be changed. A content based publisher/subscriber classifies message according to the properties of the message itself. So a content-based publisher/subscriber could be more flexible [16].
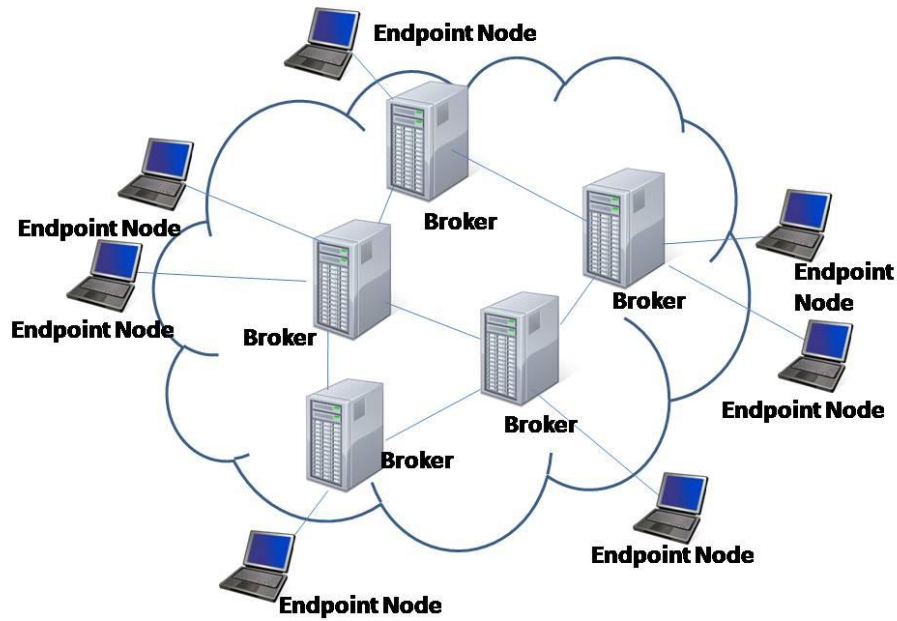
Figure 2-2 Publish-subscribe system model example

Figure 2-2 shows an example of a MOM system based on a publish/subscribe structure. Brokers are inter-connected through an overlay network and collectively provide the publish/subscribe messaging service. Each endpoint node, such as a sensor, an actuator or a processing element, is attached to a local broker [2]. There can be an arbitrary number of topics in the system. Each endpoint can publish and subscribe to one or many topics, while each broker can perform publish/subscribe matching, transport messages to local endpoints or neighbouring brokers, and optionally perform message mediation.

Using this style of interaction, the sender does not know the identity of the receivers:

it is the dispatcher, which is a function existing in the broker that identifies them dynamically. As a consequence, new components can join a federation, become immediately active, and cooperate with the others without requiring any reconfiguration of the architecture. Due to this advantage structure, MOMs could provide a service that allows content providers and consumers to concentrate on the production and consumption of transmitted information. The key advantage of the MOM architecture is that it reduces the number of point-to-point connections in a complex business critical information technology (IT) system [127].

2.2 Implementations of MOM Systems

A MOM system can be divided into two categories, one in topic based system and another is content based system. This research focuses on the topic-based publisher/subscriber paradigm, which is the most commonly used in enterprise applications. Examples on both the topic based and content based systems will be given.

2.2.1   The Java Message Service (JMS)

The Java message service is an application programming interface (API) provided by Sun Microsystems (now Oracle) as described in [24, 25, 26, 27]. In practice, the JMS framework defines the system as a set of non-implemented Java methods such as interfaces and abstract methods, where the specific implementations are up to a vendor.

The API defines Java interfaces for the publishers allowing them to generate and send messages to the JMS server. For the subscriber side, the defined Java interfaces consider the reception of these messages from the JMS server. The API provides abstract Java methods to control the message flow by various message filtering options. The JMS server itself, which represents the mediation server, is not specified

by the API and its implementation is up to a vendor, as well as the underlying communication mechanisms between publishers and subscribers. In this context the replication grade describes (defined below) the number of messages which have to be transmitted to the subscribers by the JMS server for a single published message. In this section, the JMS framework and the most important features provided by the JMS API are described.

The JMS is a wide-spread and frequently used middleware technology. Therefore, lots of systems are based on it, such as FioranMQ [28], TibcoEMS [29], WebSphereMQ [30, 68, 69, 70] and RabbitMQ [31]. The RabbitMQ software fully supports the Advanced Message Queuing Protocols (AMQP) framework while being based on the JMS framework. However, the content based systems are more complex and less popular than topic based system. So this research will mainly verify pure topic based system.

After discussing the state-of-the-art MOM systems, the following sections will introduce well-known MOM system protocols, part of which will be verified later in this thesis.

2.2.2 Apache Qpid

AMQP is one of the most popular protocols used in MOM systems. It is an open, royalty-free and unpatented networking protocol for MOMs. The AMQP Working Group aims to create a de-facto standard protocol for MOMs that allows the business applications to interact [21, 22, 23] encouraging the ideas of partnership and collaboration. One of the implementations of AMQP recommended in [21] is Apache Qpid (Open Source AMQP Messaging). In Qpid, there are procedures to react to broker failure and also to adding/deleting a publisher or subscriber and for matching topics.

An **exchange** accepts messages from a producer application and routes them to message queues according to prearranged criteria. These criteria are called 'bindings'. Exchanges are matching and routing engines. That is, they inspect messages and by using their binding tables, decide how to forward these messages to message queues. Exchanges never store messages. AMQP defines a number of standard exchange types, which cover the fundamental types of routing needed to do common message delivery. Exchanges may be durable, transient, or auto-deleted. Durable exchanges last until they are deleted. Transient exchanges last until the server shuts down. Auto-deleted exchanges last until they are no longer used [22, 23].

In the general case, an exchange examines a message's properties, its header fields, and its body content, and using this and possibly data from other sources, decides how to route the message. In the majority of simple cases, the exchange examines a single key field, which is called the '**routing key**'. The routing key is a virtual address that the exchange may use to decide how to route the message. For topic based publish/subscribe routing, the routing key is the topic hierarchy value. In more complex cases, the routing may be based on message header fields and/or the message body [22, 23].

The topic exchange process performs the following steps:

1. A message queue is bound to the exchange using a binding key, K.

2. A publisher sends to the exchange a message with the routing key R.

3. The message is passed to the all message queues where K matches R.

2.2.3 The Harmony MOM System-An Example of a Publish/Subscribe MOM System

When a topic-based publish/subscribe structure is put into practice, a programming abstraction is introduced to map individual topics to distinguish communication channels. The topic name is usually specified as an initialization argument. Every topic is viewed as an event service of its own, identified by a unique name. Usually the wildcards are also allowed in the topic name by most of the MOM systems. For example, in TIBCO Rendezvous [9], a topic name with wildcards, which offers the possibility to subscribe and publish to several topics whose names match a given set of keywords, like an entire subtree or a specific level in a hierarchy. IBM T. J. Watson Research Centre has developed a MOM system called Harmony [44] and this MOM system is also a topic based publish/subscribe paradigm. This thesis use Harmony system as an example to introduce publisher/subscriber structure based MOM system.

Publish/Subscribe Structure

In Harmony, it assumed that a set of brokers is known in advance, and the topology of the broker overlay is also known. This assumption is reasonable in many application scenarios because the broker deployment only changes at very coarse timescales (e.g. once in a few weeks).

Harmony assumes that the endpoint nodes in the system are clustered into many local domains, and there is one broker node inside each domain. As shown in Figure 2-2, each endpoint node, such as a sensor, a publisher, a subscriber, an actuator or a processing element, is attached to the local broker. There can be an arbitrary number of topics in the system, which can be defined either through administrative tools or dynamically using programming APIs. Each endpoint can publish and subscribe to one or multiple topics, while each broker can perform publish/subscribe matching,

transport messages to local endpoints by looking up the local subscription table or neighbouring brokers by looking up the remote subscription table, and optionally performing message mediation (e.g., format transformation)[2].

So communication is usually asynchronous and communicating endpoints/clients are decoupled in time (they do not have to be active at the same time), space (they do not need to know each other) and flow (the sending and receiving of messages does not block participants) [18, 19]. Clients/endpoints do not even need to know the existence of each other.

Adding or Deleting a Publisher / Subscriber in Harmony

In the Harmony model, each endpoint has a local broker. The endpoint can subscribe to any topic at any time and they can just send those subscriptions to the local broker. Each broker collects the subscriptions and maintains a so called 'local subscription table' to record which topics each local endpoint subscribes to. Then the brokers propagate the topics to other brokers. So, every broker knows other broker needs and maintains those information in a so called 'remote subscription table'. In this structure, adding or deleting publishers will not affect both the 'local subscription table' and the 'remote subscription table', so there is no need to have measures to deal with it. If a subscriber is added or deleted, the local broker will firstly update the 'local subscription table' and then check whether there is a change in the variety of topic. If there is a change in topic (e.g. subscribes to more or less topics), the local broker propagates the change to other brokers. Then other brokers update their 'remote subscription table' [2].

If a publisher publishes a topic, says T, the topic is firstly sent to the local broker. The local broker checks the local subscription table and sends T to those local endpoints who are interested in T. Then the local broker checks the remote subscription table to find all remote brokers that subscribe to T and sends T to them. As long as they

receive T, they pass T to their local endpoints who subscribed to T. At the end, the message eventually arrives at all subscribers of topic T in the system.

Detecting a Broker Failure and Monitoring the Link-State

Harmony uses similar methods to open shortest path first (OSPF) [71, 72]. Brokers in Harmony keep periodically advertising its link states, including the measured processing latency for each topic and the network latency to each of its neighbours. Through this simple neighbouring forwarding mechanism, every broker in this model maintains an entire network map with all the link states. Also there are monitoring agents for every broker. Those monitoring agents measure the processing and network latencies and periodically ping neighbouring brokers to gain the network latency. This Harmony model computes an Exponentially Weighted Moving Averaging (EWMA) [128] to avoid sudden spikes and drops in the measurements [2].

To detect the failure of brokers, Harmony assumes that if a neighbouring broker fails to reply to three consecutive pings, it is regarded as failed and the link latency is marked as ∞. The broker failure will be broadcasted to all the brokers and they will update their remote subscription table. Also the monitoring agents will keep track of the broker processing latency. If they find a high latency on a publish/subscribe matching or the queuing delay, they will arrange part of the publishers and the subscribers linked to the delayed broker to a new broker, in order to decrease the load. Also the new arrangement will be broadcasted to help the brokers to update their subscription tables [2].

2.3 Overlay Network Fast Recovery

Overlay network recovery exists in conjunction with recovery in the underlay. One of the most wildly used network routing protocols is OSPF and this will be looked at first.

2.3.1   IP Fast Recovery Scheme

In OSPF adjacent routers periodically exchange Hello messages to maintain the link adjacency. If a router does not receive a Hello message from its neighbour within a RouterDeadInterval (typically 40 seconds or 4 HelloIntervals), it assumes the link between itself and the neighbour to be down and generates a new Router LSA to reflect the changed topology. All such LSAs, generated by the routers affected by the failure, are flooded throughout the network and cause the routers in the network to undertake the shortest path first (SPF) calculation and update the next hop information in the forwarding table. Thus, the time required to recover from the failure consists of: (1) the failure detection time (2) LSA flooding time (3) the time to complete the new SPF calculations and update the forwarding tables. Goyal et al [105], focus on reducing the failure detection time, which is clearly the main component of the overall failure recovery time in OSPF-based networks. While the availability of link layer notifications can help achieve fast failure detection, such mechanisms are often not available. Hence, the routers use the Hello protocol to detect the loss of adjacency with a neighbour. However, there is a limit to which the HelloInterval can be safely reduced. As the HelloInterval becomes smaller, there is an increased chance that the network congestion will lead to loss of several consecutive Hello messages and thereby cause a false breakdown of the adjacency between routers even though the routers and the link between them are functioning perfectly well.

Goyal et al. [105] examine the network wide impact of reducing the HelloInterval in terms of number of false alarms under a realistic model of network congestion. They quantify the detrimental effect of these false alarms in terms of unnecessary SPF calculations done by the routers. They examine how the network topology influences the occurrence of false alarms. Finally, they evaluate how much faster detection of network failures helps in achieving faster recovery from these failures in the operation

of OSPF networks.

They found that when traffic in the network is heavy, a smaller HelloInterval may lead to fake of link failures [105, 129] (i.e. false positives). Conversely, if traffic is quite light and fast, the HelloInterval could be reduced to milliseconds, while the OSPF schema only allows a second to be the smallest time measurement. So, we propose an idea of adjusting the panic timer, which will increase robustness of a mission critical MOM system.

Goyal et al. [105] proposed the optimal value for the HelloInterval that will lead to fast failure detection in the network whilst keeping the occurrence of false alarms within acceptable limits. However, with the minimum size of a HelloInterval expressed in 'seconds', the minimum detection time will be 4 seconds. For a mission critical MOM application, a delay of 4 seconds will still be unacceptable.

Delay time in OSPF includes the failure detection time, event propagation time, Shortest Path First algorithm running time and the Routing Information Base (RIB) [130] or forwarding information base (FIB) [131] update time. Normally, the 'event propagation time' includes Link-state advertisement (LSA) generation delay, LSA reception delay, Processing Delay, and Packet Propagation Delay [132].

Processing Delay is the amount of time it takes the router to put the LSA on the outgoing flood lists. This delay could be significant if SPF process starts before flooding the LSA. SPF runtime is not the only contributor to the processing delay, but it's the one you have control over. If you configured SPF throttling to be fast enough (see next session) – the exact time varies but mainly the initial delays below than 40ms – it may happen so that SPF run occurs before the triggering LSA is flooded to neighbours. This will result in slower flooding process. For faster convergence, it is required that LSAs are always flooded prior to SPF run. ISIS process in Cisco IOS

[133] supports the command fast-flood, which ensures the LSPs are flooded ahead of running SPF, irrespective of the initial SPF delay. On the contrary, OSPF does not support this feature and the only option (at the moment) is properly tuning SPF runtime delays [101]. The Dijkstra's shortest path algorithm running time is bounded by $O\left(L+N\times\log\left(N\right)\right)$ where N is number of the nodes and L is the number of the links in a topology under consideration [122, 123]. For a 24 nodes network, the SPF running time delay is about 32ms [101]. There is more quantifiable experiment which has been done by Cisco researchers where they claimed the time required to do shortest path calculation: observed to be $0.00000247\times x^2 + 0.000978$ seconds on Cisco 3600 series routers. In this equation, x is the number of nodes in the topology [102].

After completing SPF computation, OSPF performs sequential RIB update to reflect the changed topology. The RIB updates are further propagated to the FIB table – based on the platform architecture this could be either centralized or distributed process. The RIB/FIB update process may contribute the most to the convergence time in the topologies with a large amount of prefixes, e.g. thousands or tens of thousands. In such networks, updating RIB and distributed FIB databases on line-cards may take considerable amount of time, such as at the order of tens if not hundreds of milliseconds (varies depending on the platform).

2.3.2   Resilient Routing Layers Algorithm

The authors in [110] have proposed an alternative solution for fast recovery from link failures called Resilient Routing Layers (RRL). RRL [134, 135, 136, 137] is based on the idea of building spanning sub topologies over the full network topology, and using these sub topologies to forward traffic in the case of a failure. Routing recovered traffic according to a sub topology may cause a high concentration of traffic on certain links, and hence a loss of some of the recovery gains due to link congestion

[111].

The authors of RRL proposed to organize a network topology in sub-topologies called routing layers. In each layer there are some nodes or areas that do not carry transit traffic, and the authors called these nodes or areas the safe nodes of this layer. Layers are constructed so that all nodes are present in each layer, and there exists a path between all node pairs in each layer. Each node should be safe in at least one layer to guarantee single node fault tolerance. There are numerous ways to construct the layers so that different protection properties are optimized [112, 113, 114].

This RRL schema ensures all node-pairs can communicate with each other in all layers, and also that safe nodes will not carry any transit traffic, only traffic originating and terminating in the safe nodes:

1) Links haven't been chosen as safe links, which are directly connected to a safe node, can carry all kinds of traffic originated and terminated anywhere.

2) Links which are chosen to be safe link can only be used as the first hop or last hop of the communication.

Traffic originated in a safe node can use a safe link as first hop, and that traffic terminated in a safe node can use a safe link as last hop towards the safe node.

Although, the RRL scheme saves memory by storing fewer routing tables, it does not guarantee the shortest paths are used, which can be critical for a time-sensitive overlay network. Normally one hop in an overlay network could be a relatively long way in the underlay network. The shorter a path is the better. More overlay hops may correspond to a much longer geographical distance in the underlying network. Therefore, this research proposes to perform all the time consuming calculations

before the overlay network is operational. Moreover, this research provides a novel idea of condensing all the pre-calculated routing tables into one super routing table. So this approach will save quite a lot of space for a broker.

2.4 A Summary of the State-of-the-Art on MOM Systems and Network Fast Recovery

Existing MOMs fall into one of two categories: enterprise messaging systems and real-time messaging systems. Intended to address traditional business needs, enterprise messaging systems provide message delivery assurance and transactional guarantees. They usually implement the JMS standard [24] and can transport messages over a wide area across multiple domains. However, they do not proactively manage messaging performance. As such, applications cannot predict or depend on when messages will arrive at the destination. Real-time messaging systems, on the other hand, offer quality of service (QoS) assurance by allocating resources and scheduling messages based on application-specific QoS objectives. They often conform to the Data Distribution Service (DDS) standard [73].

AMQP is continually updated and version 1.0 (is a standard accepted by Organization for the Advancement of Structured Information Standards (OASIS) [138], which is a global consortium which drives the development, convergence, and adoption of e-business and web service standards. Cisco, JPMorgan Chase, Red Hat together with other 5 companies join in the development of AMQP uses cases. Winter Green's 2008 annual report indicated that IBM WebSphere holds 64% of users in message oriented middleware market [98]. Nowadays, AMQP is chasing up IBM WebSphere and lots of successful use cases based on AMQP such as Qpid and Rabbit are taking over the market shares from WebSphere. One of the implementations of AMQP recommended in its site is Apache Qpid (Open Source AMQP Messaging). In Qpid, there are measures to react the broker failure and for adding/deleting a publisher/subscriber and

also a procedure of matching topics.

The message queuing paradigm is used to support a point-to-point model where messages are addressed to recipients. This message queuing paradigm is suitable for the request/reply message exchange pattern and it is more reliable than the publish/subscribe paradigm. So, many message services of banks and other finical enterprises are based on the message queuing paradigm. However, message queuing paradigm does not have the characteristic of scalability. The publish/subscribe paradigm is used to support a many-to-many model that permits an efficient dissemination of messages across a distributed system. This kind of many-to-many model has higher scalability for large systems.

Message queuing middleware buffers the produced messages in a queue while supporting, when required, reliable delivery, transactional and ordering guarantees. A publish/subscribe middleware keeps the registered interests of consumers according to topics (for topic-based publish/subscribe) or matching conditions defined by consumers (for content-based publish/subscribe); and delivers a message to all consumers that match with their registered interests. Message queuing systems usually integrate publish/subscribe support inherently. Message Oriented Middleware adopts the message centric approaches and usually employs both message queuing and publish/subscribe communication schemes. Nowadays, since IBM WebSphere MQ has the longest history, it has already integrated message broker service into its products in its newest version – version 7.0. Most of the rest commercial MOM applications are employing both message queuing and message broker paradigms.

Quite a few proposals have been made to handle link failures as locally as possible, by only doing routing updates in a limited number of routers near the failure [106]. These proposals suggested that if two equal cost paths exist toward a destination, traffic can safely be switched from one to the other in case of a failure. Iselt et al. [107]

suggest using Multi-Protocol Label Switching (MPLS) tunnels to make sure that there are always are two equal cost paths toward a destination at every node. There are other proposals that consider things globally as well, and try to guarantee that there always is a valid routing entry for a given destination after a single failure. With O2-routing [108], the routing tables are set up so that there is always two valid next hops toward a destination. However, they do not guarantee the shortest path. Conversely, Hansen et al. [109] proposed that for a network topology, there could be several layers with different safe nodes and safe links. Most nodes and links will only appear in one layer. As long as the failure is confined to a particular layer, the cost of all the links within this layer will be set to infinite and routing is performed via other layers. Therefore, each node just needs to store routing tables for the normal situation and for a single complete failure situation in each layer.

The two schemes listed in section 2.3 are among the most popular IP network fast recovery measurements. Although, the IP fast recovery schema applied on OSPF network, reduces the failure detection time to a certain degree [105], the time for regenerating routing tables under a failure situation still remaining the same. The resilient routing layers algorithms, such as RRL, can save the time for regenerating new routing tables. However the RRL algorithm will not guarantee the shortest path any more when a failure happened and a pre-stored routing table is used. Moreover, the total number of routing tables a node needs to store is depending on the number of safe layers there are for a given topology. There is a trade-off between the number of safe layers and average path length. The more layers there are, the more likely a path is the shortest path [111]. So, this thesis proposes a novel algorithm for a pre-calculated routing table to reduce the time of regenerating new routing tables and condense all routing tables into one super routing table.

3 Logic Checking and Model Checking Tools

3.1 Model Checking

Model checking is an automatic technique for verifying finite-state reactive systems, such as sequential circuit designs and communication protocols. Specifications are expressed in temporal logic, and a reactive system can be modelled as a state transition graph. An efficient search procedure is used to determine whether or not the state transition graph satisfies the specifications [32, 42, 43, 141].

There are several advantages in using the model checking technique. The most important one is that the procedure is highly automatic. Typically, the user provides a high level representation of the model and the specification to be checked. The model checker will either terminate with the answer 'true', indicating that the model satisfies the specification, or it will give a counter example execution that shows why and where the formula failed by extensive simulations.

To be more specific, model checking works starting with a model described by the user, and discovers whether hypotheses asserted by the user are valid on the model. If they are not, it can produce counter examples, consisting of execution traces. Furthermore, model checking focuses explicitly on temporal properties and the temporal evolution of systems.

Model checking is based on temporal logic. The idea of temporal logic is that a formula is not statically true or false in a model, as it is in propositional and predicate logic. Instead, the models of temporal logic contain several states and a formula can be true in some states and false in others.

Therefore, there is no static truth anymore and this one is replaced by a dynamic one, in which the formulas may change their truth values as the system is changing from state to state. In model checking, the models M are transition systems and the properties φ are formulas in temporal logic [142]. To verify that a system satisfies a property, the following three things should be done:

1. Model the system using the description language of a model checker, arriving at a model M;

2. Code the property using the specification language of the model checker, resulting in a temporal logic formula φ;

3. Run the model checker with inputs M and φ.

The model checker outputs the answer 'yes' if M |= φ (means the model M matching the formula φ), and 'no' otherwise; in the latter case, most model checkers also produce a trace of system behaviour which causes this failure. The value of the counterexample helps to detect the potential problems of the system and the collision or violation of the protocol.

However, earlier model checkers could only check small circuits and protocols [33, 34, 35]. They were not able to deal with large systems because of the state explosion problem [36]. With the help of an Ordered Binary Decision Diagram (OBDD) the capability of a model checker can be increased dramatically [37]. OBDD can effectively reduce useless states. So the number of nodes in the OBDDs that must be constructed no longer depends on the actual number of states or the size of the transition relations. With this great breakthrough, a number of major companies including Intel, Motorola, Fujitsu and AT&T have started using symbolic model checkers to verify actual circuits and protocols [32].

The original model checking algorithm together with the new representation (using OBDD) for translation relation, is called Symbolic Model Checking.

3.1.1   New Symbolic Model Checker—NuSMV

NuSMV is a symbolic model checker developed by ITC-IRST and UniTN with the collaboration of CMU and UniGE [38].The NuSMV project aims at the development of a state-of-the-art model checker that is robust, open and customizable; it can be applied in technology transfer projects and can be used as research tool in different domains [38, 41, 143, 144].

NuSMV provides a language for describing a model and it directly checks the validity of LTL which will be introduced in Chapter 3.2.1 (and also CTL) formulas on those models. NuSMV takes as input a text consisting of a program describing a model and some specifications (temporal logic formulas). It produces as output either the word 'true' if the specifications hold, or a trace showing why the specification is false for the model represented by the program. A NuSMV program consists one or more modules. Just like in the programming language C, or Java, one of the modules must be called main. Modules can declare variables and assign values to them. Assignments usually give the initial value of a variable and its next value as an expression in terms of the current values of variables. This expression can be non-deterministic (denoted by several expressions in traces or no assignment at all). Non-determinism is used to model the environment and for abstraction.

The following are two examples of NuSMV modules to illustrate the basic structure of a NuSMV file.

```
MODULE main

VAR
        request : Boolean;
        states    : {ready, busy};
ASSIGN
        init(state) := ready;
        next(state) := case
                state = ready & request = TRUE    : busy;
                TRUE                                :{ready,busy};
                esac;
LTLSPEC
        G(request -> F & state = busy)
```

The program has two variables, **request** of type Boolean and **states,** which is the enumeration type {ready, busy}. The initial and subsequent values of the variable **request** are not determined within this program; this conservatively models these values by an external environment. This under specification of **request** implies that the value of variable **states** is partially determined: initially, it is ready; and it becomes busy whenever **request** is true. If **request** is false, can be represented as '$\neg$ req' or $\overline{req}$ (in Figure 3-1). In the later chapter (chapter 4) will use '$\overline{\mathrm{var}iable}$' to indicate 'not' or 'false'. The next value of **states** is not determined. Note that 'case 1' in the program signifies the default case, and that *case statements* are evaluated from the top down. If several expressions to the left of a ':' are true, then the command corresponding to the first, top-most true expression will be executed. The program therefore denotes the transition system as shown in Figure 3-1:
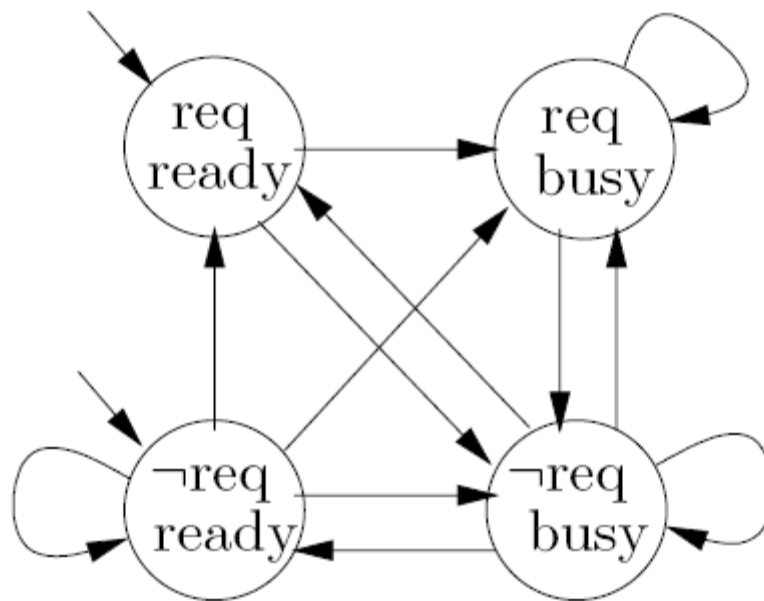
Figure 3-1 The Model Corresponding to the NuSMV Program in the Text.

There are four possible states, each one corresponding to a possible value of the two binary variables (as '**request**' is a Boolean value which has two options 'TRUE' or 'FALSE' and '**states**' is a state variable which also has two options 'ready' or 'busy'). Note that 'busy' is a shorthand for 'state=busy' and 'req' for '**request** is true.' Since variable **request** functions as a genuine environment in this model, the program and the transition system are non-deterministic: i.e., the 'next state' is not uniquely defined. Any state transition based on the behaviour of **states** comes in a pair: to a successor state where '**request**' is false, or true, respectively. For example, the state '¬req, busy' has four states it can move to (itself and three others).

LTL (which will be introduced later on) specifications are introduced by the keyword LTLSPEC and are simply LTL formulas.

The code and Figure 3-2 show a small example that models a broker's working state as 'working' when the broker=TRUE and as 'failed' when the broker=FALSE.

```
MODULE main
VAR
        broker :Boolean;
ASSIGN
        init(broker) := FALSE;
        next(broker) := !broker;
```

The program has one variable named 'broker' of type Boolean. The initial value of 'broker' is FALSE and subsequent values of 'broker' will be the reverse of current value.
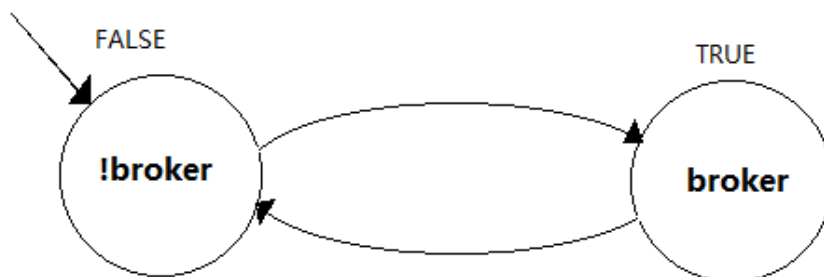
Figure 3-2Broker State Transition Diagram

This 'broker' module could be a useful example of 'broker state' and will be used later on in Chapter 5.

3.2 **Temporal Logic**

Typically, temporal logic is used for reasoning about things that change over time [145,146]. This kind of logic was developed because of the limitation of predicate

logic. Predicate logic, also called the first-order-logic, can deal quiet satisfactorily with sentence components like 'not', 'and', 'or', and 'if…then'. The predicate logic assumes that a true statement is always true and a false statement is always false. However, when it comes to the modifiers like 'eventually…', 'among…' and 'only…' first-order-logic cannot represent those modifiers with 'not', 'and', 'or' or 'if... then' [39]. With the help of temporal logic, conditions that change over time can be represented, e.g. 'I will be hungry eventually '.

There are several types of temporal logic, like the Computation Tree Logic (CTL), the Alternating-time Logic (ATL) [48], Timed CTL [49, 50], Linear-time Logic (LTL), and the 'Until'-only fragment of Linear-time Logic (μTL) to name just a few. These kinds of temporal logic could be divided into two categories, one is called 'linear-time logic', which thinks time as a set of paths and a path is a sequence of time instances. The other is called 'branching-time logic', which represents time as a tree, rooted at the present moment and branching out into the future. Branching-time logic seems to make the non-deterministic nature of the future more explicit. Linear-time temporal logic and computation tree logic will be used in this thesis.

3.2.1 The LTL

The Linear-time temporal logic is a temporal logic with connectives that allow one to refer to the future. It models time as a sequence of states, extending infinitely into the future. This sequence of states is sometimes called a computation path, or just a path. In general, the future is not determined, and every state has a unique successor, any one of which might be the 'actual' future that is realized. [39]

In LTL, the connectives X, F, G, U, R and W are called 'temporal connectives'. X means 'neXt state', F means 'some Future state', and G means 'all future states' (Globally). The next three, U, R and W are called 'Until', 'Release' and 'Weak-until'. Following are some commonly used LTL formulas.

X p:this formula will be true if p is true in the second state of the path:
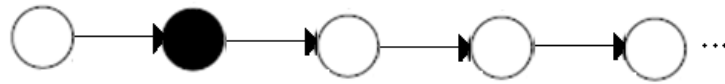
Figure 3-3 LTL Formula 'X p'

Figures of the rest formulas will be found in Appendix A.

F p: this formula will be true if p eventually become true.

G p: this formula will be true if p is true at every state.

p U q: this formula will be true if p is always true until q is true.

3.2.2   The CTL

The Computation Tree Logic, CTL for short, is branching-time logic, meaning that its model of time is a tree-like structure in which the future is not determined; there are different paths in the future, any one of which might be the 'actual' path that is realized [33].

Note that each of the CTL temporal connectives is a pair of symbols. The first part of a pair is composed of A or E. A means 'along All paths' (inevitably) and E means 'along at least (there Exist) one path' (possibly). The second part of the pair is composed of X, F, G, or U, meaning 'neXt state', 'some Future state', 'all future states' (Globally) and Until, respectively. In CTL, pairs of symbols like EU are indivisible. Notice that AU and EU are binary. The symbols X, F, G and U cannot occur without being preceded by an A or/ and E; similarly, every A and E must have X, or F, or G or U to accompany it.

Following are examples of CTL formulas. The topmost state satisfies a given formula if the black states satisfy p and the shadow states satisfy formula q.

AG p: in all paths, p is true all the time

AF p: in every path, p will finally be true
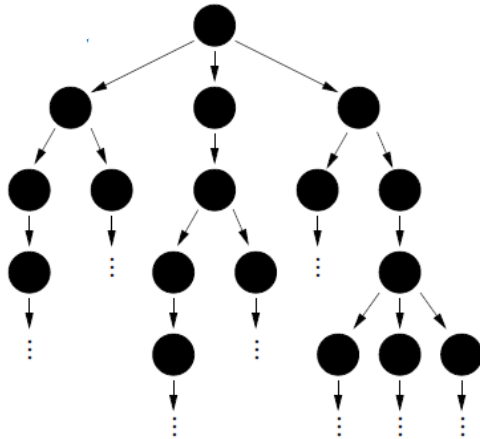
Figure 3-4 CTL Formula 'AG p' [32]   Figure 3-5 CTL Formula 'AF p' [32]
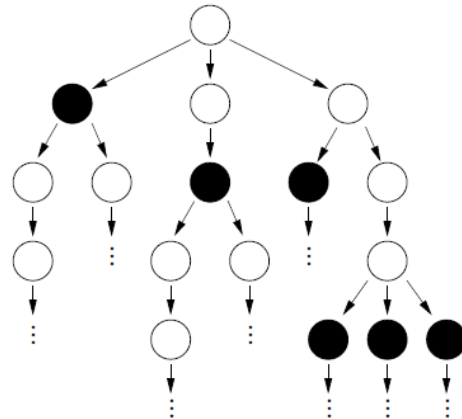
Figures of the remaining formulas can be found in Appendix A.

AX p: for all paths, p will be true in next state

p AU q: for all paths, p is true until q is true

EG p: there is at least one path in which p will be true all the time

EF p: there is at least one path in which p will finally be true

EX p: there is at least one path in which p is true in next state

p EU q: there is at least one path in which p is true until q is true

3.3 **Concluding Remarks**

A publish/subscribe based MOM system could be hard to test and reason about. Using a model checker to formally model a MOM system could facilitate the research. Model checking is an automatic technique for verifying finite-state reactive systems. It is an attractive alternative to formal reasoning a MOM system. The user provides a high level representation of the model and the specification written with temporal logic formula to be checked. A model checker finds bugs in systems by exploring all possible execution states of a finite state model to search for violations of some desired property. The model checker will either terminate with the answer 'true' or provide a counter example. This chapter gives a brief introduction of one model checker NuSMV.

In this research two temporal logics are used: Linear Tree Logic and Computation Tree Logic. These two temporal logics meet the need of the verifications in this research. In Appendix B, a comparison between LTL and CTL is presented to justify the necessity of using both of them in this research.

4 Verification Languages and Techniques

While model checking is a powerful technique, one of the stumbling blocks to using it is the creation of appropriate finite state models for the systems being checked. Since most software systems have infinite-states, one must first find suitable abstractions that reduce the system to a finite state model, without eliminating the class of errors that one wants to pinpoint.

A model should capture the basic properties that must be considered in order to verify the behavioural specification (the so called atomic propositions). Furthermore, to make the verification simpler, the model should abstract away all the details that have no effect on the correctness with respect to the specification.

4.1 End-to-End Verification of Network Reachability

In the past few years, many researchers have attempted to address various challenges in network configuration. In [74, 75, 76], different techniques are presented to identify configuration conflicts between firewall and IPSec devices. Approaches for analyzing routing configuration using static or formal analysis [60, 61, 62, 77, 78], and on-line debugging [8] was proposed. Top-down configuration approaches have also been proposed [79]. E. Al-Shaer et al. [7] presented a novel approach to model the global end-to-end behaviour of access control configuration of an entire network. The model represents the network as a finite state machine where the packet header and location determine the state. And also the packet header information determines the whole transitions for a packet. Inside the packet header, there is information on the packet source IP address, packet destination address, and packet current location. The policies for each device are also being modelled. It encodes the semantics of access control policies with Boolean functions. It uses a commercial model checking tool called ConfigChecker combined with CTL to test all future and past states of this

packet in the network and also verifies the network reachability and other security requirements. The research presented in this thesis extends the end-to-end verification proposed by E. Al-Shaer et al. and provides a way to verify a whole publisher/subscriber MOM system. The verification includes the checking of the reachability of different topics, the rules for regulating the working of the system, and checking the configuration and reconfiguration after a failure. The novelty of this work is the creation and the optimization of a symbolic model checker that abstracts the end-to-end network configuration behaviour and using it to verify reachability and security properties.

The paper [7] shows that more than 62% of network failures today are due to network misconfiguration [8]. Also the paper indicates that with the increasing scale of network, it would be a numerous work to configure the network devices in isolation and it would be not possible to manually analyse the cooperation among those devises. However, people cannot foresee whether the entire network devices will perform correctly. If the isolated configured devices could not work along with others, network administrators have to wait until the network fails and explores the problem. Then the network administrator could reconfigure the whole system. However, they still will not know in advance whether this reconfiguration is successful or not.

Paper [7] reports a method to solve this unpredictable configuration problem by using a NuSMV model checker to verify the reachability and the security of a network.

4.1.1   Model Checking Example

A model that represents a network as a finite state machine was proposed by [7]. In this model the header and location information of the message which belonging to a topic determine the state. Each device in the network can then be modelled by describing how it changes a packet that is currently located at the device. For example, a firewall might remove the packet from the network or it might allow it to move on

to the device on the other side of the firewall. A router might change the location of the packet but leave all the header information of the packet unchanged. A device performing network address translation might change the location of the packet as well as some of the IP header information. A hub might copy the same packet to multiple new locations. The behaviour of each of these devices can be described by a list of rules. Each rule has a condition and an action. The rule condition can be described using a Boolean formula over the bits of the state. If the packet at the device matches (satisfies) a rule condition, then the appropriate action is taken. As described above, the action could involve changing the packet location as well as changing IP header information. In all cases, however, the change can be described by a Boolean formula over the bits of the state. Sometimes the new values are constant (completely determined by the rule itself), and sometimes they may depend on the values of some of the bits in the current state. In either case, a transition relation can be constructed as a relation or characteristic function over two copies of the state bits. An assignment to the bits/variables in the transition relation yields true if the packet described by the first copy of the bits will be transformed into a packet described by the second copy of the bits when it is at the device in question.

32-bits are used for the source IP address, the destination IP address, and the device currently processing the packet, with 16-bit source port number and destination port numbers in the basic network model of [7]. In order to illustrate this approach, an example containing only 2 bits for the source IP, destination IP, and location IP, and 1 bit for the source port and destination port is given. The formulas use s_1, d_1, l_1 for the higher order bit in the source IP address, the destination IP address, and the location of interested IP address respectively. s_0, d_0, l_0 are used for the lower order bits. $s'_0, d'_0, s'_1, d'_1...$ represent the values of the bits in the next state with the same interpretation as the unprimed versions above.

The message header information determines the transitions for a message. For a message, the broker which publishes the message is the source, the broker which subscribes to this message is the destination. Inside the message header there is information giving the message source IP address, the destination address, and its current location. The rules for each device are also modelled in [7], but only set a router model as an example.

A router might change the location of the packet but leave all the header information of the packet unchanged. The rule condition can be described using a Boolean formula over the bits of the state (the parameters of the characteristic function σ). As described above, the action could involve changing the packet location as well as changing IP header information. In all cases, however, the change can be described by a Boolean formula over the bits of the state. Sometimes the new values are constant (completely determined by the rule itself), and sometimes they may depend on the values of some of the bits in the current state. In either case, a transition relation can be constructed as a relation or characteristic function over two copies of the state bits. An assignment to the bits/variables in the transition relation yields true if the packet described by the first copy of the bits will be transformed into a packet described by the second copy of the bits when it is at the device in question. Some examples of how real devices can be encoded in this way should help to illustrate the technique. However, to keep the formulas simple, the examples will contain only 2 bits for the source IP, destination IP, and location IP addresses, and 1 bit for the source port and destination port. Real examples are similar, but with larger (32-bit or 16-bit) fields.

4.1.2   A Router Model

To describe a router model, assume a router with IP address 3 sends all messages in different topics destined for IP addresses 1 and 0 (source) to IP address 0 (next hop), while all other topics are sent to IP address 2 as a default gateway.

The policy described above can be formulated as:

$$(\overline{d_1} \wedge \overline{l_1'} \wedge \overline{l_0'}) \vee (d_1 \wedge l_1' \wedge l_0') \qquad (4\text{-}1)$$

Then all packets that satisfy the formula (4-1) will be forwarded to the outgoing interface. This formula (4-1) shows two possible situations at a broker. The first one is $\overline{d_1} \wedge \overline{l_1'} \wedge \overline{l_0'}$. In order to let this part to be 'TRUE', the destination $\overline{d_1}$ should be true which means d_0 could be 0 or 1 but d_1 could just be 0. Hence the destination would be 01 or 00. Also $\overline{l_1'} \wedge \overline{l_0'}$ should be true, and that is to say $\overline{l_1'}$ and $\overline{l_0'}$ are true at the same time ($l_1' = 0$, $l_0' = 0$). So for any topic which destination is 01 or 00, the next location will be 00. Otherwise, let the other situation $d_1 \wedge l_1' \wedge l_0'$ to be true. Then it indicates that when the destination is 10 or 11, the next location will be 11.

No packet header information changes. This condition can be formulated as:

$$\underset{i \in \{0,1,p\}}{\bigwedge} \left(s_i' \Leftrightarrow s_i \right) \wedge \underset{i \in \{0,1,p\}}{\bigwedge} \left(d_i' \Leftrightarrow d_i \right) \qquad (4\text{-}2)$$

This formula could be separated into two parts, divided by ' $\wedge$ ' and the two should be true at the same time then this formula can be true. The first part $\underset{i \in \{0,1,p\}}{\bigwedge} \left(s_i' \Leftrightarrow s_i \right)$

means the next states of s_0, s_1 and s_p will be themselves with no change. It is the same as the later part. That is to say, this formula indicates the packet header information will never change.

Finally, this transformation only takes place when the packet is currently at the router. This condition can be formulated as:

$$l_1 \wedge \overline{l_0} \qquad\qquad (4\text{-}3)$$

This formula means the packet is in the location 10 (the router's IP).

For a simple model with several components (e.g. routers, sensors or other end nodes), using less than 5 Boolean variables to represent the IP addresses is feasible. However, in the real word, the IP addresses should be represented by 32 Boolean variables. If one designs a model using the IPv4 address structure, it will need up to 232 different states, which may lead to a state explosion. In [12], the authors' propose a basic model that has five key identity variables; two of them (ports and port ids) are 16 bits long and the rest (IPsource, IPdestination and location) that are all 32 bits. In this model, there are thus 2128 possible states. In order to get rid of the state explosion problem, this research proposes another way to build an MOM overlay network model. Since the publish/subscribe system is an overlay network, the number of brokers is much less than the number of routers in its under layer network. In MOM models, it does not use 32-bit IP addresses. This research uses the nature numbers of IDs to handle the number of brokers. However, this research potentially needs policies for each topic and so the set of policies can be large.

Although model checking is a powerful technique, creation of appropriate finite state models for the systems being checked is still one of the stumbling blocks to using it. An important challenge of this research is how to build feasible models of publish/subscribe based MOM overlay networks to reduce the system to a finite state model, without eliminating the class of errors that wanted to be checked.

4.2 Model Checked Publish/Subscribe Systems

Publish/subscribe systems are hard to reason about and to test. In particular, given the inherent non-determinism in the order of event receipt, delays in event delivery, and

variability in the timing of event announcements, the number of possible system executions becomes combinational large. There have been several attempts to develop formal foundations for specifying and reasoning about publish/subscribe systems [80, 81, 82, 83, 84], and this area remains a fertile one for formal verification. Unfortunately, existing notations and methods are difficult to use in practice by non-formalists, and require considerable proof machinery to carry out.

The key feature of [90] is a reusable, parameterized state machine model that captures publish/subscribe run-time event management and dispatch policy. Generation of models for specific publish/subscribe systems is then handled by a translation tool that accepts as input a set of publish/subscribe component descriptions together with a set of publish/subscribe properties, and maps them into the framework where they can be checked using off-the-shelf model checking tools.

To further reduce costs of using a model checker,[90] also provides a tool that translates publish/subscribe application component descriptions (specified in an XML-like input language) and properties into the a lower-level form where they can be checked using standard model checking tools.

Model checking of software systems [147, 148] is an extremely active area of research at present. Much of this effort aims to make model checking easy for practitioners to use, for example by allowing the input language to be a standard programming language (e.g., [85, 86]), and by providing higher-level languages for specifying properties to check (e.g., [87]). However, none of these efforts has focused on exploiting the regularities of one particular component integration architecture, as this research does for publish-subscribe systems.

Finally, there have been several previous efforts at providing formal, checkable models for software architectures. Some of these even use model checkers (e.g., [88,

89]) to check properties of event-based systems. However, none has been tailored to the specific needs of publish/subscribe systems development.

Authors of [90] believe that there are two main stumbling blocks to creating a state model for a publish/subscribe system that is suitable for model checking. One is the construction of finite-state approximations for each of the component behaviours (methods). Authors in [90] have adopted the following restrictions: all data has a finite range; the event alphabet and the set of components and bindings are fixed at runtime; there is a specified limit on the size of the event queue and on the length of event announcement history maintained by the dispatcher; and there is a limit on the size of invocation queues (pending method invocations as a result of event delivery).

A second problem is the construction of the run-time apparatus that glues the components together, mediating their interaction via event announcements. This involves developing state machines that maintain pending event queues, enforce dispatch regimes (correctly modelling non-deterministic aspects of the dispatch), and providing shared variable access. In principle, this part of the modelling process could be done afresh for each system using brute force. Unfortunately modelling publish/subscribe systems involves a fair amount of such runtime state machinery, and is not trivial to get right. Moreover, once built, it is hard to experiment with alternative run time mechanisms。

Model checking for Publish-Subscribe architectures [90] provides a set of pluggable modules that allow the modeller to choose one possible configuration out of a set of possible choices. However, available models are far from fully capturing the different characteristics of existing Publish-Subscribe systems. For instance, application components cannot change their subscriptions at run-time, and the message dispatching mechanism is only characterized in terms of delivery policy (asynchronous, synchronous, immediate or delayed). The same approach is extended

in [91] by adding more expressive events, dynamic delivery policies and dynamic event method bindings. These features are then used in [99] to implement a transformational framework that, starting from a dedicated programming language, produces Extensible Markup Language (XML) data for model checking as well as executable artefacts for testing. The resulting approach only deals with the specification of different delivery policies depending on the overall state of the model, and still does not capture fine-grained guarantees such as real-time constraints.

4.3 Alternative Model Checking Language: Bogor

Techniques applicable to specific Publish-Subscribe middleware systems have been considered in [92, 93, 94, 95]. Beek et al. [92] concentrate on the addition of a Publish-Subscribe notification service to an existing groupware protocol, and reports on the improvements in user awareness of the development status achieved in this way. Caporuscio et al. [93] develop a compositional reasoning technique based on an assume-guarantee methodology. The methodology is applied on a specific case study, i.e., on developing a file sharing system on top of the Siena Publish-Subscribe middleware [96]. The proposals in [94, 95] describe an approach similar to ours based on an early version of Bogor. The authors focus on modelling the real-time features of the CORBA Communication Model (CCM). Their time model is certainly more detailed than ours. All the aforementioned approaches lose generality in that they do not allow users to customize the checking engine to model Publish-Subscribe systems that provide various guarantees.

The efficiency and correctness of a system is heavily dependent on its design rules and protocols. Model checking techniques are extensively used to find vulnerabilities in rules and protocols. So system designers can fix or overcome vulnerabilities before the system suffers any losses. For example, Kerberos is one of the most popular protocols for providing a secure communication over network. It was considered the most secure protocol. However, the authors in [40] found vulnerability in this

protocol by using a model checking technique.

L.Baresi et al. [40] provide a novel approach based on Bogor for the accurate verification of applications based on Publish-Subscribe infrastructures.

The paper reported by L.Baresi et al. could be divided into three parts. The first part tries to adopt standard model checking techniques to verify the application behaviour. The second part introduced Bogor, which is a state-of-the-art extensible model checker implemented in Java. The last part addresses an application for fire monitoring in a tunnel.

Since a variety of applications build on Public-Subscribe infrastructures and they may have different requirements on the under layer, L.Baresi et al. focus on changing different verifications to meet the different requirements from the upper layer. The other useful paper [7] pays much more attention to the access control verification for network devices. Both papers provide the scenarios all of which applies to highly dynamic and flexible environments. So a reconfiguration of system is considered in both papers.

4.4 Concluding Remarks

This chapter focuses on comparing the state-of-the-art with this research. Since configuring a network is time consuming and error prone, some researchers [7] suggest verifications for the checking of the reachability of different topics, the rules for regulating the working of the system, and checking the configuration and reconfiguration after a failure. The novelty of the work in [7] is the creation and the optimization of a symbolic model checker that abstracts the end-to-end network configuration behaviour and using it to verify reachability and security properties. However, those researches did not consider the state explosion problem that the model could suffer. The research in this thesis improves the way of building an

overlay network model to decrease the total number of states. And this could better suit a MOM overlay network modelling.

Besides the state explosion problem, finding a suitable way to build an abstract model for publish/subscribe based MOM overlay system is also complex. Researchers in Carnegie Mellon [90] proposed to build several reusable components for different requirements of a MOM system and implement a transformational framework that, starting from a dedicated programming language, produces XML data for model checking as well as executable artefacts for testing. However, their model in [90] is too general to represent the detail of MOM systems in different scenarios. The research presented in this thesis focuses on building a more specific model for verifying the configuration and reconfiguration for a MOM system with reusable components. Also this research extends the model with time restriction in the future, which researchers in Carnegie Mellon failed to cover in their work.

For a communication system, the efficiency and correctness is heavily dependent on its design rules and protocols. System designers can fix or overcome vulnerabilities before the system suffers any losses by using model checking techniques to find vulnerabilities in rules and protocols. Researchers in [40] adapted SPIN, a model checking tools [149, 150], into a new one named 'Bogor' and added Bogor into a Java development environment. However, similar to SPIN, Bogor does not support CTL and time restrictions. So the research in this thesis uses NuSMV other than Bogor to support CTL and the time restrictions that are needed in verification of MOMs.

5 Formal Verification Model Checker for MOM Overlay Networks

This chapter is organised as follows. In Section 5.1, we first describe the use the language NuSMV, introduced in Chapter 3, as a means of manually creating a formal model of a 3-node MOM overlay network, which we can then use to test the validity of the configuration. A further example of a 6-node MOM overlay network is provided in Section 5.2. To our knowledge, the use of this language for formal checking MOM overlay network configurations has never be considered before in published literature. Having demonstrated that the language provides a useful means of developing a formal model that can be used for verification purposes we then propose in Section 5.3, a new tool for the automatic generation of models of realistically sized MOM systems. We validate this tool on the 6-node network already considered, to confirm its satisfactory functioning. We also show how the novel sub-path detection mechanism can be used to speed up the verification process by reducing the magnitude of the path searching in Section 错误！未找到引用源。. Next, in Section 0, we use the tool to provide a formal model of a large network, something that would not be possible to do reliably by-hand. We also examine its performance. Finally in Section 5.5, we summarise the chapter.

In our context, model checking is an automatic technique for verifying finite-state reactive systems. It is an attractive [90] alternative to formal reasoning a MOM system. The user provides a high level representation of the model and a specification written using temporal logic to be checked. The model checker finds bugs in a system by exploring all possible execution states of a finite state model to search for violations of some desired property. The model checker will either terminate with the answer 'true' (introduced in Chapter 3) or provide a counter example. While model checking is a powerful technique, one of the problems to using it is the creation of

appropriate finite state models for the systems being checked. This research must first find suitable abstractions that reduce a system to a finite state model, without eliminating the class of errors that this research wants to identify, should they exist.

As mentioned in Chapter 3 the authors of [90] believe that there are two main barriers to creating a state model for a publish / subscribe (P/S) system that is suitable for model checking. One is the construction of finite-state approximations for each of the component behaviours (methods). The second problem is the construction of the run-time apparatus that 'glues' the components together, mediating their interaction via event announcements. In this research, these problems have been addressed to a certain degree. This chapter designs and implements suitable abstract models of MOM overlay networks. Since this formal verification language and temporal logic are not easy to comprehend, particularly as the scale of a MOM system increasing, it becomes too difficult for humans to manually create the models themselves. This chapter designs a P/S platform code generator with a Graphical User Interface (GUI) front end, from which a formal verification model can be automatically generated as shown in Figure 5-1. The only information users need to provide is the routing table(s) and topic table. The routing table(s) can be obtained through automated means, such as Dijkstra's algorithm [221]. Later, Chapter 6 will solve the second problem, mentioned in [90]; that of run time challenges. By forecasting possible failures (such as link failures and node failures) and pre-calculating routing tables for those failure scenarios, this research can solve many run time issues that could arise.
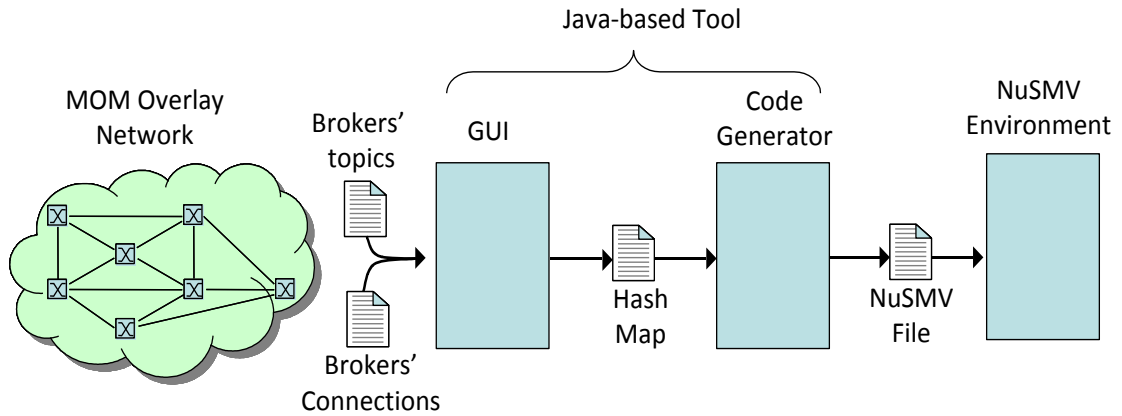
Figure 5-1 Formal Verification Model Checker Functional Layout

In order to simplify and consolidate the concepts described in Chapter 3 regarding MOM systems, model checkers, temporal logic and NuSMV, examples verifying of the correctness and validity of a MOM system's configuration are provided, as well as when the MOM reconfigures after suffering a failure. The first model is composed of three brokers and it illustrates the situation of verifying the configuration of publishers and subscribers if they can link to a broker before and after a change (broker failure and recovery) has happened. The second model uses six brokers, and it focuses more on topic reachability verification. After describing our automated P/S platform code generator for creating NuSMV models, we illustrate its usefulness by examining a third model derived from a realistic commercial mission critical MOM system. The code generator generates models mainly for the purpose of topic reachability verification and loop detection of overlay networks.

5.1 Three-Broker Model

5.1.1 Modelling a Three-Broker Example

In this model, there are three brokers and each of them takes responsibility of one publisher and one subscriber. To keep the model simple, the model does not incorporate routing. Therefore, the publisher publishes a topic to a subscriber, and

both are connected to the same broker. Consequentially, the subscriber only subscribes to a topic of a publisher connected to the same broker. A diagram of this model is provided in Figure 5-2.
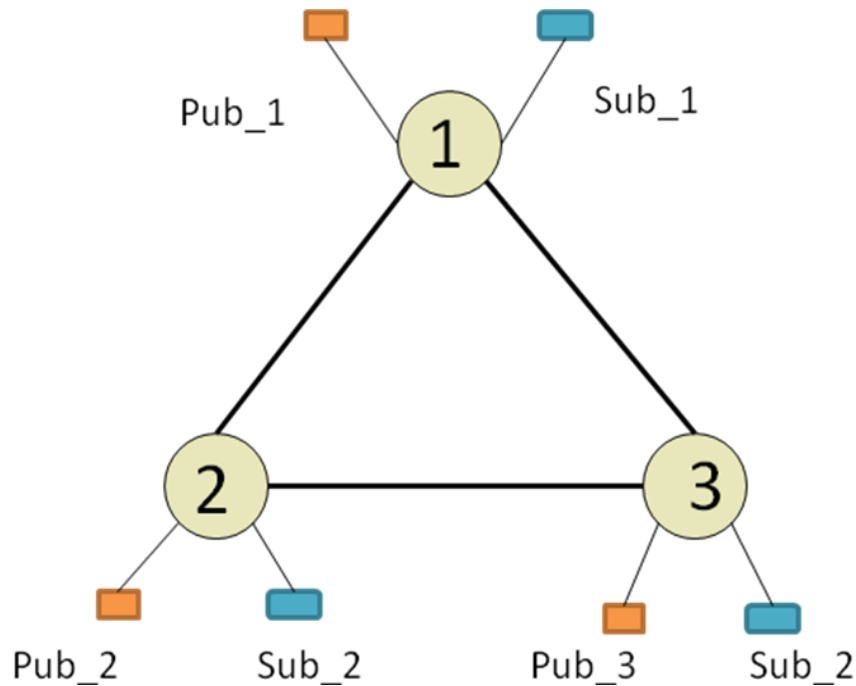
Figure 5-2 Three-Broker Connection Diagram

The Three Brokers Model contains four states (normal, broker 1 fails, broker 2 fails, broker 3 fails, since we only consider single broker failure here). They are named S_0, in which all three brokers are functioning correctly, S_1, in which broker1 fails, S_2 in which broker2 fails and S_3, in which broker3 fails. This simple model does not portray the situation when two or more brokers fail. Depending on different conditions, all of these states could be interchangeable.

The Three Brokers Model in NuSMV contains three types of modules: the broker, the publisher and the subscriber. In the broker module there is a Boolean variable used to control the working state of the broker. In the publisher module, there is one integer

variable that indicates the topic this publisher publishes and three Boolean variables indicating the broker this publisher is linked to. In the subscriber module, an integer variable identifies the topic this subscriber is subscribed to. The following is the NuSMV code for the broker module:

```
MODULE broker
    VAR
        state :Boolean;
MODULE publisher
    VAR
        pub_topic : 1..3;
        pub_bro1 :Boolean;
        pub_bro2 :Boolean;
        pub_bro3 :Boolean;
 MODULE subscriber
    VAR
        sub_topic : 1..3;
        sub_bro1 :Boolean;
        sub_bro2 :Boolean;
        sub_bro3 :Boolean;
```

The reconfiguration process in this model is described as follows. If broker1 fails, all the publishers and subscribers linked to it will be taken over by broker2. Similarly, broker3 takes over the responsibility of broker2's publishers and subscribers if broker2 fails. Finally, broker1 will take over the responsibility of broker3 if the broker3 fails. Every time there is a state transition, it will be verified by the LTL or CTL statements in NuSMV. In the initial state S_0, Pub_1 (publisher1) and Sub_1 (subscriber1) are linked to Bro_1 (broker1), while Pub_2 (publisher2) and Sub_2 (subscriber2) are linked to Bro_2 (broker2), and Pub_3 (publisher3) and Sub_3 (subscriber3) are linked to Bro_3 (broker3).

The state transition diagram is shown in Figure 5-3.
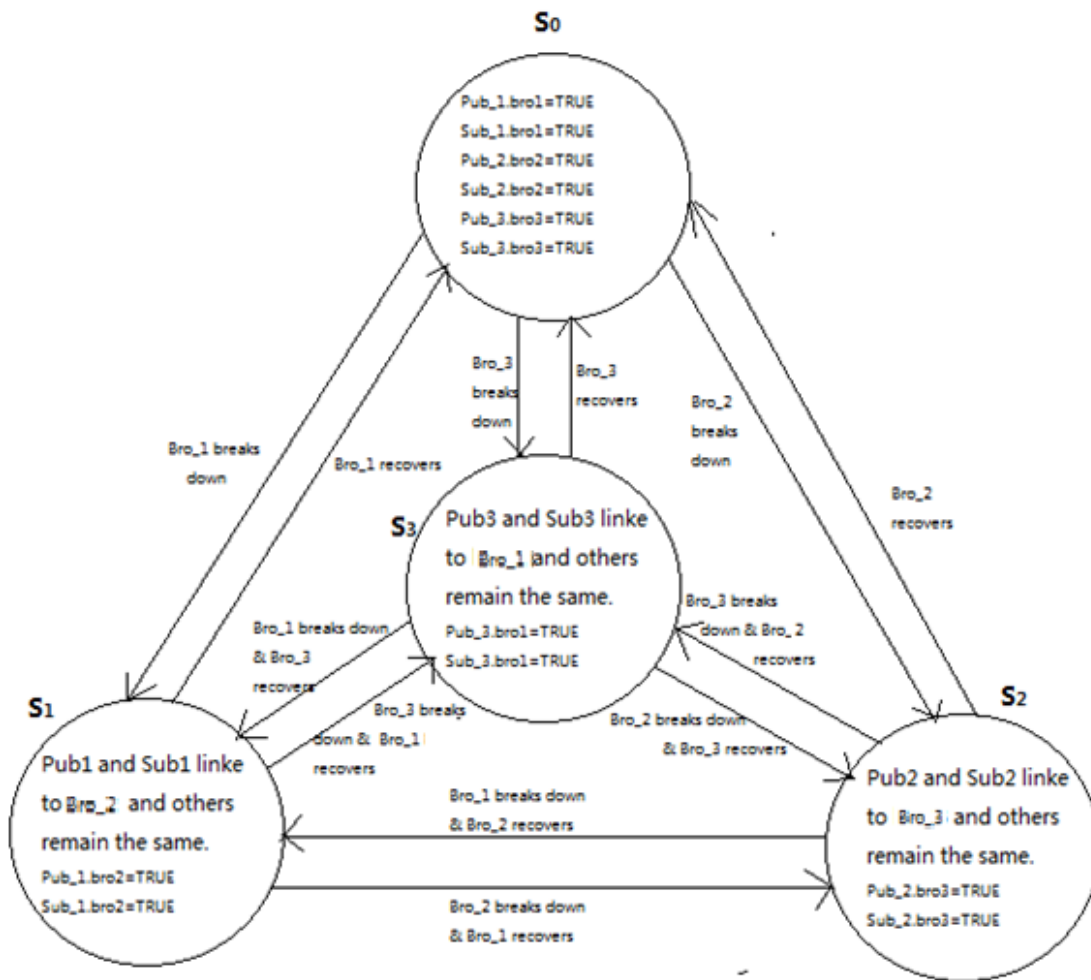
Figure 5-3 Three-Broker Model State Transition Diagram

Figure 5-3 shows the conditions that trigger state interchanges. For clarity, in each state only the changed variables are shown. The state changes from S_0 to S_1 only when broker1 fails (Bro_1=FALSE) and publisher1 and subscriber1 linked to broker1 are now linked to broker2. Thus, Pub_1.bro1=FALSE, Sub_1.bro1=FALSE, Pub_1.bro2=TRUE and Sub_1.bro2=TRUE. The situation will be the same when the state changes from S_0 to S_2 or S_3. In addition, Figure 5-3 shows the condition and the variables' new values when the state changes from S_1, S_2 and S_3 to the initial state S_0. Things are a little different when the states are not reached from the unique state S_0. When the state changes from S_1 to S_3, the conditions are broker3 fails

(Bro_3=FALSE) while broker1 recovers (Bro_1=TRUE). Publisher1 and subscriber1 are taken over by broker3 while broker1 is not alive. However, when broker1 recovers and is working again the situation changes to (Pub_1.bro3=FALSE, Pub_1.bro2=TRUE, Sub_1.bro3=FALSE and Sub_1.bro2=TRUE). If broker3 fails, publisher3 and subscriber3 will be attached to broker1 (Pub_3.bro3=FALSE, Pub_3.bro1=TRUE, Sub_3.bro3=FALSE and Sub_3.bro1=TRUE). The inverse situation can be easily derived. Other non-initial state interchanges are quite similar to the transition from S_1 to S_3.

5.1.2   Model Verification

In the three-broker model, this section verifies three points. Firstly, it verifies that if one broker fails, all the publishers and subscribers linked to it can successfully find a new broker to link to. This condition can be verified by the use of CTL statements. Assuming broker1 fails, for example:

*G !broker1.state -> X (!publisher1.pub_bro1 & !subscriber1.sub_bro1 &
publisher1.pub_bro2 & subscriber1.sub_bro2).   (5-1)*

This logical statement means that when broker1 fails, publisher1 and subscriber1 will be re-linked to broker2. Figure 5-4 shows the results of this verification. The information following '***' is just about the version of NuSMV and the copyright declaration which has nothing to do with the entire model and its verification. The salient information is the result of the specification.

Figure 5-4 Verification Output for Publisher and Subscriber Reconfiguration to Another Broker

Subsequently, we represent screen shots such as Figure 5-4 as the following block and ignore NuSMV version information.

```
Input:
E:\NuSMV\bin>NuSMV three_broker_new.smv
Output
--specification ( G ! broker1.state -> X <<< ! publisher1.pub_bro1 & ! subscriber1.sub_bro1)
& publisher1.pub_bro1) & subscriber1.sub_bro2)) is true
```

Secondly, the following conditions must be true: every publisher has at least one subscriber to its topic and the vice versa. For an example,' subscriber2 has at least one publisher for the topic that subscriber2 wants to subscribe to'. This could be represented in an LTL statement as:

LTLSPEC

(subscriber1.sub_topic=publisher1.pub_topic/subscriber1.sub_topic=publisher2.pub_topic | subscriber1.sub_topic=publisher3.pub_topic) (5-2) the following block

65

shows the results of this verification are true in this model.

```
Input:
E:\NuSMV\bin>NuSMV three_broker_new.smv
Output
--specification
(subscriber1.sub_topic=publisher1.pub_topic|subscriber1.sub_topic=publisher2.pub_topic
|subscriber1.sub_topic=publisher3.pub_topic)        is true
```

Thirdly, the following condition must be hold when a broker returns to operational status: all the publishers and subscribers, which were linked to this broker before it failed, are now re-linked to this broker again. For example, assume broker 1 firstly fails and then returns to normal functioning. The assumption needs to be verified whether the publisher1 and subscriber 1 are re-linked to broker 1in the new state. So the operator 'Y', which refers to the previous states, can be used. This could be indicated in the LTL statement:

LTLSPEC

G (Y !broker1.state & broker1.state) -> X (publisher1.pub_bro1 & subscriber1.sub_bro1 & !publisher1.pub_bro2 & !subscriber1.sub_bro2)   (5-3)

错误！未找到引用源。 shows the results of this verification.

```
Input:
E:\NuSMV\bin>NuSMV three_broker_new.smv
Output
--specification G (Y !broker1.state & broker1.state) -> X (publisher1.pub_bro1 &
subscriber1.sub_bro1 & !publisher1.pub_bro2 & !subscriber1.sub_bro2)              is true
```

5.2 Six-Broker Model

This section uses a simplified model to perform the model checking of a publisher / subscriber based system. As an illustrative example we consider six brokers and each of them has a unique 4-byte IP address. Each broker has a number of publishers and subscribers linked to it.

As shown in Figure 5-5, the address of Broker1 is '2.0.0.1'. Broker1 has a publisher who publishes topic A and a subscriber to topic C. The address of Broker2 is '2.0.1.1' and it only has a subscriber which is interested in topic A. The address of Broker3 is '2.0.0.2' and it has a publisher of topic B and a subscriber to topic D. The address of Broker4 is '1.0.1.2'. Broker4 only has a subscriber that subscribes to topic B. The address of Broker5 is '1.0.0.1' and it has a publisher of topic C and a subscriber of topic A. The address of Broker6 is '1.0.0.2' and it has a publisher to topic D and a subscriber to topic C. To simplify this six brokers model, only one broker failure is considered at a time. So there would be seven different scenarios for this model. However, they are similar to each other. Therefore, this section only shows two of them. In the first scenario, all brokers are working correctly and in the second scenario one of the brokers fails.

5.2.1 Scenraio1 – Normal Operation

In Figure 5-5, all the brokers, publishers and subscribers' information is shown. In Scenario 1, all the brokers are functioning well and their connections or routing paths are showed in Table 5-1.
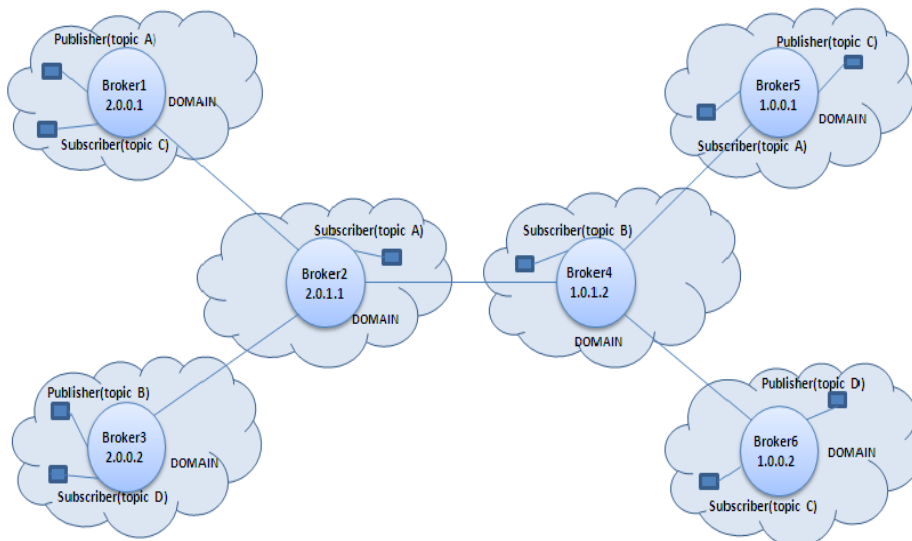
Figure 5-5 Normal Network Layout for the Six-Broker Model Example

The '*' (star sign) means that any number from 0 to 2 could appear in that part of the address:

Loc	Src.	Dest	Loc'	Src'	Dest'
2.0.0.*	*	*	2.0.1.1	Src	Dest
2.0.1.1	2.0.*.*	1.0.*.*	1.0.1.2	Src	Dest
1.0.1.2	2.0.*.*	1.0.*.*	1.0.*.*	Src	Dest
2.0.1.1	2.0.*.*	2.0.*.*	2.0.*.*	Src	Dest
1.0.0.*	*	*	1.0.1.2	Src	Dest
1.0.1.2	1.0.*.*	2.0.*.*	2.0.1.1	Src	Dest
2.0.1.1	1.0.*.*	2.0.*.*	2.0.*.*	Src	Dest
1.0.1.2	1.0.0.*	1.0.*.*	1.0.*.*	Src	Dest

Table 5-1 Routing Table for Scenario 1

In Table 5-1, the first column shows the current location of a message. The second column shows the address of the broker where the message is originated (broker's source address). The third column shows the address of the broker for which domain the message is destined (broker's destination address). The fourth column gives the next hop of a message. The fifth and sixth columns mean that after this hop the source and destination brokers will not change.

Topic	Publisher(broker IP Address)	Subscriber(broker IP Address)
A	B1(2.0.0.1)	B2 (2.0.1.1)
		B5 (1.0.0.1)
B	B3 (2.0.0.2)	B4 (1.0.1.2)
C	B5 (1.0.0.1)	B1 (2.0.0.1)
		B6 (1.0.0.2)
D	B6 (1.0.0.1)	B3 (2.0.0.2)

Table 5-2 Topics in Scenario 1

After the topology of the publish/subscribe system is constructed in the routing table, it is necessary to describe all the existing topics of the publish/subscribe system and inform the source (publisher) and destination (subscriber) of every topic. In this way, the reachability of the path that every message topic traverses in the system can be verified.

For example, based on Table 5-2, it is seen that Broker3 (2.0.0.2) publishes topic B and Broker4 (1.0.0.1) subscribes to this topic. Table 5-1 clearly shows that a message located at 2.0.0.2 (Broker3) whose destination address is 1.0.0.1 (Broker4) will traverse the path from 2.0.0.2 (Broker3) to 2.0.1.1 (Broker2) and finally to 1.0.1.2 (Broker4).

5.2.2   Scenario2 – Broker 1 Failure
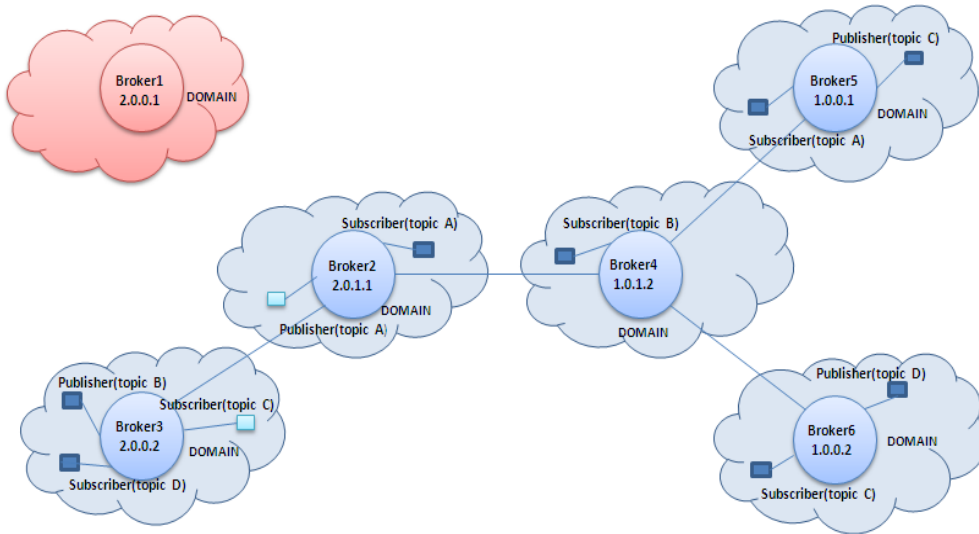
Figure 5-6 The Layout of the Six Brokers' Model When Broker1 Fails

In Scenario 2, Broker1 fails. The publisher attached to Broker1 is relocated to Broker2 and the subscriber is relocated to Broker3. As a result, the new routing and topic tables change to those shown in Table 5-3 and Table 5-4, respectively.

Loc	Src	Dest	Loc'	Src'	Dest'
2.0.0.2	*	*	2.0.1.1	Src	Dest
2.0.1.1	2.0.*.*	1.0.*.*	1.0.1.2	Src	Dest
1.0.1.2	2.0.*.*	1.0.*.*	1.0.*.*	Src	Dest
2.0.1.1	2.0.1.1	2.0.0.2	2.0.0.2	Src	Dest
1.0.0.*	*	*	1.0.1.2	Src	Dest
1.0.1.2	1.0.*.*	2.0.*.*	2.0.1.1	Src	Dest
2.0.1.1	1.0.*.*	2.0.*.*	2.0.*.*	Src	Dest
1.0.1.2	1.0.0.*	1.0.*.*	1.0.*.*	Src	Dest

Table 5-3 Routing Table for Scenario2 - Broker1 Fails

Table 5-4 presents the topics table for scenario2. It shows every topic with its publisher(s) and subscriber(s). This is used in future as input information for building the six-broker model.

Topic	Publisher(broker IP Address)	Subscriber(broker IP Address)
A	B2(2.0.1.1)	B2 (2.0.1.1)
		B5 (1.0.0.1)
B	B3 (2.0.0.2)	B4 (1.0.1.2)
C	B5 (1.0.0.1)	B3 (2.0.0.2)
		B6 (1.0.0.2)
D	B6 (1.0.0.1)	B3 (2.0.0.2)

Table 5-4 Topics Table for Scenario2 - Broker1 Fails

There are seven scenarios for single broker failure in this model. Scenario 1 in which all the brokers are working; Scenario 2 in which only broker one fails, Scenario 3 in which only broker two fails, Scenario 4 in which only broker three fails, Scenario 5 in which only broker four fails, Scenario 6 in which only broker five fails, Scenario 7 in which only broker six fails.

In NuSMV, there is one module for the broker, which contains a Boolean variable to indicate the working state of the broker. In the MAIN module (just like the main method in a Java programming language), there are four integer variables to indicate the source IP address and the same for the destination IP address. In addition, there is an integer variable used to represent the current location that a packet is in and one integer variable used to indicate the destination broker number. After building the entire routing table in NuSMV, the six brokers model can verify the reachability from a certain broker to the others in the seven different scenarios.

5.2.3   Model Generation with NuSMV

After having all the necessary information, which is the routing table and topic table, a framework of a NuSMV file meeting the rules of the NuSMV Language can be built.

Firstly, all the variables used in the NuSMV program needs to be defined with their

types like 'Boolean' or 'integer'. In this example, there are source address variable, destination address variable, current location address variable, broker address variable and so on.

```
MODULE main
VAR
s0 : 0..2;
s1 : 0..2;
s2 : 0..2;
s3 : 0..2;

d0 : 0..2;
d1 : 0..2;
d2 : 0..2;
d3 : 0..2;

loc : 0..6;

loc_des : 0..6;
```

Secondly, it is necessary to give all the variables' initial values and also the values in the next state. The source and the destination value will not change in all the states until this verification is done and new source and destination addresses will be put in by the user. The initial value of the current location address variable will be the source address and it changes according to the routing table. Following the part of the current location' initial value and values in the next state:

The initial value is given as:

```
init(loc) :=
case
s0=1 & s1=0 & s2=0 & s3=2 : 1;
s0=1 & s1=1 & s2=0 & s3=2 : 2;
s0=2 & s1=0 & s2=0 & s3=2 : 3;
s0=2 & s1=1 & s2=0 & s3=1 : 4;
s0=1 & s1=0 & s2=0 & s3=1 : 5;
s0=2 & s1=0 & s2=0 & s3=1 : 6;
TRUE                      : 0;
esac;
```

The next state value is as follows:

```
next(loc) :=
case
  loc= 1 &loc_des= 2 : 2;
  loc= 1 &loc_des= 3 : 2;
  loc= 1 &loc_des= 4 : 2;
  loc= 1 &loc_des= 5 : 2;
  loc= 1 &loc_des= 6 : 2;
  loc= 2 &loc_des= 1 : 1;
  loc= 2 &loc_des= 3 : 3;
  loc= 2 &loc_des= 4 : 4;
  loc= 2 &loc_des= 5 : 4;
  loc= 2 &loc_des= 6 : 4;
  loc= 3 &loc_des= 1 : 2;
  loc= 3 &loc_des= 2 : 2;
  loc= 3 &loc_des= 4 : 2;
  loc= 3 &loc_des= 5 : 2;
  loc= 3 &loc_des= 6 : 2;
  loc= 4 &loc_des= 1 : 2;
  loc= 4 &loc_des= 2 : 2;
  loc= 4 &loc_des= 3 : 2;
  loc= 4 &loc_des= 5 : 5;
  loc= 4 &loc_des= 6 : 6;
  loc= 5 &loc_des= 1 : 4;
  loc= 5 &loc_des= 2 : 4;
  loc= 5 &loc_des= 3 : 4;
  loc= 5 &loc_des= 4 : 4;
  loc= 5 &loc_des= 6 : 4;
  loc= 6 &loc_des= 1 : 4;
  loc= 6 &loc_des= 2 : 4;
  loc= 6 &loc_des= 3 : 4;
  loc= 6 &loc_des= 4 : 4;
  loc= 6 &loc_des= 5 : 4;
 TRUE                  : 0;
esac;
```

After the model been constructed, verification can take place. A user can change different CTL or LTL verification statements through the text file storing the specification statements. Then the statements are written into the end part of the NuSMV file. The following is an example of a specification for 6-broekr model:

SPEC

$EF(loc=loc\_des)$ (5-4)

Besides manually input the testing source and the destination, later on we build an Code generator (introduced in section 5.3) which could automatically generate test pairs of topics' source and destination by matching the topic table. For example, publisher1 publishes topic A to broker1 while subscriber3 linked to broker3 and subscriber5 linked to broker5 all subscribe to topic. Therefore, broker1 will be the source and broker3 and broker5 will be the destinations. Thus two CTL statements are automatically generated:

$EF (source=1 \ \& \ loc=3);$ (5-5)

$EF (source=1 \ \& \ loc=5);$ (5-6)

The first statement means that 'if there finally is a state that its source is broker1 and the current location is broker3'. That is performed a searched that there is a path from broker1 to broker3. Things are the same with the second statement. In either case, an executable NuSMV file generated.

5.2.4   Reachability Verification for the Six-Broker Model

As a publish/subscribe system grows in scale, there will be more brokers, publishers and subscribers. Every time one publish/subscribe system adds a new device or a failure happened, the model will change. So the reconfiguration of this system should be verified.

Based on the model built in NuSMV for the Six Brokers Model, the reachability of each topic from its source broker to its destination broker could be easily verified. In

this model, it is necessary to make sure that a packet from a certain publisher will reach all the required subscribers. The reachability condition can be represented in a CTL statement as:

EF loc (this is the location of current IP address) = loc_des (this is the destination IP address).   (5-7)

Where loc is the location of the current IP address; loc_des is the destination IP address. Formula 5-7 indicates that finally there will be a state where the current location of a message coincides with its destination address. This means that eventually the message will arrive at its destination.

The Topic table lists all the possible sources and destinations of each topic. Therefore, our publish/subscribe (P/S) tester goes through this table and tests all the sources and destinations topic by topic. If there are a large numbers of topics, the tester will take a long time to test all the sources and destinations entries. However, the path of a topic could be a sub-path of another already successfully tested path, consequently this sub-path do not need to be tested again. An algorithm that extracts all the maximum paths could be applied to the routing and topic tables and only those paths need to be verified, decreasing the testing time.

```
Input:
NuSMV > go
NuSMV > pick_state –r
NuSMV > print_current_state –v
Output:
Current state is 1.1
s0 = 1
s1 = 0
s2 = 0
s3 = 2
d0 =2
d1 =0
d2 =0
```

```
d3 =1
loc= 1
loc_des = 6
broker1.state = TRUE
broker2.state = TRUE
broker3.state = TRUE
broker4.state = TRUE
broker5.state = TRUE
broker6.state = TRUE
y=6
```

The above block shows the initial state of this six brokers' model. In this state, the source is 2.0.0.1 and the destination is 1.0.0.2; the current location is exactly the source broker- the broker1 and its destination broker is the broker6; all the brokers are functioning correctly. As it is shown in the above block, there is a variable named 'y' which can varies from 0 to 6. By using this variable, this system can model the brokers' failure. In other words, a broker will fail as long as this broker's id number equals the value of y. For example, if y=0, that means all the brokers are functionally well. In the above block, y=6, which means that in next state broker6 will be failed. The routing table is based on the scenario where broker 6 is out of order.

```
Input:
NuSMV six_broker.smv
Output:
--specification EF loc = loc_des is true
```

The above block shows the output of the verification statement 'EF (loc=loc_des)' in this six brokers' model. Moreover, the value is 'true', which means that message could be successfully sent from broker1 to broker6.

5.2.5   Validating the Reachability Verification in NuSMV

In order to validate the correctness of the reachability verification, this section designs an experiment to compare the simulation results with the result observed directly. Based on the routing table 5-3 and topic table 5-4, there is topic A published by

publisher attached to broker 1 and two brokers (broker 2 and broker 5) subscribed this topic. If letting broker 4 cannot seen broker 5 and every package at broker 4 will be sent to broker 6, this topic A's reachability (from broker 1 to broker 5) will not be possible. Following is the output of topic A's reachability from broker 1 to broker 5.

```
Input:
NuSMV six_broker.smv
Output:
--specification EF loc = loc_des is false
--as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
->State: 1.1 <-
s0 = 1
s1 = 0
s2 = 0
s3 = 2
d0 =1
d1 =0
d2 =0
d3 =1
loc= 1
loc_des = 5
broker1.state = TRUE
broker2.state = TRUE
broker3.state = TRUE
broker4.state = TRUE
broker5.state = TRUE
broker6.state = TRUE
y=0
```

The above block indicates that the topic source is broker 1 and destination is broker 5, but this topic cannot be sent to its destination. As long as a failure is known, further tracing can be done to see where the problem is.

```
input:
NuSMV > simulate –r 4
Output:
```

```
******    Simulation Starting From State 1.1    ******


Input:
NuSMV > show_traces −t
Output:
There is 1 trace currently available.


Input:
NuSMV > show_traves −v
Output:
<!-- ############### Trace number: 1 ############### -->
Trace Description: Simulation Trace
Trace Type: Simulation
->State: 1.1 <-
s0 = 1
s1 = 0
s2 = 0
s3 = 2
d0 =1
d1 =0
d2 =0
d3 =1
loc= 1
loc_des = 5
broker1.state = TRUE
broker2.state = TRUE
broker3.state = TRUE
broker4.state = TRUE
broker5.state = TRUE
broker6.state = TRUE
y=0
->State: 1.2 <-
s0 = 1
s1 = 0
s2 = 0
s3 = 2
d0 =1
d1 =0
d2 =0
d3 =1
loc= 2
loc_des = 5
broker1.state = TRUE
broker2.state = TRUE
```

broker3.state = TRUE

broker4.state = TRUE

broker5.state = TRUE

broker6.state = TRUE

y=0

->State: 1.3 <-

s0 = 1

s1 = 0

s2 = 0

s3 = 2

d0 =1

d1 =0

d2 =0

d3 =1

loc= 4

loc_des = 5

broker1.state = TRUE

broker2.state = TRUE

broker3.state = TRUE

broker4.state = TRUE

broker5.state = TRUE

broker6.state = TRUE

y=0

->State: 1.4 <-

s0 = 1

s1 = 0

s2 = 0

s3 = 2

d0 =1

d1 =0

d2 =0

d3 =1

loc= 6

loc_des = 5

broker1.state = TRUE

broker2.state = TRUE

broker3.state = TRUE

broker4.state = TRUE

broker5.state = TRUE

broker6.state = TRUE

y=0

->State: 1.5 <-

s0 = 1

s1 = 0

```
      s2 = 0
      s3 = 2
      d0 =1
      d1 =0
      d2 =0
      d3 =1
      loc= 6
      loc_des = 5
      broker1.state = TRUE
      broker2.state = TRUE
      broker3.state = TRUE
      broker4.state = TRUE
      broker5.state = TRUE
      broker6.state = TRUE
      y=0
```

The above block indicates the path of this topic trying to go through from broker 1 to broker 5. However, after this topic package arrives at Broker 6 it remains there and cannot reach Broker 5. This output is the same as the observed result. Modelling a Realistic MOM System

In previous sections of this chapter, we examine two illustrative models for a publish/subscribe based MOM. However, those two models are far too simple to represent a realistic MOM. They are used to illustrate the main approaches of modelling, verifying MOM systems by the means of using a model checker. In this chapter, a realistic MOM, an extended Harmony System [44] used in the USA, is modelled.
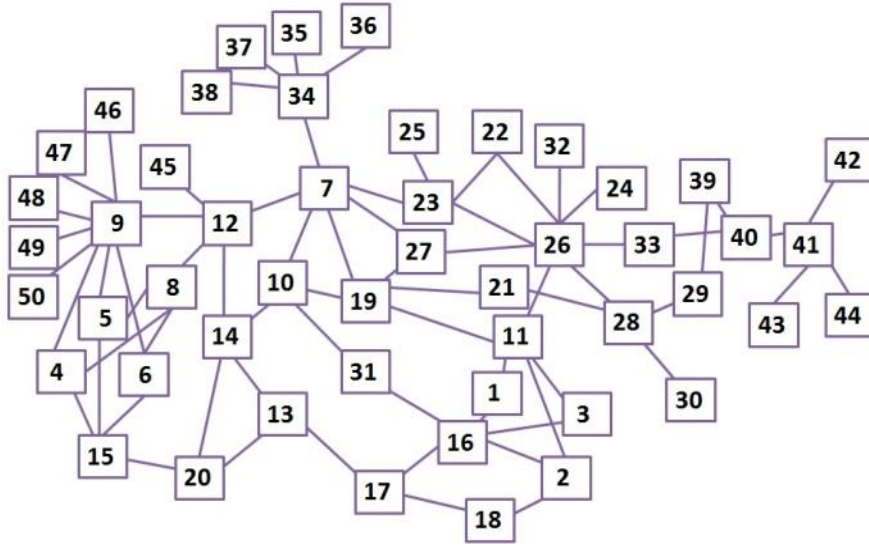
Figure 5-7 Realistic Commercial MOM Overlay Network

As shown in Figure 5-7, there are fifty brokers in total. Each broker has a set of topics (either publishing or subscribing). With the number of brokers growing, manually generating the correspondent routing table is extremely complex and error prone. However, the link information of each implemented broker is easier for user to collect. This research designed a graphical user interface (GUI) to collect the link table of the brokers and the topic table. The information collected is automatically translated into a hashmap. In the implemented tool, the link information of the brokers along with the selection of a specific routing algorithm (e.g. shortest path) are used to automatically generate the routing table of the MOM overlay network. The implemented NuSMV code generator reads the hashmap and generates a full NuSMV MOM model with all the specifications for the verification.

The following sections describe the whole processes of building and verifying a MOM system model in NuSMV.

5.2.6 The MOM Model in NuSMV

As mentioned in the beginning, the key task for constructing a NuSMV model is to find a suitable finite state model to replace the original system. We model the entire

MOM overlay network as a finite state machine, while each state is defined by a different broker ID. The state transition of the overlay network is determined by the topics and the routing protocol in the overlay network. The routing protocol in this model is the shortest path first. A topic starts from its source broker and follows its shorted path route until it reaches its destination broker.

Each of the three state variables for framing the NuSMV model (*src* , *dest* and *loc*) has the set of broker IDs (e.g. 1,2,…,49,50) as domain. So there are 1.25×10^5 possible states combinations for our model. All the variables are given initial values and rules for their next states transition. The initial values of *loc* is the initial value of *src* as a topic's initial current location is its source broker.

In the CTL specification the initial values for *src* and *dest* keep the same throughout. The rules for the next state transition of *loc* follow the matrix R (depending on the current value of *loc* and the value of *dest*) and are similar to Formula 5-11.

For example, following Figure 5-10, if Broker 1 has a topic about the 'weather' and Broker 13 subscribes to this topic, then the state transition diagram is shown in Figure 5-8.
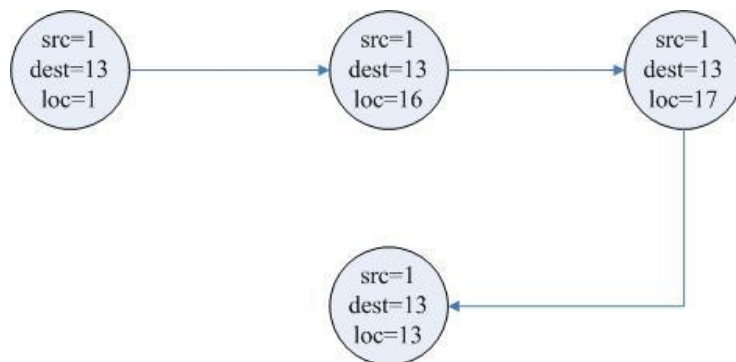
Figure 5-8 State Transition Diagram Segment (One Source and One Topic)

The first transition for this topic is from state $src = 1, dest = 13, loc = 1$, and results in the next states $src = 1, dest = 13, loc = 16$, and so on.

Loop Detection

Our implemented NuSMV model checker also provides a way of detecting the existence of loops in the routing table. The aim of loop detection is to find out whether a message goes back to a broker that it has already passed through. Equation 5-12 shows an example rule for detecting a loop path at Broker 1:

$$src = 1 \land EX(AF(src != 1)) \qquad \text{(5-12)}$$

Formula 5-12 will be true if a message does not pass through Broker 1 again after passing through it once. However, it is possible that a broker is the source broker as well as the destination broker at the same time (it has a subscriber of a specific topic, but at the same time, this broker also has a publisher to the same topic). Equation 5-12 will be evaluated to be false, as the source and destination broker are the same for that topic. There are two ways to overcome this problem. The first is to extract all the topics whose sources and destinations are in the same broker. The second way would be to use Timed CTL to impose a restriction in Formula 5-12 where it would only evaluate to false if and only if the message again passes through the same broker after it has traversed one or more hops.

Table 5-5 displays a performance comparison between using and not using the sub-path detection algorithm for the fifty-broker model.

Number of topics	Average number of paths to be tested		Average time to generate NuSMV code (ms) over 50 experiments on each different number of topics	
	Sub-path detection	No sub-path detection	Sub-path detection	No sub-path detection
10	30	50	2	7

50	120	250	3	27
100	200	500	7	67
500	468	2500	15	183
1000	550	5000	18	318
5000	570	25000	21	7942
10000	588	50000	37	58627

Table 5-5 Sub-Path Detection Performance Comparison

In order to assess the performance of our sub-path detection algorithm, we automatically generate different numbers of topics and assign each topic to 5 subscribers. From Table 5-8, we can see that by using the sub-path detection the number of paths to be verified by NuSMV and the total time of generation are significantly reduced.

5.2.7   Verifying Failures in a Large Scale MOM System

The two main types of failure within a MOM system are path failure or degradation beyond acceptable limits and broker failure. There are many reasons (e.g. path or buffer overload, power outages) that can lead to a failure. Quite a few researchers have devoted time to load balancing [63, 64, 65, 66, 67] and provide measures to prevent failures. However, failures may still happen and a robust system needs to respond quickly to such incidents. In this research, since the user needs to manually input brokers' information, incorrect configuration may happen. It is possible that the user may forget a link or accidentally isolate a broker. This thesis proposes a way to verify failures on paths and on brokers.
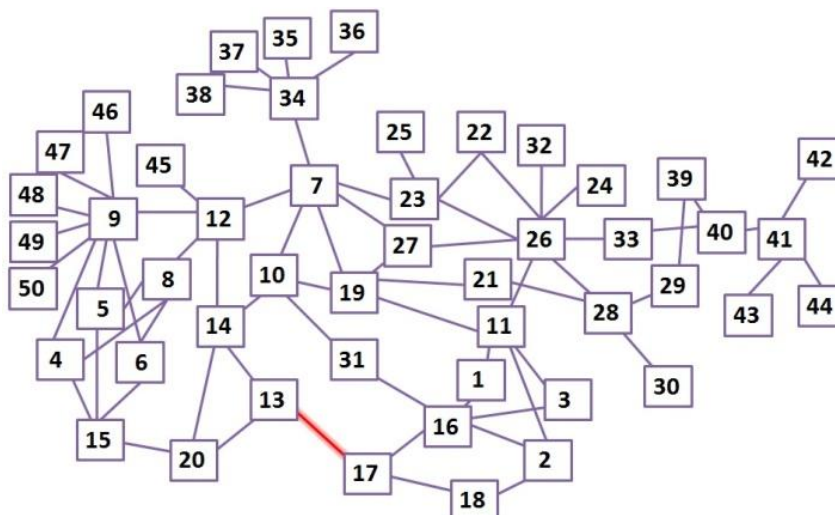
Figure 5-9 Failure of Direct Path between Broker 13 and 17

In previous sections, an integrated model for verifying the reachability for all topics has been presented. The model checker will terminate with either 'true' or 'false' to show the availability of a path or otherwise. All unavailable paths will be noted down and then their one-hop sub-paths will be tested again to locate the failed link(s). In Figure 5-9, let's assume that the direct path from Broker 13 to 17 has failed (we name this failed path as P1). Since P1 failed, all paths that involve P1 will terminate with 'false'. For example, the path from Broker 12 to 18, which including P1, will terminate with 'false'. Then, its one-hop sub-paths which are the direct path from Broker 12 to 14, the direct path from Broker 14 to 13, the direct path from Broker 13 to Broker 17 and the direct path from Broker 17 to 18 will be retested. In this scenario, only the direct path from Broker 13 to 17 will terminate with 'false' and then we can locate the failure path.
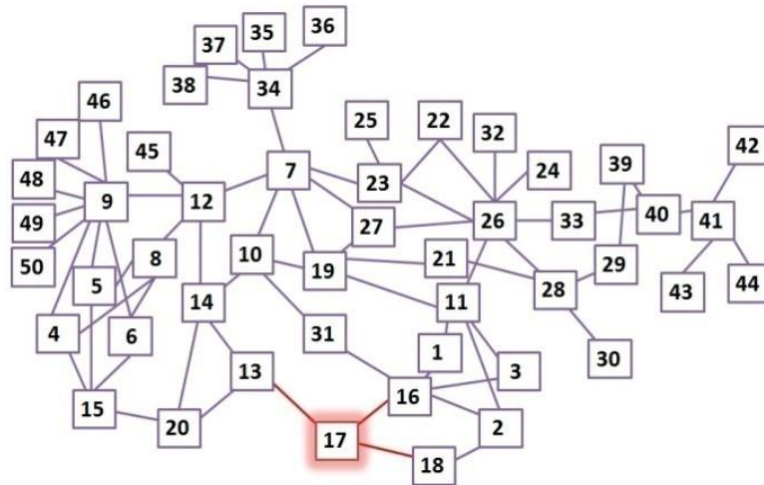
Figure 5-10 Failure of Broker 17

If a broker fails, all the direct paths traversing the failed broker will be unreachable. By the same measure, as shown in Figure 5-10, the direct path from Broker 13 to 17, the direct path from Broker 16 to 17 and the direct path from Broker 18 to 17 will test unreachable. There is then a high probability that Broker 17 has failed.

After locating the failure paths and brokers, the user can revise the topology and compensate for the failures.

5.3 **Automatic Code Generator**

As a publish/subscribe system grows in scale, there will be more brokers, publishers and subscribers. Every time one publish/subscribe system adds a new device, the reconfiguration of this system should be verified. Therefore, the model will change. It is time consuming to rebuild the model and write a new file to verify its reconfiguration. It is more efficient to have a code generator where users just need to provide the rules and a routing table.

This research develops a code generator that automatically builds the (Publisher/Subscriber) tester. A GUI first collects overlay network configuration information and stores it in a hash map. The hash map is then used directly as information to be read from or written to by the Java-based Automatic Code Generator program. In this way, the SPF routing table will be generated and a Java Parser reads this routing information and builds a new model in NuSMV along with the appropriate NuSMV rules. A flowchart of the Code Generator is shown in Figure 5-11.

Start

User input
brokers' connection
and topics through
GUI

A Hash Map generated
by GUI class ans
passed to code
generator class

While (hash map has next
record)

Yes

No

Add correspounded
information to broker
link information array

Calling SPF algorithm
method to generate
routing table and stored
as data structure *P*

Based on the two diamtional
array *P* and the rules of
NuSMV language, Code
Generator creates the model of
this overlay network in
NuSMV language.

Searching the Hash Map again,
following the rules of sub-path
detection and temporal logic,
Code Generator creates the last
part ot the NuSMV file--
verification specifications

Running NuSMV file under
the settled envirement and
checking the truth of each
verification specifications

end

Figure 5-11 Code Generator Flowchart

5.3.1 Broker Information Collection

The first step of building this verification system is to collect the broker information and then setup the overlay network in NuSMV model checker. Thus the user needs to provide the total number of brokers in the overlay network (this will only need to be input once at the beginning) and the information for each broker. The information for each broker includes the identification (ID) of the broker that is used to identify the broker in the overlay network; the IDs of the neighbouring brokers that have direct connections to this broker; and the published and subscribed topics that this broker dispatches.

A GUI is used to collect broker link information and the topic table. Figure 5-12 is a screen shot for the GUI.

Figure 5-12 A Screen Shot for the GUI Interface Implemented in This Research

This GUI interface facilities the collection of both the link table for brokers and the topic table in the following sequence:

Step 1:   The user indicates how many brokers there are in the system that need to be verified (this number is named 'TOTAL' in the remaining steps). Then press 'Create'.

Step 2:   After pressing 'Create', for each broker, the user inputs the broker ID, total number of neighbouring brokers, link information and also the topics this broker is publishing and subscribing.

Step 3:   After the user has finished inputting a broker's information, they should press the 'update' button to write the information into a hash map.

Step 4:   If the user wants to rewrite the information of a specific broker, he/she can input the broker's ID number at the space for broker ID and then press the 'clear' button. All the stored information for that broker is deleted and the user can rewrite the information.

Step 5:   After the information of all brokers has been stored in the hash map, the button named 'finish' at the bottom of the window will be active to be pressed.

Step 6:   When the user presses the 'finish' button, all the link information for brokers and topic tables stored in the hash map is sent to a class named 'RoutingTableGenerate.java' to generate the routing table, the topic table, the NuSMV model and verification specifications. For now, the routing table is based on the Shortest Path Algorithm.

5.3.2   Hash Map Generation

The first step of building this verification system is to collect the broker information

and then setup the overlay network in NuSMV model checker. Thus the user needs to provide the total number of brokers in the overlay network (this will only need to be input once at the beginning) and the information for each broker. The information for each broker includes the identification (ID) of the broker that is used to identify the broker in the overlay network; the IDs of the neighbouring brokers that have direct connections to this broker; and the published and subscribed topics that this broker dispatches. This information is used to further automatically generate a routing table that contains the shortest path between any two brokers in the network, and the subscribed and published topic distribution information. This information is stored into the hash map. The structure of the hash map is HashMap<brokerID,HashMap<<neighboringBrokers,value><publishTopics,value><subscribeTopics,value>>>

The key to the outer hash map is the ID of a broker. The inner hash map has three keys the 'neighbouringBrokers', 'publishTopics' and 'subscribeTopics' of that broker. Table 5-6 illustrates the information that is stored in the hash map structure for the first 3 brokers of Figure 5-10:

brokerID	Neighbouring Brokers	publishedTopics	subscribedTopics
1	11,16	weather, films	music
2	11,18	sports	weather, stock
3	11,16	music	sports

Table 5-6 Structure of the Stored MOM Configuration

The part in the hash map storing information for broker 1 will be:

HashMap<1,HashMap<<neighboringBrokers,'
11,16'><publishTopics,'weather,films'><subscribeTopics,'music'>>>

5.3.3   Routing Table Generation

Here we define a path as a set containing all brokers that a message passes through from its source to destination during delivery. All possible paths for the overlay network are generated by using Djikstra's shortest path algorithm [26] based upon the link information, which refers to the broker and its direct connected brokers (neighbouring brokers). The shortest paths for each source and destination are stored in a two dimensional array ($n \times n$), where n is the number of brokers in the network. This information can be represented as a matrix P where the rows correspond to the source broker IDs, the columns correspond to the destination broker IDs and the values are the corresponding sets containing the correspondent shortest paths:

$$P = \begin{bmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,n} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,n} \\ \cdots & \cdots & \cdots & \cdots \\ p_{n,1} & p_{n,2} & \cdots & p_{n,n} \end{bmatrix}$$

(5-8)

With the help of the GUI interface, link information of the brokers and the topic table are collected and stored in a hash map. This hash map is passed to the 'RoutingTableGenerate.java' class and used to generate the routing table as well as the verification specifications of the reachability of topics. The 'RoutingTableGenerate.java' class automatically performs the task of generating an abstract model for verifying MOM systems.

Firstly, this class splits link information from the hash map into a two dimension (50×50) integer array named 'broker'. The first dimension of index of 'broker' indicates the ID for a broker, and the second dimension of index displays the number of the broker's neighbouring brokers. For example, in this fifty broker model, broker 1 has two neighbour brokers. One is broker 11 and the other is broker 16. The link

information of broker 1 will be stored as: 'broker[1][0]=11', 'broker[1][1]=16'.

After translating the entire link information into 'broker[][]', this class constructs a routing table using a two dimension (50×50) String array named 'path' in which the shortest paths for any two brokers are stored. In this array, the indexes indicate the ID of the source (first index) broker and the ID of the destination broker (second index); and the value records a list of broker IDs (destination broker ID included) that are the nodes of the shorted path from the source broker to the destination broker, e.g., path[A][B] = 'C,D,B' means that the shorted path from broker A to broker B is A – C – D – B. The following steps give a brief summary of the routing table generation process:

- First, the value for 'path[][]' is initiated. In this array, the values for 'path[m][n]'($1 \leq m \leq 50$, $1 \leq n \leq 50$, $m \neq n$) are set to 'null', the values for 'path[m][m]' ($1 \leq m \leq 50$) is set to 'm', and the values for 'path[0][n]' ($1 \leq n \leq 50$) and 'path[m][0]' ($1 \leq m \leq 50$) are set to 0.

- Second, one hop paths (a broker and its neighbours) are retrieved and recorded in path[][], i.e., for each broker i, the path information for its neighbouring broker j (retrieved from broker[][]) is updated to path[i][j] = 'j'.

- Third, every broker starts to learn from its neighbouring brokers. For example, broker 1 has broker 11 and broker 16 as its neighbouring brokers. Broker 11 has broker 19, broker 26, broker 1, broker 2 and broker 3 as its neighbouring brokers. Broker 16 has broker 31, broker 17, broker 1, broker 2 and broker 3 as its neighbouring brokers. So, broker 1 will learn that path[1][19]='11,19', path[1][26]='11,26', path[1][2]='11,2', path[1][3]='11,3', path[1][31]='16,31', path[1][17]='16,17'. As long as the path[i][j] ($i, j \in \{0,1,2,...50\}$) has values, a new path from broker i to broker j will not be updated. By this, this program can

make sure every path is the shortest path from broker i to broker j. This kind of learning process continues until there is no 'null' value in 'path[][]'.

After generating the matrix *P* (showing in Formula 5-8), the routing table for the overlay network is generated. The routing table can be represented as a matrix R where each row corresponds to a broker (Broker ID) of a topic and each column represents a destination broker (Broker ID). The value stored in the matrix shows the next hop according to its current location and destination.

$$
R = \begin{bmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,n} \\ r_{2,1} & r_{2,2} & \cdots & r_{2,n} \\ \cdots & \cdots & \cdots & \cdots \\ r_{n,1} & r_{n,2} & \cdots & r_{n,n} \end{bmatrix}
$$

(5-9)

For example, for a particular topic, the first 3 brokers of Figure 5-10, we have the routing information shown in 错误！未找到引用源。:

Loc.	Dest.	Loc.
1	3,7,11,19,20,21,22,23,24,25,26,2728,29,30,31,32,33,34,35, 36,37,38,39,40,41,42,42,44	11
1	2,4,5,6,8,9,10,12,13,14,15,16,17,18,20,31,45,46,47,48,49,50	16
2	7,8,9,11,12,19,21,22,23,24,25,26,27,28,29,30,32,33,34,35,36,37,38,39, 40,41,42,43,44,45,46,47,48,49,50	11
2	1,3,10,14,16,31,	16
2	4,5,6,13,15,17,18,20,	18
3	3,7,11,19,20,21,22,23,24,25,26,2728,29,30,31,32,33,34,35, 36,37,38,39,40,41,42,42,44	11
3	2,4,5,6,8,9,10,12,13,14,15,16, 17,18,20,31,45,46,47,48,49,50	16

Table 5-7 of Routing Table Example

In this table, the first column shows the broker ID. The second column shows the

identification of the destination broker. The last column provides the next hop of that message.

5.4 Sub Path Detection for Reduced Verification Time

The examples explored in Sections 5.1 and 5.2 could be extended to a much larger system (many brokers, publishers, subscribers and topics). By using the code generator to build the NUSMV Publisher/Subscriber tester, the user could input the corresponding routing table and rules for the larger system. Those routing tables and rules could be verified through CTL and LTL formulas.

However the larger the system, the more complex the rules and logic become. If there are too many state variables, the model can suffer from a state explosion [7]. This research performed experimentation to compare the time the NuSMV takes to generate systems of different sizes. The systems are modelled using simple rules as the one used to the six-broker example. As can be seen in Table 5-10 错误！未找到引用源。, a system with 384 brokers would take 2591069ms to be generated in NuSMV. If the system, with 384 brokers, fails, the whole model should be built again to verify the reconfiguration and this would take nearly one hour. This delay would be prohibitive for many real working systems as they need a quick response for the new system reconfiguration to be correct. Fortunately, according to the authors of [14], brokers can be high capacity, which is to say even large systems typically comprise less than 50 brokers. One of the solutions to avoid the state explosion problem is proposed in [11]. The authors propose to break the system into smaller sub-systems and verify each sub-system separately.

Number of brokers	XML code lines	NuSMV code lines	Average time to generate NuSMV code (ms) (tested over 10000 experiments)
6	138	188	85
12	558	320	379
24	2262	800	729
48	9126	2624	1069
96	36678	9728	9336
192	147078	37760	144605
384	589062	149120	2591069

Table 5-8 Comparison of Code Generation for Different Sized Systems

In order to decrease the time for the model generation, this research has implemented an algorithm that extracts all the routing sub-paths contained in the topic table. In this way only the super paths need to be verified. For example, in Table 5-2, topic A will be sent from Broker 1 to Broker 5. If the path from Broker 1 to Broker 5 is proved to be reachable, then all its sub-paths (Broker 1 to Broker 2, Broker 1 to Broker 4, Broker 2 to Broker 4, Broker 2 to Broker 5 and Broker 4 to Broker 5) will be reachable too. Consequently, although a MOM system can have a very a large number of topics, the checker must only verify the reachability of the super paths of the MOM.

In order to detect all the sub-paths in this six-broker model, this research builds a topicTable.txt file to store the topic table information. Additionally, the topic table information will be stored in a simpler form (just the publisher broker's id and the subscriber's broker id per line) in the topicTable.txt file. For example, table 5-2 will be stored as:

1	2
1	5
3	4
5	1
5	6
6	3

The first column stores the publisher brokers' IDs and the second column stores the subscriber brokers' IDs. Then, this research builds a Java class called SubPaths.java. In this Java file, all the possible sub paths are stored. Setting Broker 1 as an example:

```
switch (pubBrokerID){

        case 0 : //the publisher broker is broker 1
            switch (subBrokerID){
                case 0: //the subscriber broker is broker 1 and the super path is from
                broker 1 to broker 1
                    paths[0].pathTester[0]=1; //the sub path is from broker 1 to broker
                    1
                break;
                case 1: //the subscriber broker is broker 2 and the super path is from
                broker 1 to broker 2
                    paths[0].pathTester[0]=1;
                    paths[1].pathTester[1]=1;
                    paths[0].pathTester[1]=1;// the sub path is from broker 1 to broker
                    2
                    paths[1].pathTester[0]=1; // the sub path is from broker 2 to broker
                    1
                break;
                case 2://the subscriber broker is broker 3 and the super path is from
                broker 1 to broker 3
                    paths[0].pathTester[0]=1;
                    paths[1].pathTester[1]=1;
                    paths[2].pathTester[2]=1;
                    paths[0].pathTester[1]=1;
                    paths[1].pathTester[0]=1; paths[0].pathTester[2]=1; // the sub path
                    is from broker 1 to broker 3
                    paths[2].pathTester[0]=1;
                    paths[1].pathTester[2]=1;
                    paths[2].pathTester[1]=1;
```

97

```
                        break;
                        case 3: //the subscriber broker is broker 4 and the super path is from
                        broker 1 to broker 4
                             paths[0].pathTester[0]=1;
                             paths[1].pathTester[1]=1;
                             paths[3].pathTester[3]=1;
                             paths[0].pathTester[1]=1;
                             paths[1].pathTester[0]=1;
                             paths[0].pathTester[3]=1;
                             paths[3].pathTester[0]=1;
                             paths[1].pathTester[3]=1;
                             paths[3].pathTester[1]=1;
                        break;
                                        … …
                 break;
```

The indexes of the array paths [ ] show the publisher brokers' IDs and the indexes of pathTester [ ] show the subscriber brokers' IDs. The initial values stored in a two dimensions array called paths [ ].pathTester [ ] are all ZERO. For example, a path from Broker 1 to Broker 5 is corresponding to paths [0].pathTester [4]. As long as the super path has been written into topicFormula.txt and waits for verifying, all sub paths belong to this super path will be regarded as verified as well. For example, if the super path from Broker 1 to Broker 3 is verified, all the values stored in paths [ ].pathTester [ ] which represent respective sub paths (from Broker 1 to Broker 1, from Broker 1 to Broker 2, from Broker 1 to Broker 3, from Broker 2 to Broker 3 and all the reversed paths) will be assigned to ONE. A Java class' file reader goes through the topicTable.txt line by line. Firstly, it checks the value stored in respective array (paths [ ].pathTester [ ]), based on the indexes numbers, and if the value is ZERO this Java class will write this path into a topicFormula.txt file to be verified later, then assign the values of sub paths which belong to this path to ONE. This Java class' file reader moves to the next line in topicTable.txt. If the value stored in respective array (paths [ ].pathTester [ ]), based on the indexes numbers, is ONE, the Java class' file reader moves to the next line in topicTable.txt directly.

Table 5-11错误！未找到引用源。 provides a comparison between using and not using the sub-path detection algorithm for the fifty-broker model, illustrating the benefit of sub path detection.

Topic numbers	Average number of paths to be tested		Average time of generating NuSMV code (ms) (tested over 10000 experiments)	
	Sub-path detection	No sub-path detection	Sub-path detection	No sub-path detection
10	8	50	5	7.5
50	9	250	6	27
100	9	500	5	71
200	9	1000	5	265
300	9	1500	6	601
500	9	2500	6	1730
1000	9	5000	7	9230

Table 5-9Comparing Using and Without Using Sub-path Detection

In order to validate our sub-path detection mechanism, this research automatically generates different numbers of topics and assigns each topic 5 subscribers. From Table 5-6, it can be seen that by using our sub-path detection the number of paths to be verified and the total time of generation are significantly reduced.

However, in this thesis the fact that publishers and subscribers can dynamically join and leave the MOM system still needs to be considered. Using only CTL and NuSMV, real-time systems cannot be verified.

5.4.1 Generating CTL Specifications in NuSMV for Every Stored Topic

We build a NuSMV model of the delivery process for each topic. After having collected the necessary information for NuSMV to create the finite state machine for

the MOM overlay network, the CTL specifications for checking the reachability of each topic needs to be generated for the input in the NuSMV model. What is verified here through CTL specifications is that each message from a certain publisher will reach all the required subscribers. The necessary information for building the model for each topic is the source broker ID, the destination broker ID and the broker ID of the current location. The NuSMV requires the following three state variables:

src is the source broker ID;

$dest$ is the destination broker ID;

loc is the current location broker ID.

For example, if Broker 1 is the source of a topic and one of this topic's destination broker is Broker 13 The reachability verification can be represented in a CTL specification as:

$$(src = 1 \wedge dest = 13 \wedge loc = 1) \rightarrow AF(src = 1 \wedge dest = 13 \wedge loc = 13))$$

(5-10)

The CTL specification (5-10) indicates that finally there will be a state where the current location of a topic coincides with its destination identification. This means that eventually the message will arrive at its destination.

The following pseudo-code describes the algorithm used for generating the NuSMV

CTL specifications for every topic managed by the MOM:

1. Initialize matrix F all to **FALSE**
2. N:=total number of brokers
// For each broker, extract each subscribed topic
3. **For** ID ← 1 **to** N
// Extract the subscribed topics of broker ID
4.   **For** t ← 1**to** Length Of (ID.subscribedTopics)
5.   TOPIC ← ID.subscribedTopic(t)
6.   IDdest ← ID
// Going through all the brokers to extract the publishers of TOPIC
7.   **For** IDSourcebroker ← 1 **to** N
// Search and extract the publishers of TOPIC
8.   **For** n ← 1 **to** LengthOf (IDbroker.publishedTopics)
9.   **if** TOPIC=IDbroker.publishedTopics(n)
10.   **then** IDsource ← IDbroker
// Check matrix F, will be shown in Formula 5-11, to see if CTL formulas have already been written for this path and its sub-paths
11.   **if** f*IDsource, IDdest* *!=* '**true**'
12.   **then**
13.   Write the corresponding CTL formula
// Search matrix P (formula 5-8) and extract the routing path of $p_{IDsource,IDdest}$
14.   **For** i ← 1 **to** l_p ← LengthOf ($p_{IDsource,IDdest}$)
15.   **For** j ← i+1 **to** l_p
// Go through all the possible hops between IDsource and IDdest
16.   Bsource ← $p_{IDsource,IDdest,i}$
17.   Bdest ← $p_{IDsource,IDdest,j}$
18. f*Bsource, Bdest* ← 'true'
19. f*Bdest, Bsource* ← 'true'
20.   **End** for j
21.   **End** for i
22.   **else if** goes to step 7
// The path and all sub-paths between IDsource and IDdest for TOPIC have been flagged as 'true'
23.   **End** for n
24.   **End** for IDSourcebroker
// All publishers of TOPIC have been searched and written into CTL formulas
25.   **End** for t
// All the subscribed topics of Broker ID were processed
26.   **End** for ID
// All the subscribed topics of all brokers were processed
// consequently, all CTL formulas for all sources and destinations of all topics have been processed

Our implemented NuSMV model checker can test all the sources and destinations of a topic. If there are a large number of topics, the checker will take a long time to test all the source and destination entries. We require that for a pair of source-destination brokers, there is a path connecting them based on the routing protocol. However, this path could be part of another path or include other paths of other topics. We call a path of a topic that completely contains the paths of other topics as a 'super-path' and the contained paths are 'sub-paths'. For example, in Figure 5-10, a message that must go through a path from Broker 1 to Broker 13, it must go through Broker 16 and Broker 17. This path alone has 12 different sub-paths including itself (e.g. the number of possible combinations of all possible source and destination pairs). The path of a topic could be a sub-path of another already successfully checked path, consequently this sub-path need not be tested again.

With the increasing number of brokers, manually listing all possible paths (as presented in [22] for the six-broker model) is no longer feasible. An algorithm that can automatically generate all possible paths and find all sub-paths for a tested path is required for larger MOM systems. We integrate the detection of sub-paths and already processed paths in an algorithm that writes the CTL specification for each topic, significantly decreasing the overall testing time. In the algorithm, a matrix F is created to store the states indicating if a CTL specification has been already written for a specific path (source-destination pair) or not. In this matrix, if for a specific source–destination path a CTL specification has been written then the corresponding value is set to 'true', otherwise the value is 'false'. Rows in matrix F indicate the source of a path, columns indicates the destinations of the path, and the corresponding Boolean value indicates whether a verification check for that path has been conducted or not.

$$F = \begin{pmatrix} f_{1,1} & f_{1,2} & \cdots & f_{1,n} \\ f_{2,1} & f_{2,2} & \cdots & f_{2,n} \\ \cdots & \cdots & \cdots & \cdots \\ f_{n,1} & f_{n,2} & \cdots & f_{n,n} \end{pmatrix}$$

(5-11)

5.5 Concluding Remarks

The primary disadvantage of many MOM systems is that they require an extra component in the architecture, the message transfer agent (i.e. the message broker). As with any system, adding another component can lead to a reduction in performance and reliability, and can also make the system as a whole more difficult and expensive to maintain. The goal to this research is to find a suitable means to retain the performance and reliability for a MOM system after a failure.

This work provides a code generator to automatically generate a MOM overlay checker when the links and topics of each broker are provided. Then the reachability of all topics is tested and loops within the topology are detected. The configuration could be manual. Normally this would be automatic, but manual override by the user is permitted. As such, our work is essential for detecting 'human errors'. Moreover, if some paths or brokers fail (or as a result of a misconfiguration), our checker quickly locates the failure and informs the user.

In this research, the routing table is generated based on the shortest path algorithm although it is possible to verify the contents of the table no matter how it is created (i.e. including static entries). In future work, we will provide more requisites for verifying a realistic MOM system. One of them is the ability to verify the reachability of topics when no predefined routing table is available. Moreover, since many MOM systems are expected to provide an efficient and high quality messaging service, another important requisite is to check if the rules for guaranteeing that end-to-end latency constraints are met by every broker. However, these new features are for

future work.

This chapter has described a methodology for implementing a means of model-checking MOM systems models. Three MOM systems models (three-broker model, six-broker model and 50-broker model) have been presented. The first two models focus on different aspects of configuration and reconfiguration of a MOM overlay network. The three-broker model is used to implement verifying following aspects:

- Each publisher should have at least one subscriber who subscribes the topic this publisher published.

- Each subscriber should have at least one publisher who publishes the topic this subscriber interested in.

- After a broker failed, all the end nodes should be translated to the neighbouring brokers of the failure broker until this failure broker performs normally.

The six-broker model is used to implement verifying the reachability of each topic and detecting the loops within a MOM overlay network based on the routing table and topic table of this MOM system.

In order to provide a scalable verification mechanism that can cope with requirements of larger commercial MOM systems, the third model is introduced, comprising fifty brokers. This work develops then a code generator to automatically generate a MOM overlay checker model whereby the link table of the brokers and the topic table for each broker are collected by a user-friendly GUI interface. Then the reachability of all topics is tested and any loops within the topology are detected.

Although configurations could be automatically generated, manual intervention by administrators are not untypical. Therefore a means of verifying the configuration data is essential. A new configuration can be prepared by the administrator based on the feedback provided by the model checker. The code generator that this research has developed allows any user to build a model checker for a MOM based publish/subscribe system, even if the user has little knowledge of programming languages or of NuSMV.

As the number of brokers increases, manually generating routing tables for a large-scale MOM system becomes extremely complex. As mentioned before, Code Generator, which is proposed by this research, can assist administrators to build a model. By using the tool, users can simply input link information of the brokers, which is much easier to provide than a complete routing table, and the tool automatically generates a model from this link information. In this research, the routing table is generated based on the shortest path algorithm although it is possible to verify the contents of the table no matter how it is created (i.e. including static entries).

6 Fast Recovery from Overlay Network Failures

In previous chapters, this thesis has presented methods for formally verifying topic reachability and loop free detection. These methods focus on guarantying correct configuration before launching the whole system. However, during the communication phase, brokers located at different geographical areas may suffer from unexpected failure of a broker or overlay link. Although, normally the possibility of a link failed is low and simultaneous links/nodes failure are much rarer [3, 4, 20, 21], a resilient system should provide measures to ameliorate failure scenarios. A mission critical message oriented middleware application, which this research based on, requires a reliable and efficient messages delivery mechanism [103, 104]. Messages are time sensitive so it is necessary to employ an overlay network fast recovery scheme. Given the growing size and complexity of mission critical MOM overlay networks, the presence of component failures can be an everyday occurrence [124]. Hence, considerable attention has been paid to the problem of fast recovery from link failures.

In order to recover quickly from a network failure, some researchers have proposed reducing the failure detection time whilst others have focused on saving time associated with processing state update packets, calculating new routing tables and forwarding.

For one of the most wildly used network routing protocols Open Shorted Path First, if a router does not receive a Hello message from its neighbour within a RouterDeadInterval (typically 40 seconds or 4 HelloIntervals), it assumes the link between itself and the neighbouring node to be down. Therefore, Goyal et al. [105] proposed optimizing the HelloInterval such that fast failure detection in the network is

possible whilst keeping the false alarm occurrences within acceptable limits. However, the HelloInterval is expressed in 'seconds', so the minimum detection time will be 4 seconds. For a mission critical MOM application, a delay of 4 seconds will still be intolerable.

Quite a few proposals attempt to handle link failures as locally as possible, by only undertaking routing updates in a limited number of routers near the point of failure, such as [106]. This proposal announces that if two equal cost paths exist towards a destination, traffic can safely be switched from one to the other in case of a failure. Similarly, Iselt et al. [107] propose using Multiprotocol Label Switching (MPLS) tunnels to make sure that there always are two equal cost paths towards a destination at every node. Other proposals consider the situation from a global perspective, and try to guarantee that there always is a valid routing entry for a given destination even after a failure. With O2-routing [108], the routing tables are set up so that there are always two valid next hops toward a destination. However, the proposal does not guarantee the shortest path. Hansen et al. [109] propose that for a given network topology, there could be several layers with different safe nodes and safe links. Most nodes and links will only appear in one layer. If a failure arises in a particular layer, the cost of all the links within this layer is set to infinity. Therefore, each node just needs to store a routing table for normal operations and for each layer failure situation.

In order to recover rapidly from failure, this thesis proposed a Pre-Calculated Routing Tables (PCRT) algorithm. All the routing tables for normal and different failure scenarios are calculated in advance to save the time by avoiding the need to update a Link State Database (LSD) in response to topology changes triggering reactive flooding of Link State Advertisements (LSAs), and the subsequent calculation of new routing tables. Other than sending LSD flooding messages and exchanging Hello messages, another advantage for this PCRT algorithm is employing the exchange of

regular Heartbeat messages with a 'colour' flag, the colour indicating the state of the network (i.e. in terms of which links are operational) as perceived by the sender. In our scheme, all the pre-calculated routing tables have a one-to-one mapping onto a colour with global significance; that is, all brokers know that a particular colour is uniquely associated with a particular combination of operational and failed links for the MOM topology. If a broker receives a Heartbeat message with different colour and larger sequence number from its own, or if a local failure is detected, an update mechanism is triggered and updated Heartbeat messages will be generated.

6.1 Pre-Calculated Routing Tables Scheme

In a typical OSPF network, with a HelloInterval value of 10 seconds and RouterDeadInterval of 40 seconds, the failure detection can take anywhere between 30 to 40 seconds. The LSA flooding times consist of transmission and propagation delays and any delay resulting from the rate-limiting of LSA Update packets sent down an interface. Once a router receives a new LSA, it schedules an SPF calculation. Since SPF calculations using Dijkstra's algorithm [115] constitute a significant processing load, the router waits for some time (spfDelay - typically 5 seconds) for other arriving LSAs to trigger an SPF calculation. Moreover, routers place a limit on the frequency of SPF calculations (governed by spfHoldTime, typically 10 seconds between successive SPF calculations), which can introduce further delays.

The rationale behind the PCRT scheme is to perform all the time consuming calculations before placing the MOM network enters its operational state. There is no Hello message or LSA flooding in the PCRT scheme. Instead a new type of message, called a Heartbeat is introduced. Since the topology of a MOM network does not change frequently and the total number of brokers is normally less than fifty [104], this research does not employ Hello messages to build adjacencies between neighbouring brokers. With PCRT, besides the forwarding of data messages, Heartbeat messages are periodically sent between adjacent brokers to detect link

failure(s) and keep network state information synchronized.

At time zero, all the routing tables for each broker for all the possible / supported failure scenarios and the default normal conditions are automatically generated and then condensed and distributed to the brokers. In addition, colours and corresponding failures are associated by means of a shared list. Thus each possible entry in the condensed super routing table at a broker is located with appropriate colour and utility destination, corresponding to a given overlay topological state.

6.1.1   Super Broker

PCRT is a pre-calculated algorithm. One of the brokers, referred to as the 'super broker', is used during the configuration phase to pre-calculate the routing tables for both normal and failures situations for each broker, which are then condensed and sent to that appropriate broker. The super broker knows the complete topology of the network. An example topology is illustrated in Figure 6-1.
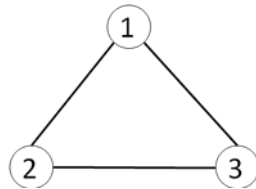
Figure 6-1 Example Three-Broker Topology

In this thesis, our research considers single link failures to explain our algorithm and validation was performed using a bespoke Pascal-based simulation tool. The structure of Super Link State Database (SLSD) is shown in Figure 6-2.

For each topology, the PCRT algorithm generates an SLSD listing of all the brokers, shown on the left. On the right-hand side, the directly connected neighbouring brokers to those brokers are shown in Figure 6-2.

Figure 6-2Super Link State Database

6.1.2　Routing Table Generation under Normal and Failure Conditions

Current Sequence Number List

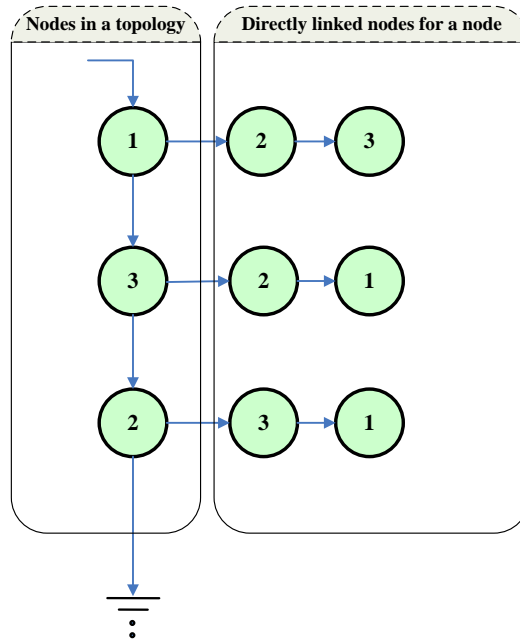A broker current sequence-number list exists in every broker. It is a list of all the nodes/brokers in a network and most recently observed sequence-number corresponding to each node/broker. Initially, each broker's sequence-number is set to '1'. When a change (i.e. a failure or recovery) happens, nodes/brokers adjacent to the affected links will increase their sequence number by one and immediately send flood notification messages with the new colour (based on the revised perceived topology state) and sequence-number to their neighbouring nodes/brokers. When a sequence number reaches its largest permitted value, after next change happens, it is reset to 1.

Current Colour

Each broker stores the current colour of the network, indicating the network state, as it perceives it. If a broker receives a flood notification message (this is one of the

heartbeat messages and will be introduced in later Section 6.1.3) with a different colour from its own current colour, it will compare the sequence number listed in this flood notification message with the one in its current sequence number list. If the sequence number stored in the flood notification message is greater/newer than the number stored in the sequence number list, this broker will update the sequence number list with this new number and search the colour lookup table to determine the failure/failures which relate to this new colour. Together with its own knowledge of failure/failures, this broker will take note of all the failure/failures existing in this network and then lookup the corresponded colour to become the new current colour (by checking the colour lookup table). This new colour may be different from the received one as multiple failures could arise concurrently, unbeknown to the broker sending a particular message. Then the receiving broker forwards flood notification messages to its neighbouring brokers except the one where this flood notification message came from but a flood ACK message is sent in this case. Otherwise, if the sequence stored in this flood notification message is equal to or less/earlier than the sequence number listed in the broker's sequence number list, this broker will discard this flood notification message and send a flood ACK message to the sender of the discarded flood notification message's source.

Conversely, if a broker receives a flood notification message with the same colour as its own current colour, it compares the sequence number listed in this Notification message with its own record of the sender's sequence number. If the sequence number stored in this Notification message is greater/newer than the number stored in the sequence numbers list, this node/broker will update the sequence-number list with this new number and forward this flood notification message along egress interfaces except the one on which this Notification message arrived. Otherwise, this node/broker discards the flood notification message and sends back a flood ACK message (introduced in Section 6.1.3).

Generating a Super Routing Table

After the Super Broker generates all the routing tables, a routing table set will be generated for each broker where each routing table uniquely corresponds to one colour (associated with a particular link state configuration). These routing table sets are only a temporary data-structure held at the Super Broker; they are not distributed to the remaining brokers but are simply used as an interim step in the construction the Condensed Super Routing Table for each broker.

The following pseudo-code shows how the routing tables for each different colour are generated. The original link state database (LSD) holds the complete topology and provides a reference template. We generate clones of this LSD which we modify based on specific failure scenarios we wish to handle. These modified clones are used to make specific Shortest Path Trees and thus routing tables for the brokers for the failure condition being considered.

1. Initialize the link state database Super_LSD
2. Initialize a global integer variable Colour =1
3. //Generate routing table for normal situation
4. **while** (Super_LSD has next node)
5.   updateRT(node)
6.   Associate this routing table with Colour
7.   Move to next node
8. **end** while
9. Colour = Colour   + 1
10. //Generate routing tables for all the single link failure situations
11. **while** (Super_LSD has next node)
12.   **while** (directLinkedNode has next)
13.   **if** (directLinkedNode_ID>Node_ID)
14.   Clone_LSD ← Super_LSD
15.   Fail the links between directLinkedNode_ID and Node_ID
16.   **while** (Clone_LSD has next node)
17.   updateRT(node)
18.   Associate this routing table with Colour
19.   Associate this failure with Colour
20.   Move to the next node
21.   **end** while
22.   Dispose(Clone_LSD)
23.   **end** if
24.   Move to next direct linked node
25.   Colour = Colour + 1
26.   **end** while
27. **end** while

Table 6-1 and Table 6-2 are examples of routing tables temporarily created in sequence at the Super Broker to permit the construction of the condensed super routing table for broker 1 which are based on the topology shown in Figure 6-1.

Colour 1		Colour 2	
Ult. Destination broker	Next hop broker	Ult. Destination broker	Next hop broker
2	2	2	3
3	3	3	3

Table 6-1Routing Tables for Broker 1 (Part 1)

Colour 3		Colour 4	
Ult. Destination broker	Next hop broker	Ult. Destination broker	Next hop broker
2	2	2	2
3	2	3	3

Table 6-2 Routing Tables for Broker 1 (Part 2)

Table 6-3 shows the 'colour and failure mapping table' where a one-to-one mapping between a colour and a specific failure situation are listed. Here we confine ourselves to single link failures as an example to keep the explanation simple. A failure link between broker i and broker j, where $i, j \in \{1, 2, 3\}$, can be represented as $F(i, j)$.

Colour	Failure
1	None
2	$F(1, 2)$
3	$F(1, 3)$
4	$F(2, 3)$

Table 6-3 Colour and Failure Mapping Table for Three-Brokers Topology

This so-called 'colour and failure mapping table' is a list of mapping from colours to failures. One colour is uniquely mapped onto a particular failure or a combination of failures. The reverse mapping is also true. This will be copied and stored in all the nodes/brokers so they will have the same 'dictionary' of colours and failures. There are two parts to this table. The colours are represented by several bits, which depend on the total number of links in the network and the number of failure combinations that are to be protected against. Then the remainder of this table stores the failures and

the combination of failures corresponding to each colour.

Based on Table 6-1, 6-2 and Table 6-3, all the different combinations of destination broker and next hop broker are used to form the following super routing table. In this super routing table, shown in Table 6-4, all possible entries for message processing at broker 1 for all considered failure situations are represented.

1.   Assume there are **N** brokers
2.   Routing tables set for broker n: 'RTsSet(n)' where **0<n<=N**
3.   **n** ← 1
4.   Let SuperRT(n) be the super condensed routing table for broker **n**
5.   Boolean addEntry=**True**
6.   **while** (n<=N)
7.   **while** (RTsSet(n) has next entry)
8.   **while** (SuperRT(n) has next entry)
9.   **if** (RTsSet_E==SuperRT_E)
10.   Add the corresponded colour of RTsSet_E to SuperRT_E states   link
11.   addEntry=False;
12.   **break**;
13.   **end if**
14.   **end while**
15.   **if**(addEntry==True)
16.   Add REsSet_E to the end of SuperRT(n);
17.   **end if**
18.   **else**
19.   addEntry=True;
20.   **end else**
21.   **end while**
22. **n++**
23. **end while**

Super routing table			
Address	Ult. Destination broker	Next hop broker	Colour
1	2	2	1,3,4
2	3	3	1,2,4
3	2	3	2
4	3	2	3

Table 6-4 Super Routing Table for Broker 1

Rather than storing four routing tables in Broker 1, one condensed super routing table is constructed whereby redundant/ duplicate entries are omitted. Let N be the total number of brokers in a topology and L_i is the number of directly linked broker/brokers of broker i, while $1 \leq i \leq N$. So there are $(N-1)$ destinations for messages processing in broker i. The worst case of a super routing table contains maximum *Max* different entries:

$$Max = (N-1) \times L_i \qquad (6\text{-}1)$$

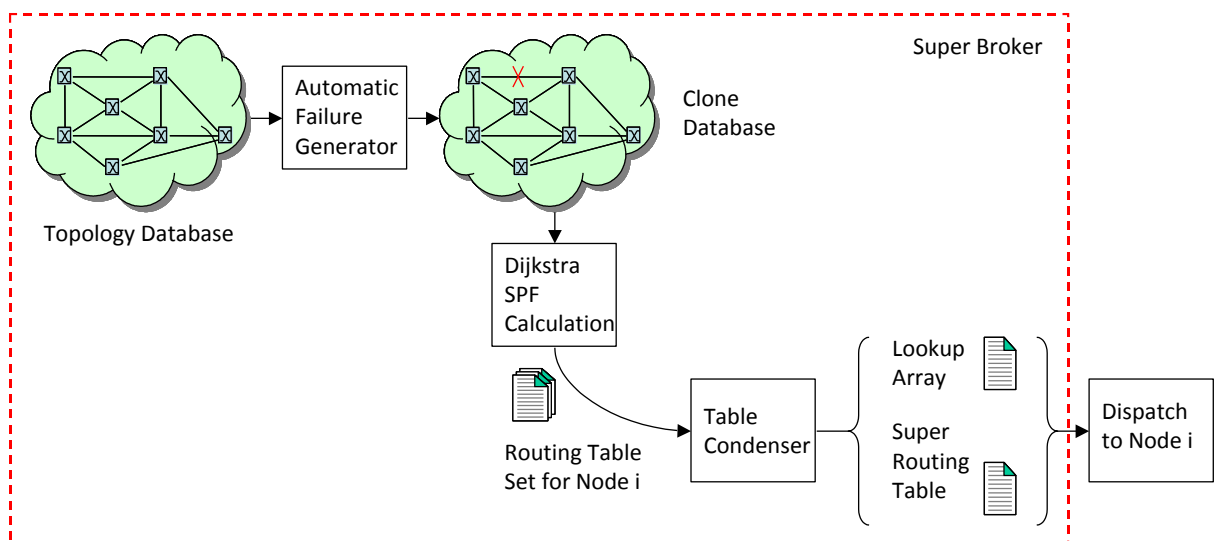The compete PCRT initialisation phase is shown in Figure 6-3.

Figure 6-3 PCRT Initialisation Phase

The super node uses the true topology database to generate a series of clone databases, each with a specific combination of failures and assigns each one a unique 'colour' identifier. Dijkstra is then performed on each clone database to generate an appropriate SPF routing table for every node in the overlay network. Once Dijkstra has been performed for all the considered clone database failures, each node has a set of routing tables, one for each colour state. The table condenser function then removes redundant (duplicate) information, resulting in a single much condense super routing table for a given node. In addition a 2-dimensional array provides a simple means of accessing the appropriate entry within the super routing table. In many instances, separate entries in the array (associated with differing failure conditions) will point to the same routing information in the super routing table.

6.1.3   Heartbeat Messages

The Heartbeat message is a new entity proposed in our PCRT algorithm, which is periodically sent to each broker's neighbouring broker/brokers to detect failures and update the latest topology structure. Figure 6-4 illustrates the structure of the Heartbeat message. There are three parts to the message. The first part is called 'type flag' containing three possible types of a Heartbeat message, namely: flood notification message, test link notification message and regular heartbeat message. The 'Acknowledgement (ACK) flag' is a Boolean variable to show if this Heartbeat message is a response to a certain type of a previous received message. The last part, 'packet information', may contain network change information when a change (path failure or path recovery) happens or maybe null if it is just a regular Heartbeat message.

Figure 6-4 Heartbeat Packet Structure

Regular Heartbeat Message

Heartbeat messages are periodically exchanged between adjacent brokers. These normal Heartbeat messages have 'type' showing regular Heartbeat and 'ACK flag' set to FALSE and the 'packet information' part null, are used to detect change of link status. Regular Heartbeat messages are sent periodically and if a broker has not received any Heartbeat message for a specified dead-interval, the link will be assumed to be 'dead' and the state of that link interface will be set to 'failure'. Then this broker immediately generates a corresponding flood notification message and sends it to neighbouring broker/brokers along all operational interfaces. Conversely, if a node receives any kind of Heartbeat message from a 'failed' link, our scheme will test to see if this link has stably recovered by sending link test notification message whilst keeping the link state as 'failure'. There are three states for each interface and they are 'normal', 'waiting' and 'failure'. If an interface receives any type of Heartbeat message within the dead time interval, the dead time interval will be renewed to its largest number, similar to OSPF. In this situation, if this interface is in the state 'normal', it will continue to be 'normal'; if the state of this interface is 'failure', it will send a link test message. If a matched link test ACK received then it will be set to 'normal'; if the state is 'waiting'; only when a matched ACK is received, the state will change to 'normal'. If an interface has not received any type of Heartbeat message before the dead time interval reaches '0', the link for this interface will be set to 'failure' whilst continuing to wait for a Heartbeat message. If an interface needs to send out a flood notification message, the state of this interface will be set to 'waiting'. In this state, this interface periodically sends flood notification messages until a suitable ACK, with the right copy of the sent flood notification message, is received.

Flood Notification Message

A flood notification message will be generated only when a node detects a change of one of its adjacent links and will be sent to all the directly connected neighbouring brokers. At the same time, each link's state will be set to 'waiting' if it is previously

in the 'normal' state. This flood notification message's copies will be stored into each interface's ACK waiting list and a corresponding ACK timer for each copy in different interfaces will be initialized. If a matching ACK message is been received within the ACK time, the corresponding flood notification message will be removed from the ACK waiting list, otherwise another copy of the flood notification message will be sent out through that interface to make sure this change can be learnt properly. Only when the ACK waiting list is empty, the state of that interface will be returned to 'normal' again.

It should be noted that an ACK list is needed for each interface as multiple changes may arise causing several flood notification messages to be emitted prior to any acknowledgements. Each flooded message must therefore be matched to a specific ACK message

Other brokers who receive a valid flood notification message forward this message to all their neighbouring brokers apart from the one sending this message. Nothing in the 'packet information' section will be changed when forwarding the message. A corresponding flood notification ACK will be sent back through the interface on which it was received no matter if this flood notification message is up-to-date or not. For a flood ACK message, the packet information section will remain the same as the original flood notification message, other than the ACK flag being set. A timer (ack_timer) for this flood message starts and is added to an 'ack_list record' prepared for each interface. Each flood message has its own ack_timer. If the interface does not receive a matched flood ack message while the ack_timer goes to '0', this interface will resend the corresponded flood message again and reset the ack_timer. If the interface receives a matched flood ack message before the ack_timer goes to '0', this record stored in 'ack_list record' will be deleted.

When a broker confirms a change (i.e. a failure or a recovery) has happened, it will go

through the following steps:

- Check the failure lookup table, find the colour, which corresponds to this failure or recovery, and update its CurrentC to the appropriate colour and increase its own SequenceNo by one.

- Generate flood notification messages with CurrentC, NodeID and SequenceNo written into the 'packet information' section. The 'type' field shows 'flood' and 'ACK flag' is set FALSE. These new flood notification messages are sent to all the directly connected neighbouring brokers even through the 'failure' link (in case it recovers)..

Link Test Notification Message

A *test link notification* message is generated when a message is received along a link that has been considered to be a 'failure'. For a robust system, it is necessary to have a mechanism to double check whether a link is fully recovered from a failure. This research proposes a method of sending a test link notification message through the 'failed' link and waiting for a response from the broker on the other end to make sure the link is functioning well by return of a specific acknowledgement.

ACK Messages

There are three different types of Heartbeat message, so there could be three corresponding ACK messages. However, for the regular Heartbeat message, a response is not expected. In future work, the research could use the combination of regular Heartbeat notification massage with ACK flag 'TRUE' to be a message which can record Round Trip Time [116] as in Transmission Control Protocol (TCP) to help dynamically configure the expiry timer in our algorithm. Table 6-5 shows the meaning of each message / acknowledgement combination.

Type list	ACK flag	ACK type
Flood	TRUE	Flood notification ACK message
LinkTest	TRUE	Link test ACK message
Heart-beat	TRUE	Round Trip Time test message

Table 6-5 List of All Ack Message Types

The packet information field in an ACK message is the same as the one in a notification message. The current colour, broker ID and the latest sequence number of that broker are listed in the packet information section. When a broker receives a Heartbeat message with 'flood' and 'TRUE' in the TYPE and ACK tag fields, respectively, it means that this is an ACK message responding to a flood notification message from a neighbouring broker. This broker will compare the information this message holds with its own knowledge in ACK list. If the ACK message matches the record in this interface's ACK list, the record will be removed. If a link test notification message is received through an interface, this broker will generate a link test ACK message sending it back through that interface. At the same time, if this interface is associated with a 'failure' and this is the first Heart-beat message that this interface received after the failure has happened, a link test notification message will be sent through that interface.

6.2 Node/Broker Operation with PCRT

For each interface in a node/broker, there are three different states: normal, waiting and failure. The transitions between those states are shown in Figure 6-5.
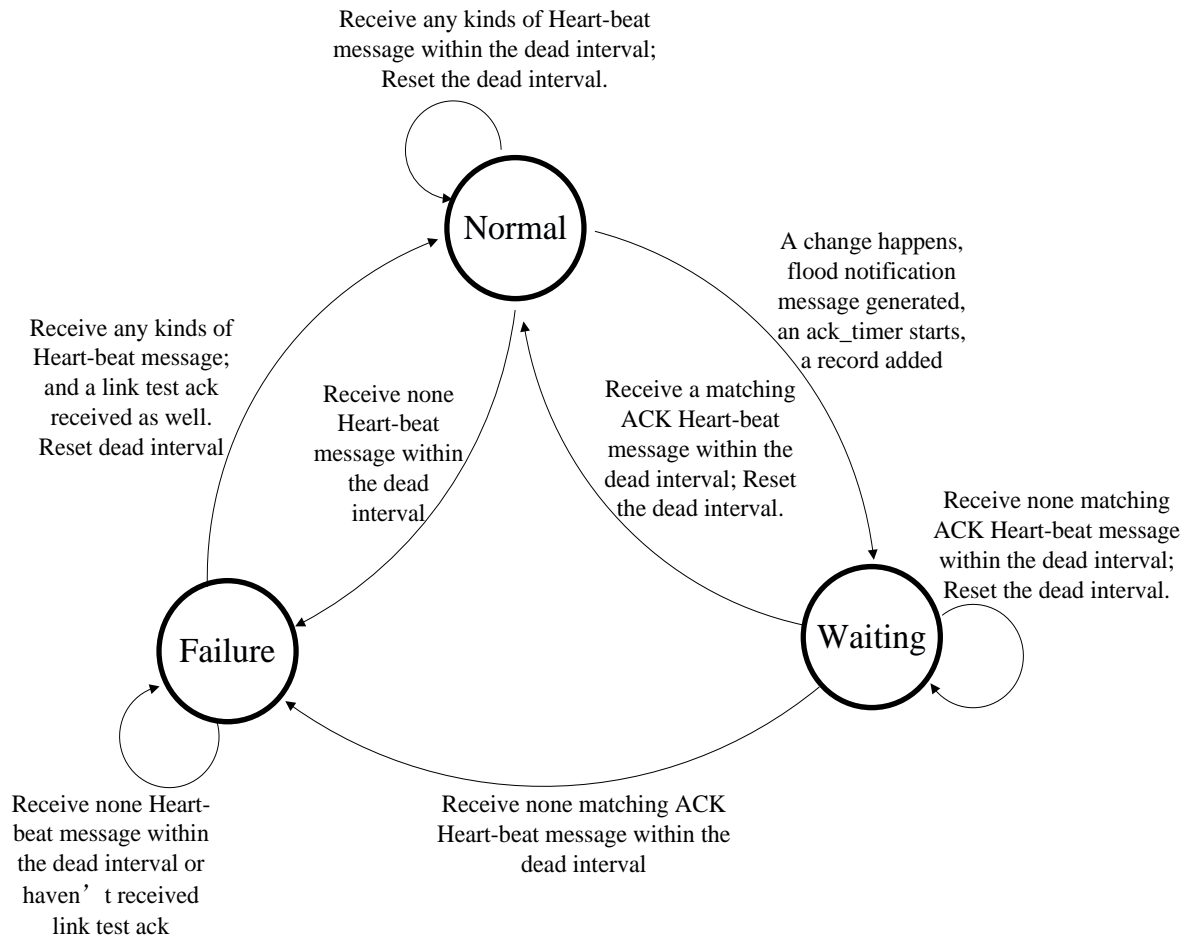
121

Receive any kinds of Heart-beat
message within the dead interval;
Reset the dead interval.

Normal

A change happens,
flood notification
message generated,
an ack_timer starts,
a record added

Receive any kinds of
Heart-beat message;
and a link test ack
received as well.
Reset dead interval

Receive none
Heart-beat
message within
the dead
interval

Receive a matching
ACK Heart-beat
message within the
dead interval; Reset
the dead interval.

Receive none matching
ACK Heart-beat message
within the dead interval;
Reset the dead interval.

Failure

Waiting

Receive none Heart-
beat message within
the dead interval or
haven't received
link test ack

Receive none matching ACK
Heart-beat message within the
dead interval

Figure 6-5 Interface States Transition Diagram

Each node has its own count down timer (the counting_down_timer automatically counts down based on the system timer) to periodically send out normal Heart-beat messages. When this timer reaches the value of '0', it will send out normal Heart-beat message through all interfaces irrespective of their current status. Then the timer returns to its maximum value to restart the countdown.

There are other timers in a node/broker. For all interfaces, they have (dead_time_interval) timers to check how long they haven't receive any thing and trigger further procedures. The max value of a dead_time_interval is four times the counting_down_timer (the same as OSPF's mechanism). If any kind of Heart-beat message is received by an interface, the dead_time_interval will be reset to its maximum value. If not, the dead_time_interval will continue to count down one unit

at a time until it reaches '0'. When dead_time_interval equals to '0', this interface will go to the 'failure' state. An interface will remain in the 'failure' state until a Link test ACK message is received and it will then return the to 'normal' state.

When a node/broker detects a change, it sends out flood messages along all the necessary interfaces (introduced in the previous section). Each interface then goes to the state 'waiting' or remains there. A timer (ack_timer) for this flood message starts and is added to an 'ack_list record' prepared for each interface. Each flood message has its own ack_timer. If the interface does not receive a matched flood ack message while the ack_timer goes to '0', this interface will resend the corresponded flood message again and reset the ack_timer. If the interface receives a matched flood ack message before the ack_timer goes to '0', this record stored in 'ack_list record' will be deleted. This repeats until the 'ack_list record' if empty, the state will go back to normal. It should be noted that an ack list is needed for each interface as multiple changes make arise during the time the first flood message is sent. Each flooded message must therefore be matched to a specific ack message.

Figure 6-6 provides a flowchart giving further details of the interface behaviour when it receives a heart-beat message:
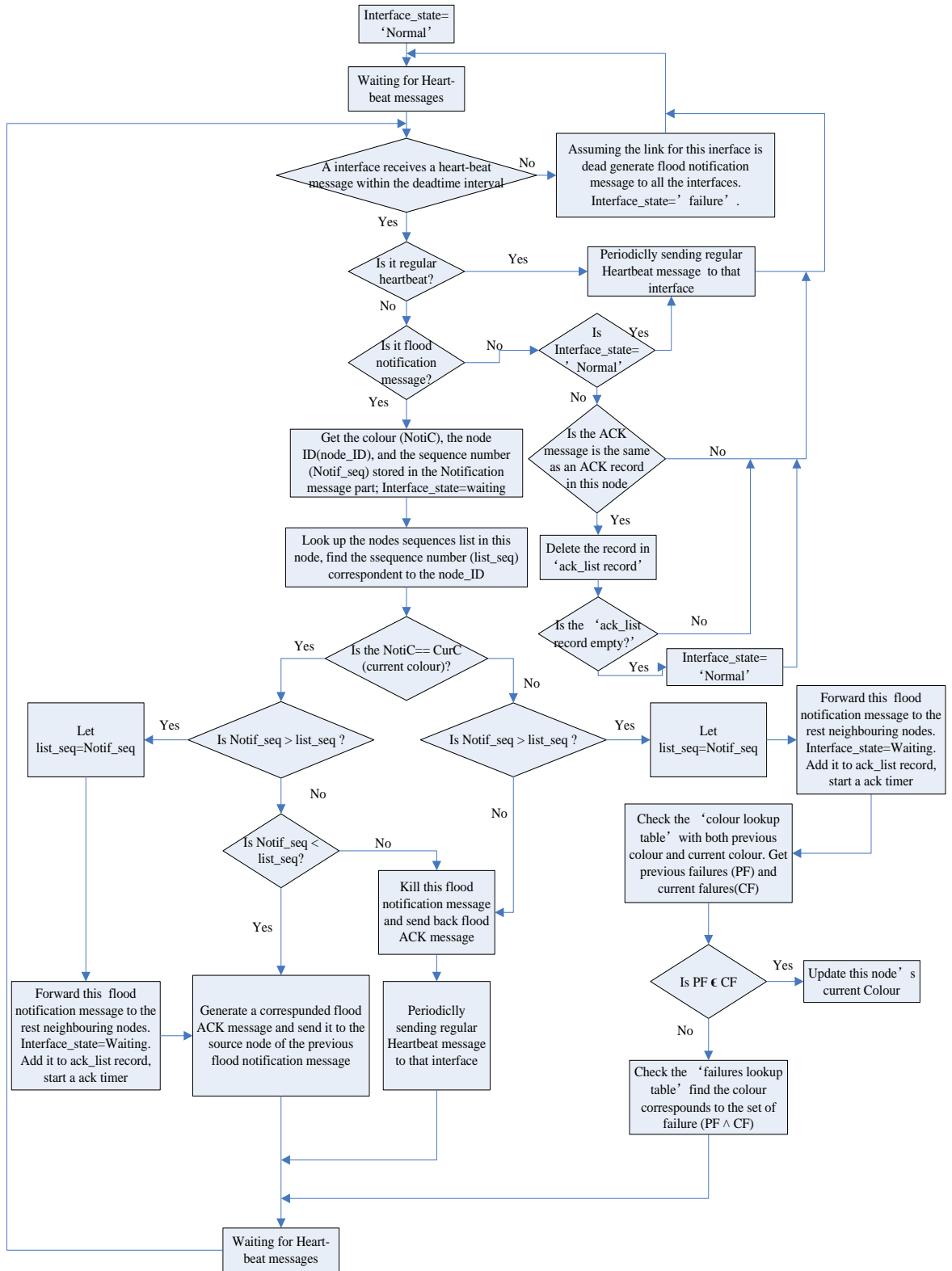
Figure 6-6 Flowchart for Interface Behaviour

The following pseudo-code shows how the overlay network reconverges when a change happens.

1. A node detected an adjacent link failure 'NewF'
2. **If** CurrentC==1
3. //CurrentC is the current colour of this node
4.   PreFail ← ColourLookupTable(CurrentC)
5.   **If** NewF∈PreFail
6.   NodeID.SequNo= NodeID.SequNo +1
7.   Heart-beat message (CurrentC, NodeID.SequNo,)
8.   **End if**
9.   **Else**
10.   Failure=NewF U PreFail
11.   Colour ← FailureLookupTable(Failure)
12.   RoutingTable ← RTsSet(Colour)
13.   NodeID.SequNo= NodeID.SequNo +1
14.   Heart-beat message (CurrentC, NodeID.SequNo,)
15.   **End else**
16. **End if**
17. **Else**
18.   CurrentC ← FailureLookupTable(Failure)
19.   RoutingTable ← RTsSet(CurrentC)
20.   NodeID.SequNo= NodeID.SequNo +1
21.   Heart-beat message (CurrentC, NodeID.SequNo,   )
22. **End else**
23. A node detected an adjacent link failure 'NewF'
24. **If** CurrentC==1
25. //CurrentC is the current colour of this node
26.   PreFail ← ColourLookupTable(CurrentC)
27.   **If** NewF∈PreFail
28.   NodeID.SequNo= NodeID.SequNo +1
29.   Heart-beat message (CurrentC, NodeID.SequNo,)
30.   **End if**
31.   **Else**
32.   Failure=NewF U PreFail
33.   Colour ← FailureLookupTable(Failure)
34.   RoutingTable ← RTsSet(Colour)
35.   NodeID.SequNo= NodeID.SequNo +1
36.   Heart-beat message (CurrentC, NodeID.SequNo,)
37.   **End else**
38. **End if**
39. **Else**
40.   CurrentC ← FailureLookupTable(Failure)
41.   RoutingTable ← RTsSet(CurrentC)
42.   NodeID.SequNo= NodeID.SequNo +1
43.   Heart-beat message (CurrentC, NodeID.SequNo,   )
44. **End else**

6.3 **Simulation Results**

Some researchers who focus on optimizing networks, express concerns that pre-calculating all possible routing tables and storing them in routers could lead to a router 'out of memory' problem or slower performance [210]. This thesis proposes to condense routing tables and remove redundant entities, so each broker will only need to store a single super condensed routing table.

A broker is an application platform, which needs to be embedded into a third part container. The memory that the broker is allowed to use is not determined by the amount of memory allocated to the platform (i.e. it could be a Java Virtual Machine). Although the broker is constrained by the amount of memory given to the platform, the broker manages its memory independently. If any issues with an OutOfMemory were to happen, the administrator could increase the broker storage memory and additional data storage files until reach the limit of the container. Normally the container (being a server/PC) can have quite a large memory capacity.

Potentially, each broker would require its own routing tables set for both normal and failures situations. If there were n links, including all single link failure situations and the normal situation, there would be $(n+1)$ routing tables. However, we propose constructing a condensed super routing table with all possible unique entries from those $(n+1)$ routing tables. Redundant entries are omitted, saving space; so our main concern becomes the fast selection of the suitable next hop entry for arriving messages.
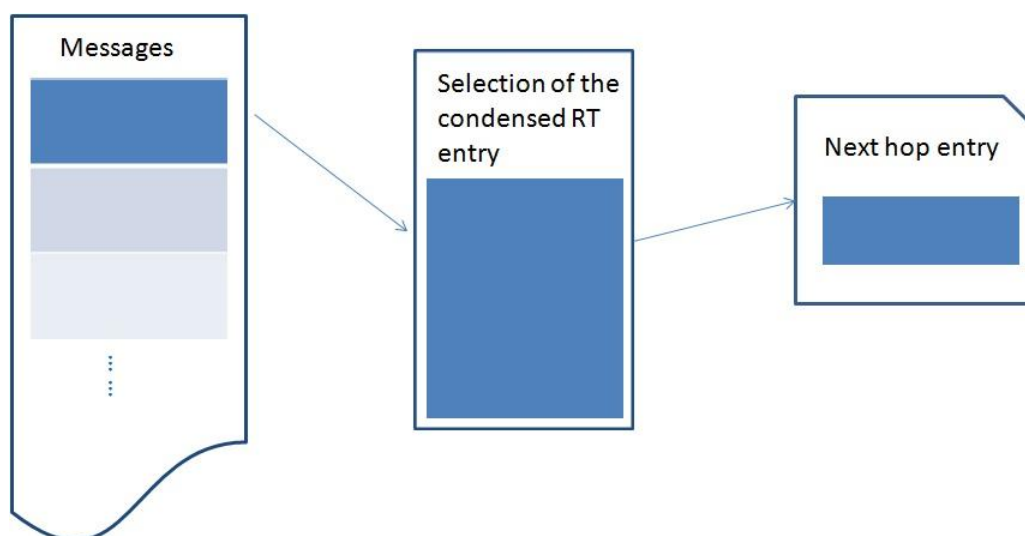
Figure 6-7 PCRT Selection Mechanism

This thesis considers two different selection mechanisms. One is a well-known hardware device called a content-addressable memory (CAM) and another is a two-dimensional array data-structure.

A CAM provides a lookup table function within a single clock cycle by using dedicated comparison circuitry. It compares input search data against a table with pre-stored data, and then returns the address of the matched data [117, 118]. Nowadays, the most popular commercial application of CAMs is for classifying and forwarding Internet Protocol (IP) packets in network routers [119, 120].

However, instead of employing expensive hardware equipment, here this thesis we focus on a proposal using a software data-structure, i.e. a two-dimension array, to locate the address in the pre-stored lookup table corresponding to the matched input data. Without using CAM to compare input data with data stored in lookup table, a two-dimensional array will assist locating the address of the data without the need for a search operation. In this two-dimensional array, the first dimension indicates the current colour and the second dimension shows the destination broker. Based on the condensed super routing table, the two-dimension array is filled with a pointer to the appropriate next hop entry in the condensed super routing table. The use of a pointer

into the routing table, rather than storing the next-hop entry directly in the 2-dimensional array is that more complex records are held in the routing table, such as counters, link state information and so forth.

The following is the structure of the two-dimension array.

$$
A = \begin{matrix} Dest.Bro\,ker \\ \downarrow \end{matrix} \overset{Colour \rightarrow}{\begin{bmatrix} a_{1,1} & a_{1,2} & ... & a_{1,n} \\ a_{2,1} & a_{2,2} & ... & a_{2,n} \\ ... & ... & ... & ... \\ a_{m,1} & a_{m,2} & ... & a_{m,n} \end{bmatrix}}
$$

(6-2)

Given the topology shown in Figure 6-1) as an example, the lookup two-dimension array will be the following and number 1 represents this broker itself.

$$
A = \begin{matrix} Dest.Bro\,ker \\ \downarrow \end{matrix} \overset{Colour \rightarrow}{\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 3 & 2 & 2 \\ 3 & 3 & 2 & 3 \end{bmatrix}}
$$

(6-3)

Figure 6-8 shows how the combination of the 2-D lookup array and a single condensed routing table provides an efficient data structure arrangement. Multiple colour / destinations can be mapped to a single next hop entry. In addition no searching of the routing table is undertaken. Providing the index information (i.e. the broker's current colour and the packet's ultimate destination) into the array quickly obtains the appropriate pointer that is then used to obtain the next hop information.
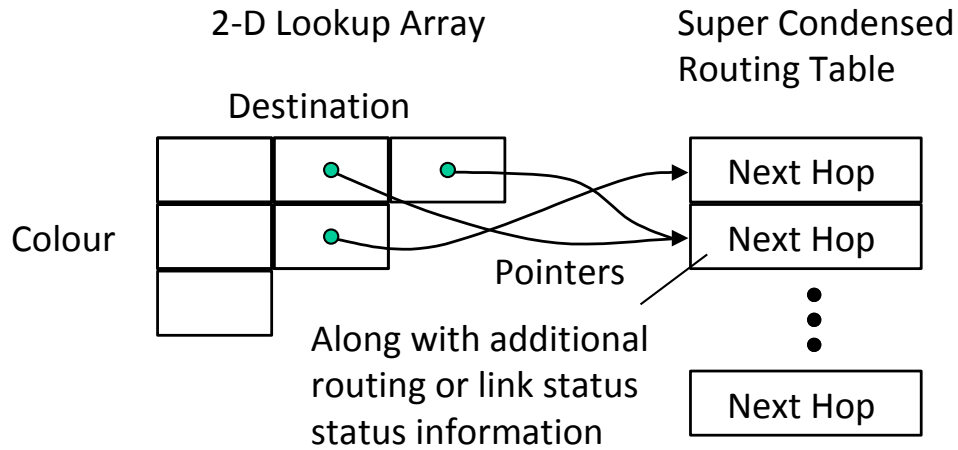
Figure 6-8 Relationship between 2D Lookup Array and Condensed Routing Table showing Many to One Mapping

The super broker prepares the routing tables for different failure situations for all nodes, followed by the creation of the condensed super routing table and the lookup two-dimensional array. This is relatively time-consuming. However, as this is performed during initialisation, this is of little consequence. Our concern is whether it is time consuming to find the right next hop entry via this array. Figure 6-8 shows the time, in milliseconds on a computer with an Intel i3 CPU, M 350 @ 2.27GHz and 2.00 GB RAM, of accessing different sized two-dimensional arrays a million times. We can see from this figure, for running the lookup once, it will take average seconds. This time for searching once is so small we can thus ignore it.
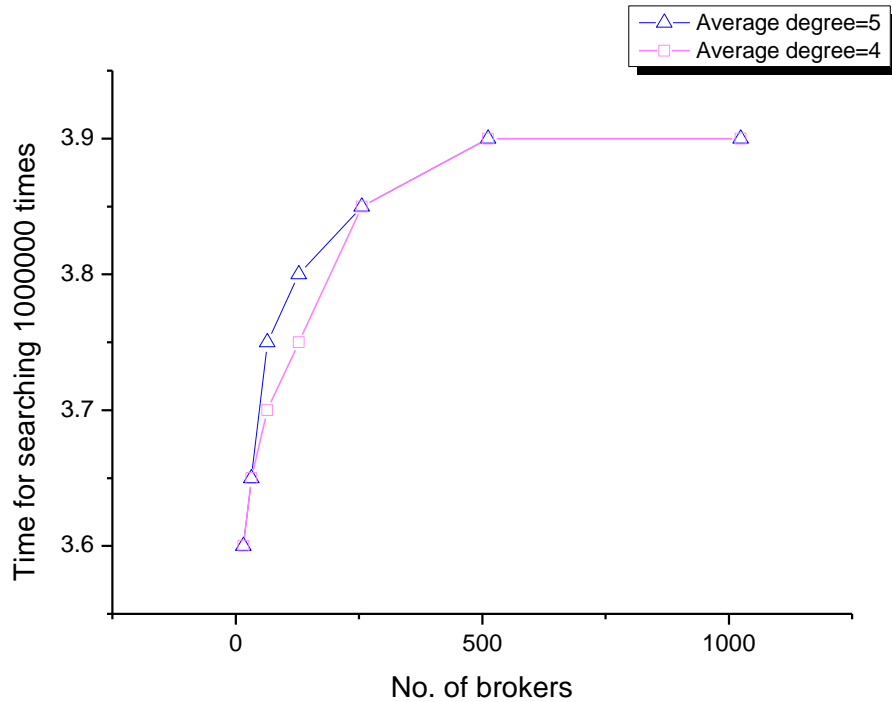
Figure 6-9 Time for locating the right address in two-dimensional arrays

6.3.1   Shortest Path First Guarantee

A mission critical MOM overlay network, such as a stock market application, could exist over quite a large geographical area or maybe worldwide. Between two brokers, an overlay link may be supported by long underlying network path. Therefore, to guarantee the shortest path first routing in the overlay is highly desirable. One hop more in the overlay network might lead to a message traversing a considerably longer physical distance. Compared with many proposed network recovery algorithms, to our knowledge, our scheme is the only one that guarantees shortest path when building the routing table for various failure situations. Figure 6-10 shows a comparison of path length for a 32 node - 64 link network for different network recovery abstractions.
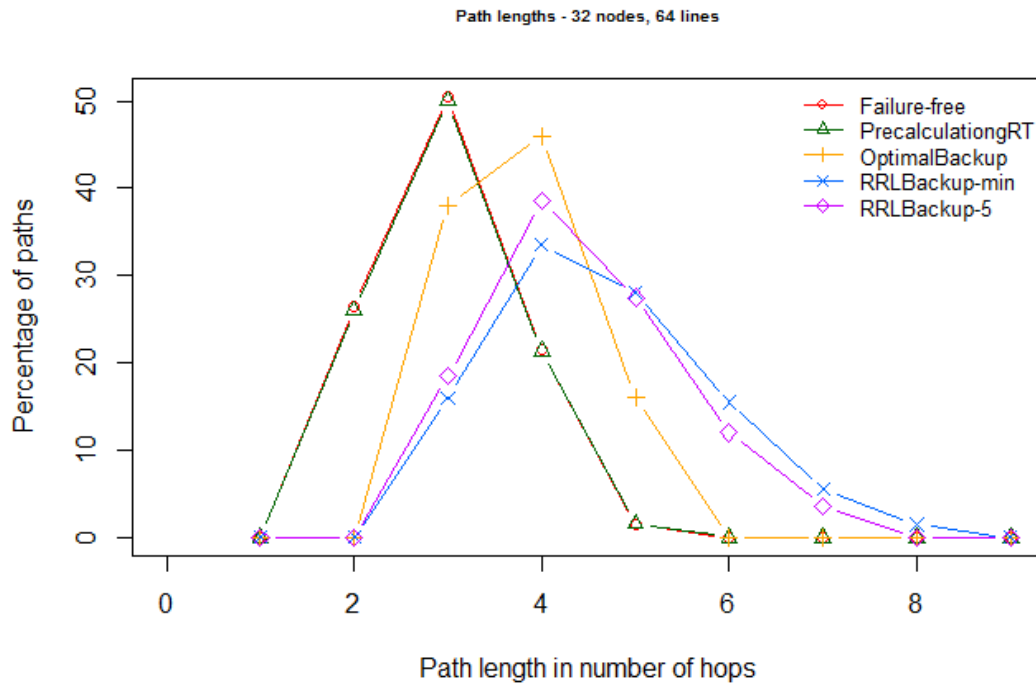
130

Figure 6-10 Path Length of a 32 Nodes, 64 Links Topology

Experiments are based on single link failure for a 32 nodes and 64 links network topology. By failing links one per time, we can calculate the total number of each different path length and then calculate each path length's percentage among all the paths. From Figure 6-10, it can be seen that the proposed PCRT recovery method is most closely matches the failure free situation which is to say our algorithm has a better performance in providing the shortest paths than the other schemes considered [210].

Although Resilient Routing Layers (RRL)[110] claims to scale well and shows several results to indicate that six layers are more than enough to support a very large network (1024 nodes and an average node degree of 4), our proposed algorithm, PCRT, only needs one condensed super routing table to cover all possible situations. Moreover, the most important shortcoming of RRL is that it cannot guarantee the shortest path at all when a failure happens and the fewer routing layers there are the higher the possibility of employing longer paths. This is the trade off of RRL—less

131

routing tables or shorter paths.

Figures 6-11 and 6-12 show a comparison between the average routing entries stored at a node without using condensed super routing table and the number of entries for that node when the condensed super routing table is employed. The difference between Figure 6-11 and Figure 6-12 is the scale of y-axis. In the Figure 6-11, y-axis is exponentially distributed and the y-axis is distributed in the Figure 6-12 according to the average.
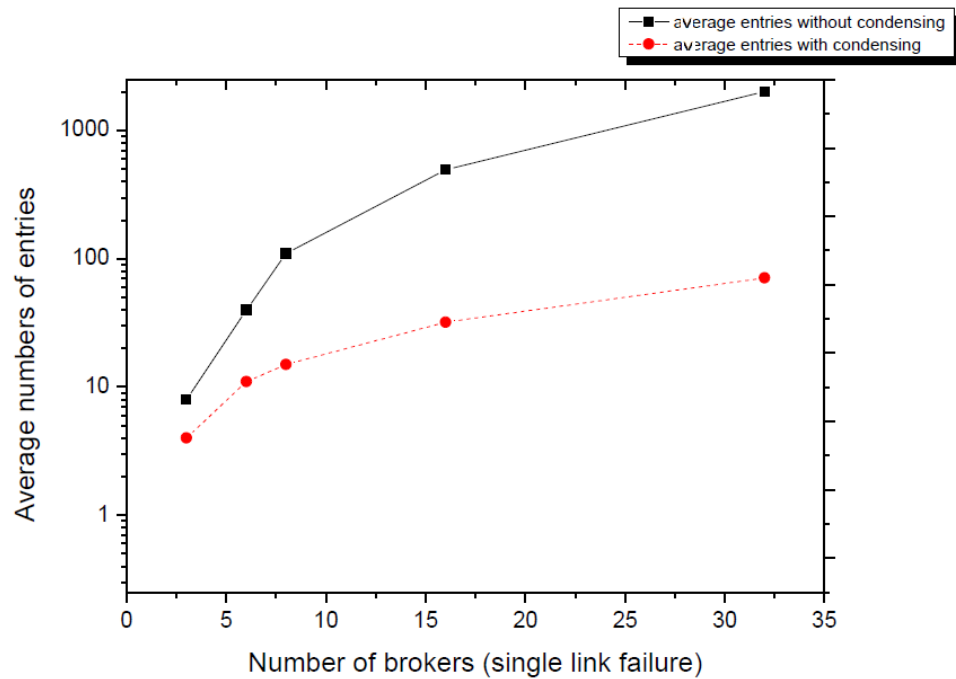

Figure 6-11A Comparison between the Average Entries for Different Measurements (part 1)

Figure 6-11 indicates that with increasing of number of brokers, the average number of entries increases in those two different measures The average entries a node will store without using condensed super routing table is always greater than the average entries this node can have when the condensed super routing table is applied.
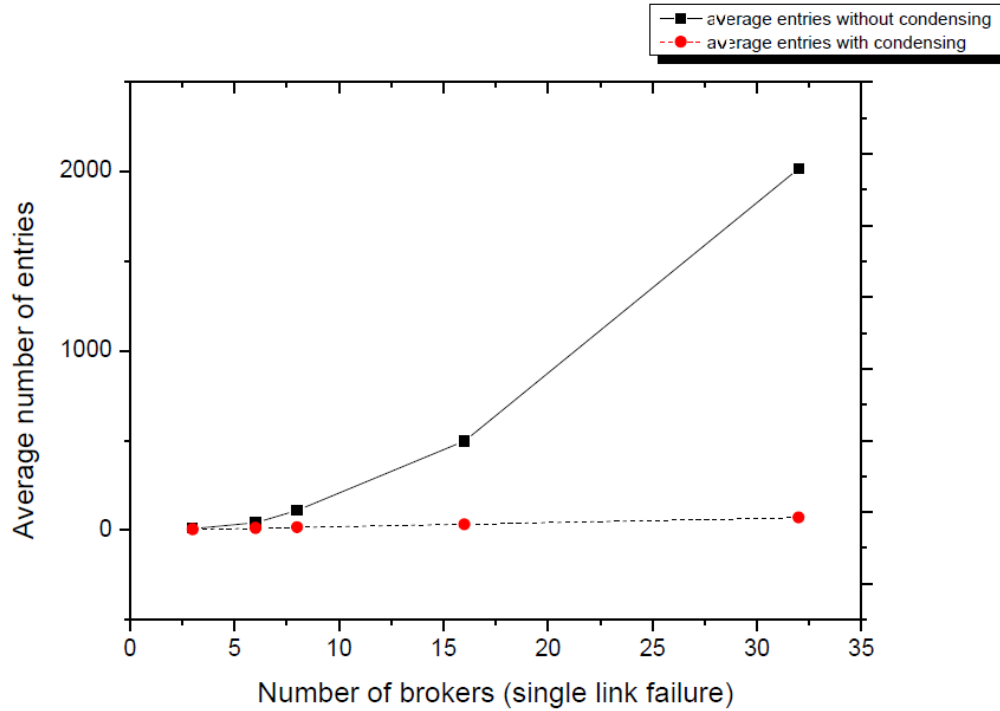
Figure 6-12 A Comparison between the Average Entries for Different Measurements (part 2)

From Figure 6-12, the increasing trend of those two lines under different measurements are shown clearly. The average number of entries a node will store without using a condensed super routing table will grow exponentially with an increasing number of brokers [100] and the average entries this node can have when the condensed super routing table is applied results in linear growth [100].

Let N be the total number of brokers, D be the average degree of the given topology and L be the total number of links this given overlay network has. Now, for the single link failure only situations, the total number of routing tables RTs will be:

$$RTs = L + 1 \qquad\qquad (6\text{-}4)$$

because there are L different failures possibility and one routing table for the normal situation.

In a routing table, every broker, except the broker storing this routing table, will be listed as a reachable entry. Therefore, the total entries E a routing table can have will be:

$$E = N - 1 \qquad (6\text{-}5)$$

Since the average degree of the given topology is known, the total number of links of this topology will be:

$$L = N \times D \div 2 \qquad (6\text{-}6)$$

Based on Formula 6-3 and Formula 6-6, the total number of routing tables will be:

$$RTs = N \times D \div 2 + 1 \qquad (6\text{-}7)$$

So the total number of entries $\sum E$ of all the routing tables a broker needs to store will be:

$$\sum E = \sum_{i=1}^{N} E_i = RTs \times E \qquad (6\text{-}8)$$

Based on the Formula 6-7 and Formula 6-5:

$$\sum E = N^2 \times D \div 2 + N(D \div 2 - 1) - 1 \qquad (6\text{-}9)$$

Based on the Formula 6-9, the trend of the black line with squares in Figure 6-8 is correct.

However, the complexity of an exponential growth shown in Formula 6-9 is $O(n^2)$ and the complexity of a linear growth is $O(n)$. Thus, the condensed super routing table can save huge amount of storage space and this advantage becomes more significantly when multi-failure situations are involved or as the overlay network grows.

Figure 6-13 shows the relationship between the average node degree and the total number of possible routing tables for a node. From this figure, the following observations can be made. Firstly, the larger the average node degrees, the larger number of routing tables required (without condensing). Secondly, the more brokers a topology has, the more number routing table variants there are. Then the more brokers a topology includes, the steeper the gradient. Those observations can be confirmed via formula 6-7
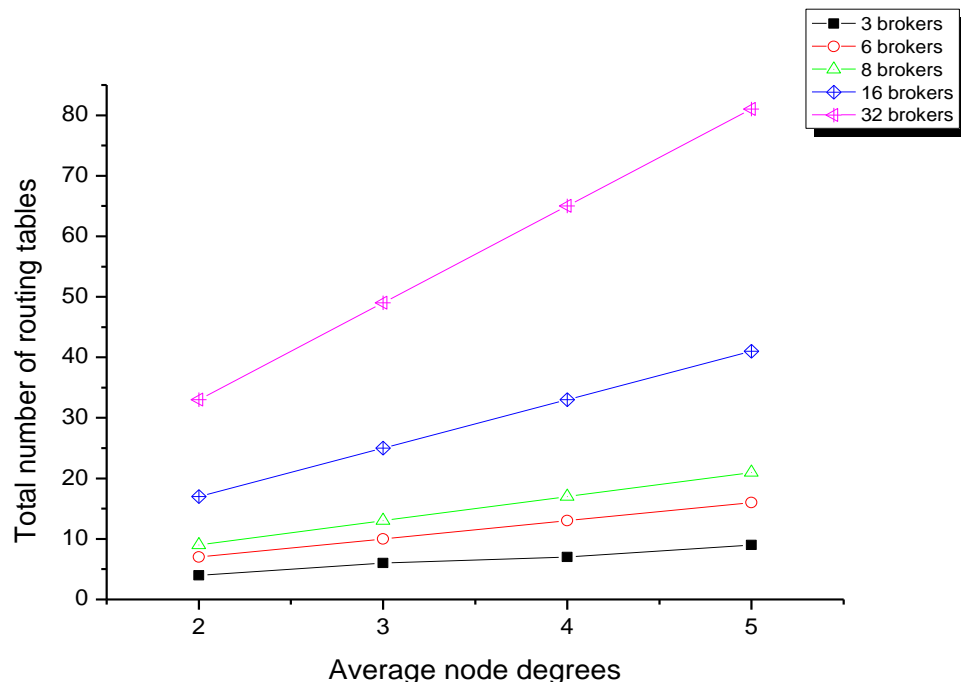

Figure 6-13 The Relationship between the Average Node Degree and Possible Routing Tables

6.3.2   PCRT / OSPF Performance Comparison

Using a bespoke packet-level simulator, to demonstrate the benefit of our scheme for fast recovery from failures, various topologies were constructed, including the simple 6-node one. This thesis compares PCRT and OSPF based on the topology shown in Figure 6-14 and has the following set up for all queues: the notional service bitrate is 10000bits/unit time with a buffer capacity of 10 packets; the data packet generation rate at the source is 1 per unit time at a fixed interval; the data packet size is 1000 bits.
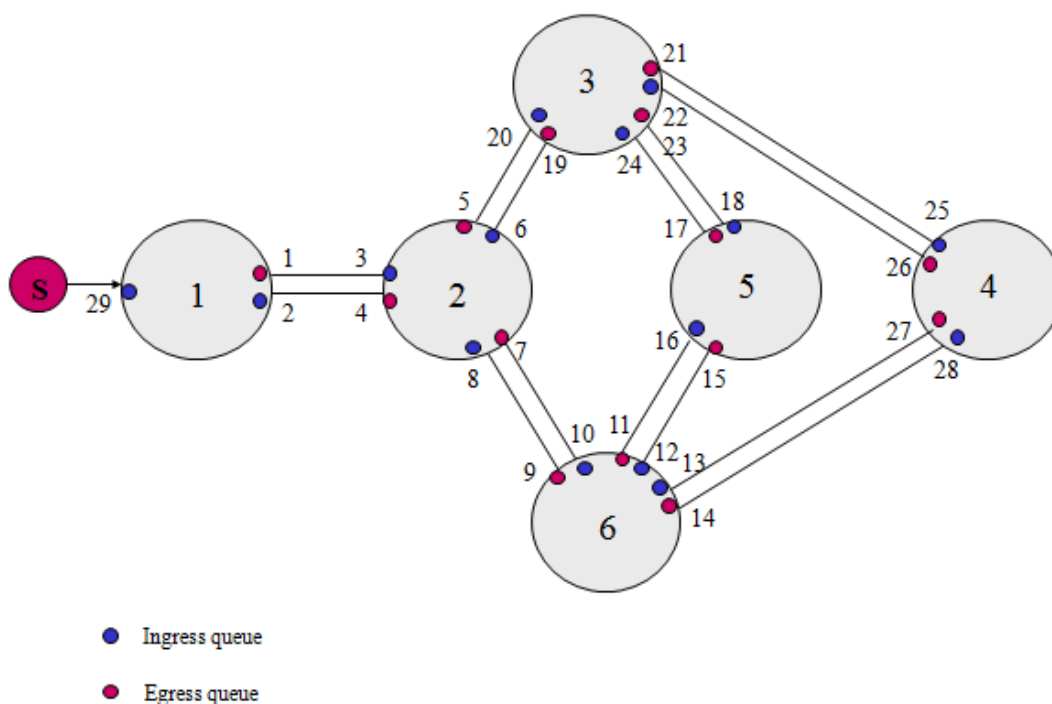

Figure 6-14Six Nodes/Brokers Topology

There is a source attached to the queue 29. All data packets will be sent to node/broker 3. Based on the routing table under the normal situation, the chosen path is node 1-→node2-→node 3 (i.e. least hop count). We run this simulation for 150 time units and fail the link between node 2 and node 3 at time 50 and recover it at time 100. While the failure is in progress, the path from source to destination will be

node 1→node 2→node 6→node 5→node 3.

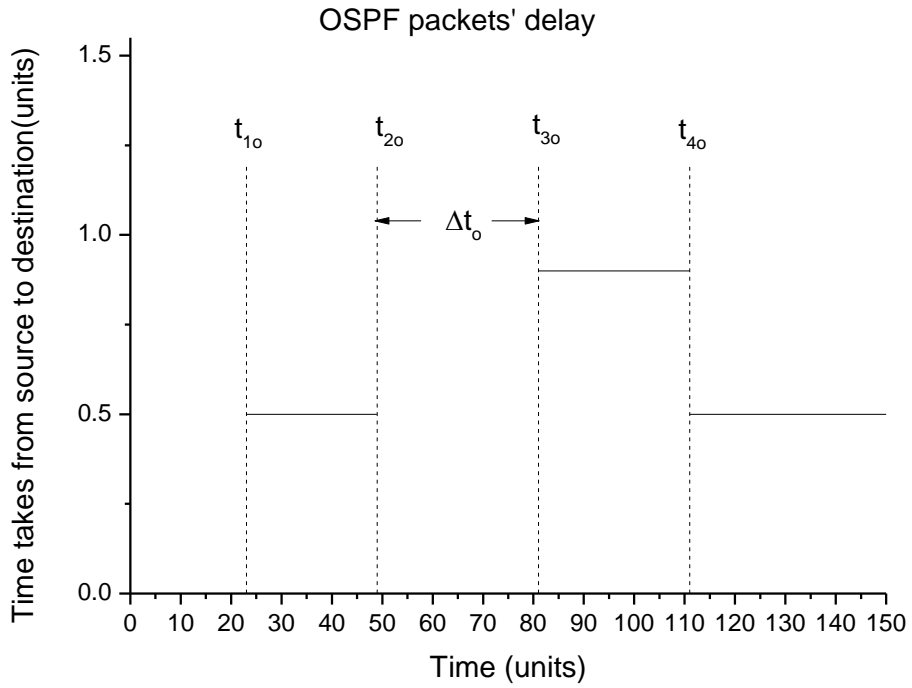The following figures (Figure 6-15 and Figure 6-16) show the time for a packet to traverse from source to destination:

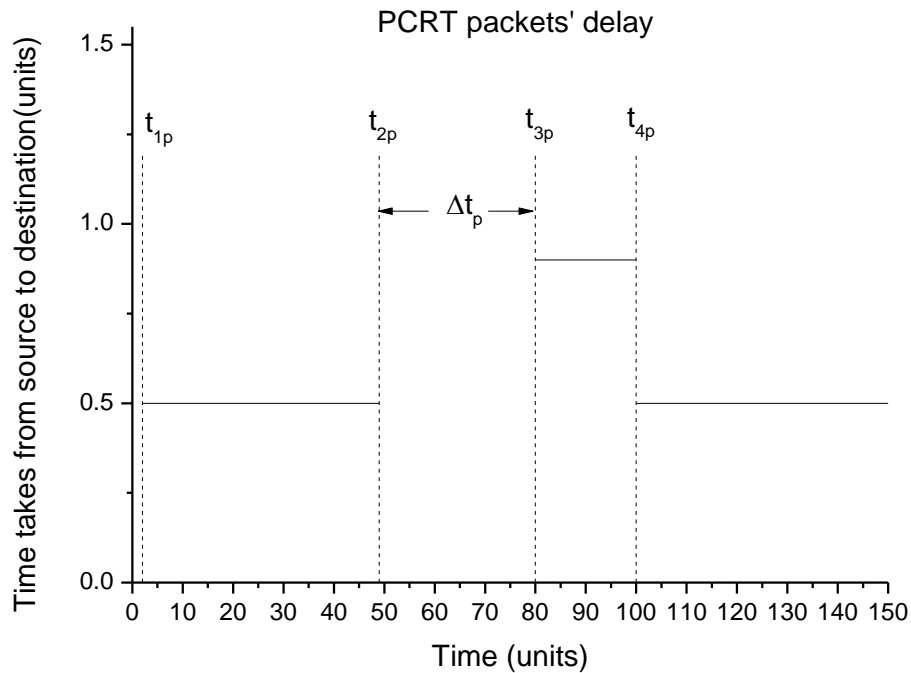Figure 6-15 End-to-End Packet Transfer Latency for Fast OSPF

Figure 6-16 End-to-End Packet Transfer Latency for PCRT

t_{1o} and t_{1p} are the time at which this system starts to send packets; t_{2o} and t_{2p} are the time at which the link between node/broker 2 to node/broker 3 fails; t_{3o} and t_{3p} are the time at which this system reconverges and successfully delivers packets again; t_{4o} and t_{4p} are the time in which the system starts to use the recovered link and successfully deliver packets via the original path; Δt s are the time it takes for this system to reconverge. The PCRT does not need to exchange LSA messages as the whole network has been set up in advance. PCRT gets converge faster than OSPF, although for 'fairness' we set the heartbeat interval to be the same as the OSPF hello interval, so the detection times are the same. Even so the processing time of PCRT is less leading to a quicker switch to the recovery shortest path. In addition, PCRT is able to use the healed link quicker than for OSPF. Figure 6-17 shows the package loss during the time the system reconverges.
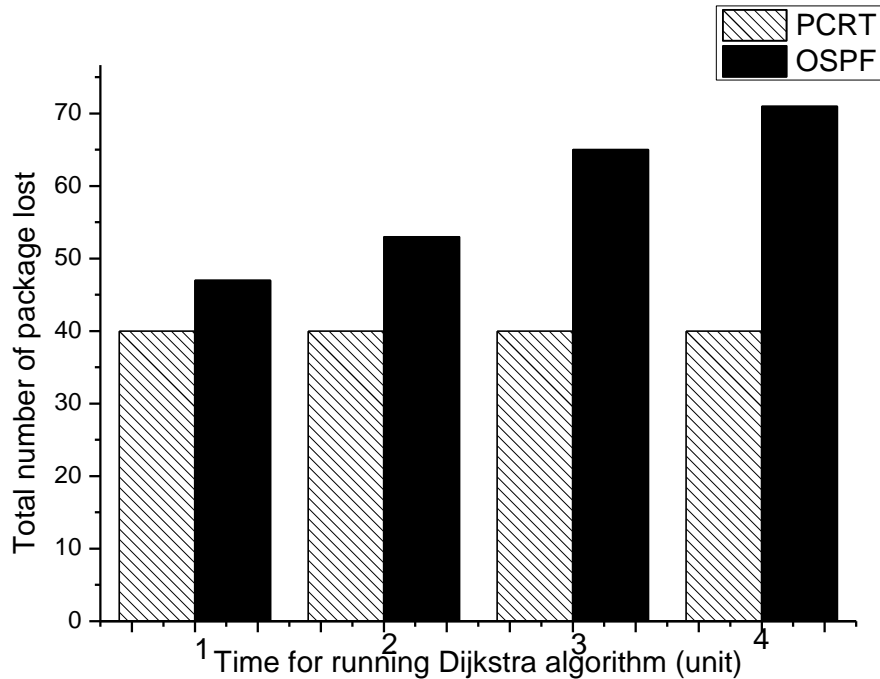
Figure 6-17 Reconvergence Packet Loss for OSPF / PCRT

As the reconvergence time is still notably longer for 'fast' OSPF more packets are lost until an alternative path is found to the destination. For PCRT, the updated routing table information is ready to-hand as soon as the failure is discovered.

6.4 **Condensed Super Routing Table Validation for PCRT by using Model Checker**

The resilience and QoS of the messaging substrate plays a critical role in the overall system performance as perceived by the end users. While current systems and standards provide essential features of reliability, security, transactionality and persistence, there is little consideration for real-time QoS such as end-to-end latency and resilience.

The advantages of overlay network routing have been discussed in Chapter 2.

However, sometimes human being will change the configuration of the routing table based on end user requirements or finding a shorter path geographically or predictable load balancing. As long as human beings are involved, it is necessary to have a mechanism of verifying this new configuration. On the other hand, even if no change has been made by human beings, a verification procedure can also guarantee the correctness of generating the condensed super routing table. Therefore, after the program has automatically generated the condensed super routing table and the administrator has changed the configuration based on policy requirements, a verification procedure is applied for each broker.

The data structure of a condensed super routing table has been described in the previous section. Comparing the model-checking model in Chapter 5, in this situation, the routing table is available and a new factor 'Colour' should be taken into consideration in the states transition conditions. This research will not verify the reachability or detect loops for the period when a failure happens and the overlay network has yet to converge. This is because if the overlay network has not converged and the brokers are under different colours, the reachability verification might fail due to temporary inconsistencies. However these kinds of errors are acceptable as they will only last milliseconds and the network will soon reconverge. Therefore, this research verifies the reachability and performs loop detection for each broker under the same colour, for all possible colours.

The condensed super routing tables (Table 6-6, Table 6-7, Table 6-8 and Table 6-9) for broker 2 and broker 3 are listed as following:

Colour 1		Colour 2		Colour 3		Colour 4	
Ult. Destination broker	Next hop broker	Ult. Destination broker	Next hop broker	Ult. Destination broker	Next hop broker	Ult. Destination broker	Next hop broker
1	1	1	3	1	1	1	1
3	3	3	3	3	3	3	1

Table 6-6 A Part of Routing Tables Stored in Broker 2

Super routing table			
Address	Ult. Destination broker	Next hop broker	Colour
1	1	1	1,3,4
2	1	3	2
3	3	3	1,2,3
4	3	1	4

Table 6-7 The Super Routing Table for Broker 2

Colour 1		Colour 2		Colour 3		Colour 4	
Ult. Destination broker	Next hop broker	Ult. Destination broker	Next hop broker	Ult. Destination broker	Next hop broker	Ult. Destination broker	Next hop broker
1	1	1	1	1	2	1	1
2	2	2	2	2	2	2	1

Table 6-8 A Part of Routing Tables Stored in Broker 3

Super routing table			
Address	Ult. Destination broker	Next hop broker	Colour
1	1	1	1,2,4
2	2	2	1,2,3
3	1	2	3
4	2	1	4

Table 6-9 The Super Routing Table for Broker 3

$$A = \underset{\downarrow}{\overset{Dest.Broker}{}} \overset{Colour \rightarrow}{\begin{bmatrix} 1 & 1 & 2 & 1 \\ 2 & 2 & 2 & 1 \\ 3 & 3 & 3 & 3 \end{bmatrix}}$$

(6-10)

In the PCRT simulator, the routing tables for all the concerned situations and condensed super routing table as well as the two-dimensional lookup array can be generated automatically. After the administrator updates the configuration, the validation procedure commences. The NuSMV model checker builds a topology model based on the super routing table. The following is part of the program showing how to select the condensed super routing table entry:

```
next(loc) :=
    case
        loc=1 & colour=1 & loc_des=1:1;
        loc=1 & colour=1 & loc_des=2:2;
        loc=1 & colour=1 & loc_des=3:3;

        loc=1 & colour=2 & loc_des=1:1;
        loc=1 & colour=2 & loc_des=2:3;
        loc=1 & colour=2 & loc_des=3:3;
            ⋮
```

This part of the code shows all the possible entries when the current colour is '1' and a message is currently located at broker 1. 'Colour' is a new state variable in the super routing table model. For example, if this message's utility destination is broker 3, the next hop will be broker 3. The remaining code is similar to that described in Chapter 5.

The following blocks provide validation simulation results:

```
Input:
NuSMV > go
NuSMV > pick_state –r
NuSMV > print_current_state –v

Output:
Current state is 1.1
loc = 1
colour = 1
loc_des = 3
```

The above block sets up the three broker model.

```
Input:
NuSMV > simulate –r 3
Output:
******* Simulation Starting From State 1.1 *******

Input:
NuSMV > show_traces –t
Output:
There is 1 trace currently available.

Input:
NuSMV > show_traces –v
Output
<!-- ############### Trace number: 1 ############### -->
Trace Description: Simulation Trace
Trace Type: Simulation
->State: 1.1 <-
     loc = 1
colour = 1
loc_des = 3
->State: 1.2 <-
     loc = 3
colour = 1
loc_des = 3
->State: 1.3 <-
     loc = 1
colour = 1
loc_des = 3
```

```
->State: 1.4 <-
      loc = 1
colour = 1
loc_des = 3
```

The above block traces a route for three brokers model simulation.

```
Input
E:\NuSMV\bin>NuSMV PCRTthreeBrokers.smv

Output
--Specification EF loc = loc_des is true
```

The above block shows results of verifying configurations

6.5 Concluding Remarks

The PCRT algorithm introduced in this thesis is simpler, faster, loop free and more robust than several alternative state-of-the-art fast recovery mechanisms considered. Employing a new type of message named 'Heart-beat' message and creatively using 'colour' to identify each different failure scenarios, this approach can save time associated with flooding link state database information and running Dijkstra's algorithm to calculate new routing tables. In addition, the PCRT algorithm can decrease the time between sending regular Heartbeat notification messages to discover a failure faster without leading to a large amount of flooded messages.

So far, the PCRT algorithm has focused on single link failures and the scheme appears to work well. Many researchers have expressed concern about pre-calculation of routing tables and their storage in a node/broker. Firstly, for a MOM overlay network, there not be numerous brokers so there will not be a great number of routing tables. Furthermore, based on this research, a mechanism for creating a condensed super routing table has been provided which saves a considerable amount of storage space for the brokers.

It is feasible for this PCRT algorithm to handle multi-link failures and node/nodes failures. Since all the calculations and condensing happen before launching the whole MOM system, there is enough time to configure data-structures for all possible failures combinations and condense routing tables into one super routing table. Then, modification and verification will be undertaken to guarantee the correctness of the configuration under each different colours (states) for a resilient mission critical MOM system.

7 Conclusions and Future Work

7.1 Conclusions

Mission critical message oriented middleware has been widely used in modern network systems, such as some wireless sensor networks [97] and some Internet-enabled enterprise systems [98]. However, the primary disadvantage of many message oriented middleware systems is that they require an extra component in the architecture, the message transfer agent (message broker). As with any system, adding another component can lead to a reduction in the performance and reliability, and can also make the system as a whole more difficult and expensive to maintain. The goal to this research is to find a suitable way to retain the performance and reliability for a MOM system after a failure or adding or removing components such as brokers.

This work provides a means to proactively address possible failures in three ways. Firstly, before launching a mission critical MOM system, we propose a mechanism for checking the configuration and every topic's reachability. Secondly, pre-calculate routing tables are generated under different failure situations, which may include all single link failure, certain double link failures or other multi-failures as well as node failures. When all the possible routing tables are created, a super condensed routing table will be generated where redundant information is omitted. Thirdly, given that modification can be made to routing information by humans (e.g. administrators) in response to traffic engineering preferences or policy decisions (i.e. static routing entries), we propose verifying the configuration of this super condensed routing table at a node whenever it is altered.

This research automatically generates a publish/subscribe tester when a link table and a topic table are provided. The Java code generator that this research has developed

146

allows any user to build a model checking mechanism for a publish/subscribe system, even if the user has little knowledge of programming languages or of NuSMV. As mentioned in Chapter5 the number of topics may affect the number of verification rules. In our simple six-broker model, where there are only four different types of topics, the verification time can be neglected. However either in commercial products or in more complexes publish/subscribe system models the number of topics can be far greater. In this case, the number of topics could become a critical factor in the efficiency of the model checking.

With the increasing number of brokers, manually generating a routing table for a large-scale MOM system is extremely complex. This research has designed a GUI interface to assist users to build a model and reduce the possibility of erroneous data entry. By using the GUI interface, users can input link information of brokers, which is much easier to obtain or understand than a routing table, and the program automatically generates a model of the link information. In this research, the routing table is then automatically generated based on a shortest path first algorithm.

However, verifying the reachability and reconfiguration are not enough for ensuring the resilience of MOM systems. It is necessary to quickly recover from overlay network failures. So a verified pre-stored super routing table for all or most possible failures provides a way to improve the performance of a resilient mission critical MOM.

7.2 Future Work

There can be several improvements for the current research.

1. When the model checker finds a misconfigured router or an unreachable path for a specific topic, a new function should be added to automatically resolve this fault rather than just highlighting its discovery. When an unreachable path is found, all

the sub-paths for that unreachable path could be checked one by one and errors rectified.

2. This research could provide additional functions for verifying a realistic MOM system. Since lots of MOM systems are expected to provide an efficient and high quality message service, this research could help to check the rules for guaranteeing the end-to-end latency by using a probability model checker. Every broker would have a latency property and this research will check if paths are able to ensure a topic can reach its destination broker within the latency constraint [44].

3. In future, we will focus on additional means of improving the failure response time of the PCRT scheme. Similar to the smoothed Round Trip Time mechanism, used in the Transmission Control Protocol (TCP) to optimise the Retransmission Time-Out (RTO) timer, our scheme could dynamically calibrate the transmission interval between heartbeat messages based on traffic load in the underlying network which affects the overlay link performance. On a 'quiet' link heartbeat messages are sent more frequently and the expiry time of link integrity timer, which is reset when heartbeat messages are received, could be set to a smaller value. This would allow for a rapid response to a link integrity failure. Conversely, when the overlay link is busy the expiry time would be increased to allow for the additional load. Additionally we might choose to increase the heartbeat emission interval to reduce the overhead on the link.

References

[1]N. Ibrahim, Grenoble.'*Orthogonal Classification of Middleware Technologies*'2009Third International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies.

[2] A.F. Xiao, Y.B. Li.'*Design of Message-Oriented Middleware of Distance Teaching Platform Based on Distributed Message Control*' 2010 International Conference on Computational Aspects of Social Neworks.

[3]H. Li and G. Jiang.'*Semantic message oriented middleware forpublish/subscribe networks*'2004 Sensors, and Command, Control,Communications,and Intelligence (C3I) Technologies for HomelandSecurity and Homeland Defense III. In proceedings of the SPIE.

[4]D. Lewis, J. Keeney, D. OSullivan, and S. Guo, '*Towards a managedextensible control plane for knowledge-based networking*',LectureNotes in Computer Science, Large Scale Management of DistributedSystems, Springer Berlin / Heidelberg, 4269/2006(0302-9743):98–111, 15 October, 2006.

[5]S. Parkin, D. Ingham, and G. Morgan.'*A message orientedmiddleware solution enabling non-repudiation evidence generationfor reliable web services*'.Lecture Notes in Computer Science,Springer Berlin / Heidelberg, 4526/2007(0302-9743):9–19, 06 June,2007.

[6]H. Abie. '*Adaptive Security and Trust Management for Autonomic Message-Oriented Middleware*'MobileAdhoc and Sensor Systems, 2009. MASS'09. 2009

[7] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. ElBadawi. '*Network Configuration in A Box: Towards End-to-End Verification of Network Reachability and Security*' In Proceedings of the 17th IEEE International Conference onNetwork Protocols (ICNP), pages 123–132, 2009.

[8]R. Alimi, Y. Wang, Y.R. Yang.'*Shadow configurationas a network management primitive*' In SIGCOMM '08: Proceedings ofthe ACM SIGCOMM 2008 conference on Data communication, pages 111–122, New York, NY, USA, 2008. ACM.

[9] TIBCO. *TIB/Rendezvous (White Paper),* 1999.

[10]A.L. Ananda, B.H. Tay, and E.K. Koh, '*A survey of asynchronous remote*

procedure calls'. SIGOPS Oper. Syst. Rev., 1992. 26(2): pages. 92-109.

[11] A.D. Birrell and B.J. Nelson, '*Implementing remote procedure calls'.* ACM Trans. Comput. Syst., 1984. 2(1): pages 39-59.

[12] Sun. Java Remote Method Invocation Specification. 2000; Available from: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html.

[13] OMG. '*CORBA Event Service Specification'.* 2001.

[14] OMG. '*The Common Object Request Broker: Core Specification'.* . 2002.

[15]B. BLAKELEY, H. HARRIS, J. LEWIS. '*Messaging and Queuing Using the MQI'.* 1995: McGraw-Hill, New York, NY.

[16] P. T. Eugster, P. Felber, R. Guerraoui, and A.M. Kermarrec, '*The many faces of publish/subscribe,*' EPFL, Tech. Rep. DSC ID: 2 000 104, Jan. 2001.

[17]H. Li and G. Jiang, '*Semantic message oriented middleware for publish/subscribe networks,*' in Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security and Homeland Defense III. In proceedings of the SPIE, vol. 5403, pages. 124–133, SPIE, April 2004.

[18] Y. Liu and B. Plale, '*Survey of publish subscribe event systems,*' Tech. Rep. TR574, Department of Computer Science (CSCI) at Indiana University, May 2003.

[19] R. Baldoni, M. Contenti, and A. Virgillito, '*The evolution of publish/subscribe communication systems,*' in Future Directions of Distributed Computing. Research and Position Papers, vol. 2584 of Lecture Notes in Computer Science, pages. 137–141, Springer, 2003.

[20] Red Hat Inc, '*redhat.com — MRG.*' http://www.redhat.com/mrg/.

[21] J. O'Hara, '*Toward a commodity enterprise middleware,*' ACM Queue, vol. 5, pages. 48- 55, May/June 2007.

[22] AMQP Working Group, '*Advanced Message Queuing Protocol.*' http://www.amqp.org/.

[23] AMQP Working Group, AMQP. '*A General-Purpose Middleware Standard',* *0-10* ed., 2008.

[24] Oracle Corporation, '*Java Message Service API' Rev. 1.1*, 2002. Available at http://java.sun.com/products/jms/.

[25] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Haase, '*Java Message Service API Tutorial and Reference: Messaging for the J2EE Platform*'. Addison-Wesley, 2002.

[26] S. Terry and T. Shawn, '*Enterprise JMS programming*'. New York, NY, USA: John Wiley & Sons, Inc., 2002.

[27] International Organization for Standardization, '*Information Technology - Database Languages - SQL.*' ISO/IEC 9075, 1992.

[28] Fiorano Software, Inc., FioranoMQTM: '*Meeting the Needs of Technology and Business*', 2004. Available at http://www.fiorano.com/whitepapers/ whitepapers_fmq.pdf.

[29] Tibco Software, Inc., '*TIBCO Enterprise Message Service*', 2004. Available at http://www.tibco.com.

[30] IBM Corporation, '*IBM WebSphere MQ 6.0*', 2005. Available at http://www-01.ibm.com/software/integration/wmq/.

[31] SpringSource, '*RabbitMQ*', 2010. Available at http://www.rabbitmq.com.

[32] E.Clarke, O. Grumberg, and D. Long. '*Model Checking*' 1990.

[33] M. C. Browne, E. M. Clarke and D. Dill. '*Automatic circuit verification using temporal logic: Two new examples. In Formal Aspects of VLSI Design*'. Elsevier Science Publishers (North Holland) 1986

[34][M. C. Browne, E. M. Clarke, D. Dill and B. Mishra. '*Automatic circuit verification using temporal logic*'. IEEE Transactions on Computers, C-35(12): 1035-1044. 1986

[35] M. C. Browne, E. M. Clarke and O. Grumberg. '*Characterizing finite kripke structures in propositional temporal logic*'. Theoretical Computer Science, July 1988

[36]M. Kot. '*The State Explosion Problem*' 2003[online]: http://www.cs.vsb.cz/kot/down/Texts/StateSpace.pdf

[37] R. E. Bryant. '*Graph-based algorithms for Boolean function manipulation*'. IEEE Transactions on Computers, C-35(8), 1986

[38]M. Pistore and M. Roveri '*The NuSMV Model Checker*' 2003[online]: http://www.cse.iitd.ernet.in/~sak/courses/foav/nusvm-iitd-2.pdf

[39]N. Rescher and A. Urquhart, '*Temporal Logic*' 1971 by Springer-Verlag/Wien,

ISBN 74-141565

[40]L. Baresi, C. Ghezzi, and L. Mottola, P. Leonardo da. Vinci, Milano.*'On Accurate Automatic Verification of Publish-Subscribe Architectures'*2007

[41] M. Panti, L. Spalazzi , S. Tacconi . *'Using the NuSMV Model Checker to verify the Kerberos Protocol'*. Istitutodi Informatica. Via Brecce Bianche: University of Ancona, 2001.

[42]O. Sheyner, J. Haines, S. Jha, R. Lippmann, & J. M. Wing. *'Automated generation and analysis of attack graphs'*. In Security and privacy, 2002. Proceedings. pages. 273-284.

[43]N. Chabrier, and F. François. *'Symbolic model checking of biochemical networks.'* Computational Methods in Systems Biology. Springer Berlin Heidelberg, 2003..

[44]H. Yang, M. Kim, KyriakosKarenos, F. Ye, and H. Lei,' *Message-Oriented Middleware with QoS Awareness* 'IBM T. J. Watson Research Centre 2009.

[45]M. Kim, K. Karenos, F. Ye, J. Reason, H. Lei, and K. Shagin. *'Efficacy of techniques for responsiveness in a wide-area publish/subscribe system.'* Proceedings of the 11th International Middleware Conference Industrial track. ACM, 2010.

[46]R. Z. Frantz, R. Corchuelo, and J. L. Arjona. *'An efficient orchestration engine for the Cloud.'* Cloud Computing Technology and Science (CloudCom), 2011 IEEE.

[47]S. Guo, K. Karenos, M. Kim, H. Lei, and J. Reason. (2011, June). *'Delay-cognizant reliable delivery for publish/subscribe overlay networks'*. In Distributed Computing Systems (ICDCS), 2011 31st International Conference on pages. 403-412 IEEE.

[48]R. Alur, T. A. Henzinger, and O. Kupferman. (2002). *'Alternating-time temporal logic'*. Journal of the ACM, pages 672-713.

[49]R. Alur, C. Courcoubetis, and D. Dill. (1990, June). *'Model-checking for real-time systems'*. In Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on pages 414-425. IEEE.

[50]R. Alur and D.L. Dill. *'Automata for modelling real-time systems'*. In Proc. of Int. Colloquium on Algorithms, Languages, and Programming, volume 443 of LNCS, pages 322–335, 1990.

[51] L. Soares, M. Stumm, S.C. Flex: *'Flexible system call scheduling with*

exception-less system calls 'Proceedings of the 9th USENIX conference on Operating systems design and implementation. USENIX Association, 2010: 1-8.

[52] P. Burke, P. Prosser. '*A distributed asynchronous system for predictive and reactive scheduling* '. Artificial Intelligence in Engineering, 1991, 6(3): 106-124.

[53] O. Bonaventure, C. Filsfils, P. Francois. '*Achieving sub-50 milliseconds recovery upon BGP peering link failures* '. Networking, IEEE/ACM Transactions on, 2007, 15(5): 1123-1135.

[54] A. Kvalbein, A. F. Hansen, T. Cicic, et al. '*Fast IP network recovery using multiple routing configurations*'. INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings. IEEE, 2006: 1-11.

[55] C. Ou, H. Zang, N. K. Singhal, et al. '*Subpath protection for scalability and fast recovery*' in optical WDM mesh networks. Selected Areas in Communications, IEEE Journal on, 2004, 22(9): 1859-1875.

[56] G. Banavar, T. Chandra, B. Mukherjee, et al. '*An efficient multicast protocol for content-based publish-subscribe systems*'. Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on. IEEE, 1999: 262-272.

[57] R. W. Floyd.' *Algorithm 97: shortest path* '. Communications of the ACM, 1962, 5(6): 345.

[58] S. E. Dreyfus. '*An appraisal of some shortest-path algorithms*'. Operations research, 1969, 17(3): 395-412.

[59] A. Rowstron, P. Druschel. '*Scalable, decentralized object location, and routing for large-scale peer-to-peer systems*'. Middleware 2001. Springer Berlin Heidelberg, 2001: 329-350.

[60] W. Yi, P. Petterson, and M. Daniels. '*Automatic verification of real-time communicating systems by constraint-solving*'. In Seventh International Conference on Formal Description Techniques, pp 223–238, 1994.

[61] S. Nagano,Y. Kakuda, T. Kikuno, '*Experience of responsiveness verification for connection establishment protocols*', Object-Oriented Real-time Distributed Computing, Apr 1998, pages 383-392.

[62] Y.B. Kartal, E.G. Schmidt, K.W. Schmidt, '*The verification of a novel framework for real-time shared medium communication network protocols.*' Signal Processing and Communications Applications Conference (SIU), April 2012, Ankara,

Turkey, pp. 1-4.

[63] INFSO D.4 Networked Enterprise & RFID INFSO G.2 Micro & Nanosystems, in: Co-operation with the Working Group RFID of the ETP EPOSS, Internet of Things in 2020, Roadmap for the Future, Version 1.1, 27 May 2008.

[64]B. Chew and J. Bigham, '*Bottleneck detection and forecasting in Message-Oriented-Middleware,*' in Proceedings of the European Safety and Reliabitily Conference (ESREL 2011), Troyes, France, 2011, pages 2631–2638.

[65]K. Sachs, S. Kounev, J. BACON, A. Buchmann, 2009. '*Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark*'. Perform. Eval. 66, 8, 410–434.

[66] A. K. Y. Cheung, H. A. Jacobsen. '*Load balancing content-based publish/subscribe systems*'. ACM Transactions on Computer Systems (TOCS), 2010, 28(4): 9.

[67]F. E. Redmond. '*Dcom: Microsoft Distributed Component Object Model with Cdrom*'. IDG Books Worldwide, Inc., 1997.

[68] IBM WebSphere MQ Homepage http://www-306.ibm.com/software/integration/wmq/

[69] IBM MQ Fundamentals http://www.redbooks.ibm.com/abstracts/sg247128.html

[70] IBM WebSphere MQ Information Centre http://publib.boulder.ibm.com/infocenter/wmqv6/v6r0/index.jsp

[71] J. Moy, OSPF version 2. RFC 2328 (1998)

[72] Network, M. Y. O. (2003). OSPF network design solutions.

[73] DDS: Data distribution service for real-time systems. http://www.omg.org/technology/documents/formal/data_distribution.htm

[74]E. Al-Shaer and H. Hamed. '*Discovery of policy anomalies in distributed firewalls*'. In Proceedings of IEEE INFOCOM'04, March 2004.

[75]H. Hamed, E. Al-Shaer, W. Marrero. '*Modelling and verification of IPSec and VPN security policies*'. Network Protocols, 2005. ICNP 2005. 13th IEEE International Conference on. IEEE, 2005: 10 pp..

[76] L. Yuan, J. Mai, Z. Su, H Chen, C. Chuah, and P. Mohapatra. '*FIREMAN: A toolkit for firewall modeling and analysis*'. In IEEE Symposium on Security and

Privacy (SSP'06), May 2006.

[77] R. Bush and T. Griffin. '*Integrity for virtual private routed networks*'. In IEEE INFOCOM 2003, volume 2, pages 1467– 1476, 2003.

[78] G. G. Xie, J. Zhan, D.A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. '*On static reachability analysis of ip networks*'. In IEEE INFOCOM 2005, volume 3, pages 2170– 2183, 2005.

[79] S. Narain. '*Network configuration management via model finding*'. In LISA, pages 155– 168, 2005.

[80] G. Abowd, R. Allen, and D. Garlan. '*Using style to understand descriptions of software architecture*'. In Proceedings of SIGSOFT'93: Foundations of Software Engineering, Software Engineering Notes 18(5). ACM Press, December 1993.

[81] D. J. Barrett, L. A. Clarke, P. L. Tarr, and A. E. Wise. '*A framework for event-based software integration*'. ACM Transactions on Software Engineering and Methodology, 5(4):378–421, October 1996.

[82] J. Dingel, D. Garlan, S. Jha, and D. Notkin. '*Reasoning about Implicit Invocation.*'. In Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6), Lake Buena Vista, Florida, November 1998. ACM.

[83] J. Dingel, D. Garlan, S. Jha, and D. Notkin. '*Towards a formal treatment of implicit invocation*'. Formal Aspects of Computing, 10:193–213, 1998.

[84] D. Garlan and D. Notkin. '*Formalizing design spaces: Implicit invocation mechanisms*'. In VDM'91: Formal Software Development Methods, pages 31–44, Noordwijkerhout, The Netherlands, October 1991. Springer-Verlag, LNCS 551.

[85] K. Havelund and T. Pressburger. '*Model checking java programs using java pathfinder*'. Intern' Journ' on Software Tools for Technology Transfer, 2(4), April 2000.

[86] Microsoft. Slam. http://research.microsoft.com/slam.

[87] J. Corbett, M. Dwyer, and J. Hatcliff. Bandera. '*Extracting finite-state models from java source code*'. Proceedings of the 22nd International Conference on Software Engineering, June 2000.

[88] R. Allen and D. Garlan. '*Formalizing architectural connection*'. In Proceedings of the 16th Intern' Conference on Software Engineering, Sorrento, Italy, May 1994.

[89] J. Magee and J. Kramer. '*Concurrency: state models & JAVA programs*'. John Wiley & Son, April 1999.

[90]D. Garlan, S. Khersonsky, J. S. Kim. '*Model checking publish-subscribe systems*'. Model Checking Software. Springer Berlin Heidelberg, 2003: 166-180.

[91] J.-S. Bradbury and J. Dingel. '*Evaluating and improving the automatic analysis of implicit invocation systems*'. In Proc. of the 9th European software engineering Conf., pages 78–87, 2003.

[92] M.-H. Beek, M. Massink, D. Latella, S. Gnesi, A. Forghieri, and M. Sebastianis. '*A case study on the automated verification of groupware protocols*'. In Proc. of the 27th Int. Conf. on Software engineering (ICSE05), pages 596–603, 2005.

[93]M. Caporuscio, P. Inverardi, and P. Pelliccione. '*Compositional verification of middleware-based software architecture descriptions*'. In Proc. of the 19th Int. Conf. on Software engineering (ICSE04), pages 221–230, 2004.

[94]X. Deng, M.-B. Dwyer, J. Hatcliff, and G. Jung. '*Model-checking middleware-based event-driven real-time embedded software*'. In Proc. of the 1st Int. Symposium on Formal Methods for Components and Objects, pages 154–181, 2002.

[95]J. Hatcliff, X. Deng, M.-B. Dwyer, G. Jung, and V. Ranganath. Cadena. '*An integrated development, analysis, and verification environment for component-based systems*'. In Proc. of the 25th Int. Conf. on Software Engineering (ICSE03), pages 160–173, 2003.

[96]A. Carzaniga, D.-S. Rosenblum, and A.-L. Wolf. '*Design and evaluation of a wide-area event notification service*'. ACM Trans. Comput. Syst., 19(3), 2001.

[97]E. Souto, G. Guimarães, G. Vasconcelos, et al. '*A message-oriented middleware for sensor networks*'. Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing. ACM, 2004: 127-134.

[98]P. Tran, P. Greenfield, I. Gorton. '*Behaviour and performance of message-oriented middleware systems*'. Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on. IEEE, 2002: 645-650.

[99] H. Zhang, J. S. Bradbury, J. R. Cordy, and J. Dingel. '*Implementation and verification of implicit-invocation systems using source transformation*'. In Proc. of the 5th Int. Wkshp. on Source Code Analysis and Manipulation (SCAM05), pages 87–96, 2005.

[100]J. Monod. '*The growth of bacterial cultures*'. Annual Reviews in Microbiology, 1949, 3(1): 371-394.

[101]J. Wang, P. Jiang, J. Bigham, B. Chew, M. Novkovic, and I. Dattani. '*Adding resilience to message oriented middleware*'. In Proceedings of the 2nd International Workshop on Software Engineering for Resilient Systems (SERENE '10). ACM, New York, NY, USA, pages 89-94, 2010

[102]X.F. An, L.Y. Bian. '*Design of Message-Oriented Middleware of Distance Teaching Platform Based on Distributed Message Control*'. Computational Aspects of Social Networks (CASoN), International Conference on. IEEE, pp141-143, 2010.

[103]J. Wang, J. Bigham, and J. Wu, '*Enhance Resilience and QoS Awareness in Message Oriented Middleware for Mission Critical Applications*', Information Technology: New Generations (ITNG), 2011 Eighth International Conference on. IEEE, 2011: 677-682.

[104]Y. Jia, E. Bodanese, J. Bigham, '*Checking the Robustness of a Publish/Subscribe Based Message Oriented System*', IV International Congress on Ultra Modern Telecommunications and Control Systems, St. Petersburg, pp 291-296, October 2012.

[105]M. Goyal, K.K. Ramakrishnan, W. Feng. '*Achieving faster failure detection in OSPF networks*', Communications, 2003. ICC'03. IEEE International Conference on. IEEE, pages 296-300, 2003.

[106]Y. Liu, A.N. Reddy. '*A fast rerouting scheme for OSPF/IS-IS networks*', Computer Communications and Networks, 2004. ICCCN 2004. Proceedings. 13th International Conference on. IEEE, pages 47-52, 2004.

[107]A. Iselt, A. Kirstadter, A. Pardigon, et al. '*Resilient routing using MPLS and ECMP*'. High Performance Switching and Routing, 2004. HPSR. 2004 Workshop on. Phoenix, Arizona, USA, IEEE, pages 345-349, 2004.

[108]G. Schollmeier, J. Charzinski, A. Kirstadter, et al. '*Improving the resilience in IP networks*', High Performance Switching and Routing, HPSR. Workshop on. IEEE, Torino, Italy, pp91-96, 2003.

[109]A.F. Hansen, A. Kvalbein, T. Čičić, et al. '*Resilient routing layers for network disaster planning*', Networking-ICN 2005. Springer Berlin Heidelberg, pp1097-1105, 2005.

[110]A. Kvalbein, A.F. Hansen, T. Cicic, S. Gjessing, O. Lysn: '*Fast recovery from*

link failures using resilient routing layers'. 10th IEEE Symposium on Computers and Communications (ISCC 2005), La Manga, Spain, pp 554-560 , 2005.

[111]A.F. Hansen, A. Kvalbein, S. Gjessing, et al. '*Fast, effective and stable IP recovery using resilient routing layers*' The 19th international teletraffic congress (ITC19). 2005.

[112]R. Bartos, M. Raman. '*A heuristic approach to service restoration in MPLS networks*'. In Proc. ICC, pages 117–121, June 2001.

 [113]K. Menger. Zur allgemeinen kurventheorie. Fund. Math., 10:95–115, 1927.

[114]F. Otel. '*On fast computing bypass tunnel routes in MPLSbased local restoration*'. In Proceedings of 5th IEEE International Conference on High Speed Networks and Multimedia Communications, Jeju, Korea, pages 234–238, 2002.

[115]E. Dijkstra, '*A note on two problems in connection with graphs*,'Numerische mathematik, pages 269-271, 1959.

[116]H. Jiang, C. Dovrolis. '*Passive estimation of TCP round-trip times*'. ACM SIGCOMM Computer Communication Review, pages 75-88,July 2002.

[117]K. Pagiamtzis, A. Sheikholeslami. '*Content-addressable memory (CAM) circuits and architectures: A tutorial and survey[J].*' Solid-State Circuits, IEEE Journal of, 2006, 41(3): 712-727.

[118]S. Stas, '*Associative processing with CAMs*'. Northcon/93 Conf. Record, pages 161-167 1993

[119]G. Qin, S. Ata, I. Oka, and C. Fujiwara, '*Effective bit selection methods for improving performance of packet classifications on IP routers*'. Proc. IEEE GLOBECOM, vol. 2, pages 2350-2354, 2002

[120]N.F. Huang, W.E. Chen, J.Y. Luo, and J.M. Chen, '*Design of multi-field IPv6 packet classifiers using ternary CAMs*'. Proc. IEEE GLOBECOM, vol. 3, pages 1877-1881, 2001

[121]Dijkstra, Edsger; Thomas J. Misa, Editor (August 2010). '*An Interview with Edsger W. Dijkstra*'. Communications of the ACM 53 (8): 41–47.

[122]Donald B. Johnson. 1973. '*A Note on Dijkstra's Shortest Path Algorithm*'. *J. ACM* 20, 3 (July 1973), 385-388. DOI=10.1145/321765.321768 http://doi.acm.org/10.1145/321765.321768

[123] M.H. Xua, , , Y.Q. Liua, Q.L. Huanga, Y.X. Zhanga, G.F. Luanb. '*An improved Dijkstra's shortest path algorithm for sparse network*'. Applied Mathematics and Computation, 2007, 185(1): 247-254.

[124]J. Postel. '*DoD standard transmission control protocol[J]*'. 1980.

[125]B. J. Nelson. '*Remote procedure call*'. Carnegie-Mellon Univ. Dept. Comput. Sci., 1981.

[126]E. Roman, R. P. Sriganesh, G. Brose. '*Mastering enterprise javabeans*'. John Wiley & Sons, 2005.

[127]V. Venkatesh, M. G. Morris, G. B .Davis, et al. '*User acceptance of information technology: Toward a unified view*'. MIS quarterly, 2003: 425-478.

[128]J. S. Hunter. '*The exponentially weighted moving average*'. J. QUALITY TECHNOL., 1986, 18(4): 203-210.

[129]P. Jacquet, P. Muhlethaler, T. Clausen, et al. '*Optimized link state routing protocol for ad hoc networks*'//Multi Topic Conference, 2001. IEEE INMIC 2001. Technology for the 21st Century. Proceedings. IEEE International. IEEE, 2001: 62-68.

[130]D. W. Inderieden, A. J. Lehtola, P. A. Laverdiere, et al. '*Notification to routing protocols of changes to routing information base*': U.S. Patent 20,040,006,640. 2004-1-8.

[131]A. Mehta, S. Shenoy. '*Synchronizing multiple instances of a forwarding information base (FIB) using sequence numbers*': U.S. Patent 7,499,447. 2009-3-3.

[132]I. Glover, P. M. Grant. '*Digital communications[M]*'. Pearson Education, 2010.

[133]B. Y. Choi, S. Bhattacharyya. '*On the accuracy and overhead of cisco sampled netflow*'. Proceedings of ACM SIGMETRICS Workshop on Large Scale Network Inference (LSNI). 2005.

[134]A. F. Hansen, A. Kvalbein, T. Cicic, et al. '*Resilient routing layers for recovery in packet networks*'. Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on. IEEE, 2005: 238-247.

[135]A. F. Hansen, A. Kvalbein, T. Čičić, et al. '*Resilient routing layers for network disaster planning*'. Networking-ICN 2005. Springer Berlin Heidelberg, 2005: 1097-1105.

[136]D. Pompili, T. Melodia, I. F. Akyildiz. '*A resilient routing algorithm for long-term applications in underwater sensor networks*'. Proc. of Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net). 2006.

[137]T. Cicic, A. Kvalbein, A. F. Hansen, et al. '*Resilient routing layers and p-cycles: Tradeoffs in network fault tolerance*'. High Performance Switching and Routing, 2005. HPSR. 2005 Workshop on. IEEE, 2005: 278-282.

[138]F. Curbera, R. Khalaf, N. Mukhi, et al. '*The next step in web services*'. Communications of the ACM, 2003, 46(10): 29-34.

[139]C. Blanchard, S. Burlingame, S. Chandramohan, et al. '*IBM Websphere RFID handbook: A solution guide*'. IBM, International Technical Support Organization, 2006.

[140]K. Trivedi, D. Wang, D. J. Hunt, et al. '*Availability modeling of SIP protocol on IBM© WebSphere*'. Dependable Computing, 2008. PRDC'08. 14th IEEE Pacific Rim International Symposium on. IEEE, 2008: 323-330.

[141]K. L. McMillan. '*Symbolic model checking*'. Springer US, 1993.

[142]E. M. Clarke, O. Grumberg, D. Peled. '*Model checking*'. MIT press, 1999.

[143]A. Cimatti, E. Clarke, F. Giunchiglia, et al. '*NuSMV: a new symbolic model checker*'. International Journal on Software Tools for Technology Transfer, 2000, 2(4): 410-425.

[144]A. Cimatti, E. Clarke, E. Giunchiglia, et al. '*Nusmv 2: An opensource tool for symbolic model checking*'. Computer Aided Verification. Springer Berlin Heidelberg, 2002: 359-364.

[145] Logic T. '*Temporal logic*'. Stanford Encyclopedia of Philosophy, 2002.

[146]E. M. Clarke, E. A. Emerson, A. P. Sistla. '*Automatic verification of finite-state concurrent systems using temporal logic specifications*'. ACM Transactions on Programming Languages and Systems (TOPLAS), 1986, 8(2): 244-263.

[147]B. Bérard, M. Bidoit, A. Finkel, et al. '*Systems and software verification: model-checking techniques and tools*'. Springer Publishing Company, Incorporated, 2010.

[148]C. Flanagan, P. Godefroid. '*Dynamic partial-order reduction for model checking software*'. ACM Sigplan Notices. ACM, 2005, 40(1): 110-121.

[149]G. J. Holzmann. '*The SPIN model checker: Primer and reference manual*'. Reading: Addison-Wesley, 2004.

[150]G. J. Holzmann. '*The model checker SPIN*'. IEEE Transactions on software engineering, 1997, 23(5): 279-295.

Appendix A

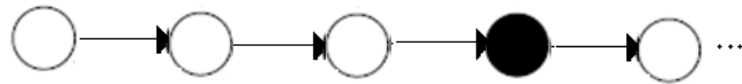F p: this formula will be true if p eventually become true:

Figure A - 1LTL Formula 'F p'

G p: this formula will be true if p is true at every state:
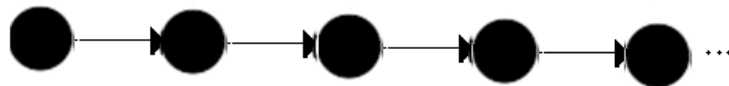

Figure A - 2LTL Formula 'G p'
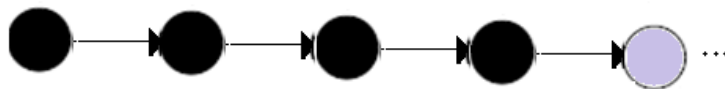
p U q: this formula will be true if p is always true until q is true

Figure A - 3LTL Formula 'p U q'

AX p: for all paths, p will be true in next state

p AU q: for all paths, p is true until q is true
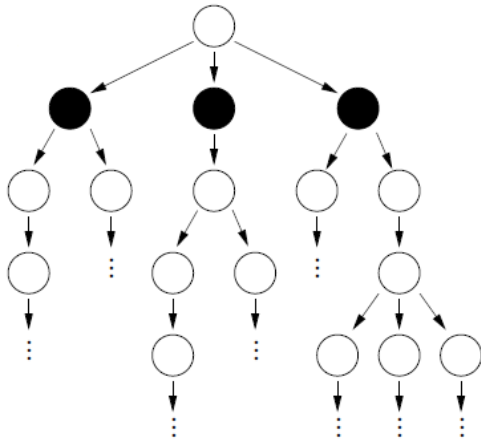
Figure A - 4CTL Formula 'AX p' [32]   Figure A - 5CTL Formula 'p A U q' [32]
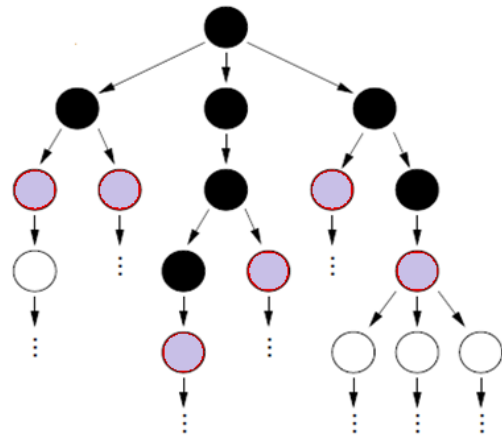
EG p: there is at least one path in which p will be true all the time

EF p: there is at least one path in which p will finally be true
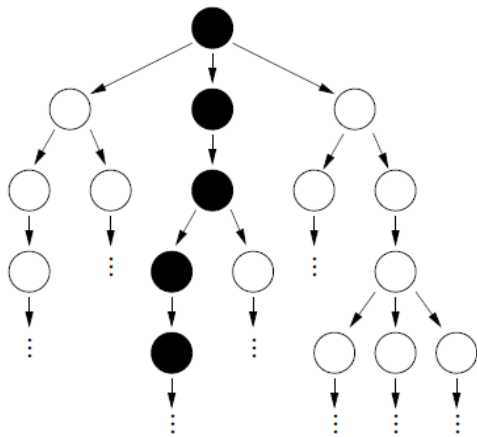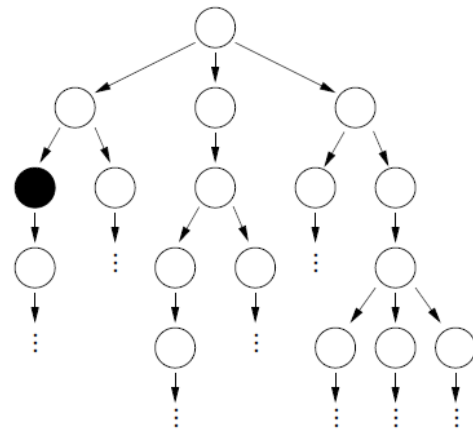

Figure A - 6CTL Formula 'EG p' [32]   Figure A - 7CTL Formula 'EF p' [32]

EX p: there is at least one path in which p is true in next state

p EU q: there is at least one path in which p is true until q is true
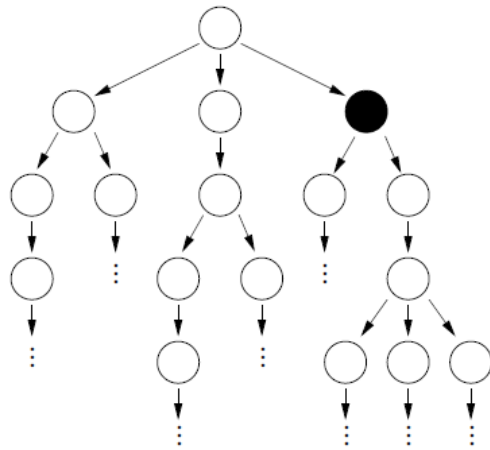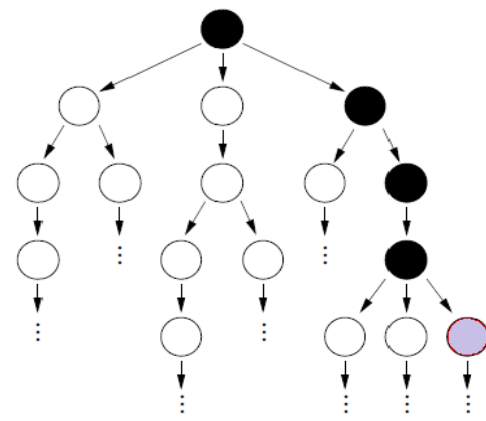
Figure A - 8 CTL Formula 'EX p' [32]

Figure A - 9CTL Formula 'p EU q' [32]