Message Processing on XML-based Publish/Subscribe Systems

by

Yang Cao

A Thesis Submitted to
the Faculty of Graduate and Post Doctoral Affairs
in Partial Fulfilment of
the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

School of Computer Science

at

CARLETON UNIVERSITY

Ottawa, Ontario

August 2012

Your file Votre référence
ISBN: 978-0-494-93661-0

Our file Notre référence
ISBN: 978-0-494-93661-0

Abstract

With new innovations in information and communication, XML-based publish/subscribe systems have received more and more attention from academic communities and industries. In order to design a high-performance pub/sub system, this thesis considers the design of XML-based pub/sub systems from an integrated standpoint which includes using: caching technique, query aggregation algorithms, query routing algorithms, and networking. Existing research mainly focuses on the efficiency of filtering algorithms. Little research, however, has considered using computing in combination with network technologies in the context of XML filtering.

First of all, this thesis proposes two caching schemes to be used in conjunction with an XML filtering engine. In addition, this thesis experiments with synthetic data and real data from web applications. Secondly, this thesis presents a novel algorithm for computing XPath query aggregation using a region encoding scheme. Extensive experiments have been performed to study the effectiveness of the proposed approaches and examine the trade-off between costs and gains. Thirdly, an innovative XML routing model, a peer model, is presented. This peer model can significantly improve the system throughput and scalability, and is robust for network changes.

Acknowledgements

I am most grateful to my co-supervisors, Professor Shikharesh Majumdar and Professor Chung-Horng Lung from the Department of Systems and Computer Engineering. Much of their devoted time and help have enabled me to complete this work. My supervisors are not only excellent professors; they have also been for me friends, mentors, and teachers.

I am extremely grateful to my parents. I owe them my love and gratitude. I wish to acknowledge my landlords, Mr. and Mrs. Kirschman. They care about me a lot and teach me some valuable practical life experiences. I also appreciate Ms. Jenny Meldrum who has been a nice elder sister. I am also grateful to the technical staffs, for their support and help. Thanks also to the department's professors and secretaries. I equally wish to express my appreciation to all the people who gave me guidance and help while living in Canada.

I also would like to remember the late Professor Sivarama Dandamudi. I wish to thank him for accepting me and giving me the opportunity to join this group.

The six years of research for my dissertation have offered me nothing less than the pure enjoyment of learning something new. This education will no doubt be helpful to me in any further investigation, providing solutions to problems, including those previously mentioned in this thesis. The lessons I have learnt here will definitely be a great asset for investigating problems in the future. I would like to thank Alcatel-Lucent, the Ontario Centres of Excellence and NSERC for providing financial support for this research.

List of Abbreviations

CE	Customer Edge
DFA	Deterministic Finite Automaton
DHT	Distributed Hash Table
DOM	Document Object Model
DTD	Document Type Definition
FSM	Finite State Machine
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
Java NIO	Java New I/O
LRU	Least Recently Usage
MEDYM	Match Early with Dynamic Multicast
NFA	Non-deterministic Finite Automaton
ONYX	Operator Network using Yfilter for XML
PE	Publish Edge

REST	Representational State Transfer
RSS	Rich Site Summary
SAX	Simple API for XML
SOAP	Simple Object Access Protocol
UDDI	Universal Description Discovery and Integration
URL	Uniform Resource Locator
VPN	Virtual Private Network
WSDL	Web Services Description Language
XPath	XML Path Language
XML	Extensible Markup Language
XNET	XML Content Network
XSLT	Extensible Stylesheet Language Transformations
YF	Yfilter with No Caching
YF-C-Cache	Yfilter with Complete Caching
YF-S-Cache	Yfilter with Structure-based Caching

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As more and more information becomes available on the Internet, both information users and producers face more challenges, including the volume of information and various formats of information. Users can describe their interests and producers can publish their data. Users demand the latest and most intelligent information. Publishers want the published information to reach the targeted users. In traditional Internet usage, users must know the location of the data (URL). Another way for users to obtain information is the query/response model. In the query/response model, users send individual queries, such as keywords, to individual producers for processing, such as google. A list of responses is returned to the individual users. The third way is the publish/subscribe (pub/sub) model. In the pub/sub model, users specify their interests to producers. The producers push published information to users who have specified interests in specific published information.

Pub/sub model is asynchronous. Such a system involves publishers producing messages and subscribers selectively receiving the messages in which they are interested. The advantage of asynchronous messaging is the loose coupling between the sender an3d the receiver. They do not have to wait for each other, allowing more flexibility. The pub/sub paradigm can be used in different applications to facilitate creation and management of information, and also to connect people in new ways.

Web content syndication, digital weather forecasting and intelligent emergency alert systems are examples of such Web applications that are becoming increasingly popular.

1.1 XML-based publish/subscribe system

Extensible Markup Language (XML) [9] eliminates differences of proprietary protocols between networkings, operating systems and platforms, and enables systems using different hardwares and softwares to communicate with each other.

With the evolution of XML and its impact on the Web, Web services are being adopted as subsystems for distributed computing. A web service application can be published, discovered, and invoked over a network using standard network protocols. Typically, XML is used to tag the data, Simple Object Access Protocol (SOAP) is used to transfer the data, Web Services Description Language (WSDL) is used for describing the services available, and Universal Description Discovery and Integration (UDDI) is used for listing what services are available. Web 2.0 allows Web-based access to Web services using Representational State Transfer (REST) and SOAP. REST uses Hypertext Transfer Protocol (HTTP) messages to access XML data where SOAP uses more complex messages [73]. XML pub/sub systems have recently emerged as XML routers, such as Sarvega XML Context Router [26]. An XML router is a complement to an IP router that carries the stream of data traffic across the Internet.

An XML pub/sub system matches XML messages against a large number of user subscriptions and delivers messages to matched subscribers across a network using a routing algorithm at the application layer. The architecture of an XML-based pub/sub system which includes publishers, subscribers and brokers, is depicted in Figure 1.1. A broker performs the function similar to a router in a pub/sub system. It is responsible for selecting and delivering messages based on a set of stored subscriptions or queries. The pub/sub system supports advanced services, such as guaranteed delivery, security and enterprise application integration [83]. In Figure 1.1, Brokers

PE_1 and PE_2, connecting publishers, are called publisher edge (PE) brokers or nodes. Subscribers $Subscriber_1$ and $Subscriber_2$ submit subscriptions or queries S_1 and S_2. They are connected to CE_1 and CE_2, which are called subscriber edge or customer edge (CE) brokers. The other brokers are not directly connected to any publisher or a consumer, and are called intermediate or tandem brokers. PEs, CEs, and intermediate brokers are inter-connected together, enabling them to exchange messages. This allows subscribers connected to one of the brokers to receive messages that have been published through another broker, further freeing the subscriber from the constraint of using the same broker as that used by the publisher [83]. Each PE broker maintains a spanning tree. Each broker is assumed to know its neighbors and the best path leading to a PE node [42]. For example, $Broker_0$ to $Broker_3$ in Figure 1.1 are connected by two spanning trees, rooted at PE_1 and PE_2, respectively, ensuring short end-to-end delay for message delivering and avoiding loops on delivery paths.

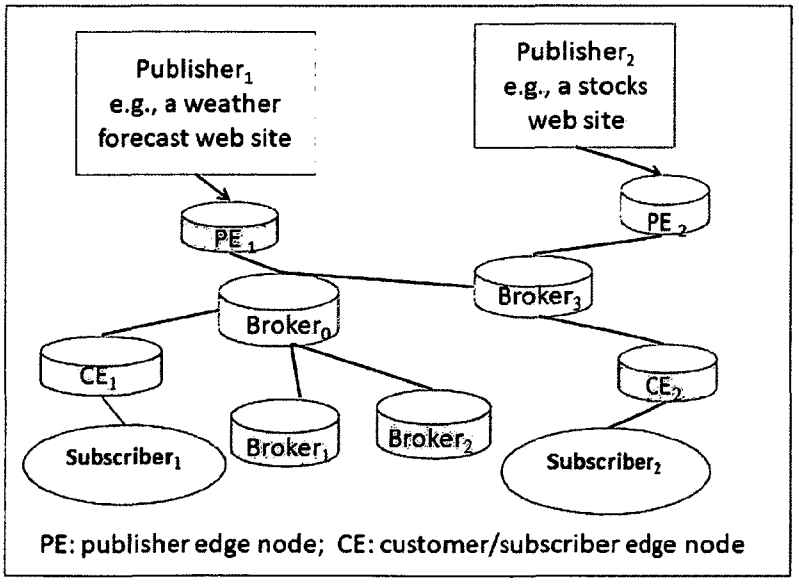


Figure 1.1: Publish/subscribe systems (adapted from [42])

In the context of this thesis, messages are encoded in XML and user subscriptions or queries are specified in XPath syntax. XML is a markup language for creating documents containing structured information. Our project supports a subset of XPath

1.0 syntax including the parent/child operator ('/'), ancestor/descendant operator ('//'), wildcard ('*') and predicate ('[ ]'), denoted as $XP^{/,//,*,[\ ]}$ [3]. An XPath query can access nodes of an XML document. The query consists of a sequence of location steps. Each location step contains an axis, a node test and zero or more predicates. An axis specifies the hierarchical relationship between nodes, for example, parent/child operator or ancestor/descentant operator.

The XML-based pub/sub system has challenges: filtering XML messages, aggregating XPath queries, and transmitting XML messages. The structural attribute of XML and the flexibility of XPath give rise to complexity in XML filtering. Besides content matching, structure matching is required to select message nodes matched by a filtering engine. Secondly, XML routing is problematic at foundational levels with XML-based pub/sub systems. In a distributed environment, existing approaches for XML routing match published messages with aggregated subscriptions or queries at each node along the overlay path to relevant subscribers. Such approaches are time-consuming in a distributed environment. New architecture for XML routing is required for addressing such problem. How to design a new architecture for XML routing? All these interesting questions motivate us to examine the existing approaches for XML caching, XPath aggregation, and XML routing.

As more and more data are represented in XML format, XML-based network traffic experiences significant growth. The volume of increased messaging can lead to network saturation. In order to design a high-performance pub/sub system, we consider the design of XML-based pub/sub systems from an integrated point of view, including using caching techniques and query aggregation algorithms at the node level, and query routing algorithms at the network level. Each of the three areas are elaborated further in the following subsections.

1.2 Motivations for caching in XML pub/sub systems

In practice, caching provides fast access for most applications by storing previous computed results locally. With the increasing demand for reducing the processing time for XML documents filtering, we have analyzed the potential for XML document caching in XML pub/sub services.

Repeated invocations of the XML filtering engine with the same published data may often produce the same response from the filtering engine and can be considered a useless work. Such messages are often caused by periodic publication of data (such as weather condition). Caching of the results of message filtering can be an effective approach that can exploit such a behavior and avoid unnecessary filtering operations when there is a cache hit. To the best of our knowledge, most test data sets in current related research are synthetic. This thesis collects and analyzes real-life data including long-term type data such as weather data and short-term type data such as stock data. The performance of caching for the real-life data set is analyzed.

1.3 Motivations for XPath query aggregation

XML pub/sub systems have given rise to application-layer XML routers, such as Sarvega XML Context Router [26]. The problem with the state-of-the-art XML routing schemes is the cost of filtering which is expensive at each hop. Typically, a large number of subscriptions have to be matched with constantly arriving XML documents. Often, the filtering speed cannot match the XML document arrival speed on a typical pub/sub system. The XML filtering cost is observed to increase as the number of queries to be filtered is increased.

One way to improve system performance is to reduce the number of XPath queries stored at an XML broker. Query aggregation is a technique to combine individual queries that are used to fetch subscribed data by multiple subscribers into a single

query. Investigating effective query aggregation techniques is important in the context of pub/sub systems.

1.4 Motivations for investigating different models for XML routing

The exploration of data delivery protocols is the third important research concern for this thesis. XML-based network traffic is expected to grow significantly in the coming years. The main problem of the state-of-the-art XML routing schemes is that the cost of filtering and aggregation for each hop is expensive as the tasks have to be conducted at each node. The conventional XML multicast model is to deliver a message only to those subscribers who declare their interest. In the worst case scenario, all message brokers on the network need to receive and filter messages.

To reduce the time for query aggregation and message filtering, this thesis investigates new XML routing models for performance improvement. These XML routing models are the cross-layer XML multicast model and the peer model. Those models can either reduce the number of queries that are to be merged into the query tree or reduce the number of filtering operations that need to be performed at each broker.

1.5 System overview and contributions

This section presents an overview of XML-based pub/sub system and presents a summary of thesis contributions. The thesis targets fixed networks in which publishers and subscribers are not in a mobile environment.

1.5.1 System overview

This thesis considers the architecture of an XML pub/sub broker shown in Figure 1.2, which summarizes the main subsystems for an XML-based pub/sub system. A filtering engine generates matched queries by matching an XML message X with queries.

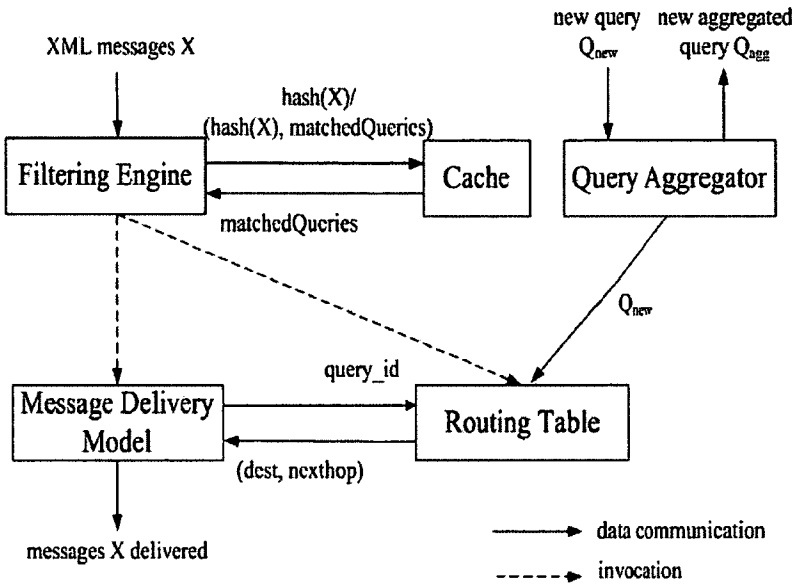


Figure 1.2: XML pub/sub broker architecture

The cache subsystem keeps track of cached instances of previously processed messages in memory. A query aggregator analyzes new queries expressed in XPath to detect covering relationships, generates subscription or query tokens and subscriber information for insertion in the query table and forwards new aggregated queries to upstream brokers. A query token corresponds to a location step of an XPath query. The routing table stores the network topology and reachability information for queries and subscribers. The message delivery model notifies identified subscribers or next broker by checking the routing table.

1.5.2 Contributions

The contributions of this thesis are summarized.

A. Chapter 3 presents two caching techniques to improve the performance of XML filtering by reusing previous filtering results. One is based on caching the whole message and the other is based on caching the message structural summary. The characteristics of real data for XML-based pub/sub systems are examined. The performance of the caching technique using real-life data is investigated.

Moreover, this thesis performs a methodic and rigorous performance investigation of an existing open-source XML filtering engine–Yfilter also described in the existing literature. Results of the experiments are analyzes and insights gained into system behavior and performance are presented in Chapter 3. A paper presenting the idea of the caching techniques and some preliminary results is published:

– i – Cao, Y., Majumdar, S., and Lung, C.-H. Caching techniques for XML message filtering. In Proc. of the IPCCC (2009), pp. 315–322, Phoenix, Arizona, USA [34].

B. Chapter 4 discusses new XPath query aggregation algorithms. Experimental results demonstrate the effectiveness of the presented algorithm. For the experiments described in this thesis, the proposed algorithm is observed to give rise to a performance improvement in comparison to the existing XSearch algorithm from the literature that is known as an efficient algorithm for XPath query aggregation. A paper presenting the idea of the new query aggregation algorithm and some preliminary results is published:

– i – Cao, Y., Lung, C.-H., and Majumdar, S, An XPath query aggregation algorithm using a region encoding, in Proc. of the SAINT (2011), pp. 27–36, Munich, Germany [32].

C. The contributions of this thesis on XML routing in the context of pub/sub systems, the cross-layer XML multicast model and the peer model, are presented in Chapter 5. Different communication models, including the conventional XML multicast model, the cross-layer XML multicast model, the unicast model and the peer model, for XML routing are evaluated. Experimental results demonstrate the effectiveness of the peer model. Two papers presenting the idea of the peer model and some preliminary results are published:

– i – Cao, Y., Lung, C.-H., and Majumdar, S, A subscription coverage technique

for XML message dissemination, in Proc. of the SAINT (2009), pp. 137–140, Seattle, USA [31].

– ii – Cao, Y., Lung, C.-H., and Majumdar, S, A peer-to-peer model for XML publish/subscribe services, in Proc. of the CNSR (2011), pp. 26–32, Ottawa, Canada [33].

1.6 Dissertation outline

In this thesis, queries and subscriptions are used interchangeably. Conventional model and application-layer multicast are interchangeable as well. Furthermore, XML document and XML messages are used interchangeably in this thesis.

The remainder of this thesis is structured as follows:

Chapter 2 provides the related work on pub/sub systems, different routing protocols for overlay networks, stream processing XML data on XPath subscriptions, XML routing, caching for XML filtering as well as aggregating XPath queries. Chapter 3 discusses caching techniques for XML messages. Chapter 4 presents XPath query aggregation algorithms. Chapter 5 presents the peer model and an evaluation of different XML routing schemes. Chapter 6 concludes this thesis and discusses future work.

Chapter 2

Background and related works

In this chapter, an overview of pub/sub systems is provided, a summary for XML, XPath, and XSLT are presented, and the current state-of-the-art in XML filtering, caching, routing and subscription aggregation are discussed. Firstly, existing work on pub/sub systems is described. Thereafter, three widely used real-life applications using pub/sub systems are chosen as examples for discussion. Secondly, a discussion of the application-layer multicast and a comparison of a number of existing pub/sub systems is presented. Thirdly, a brief introduction of XML, XPath, and XSLT is provided. Existing research work on XML filtering, caching, routing and subscription aggregation are summarized in the later part of this chapter.

2.1 Pub/sub systems: background

Pub/sub is a powerful abstraction for building distributed applications. A high-level overview is depicted in Figure 2.1. Publishers create information, e.g., weather forecast or other web content. Subscribers publish interests and consume information. Publish/Subscribe brokers filter the published data and notify the matched subscribers, using an asynchronous message-based communication mechanism. The main characteristic of the pub/sub systems is decoupling of the participants from each other. There is no need for the publishers and subscribers to be directly connected or

know each other with regard to time and space. With regard to message flow, there is no need to be synchronized. The publishers and subscribers need not be active at the same time [56].

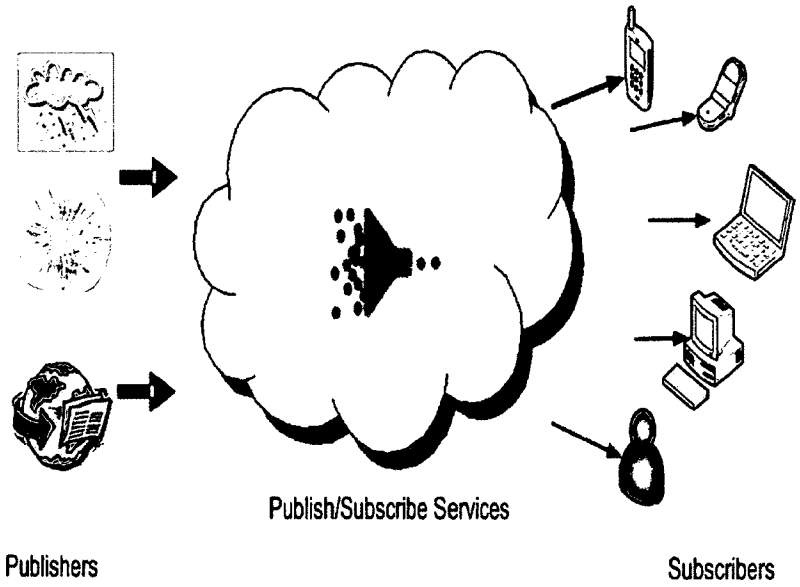


Figure 2.1: A high level overview of a pub/sub system

Pub/sub systems can be classified into two broad categories according to their expressiveness: (1) topic-based pub/sub systems; and (2) content-based pub/sub systems. In topic-based systems, the published data space is classified into a set of logical topics. Subscribers can subscribe to a topic and publishers can publish messages addressed to a topic. Each subscriber that subscribes to that topic can receive a message copy. The Information Bus/Rendezvous (TIB/Rendezvous) [11] is an example of topic-based pub/sub systems. A content-based system consists of publishers who can publish messages and subscribers who can specify queries identifying messages in which they have interest. The difference between topic-based and content-based approaches are: in a topic-based system, users subscribe for messages through a set of predefined subjects (topics), while in a content-based systems, subscriptions are related to specific message content.

2.1.1 Pub/sub applications

The pub/sub concept can be used in different applications to facilitate creation and management of information, and to connect people in new ways. Examples include web content syndication, digital weather forecasting and intelligent emergency alert systems which are increasingly popular web applications. This section provides brief descriptions of each of the applications.

2.1.1.1 Web content syndication

A web feed is a document including web links to web contents. Two web content syndication formats are Rich Site Summary (RSS) and Atom Syndication Format [12]. Technologies for web feeds have traditionally used a pull-based mechanism for delivering news items to receivers, in the form of feeds transported using the HTTP protocol. The process of sampling sites for new items [75] is referred to as aggregation.[1] Aggregation applications for feed readers poll the news provider at a specific URI to retrieve a feed containing updated news items, such as web-based applications. Alternatively, a push mechanism is used for news delivery, where a news provider publishes a news item that can be actively delivered to interested receivers. Therefore the pub/sub paradigm is a proper candidate for implementing such a push-based news delivery system.

As mentioned in [75], an existing application that uses the push method for transport is `pubsub.com`. It aggregates news from several sources and allows subscribers to submit queries on all aggregated items. As soon as a matching item is found, the corresponding subscribers receive the matched item in a publish-subscribe notification. The pubsub.com company provides a browser plugin that shows notifications as they are received. Another example introduced in [75] is Mimír (`http://mimir.ik.nu/`), a service that is a web-based news reader. When new news items are received, on-line subscribers can be alerted by an instant message via Jabber. Off-line users have the

[1] For better efficiency, the HTTP protocol provides features for compression and only sending full feeds when the news has actually changed.

item marked as unread. Unread items can then be read on a personalized web page, much like other web-based news readers.

2.1.1.2 Digital weather report

Communication and information innovations have provided opportunities to improve public weather services (PWS). These innovations allow the World Meteorological Organization (WMO) and National Meteorological/ Hydrometeorological Services (NMHSs) to provide powerful forecasts and warning services. In addition, these innovations can impact NMHS service delivery capabilities.

For example, Environment Canada (EC) uses a National Weather Element Forecast Database (NWEFD) for storing the output from the EC numerical weather prediction models. EC forecasters make adjustments to forecast fields based on the current state of the atmosphere and known model biases and trends. Thereafter, software is executed to output text-based forecasts. EC has also developed an expert system called SCRIBE for a region or a specific locality. SCRIBE's temporal resolution is 3 hours. SCRIBE produces forecasts twice a day for 1,145 Canadian locations. The output is processed, aggregated and resized by regional SCRIBE systems, and used by the regional offices to generate local weather forecasts [63].

New computer and telecommunication technologies, such as the Internet, wireless communication technologies, GIS, GPS and mobile communication networks, allow NHMSs to exploit the latest telecommunication networks to improve processing and delivery. Information technology systems and associated applications, including XML, Common Alerting Protocol (CAP), and RSS, allows NHMSs to provide weather forecasts and warnings in a variety of new formats to better serve the user community [63].

2.1.1.3 Intelligent emergency alert

CAP [25] is an open standard data interchange format, a template for effective warning messages. It can be used to collect warnings and reports locally, regionally or

nationally. CAP uses XML to represent the alerts and notifications of disasters including hazardous materials release, radioactive release, thunderstorms, fires, earthquakes, volcanic eruptions, flooding and tsunamis. Systems using CAP can quickly launch Internet messages, news feeds, television text captions/scrolls, highway sign messages, and synthesized voice-over automated telephone calls or radio broadcasts to effectively alert the public [63].

2.1.2 Pub/sub architectures and overlay networks

The term *architecture* refers to the network topology and protocols for message delivery in a pub/sub system.

2.1.2.1 The pub/sub architecture

The pub/sub system could have a centralized architecture shown in Figure 2.2a and a distributed architecture Figure 2.2b. In a centralized architecture, there is only one pub/sub broker. All queries are stored at the broker and all published messages are processed by the same broker. The centralized architecture has a scalability problem. In a distributed architecture, there are multiple pub/sub brokers. These brokers are built on top of an overlay network. Published data are routed on the overlay.

The *overlay network* is an effective way to build a network on top of another without changing the underlying network. The nodes in an overlay network are connected by logical links, each of which corresponds to one or many physical links in the underlying network. An overlay has no control over how packets are routed in the underlying network between two overlay nodes; but it can control, for example, the sequence of the overlay nodes through which a message traverses before reaching its destination. Based on the classification of the overlay network, the overlay may be unstructured or structured.

In an *unstructured overlay*, an event is routed to a node based on the subscription summary stored at that node. Based on the interconnection topology, unstructured overlays can be classified into two classes: a hierarchy tree and an acyclic P2P graph.

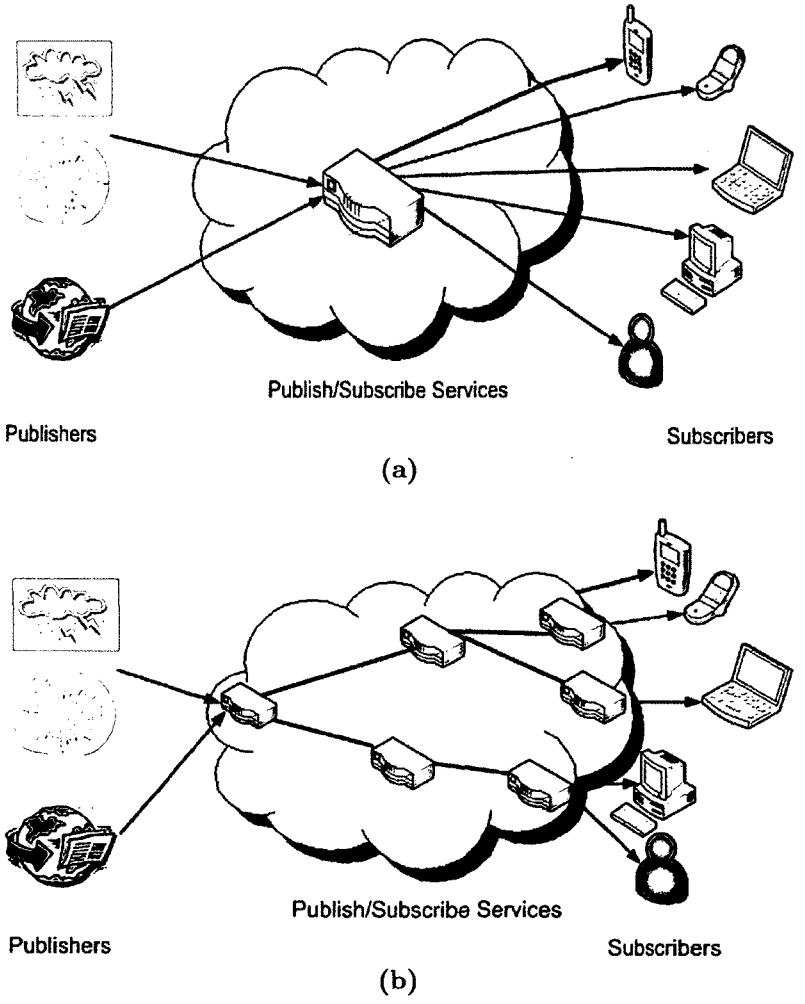


Figure 2.2: Pub/sub architecture:(a) centralized;(b)distributed.

In the first case, all the nodes are organized in a client/server relationship. In the latter case, all nodes communicate with each other as peers. The main drawback of an unstructured network is that the number of routing hops an event traverses increases with the number of nodes in the system and content matching has to be preformed at each hop. Therefore, the unstructured overlay is suitable only for systems with a maximum of hundreds of nodes [22]. Another shortcoming of the unstructured overlay is that it cannot provide good reliability.

In *structured overlays*, a distributed hash table (DHT) is used. The DHT scales to a large numbers of nodes and handles node arrivals and failures. The DHT splits the key space among the participating overlay nodes using a key space partitioning scheme; each node is assigned a range for which it is responsible. Four DHT systems (CAN [89], Chord [97], Pastry [91], and Tapestry [109]) were published in 2001. Rendezvous-based routing is used in the structured overlay. The downside is that DHTs only support exact-match lookups [22].

2.1.2.2 Forwarding and routing

The Internet is a network of networks that interact with each other through exchange of data packets. A router connects two or more logical subnets and performs the functions of routing and forwarding information. Routing and forwarding are two basic terms often used in computer networking.

The term *routing* refers to selecting paths in a network along which to send data. Routing is a concept related to the control plane: it directs forwarding, the passing of logically addressed packets from their source to their ultimate destination through routers. The routing process usually directs forwarding on the basis of routing tables stored within the routers, which maintain a record of the best routes to various network destinations. Routing concerns the propagation of the subscriptions and necessary topological information in order to maintain acyclic and possibly minimal forwarding paths for messages [36].

The term *forwarding* is used in the data plane. In the context of XML pub/sub

systems, it is associated with the filtering operation (i.e., how to determine which events match the subscriptions) and notification (i.e., where to deliver the matched message to the subsequent nodes or subscribers).

Take an IP router as an example. "The basic functionalities in an IP router can be categorized as: route processing, packet forwarding, and router special services. The two key functionalities are route processing (i.e., path computation, routing table maintenance, and reachability propagation) and packet forwarding" [21]. The architecture of an IP router is depicted in Figure 2.3.

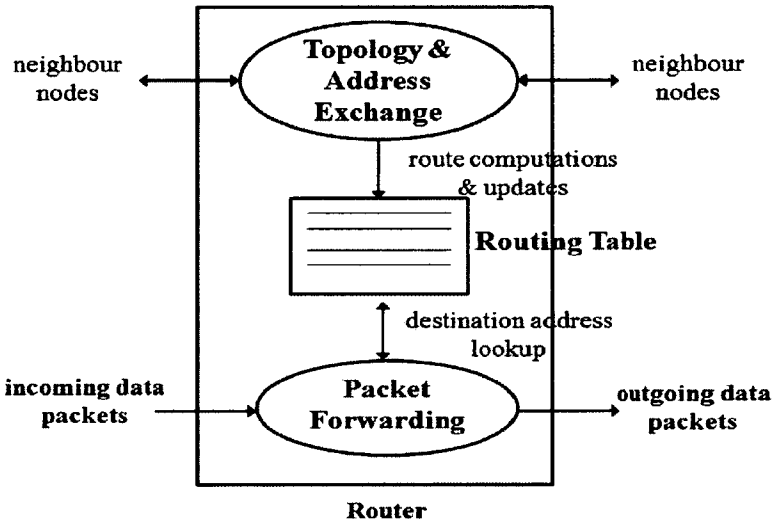


Figure 2.3: Generic architecture of an IP router from [21]

2.1.3 Typical delivery protocols in pub/sub systems

To deliver messages to the identified receivers, there are three delivery mechanisms in a distributed network: flooding, matching-first, and multicasting.

The *flooding* method shown in Figure 2.4a is to send a message to all receivers in a network and unwanted messages are discarded by the receivers. For example, in Figure 2.4a, message 1 and message 2 are sent to all receivers in a network and the receivers decide whether or not to accept the message. The flooding method is good for small networks [23] and has a network saturation problem.

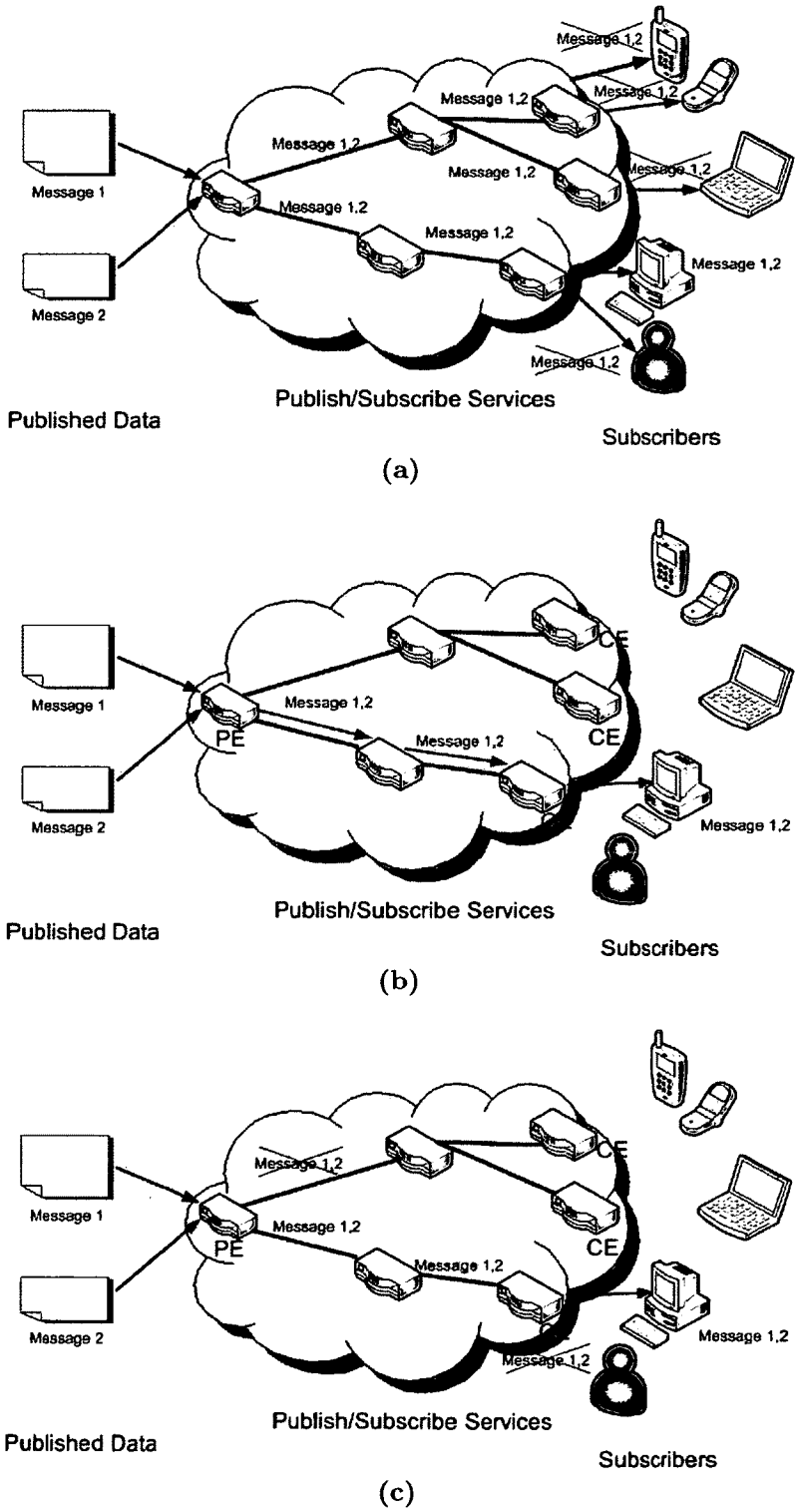


Figure 2.4: Typical delivery protocols in pub/sub systems:(a) flooding protocol;(b) matching first;(c) multicast.

The *matching-first* approach is shown in Figure 2.4b. In the matching-first protocol, all subscriptions or queries are sent to the PE and no query aggregation technique is applied. When the PE receives a message, it first matches a message against all the subscriptions, generates a destination list, and then routes the message to identified subscribers [23]. The message is delivered to each subscriber using unicast. So there are multiple copies of the same message traversing the network. For example, in Figure 2.4b, two unicast for message 1 and message 2 are sent along the same delivery path. Thus, the matching-first method only works well in small systems. Moreover, no query aggregation applied leads to a large query table at PE because all subscriptions are stored at the PE.

The third approach is *multicasting* shown in Figure 2.4c. Multicasting is a one-to-many method whereby a message can be sent simultaneously to several nodes. Filtering is exploited to identify the next hops and aggregation is used to reduce the number of subscriptions or queries. A subset of subscriptions or queries are stored at the PE. There are two types of multicasting schemes: IP-layer multicast and application-layer multicast. IP-layer multicast [81] works on IP address and is slowly being adopted in real networks today. On the other hand, the application-layer multicast is implemented at the application layer on top of an overlay network. For example, there is only one message sent from PE to CE in Figure 2.4c. At the CE, the XML filtering engine identifies only one receiver who is a subscriber of the message. Therefore, only one copy of message is sent to the identified receiver. A more detailed discussion of application-layer multicasting is presented next.

2.1.4 Application-layer multicast

There are two variations of the application-layer multicasting, which are filter-based forwarding for unstructured overlay and rendezvous-based routing for structured overlay.

2.1.4.1 Filter-based forwarding

In unstructured overlay systems, a filter-based forwarding method is often used to route messages from the publishers to the final subscribers. It is used in a number of systems that includes Gryphon [16], Siena [35], ONYX [55] and XNET [43]. In this method, a message is forwarded to its next hop when there are matched receivers. It includes the following phases: (1) setting up the query table; and (2) matching and forwarding of published messages.

In the first phase of setting up query tables, subscriptions or queries are propagated along a delivery tree rooted at each PE. This phase ends when the queries arrive at the PEs. At this time, each router keeps a query table: each entry in the query table records the queries from its child routers. The delivery tree connects all the intended publishers and subscribers [36]. In Siena, the tree is built by flooding the advertisement message to all the nodes in the network by publishers. Based on the advertisements, every node knows message information from each of its neighbors. A subscription or query matching an advertisement will be forwarded to a broker upstream until it reaches the PEs [35].

The size of the query table significantly affects system performance. A large query table increases message processing time and requires more memory space. Subscription or query aggregation is used to reduce the size of the query table.

A precise subscription or query summary scheme is proposed for query aggregation. By using the precise query summary, each message is only forwarded to the nodes in the system that contains the exactly matched queries. This scheme is used in Siena [35], ONYX [55] and XNET [43]. In Siena [36], the authors tackle dynamic subscription management by periodically sending the sender request (SR) by the publisher to collect update reply (UR) generated by the routers and update routing tables. A precise query summary can minimize message traffic.

On the other hand, for a skewed message distribution, a precise summary cannot always provide good system throughput. The bloom filter is an approach for

generating a imprecise query aggregation. Extension has been added to the bloom filter to support XML messages [101, 69, 106]. Another imprecise summary method is proposed in [102]. By properly adjusting the summary precision, the authors can dynamically optimize the overall throughput.

In the second phase of message forwarding, filtering is conducted at each broker and the matched messages are forwarded along the reverse path of query routing, e.g. in Siena, ONYX and XNET. Event Distribution Network (EDN) system is an exception [102]. EDN uses a separate notification routing service to forward messages to subscribers via instant messaging, email, cell phone SMS.

2.1.4.2 Rendezvous-based routing

The rendezvous-based routing scheme is a method adopted in DHT systems, such as Scribe [38] and Hermes [85]. It can improve routing efficiency by decreasing the number of routing hops. A typical use of the DHT for storage and routing is discussed next. The distribution of data items is based on the DHT. Each data item is assigned a key based on hashing on the value of the data item. A message *put (key, data)* may be sent to any node participating in the DHT. The message is forwarded from node to node through the overlay network until it reaches the single node responsible for the key, where the pair (key, data) is stored. Any query for that data item can then retrieve the data by asking any DHT node to find the data associated with key by using a message *get (key)*. The message will be routed through the overlay to the node responsible for it.

The DHT-based system deals with the scenario where nodes join and leave. In Chord, as a new node joins the system, it is assigned a key generated by the DHT and distributed to the node with an identifier that has the nearest hash-value to the key. After a node leaves the network, keys it is responsible for are reassigned to its successor. A detailed discussion is provided in [97].

2.1.5 Classification of pub/sub systems

A number of pub/sub systems have been proposed. A classification, providing a complete view of these systems, is presented in Table 2.1. A number of systems discussed in the previous sections are compared. Each row in the table corresponds to a system feature; each column corresponds to a system discussed earlier and lists the various characteristics of the system.

Table 2.1: Classification of existing content-based systems

	Scribe	TIB/ Rendezvous	Gryphon	Siena	ONYX	XNET	Mesh-based XML	XTreeNet	EDN	Sieve
Topic-based	yes	no	no	no	no	no	no	no	no	no
Content-based	no	yes	yes	yes	yes	yes	yes	yes	yes	yes
XML-based	no	no	no	no	yes	yes	yes	yes	yes	yes
Application layer multicasting	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
Structured overlay	yes	no	no	no	no	no	no	no	no	no
Unstructured overlay	no	hierarchy tree	acyclic P2P	acyclic P2P	acyclic P2P	acyclic P2P	acyclic P2P	acyclic P2P	one level branching tree	acyclic P2P
Subscription summary	no	no	no	precise	precise	precise	no	no	imprecise	yes

2.2 Background for XML, XPath, and XSLT

In this section, a brief introduction of XML, XPath and XSLT is presented, which is related to the topic of this thesis: XML-based pub/sub systems.

2.2.1 XML introduction

Extensible Mark-up Language (XML) [9] is a simple, very flexible text format derived from SGML (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere [9]. XML represents structured information in a machine-readable format which eliminates differences of proprietary protocols between networks, operating systems, and platforms, and enables different systems to automatically communicate with each other.

XML has a fixed syntax and an unlimited vocabulary. This extensibility allows the addition of new capabilities; for example, two distributed application components can communicate by exchanging XML messages [74]. The tag names and attribute names are referred as *structural information*, and the text and attribute values as *value information*. Document Type Definition (DTD) and XML schema define the structure, content and semantics of XML documents for a particular industry or a set of tasks.

Each XML document can be modeled as a tree of nodes. There are seven types of nodes: root nodes, element nodes, text nodes, attributes nodes, namespace nodes, processing instruction nodes, and comment nodes. Figure 2.5 presents a sample XML document which documents the book information at a bookstore. Information for each book includes title, author, year and price. Figure 2.6 depicts a tree modeled from this example. The root element in the example is <bookstore>. All <book> elements in the document are contained within <bookstore>. The <book> element has four children: <title>, <author>, <year>, <price>. The *attributes* nodes in this example include @lang and @category. The *processing instruction node* in this

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bookstore>
    <book category="COOKING">
        <title lang="en">Everyday Italian</title>
        <author>Giada De Laurentiis</author>
        <year>2005</year>
        <price>30.00</price>
    </book>
    <book category="CHILDREN">
        <title lang="en">Harry Potter</title>
        <author>J K. Rowling</author>
        <year>2005</year>
        <price>29.99</price>
    </book>
    <book category="WEB">
        <title lang="en">Learning XML</title>
        <author>Erik T. Ray</author>
        <year>2003</year>
        <price>39.95</price>
    </book>
</bookstore>
```

Figure 2.5: An XML document sample from [2]

example is <?xml version"1.0" encoding="ISO-8859-1"?> which begins with <? and ends with ?>. Each element and attribute has one parent. Element nodes may have children. There is a *sibling* relationship between element nodes if they have the same parent. The title, author, year, and price elements are siblings. An *ancestor* of a node t is the node in the path from t to the root except t. The ancestors of the title element are the book element and the bookstore element. The *descendants* of a node t are all the nodes of subtree rooted at t. For instance, the descendants of the bookstore element are the book, title, author, year, and price elements.

Typically, an XML document needs to be parsed before it can be consumed by XML applications. The data in the XML document are available to the applications after a document is parsed. There are two methods for XML parsing. In the first method, Document Object Model (DOM) parser, an XML document has to be converted to a tree in memory as long as the tree can fit in memory. The second method is to use a Simple API for XML (SAX) parser which is an event-based parser. The SAX parser can interject application code at important events within the document is parsed. The difference between the SAX parser and the DOM parser is that the SAX

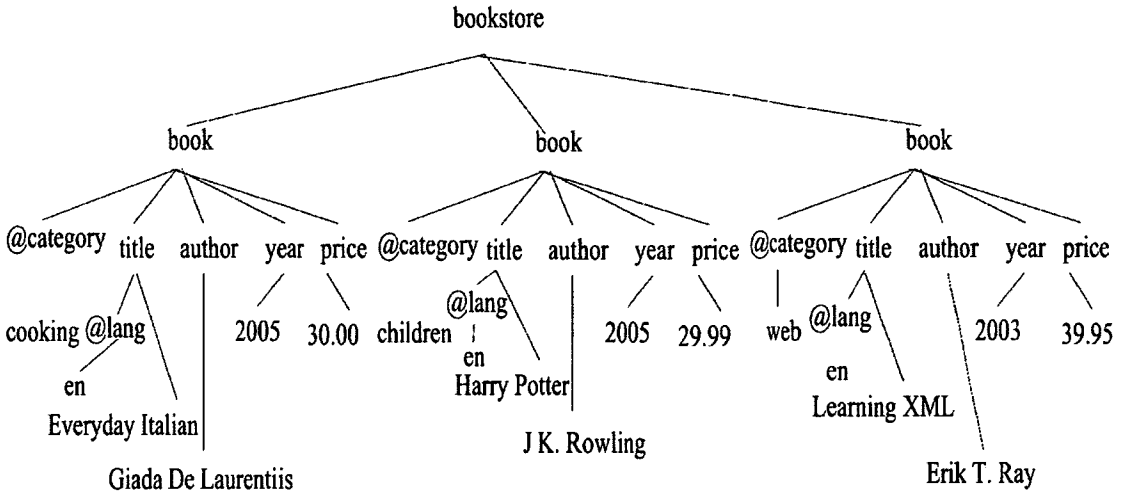


Figure 2.6: An XML tree modeled from Fig. 2.5

parser allows an XML document to be handled sequentially without first reading the entire document into memory. To use the SAX parser, the users have to instantiate a SAX parser instance and register what methods are available for callback and use within an application. The example methods defined in the ContentHandler interface in an SAX parser, which are associated with an XML document's content, include startDocument(), endDocument(), startElement(), endElement(), characters().

2.2.2 XPath introduction

XPath [3] is a language for retrieving, selecting, and filtering information from an XML document. After becoming a W3C standard in 1999, XPath has been used in different XML-based languages. For example, XQuery [13] and XSLT [5] are built on XPath expressions.

The basic syntactic construct of XPath is the *expression.* The types of expression include: node set (an unorderd collection of nodes without duplicates), boolean(true or false), number, and string. *Location path expressions* are the most important expression in XPath. They select a set of nodes that match the selection at its

place whenever it is in an XML document and can also contain expressions that filter the set of nodes that has been selected. There are two kinds of location path expressions: absolute location paths and relative location paths. The absolute path expression selects nodes from the root and the relative path expression selects nodes in the document from the current node that match the selection from all locations. A location path expression consists of several *location steps*. Each location step returns a set of nodes that is subsequently used for the next location step as the current context. A location step has three sub-parts: an axis, a node-test, and zero or more predicates. The syntax for a location step can be expressed as :

axis::node-test[predicate].

The axis specifies the tree relationship between the context node and the nodes selected by this location step. For example, child (/), parent, ancestor, descendant (//), following-sibling, preceding-sibling, following, preceding, attribute (@), namespace, self (.), descendant-or-self (.//), and ancestor-or-self are example axes in XPath. The node test specifies the node type and the nodes selected by the location step. The predicates are expressions which can refine the set of nodes selected by the location step.

XPath offers more than 100 functions. There are functions for node and qualified names (QName) manipulation, date and time comparison, numeric values, string values, boolean values, and more. QName defines valid identifiers for elements and attributes. QName creates a mapping between the URI and a namespace prefix. Table 2.2 lists some core XPath functions.

All these provide an easy data search method from any part of the XML document. An example XPath query for weather information of Ottawa, including temperature, wind, and snow is represented as:

/weather[@city=Ottawa][Temperature][Wind]/Snow

Table 2.2: Examples of core function library in XPath

Node Functions	String Function	Boolean Function	Number Function
last()	string(object?)	boolean(object)	number(object?)
position()	concat(string,string,string)	not(boolean)	sum(node-set)
count(node-set)	starts-with(string,string)	true()	floor(number)
id(object)	contains(string,string)	false()	ceiling(number)
local-name(node-set?)	substring-before(string,string)	lang(string)	round(number)
namespace-uri(node-set?)	substring-after(string,string)		
name(node-set?)	substring(string,number,number?)		
	string-length(string?)		

2.2.3 XSLT introduction

Extensible Stylesheet Language Transformations (XSLT) [5] is a language for transforming one XML document into another XML document or an html document or other text-based formats. XML data is rarely used directly in the form in which it arrives and has to be transformed. For example, the weather message has to be transformed to a format defined by receivers before being displayed. An XML message is converted into different text formats or multimedia format for human receivers. To transfer data between applications, XML messages need be transformed from one data model to the model used by another application [67].

A transformation expressed in XSLT is called a stylesheet. Every XSLT document is well-formed XML document. A stylesheet document contains a set of template rules. Each of these template rules is composed of two parts: a pattern that is matched against the nodes of the source tree and a template that can be called to form a part of the result tree. When the XSLT processor receives an XML document to be transformed by an XSLT stylesheet, the XSLT processor looks at every node in the XML document and checks it against the pattern of each template rule in the stylesheet. If there is a match, the processor outputs the template's rule.

<xsl:template match="expression">

This element *xsl:template* corresponds to a template's rule. The pattern is specified in the *match* attribute of the *xsl:template* element. The template of the content's

output is specified inside the *xsl:template* element.

<xsl:element name="element-name">

The *xsl:element* does the effective transformation from the source XML message tree into the result XML tree.

<xsl:call-template name="template-name">

The *xsl:call-template* element calls a named template.

2.3 Existing work on XML filtering and matching

XML filtering has a well known problem with large number of subscriptions or queries and with indexing queries for efficient matching. The XML filtering technique is used to match an XML document d with an XPath query Q to identify whether d has the tree structure and values defined by Q. Figure 2.7 shows an example. In this example, the XPath query specifies a user's interest: the information of Ottawa weather including temperature and wind chill. The corresponding matched elements in the XML document is highlighted using a dashed rectangle.

Various XML filtering techniques are proposed. They can be classified into the following categories: top-down matching, bottom-up matching and sequence-based matching. Filtering techniques such as Xfilter [18], Yfilter [54, 53], Xtrie [39], Afilter [28], Gfilter [47], and Bfilter [50] are reported in the literature.

2.3.1 Top-down XML filtering and matching

The top-down approach is designed to process XML messages by searching from the root of a subscription index tree, which indexes a set of subscriptions, to enumerate every subscription node matched at every location step. This task splits a path expression into a set of multiple simple path expressions branch by branch. Then the XML document is compared with sub-queries by following a set of filtering steps to produce matched queries. Existing projects like Xfilter [18], Yfilter [54, 53], Xtrie [39]

```
<?xml version="1.0" encoding="UTF-8"?>
<file_header>
   <root>
      <Toronto>
         <temperature>-17.5 Light Snow</temperature>
         <windchill> -22</windchill>
         <wind> 6.4</wind>
         <clouds> Mostly Cloudy 335</clouds>
      </Toronto>
      <Ottawa>
         <temperature>0.2 Overcast</temperature>
         <windchill> -6</windchill>
         <wind> 24.1</wind>
         <clouds> Overcast 609</clouds>
      </Ottawa>
      <Sudbury>
         <temperature>-8.2 Light Snow</temperature>
         <windchill> -8</windchill>
         <wind> 4.8</wind>
         <clouds> Scattered Clouds 914</clouds>
      </Sudbury>
   </root>
</file_header>
```

XPath query example: //Ottawa[temperature][windChill]

Figure 2.7: An example of filtering an XML document with an XPath query.

and Bfilter [50] belong to this category. We find that many of the filters are based on automata (either a Nondeterministic Finite Automaton (NFA), or a Deterministic Finite Automaton (DFA)) besides Xtrie which uses a *trie* index. NFA-based approaches are space efficient because they require a relatively small number of states to represent a large set of queries [66].

Xfilter and Yfilter are two representative Finite State Machine (FSM)-based filtering systems. They use an FSM to represent a subscription, where a location step of a subscription is mapped to a state. Thereby, the problem of path expression matching is reduced to a matching problem of finite state automata. A Simple API for XML (SAX) parser is used to drive state transitions of the automata to process the filtering task. State transition occurs in a case of a state match. Xfilter creates an FSM for each subscription and employs a dynamic index to hold all subscription states. Yfilter is designed to extend Xfilter by sharing states among queries. It merges common

states of query FSMs into a single NFA machine. The operator '//' of a subscription may result in a state being visited many times. Bfilter starts its operations from the lowest branch point. Bfilter is good for complex queries [50].

In comparison, DFA-based approaches are time efficient because their state transitions are deterministic. Since the conversion from an NFA to a DFA through subset construction [104] increases the number of states exponentially, recent work such as the lazy DFA [62] chose to convert an NFA into a DFA lazily at runtime. The lazy DFA attempts to perform the subset construction only when needed in order to improve system scalability.

Xtrie is an efficient XML filtering engine [39]. It is based on a string matching algorithm. First, it treats a subscription as a string and splits the subscription string at the position of a descendant operator '//'. Then these substrings are indexed and organized into a trie data structure. Matching of substrings in the trie is shared among queries but transitions between substrings are done using level check query by query. Xtrie supports ordered matching of XML data against tree-pattern XPath expressions.

2.3.2 Bottom-up XML filtering and matching

The bottom-up approach is described in [28]. In the bottom-up approach, result enumeration is delayed until a match is found for the last location step of a subscription. A representative filtering engine of the bottom-up method is Afilter [28].

Afilter represents and clusters location steps of all XPath subscriptions in a query graph. The query graph is called AxisView. Afitler can process subscriptions containing child axis ('/'), descendant axis ('//') and wildcard ('*'). It does not support the twig-structure subscriptions and predicate matching. The job of a filtering task is as follows: a SAX parser is used for dissecting an XML message into discrete elements and attributes. When a start tag is met, a stack object is created and stored in a stack identified by the start tag name and in a stack for the wildcard when any

subscription contains a wildcard symbol. Objects in these stacks are organized by pointers corresponding to outgoing edges of a node (corresponding to the start tag) in AxisView. The traversing these stacks is only triggered to check whether levels between data elements satisfy the level requirements in a subscription when a trigger condition is satisfied. A trigger condition means a match of the final location step of a subscription. This is different from Yfilter which tracks all matched transition states for each start tag. Afilter avoids this early result enumeration. The basic memory requirement for Afilter is linear to the message depth. The time complexity is the product of the message size and the number of subscription. The empirical results show that the basic performance of Afilter is slower than Yfilter. With more memory resources provided, Afilter outperforms Yfilter by using the prefix caching for subscriptions with common prefixes.

A new bottom-up NFA-based path matching algorithm, called Gfilter, is proposed in [47]. The improvement of Gfilter lies in that it encodes the matching document nodes into a tree, called Tree-of-Path (TOP) encoding scheme. This encoding scheme is applied into Gfilter's bottom-up content matching algorithm. Gfilter enables the suffix and prefix sharing between subscriptions. Gfilter is efficient for post-processing general-tree-pattern subscriptions. Empirical performance evaluations show that Gfilter achieves better filtering performance in comparison to other state-of-the-art algorithms, Yfilter and Afilter.

2.3.3 Sequence-based XML filtering and matching

Besides the two approaches described earlier, other sequence-based filtering engines, such as Filtering by Sequencing Twigs (FiST) [70], XFIS [19] and branch sequencing [87], are proposed. The allowable syntax of XPath expressions in FiST is $P^{/,//,*,[\,]}$. An advantage of FiST is that it filters the subscription holistically by transforming an XPath expression and an XML message into a prüfer sequence because each tree has a unique sequence and a tree can be reconstructed from this

sequence [10]. The limitation of FiST is that there is no sharing across subscriptions, and each subscription is filtered independently.

XFIS [19] is a sequence filtering technique. It is based on a prüfer sequence. XFIS removes the post-processing step for branch node verification by adding an additional check while performing subsequence matching. The performance of XFIS is superior in terms of time and space complexity to the FiST system.

A third sequence matching method called branch sequence matching is proposed [87]. In this method, a twig (tree-structure) subscription is transformed into a branch sequence. Each node in a subscription is assigned a pre-order number. The sequence is constructed branch by branch. Two rules are associated with the construction for an XML document. First, if a leaf node is reached, the nodes from the closest branch node to this leaf node are output and this branch, excluding the branch point, is removed. A branch node is a node with two or more children. Second, if the last leaf node of a twig query is encountered, the nodes from the root of this pattern to the leaf node are output. Branch sequence matching scheme is slightly different from the one introduced by FiST because it can retrieve the matched nodes in a single parse of the input document. In contrast, FiST has to parse the document twice, indexing the document nodes in the first parse [60]. The branch sequence matching scheme can provide false positive matched answers.

Authors in [92] have designed a new sequence-based matching algorithm by transforming subscriptions to Node Encoded Tree Sequences (NETS). This technique identifies the set of matched queries free of false positives with a single scan of the XML document. Extensive experimental results show that this method outperforms Yfilter and FiST.

The authors in [99] propose to aggregate multiple XML messages into one message using XML structural summaries [78]. XML filtering is applied to this aggregated message in a batch mode, instead of the original message. The advantages of XML message aggregation and batch processing are that it may decrease communication

costs and increase the performance of the message routing process. This scheme is based on the assumption that XML routers will receive the published documents in streams (batches). This assumption is not supported by many applications. Therefore, the use of this technique is limited to a certain category of applications [60].

2.3.4 Others

In addition to the techniques discussed earlier, there are other XML filtering methods, such as Bloom Filter [61]. Silvasti et al. describe a schema-conscious XML filtering engine to remove descendant operators (//) and wildcards (*) from the linear XPath subscriptions through rewriting XPath queries using information in a DTD [95]. A more detailed discussion is provided in [60, 17].

Querying XML streams has become a crucial problem in modern information systems. In contrast to XML that is parsed and stored in databases, streaming XML arrives in order (typically as a sequence of SAX events) and can be most efficiently processed by consuming such SAX events on-the-fly without extensive buffering. Han et al. [65] adapt the holistic twig join algorithm for matching queries with streaming XML and extracting the matched elements.

2.4 Existing work on caching for pub/sub systems

Caching for pub/sub systems has been investigated by researchers. Some researchers discuss caching system in a general pub/sub system and some researchers consider caching intermediate results during XML filtering. A summary of the work on caching in the context of pub/sub systems is presented.

In [58], the authors present a solution to optimize the performance of web service applications by caching SOAP messages at the web service level. The caching information is added to the header of SOAP response messages. The web service application decides what caching information will be included and is responsible to send caching information to clients. In order to take advantage of caching, the web service

entity and the client entity should understand the caching information included in SOAP messages. A traditional HTTP-level caching system uses HTTP headers to manage the expiration of the HTTP responses as presented in [14]. The limitation of HTTP-level caching system is that it depends on a transport layer protocol, HTTP, for instance and programmers cannot control the duration of a response depending on the logic of the web service. Furthermore, besides the caching architecture, caching consistency is investigated in [58] as well. Weak caching and strong caching can be used to update cached messages to keep cache consistency. In weak caching, the messages sent to clients can be stale. It is up to the client entity to decide whether to accept stale messages or to make an invocation. Strong caching does not allow stale message to be sent to clients and is implemented either by letting a client application to perform a poll of the web service every time an invocation is to be started or by letting a web service send invalidation messages to clients, for instance, invalidation with a time-to-live attribute.

The work in [96] incorporates cache to help subscribers request history data by introducing a request/response module to a pub/sub system. A cache buffers transmitted messages for future requests and is placed either at every broker (basic policy) or at leaf brokers only (leaf policy). The experiments show that the basic policy outperforms the leaf policy when serving requests for history data from subscribers. In the paper [52], caching is combined with a pub/sub system. In the proposed service model, a subscriber is associated with the maximum number of messages received per service period and the maximum age of a message a subscriber is interested in. A router in the proposed pub/sub model is provisioned with a cache which stores messages from local publishers, messages received for local subscribers and messages opportunistically cached as they transited through them. By default, a message is cached in a router where it is initially uploaded and to the router that subscribes it. A message also can be cached in every router along the path from a publisher towards the requesting router. The protocols for message forwarding and subscription

forwarding behave the same as a generic pub/sub system. A refreshing protocol is introduced to deal with expired subscriptions. Different caching selection and dropping strategies are evaluated including pending publication first policy (PPF), dispatched publication first policy (DPF). Further discussion of them is available from [52].

Caching intermediate results in XML filtering is discussed in [66]. XPath subscriptions are indexed using an NFA. The NFA is transformed into a DFA. Frequently occurring state transitions from the automata are cached in memory [66] and can thus be served directly from the cache instead of from the automata. However, there are two limitations for this scheme. The first limitation [66] is that predicate matching is not provided whereas the schemes described in this thesis support full predicate matching. The second limitation is that the system may invoke the automata even when previously processed messages arrive.

In [80], Obermeier and Böttcher present an XML fragment data caching technique for large-scale mobile systems with many mobile participants. The authors assume that each user can search its neighboring devices for parts of the requested information, and each user can advise its neighbors to forward the request to other users. They also assume that cached contents do not change frequently. The technique splits an XML document into several disjoint fragments $\{F_1 \ldots F_n\}$ using a split schema that depends on application requirements. The split schema is generated from a DTD by applying a sequence of split operations, each of which splits a given XML document into two or more sub-fragments F_i. The cached data unit is a fragment F_i. For example, if the range of an attribute *ID* in an XML document is [1..100], the scheme can split the data into two ranges [1..50] and [51..100], each containing 50 IDs. The benefit of this scheme is that users can share cached fragments to other devices and relieves workloads of the central system. Significant reduction of performance metrics, such as the overall data transfer times and query response times, can be achieved.

In [72], the authors present a framework for cooperative XPath caching in XML database. XPath caching is often discussed in a client/server architecture, while the

cooperative caching scheme is discussed in the context of DHT. The motivation for cooperative caching is briefly explained. In many applications, such as when deploying web services, or accessing XML sites from the web, XML documents are queried by a number of peers in close network proximity with one another. The actual documents themselves are located in a large number of remote data sources, thus the cost of locating them and transferring the answers to the peers is high. Reusing the results of the queries of each peer can be reused to answer subsequent queries posed by other peers that submit similar queries can improve performance by reducing the network cost. The proposed scheme allows sharing cache content among a number of peers. To facilitate sharing, a distributed prefix-based index is built based on the queries whose results are cached.

2.5 Existing work in the XML message routing

Existing content-based pub/sub network architecture design can be broadly categorized into two classes: the conventional application-layer multicast approach [35, 36, 37, 55, 44, 43, 42, 41, 71], and the channelization approach [15, 90, 103, 82].

In the conventional application-layer multicast approach, brokers are organized into dissemination trees first. Every broker maintains a query table that summarizes subscriptions from subtrees. A publication message is matched against subscriptions in the query table at every hop, and is forwarded only towards matched subscribers [30].

In the channelization approach, the message space is partitioned into a small number of event channels. For each channel, a multicast group which spans all brokers that subscribe to any message in that channel is built [30].

Overlay networks can be unstructured and structured. If brokers of a network are organized in a tree, all brokers communicate either as a client or as a server, such as in the event-based system JEDI [49, 48] and in TIB/Rendezvous message middleware [11]. If brokers of a network form an acyclic P2P topology, all brokers

communicate with each other as peers. Existing systems, like Siena [35, 36, 37]and IBM Gryphon, are built on the top of an acyclic P2P networks.

2.5.1 Filter-based XML message routing and forwarding

In the conventional message routing approach, a message is received and then is filtered at each broker along a message forwarding route. In such a scheme, subscriptions are stored at each broker, and filtering has to be performed at every broker in order to find the next hop. When a subscription is received, the new subscription is inserted into a subscription table by the broker. It is continually forwarded to the next nodes towards the publisher edge node. For the process of message forwarding, if the message is found to be a matched item, the message is forwarded to each of the matched nodes that in turn repeats the same filtering and forwarding steps.

Siena is a representative non-XML content-based pub/sub system on a unstructured overlay network. The data model it supports is attribute-value pairs. Message brokers are connected by minimum spanning trees rooted at publisher edge nodes. An application-layer multicast scheme is used for message routing. All user subscriptions are stored and aggregated at each node along the path from the leaf node to the root node of a distribution tree rooted at a publisher. In turn, a published message needs to be matched at each intermediate node along the paths in the tree. A message is forwarded to a node when a subscription stored on it can match this message.

Existing XML-based pub/sub systems adopt such an approach for XML routing. For example, ONYX [55] is one of the first XML-based pub/sub systems using the conventional content-based routing method. All subscriptions are stored at each node along the path from the leaf node to the root node of a distribution tree of a publisher. Published messages need to be repeatedly matched to subscriptions at each node along the overlay paths to relevant subscribers. ONYX uses Yfilter [53] as a filtering engine; XML message forwarding and subscription routing schemas are the same as used in Siena. However, ONYX does not address the issues of subscription

management and network reliability.

XNET [44, 43, 42, 41] is another representative XML content network. All subscriptions are stored at each node along the overlay path in a publisher-based distribution tree. Published messages need to be matched to subscriptions at each node along the path. Xtrie is the filtering engine, which makes use of subscription aggregation to reduce the size of routing tables while ensuring precise routing. Additionally, XNET designs a reliability mechanism to deal with network failures. Moreover, it addresses the issue of subscription cancelation in order to support subscription management.

In [71], the authors present an XML-based pub/sub system. Covering and merging XPath subscriptions are used to generate a compact routing table. Message advertisement is exploited to reduce unnecessary traffic by flooding subscriptions; however, the protocol for XML routing is the same as in previous approaches, where subscriptions are stored at each node along the overlay path of a distribution tree.

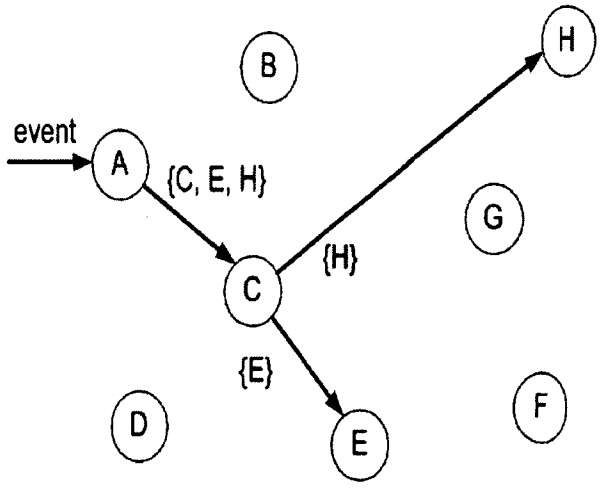


Figure 2.8: Event delivery in MEDYM network from [30]

In [30] and [29], a new architecture for content-based pub-sub network, called Match Early with Dynamic Multicast, shown in Figure 2.8. The term *event* in [30, 29] and the term *publication message* in this thesis are interchangeable. This system matches subscriptions from remote servers to calculate a destination list of servers with matching subscription, then routes a message to the destination servers. A

broker computes the next hops and a new destination list for each next hop as it receives a message.

2.5.2 Channelization-based XML message routing

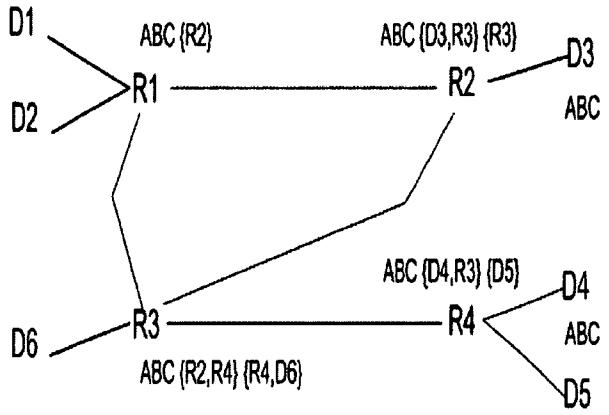


Figure 2.9: XTreeNet: Protocol Example

XTreeNet, as depicted in Figure 2.9, is an architecture for supporting XML-based systems, which utilizes a combination of the pub/sub and the query/response model to obtain better performance [57]. XTreeNet uses a content descriptor (CD) to describe the information of subscriptions and published messages. A distribution tree is established for each CD that is rooted at each publisher. The first distinction between systems like ONYX, XNET and XTreeNet is that in ONYX, every broker needs to perform matching. In XTreeNet, only the first hop (source overlay) router needs to map messages to CDs, whereas the intermediate brokers only need to do forwarding. The second distinction is that although the publisher-based distribution tree is optimal in ONYX and XNET, it is expensive to maintain when the number of publishers is large [57].

Figure 2.10 depicts the basic SemCast [82] system model. There are source brokers, gateway brokers, and internal brokers. The source brokers connect to publishers and the gateway brokers connect to subscribers. The coordinator is responsible for channelization management. It communicates with the source brokers and gateway

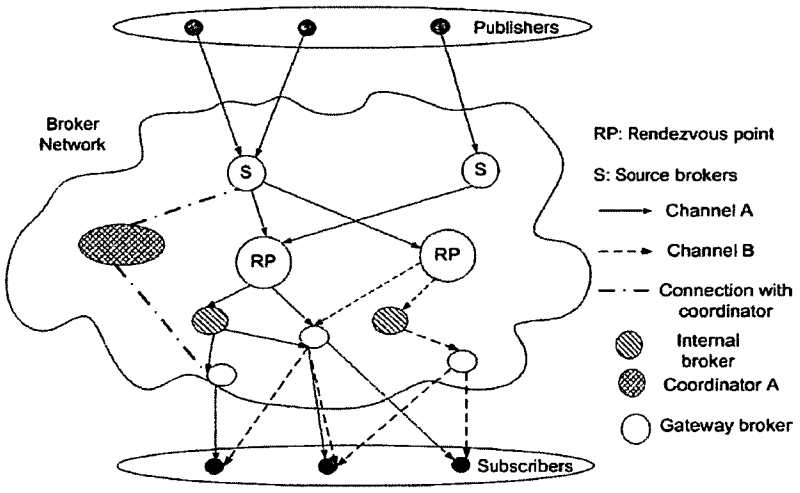


Figure 2.10: Basic SemCast system model from [82]

brokers. SemCast aggregates and merges subscriptions into channels. A published message is matched to such aggregates. After matching, each published message needs to be mapped to corresponding channels and forwarded to destinations via specific channel(s). Each channel is determined in a centralized manner using a cost-based model and is implemented as a multicast tree. Subscriptions are sent to the coordinator to check whether an existing channel covers it using a containment algorithm, then join corresponding trees.

2.5.3 Rendezvous-based XML message routing

Net-X [88] is a proposed pub/sub system built on top of a distributed hash table. The overview of Net-X is depicted in Figure 2.11. Net-X uses signatures to discover interest match between user interests and published data. The published document varies, for example, small text files or MPEG7 movies, require a different distribution tree for different documents. Net-X builds a per-document tree. For each published data, there is an overlay distribution tree rooted at the publisher connecting only the users who are interested in this specific document.

Hermes [85] is an example of type-based pub/sub systems, which has an XML-based API with Java language binding. Hermes uses network of event brokers built on

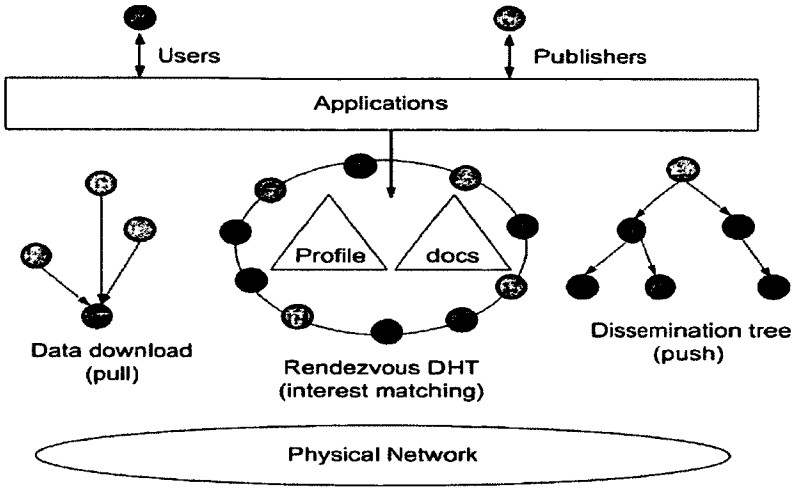


Figure 2.11: Overview of Net-X from [88]

top of a structured overlay network for rendezvous-based routing and fault-tolerance.

2.5.4 Others

In [26], the author proposes to deliver a pub/sub XML multicast service within a Virtual Private Network (VPN). A VPN is implemented on top of a backbone network, such as an IP/MPLS backbone [26]. Each VPN has a unique label. An XML router can be embedded in a publisher edge node (PE) or connected to a PE. An XML router can communicate on multiple carrier-based VPNs crossing a PE based on unique identifiers [26].

In [84], the authors propose a common API for pub/sub systems in order to build high-level services on top of existing pub/sub systems. XML messages are defined using the XML-RPC data model and subscriptions are described using a lightweight XPath grammar. The API supports three levels of compliance (with optional extensions). The level 1 compliance specifies abstract operations without specifying an implementation or data model; the level 2 compliance describes interactions using an XML-RPC mechanism; the level 3 compliance uses an XML-RPC data model to wrap different distributed pub/sub systems and hides the semantics difference of events and subscriptions used in the systems. Siena [35, 36, 37], Hermes [85], and Scribe [38] are

used as case studies in [84].

2.6 Existing work on XPath query aggregation

In this section, the results of theoretical analysis and a summary of existing algorithms for the XPath query aggregation are presented. Definition 1 describes the XPath query containment relationship [76, 93].

Definition 1. "For two XPath subscriptions p and q and an XML document t, a containment (partial order) holds if every XML document t that matches p also matches q (denoted $p \subseteq q$)."

For the XPath queries containment problem, the complexity has been studied by a considerable number of forerunners of the field. Miklau and Suciu [76] prove that the containment problem for any combination of two operators in the set of {*, //, [ ]} is of polynomial time (PTIME). For $XP^{\{*,//\}}$, the containment problem is equivalent to the string matching problem; for $XP^{\{[\ ],//\}}$, there is a polynomial time containment algorithm; for $XP^{\{[\ ],*\}}$, a polynomial time containment algorithm follows from classic results on acyclic conjunctive queries. Miklau and Suciu also prove the containment problem of $XP^{\{[\ ],*,/,//\}}$ queries is co-NP complete. Wood [105] studies the problem of XPath query containment under DTD constraints and shows that the containment problem can be decided in polynomial time. In [79], Neven and Schwentick discuss the complexity of the containment of various types of XPath queries in the presence of disjunction, DTDs and variables. The complexity results when alphabet Σ is infinite and finite are summarized in Table 2.3 in the absence of a DTD. If a DTD is present, the complexity results for different XPath types are listed in Table 2.4. The complexity of almost all decidable XPath queries lies between co-NP and exponential polynomial time (EXPTIME). The + means the XPath operator in the corresponding column is selected.

In order to check containment $p \subseteq q$, the technique of checking $p(t) \Rightarrow q(t)$ for all XML trees t can not be used in practice because the number of comparisons can be

Table 2.3: The complexity of containment of different types of XPath queries

Σ	/	//	[ ]	|	*	complexity
infinite	+	+	+	+	+	CO-NP
infinite	+			+		CO-NP-hard
infinite		+		+		CO-NP-hard
finite	+	+	+	+	+	PSPACE
finite	+	+		+		PSPACE-hard

Table 2.4: The complexity of containment in the presence of DTDs (from [79])

DTD	/	//	[ ]	|	*	complexity
+	+	+			+	in P
+	+		+			CO-NP-complete
+		+	+			CO-NP-hard
+	+	+	+	+	+	EXPTIME-complete
+	+	+		+		EXPTIME-complete
+	+	+	+		+	EXPTIME-complete
+	+	+	+		+	undecidable with nodeset comparisons

an exponential function of of the number of trees t. Practical techniques for checking query containment are: canonical model technique, homomorphism technique, automata technique, and chase technique [76]. Each of these techniques uses the simple fact that $p \not\subseteq q$ iff there is a counter-example, i.e., a tree t such that $t \models p$ but $t \not\models q$ [93]. The canonical model restricts the search space to canonical trees with a similar shape as the pattern p. The homomorphism technique finds a homomorphism from p to q. The automata technique constructs two tree automata and checks containment between the languages defined by the automata. The chase technique translates the XPath queries into relational queries and uses the relational chase method.

Let x be a node of an XPath query p. We denote the root node of that query p by root(p) and the label of that node by label(x). A child edge exists between two nodes x and y if and only if x and y have a parent-child operator. A descendant edge exists between two nodes x and y if and only if x and y have an ancestor-descendant operator. A definition of homomorphism [76] is presented next.

Definition 2. "A homomorphism is a function h: $nodes(p') \rightarrow nodes(p)$ between two patterns p' and p. A homomorphism should satisfy the following conditions:

- $h(root(p')) = root(p)$;
- For each $x \in node(p')$, label(x)=* or label(x)=label($h(x)$);
- For each $x, y \in node(p)$, if (x, y) is a child edge in p', then $(h(x), h(y))$ must be a child edge in p; if (x, y) is a descendant edge in p', then $(h(x), h(y))$ must be a path in p of *length* ≥ 0, which may include child edges and/or descendant edges. The length of a path here is defined to be the number of intermediate nodes between x and y. For example, if ($h(x)$,$h(y)$) is a path of length 0, then $h(x)$ is the parent of $h(y)$."

The existence of a homomorphism is a sufficient condition for $p \subseteq p'$. A homomorphism between two XPath patterns can be found in polynomial time [76]. The time complexity for checking the existence of a homomorphism from p' to p is O ($|p|^2|p'|$). Homomorphism is a sufficient but not necessary condition for containment. If two queries have a containment relationship but do not have a homomorphism relationship, then an XML pub/sub system would forward unnecessary messages for query routing. However, this would not affect the system correctness. Figure 2.12 shows a simple example for containment and homomorphism. In this example, there is a homomorphism between query p' and p and $p \subseteq p'$. Node a of query p' has a matched node a in query p and node //c of query p' has a matched node c in query p.

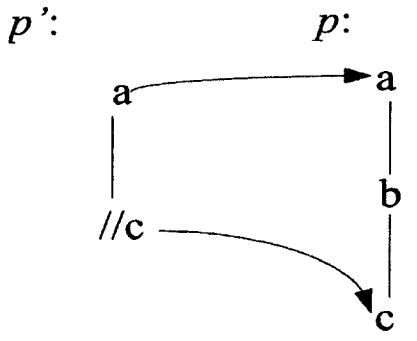


Figure 2.12: A simple example for containment and homomorphism

2.6.1 XML documents aggregation

The paper [99] proposes to aggregate multiple XML messages into one message using XML structural summaries [78] to enable batch processing. The data structure, called RoxSum, aggregates the structural information for multiple documents. A RoXSum tree is built during the XML document parsing process when a stream of XML documents is accepted by a router. The parser reads the XML document in-order and the parsing process adds either new nodes or new document identifiers to the RoxSum tree. The automata-based XML filtering algorithm evaluates subscriptions against the RoxSum tree. Vegenna et al. extend the RoXSum data structure by proposing a new data structure Value-Aware Routing XML Summary (VA-RoXSum) to aggregate the structure information and element values of multiples messages [100] and [77, chap. 5]. A set of element values (values under each root-to-leaf path) are encoded to a signature using a bloom filter and the generated signature is added to the finite state machine along with document identifiers. Message filtering is performed by combining a finite state machine (to verify structural constraints) and a bloom filter (to verify value constraints).

In [51], authors propose a method to compute XML message similarity by counting the structural similarity and value similarity; nodes in an XML message are converted to a pre-order sequence of the message. The structural similarity between SOAP messages is computed as a ratio of the number of common node tags in two SOAP messages and the maximum number of node tags in two SOAP messages. The value similarity is measured by using an edit distance method that counts the minimum number of edit operations needed to transform one string into another, the XML message similarity is the minimum value of the structural similarity and value similarity.

Chapter 3

Caching for optimizing XML pub/sub system performance

Caching is useful to improve the performance of systems in which information used in the past is reused in the future. This chapter presents two software-based caching schemes for XML filtering. Extensive experiments are performed using synthetic data. In addition, real-life traces for stock data and weather data are collected and experimented with.

3.1 Caching schemes for improving XML filtering performance

XML filtering is complex and a time consuming operation. After receiving a duplicate message, the broker in a traditional pub/sub system reincurs the same significant filtering overhead and obtains the same subscription matches. Since caching is effective in speeding up system performance in many different contexts, this thesis investigates caching to improve the performance of XML filtering by reusing previous filtering results. Two caching schemes: (i) caching entire messages, and (ii) caching the structural summary of messages are experimented with. In the following section,

a complete message caching solution is addressed. This is followed by a discussion of structure-based message caching (see Section 3.1.2). Results of experiments with synthetic data are discussed first. The experiments with real-life data traces are presented at the end of this chapter.

3.1.1 Complete message caching

The architecture of a content filter engine with a cache component is shown in Figure 3.1. A subscription-based content filter engine typically includes an XML parser to parse a new XML message to generate a sequence of XML events; an XPath parser to analyze user profiles expressed in XPath queries to generate subscription tokens and subscriber information; and an XPath-based filtering engine to index subscription tokens and to generate a subscription match set. A cache component is applied to the filtering engine to keep track of cache instances of previously processed messages. The parsing function is performed by an XML parser, which may be a Simple API for XML (SAX) or a Document Object Model (DOM) parser. A SAX parser is an event-driven parser that implements SAX functions as a stream parser and a DOM parser is a parser that converts an XML document into an XML DOM object which can then be manipulated. The parser is used to dissect XML document into elements and attributes, to check message validity against DTD or XML schema, and is a bridge between the XML document and the application that processes XML.

Figure 3.2 illustrates the detailed functional flow for the pub/sub system architecture shown in Figure 3.1. When a new XML message arrives, it is analyzed to check whether or not the message is a duplicate of an earlier message. A subscription is parsed by the XPath query parser and the generated subscription tokens are indexed by the filter engine. In Figure 3.2, the XML filtering and matching system can access the filter engine and cache engine based on the generated hash code of a message. The cache engine performs two main operations (receive and generate hash code) when a message PM_i arrives.

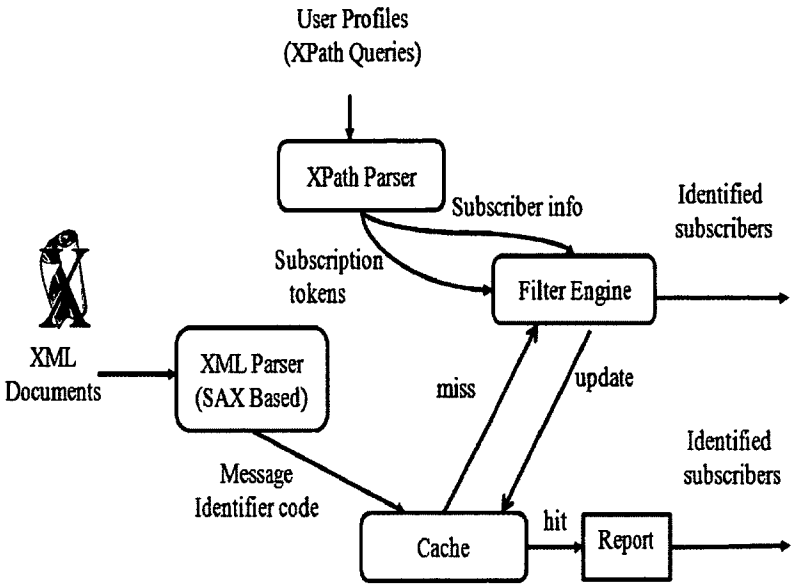


Figure 3.1: System architecture for message caching

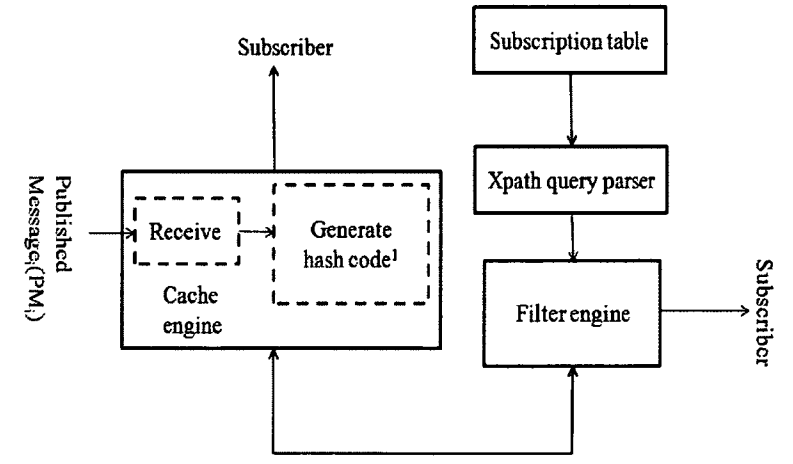


Figure 3.2: Detailed functional flows of a pub/sub system architecture

An example of a cache hit detect step is provided. For a message PM_i, after the parsing is done, a hash code K_i is generated, where K_i= hash (PM_i). The XML filtering and matching system searches the cache using K_i to locate a valid entry in the cache. If a cache hit is detected, matched subscribers are notified with the message PM_i. The message can be in its original format or in a modified message that only contains the information required by subscriptions.

If no entry for K_i is found (a miss), PM_i is sent to the filtering engine. After searching in the cache is complete, the message PM_i is filtered with subscriptions in the subscription table to generate a matched result MQ that identifies the queries that are satisfied, and from MQ the identified subscriber references are obtained. For example, if $MQ = \{Q_1, Q_2\}$, queries Q_1 and Q_2 are found to be a match, and all the subscribers interested in Q_1 and Q_2 are obtained from a subscription table.

If MQ is not empty, the XML filtering and matching system updates the cache by storing the filtering results in the cache and performs a notifying operation to those subscribers who are interested in the message PM_i. If on the other hand MQ is empty, the message is ignored.

Algorithm 1: Complete message caching pseudo code at every broker

```
   input  : An XML message to be filtered
   output: A set of queries that matched the message
 1 receive a particular message PM_i;
 2 key ← hash(PM_i);
 3 isCached ← cache.lookup(key);
 4 if isCached then
 5 |   MQ ← cache.get(key) ;
 6 |   notify corresponding brokers or subscribers with PM_i;
 7 else
 8 |   forward PM_i to the filtering engine;
 9 |   MQ ← filter(PM_i);
10 |   if MQ is not empty then
11 |   |   write (key, MQ) to the cache;
12 |   |   notify corresponding brokers or subscribers with PM_i;
13 return
```

Algorithm 1 presents the sequence of steps for the complete message caching scheme which can be deployed at every broker. The complete caching scheme can be installed at every broker. In Algorithm 1, *key* is the hash code of a particular message PM_i; MQ represents the matched query set; and *isCached* is set to be true for a cache hit. *cache.lookup()* searches a specified item in the cache and *cache.get()* retrieves a specified item from the cache. In some cases, XML messages include some elements with changing values. For example, different sensors detect the same event or report unchanged events but with different timestamps. In order to apply the complete message caching scheme into these specific cases, these fields such as the *timestamps* element/attribute of an XML message can be selectively ignored in the process of parsing. This changing element/attribute is not related to information to be received. Hence, it does not need to be processed.

3.1.2 Structure-based message caching

The previous section describes a complete message caching scheme that generates a hash code for each message and searches the cache for a match, which indicates the same message has already been received. The next step is to investigate whether or not to deploy caching when the new message shares the same structure as a previous one (static parts), but has different "value" fields (dynamic parts). There are techniques for using XML structural summaries for database-style query processing and information retrieval-style search in the context of both centralized and distributed environments [20, 78]. Is it possible to exploit message structure information in stream-based XML filtering systems as XML messages arrive? This question motivates the exploration of an alternate caching method based on the structure of XML messages.

The reason for using structural caching is that for a cache hit the cost of structural selection can be saved and a structural query match set is provided from the cache, instead of through a filtering engine. Messages with the same structure can

be captured through message structural summaries, and share the same structure subscription match set. In the initial design, a cache post-processing step, which includes predicate checking and branching node verifying, is applied to the structure subscription match set. The task of branching node verification is to check if matching elements from different branches for a branching node of a twig subscription (query) are mapped to the same element in a message. However, this design complicates the predicate checking and branching node verification. To compute the predicate and the branching node of a query, the whole message has to be processed again. A complete cache post-processing algorithm for the structure-based caching scheme was implemented, but its performance was not always good. This shows that such a complicated post-processing is not applicable to a structure-based caching scheme.

In order to simplify the structure-based technique, this thesis proposes to separate subscription structure and predicate, and simply send rewritten subscription (structure only) across a network. The complete information of a subscription is stored at each local CE that contains a local filtering engine. For example, for a subscription "/book/section/figure[@height=10][title]/image", only the rewritten query "/book/section/figure[title]/image" is forwarded across the network. An incoming message needs to do predicate and branching node checking only at the filtering engine installed on the edge node. Using structure-based caching, the cost of structure path evaluations is not incurred in the case of a cache hit. Pushing the work of predicate checking and branching node verification to the edge node can reduce the filtering cost at intermediate brokers and achieve a scalable pub/sub system.

The process of generating a message structure summary is described. When a new message PM_i is received, it is sent to an XML parser. During parsing, all tag names are recorded and a message structure SM_i is constructed for message PM_i. Attribute names are ignored because they have no impact on structure matching. After the message PM_i is parsed, a hash code K_i, where $K_i = \text{hash}(SM_i)$, is generated as a message identifier code which is different from the message code generated by a

complete caching method consisting of tag names, attribute names and values.

At intermediate brokers, the XML filtering and matching system with cache uses K_i to locate a valid entry in the cache. If there is a cache hit, a structure subscription match set MQ is returned. If no entry for K_i is found (a miss), PM_i needs to be processed by the filtering engine. The message PM_i is filtered by the filtering engine to generate a result MQ and the system performs a notifying operation to identified brokers, if any. If MQ is not empty, the XML filtering and matching system adds the pair (K_i, MQ) to the cache and notifies the identified brokers. If the answer is empty, the message is ignored. Algorithm 2 presents the pseudo code for structure-based message caching that is incurred at PE and intermediate brokers. In Algorithm 2, SM_i refers to the structural summary of a particular message PM_i; K_i is the hash code of SM_i; MQ represents the matched query set; and *isCached* is set to be true for a cache hit. *cache.lookup()* searches a specified item in the cache and *cache.get()* retrieves a specified item from the cache.

Algorithm 2: Message structure caching pseudo code at PE and intermediate brokers

```
   input : An XML message to be filtered
   output: A set of queries (query predicates not considered) that matched the message
 1 SM_i ← generate structure summary for PM_i during parsing;
 2 K_i ← generateKey(SM_i);
 3 isCached ← cache.lookup(K_i);
 4 if isCached then
 5 |   MQ ← cache.get(K_i);
 6 |   notify identified brokers or subscribers with PM_i;
 7 else
 8 |   forwards PM_i to the filtering engine;
 9 |   MQ ← filter.getStructureMatchedQryID(PM_i);
10 |   if MQ is not empty then
11 |   |   write(K_i, MQ) to the cache;
12 |   |   notify identified brokers or subscribers with PM_i;
13 return
```

Applying a structure-based caching scheme to a filtering engine can reduce the average filtering time, but it may incur false positives. For example, some messages

can be transmitted to the network which are identified as unmatched messages at CEs. When the system checks the subscription match set MQ, if the receiver is a broker, the message is delivered to the corresponding broker. Otherwise the subscription (query) has been submitted from a local subscriber of a CE. In this case, a special flag is set and the message is content filtered including predicate checking to identify subscribers at the CE. This is accomplished by a local filtering engine at the edge node which contains a lower number of subscriptions in comparison to the number of subscriptions held by a filtering engine at the higher level brokers. If a match is found, the message is sent to the identified subscribers. Predicate checking is performed at the last broker.

3.1.3 System overhead

The structure-based caching scheme adds processing overhead to the edge brokers, the overhead requires further analysis for several reasons. First, if XML messages are large in terms of number of bytes, the system may become constrained by network bandwidth. Secondly and more significantly, brokers in the structure-based scheme are not just processing unmatched messages but also messages that are transmitted but are found as unmatched at a CE. This causes significant processing overhead for these messages, consuming resources on the brokers, which would further introduce additional queueing delays. Therefore, it is necessary to discuss the network and broker overhead of the structure-based scheme.

YF-C-Cache represents the complete message caching, YF-S-Cache represents the structure-based message caching, and YF represents no caching scheme. The number of nodes along a delivery path from a PE to a CE is K.

The computational cost for Yfilter and the YF-S-Cache schemes is analyzed as follows.

t_{YF} is the average filter processing time for YF. For a system that does not use caching, the end-to-end delay is $K \times t_{YF}$ because a stand-alone YF is installed at

every node.

For a system that uses YF-S-Cache scheme, PE and K-2 intermediate brokers enable YF-S-Cache and CE uses YF with no caching because the false positive messages should be filtered out at CE. t_{hit} is the processing time for a cache hit and t_{miss} the processing time for a cache miss and h is the hit ratio. The average processing time for YF-S-Cache depends on the cache hit ratio:

$$h \times t_{hit} + (1 - h) \times t_{miss} \tag{3.1}$$

The end-to-end delay from PE to the last intermediate broker before CE is:

$$(K - 1) \times (h \times t_{hit} + (1 - h) \times t_{miss}) \tag{3.2}$$

The end-to-end delay from PE to CE is the sum of the processing time taken by CE and Equation 3.2.

$$t_{YF} + (K - 1) \times (h \times t_{hit} + (1 - h) \times t_{miss}) \tag{3.3}$$

In a worst case scenario for the YF-S-Cache scheme, a new message is passed through a PE and $K - 2$ intermediate brokers with a cache miss at each broker, and does not match any subscriber at CE. The cost for the worst case is $(K - 1) \times t_{miss} + t_{YF}$. The best case for the YF-S-Cache scheme is the situation in which a cache hit on each intermediate broker. The cost for the best case is $(K - 1) \times t_{hit} + t_{YF}$. Typically, the average processing time for a cache hit is much smaller than that is incurred in a system that does not use caching. If the hit ratio is high, the overall processing time can be significantly reduced.

3.2 Automatic data aggregation

This section briefly presents an automatic data aggregation method that is used to collect real-life data traces. In most existing work on XML filtering, only synthetic

data sets are used for performance evaluation. The synthetic data are generated through real-world XML DTDs or schemas using an XML generator. Recently, authors of [46, 45, 64] have presented experiments with real Yahoo! financial data stored in a traditional format where data values are separated by commas or spaces. The method of retrieving real data used in this thesis is the same as that used in [46, 45, 64]. Furthermore, these real-life data values in the comma-separated values (CSV) file format are then converted to an XML format using XSLT [5].

The process for automatic aggregation is described next. It is a three-step process. Initially, the set of stocks to be monitored is determined. A perl script is executed to search company profiles by exploiting the regularity of html structure. First, an industry list is requested from the web page Industry Browser - Yahoo! Finance - Full Industry at `http://biz.yahoo.com/p/sum_conameu.html`. Second, for each industry in the industry list, an http *get* query for a company list is made. The query is sent by *libcurl* to download html files [8]. Third, after the html files are downloaded, company stock information is obtained by parsing the html code of the company list web page. Once this is done, a stock list for different companies is stored in a file called *stocklist.*

To produce a message of reasonable size, one hundred stock codes are randomly picked from the stocklist. Another perl script is executed to retrieve documents for stocks listed in *stocklist.*

The third step is to convert the data in normal text format into an XML format. XSLT is a style sheet for transforming structure and value information in XML. An open source XSLT 2.0 [5] processor Saxon [68] is exploited for this job. Each plain text file under the directory storing the collected financial data is transformed into an XML-based text file using an XSLT style sheet. Thus, a text file containing comma-separated data is transformed into an XML file. Once transformed, an XML message containing the data is ready for processing by the filtering engine.

3.3 Experimental setup and performance metrics

Two caching schemes are implemented in Java. All experiments are conducted on a system consisting of two 3.0 GHz Intel Pentium cores with 4.0 GB of RAM running under Windows XP. A thousand messages are used for the warm-up phase. Garbage collection is explicitly called before the measurement phase. Each experiment is repeated five times during the measurement phase and the average of the five measurements is calculated for the final result. The relative performance of the caching techniques and a system using no caching is expected to be similar on other systems.

3.3.1 Experimental setup for performance study of XML filtering

In the caching experiments, Yfilter [54] is used as the filtering engine because Yfilter is an open-source software system. Authors of [47] report that Gfilter outperfroms Yfilter. However, Gfilter is commercial software; the program is not available to the public. Yfilter has also been used by a number of other researchers as a benchmark for performance comparison (see [28, 47, 50]).

The messages are generated by ToXGene using three different real-life DTDs [28, 47, 18, 54]. The *book.dtd*, a recursive DTD, is used as a test XML schema in the works proposing Afilter [28] and Gfilter [47]. The *nitf.dtd*, developed by the International Press Telecommunications Council and representing the content and structure of news articles and accompanying metadata, is used in the research concerning Xfilter [18] and Yfilter [54]. The *bib.dtd* comes from an example of XQuery use case in [7]. The characteristics of messages used in experiments are shown in Table 3.1. Table 3.1 shows 6 sample test messages generated from the three DTDs described earlier. In the test *book* XML message, there is a recursion element *section*. For example, a *book* can have many *sections* and each *section* has several *subsections*. The recursion

depth of element *section* in an XML message is varied in order to test the impact of the recursive depth on system performance. The depth is varied from 1 to 6 based on the *book.dtd*. All element/attribute values of test messages are integers between 1 and 100.

Table 3.1: Characteristics of test messages

	Avg. length	Max. depth	Avg. path depth	No. paths	Size (KB)
bib.xml	769	3	3.0	576	16
nitf22.xml	165	7	6.2	99	9.2
book(r=1).xml	301	4	3.6	216	6.8
book(r=2).xml	324	5	4	230	7.5
book(r=3).xml	293	6	4.7	212	6.5
book(r=6).xml	285	9	7.3	205	6.9

To choose the parameter values used in our experiments, an important work [47] in the literature is referred. For example, the default values for the experimental parameters are: query depth=6, prob(//)=0.2, prob(*)=0.1, number of predicates=0, number of branches=0 and query population=20,000. The values of prob(//) is chosen from a set of {0.2, 0.3, 0.4} and the values of prob(*) is chosen from a set of {0.2, 0.3, 0.4}. The maximum recursive depth of the book XML messages, i.e., the self-nesting of *section* element, is increased from 2 to 4.

In our experiments, the XPath subscriptions are randomly generated using the Yfilter XPath generator [53]. The *nitf.dtd*, *book.dtd* and *bib.dtd* are used as XML schemas. *Bib.dtd* used in the experiment is from [4]. Table 3.2 shows the default settings of test queries. Each column in the table corresponds to a query feature. There is no duplicate query in the experiment. The parameters, such as number of subscriptions, probabilities of '//' and '*', are varied in the experiment for the *nitf* data set. For the other test message data set, the default setting of the test query set is used (see Table 3.2).

Table 3.2: Default setting of the test query set

query length	proba (*)	proba (//)	No. of predicates	No. of branches
6	0.1	0.2	0	0

3.3.2 Performance metrics

The following metrics are used in evaluating the performance of XML caching.

1. **Processing time for Yfilter(t_{YF}):** is the difference between the time a message leaves the filtering engine and the time of arrival of the message at the filtering engine.

2. **Average processing time (A_{YF}):** is the average processing time for a sequence of messages for a Yfilter.

3. **Hit processing time for the YF-C-Cache (t_{hit} for YF-C-Cache):** is the average processing time for a cache hit when a YF-C-Cache is used. The processing time for a cache hit with YF-C-Cache is the difference between the time of obtaining the matched subscriptions associated with the message and the arrival time for the message. The average is computed over multiple experimental runs.

4. **Miss processing time for the YF-C-Cache (t_{miss} for YF-C-Cache):** is the average processing time for a cache miss when a YF-C-Cache is used. The processing time for a cache miss with YF-C-Cache is the difference between the time of obtaining the matched subscriptions associated with the message and the arrival time for the message. The average is computed over multiple experimental runs.

5. **Hit processing time for the YF-S-Cache (t_{hit} for YF-S-Cache):** is the average processing time for a cache hit when a YF-S-Cache is used. The processing time for a cache hit with YF-S-Cache is the difference between the time

of obtaining the matched subscriptions associated with the message and the arrival time for the message. The average is computed over multiple experimental runs.

6. **Miss processing time for the YF-S-Cache (t_{miss} for YF-S-Cache):** is the average processing time for a cache miss when a YF-S-Cache is used. The processing time for a cache miss with YF-S-Cache is the difference between the time of obtaining the matched subscriptions associated with the message and the arrival time for the message. The average is computed over multiple experimental runs.

7. **Average end-to-end delay (*Average E2E delay*):** is the average processing time for a message on a system with multiple brokers, which represents the average time for delivering a message from a PE to a CE. *Average E2E delay* for a system that does not use caching is calculated by Equation 3.4. *Average E2E delay* for a system that uses YF-C-Cache is calculated by Equation 3.5. *Average E2E delay* for a system that uses YF-S-Cache is calculated by Equation 3.6.

$$Average\ E2E\ delay = K \times t_{YF} \tag{3.4}$$

$$Average\ E2E\ delay = K \times (t_{hit} \times h + t_{miss} \times (1 - h)) \tag{3.5}$$

$$Average\ E2E\ delay = t_{YF} + (K - 1) \times (t_{hit} \times h + t_{miss} \times (1 - h)) \tag{3.6}$$

where h is the cache hit ratio. When a YF-C-Cache is used, t_{hit} and t_{miss} are t_{hit} and t_{miss} for the YF-C-Cache respectively; when a YF-S-Cache is used, t_{hit} and t_{miss} are t_{hit} and t_{miss} for the YF-S-Cache respectively. K is the number

of nodes along a delivery path from PE to CE. YF-C-Cache is installed at K nodes. YF-S-Cache is installed at $K-1$ nodes and YF is installed at the CE. When caching is used, the average processing time at each intermediate node is $t_{hit} \times h + t_{miss} \times (1-h)$. The right hand side of Equation 3.6 is the sum of the processing time at the CE and the sum of the processing times at intermediate nodes. In addition, with YF-S-Cache, queries stored at brokers except CE do not have predicates. Therefore, the processing time for matching a message against queries without predicates is less than that for queries with predicates. With YF and YF-C-Cache, queries stored at every broker can have predicates.

3.4 Experiments with synthetic data

The two caching schemes described in Section 3.1.1 and Section 3.1.2 are implemented in Java. Experiments are run to evaluate the performance of the different caching methods, and the performance results are presented. The experimental configurations and the workload that includes both XPath subscriptions and XML messages are introduced.

The experiments are run by first reading the message into memory and then sending it to the system. The processing times of a message for two caching schemes are measured and compared with the case when no cache is used. The time reported includes the message parsing time and message filtering time. Where a cache is used, time for performing additional caching operations, such as time to cache the entry in the case of a cache miss and time to retrieve cached data in the case of a cache hit, are included.

This section focuses on comparing the performance of the two caching methods with the Yfilter (YF) that does not use any caching. Three groups of experiments are presented. The first group of experiments examines the impact of different probabilities of the operators '//' and '*' on performance. The second group examines the effect of different recursion depths for an XML message on average processing time

of a message. The third is designed to check the impact of increasing the number of elements contained in a message on performance. Yfilter runs in a regular mode and no optimization is used for a general subscription. The optimization mode of Yfilter is a special case for subscriptions with no predicate and no branch. It is a short-cut evaluation strategy to find matched queries as long as the first match is found.

3.4.1 Varying probability of // and * (*nitf* data set)

The message used in this experiment is *nitf* message which does not have a recursive structure. Fifty thousand subscriptions are generated, and the probability of '//' and '*' is varied to examine how it affects the processing time of the three different schemes: YF, YF-C-Cache and YF-S-Cache. Table 3.3 lists the results. The processing time of a cache miss includes the time for message parsing, cache key generation, cache looking up, content-filtering by a filter engine and creating a cache entry. As seen in Table 3.3, the cost for generating hash keys for the YF-C-Cache is very small. The cost for generating caching keys for the YF-S-Cache includes generating the message structure during the message parsing phase. t_{hit} for both caching schemes includes times for generating caching key, looking up in the cache, and putting returned caching values into the result. With the YF-C-Cache, when there is a hit, the processing time is between 3.07 ms and 6.8 ms. With the YF-S-Cache, when there is a hit, the processing time is between 5.0 ms and 8.2 ms. For any given probability of (//,*), the t_{hit} for both YF-C-Cache and YF-S-Cache is significantly better than the processing time for YF. t_{miss} for YF-C-Cache and t_{miss} for YF-S-Cache are comparable to the processing time for YF.

3.4.2 Varying the recursion depth (*book* data set)

The *book* messages *book (r=1).xml*, *book (r=2).xml*, *book (r=3).xml*, and *book (r=6).xml*, generated from a recursive DTD, are considered. Book messages have a recursive structure and the recursion depth is varied from 1 to 6. The maximum number of subscriptions generated is 910 because the DTD is small. Table 3.4 shows that the

Table 3.3: The effect of probability of // and * (*nitf* data set) on the processing time (in ms)

(//,*)	YF	YF-C-cache		YF-S-Cache	
		t_{miss}	t_{hit}	t_{miss}	t_{hit}
0.2,0.1	35.9	37.95	3.07	37.1	5.0
0.3,0.1	42.2	40.2	5.6	39.6	8.2
0.4,0.1	43.8	46.1	3.6	43.9	5.4
0.2,0.2	40.6	43.7	3.5	40.9	5.4
0.2,0.3	43.8	42.2	6.8	43.0	5.7
0.2,0.4	43.8	44.8	6.5	44.2	6.3

cost for YF increases when the recursion depth increases but the increment is not significant. The theoretical analysis in [28] indicates that in the worst-case for the number of matched transitions from a node would be an exponential function of the recursion depth of a message and subscriptions. However, the trend represented in Table 3.4 is not exponential because of the structures of the generated queries. Some queries do not have recursive paths. As the recursion depth r increases, the processing time for each scheme increases. This is because the ancestor/descendant operators (//) need more comparison times and all message nodes of a subtree have to be compared. For any given r, t_{hit} for YF-C-Cache and that for YF-S-Cache are lower than the processing time for YF. t_{hit} for YF-S-Cache is higher than that for YF-C-Cache because of the differences in generating the message hash code by these two caching schemes. YF-C-Cache directly uses a hash function to generate message hash code; YF-S-Cache parses the XML message to obtain the message structures first, and then uses a hash function on the message structure to generate the hash code. t_{miss} for two caching schemes are comparable to the processing time for YF.

3.4.3 Varying number of elements (*bib* data set)

The *bib* data set is considered next. The *bib.dtd* is very small and the maximum number of subscriptions generated is 747, with each subscription being distinct from the others. A *bib* XML message contains many *books*. In order to examine the impact

Table 3.4: The effect of recursion depth (*book* data set) on the processing time (in ms)

Recursion depth	YF	YF-C-cache		YF-S-Cache	
		t_{miss}	t_{hit}	t_{miss}	t_{hit}
r=1	9.16	9.12	0.15	9.4	2.35
r=2	10.99	10.79	0.22	11.09	2.49
r=3	11.39	10.88	0.2	10.90	2.28
r=6	11.76	11.89	0.19	12.22	2.82

of the number of elements in a message, the number of *books* in the test *bib* message is varied. The results are shown in Table 3.5, in which the first column indicates how many *book* elements are contained in a *bib* message. The processing time for YF increases as the number of elements increases. As before, for any given *number of book elements* contained in the *bib* messages, t_{miss} for YF-C-Cache and YF-S-Cache are comparable to the processing time for YF, while t_{hit} for YF-C-Cache and YF-S-Cache are much lower than that for YF. The relative performance of YF-S-Cache, YF-C-Cache and YF, for a given number of elements, is expected to be the same for other types of messages.

Table 3.5: Varying number of elements (*bib* data set) on the processing time (in ms)

# of **books** (x)	YF	YF-C-cache		YF-S-Cache	
		t_{miss}	t_{hit}	t_{miss}	t_{hit}
1	1.21	1.08	0.02	1.15	0.84
3	1.36	1.34	0.026	1.15	0.86
5	1.76	1.70	0.026	1.44	1.01
10	2.1	1.99	0.025	2.09	1.28
15	2.57	2.61	0.024	2.36	1.24
20	3.08	3.06	0.025	2.83	1.33
50	6.40	6.39	0.026	6.57	1.94
100	11.59	11.55	0.026	11.88	2.97
200	22.83	22.81	0.026	23.00	4.58
300	37.63	36.06	0.028	37.35	6.35

3.4.4 End-to-end delay

In this section, the end-to-end (E2E) delay from a PE node to a CE using the message processing time obtained from previous experiments is computed. The E2E delays for the three schemes, YF, YF-C-Cache and YF-S-Cache, are compared. A new experiment is designed to test subscriptions with predicates.

Table 3.6 lists the time taken by YF when a single predicate is present in a subscription. The level of a predicate is in the range of 1 to 6 and is uniformly distributed. The range of predicate values for each level is between 1 and 20 and are uniformly distributed. A wildcard ('*') can match the name of any tag. When the probability of a wildcard increases in a query, more elements might be selected as matching instances.

The performance of a YF (no caching) is discussed first (see Table 3.6). An increasing trend in average processing time is observed in Table 3.6a as recursion depth of a message increases. There are differences between the YF processing time for queries with predicates for bib messages as shown in Table 3.6b and that for queries without predicates in Table 3.5, especially when x is larger than 50.

Similarly, for nitf messages, the YF processing time for queries with predicates is observed to be higher than that for queries without predicates (see Table 3.3 and Table 3.6c). In Table 3.6c, when the $proba(*)$ increases from 0.2 to 0.4, the YF processing time increases because *-operator makes more elements be selected for predicate checking. For $proba(//)$, the YF processing time is observed to be increased but not significantly because the $nitf.xml$ message has no recursive structure and cannot trigger //-transitions on the NFA. A comparison of Table 3.6c and Table 3.3 indicates that the processing time taken by predicate checking is a large component of the processing time for XML filtering. The computation cost for obtaining a matched result increases when branching node verification is incurred for a twig subscription. It is reasonable to decouple the subscription structure and predicate, and push predicate checking and branching node verifying to the edge node.

Table 3.6: The processing time for one predicate (Yfilter): (a) results for *book* data set; (b) results for *bib* data set; (c) results for *nitf* data set.

(a)

Recursion depth	r=1	r=2	r=3	r=6
Average processing time (ms)	33.17	38.36	38.19	48.08

(b)

x	1	3	5	10	15	20	50	100	200	300
Average processing time (ms)	1.21	1.38	1.88	2.69	3.55	4.21	9.61	17.39	35.47	55.56

(c)

(//,*)	(0.2,0.1)	(0.3,0.1)	(0.4,0.1)	(0.2,0.2)	(0.2,0.3)	(0.2,0.4)
Average processing time (ms)	93.8	106.4	115.6	128.0	159.4	200.0

Table 3.7: The processing time for one predicate using YF-C-Cache: (a) results for *book* data set; (b) results for *bib* data set; (c) results for *nitf* data set.

(a)

Recursion depth	Avg. processing time (ms)	
	t_{miss}	t_{hit}
r=1	33.99	0.09
r=2	38.75	0.1
r=3	38.33	0.09
r=6	49.93	0.07

(b)

x	Avg. processing time (ms)	
	t_{miss}	t_{hit}
1	1.14	0.017
3	1.49	0.017
5	1.90	0.017
10	2.56	0.018
15	3.52	0.019
20	3.97	0.019
50	10.13	0.02
100	17.61	0.02
200	34.95	0.02
300	55.27	0.02

(c)

(//,*)	Avg. processing time (ms)	
	t_{miss}	t_{hit}
(0.2,0.1)	91.67	0.28
(0.3,0.1)	104.5	0.3
(0.4,0.1)	112.6	0.31
(0.2,0.2)	126.1	0.31
(0.2,0.3)	159.6	0.34
(0.2,0.4)	200.5	0.38

In the experiments for analyzing the E2E delay, the size of the overlay network K is set to be 5 nodes and 10 nodes. Equation 3.4, Equation 3.5, and Equation 3.6 are used to compute the average E2E delay in the experiments. The YF processing times in Table 3.6, the YF-C-Cache processing times in Table 3.7 and the YF-S-Cache processing times in Table 3.3, Table 3.4 and Table 3.5 are used to compute the end-to-end delay.

The hit ratio, h, is varied, and Figure 3.3 displays the E2E delay for the different hit ratios and the nitf message. The query workload "Query(0.2,0.4)" in the labels of Figure 3.3a and Figure 3.3b implies that parameters for the generated queries, $proba(//) = 20\%$ and $proba(*) = 40\%$. For the YF-S-Cache, the *nitf* data set, a hit ratio of 50% and a query workload characteristic by prob(//) = 20% and prob(*) = 40%, the average E2E delay is 301 ms when K is 5; on the other hand, the average E2E delay is 427.25 ms when K is 10. For the YF, with the same workload, the average E2E delay is 1000 ms when K is 5, and the average E2E delay is 2000 ms when K is 10. The improvements achieved with YF-S-Cache are 69.9% and 78.6% for $K = 5$ and $K = 10$ respectively. Clearly, as the number of intermediate nodes between the PE and the CE increases, the performance improvement is observed to increase. For *nitf* messages, results for the different hit ratios and the different query workloads are shown in Figure A.1 and Figure A.2 in Appendix A when K is 5; in Figure B.1 and Figure B.2 in Appendix B when K is 10. Both the figures capture a similar trend in the variation of average E2E delay. For low hit ratios, the performance of YF-S-Cache is superior, whereas for higher hit ratios, YF-C-Cache demonstrates a better performance.

Figure 3.4 presents the results for *bib* messages. YF-C-Cache and YF-S-Cache demonstrate better performance than YF irrespective of the value of h when $h > 0$. The relative performance of YF-C-Cache and YF-S-Cache as presented in Figure 3.4 captured for *bib-x.xml* messages depends on the hit ratio. "*bib-x.xml*" indicates that there are x *book* elements in a *bib-x* message (see Figure 3.4). For lower hit ratios,

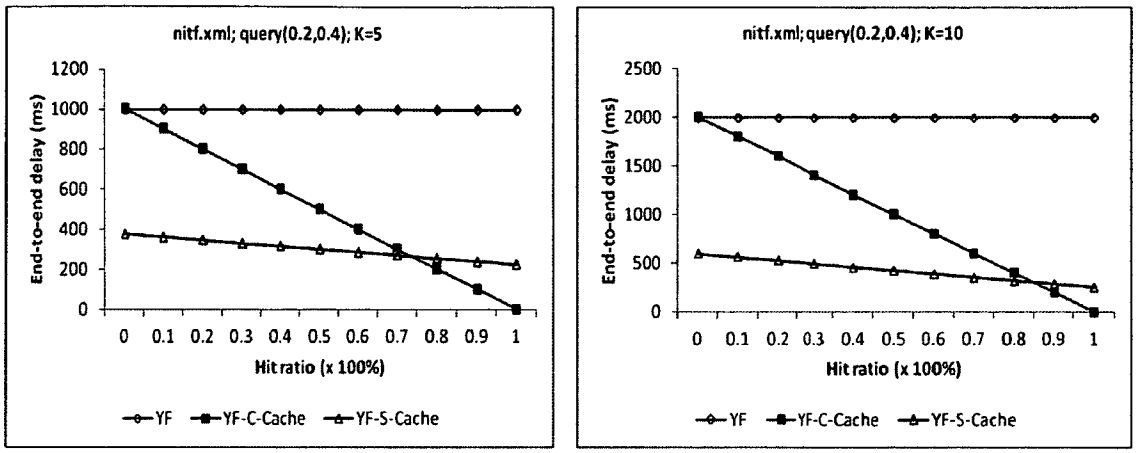


Figure 3.3: Average E2E delay results for *nitf* messages: (a) prob(//)=20% and prob(*)=40% and K=5; (b) prob(//)=20% and prob(*)=40% and K=10.

the performance of YF-S-Cache is superior, whereas for higher hit ratios, YF-C-Cache demonstrates a better performance. This is because of the relationship between caching cost and the message processing time of YF.

The relative performance of YF-S-Cache and YF are investigated for other *bib* messages. Results for *bib-3.xml*, *bib-5.xml*, *bib-10.xml*, *bib-15.xml*, *bib-50.xml*, *bib-200.xml*, show a trend similar to the one displayed in Figure 3.4 and are presented in Figure A.4 and Figure A.5 in Appendix A, Figure B.4, Figure B.5 and Figure B.6 in Appendix B.

Similarly, results for other *book* messages, including *book(r=1).xml*, *book(r=2).xml*, *book(r=3).xml*, *book(r=6).xml*, are presented in Figure A.3 in Appendix A and Figure B.3 in Appendix B. From the results for *book* messages, we can find that for low hit ratios, the performance of YF-S-Cache is superior to that of YF-C-Cache, whereas for higher hit ratios, YF-C-Cache demonstrates a better performance.

3.4.5 Discussion

This section highlights the impact of different parameters on performance based on the simulation results and observations.

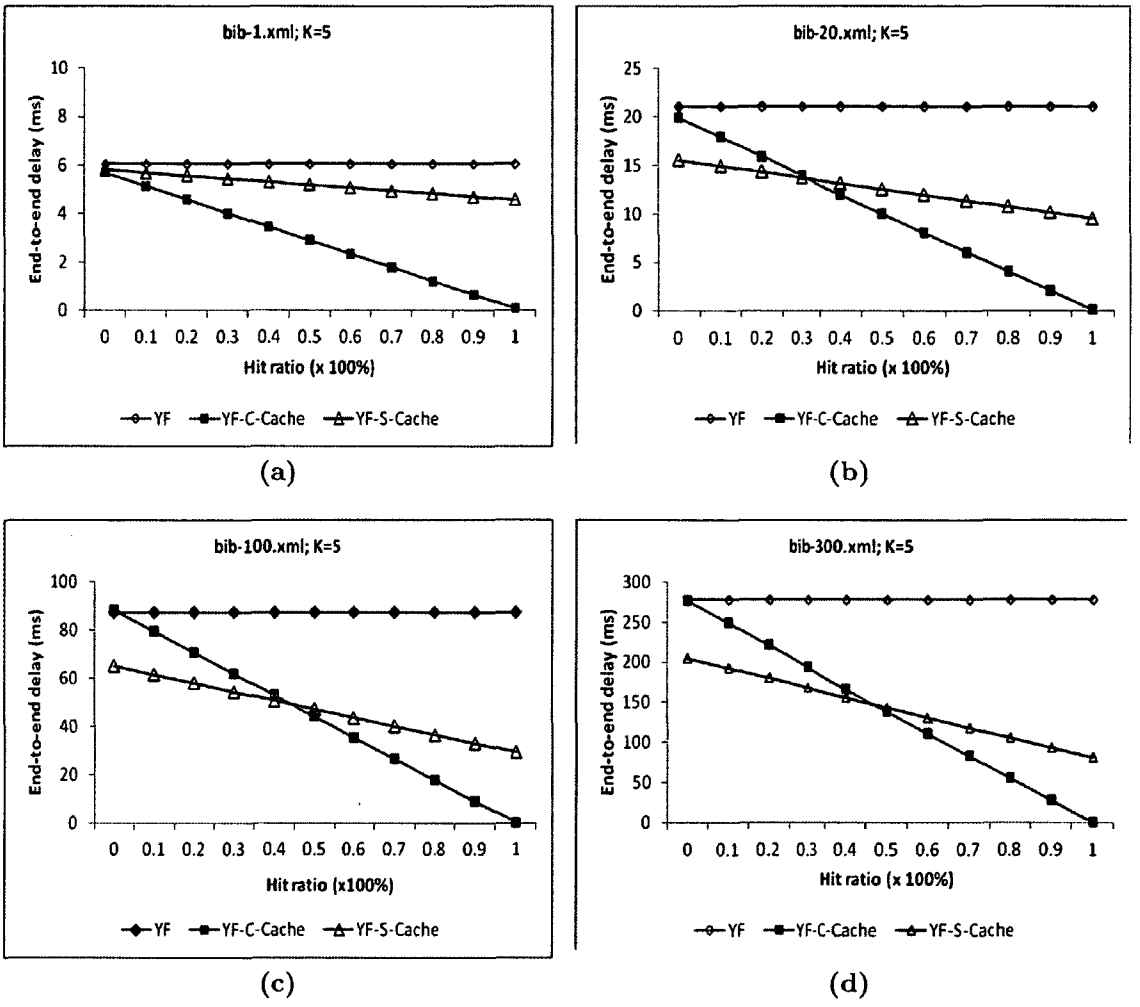


Figure 3.4: E2E delay results for *bib* messages for K=5: (a) bib-1.xml; (b) bib-20.xml; (c) bib-100.xml; (d) bib-300.xml.

The effect of *prob*(//) and *prob*(∗) on post-processing. The parameter *prob('//')* has impact on the message processing time for non-recursive messages when only one predicate is present on the subscriptions (see Table 3.6c). The parameter *prob('*')* has a significant impact on performance because for a higher value of this parameter more elements are selected for post processing as shown in Table 3.6c.

DTDs. The graphs for YF, YF-C-Cache and YF-S-Cache show different performance results with respect to different DTDs. For the *nitf.dtd*, there are 50,000 subscriptions generated for the experiments. The *nitf.dtd* is a complex DTD used in the news industry. For the *book.dtd* and *bib.dtd*, 910 and 747 subscriptions are generated. These are the maximum number of queries that can be generated by the tool for the respective DTDs. The processing times for the *book* messages and *bib* messages are small in comparison to that achieved with the complex *nitf* data set. The processing times for the two caching schemes are always lower than that of YF for messages generated from these three different DTDs, except when the hit ratio =0 for YF-C-Cache, in which case the E2E delay for YF-C-Cache that incurs caching overhead is slightly higher than that of YF.

Number of elements in a message. The *bib* message tree is flat, which contains many *book* elements. As expected, with an increase in the number of books in a message, the processing time increases for all the three schemes as demonstrated in Table 3.5.

Number of nodes, K. As the number of overlay nodes in a network increases, the benefits of the two caching schemes increase because of additional filtering performed at the intermediate nodes on a YF system.

3.5 Experiments with the real-life data traces

There is a lack of recognized published XML data traces from real pub/sub systems. Real-life data traces for weather data and stock data have been collected from websites and used to generate XML messages.

Figure 3.5: Screen shot of an example web page of Ottawa weather of the weather underground web site

The Weather Underground web site (`http://www.wunderground.com`) is a web site for meteorological information. The weather information of three cities Toronto, Ottawa, and Sudbury in Ontario are collected. The collected data include temperature, temperature feel like, current condition, humidity, dew point, wind, wind gust, pressure, visibility, uv, clouds, rainfall for August 3, August 4, and August 5 of 2011 starting from 0:00 AM and ending at 11:59 PM on each day. A web page capturing the weather data for Ottawa is presented in Figure 3.5. The stock data traces are collected for three days (August 3, August 4, and August 5) of 2011 from 9:30 AM to 16:00 PM on each day from Yahoo! Finance. The data collected include Stock Symbol, Last Trade, Pre Close, Open, Bid, Ask, Target Est, Avg. Vol, Market Cap, Price-to-earnings Ratio (P/E), EPS, Div & Yield, Rating. There are 79 messages in the data trace for each day. The web site is polled every 5 minutes and data on 100 stock symbols are collected.

The weather subscriptions contain 10,000 distinct queries. The query features are: the query length = 4, proba(*) = 20%, proba(//) = 20%, proba([ ]) = 5%, and proba(branches) = 30%. The XML messages are sent in the order of the message creation time and the time interval between two consecutive messages is equal to that between the two message timestamps.

Figure 3.6 depicts the network topology used for the caching experiments. Typically, a pub/sub system consists of PEs, CEs and intermediate brokers. In our experiments, three machines with the same configurations are connected by a switch to form a local area network and the network speed is 100 Mbit/sec. Each machine comprises two 3.0 GHz Intel Pentium cores with 4.0 GB of RAM running under Windows XP. An XSLT pre-processor transforms stock and weather data stored in a plain text into XML format and passes the generated XML messages to an XML filtering engine which is used to filter the messages. The system architecture that does not use caching is shown in Figure 3.6a; YF-C-Cache is used at every node as shown in Figure 3.6b; YF-S-Cache is used at PE and intermediate brokers, while a YF is used

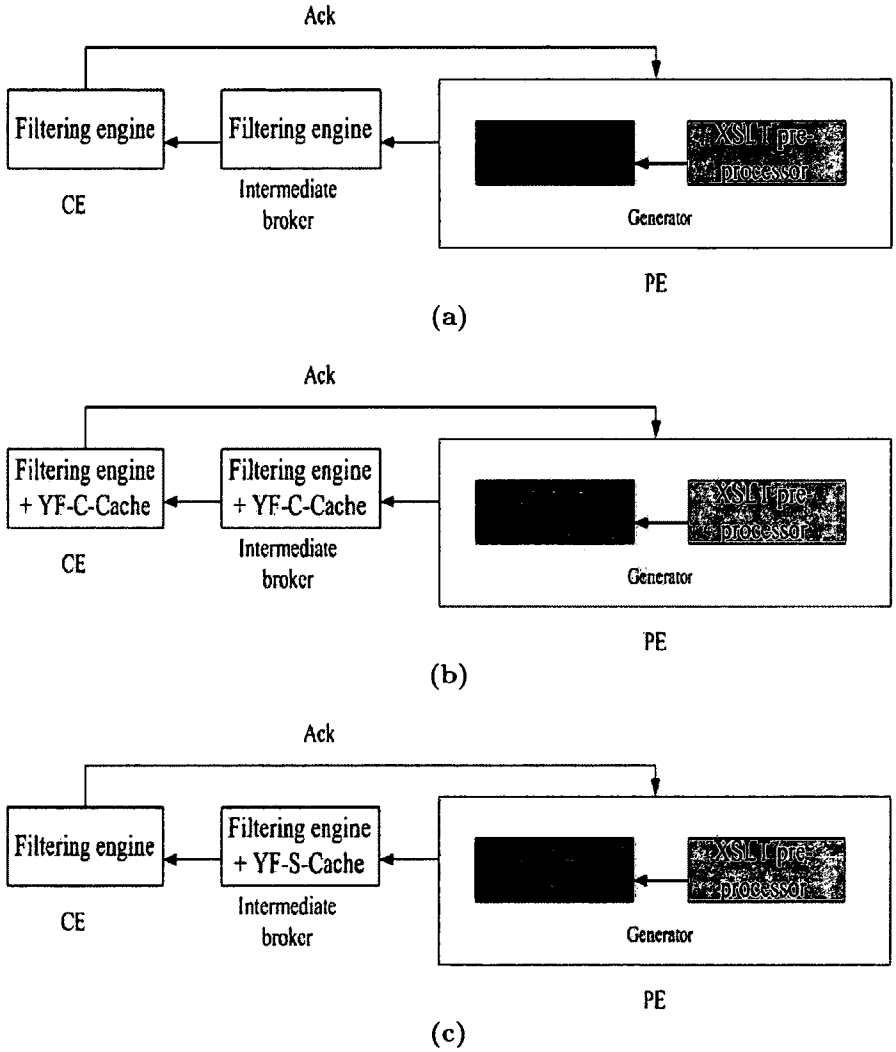


Figure 3.6: Experimental architecture for caching real-life data traces

at the CE (see Figure 3.6c).

Table 3.8: End-to-end delay results for weather data using different caching methods

		August-3-2011	August-4-2011	August-5-2011
No caching (YF)	Average E2E delay	92.07 ms	93.11 ms	93.47 ms
YF-C-Cache	Hit ratio	50.92%	47.90%	84.74%
	Average E2E delay	56.21 ms	65.71 ms	29.92 ms
YF-S-Cache	Hit ratio	100%	100%	100%
	Average E2E delay	46.05 ms	45.70 ms	46.68 ms
Total number of messages		3001	3000	2817

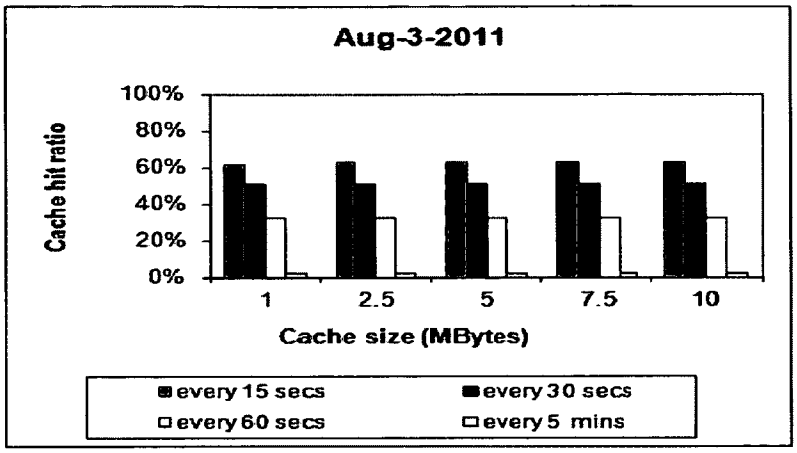
The end-to-end delay results for weather data are shown in Table 3.8 and the average data publishing frequency is once every 30 seconds approximately. The messages are sent at a rate of 1 message/sec in the experiments. The size of weather XML messages is around 1580 bytes. The hit ratio for YF-S-Cache is 100% because all weather messages have the same structure. For different days, different hit ratios are achieved with the YF-C-Cache. For example, the hit ratio is 50.92% for the data of August-3-2011, whereas the hit ratio is 84.736% for the data of August-5-2011. Results for the weather trace indicate that with a high hit ratio, caching can significantly improve the performance of XML pub/sub systems.

The minimum hit ratios required for the end-to-end delay using a YF-C-Cache to outperform no caching are 7.879%, 4.386% and 6.70% for August-3, August-4 and August-5, respectively. The minimum hit ratio required for YF-C-Cache to outperform YF is the minimum value of h that satisfy:

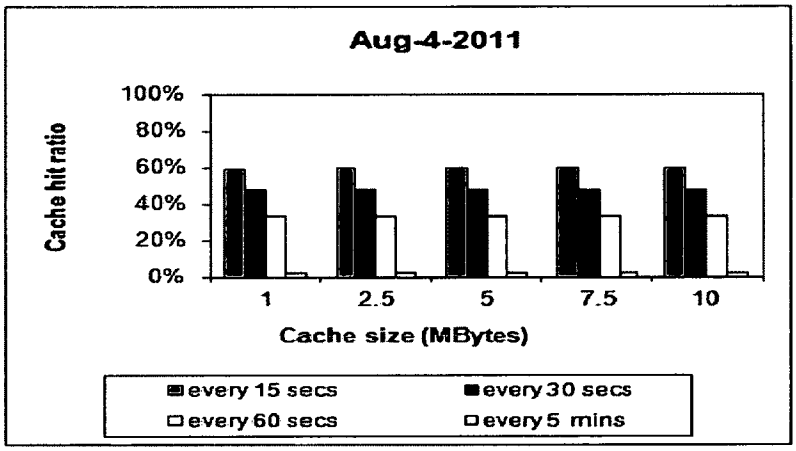
$$t_{YF} \geq t_{hit} \times h + t_{miss} \times (1 - h) \tag{3.7}$$

As discussed earlier, t_{YF} represents the average processing time for no caching and t_{hit}; t_{miss} are the average processing time for a cache hit and the average processing time for a cache miss, respectively. Note that the expression on the right hand side of the inequality is the average processing time for the YF-C-Cache.

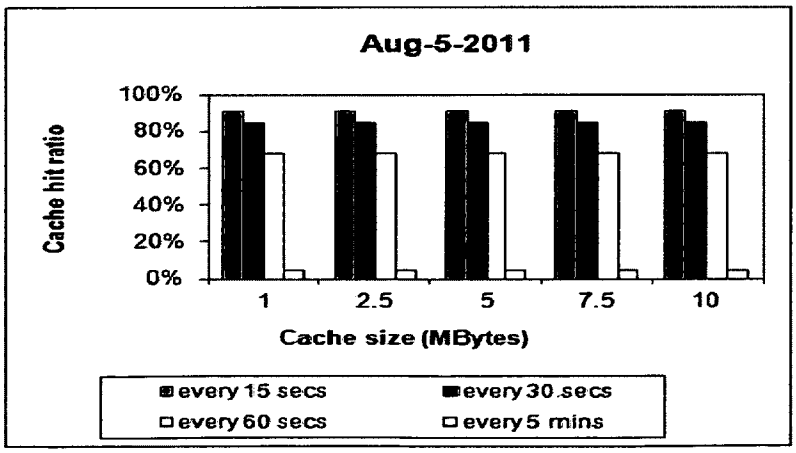
Results shown in Table 3.8 correspond to an "infinite cache" which does not have



(a)



(b)



(c)

Figure 3.7: Cache performance for weather data with different publishing frequencies: (a) August-3-2011; (b) August-4-2011; (c) August-5-2011.

any space limitation. The performance of the YF-C-Cache method is investigated for various cache sizes. The replacement policy is least-recently-usage (LRU) policy which is a common policy in the caching literature [94]. Whenever the cache is full and a new entry is to be added, the least recently used entry is replaced. The published messages consist of stock and weather messages. Figure 3.7 and Figure 3.8 depict the results of weather data, while Figure 3.10 presents the results of messages stream containing both stock and weather data. The stream is mixed and the messages in the stream are ordered based on their timestamps.

The purpose of Figure 3.7 is to examine the caching performance when data is published at different frequencies. Only weather data is considered. The frequencies used are once every 15 seconds, once every 30 seconds, once every 60 seconds, and once every 5 minutes. From the results, we observe that for any cache size the cache hit ratio drops as the publishing frequency is decreased.

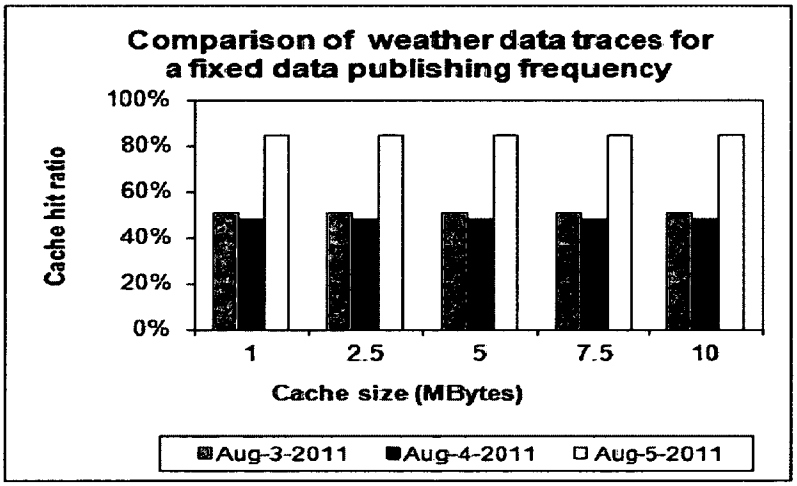


Figure 3.8: Comparison of caching performance for different days when data publishing frequency is fixed at once every 30 seconds

Figure 3.8 displays the caching performance when weather data set is used. The data publishing frequency is fixed at once every 30 seconds, while the weather data shown in Figure 3.7 are published at four different frequencies. From the results, it seems that the cache hit ratio for the data of August-3 and August-4 are close to each

other, while the cache hit ratio for the data of August-5 is much higher, indicating less variation in the weather data for August-5.

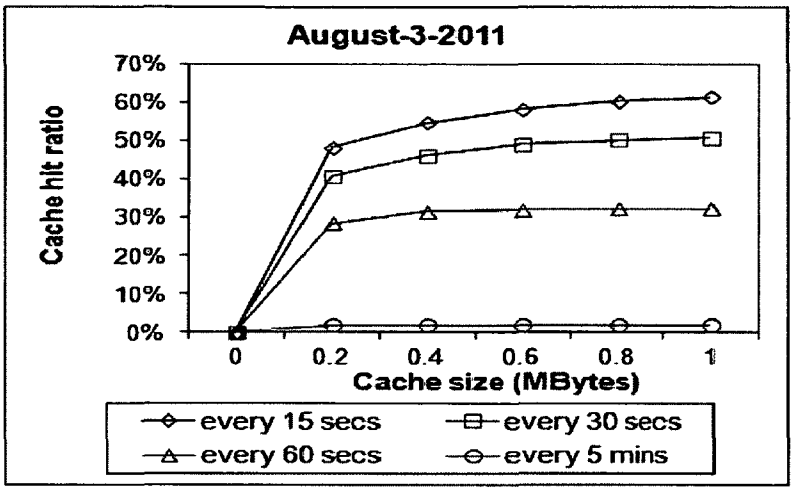
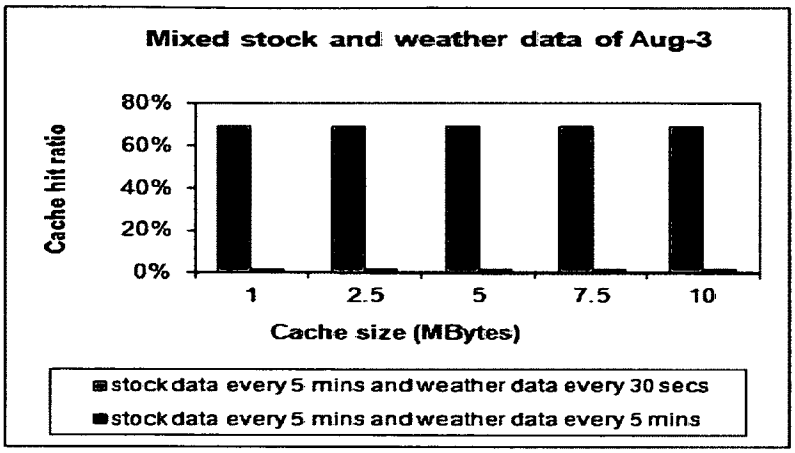


Figure 3.9: Cache hit ratios for weather data when cache size is varied

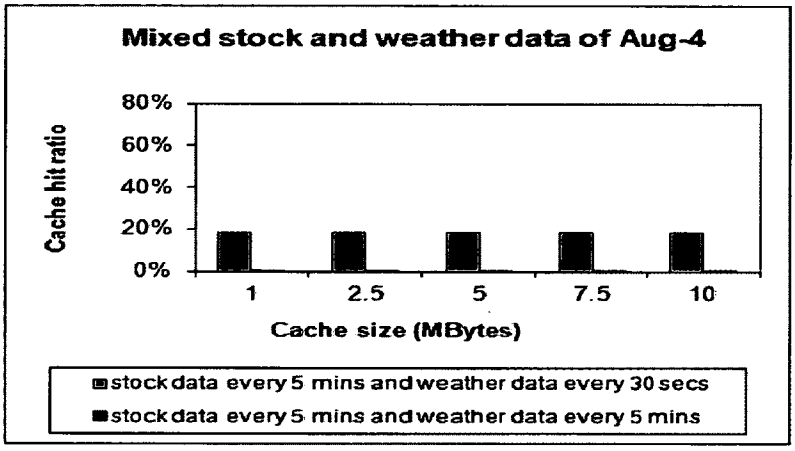
As expected, the cache hit ratio increases when the available cache size is varied from 0 Mbytes, 0.2 MBytes, 0.4 MBytes, 0.6 MBytes, 0.8 MBytes and 1 MBytes for weather data of August-3-2011 (see Figure 3.9). Caching seems to be more effective for higher data publishing frequencies, such as once every 15 seconds and once every 30 seconds.

Figure 3.10 displays the performance when data from multiple data sources are transmitted as a single message stream. We collect stock and weather data in parallel and record the data creation time. The starting time of the first message is 9:30 am and the time of the last published message is at 4:00 pm. The rationale behind choosing the time window is that we want to investigate the stock data when the stock market is open. If the data after 4:00 pm are used, the cache hit ratio is observed to be very high because of no variance in the data value.

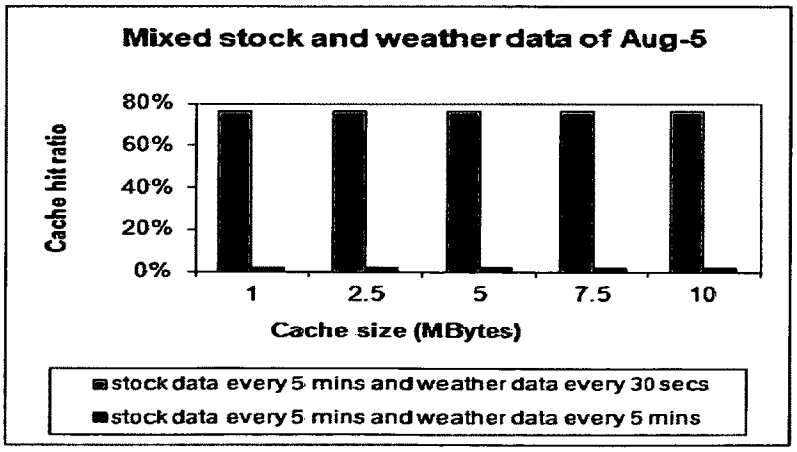
From the results presented in Figure 3.10, the cache hit ratio when data publishing frequency is once every 5 minutes is much lower than the hit ratio when the data publishing frequency is once every 30 seconds. This indicates that caching is effective for higher publishing frequencies. Data collected on different dates give rise



(a)



(b)



(c)

Figure 3.10: Cache performance for mixing stock and weather data: (a) for data collected on August-3-2011; (b) for data collected on August-4-2011; (c) for data collected on August-5-2011.

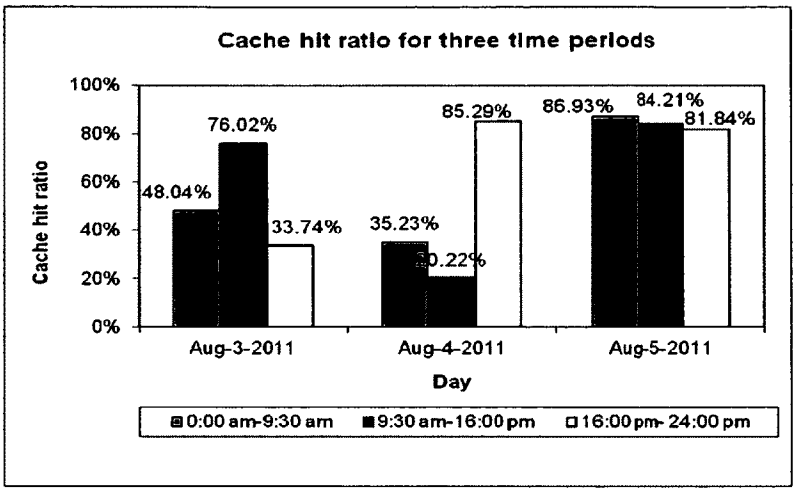


Figure 3.11: Cache hit ratios for different time periods for weather data

to different system performance. For example, for a given cache size, the cache hit ratio for the data of August 4 is much lower than the ratio for the data of August 3 and August 5.

Figure 3.11 depicts the hit ratios of weather data trace in three different time periods for different days. The hit ratio seems to depend on the day as well as the time of the day. For example, for the time period starting from 9:30 am and ending at 16:00 pm, the hit ratios for weather data of August 3 and August 4 are 76.019% and 20.22%. The hit ratio is observed to change with the time of the day on August 3 and August 4. Less variability in hit ratios is observed to occur for the different time periods on August 5.

3.6 Summary

Caching is effective in reducing the processing time in pub/sub systems. Two caching techniques YF-C-Cache and YF-S-Cache were proposed. The performance of these techniques were compared with the performance of the conventional YF (no caching) system. Performance analysis based on synthetic data shows that YF-C-Cache and YF-S-Cache demonstrate better performance than YF. From the end-to-end delay point of view, for low hit ratios, the performance of YF-S-Cache is superior, whereas

for higher hit ratios, YF-C-Cache demonstrates a better performance. Two real-life data traces were experimented with. YF-C-Cache seems to be effective for the weather data trace for which the data does not change frequently. YF-S-Cache shows good performance even if data changes frequently but the structure of the data stays the same.

Chapter 4

XPath query aggregation for XML routing

Existing approaches for XML routing are based on query aggregation. Such approaches match published messages to query aggregates in order to identify the queries that match a certain published message. Hence, XPath query aggregation is an important topic in XML-based pub/sub systems. However, existing XML aggregation algorithms, as presented in Section 4.4 for related works, are inefficient for handling ancestor/descendant operators (//). This chapter presents two new XPath query aggregation algorithms that are adapted from region codes [27]. One algorithm is used to check whether the new arriving query covers part of the existing query tree, while the other does the opposite, i.e., it is used to check whether the new arriving query is covered by the existing query tree. These two new aggregation algorithms, based on the evaluations, are more efficient than the existing approaches.

This chapter starts with an introduction to XPath aggregation, including a description of the limitations of the existing aggregation algorithms. Following that, Section 4.2 presents the new XPath query aggregation algorithms. Experimental results are described and explained in Section 4.3.

4.1 Introduction

XML pub/sub systems have given rise to application-layer XML routers, such as Sarvega XML Context Router [26]. An XML router is a complement to the IP router that carries the stream of data traffic across the Internet. XML-based network traffic is expected to continue to grow. However, the main problem of the state-of-the-art XML routing schemes is that the cost of filtering is expensive at each hop. A large number of queries have to be filtered against constantly arriving XML documents. Often, the filtering speed cannot match the XML document arrival speed on a typical pub/sub system. The XML filtering speed is proportional to the number of queries to be filtered. Typically, the queries are expressed as XPath queries. The most widely studied operators of an XPath query include parent/child operator (/), ancestor/descendant operator (//), wildcard operator (*), and predicates ([ ]) (see Section 2.2 for more explanation over the operators). An effective way to improve system performance is to organize various XPath queries stored in an XML router in a systematic and effective manner. XPath query aggregation is a crucial technique for XPath query routing, as it is a technique for checking the inclusion relationship between queries.

Query aggregation is a technique for combining individual queries that are used to fetch subscribed data by multiple subscribers into a single query in the context of a pub/sub system.

Existing XPath query aggregation algorithms traverse the query tree node by node to compare any new query with existing queries in the query tree. This is not efficient for the ancestor/descendant operators (//) because all the descendant nodes under the current node need to be compared. An example in Figure 4.1 depicts a scenario for which the tree traversal algorithm is inefficient. In order to find a match for node *//b* in the sample new query /a//b, six comparisons are required, namely x, m, n, o, p, and b.

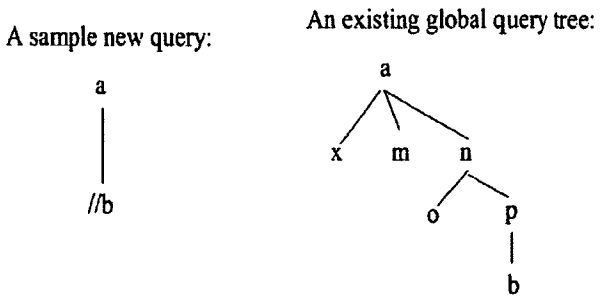


Figure 4.1: A problem with existing aggregation algorithms which has to compare each node pair in both trees

The algorithm proposed in [107] shares common prefixes between query branches. But it does not represent branch node positions in its query index tree. Therefore, there is a post-processing operation to determine the final aggregated results of a new tree-structured query. The algorithm of [71] splits a twig query into paths/branches and performs containment check for paths/branches, even though the branch structure information is lost.

The XSearch algorithm [42] is chosen for performance comparison in this thesis because it is well-known and more efficient compared to other approaches. Comparison of system performance achieved with the algorithm proposed in this thesis with that achieved with the XSearch is presented in Section 4.3. The XSearch algorithm shares common prefixes with different XPath queries and treats a twig query as a unit without a branch split. There is no post-processing operation; however, the XSearch algorithm maps a //-node to two paths. One path is an empty chain of nodes, and the other path is a non-empty chain. If the number of //-nodes is large, the number of comparisons is proportional to $O(|s| \times |T(R)|)$, where $|s|$ is the number of nodes in the new query to be aggregated and $|T(R)|$ is the number of nodes in the factorization tree [42].

The TwigStack algorithm [27] uses a holistic approach for matching twig queries with the publisher's *XML documents* stored in a database using a region encoding scheme to represent each node position ([*left*: *right*], *level*) within an XML document.

The TwigStack algorithm treats an entire XPath twig query as one unit instead of multiple units, one for each branch. It is an optimal algorithm for computing ancestor/descendant operators present in an XPath query [27]. This prompts us to encode nodes in a subscriber's *query* index tree first, then compute the aggregated results for container and containee based on region encoding representations, instead of navigation on a tree.

The purpose of this chapter is to present a new XPath query aggregation approach using the region encoding scheme. The new aggregation approach consists of two new algorithms. The region encoding scheme assigns region codes to nodes in a tree each of which is characterized by a triplet ([*left*: *right*], *level*) that provides positional information for each node [27]. For instance, Section 4.2.1 illustrates such a region code encoding scheme.

The main contribution of the thesis to XPath query aggregation is that two new XPath aggregation algorithms are more effective in evaluating the ancestor/desendant operators (//) than the XSearch algorithm. The complexity analysis is presented in Section 4.2.6 and the performance evaluation is presented in Section 4.3.

4.2 XPath query aggregation algorithm using region encoding scheme

This section presents an adapted method for checking containment between XPath queries. It is based on the region codes [27]. The new query aggregation approach has three main parts. The first is to create a global query index tree, in which each node is assigned with a region code (left, right, level). The region code represents the positional information of a node. The second and third parts of the approach are the two new aggregation algorithms, containee and container. The containee algorithm is used to determine whether the new arriving query covers or contains part of the global query tree. The container algorithm, on the other hand, is used to check whether the new arriving query is contained in the global query tree.

Section 4.2.1 introduces the region encoding scheme for tree nodes and presents the data structures used in the two new algorithms. And then, Section 4.2.2 describes the creation of a global index tree, which is followed by the presentation of the new XPath query aggregation algorithms in Sections 4.2.3 and 4.2.4. The update mechanism for node addition/removal is discussed in Section 4.2.5.

4.2.1 Region node coding and the data structures

This section describes the notion of region node coding and explains how the code is generated. Region encoding [27] is performed through a pre-order traversal of the tree. Each node t in the global query index tree is associated with a tuple ($sub(t)$, [left: right], level). The first term, sub(t), represents the set of query ids which share node t. The value of the *left* attribute is the number given to a tree node in a pre-order traversal of the tree. The value of the *right* attribute is the number given to the tree node after its children are recursively traversed from left to right. If the node is a leaf, the value of its right attribute is equal to its left value plus 1. The left attribute denotes the left position of t in the global query index tree; the right attribute is the value of the right position of t in the tree; and the *level* is the depth of node t as measured from the root node. Consider the left-most element $//b^1$ at the 2^{nd} level of the global query index tree in Figure 4.2a, the region code for node $//b^1$ is ({1,2,3,4}, [3:10],2). The first parameter {1,2 ,3, 4} represents queries q_1, q_2, q_3 and q_4. Queries q_1, q_2, q_3 and q_4 share $//b^1$. The second parameter [3:10] depicts the left and right values. The left value of node $//b^1$ is 3 which is numbered based on the tree pre-order sequence. The right value is one larger than the largest right value of its children. For example, since node $//b^1$ has three children e^1, d^1, and c^1, and the largest right value of the children of $//b^1$ is 9 for node c^1, the right value for $//b^1$ is therefore $9 + 1 = 10$.

The region codes can be used to determine the ancestor/descendant and parent/child relationship. For instance, consider two nodes t_1 and t_2; t_1 with region

encoding ($[l_1{:}r_1]$, d_1) and t_2 with region encoding ($[l_2{:}r_2]$, d_2). The structural relationship between these two nodes t_1 and t_2 can be determined by the following:

1. t_1 and t_2 have an ancestor/descendant relationship if and only if $l_1 < l_2$ and $r_1 > r_2$;

2. t_1 and t_2 have a parent/child relationship if and only if $l_1 < l_2$, $r_1 > r_2$ and $d_2 = d_1 + 1$.

For instance, consider node a with region code ([2:21],1) and node $//b^1$ with region code ([3:10],2). Node a and node $//b^1$ satisfy the parent/child relationship. Moreover, for node a with region code ([2:21],1) and node c^1 with region code ([8:9],3), node a and node c^1 satisfy the ancestor/descendant relationship. However, node $//b^1$ with region code ([3:10],2) and node c^2 with region code ([11:12],2) do not satisfy either relationship.

The steps of operations for generating the region code is presented in Algorithm 3.

Algorithm 3: *generateRegionCode*(*t*, *num*, *level*)

input : *t* is the current working node in the global query tree, *num* is the sequence number generated using pre-order traversal and *num* is an integer number, and *level* is the level of *t* in the global query tree

output: Each node of the subtree under node *t* has a region code

```
1  t.left = ++num;
2  t.level = level;
3  level ++;
4  if t is non-leaf node then
5  |   foreach child t' of t do
6  |   |   generateRegionCode (t', num, level);
7  |   |   num = t'.right;
8  |   t.right = ++num;
9  |   set query IDs, leaveID, and ancestor/descendant operator information;
10 else
11 |   t.right = ++num;
12 |   set query IDs, leaveID, and ancestor/descendant operator information;
13 |   return;
```

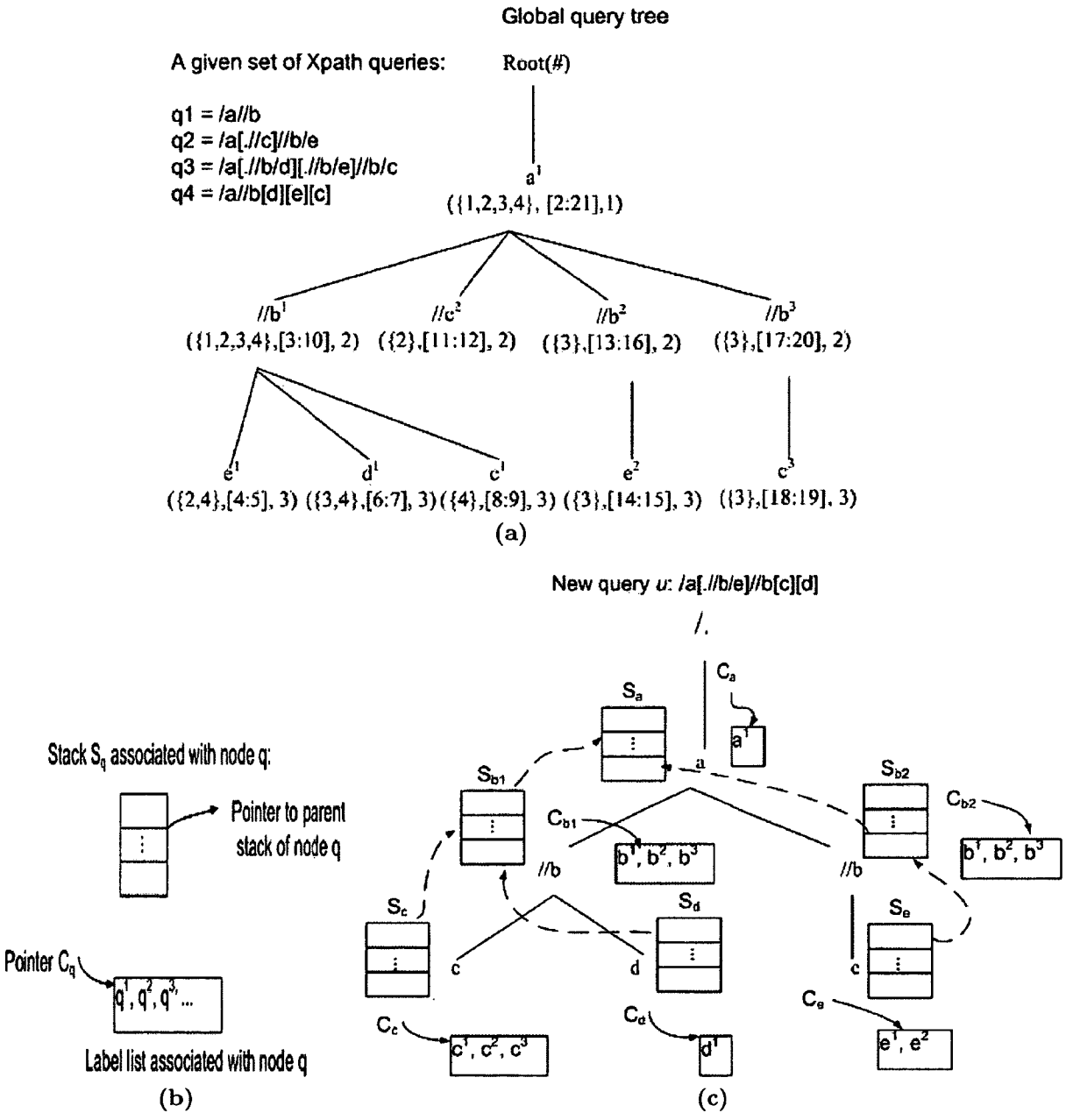


Figure 4.2: The data structures associated with a new XPath query : (a) an existing global query tree example; (b) an example of the data structures associated with a new query node q; (c) an example of the data structures for the new query tree.

q is a general term that refers to actual nodes in a new query. The function $q.getChildren()$ gets all the child nodes of q. For example, $a.getChildren()$ is the list $\{//b^1, //c^2, //b^2, //b^3\}$ in Figure 4.2a. The function $q.sub()$ gets all the query ids associated with node q. For example, $a.sub()$ returns a list of {1, 2, 3, 4}, the first parameter under node a, which means that queries q_1, q_2, q_3 and q_4 share node a in the global query tree.

Next, a hash table is used to index region codes based on node labels. A label list is a list of region codes for nodes which have identical node labels. For example, Figure 4.3 shows a set of five label lists for the global query index tree shown in Figure 4.2a.

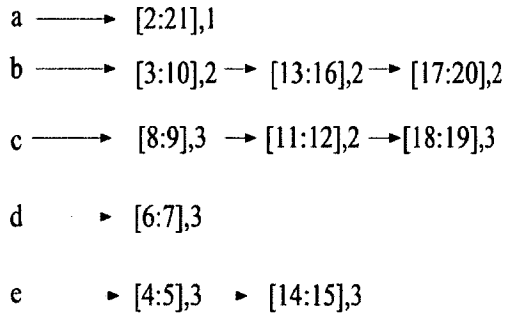


Figure 4.3: Label lists for the global query index tree in Figure 4.2a

As shown in Figure 4.2b, there is a stack and a label list associated with each node q. The stack associated with q is denoted as S_q and each stack has a pointer to the stack of the parent node of q. For each q, there is a pointer pointing to an entry in the corresponding label list of q, denoted as C_q. All entries in the label lists store information of region codes of nodes in the global query tree and they are sorted according to their $left$ values in each list. The attributes of a region code can be accessed by $C_q.left$, $C_q.right$ and $C_q.level$. For example, for node $//b$ of the new query, the pointer associated with node $//b$ is represented as C_{b1} and $//b$ has a label list $\{b^1, b^2, b^3\}$ as shown in Figure 4.2c. If C_{b1} points to b^1 in the label list, then $b^1.left$ is 3, $b^1.right$ is 10 and $b^1.level$ is 2.

Matched elements of node q are pushed onto S_q. Each element in S_q has a pointer

to the corresponding parent element stored in the stack of $q.parent$. Stacks encode the matched elements during the comparing process in a compact way. For instance, for a branch node, only one copy of the matched element of node q needs to be stored, instead of multiple copies of the matched elements for multiple branches.

Aggregating queries is needed when new queries are to be added to a query index tree. To identify the containment or covering relationship, two methods are devised. One algorithm (containee) is used to find the existing queries in the query tree covered by a new query; the other algorithm (container) is used to identify existing queries that cover the new query. These two algorithms are described in Section 4.2.3 and Section 4.2.4, respectively.

4.2.2 Construction of the global query index tree

The query aggregation algorithm builds on a global query index tree which is the same as the XSearch algorithm [42]. Figure 4.2a shows an example of four XPath queries and the corresponding global index tree.

A global query index tree represents a set of XPath queries and enables the prefix sharing between XPath queries. The difference between our tree and the factorization tree used by the XSearch algorithm is that a $//n$ node is represented as one node in our tree instead of two separate nodes (// and node n).

Consider node q as a node of a new query and node t as a node in a *global query index tree*. Each node t of a global query index tree has a node label and is associated with a set of query ids which can be accessed by the function $t.sub()$.

The process of adding a new query into a global query index tree is performed in a top-down fashion. For a node q with a query id of id to be added to the tree, the adding algorithm needs to find a child node t of the current working node t_0 in the global query index tree with the same label as q such that id is not a member of $t.sub()$. If there is an existing child already in the tree, then id is added to $t.sub()$. Otherwise, a new node t is created and id is added to $t.sub()$. Then, t is added as a

new child of t. The addition process of the subtree rooted at q continues recursively. The function $q.leaves()$ gets all the query ids if q is a leaf node.

Algorithm 4 presents the process of adding an XPath query node to the query index tree. Algorithm 5 illustrates the steps for removing an XPath query based on *id*.

Algorithm 4: *addQuery*(q, *id*)

```
   input  : q is the new query tree to be added to the global query tree and id is the
            query id associated with q
   output: Query q is added to the global query tree
 1 Let t0 be the current working node in the global query tree;
 2 foreach child t of t0 do
       /* first check whether q is already in the children of t0          */
 3     if ∃t, such that t.name is q.name ∧ id ∉ t.sub() then
 4         add id to t.sub() ;
 5         if q is a leaf node then add id to t.leaves() ;
 6         break ;
 7 if no such t exists then
 8     t = create a new node instance ;
 9     add t as a child of t0;
10     add id to t.sub() ;
11     if q is a leaf node then
12         add id to t.leaves();
13 if q is a non-leaf node then
14     foreach child q' of q do
15         t.addQuery (q', id);
```

Figure 4.4a shows a global query tree after q_1 (/a//b) and q_2 (/a[.//c]//b/e) are added and Figure 4.4b illustrates the result of adding q_3 (/a[.//b/d][.//b/e]//b/c) to the global tree. The adding process starts from the root node *Root* of the global query tree. Since there is a child node a under the node *Root* that has the same label as the root node a of q_3 and the id of q_3 is not in the $a.sub()$, the adding process makes node t point to node a and adds $\{3\}$ to $a.sub()$ (line 2-4) in Algorithm 4. Because a is not a leaf node, the adding process recursively calls itself to complete the addition of q_3 (line 13-15). To add the second branch ([.//b/e]) of q_3, Algorithm 4 finds no

Algorithm 5: *removeQuery(id)*

input : *id* which is a query id associated with the query q to be removed
output: Query q is removed from the global query tree

```
1  let t be the current working node;
2  remove id from t.sub();
3  if t is a leaf node then
4      remove id from t.leaves();
5  if t is a non-leaf node then
6      foreach child t' of t do
7          if id ∈ t'.sub() then
8              t'.removeQuery (id) ;
```

such node t existing in the global tree that has the label name b and q_3 is not in the $b.sub()$. The first node $//b$ has q_3 in query id list after adding the first branch [.//b/d]. Hence, the algorithm creates a new node t such that t has b as label and $t.sub()$={3} (line 7-10). The node $//b^2$ in Figure 4.4b is the result of node t.

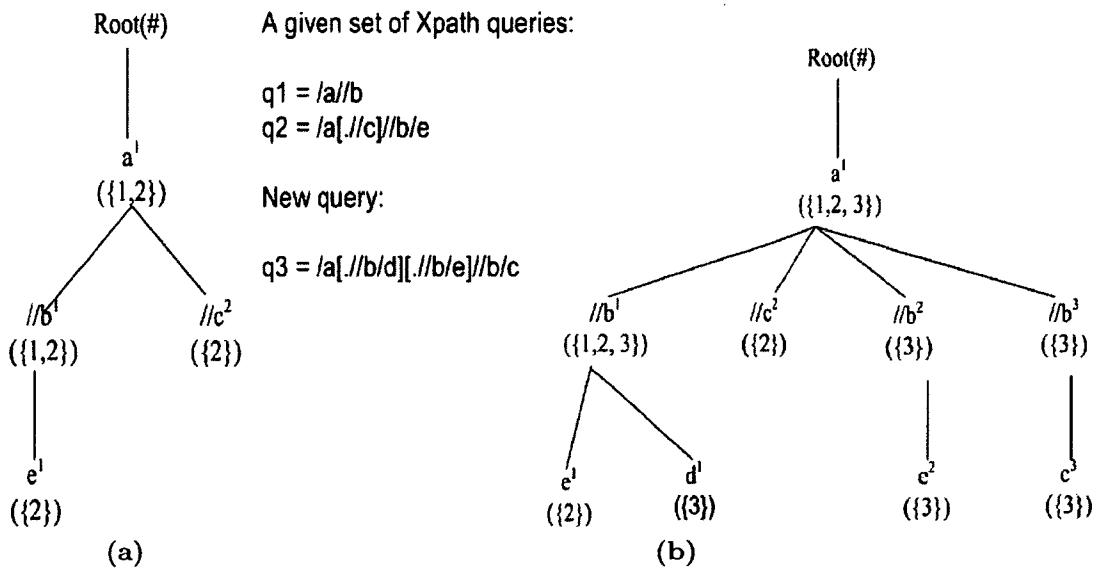


Figure 4.4: Example of global query tree construction: (a) before q_3 is added to the global query tree; (b) after q_3 is added.

4.2.3 Containee algorithm of the new approach

This section presents the containee algorithm which is needed to find a subset of existing queries contained by a new query. Figure 4.5 shows the concept of finding the queries which are contained or covered by the new query u: /a[.//b/e]//b[c][d]. The idea is to determine whether every node of u can be mapped to nodes of the global query tree. In addition, the parent/child relationship and the ancestor/descendant relationship among nodes in the global query tree should be consistent with query u. For example, in Figure 4.5, both nodes and their structures of /a//b/e of u are marked in a slash pattern and the mapped nodes of /a//b[c][d] are marked in a dot pattern. The leftmost node $//b$ in the global query tree is shared by slash pattern and dot pattern areas.

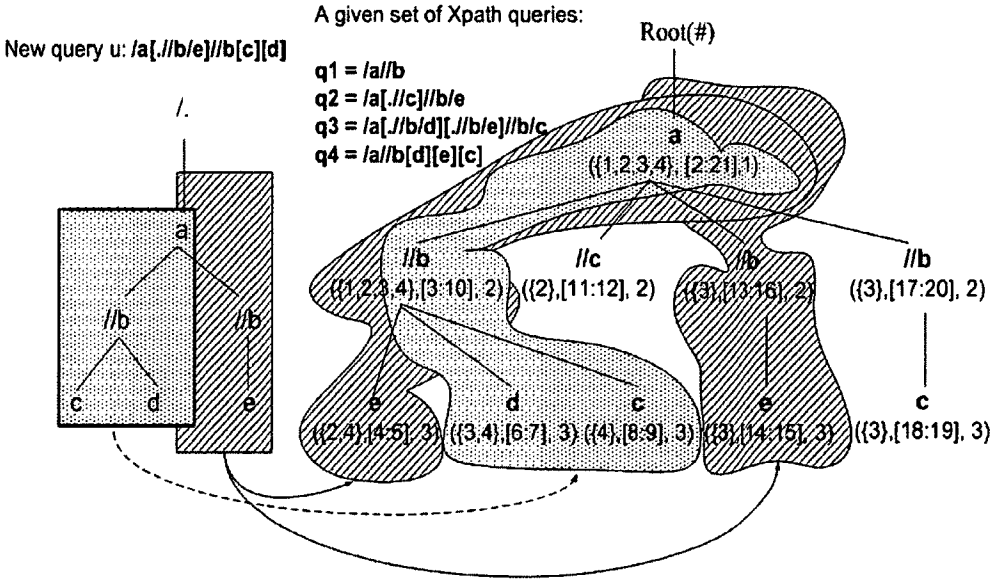


Figure 4.5: The concept of a containee algorithm

The containee process is presented in Algorithm 6 to Algorithm 10. It operates on the label lists which store region codes for nodes in the global query tree, instead of operating on the global tree directly. The region code for node n contains the position information of n in the tree as described earlier in Section 4.2.1.

Algorithm 6 is an adaptation of the TwigStack algorithm [27]. It associates a stack

S_n and a label list T_n with each node n of a given new query. The stack keeps track of matched nodes from the global query tree. Algorithm 6 is a bottom-up process which searches all potential solutions guaranteed to join the final results. If a leaf node of the new query is met, the algorithm outputs the solution currently in stack from the root node to the leaf node and stores the solution at the leaf node and fill the match query id to the nearest branch node. The variable *MatchSet* for each node is just a data structure implemented as an array list that holds the id information for queries in the global query tree covered by the corresponding query node of new query. The *actual* node is a node of the new query returned by the function $getNext(q)$ and pointing to the working node.

The key functions used are described as follows:

(i). *containee(q)* is the main algorithm which computes the queries covered by the new query q.

(ii). *getNext(q)* is called by the containee algorithm with node q as input and returns the highest possible node in the new query tree which may have a mapping node in the global query tree.

(iii). *cleanStack(actL)* pops elements from the current stack which have right positions that are smaller than $actL$. If the stack of a descendant node is not empty, clean this stack first. The process of $cleanStack()$ method operates recursively.

(iv). *computeSubResult()* returns the match set results for one branch.

(v). *isEndOf(q)* checks whether a list associated with q has reached the end of the list.

The helper methods used by the containee algorithm are explained in the following. The $showSolutionFromStack()$ method checks the parent-child relationships between stacks and outputs matching elements from stacks. The function $q.cur()$ returns the

Algorithm 6: *containee*(q)

input : q is a new query tree which wants to identify the set of queries in the global query tree that are contained by q

output: A set of queries in the global query tree that are contained by q

```
   /* a bottom-up process                                                    */
 1 while the ends of lists associated with q have not been reached do
       /* returns highest possible node in the new query tree which may
          have a mapping node in the global query tree                      */
 2     actual = getNext (q.root);
 3     if actual is not the root node of q then
           /* clean elements from the parent stack S_actual.parent whose right
              value of its region code is smaller than the left value of
              the region code for the current element C_actual in the label
              list T_actual because they cannot be ancestors of C_actual    */
 4         actual.parent.cleanStack (C_actual.left);
 5     if actual is the root node of q || the stack S_actual.parent is not empty then
           /* clean S_actual by popping elements in S_actual whose right value
              of the region code is smaller than the left value of the
              region code for C_actual being pushed onto S_actual           */
 6         actual.cleanStack (C_actual.left);
 7         push current element C_actual of the label list T_actual onto the stack S_actual ;
 8         if actual is a leaf node then
               /* showSolutionFromStacks checks the parent/children operator
                  (/) traversing from leaf stack to root stack              */
 9             success = actual.showSolutionFromStacks (actual);
10             if success then
                   /* set sub-results to the nearest branch node p and
                      realRank is the branch number associated with actual  */
11                 add the query ids of the top element in Stack S_actual to
                   actual.MatchSet;
12                 p = findNearestBranchNode (actual);
13                 p.MatchSet[realRank].add(actual.MatchSet);
14             pop the top element from the stack S_actual ;
15     else
16         make the next element to C_actual to be the new C_actual in the label list T_actual;
   /* empty remaining elements in stacks and set the sub-results            */
17 if the lists associated with q have reached end of lists then
18     foreach node n on the path from actual to root do
19         n.cleanStack (maxValue);
```

current element of the label list T_q pointed by C_q. The $C_q.left$ and $C_q.right$ functions return values of the left/right positions of C_q.

Algorithm 7: *getNext*(q) sub-algorithm

input : q is a new query tree which is used to identify the set of queries in the global query tree that are contained by q

output: A query node of q which all its descendant nodes have matching elements from corresponding label lists associated with them or a query node which current element in the associated label list has the minimum left value

```
1  if q is leaf node then
2      return q;
3  foreach child q_i of q do
       // recursive call
4      n_i = getNext (q_i);
5      if n_i ≠ q_i then
6          return n_i;
   /* if parent and child elements are the same, move current child
      element to the next one                                        */
7  if C_{q_i}.left == C_q.left then
8      move C_{q_i} to the next element in the label list T_{q_i};
9  q_min = arg min_{q_i}{C_{q_i}.left};
10 q_max = arg max_{q_i}{C_{q_i}.left};
   /* advance T_q to make sure the current element pointed by C_q is a
      common ancestor of all child elements pointed by (C_qi)       */
11 while C_q.right < C_{q_max}.left do
12     move C_q to the next element in the label list T_q ;
13     if the end of the label list T_q is met then break;
   /* if no common ancestor for all C_qi is found, return the child node
      with the smallest start value q_min; else return q             */
14 return C_q.left > C_{q_min}.left? q_min : q;
```

The function $getNext(q)$ is a key method which checks the corresponding label list T_q for the matched nodes and identifies matched elements. The containee algorithm pushes or pops elements onto stacks returned by $getNext(q)$ and outputs matching elements via $showSolutionFromStack()$.

The function $getNext(q)$ first recursively calls itself for each child node of q (lines 3-4). When a leaf node is reached, $getNext(q)$ processes from the bottom up (lines 1-2). For node q, if every child q_i is equal to the returned result of $getNext(q_i)$, we

Algorithm 8: *cleanStack(value)* sub-algorithm

input : *value* is an integer number
output: Compute sub-results and remove elements from the stack S_q which right value of its region code is smaller than *value* and remove elements from descendant stacks

```
1  Let the current working node is q ;
2  while (S_q is not empty) ∧ (S_q.topElement.right < value) do
       /* clean stacks of descendant nodes of q and maxValue is the
          maximum integer value                                    */
3      foreach each child q' of q do
4          if S'_q is not empty then
5              q'.cleanStack (maxValue);
6              foreach child c_1 of q' do
7                  c_1.cleanStack (maxValue);
8      q.computeSubResult ();
9      pop the top element from S_q;
```

need to find an element pointed by C_q in the label list associated with q which is a common ancestor of all the elements pointed by C_{q_i} via advancing C_q (line 12). C_{q_i} is a pointer associated with child node q_i. If such a common ancestor element pointed by C_q is found, node q is returned; otherwise, the child node with the smallest $left$ value q_{min} is returned (line 14). The function $q.cur()$ returns the current element pointed by C_q. The function $\arg\min\{C_{q_i}.left\}$ returns the child node q_i of q with the smallest $left$ values; the function $\arg\max\{C_{q_i}.right\}$ returns the child node q_i of q with the biggest $right$ values (lines 9-10). The condition of a common ancestor element is $C_q.left < C_{q_{min}}.left$ and $C_q.right > C_{q_{max}}.left$. Lines 7-8 handle the case where parent and child nodes have the same label, for instance, a new query /a/b/b/c.

The computation of queryIDs for the contained queries is conducted by *comute-SubResult()*. The *computeSubResult()* method returns the matching set result for one branch. The *computeSubResult*() is called either when a node n of the new query is a branch node and its matching element is being popped from its stack or when all

Algorithm 9: *computeSubResult*() sub-algorithm

```
   input :
   output: Compute sub-results
1  Let the current working node be q ;
2  if q is a branch node then
       // intersection
3      matchSet = ∩0≤i≤children.size−1 q.MatchSet[i];
       // assign subResult to the nearest branch node p and realRank is
          the branch number associated with q
4      p = findNearestBranchNode (q);
5      p.MatchSet[realRank].add(matchSet);
6  else
7      matchSet = q.MatchSet[0];
```

its childrens end conditions are satisfied. An example for the reason of using intersection is explained in Figure 4.8. Subsequently, an intersection operation is needed among all the match sets of every child of n (line 3 of Algorithm 9) to determine the match set for node n. Sub-results for all matching elements of this node n are added together (line 5 of Algorithm 9). This is one of the differences between our algorithm and the TwigStack algorithm [27]. The proposed new algorithm does not need the TwigStack's merge-join operation, the post-processing phase to obtain the query ids results for existing queries that are contained by a new query. The final result of the set of ids for queries which are contained by the new query is obtained after the completion of Algorithm 6.

The *isEndof()* method checks the end condition, computes the sub-result for node q and cleans the stacks if all nodes of *subtree*(q) reach the end condition.

In summary, the containee algorithm is a new aggregation approach that uses region encoding to effectively evaluate the ancestor-descendant or parent-child relationship between query nodes. In addition, the label lists for the global query index tree enable the algorithm to search only the label lists associated with the new query, instead of searching the whole query index tree. The XSearch algorithm [42], on the other hand, has to search the complete query index tree and map a //-node to paths

Algorithm 10: *isEndOf*(q) sub-algorithm

```
   input  : q is a query node of the new query
   output: Check whether the end condition of the containee algorithm is satisfied
   /* check the end condition for leaf nodes                                  */
 1 if q is a leaf node then
 2     return (C_q reaches the end of T_q) ∧ (S_q is empty) ;
 3 else // for a non-leaf node
 4     foreach child q' of q do
 5         if !isEndOf (q') then
 6             return false;
       /* clean the remaining elements in stacks when this node's end
          condition is met and maxValue is the maximum integer value          */
 7     if S_q is not empty then
 8         q.cleanStack (maxValue);
 9     return true;
```

of $length = 0$ and $length \geq 1$.

Figure 4.6 depicts snapshots of the stacks at runtime. There are four existing queries in the global query tree as shown in Figure 4.5. These queries are evaluated against the new query u. There is a stack associated with each node of the new query u. There are two branches heading with b, so there are two stacks for b in the new query u, e.g., S_{b1} and S_{b2}. The content of the stack represents the left value and right value of the region code, and the level of a matching node in the global query tree. For instance, ([6:7],3) in stack S_d, as illustrated in Figure 4.6a, represents the region code for node d in the query tree, shown in Figure 4.5. A pointer associated with a stack element of stack S_d pointing to stack S_{b1} which in turn is pointing to S_a represents a path from a leaf node (e.g., d) to the root (e.g., a). Figure 4.6a shows the stack status at Iteration 3 of Figure 4.7. The right-most branch a//b/c in the global tree does not match /a//b/c in the new query u because d is not a child node of the node //b with the region code ([17:20],2) in the global tree. Figure 4.6b presents the status at Iteration 4 of Figure 4.7. In the global query tree, there are two /a//b/e branches which match the right-most branch /a//b/e of the new query u. Figure 4.6c

New query u: /a[.//b/e]//b[c][d]

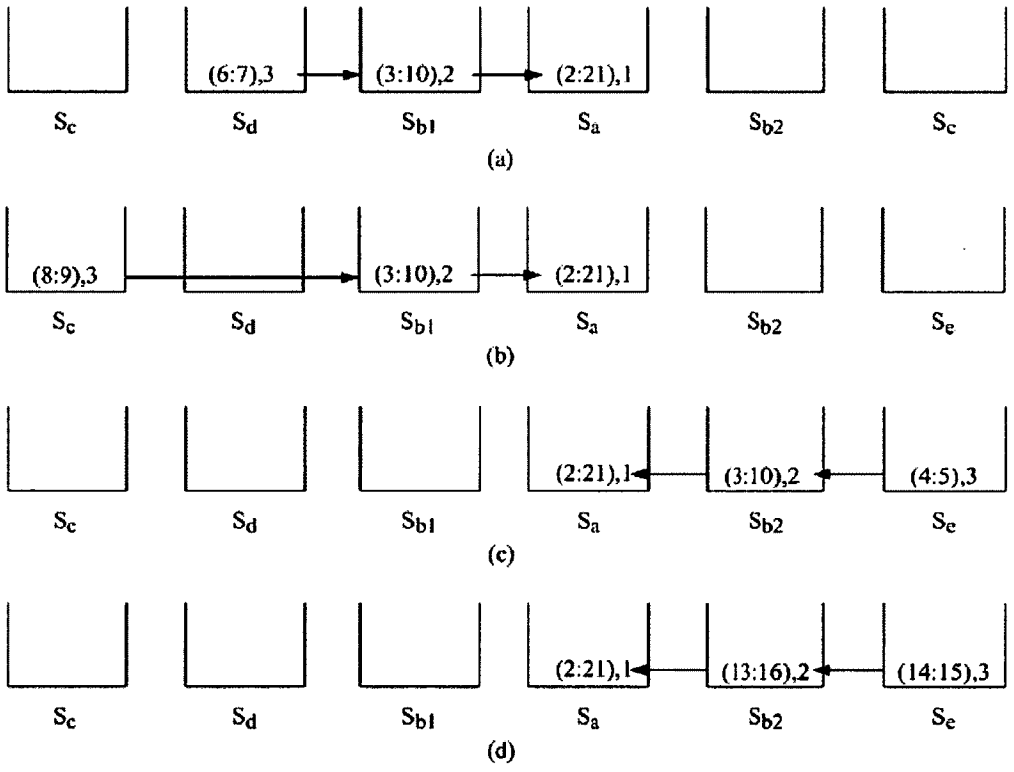


Figure 4.6: Snapshots of the stacks for processing the new query u shown in Figure 4.5

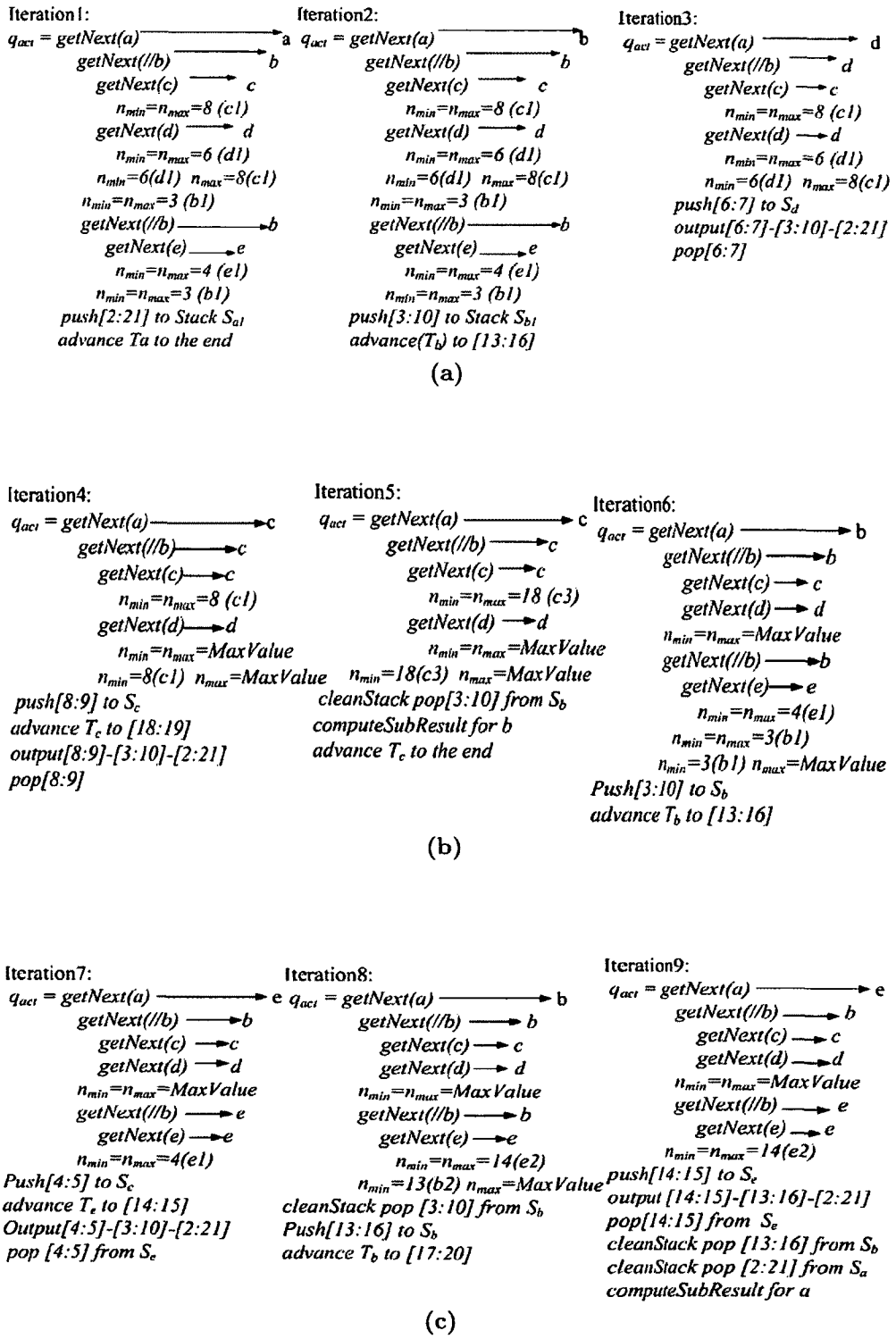


Figure 4.7: A containee example run

is the status of Iteration 9 and Figure 4.6d is a snapshot at Iteration 9 of Figure 4.7.

Figure 4.7 illustrates an example run of the containee algorithm, where a new XPath query u in Figure 4.2c is matched against the global query index tree containing four XPath queries in Figure 4.2a. The matched results are listed thus:

(i). ([8:9],3)-([3:10],2)-([2:21],1) for path /a//b/c, and the corresponding query id is {4};

(ii). ([6:7],3)-([3:10],2)-([2:21],1) for path /a//b/d, and the corresponding query ids are {3,4};

(iii). ([4:5],3)-([3:10],2)-([2:21],1) and ([14:15],3)-([13:16],2)-([2:21],1) for path /a//b/e, and the corresponding query ids are {2,3,4}.

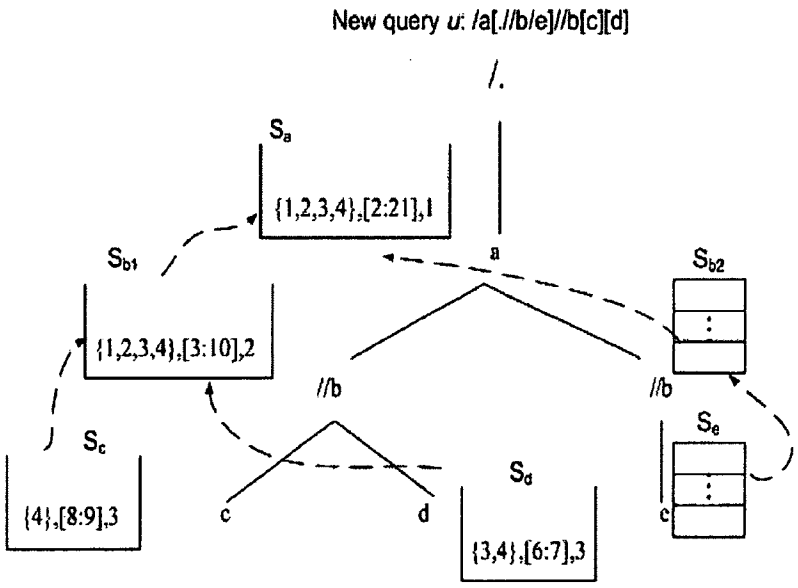


Figure 4.8: An example of *computeSubResult*() for the new query u

When ([3:10],2) is popped from stack S_{b1} at Iteration 5 in Figure 4.7, the *computeSubResult()* function is called upon as shown in Figure 4.8. Before the element ({4},[8:9],3) in the stack S_c is popped, the sub-result (query id {4}) is recorded at the nearest branch node *//b* of the leaf node *c*. Similarly, before the element ({3,4},[6:7],3)

is removed from the stack S_d, the sub-result (query ids {3,4}) is recorded at the nearest branch node //b of the leaf node d. Since //b is a branch node in the new query u and has two child nodes c and d, an intersection is applied here to filter out unsatisfactory queries, e.g., query q_3. Query q_3 (/a[.//b/d][.//b/e]//b/c) has node a as the branch node, while query q_4 (/a//b[d][e][c]) has node //b as the branch node. The sub-result $\{4\} \bigcap \{3,4\} = \{4\}$ is obtained. At Iteration 8 of Figure 4.7, after ([4:5],3) is popped from the stack S_e, ([3:10],2) is popped from its stack and ([13:16], 2) is moved to stack S_{b2}. At Iteration 9, after the solution ([14:15],3)-([13:16],2)-([2:21],1) is found from stacks, ([14:15],3) is popped. For the query node b_2 in u, since its child query node e has reached its end condition, the element ([13:16],2) is popped from stack S_{b2}. Furthermore, the remaining element ([2:21],1) is popped off from the stack S_a. Since node a is a root node and a branch node, the *computeSubResult*() is called for node a to compute the intersection result: $\{2,3,4\} \bigcap \{4\} = \{4\}$. The final answer is {4}. Query q_4 is covered by the new query u.

There is one important issue that needs to be explained: how to determine if all matching elements corresponding to a node of a new query have already found all matching elements in the global query index tree. If a matching element is popped from the stack when another matching element is to be stacked, the parent element, should be popped before the child elements. Figure 4.9 depicts such an example. In the global query tree, there are left *body* element with region code ($\{Q_1$–$Q_6\}$,[7:58],2) and right *body* element with region code ($\{Q_1,Q_3,Q_5\}$,[59:70],2) at the second level of the global query tree. In this example, the right *body* element in the global query tree contains all elements specified in the new query. So the right *body* element ($\{Q_1,Q_3,Q_5\}$,[59:70],2) can be put in the stack of S_{body} and the left *body* element ($\{Q_1$–$Q_6\}$,[7:58],2) should be popped from stack S_{body} according to line 6 in Algorithm 6, because the left *body* element has been processed and should be cleared. At this point, the matching nodes are still in the stack for *body.content*, *hr*, *body.end* and *bibliography*, e.g. ($\{Q_2\}$,[40:43],3), ($\{Q_2\}$,[41:42],4), ($\{Q_2,Q_5,Q_6\}$,[14:23],3) and

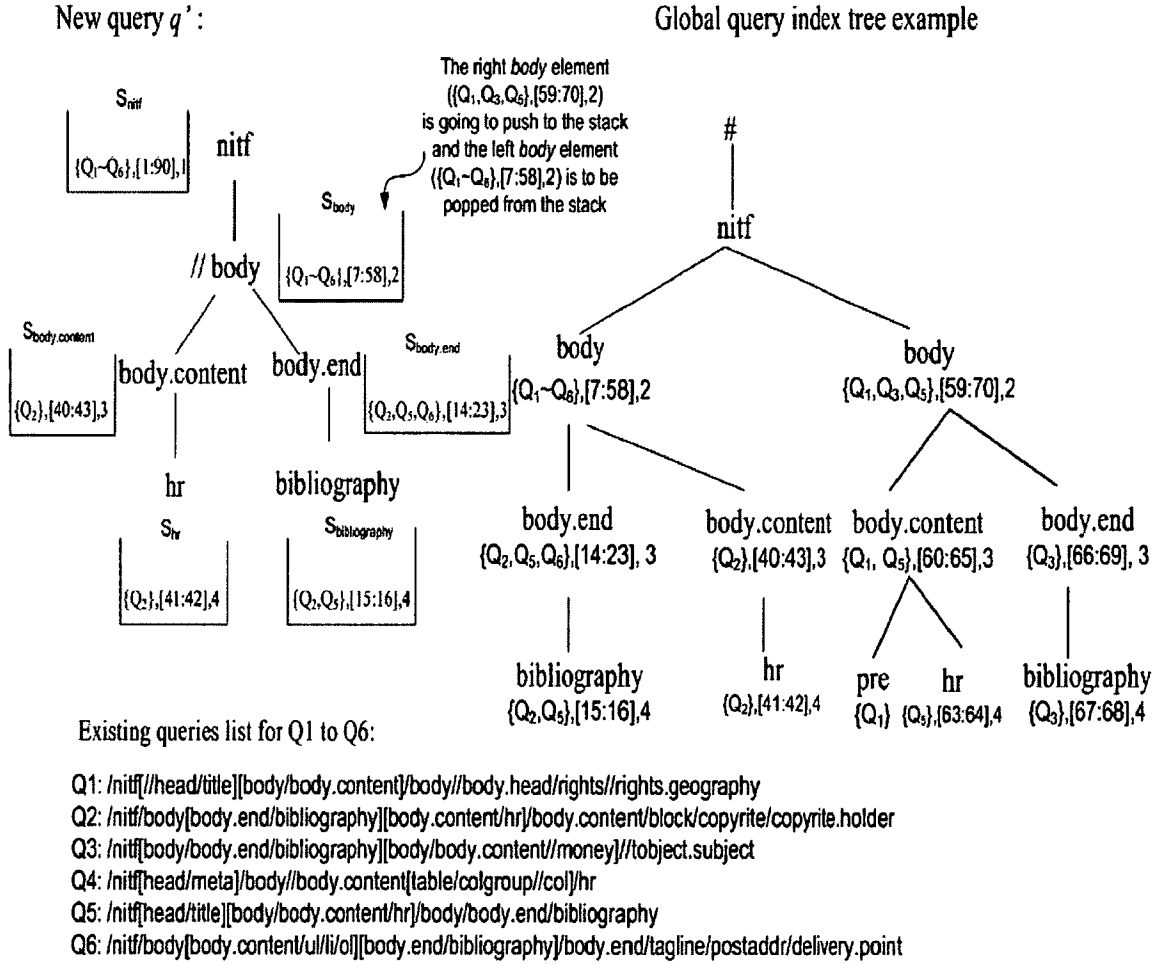


Figure 4.9: An example of parent node popped before child node of the containee algorithm

($\{Q_2,Q_5\}$,[15:16],4). To handle this situation, when an element is to be removed from its stack, the algorithm first checks whether the stack for its descendant elements are empty. If it is not empty, the elements in the descendant stack are popped first, then the node computes sub-results which are an intersection of the results from its children and passes the sub-results to their nearest ancestors. Lines 7 and 8 of Algorithm 10 describe these steps.

The differences between the new containee algorithm and TwigStack algorithm [27] are summarized as follows.

(i). The target problem is different. The containee algorithm is used to determine whether a new arriving query contains part of the existing queries, while the TwigStack algorithm matches a query with an XML publication document.

(ii). The new containee algorithm is efficient because the intermediate results are computed from bottom up. The stacks associated with descendant nodes of a node q in a new query are cleaned before popping the matching element from the stack associated with q and the intermediate results are recorded at the corresponding nearest ancestor branch nodes. As a result, the containee algorithm does not require another merge-join post-processing to get the matched query ids as used in TwigStack algorithm.

Due to those differences, the new containee algorithm can identify the queries contained by a new query correctly and efficiently. Section 4.3 presents the performance evaluation for the new containee algorithm.

4.2.4 Container algorithm of the new approach

The aims of the container algorithm are to check if the existing query covers the new query. The idea of the container algorithm is to first identify queries in the global query tree that do not cover the new query, and then to compute the complement of a set of queries that do not cover the new query in order to compute the set of

queries that cover the new query. These operations of the container algorithm are conceptually similar to operations performed in the XSearch algorithm.

There is a data structure called label list for the new query. To compute the container result, the new query is encoded using the region encoding scheme and indexed using a hash table based on node labels. Figure 4.10 depicts the encoding results and the label lists for a new XPath query u: /a[.//b/e]//b[c][d].

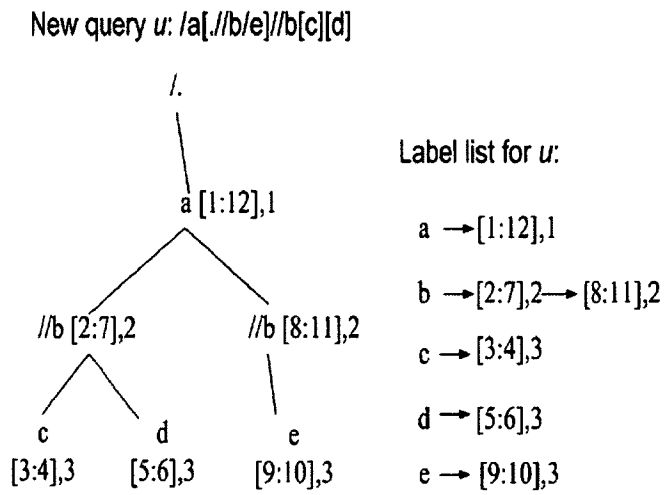


Figure 4.10: Node encoding example for a given new query

Figure 4.11 shows the data structure for the global query index and the new query. There is a label list from a new query associated with each node t in the global query tree. All entries in the lists are assigned with region codes and they are sorted according to their $left$ values in each list. For each node t, there is a pointer pointing to an entry in the corresponding label list of t, denoted as C_t. The attributes of a region code can be accessed by $C_t.left$, $C_t.right$ and $C_t.level$. The function $t.sub()$ returns the query ids associated with node t. For example, $e^1.sub()$ returns a list of {2,4}. For example, the leaf node e^1 has a label list $\{e([9:10],3)\}$ and a pointer C_e.

The container algorithm is presented in Algorithm 11. Algorithm 11 works recursively on the nodes of the global query index tree in pre-order sequence to search paths in the new query that are contained by the queries in the global query tree. The

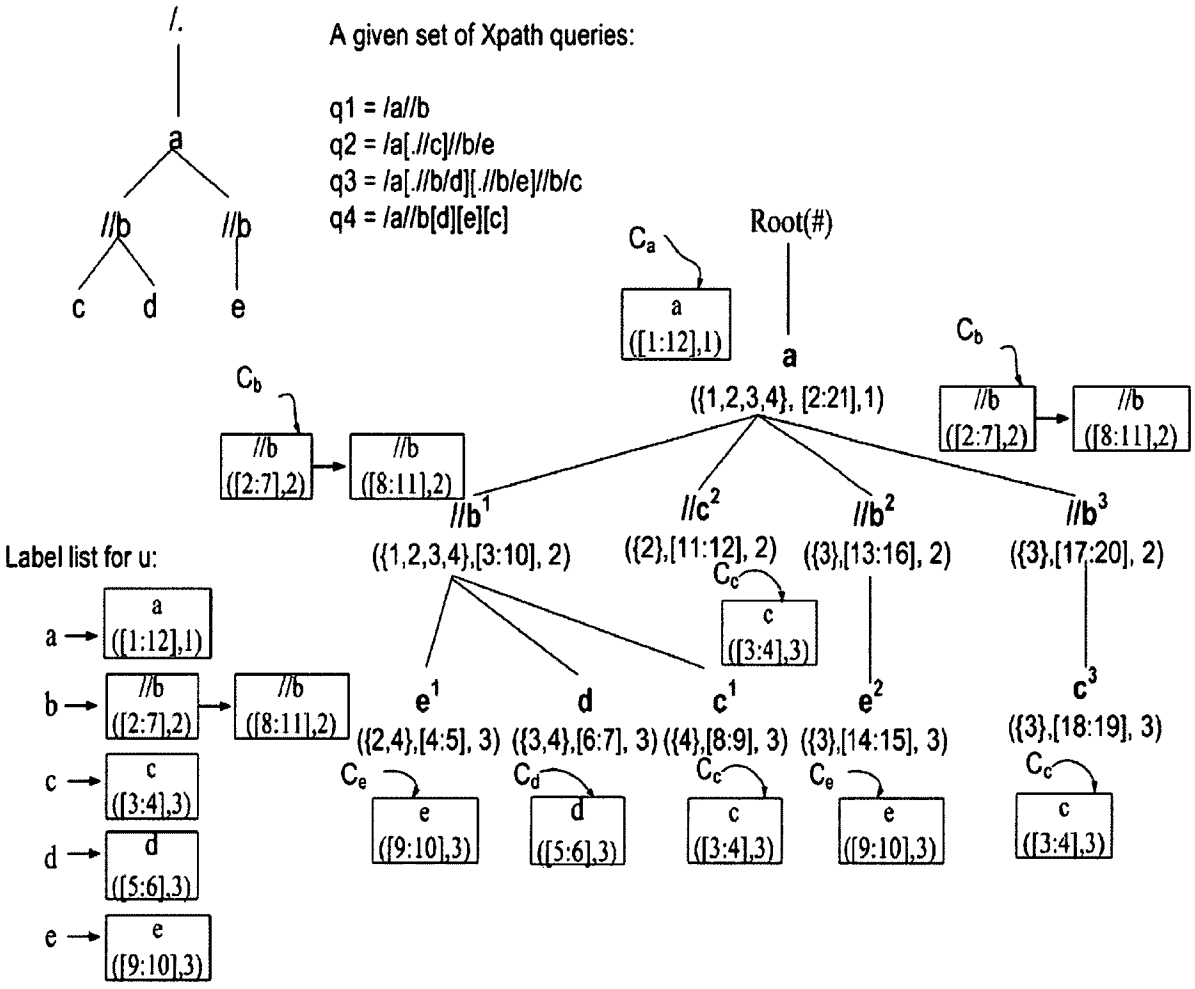


Figure 4.11: An example of the data structure used by the container algorithm

recursive function in Algorithm 11 returns the complement results that do not contain the new XPath query. A query path in the global query tree that has a covering mapping to the new query path should have both label match and the parent/child or ancestor/descentant match and should have a shorter or equal path length of the new query. A query that does not contain the new query includes the following cases:

(i) a node with a node label that is absent in the new query.

(ii) a query path in the global query tree that is incompatible with the corresponding path in the new query. For instance, the relationship of parent/child of the global tree cannot be compatible with the relationship of ancestor/descendant in the new query.

(iii) a query whose depth is deeper than that of the new query.

For case (i), the associated query ids will be returned (line 5). For case (ii), the associated query ids will be returned as shown at line 25 for parent/child relationship and line 31 for ancestor/relationship. For case (iii), a query node in the global query tree that is compatible to the new query path and has a shorter path (line 24 and line 30) is acceptable and an empty set is returned. If a node t of the global query tree is not a leaf node, Algorithm 11 recursively calls itself for each child t' of t (line 23 and line 29). C_t represents the current working element in the list *labelList* associated with t in Algorithm 11.

A concrete example is used to illustrate the container algorithm (see Figure 4.11 and Figure 4.12). The container algorithm starts from the root node a of the global query tree. There is a dummy node r_T represented as # which is the root node of all existing XPath queries. The matching node for node a in the global query index tree is the root query node a of the new query with region code ([1:12],1). The variable *a.solution* is ([1:12],1). The container algorithm recursively calls itself for each child node of a since node a is not a leaf (line 10) in Algorithm 11. The container algorithm proceeds from top to bottom and left to right on the global query tree.

Algorithm 11: *container*(t)

input : t which is a node in the global query tree
output: A set of existing queries that contain the new query

```
1  if t is the root r_T of the global query tree then // the dummy root of the global
   query tree
2      container(r_T)= sub(r_T)− ∪_{t'∈child(r_T)} container(t');
3  else
4      Let labelList be the label list associated with t ;
5      if no such labelList exists then return t.sub() ;
6      if t is real existing query root then
7          C_t is the current element in the label list associated with t;
8          if (t is not //-node) ∧ (C_t is not //-node) then
9              t.solution= C_t;
10             if t is not a leaf node then container(t) = ∪_{t'∈child(t)} container(t');
11             else // root and leaf
12                 return;
13         else // t is a //-node and t is a real query root
14             foreach entry in labelList do
                   /* C_t ⪯ t means that t covers C_t                              */
15                 container(t) = ∩_{C_t⪯t} ∪_{t'∈child(t)} container(t') ;
16     if t is not query root then
17         foreach entry in labelList do
18             C_t is the current element in the label list associated with t;
               /* the variable pSolution holds the matching element of t's
                  parent                                                          */
19             pSolution = t.parent.solution;
20             if t is not //-node then
21                 if (C_t is not //-node) ∧ (C_t and pSolution have parent/child
                   relationship) then
                       /* the variable solution holds the matching element of
                          t                                                       */
22                     t.solution = C_t;
23                     if t is non-leaf then
                       container(t) = ∩_{C_t⪯t} ∪_{t'∈child(t)} container(t');
24                     else container(t) = ∅ ;
25                 else container(t) = t.sub();
26             else // ancestor/descendant relationship
27                 if C_t and pSolution have ancestor/descendant relationship then
                       /* the variable solution holds the matching element of
                          t                                                       */
28                     t.solution = C_t;
29                     if t is non-leaf node then
                       container(t) = ∩_{C_t⪯t} ∪_{t'∈child(t)} container(t');
30                     else container(t) = ∅;
31                 else container(t) = t.sub();
```

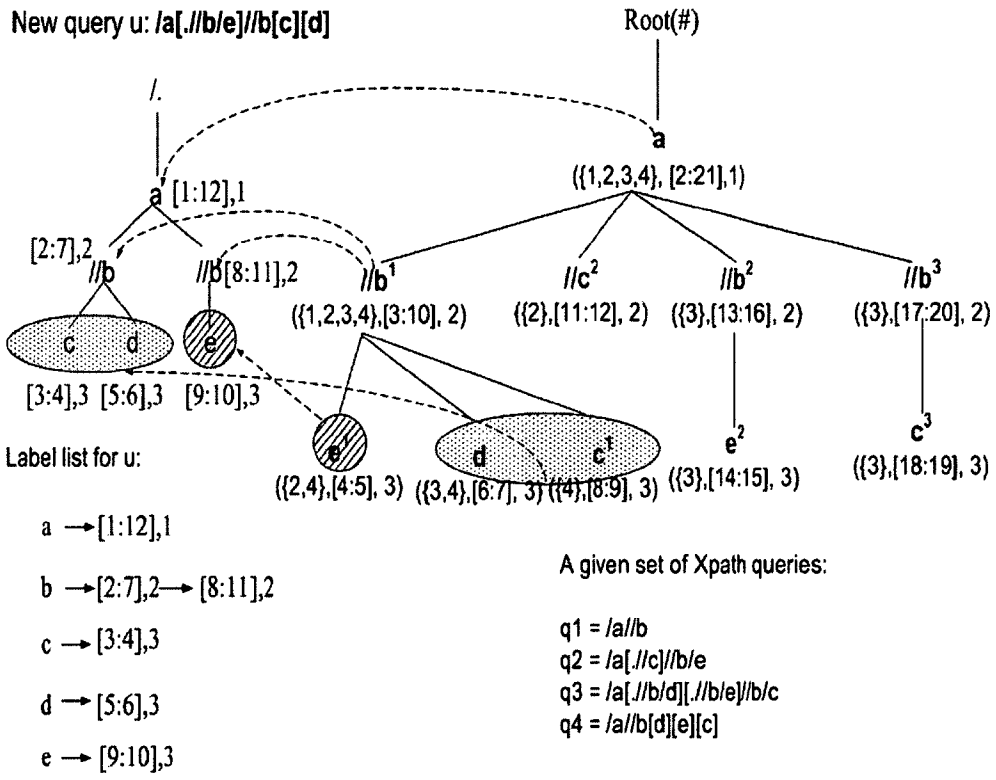


Figure 4.12: An example of container

The container algorithm processes the node $//b^1$ in the global query tree first. There are two elements in the b-list {([2:7],2) and ([8:11],2)} for the new query u. Node a with region code ([1:12],1) and node $//b$ with region code ([2:7],2) satisfy the ancestor/descendant relationship (lines 26-29). Since the query node $//b$ in the new query with region code ([2:7],2) covers $//b^1$ ($\preceq$) in the global query tree, the container algorithm proceeds from node $//b^1$ in the global query tree to match the children of $//b^1$ (node e^1, d, and c^1) (line 23).

Next, the container algorithm recursively processes node e^1 in the global query tree by calling *container*(e^1). Node $//b^1$.*solution* is ([2:7],2) and the current entry in the label list associated with node e^1 in the global query tree is ([9:10],3). Hence, there is no parent/child relationship between the two nodes. The query node $//b$ in the new query with region code ([2:7],2) does not have e as its child and e^1.*sub*() is returned with a list of {2,4} (line 25) as the result of *container*(e^1). In addition, for the other two child of $//b^1$, *container*(d)= $\emptyset$ and *container*(c^1)=$\emptyset$. Therefore, the union of container results for all children of $//b^1$ is {2,4} when $C_{//b^1}$ is ([2:7],2) according to line 29.

There is one remaining element in the label list associated with node $//b^1$ in the global query tree; that is the second b node with region code ([8:11],2) in the new query u. The container algorithm continues to process it. The ancestor/descendant relationship holds for the second element $//b$ with region code ([8:11],2) and node a with region code ([1:12],1) in the new query. After identifying the node with region code $([8:11],2) \preceq //b^1$, the algorithm expands node $//b^1$ again to recursively match the children (node e^1, d, and c^1) by calling *container*(e^1), *container*(d) and *container*(c^1), respectively. The current elements C_{e^1}, C_d, C_{c^1} in the e-list, d-list and c-list are ([9:10],3), ([5:6],3) and ([3:4],3) in Figure 4.11, respectively. The parent/child relationship exists between node $//b$ with region code ([8:11],2) and node e with region code ([9:10],3). The parent/child relationship does not exist between node $//b$ with region code ([8:11],2) and node d with region code ([5:6],3). Similarly,

the parent/child relationship does not exist between $//b$ with region code ([8:11],2) and node e with region code ([3:4],3) in the new query. Node $//b$ in the new query with code ([8:11],2) does not have d and c as its children. Therefore, nodes d and c^1 of the global query tree do not have mapped nodes in the new query when b^1 in the global query tree is mapped to node $//b$ with region code ([8:11],2) in the new query as shown in Figure 4.12. $d.sub()$ and $c^1.sub()$ are returned. Hence, $container(e^1)=\emptyset$, $container(d)=\{3, 4\}$ and $container(c^1)=\{4\}$. According to line 29, the union of the container results for all children of $//b^1$ is $\{3, 4\}$ when $C_{//b^1}$ is ([8:11],2). In the end, according to line 29, the $container(//b^1) = \{2, 4\} \cap \{3, 4\} = \{4\}$.

Similarly, $container(//c) = \emptyset$. For the 2^{nd} child node $//b^2$ of node a in the global query tree, the element ([2:7],2) is a matching node of node $//b^2$ in the global query tree, so the container algorithm expands to match $//b^2$'s child node e^2 in the global query tree. The element ([2:7],2) does not have e as its child because code ([2:7],2) and code ([9:10],3) have no parent/child relationship, so the $e^2.sub()$ is returned and the result is $\{3\}$. The other element ([8:11],2) in the b-list is a matching node of $//b^2$ node and has an e child node. An empty set is returned (line 24 of Algorithm 11) for ([8:11],2). Hence, the $container(//b^2)$ is $\emptyset$ ($\{3\} \cap \emptyset$). Furthermore, the result of the 3^{rd} $//b^3$ in the children of a $container(//b^3)$ is $\emptyset$. Line 10 leads to $container(a)=\{4\}$. So query q_4 cannot contain the new query u. The queries $\{q_1, q_2, q_3\}$ contain the query u.

In the implementation, there is a pointer to the parent node for each node t in the global query tree. This pointer is used to access the membership *solution* of the parent node $t.parent$ of t. The variable $t.parent.solution$ is used to determine the ancestor/descendant and parent/child relationships with the C_t in the label list associated with node t together.

The container algorithm in the proposed new algorithm uses the region encoding to evaluate the ancestor/descendant or parent/child relationship between the query nodes. Further, at the same time, the label lists can reduce the search space to a

smaller range. On the other hand, the XSearch algorithm has to map a //-node to paths of $length = 0$ and $length \geq 1$. For example, in Figure 4.12, to find the mapping of a path /a//c in the global query tree to the new query, two comparisons (a and c) are needed by the new algorithm that directly examines the label list a of {[1:12],1} and list c of {[3:4], 3}. With the XSearch algorithm, five comparisons ($//b$, c, d, $//b$ and e) are needed because of the ancestor/descendant operator ($//$) of c.

4.2.5 Label maintenance for dynamic query updates

The discussions of the containee and container algorithms focus on the scenario of existing static queries. In a dynamic scenario, queries can be added or removed. When a query is added or removed, the query index tree structure changes and the region encoding (or index) for nodes in the tree needs to be updated as well. Dynamic query updates affect the containee algorithm, but not the container algorithm, because the containee algorithm makes use of the region codes for nodes in the global query tree.

In order to handle query additions or removals, one approach is to reserve some labeling gap between the parent node and the child node in order to reduce the number of relabeling operations. For example, in Figure 4.12, the region code for the left node c^1 at level three is [8:9]. Node c's parent, $//b$, could have a larger right value than $9 + 1 = 10$ for future node additions. In other words, the region code of node $//b$ can be [8:15] instead of [8:10], for example, to accommodate more child nodes.

In [98], the authors investigate the existing XML labeling schemes and their support for dynamic updates. Some schemes that deal with dynamic query updates can be used together with our proposed algorithm. For example, the approach discussed in [108] deals with a nested tree, which can reduce the number of relabeling operations and supports XML data updates. The labeling format for a node in a nested tree is $[prefix{:}localPosition]$. Hence, elements that are to be added later can use the region code ($[prefix : left\ ,\ prefix : right]$, $level$), instead of re-encoding the nodes. For example, in Figure 4.12, the region code for the left node c^1 at level three is [8:9].

When a new tree node t is to be added under c^1, the region code for node c and the new node t can be ([8:1, 8:4],3) and ([8:2, 8:3],4), for example. The aggregation algorithm described in this thesis can simply make use of other label update algorithms to support dynamic label updates.

4.2.6 Complexity analysis

This section presents the time complexity and space complexity for both the containee and container algorithms.

Time complexity. The containee algorithm checks each element in the label lists T_q using *getNext*() in Algorithm 7. For the containee algorithm, there are two cases for the aggregation.

Case 1: there is no "//". For this case, the complexity of the containee algorithm is O(N), where N is the sum of the number of entries in the label lists that are associated with the new query. For this case, each corresponding label list will be searched to find the matched entry; hence, the time complexity is O(N).

Case 2: there is at least one "//". For case 2, the time complexity of the new containee algorithm is still O(N) for the same reason.

For the new container algorithm, the complexity is O($|T(R)|$), where $|T(R)|$ is the number of nodes in the global query tree. The container algorithm traverses the global query tree in pre-order and compares each node in the tree with the corresponding label list based on the new query.

The time complexity of both algorithms (containee and container) of XSearch is O($|s| \times |T(R)|$), where $|s|$ is the number of nodes in a new query and $|T(R)|$ is the number of nodes in the factorization tree [42]. The factorization tree is similar to the global tree in our algorithm. In XSearch, each pair of nodes in T(R) and s are checked at most once. For example, the XSearch algorithm has to check all the descendent nodes under the node with the ancestor-descendant operator (//). In this case, the number of nodes that needs to be compared can be close to $|T(R)|$ if the

node associated with "//" is high in the tree.

Space complexity. The space complexity of the containee algorithm is $O(|T(R)|+|L|)$, where $|T(R)|$ is the number of nodes of the global query tree and $|L|$ is the size of label lists for region codes of the global query tree. The space complexity for the container algorithm is $O(|T(R)| + |l|)$, where $|l|$ is the size of label lists for region codes of the new query. However, the space complexity for the XSearch approach requires $O(|T(R)|)$ for both containee and container algorithms, where $|T(R)|$ is the number of nodes in the factorization tree [42].

4.3 Experimental evaluation

The performance of the XSearch algorithm and the proposed containee and container algorithms are evaluated in this section. XPath queries are generated using the XPath query generator of Yfitler [53]. Yfilter is a prototype developed for filtering XML messages against XPath queries.

Performance metrics for evaluating the XSearch and the new aggregation algorithm include the following:

(i) Processing time for the containee and container algorithms;

(ii) Parsing time for XPath queries and building the global query tree;

(iii) Building time for build the label list for region codes; and

(iv) The space complexity for *nitf* experiments.

The processing time for the containee algorithm is measured between the end of the algorithm and the beginning of the algorithm. The processing time for the container algorithm is measured between the end of the algorithm and the beginning of assigning the region codes to the new query. The processing times for the containee and container algorithms and the XSearch algorithm are compared. The total processing time is the sum of the processing times of the containee and container

algorithms. This is the total processing time incurred for aggregating a new query into the system.

All the processing times presented in this section are the average value over 20 runs. In order to exclude the effect of the JVM garbage collection, the system call for garbage collection is explicitly invoked before each measurement. Before the performance evaluation, we first warm up the JVM and the CPU to mitigate the effect of cache faults and JVM warm up times.

The experiments are conducted on a system consisting of two 3.0 GHz Intel Pentium cores with 2.0 GB of RAM running under Windows XP. The results of these experiments are presented in the following subsections.

To determine the parameters to be used in the performance evaluation, the parameter values used in XSearch are referred [42]. In [42], the parameter values for the experiments evaluating the efficiency of the XSearch algorithm are listed as follows.

1. the maximum query depth: 10
2. prob(//)=0.05
3. the probability of having more than one child at a given node is 0.1
4. the number of queries is varied between 1000 and 100,000.

In addition, to measure the impact of prob(//) and prob(branching), values of prob(//) and prob(branching) are varied in the interval [0, 0.2] by steps of 0.05.

4.3.1 Experiments with *nitf* queries

Test queries used in this section are *nitf* queries that are generated based on News Industry Text Format (NITF) *Nitf.dtd*. The *nitf.dtd* is widely used in XML pub/sub systems [6]. Six queries that are randomly selected are used in the experiments and they are listed in Table 4.1. Q1, Q2 and Q3 are linear path queries with only one branch. There is one ancestor/descendant operator (//) in Q3. Q4, Q5 and Q6 are twig queries with two branches ([ ]). There is no ancestor/descendant operator

(//) in Q4 while there are two ancestor/descendant operators (//) in Q5. Q6 is a complex query, consisting of both an ancestor/descendant operator (//) and a twig ([ ]) operator. In addition, the depth of Q6 is five which is higher than the general depth of the XML queries. These queries shown in Table 4.1 are used in Sections 4.3.1.1, 4.3.1.2 and 4.3.6.

Table 4.1: The *nitf* queries to be tested in experiments

	Query content
Q1	/nitf/body/body.content
Q2	/nitf/head
Q3	/nitf//head
Q4	/nitf[body/body.content]/head
Q5	/nitf[body//body.content]//head
Q6	/nitf[body/body.content//hr]/head/docdata/doc-scope/xt

4.3.1.1 Processing time versus the number of existing *nitf* queries

The parameters of XPath queries are: probability(//)=20%, probability(*)=0, the number of branches=2, and the query length=6. The number of a given set of existing XPath queries varies from 100, 500, 1000, 2000, 3000, 4000, to 5000. Duplicated queries are allowed in the existing query set. The results of the processing time for Q1-Q6 are shown from Figure 4.13 to Figure 4.18.

Q1 and Q2 are linear path queries with parent/child operator only and do not have ancestor/descendant operators (//). In Figure 4.13 and Figure 4.14, the number of queries varies from 100, 500, 1000, 2000, 3000, 4000 to 5000. For Q1, the total processing time based on the XSearch algorithm is at least 1.1 × that for the new algorithm. It is observed to be as high as 1.8 × of the total processing time for the new algorithm. The discrepancy of the processing time between the new containee algorithm and the XSearch algorithm ranges from 56.5% to 81.1% (except when N is 100), and that for the container algorithm, from 15.5% to 34.7%. When N is 100, for Q1, the processing time for the new containee approach is 0.059 ms, while the processing time for the XSearch algorithm is 0.056 ms.

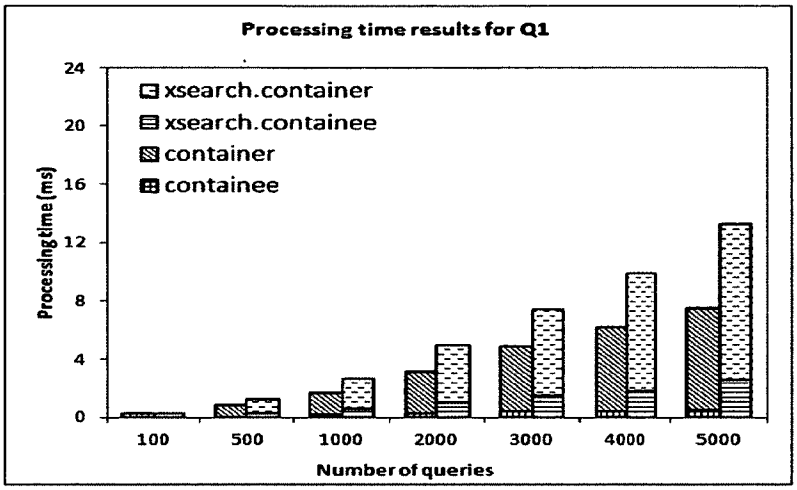


Figure 4.13: Processing time results for Q1

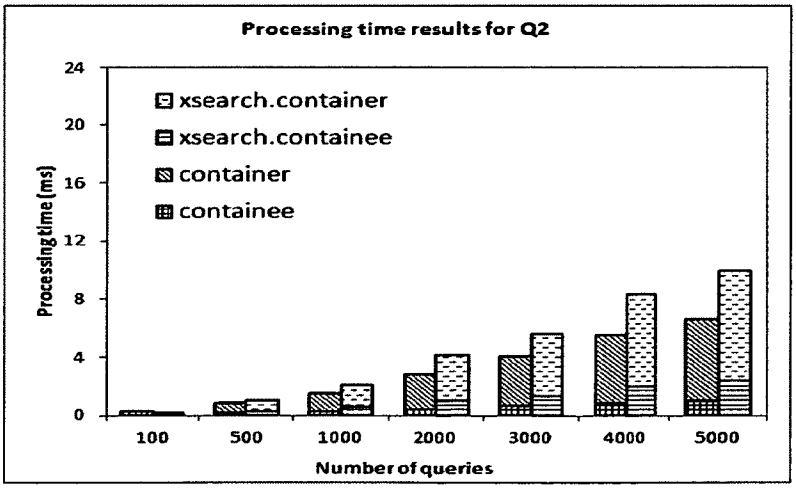


Figure 4.14: Processing time results for Q2

For Q2, when N is 100, the total processing time of the new aggregation approach, which is 0.242 ms, is a little longer than that of the XSearch algorithm which is 0.173 ms in this case. Except for the result when N is 100, for Q2, the total processing time for the XSearch algorithm is 1.1 × to 1.5 × of the total processing time for the new algorithm. The results for the containee algorithm show that the processing time for the new containee algorithm is lower than that of the XSearch by 43.9% to 60.1% (except when N is 100). The processing time for the new container algorithm is lower than that for the XSearch algorithm by 18.8% to 25.5% (except when N is 100).

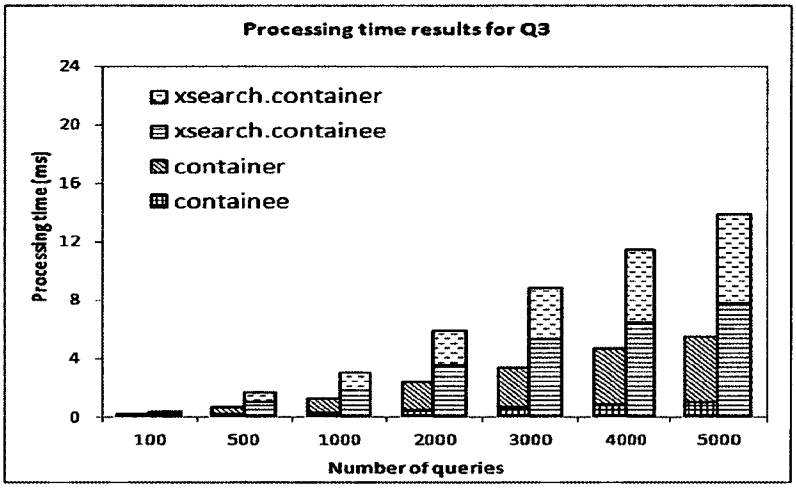


Figure 4.15: Processing time results for Q3

Q3 is a query with ancestor/descendant operators (//) and is a linear path query. For a query with n nodes and only ancestor/descendant operator (//), the new algorithm shows significant improvement over XSearch. The total processing time for the XSearch can be up to 2.7 × that of the new algorithm. The results for the containee algorithm for Q3 show that the processing time for the new containee algorithm is lower than that of the XSearch by 84.4% to 87.6%. For example, for Q3 in Figure 4.15, when N is 5000, the processing time for the containee algorithm for the new algorithm is 0.964 ms, whereas the processing time of the containee algorithm based on the XSearch algorithm is 7.746 ms, which is about eight times that of the new algorithm. Even when N is 100, the processing time for the containee in the XSearch

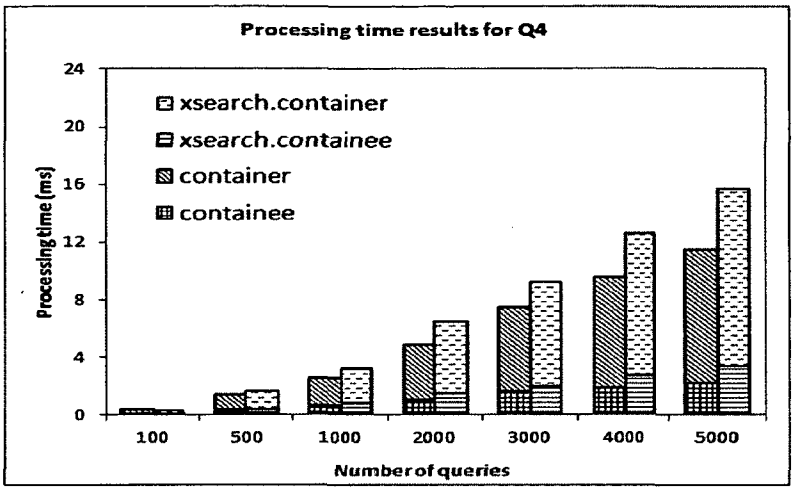


Figure 4.16: Processing time results for Q4

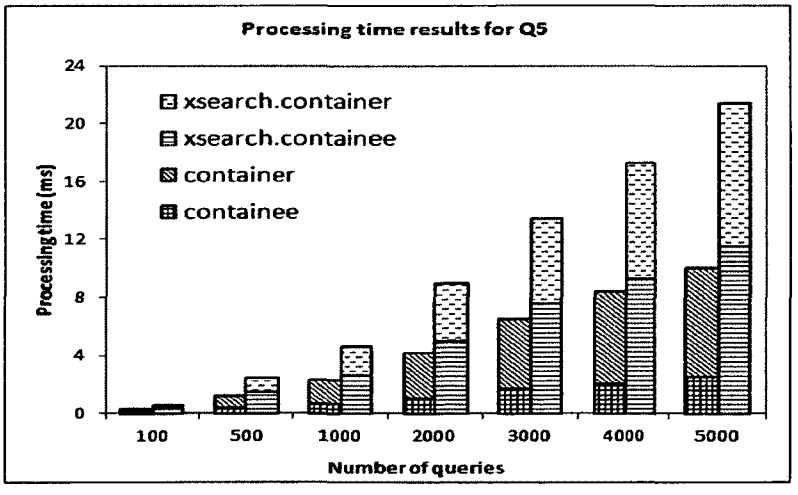


Figure 4.17: Processing time results for Q5

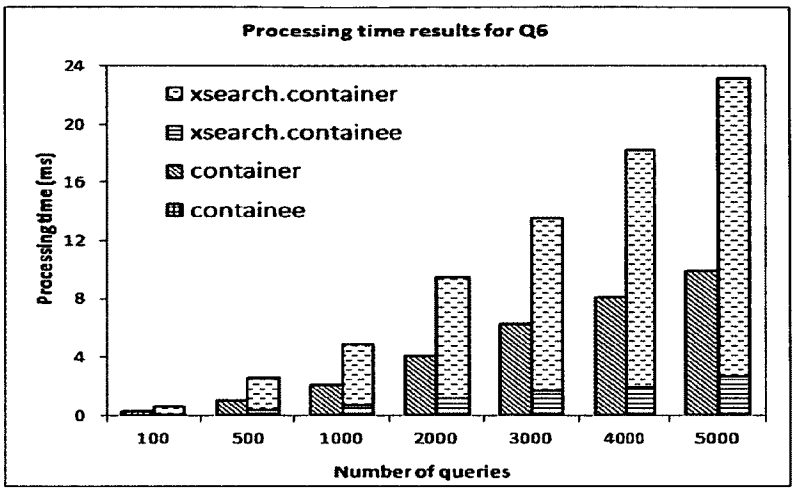


Figure 4.18: Processing time results for Q6

algorithm is 0.277 ms, while the processing time for the new containee algorithm is 0.061 ms. The processing time for the new container algorithm is lower than that of the XSearch by 14.5% to 25.6% (except in the case when N is 100). When N is 100, the new container algorithm for Q3 is higher than that of the XSearch, which are 0.134 ms and 0.1 ms, respectively.

Q5 is a query with ancestor/descendant operators (//) and is a twig query. The total processing time of the XSearch algorithm is observed to be up to 2.1 $\times$ that of the new algorithm. In Figure 4.17, when N is 5000, the processing time of the containee algorithm is 2.525 ms for the new algorithm, whereas it is 11.533 ms for the XSearch algorithm. Further, the processing time of the new containee algorithm remains low when N increases, and that is because the new containee algorithm only searches two lists *nitf* and *head*, instead of visiting the global query tree based on the XSearch algorithm. Meanwhile, the results for the new containee algorithm for Q5 show that the processing time for the new containee algorithm is lower than that of the XSearch by 60.6% to 78.1%. The processing time for the new container algorithm is lower than that of the XSearch by 4.5% to 23.2%.

Q4 is a twig query and there is no ancestor/descendant operator (//). As shown in Figure 4.16, the processing time of the new container algorithm is lower than that of

the XSearch algorithm. Although the new algorithm is suboptimal for parent/child operator (/), the performance of the new algorithm still outperforms the XSearch algorithm. The total processing time for the XSearch algorithm is 1.2 × to 1.4 × that of the new algorithm (except when N is 100). When N is 100, the total processing time for the XSearch is 0.307 ms, and 0.382 ms for the new algorithm, respectively. The difference is small for a small number of queries in the existing query tree. Except in the case when N is 100, the improvement produced by the new containee algorithm ranges from 4.7% to 34.6%, whereas the improvement produced by the new container algorithm ranges from 18.5% to 24.9%.

Q6 is a query with a new node name which is not present in the global query tree. The total processing time for the XSearch algorithm ranges from 2.2 × to 2.4 ×. As illustrated in Figure 4.18 for Q6, the processing time of the new containee algorithm is small, around 0.025 ms, and that is because a pre-processing operation is applied. The pre-processing operation uses a name pool to gather all labels present in the existing queries. If there is a new label present in the new query being tested, then the new query cannot cover any existing queries. The XSearch algorithm can be improved using this pre-processing operation as well. The current XSearch algorithm does not have this pre-processing operation. Hence, the processing time of the containee method for XSearch is significantly higher than that of the new containee algorithm. The processing time for the new container algorithm and the XSearch algorithm has a difference of 48.8% to 51.7% for Q6.

For Q1, Q2 and Q4, the processing times for the containee algorithm are smaller than the times for computing the results for the container algorithm. For the containee algorithm, there is only the parent/child operator (/) in Q1, Q2 and Q4; hence, the algorithm only needs to iterate all the child nodes of a node in the global query tree, instead of all the descendant nodes. To compute the containee results, each node of the new query should be mapped to nodes in the global query tree. On the other hand, to compute the container results, each node in the global query tree should

be mapped to the new query tree. Since the size of the global query tree is much larger than that of a new query, more node comparison operations are required for the container results. Hence, the processing time for the containee algorithm is shorter than that for the container algorithm.

For Q3 and Q5, the processing time for the containee results is higher than the processing time for Q1, Q2, and Q4. This is because there is at least one ancestor/descendant (//) operator in them. The ancestor/descendant operator (//) requires visiting all nodes of a subtree, which takes a longer time.

4.3.1.2 Processing time versus the number of branches in *nitf* queries

The purpose of this experiment is to evaluate the change in processing time when the complexity of existing queries is increased. The number of branches of queries in the global query tree is varied from 2 to 4. In this set of experiments, the total number of existing *nitf* queries is fixed at 3000. Table 4.2 shows three example queries used in the testing with 2, 3 and 4 branches, respectively. For example, the query (/nitf[body//bibliography]/head[title]/meta) has three branches: /nitf/body//bibliography, /nitf/head/title, and /nitf/head/meta. The new queries to be aggregated are described in Table 4.1 which are the same as the queries used in Section 4.3.1.1.

Table 4.2: Example queries with 2, 3 and 4 branches

Number of branches	Query content
2	/nitf/head[meta]/title
3	/nitf[body//bibliography]/head[title]/meta
4	/nitf[head/pubdata]/body[body.head//location//state] /body.content/p[q/pronounce]/person/alt-code

The processing times for the containee and container algorithms for Q1 to Q6 are presented in Figures 4.19, 4.20 and 4.21. The new algorithm is observed to outperform the XSearch algorithm.

In Figure 4.19, all existing queries in the global query tree have two branches. Q4

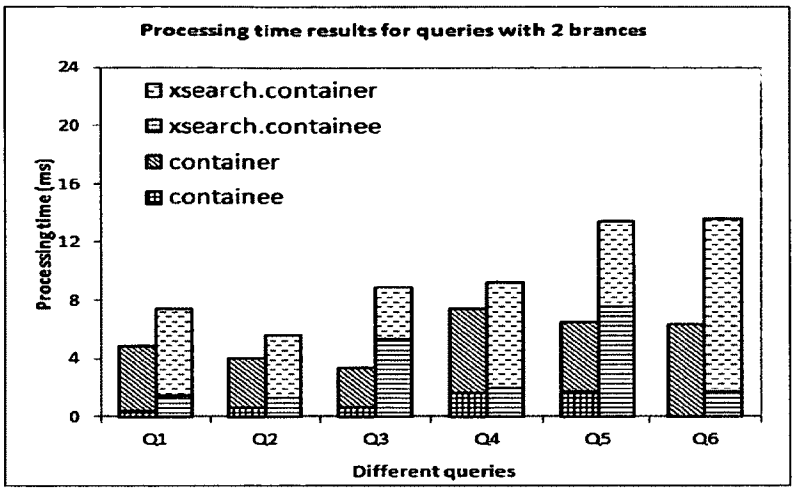


Figure 4.19: Processing time results for queries with 2 branches

gives rise to the minimum performance improvement: the total processing time for the XSearch algorithm is 1.2 × that of the new algorithm. Q3, Q5 and Q6 achieve the maximum performance improvement in terms of the total processing time. The total processing time using XSearch is 2.7 ×, 2.1 × and 2.2 × that of the new algorithm, respectively. The total processing times using XSearch for Q1 and Q2 are in between: 1.5 × and 1.4 × that of the new algorithm. For queries from Q1 to Q6, the processing times for the new containee algorithm are lower than that of XSearch by 75.9%, 52.2%, 87.9%, 17.8%, 77.3% and 97.3%, respectively. The processing times for the new container algorithm are lower than that of the XSearch by 24.6%, 20.3%, 23.2%, 19.8%, 17.3% and 47.3%, respectively.

In Figure 4.20, all existing queries in the global query tree have three branches. Q4 achieves the minimum performance improvement and the total processing time for XSearch algorithm is 1.2 × that of the new algorithm. Q3, Q5 and Q6 achieve the maximum performance improvement in terms of the total processing time: the total processing times are 2.7 ×, 2.2 × and 2.0 × that of the new algorithm, respectively. The total processing times for XSearch for Q1 and Q2 are in between: which are 1.6 × and 1.4 × that of XSearch. For queries from Q1 to Q6, the processing times for the new containee algorithm are lower than that of XSearch by 78.6%, 58.6%, 88.9%,

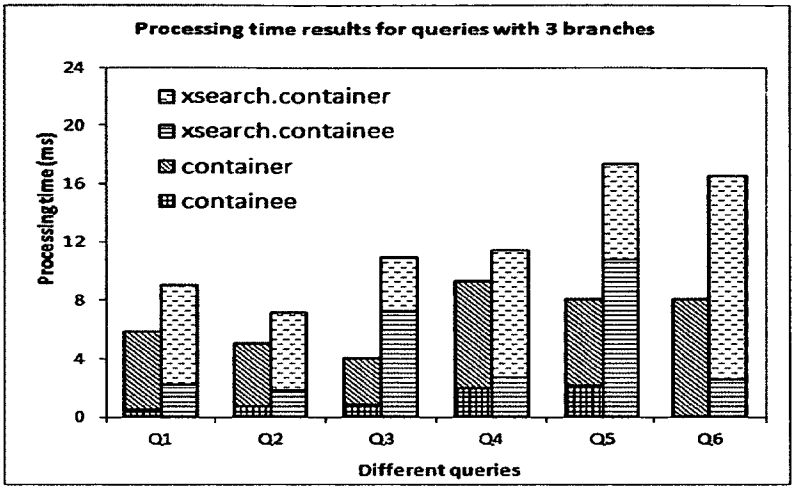


Figure 4.20: Processing time results for queries with 3 branches

28.5%, 80% and 98.2%, respectively. The processing times for the new container algorithm are lower than that of XSearch by 22.9%, 20.2%, 12.6%, 15.9%, 11.3% and 42.8%, respectively.

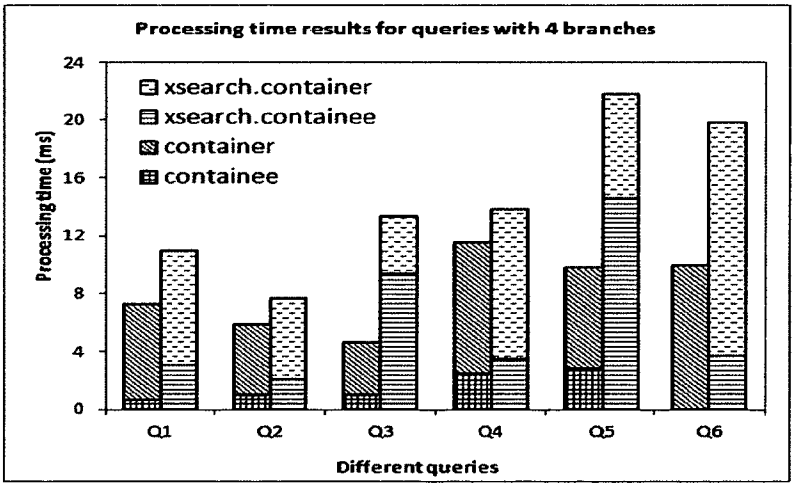


Figure 4.21: Processing time results for queries with 4 branches

In Figure 4.21, all existing queries in the global query tree have four branches. Q4 achieves the minimum performance improvement and the total processing time for XSearch is 1.2 × that of the new algorithm. Q3, Q5 and Q6 give rise to the maximum improvement in terms of the total processing time. The total processing times for

XSearch for Q3, Q5 and Q6 are 2.3 ×, 2.2 × and 2.0 × that of the new algorithm, respectively. For Q1 and Q2, the total processing times are in between: the total processing times for XSearch are 1.5 × and 1.3 × that of the new algorithm. For queries from Q1 to Q6, the processing times for the new containee algorithm are lower than that of XSearch by 78.9%, 54.9%, 89.9%, 30.3%, 81% and 98.9%, respectively. The processing times for the new container algorithm are lower than that of XSearch by 16.9%, 12.7%, 7.4%, 11.7%, 2.5% and 38.6%, respectively.

4.3.2 Experiments with *bib* queries

To test the effects of the new algorithms, a much simpler DTD is used to generate XPath queries. *Bib* queries are generated based on *bib.dtd* which is a simple DTD which has three levels and eleven elements [7]. There are eight *bib* queries used in the experiment (see Table 4.3). The total number of existing queries is 1000. Other parameters of the XPath queries are: probability(//)=20%, probability(*)=0, # of branches is 2.

Table 4.3: The *bib* queries used in experiments

	Query content
Q1	/bib/vendor/book/author
Q2	/bib/vendor/book
Q3	/bib/vendor[name]/book[title][author]
Q4	/bib/vendor//author
Q5	/bib/vendor[name]//book[title][author]
Q6	/bib/vendor[.//name]//book[title][author]
Q7	/bib/vendor[.//name]//book[.//title][author]
Q8	/bib/vendor[.//name]//book[.//title][.//author]

From the results shown in Figure 4.22, the performance of the new algorithm from Q1 to Q8 are observed to be superior to that of the XSearch algorithm. Q1 gives rise to the minimum performance improvement and the total processing time for XSearch is 1.2 × that of the new algorithm. Q8 achieves the maximum performance improvement and the total processing time for XSearch is 1.6 × that of the new algorithm. The

ratios of the total processing times for the XSearch over that of the new algorithm for other queries including Q2, Q3, Q4, Q5, Q6 and Q7 are in between. Moreover, for queries from Q1 to Q8, the processing times for the new containee algorithm are lower than that of XSearch by 8.9%, 73.3%, 3.5%, 61.2%, 35.9%, 47.5%, 59.6% and 70.2% respectively. The processing times for the new container algorithm are lower than that of XSearch by 19.6%, 8.4%, 27.9%, 21.8%, 26.2%, 19.1%, 20.5% and 17.7%, respectively.

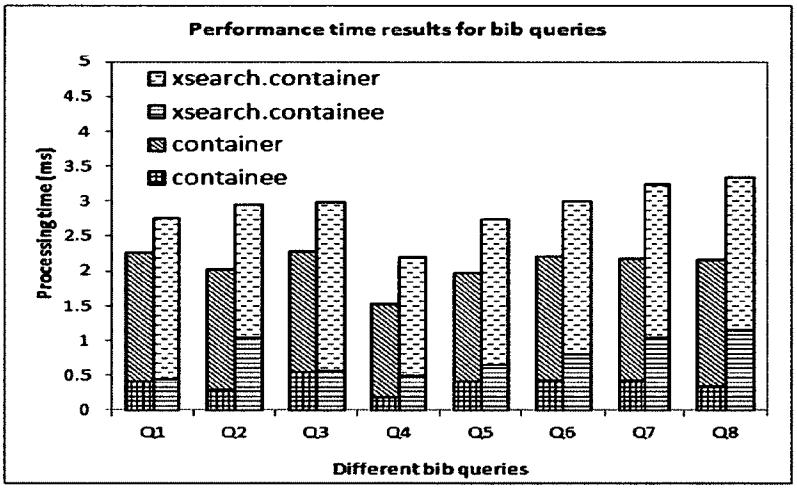


Figure 4.22: Processing time results for for *bib* queries

4.3.3 Parsing time for XPath queries and building time for the global query tree

Table 4.4 lists the times used for parsing XPath queries and building the global query tree by the new algorithm and XSearch. The number of existing queries are varied from 100 to 100,000. An XPath query is first parsed by a Yfilter query parser. The outputs of the Yfilter query parser are separated branches. A wrapper class called *XPathTree* is used to construct an internal tree format for an XPath query. Then, parsed *XPathTrees* are added to the global query tree. The parsing and building time starts when the first query is parsed and ends when the last query is added to the global query tree. Based on the results presented in Table 4.4, we can see that the

costs of both algorithms are close to each other.

Table 4.4: Time for parsing XPath queries and building the global query tree using the XSearch and the proposed new algorithms

N	XSearch (ms)	The new algorithm (ms)
100	103.42	103.33
500	217.08	212.44
1000	298.77	295.35
2000	463.82	469.46
3000	638.25	627.72
4000	770.76	776.99
5000	947.39	950.66
10,000	1734.29	1729.47
20,000	3436.86	3430.69
40,000	6309.87	6384.03
60,000	8420.67	8388.55
80,000	10677.35	10752.10
100,000	12842.37	12852.77

4.3.4 Building time for region codes and label lists

Table 4.5 lists the time used to build region code and to create label list for the global query tree. N is the number of existing queries. Based on the processing time for parsing XPath queries and building the global query tree for the proposed algorithm (see Table 4.4) and Table 4.5, we can observe that the ratio between the time for building region codes for nodes in a global query tree and the time for building a global query tree is in a range from 1% to 2.17%. Figure 4.23 depicts the pre-processing (building) time of the XSearch and the proposed algorithms as a function of query populations.

Table 4.5: Time for encoding and building label list for the global query tree using the proposed algorithm

N	100	500	1000	2000	3000	4000	5000
Time(ms)	1.05	3.46	6.41	10.51	14.16	17.80	20.86
N	10,000	20,000	40,000	60,000	80,000	100,000	
Time(ms)	41.00	68.74	124.38	175.91	226.67	278.10	

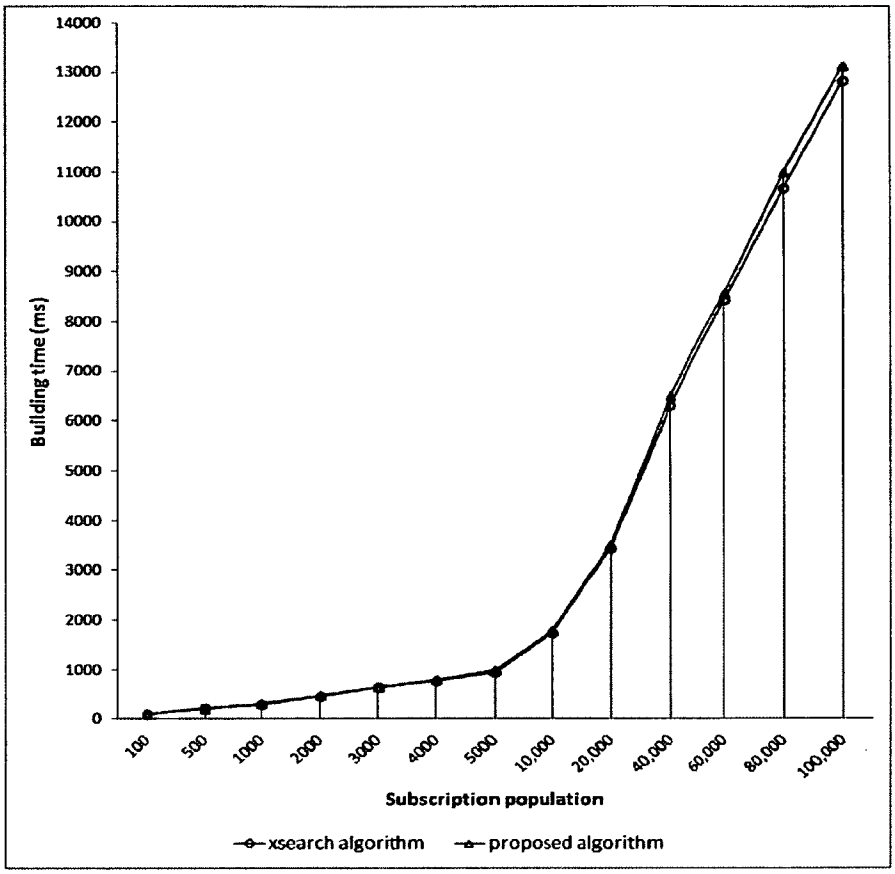


Figure 4.23: The preprocessing time relationship between the XSearch and the proposed algorithms

4.3.5 Space complexity for *nitf* experiments

We have analyzed and examined the space requirements for the XSearch and the proposed algorithms. For the XSearch algorithm, the space requirements are the global query index tree space cost; while for the proposed algorithm, the space requirements include the global query tree space cost and the linked list cost. The space cost of the global query index tree is measured as the total number of nodes in the tree. The space cost of the linked list is the total number of nodes in a global query tree times the space cost of a region code object which includes *left*, *right*, *level*, the set of query IDs, and the set of leaf IDs. Table 4.6 shows the total number of nodes in a global query tree using *nitf* queries. In Table 4.6, the heights for all the global query trees

are 7, including the dummy root node r_T (see the descritpion for *nitf* XPath query parameters in Section 4.3.1.1).

Table 4.6: The total number of nodes in a global query index tree for a given query population N for both algorithms

N	100	500	1000	2000	3000	4000	5000
total number of nodes	263	864	1418	2327	3111	3632	4285
N	10,000	20,000	40,000	60,000	80,000	100,000	
total number of nodes	6,629	10,328	15,685	19,678	23,415	26,474	

4.3.6 Processing time for a very large number of queries

The purpose of this section is to examine the processing time of the new algorithms when the number of existing queries is very large. Queries shown in Table 4.1 are used as new queries to be aggregated. The parameters for generating the existing queries are: query path length is 6, probability(//)=20%, and number of branches is 2. The number of existing queries N varies from 10,000, 20,000, 40,000, 60,000, 80,000 to 100,000. Existing queries are unique. The average processing time for containee, container, and total is the mean processing times over 20 experimental runs.

Results are presented in Figure 4.24 to Figure 4.29. The results show that the new algorithm outperforms the XSearch algorithm for very large number of queries. For example, Q3, Q5 and Q6 achieve the maximum performance gain. Q4 achieves the minimum performance gain. Q1 and Q2 are in between.

Q1 (/nitf/body/body.content) represents a class of linear path queries and there is no ancestor/descendant operator (//) in it. For Q1, the maximum performance gain is obtained when N is 10,000. The total processing time for the XSearch algorithm is 1.7 $\times$ that of the new algorithm; the minimum performance improvement is obtained when N is 100,000 and the total processing time for the XSearch algorithm is 1.6 $\times$ that of the new algorithm. For Q1, the results for the containee algorithm show that the processing times for the new containee algorithm are lower than that of XSearch from 82.1% to 84.2%. The processing times for the new container algorithm are lower

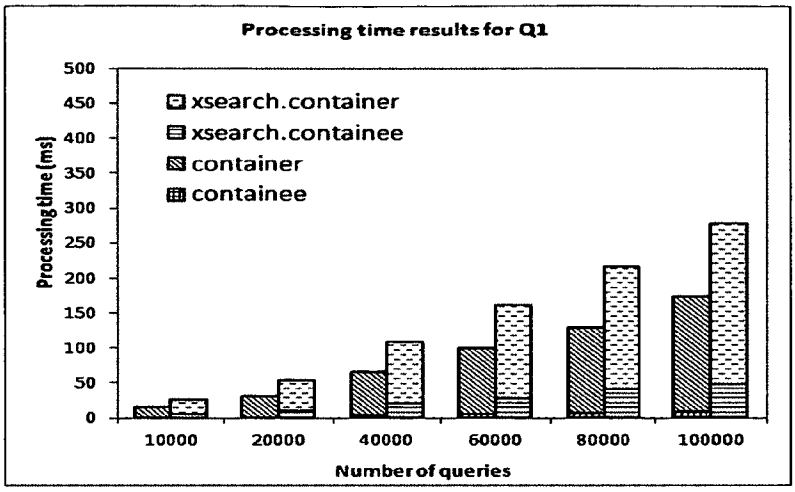


Figure 4.24: The processing time results for Q1 when N is very large

than that of XSearch from 27.8% to 31.7%.

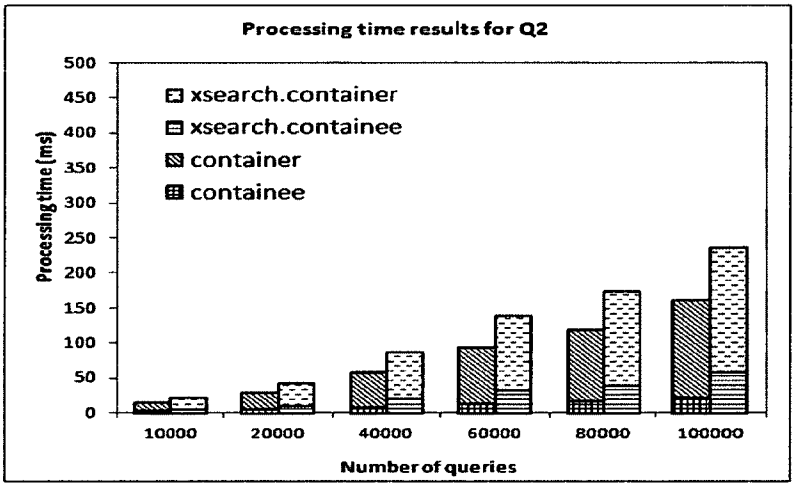


Figure 4.25: The processing time results for Q2 when N is very large

Q2 (/nitf/head) is a linear path query with no ancestor/descendant operator (//). For Q2, the maximum performance improvement is achieved when N is 20,000 and the total processing time for XSearch is 1.5 $\times$ that of the new algorithm; the minimum performance improvement is achieved when N is 100,000 and the total processing time for XSearch is 1.5 $\times$ that of the new algorithm. For Q2, the results for the containee algorithm show that the processing times for the new containee algorithm

are lower than that of XSearch by 56.9% to 63.3%. The processing times for the new container algorithm are lower than that of XSearch by 21.7% to 25.9%.

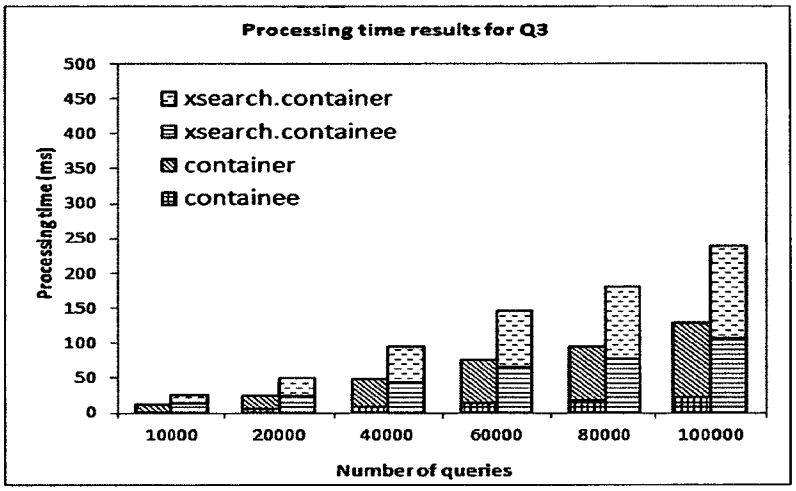


Figure 4.26: The processing time results for Q3 when N is very large

Q3 (/nitf//head) has an ancestor/descendant operator (//) and Q3 is a linear path query. Because of the operator (//), the XSearch algorithm requires more comparisons than the new algorithm. For example, when N is 10,000, to compute Q3, the total processing time for the XSearch algorithm is 2.26 $\times$ that of the new algorithm. For Q3, the results for the containee algorithm show that the processing times for the new containee algorithm are lower than that of XSearch by 78.8% to 85.2%. The processing times for the new container algorithm are lower than that of XSearch by 19.4% to 25.3%.

Q4 (/nitf[body/body.content]/head) is a twig query without ancestor/descendant operator (//). The maximum performance improvement is achieved when N is 10,000 and the total processing time for the XSearch algorithm is 1.4 $\times$ that of the new algorithm. The minimum performance improvement occurs when N is 100,000 and the total processing time for the XSearch algorithm is 1.3 $\times$ that of the new algorithm. Moreover, for Q4, the results for the containee algorithm show that the processing times for the new containee algorithm are lower than that of XSearch by 32.4% to 35.2%. The processing times for the new container algorithm are lower than that of

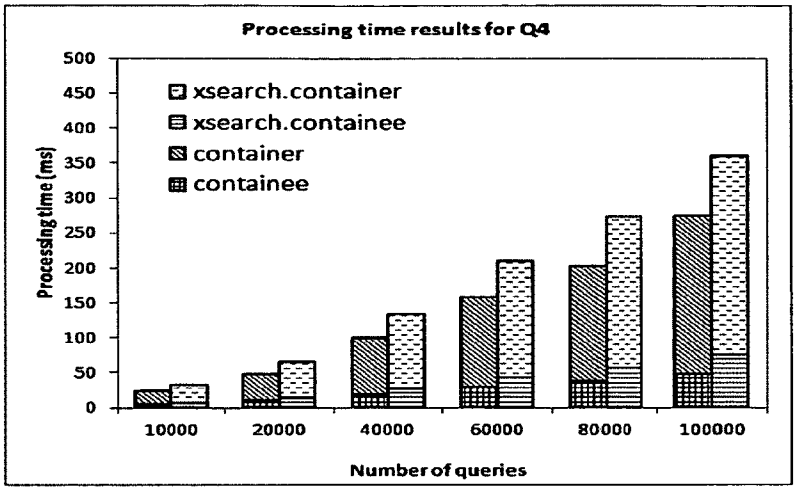


Figure 4.27: The processing time results for Q4 when N is very large

XSearch by 21.0% to 25.2%.

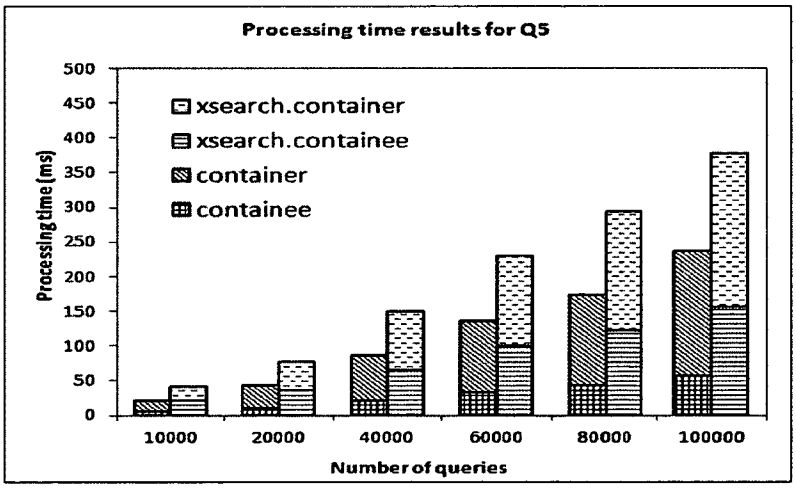


Figure 4.28: The processing time results for Q5 when N is very large

Q5 (/nitf[body//body.content]//head) has the ancestor/descendant operator (//) and Q5 is a twig query. To compute Q5, when N is 10,000, the total processing time using the XSearch algorithm is 1.9 × that of the new algorithm. Even when N is much larger, for example when N is 100,000, the total processing time for Q5 using the XSearch algorithm is 1.9 × that of the new algorithm. The total processing time for Q5 based on the XSearch algorithm is 1.6 × that of the new algorithm. For

Q5, the results for the containee algorithm show that the processing times for the new containee algorithm are lower than that of XSearch by 63.9% to 74.2%. The processing times for the new container algorithm are lower than that of XSearch by 18.6% to 24.1%.

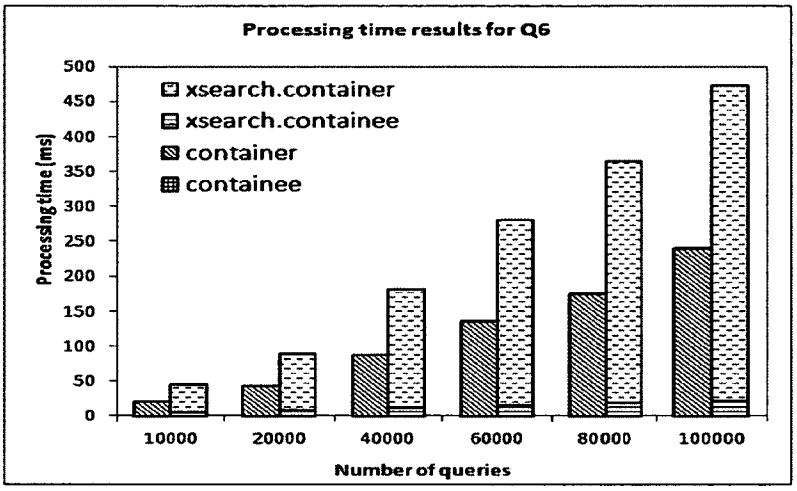


Figure 4.29: The processing time results for Q6 when N is very large

Q6 (/nitf[body/body.content//hr]/head/docdata/doc-scope/xt) is a twig query with the ancestor/descendant operator (//). Q6 contains a new tag name which is not present in the existing queries. The total processing time for the new algorithm outperforms that of the XSearch algorithm. For example, the maximum performance improvement is achieved when N is 10,000 and the total processing time for XSearch is 2.2 $\times$ that of the new algorithm. The minimum performance improvement occurs when N is 100,000 and the total processing time for XSearch is 2.0 $\times$ that of the new algorithm. In addition, for Q6, the results for the containee algorithm show that the processing times for the new containee algorithm are lower than that of XSearch by 98.9% to 99.7%. The processing times for the new container algorithm are lower than that of XSearch by 47.7% to 49.8%.

4.4 Existing XPath query aggregation approaches

XNET is an XML-based pub/sub system for a wired network scenario. Chand et al. [42] propose an XPath query aggregation algorithm, called XSearch, to reduce the transmission of a large number of XPath queries at an XML router. XSearch builds a factorization tree to share common prefixes among queries that is similar to the global query tree in the our new approach. The containment relationships between the new query and the existing queries are identified by homomorphism mapping. Figure 4.30 shows a set of queries (P_1, P_2, P_3, P_4 and P_5), a new XPath query P_{new} and a query index tree constructed from P_1 to P_5. For instance, the set under each node name, e.g., {1, 2, 3, 4, 5} under node a, represents the set of queries associated with the node. For an XPath query $//t$, the XSearch algorithm uses two nodes to represent it. One node is for the ancestor/descendant operator (//) and the other node is for node t.

Yoo et al. [107] propose another XPath query aggregation algorithm. A partially ordered set for XPath queries is constructed using the idea of homomorphism. The definition of homomorphism is explained in Definition 2 of Chapter 2. Twig (tree-structured) queries are decomposed into paths/branches. A query index tree is built to share common prefixes among query paths/branches. Homomorphism mapping is checked between branches. A prefix-sharing tree shown in Figure 4.31 is constructed from queries P_1 to P_5 in Figure 4.30. This algorithm first searches for the existence of a homomorphism between branches in a new query and branches in the query index tree. For example, x is a leaf node of P_{new} and x' is a matched node of x and $x \in$ *branch* b and $x' \in$ *branch* b'. If a homomorphism exists from b to b', branch b' is a subset branch of branch b and b is super domain branch of b'. A post-processing operation for branching points is then applied to remove false positives and compute the final results.

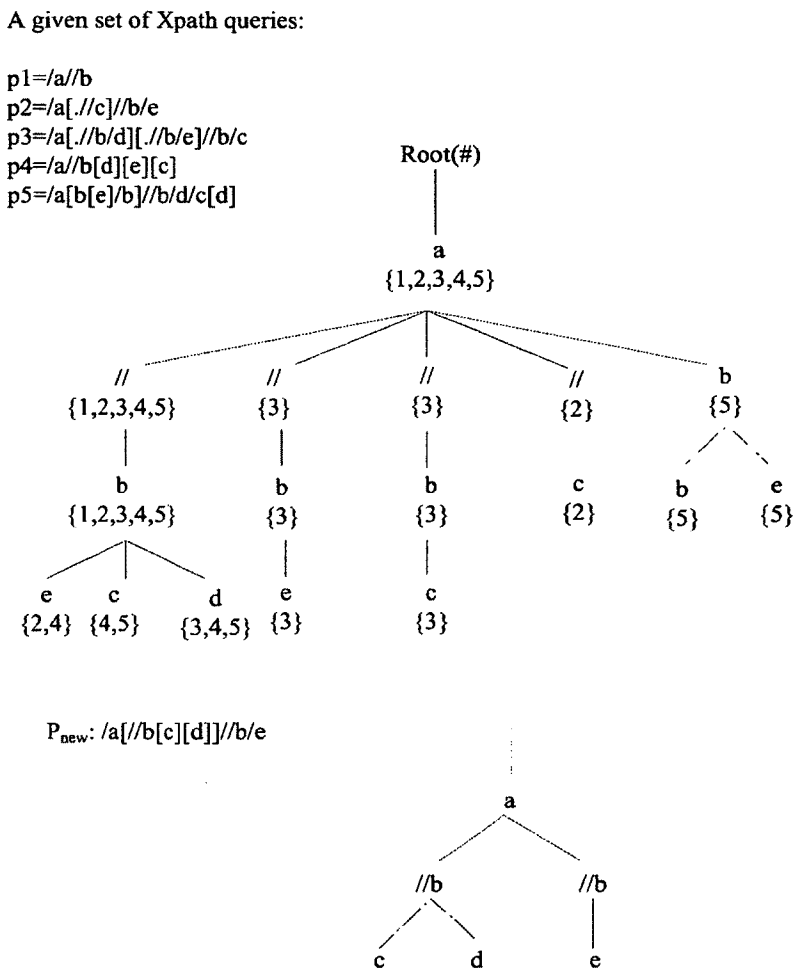


Figure 4.30: Query index tree constructed using XNET algorithm

Li et al. [71] propose an advertisement-based routing algorithm to prune the number of XML messages transmitted. Advertisements are generated from the DTDs of the XML documents to be published, because a DTD allows for deriving all possible paths from the root to the leaves appearing in a valid XML document. They also introduce a scheme to capture containment among queries and a merging scheme to further reduce the routing table size.

Fu and Zhang [59] present an automata-based algorithm to check the containee relationship between multiple queries and a single XPath query. The algorithm identifies a containment by finding a homomorphous mapping between two XPath queries.

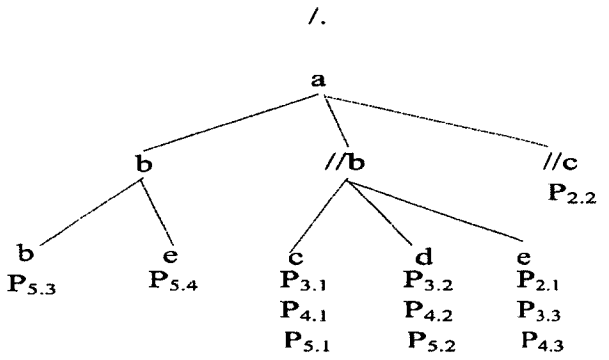


Figure 4.31: Query index tree as constructed in [107]

Figure 4.32 illustrates the automata construction process for XPath queries. Common prefixes are shared in the NFA. Figure 4.32b shows an automata constructed from the queries p_1, p_2 and p_3 shown in the left Figure 4.32a. In Figure 4.32b, $p_i.x$ represents node x in the XPath tree p_i, e.g., $p_2.1$; LP represents a set of leaf nodes in an XPath query; LB holds a set of branch nodes in an XPath query.

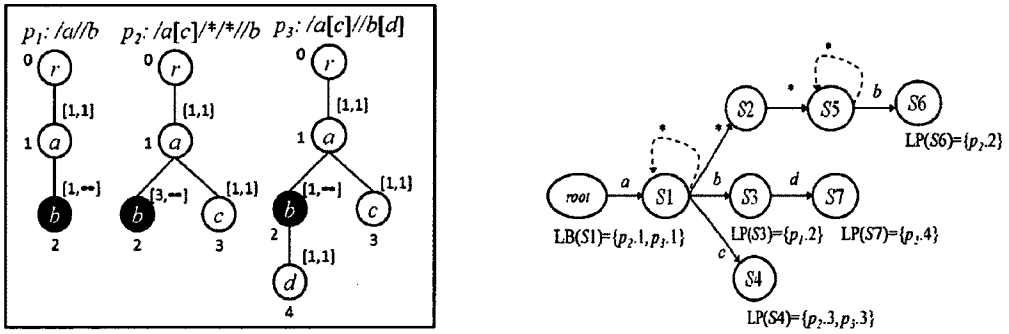


Figure 4.32: A set of XPath queries and an automata constructed using Fu's algorithm (from [59]): (a) a set of XPath queries; (b) a constructed automata.

Homomorphism can be determined by running the automata with an input new query q. A depth-first search traversal algorithm is used to read an input XPath query. A stack is used to keep track of previous visited states. If a leaf node v in the automata is reached, the corresponding state is marked. If descendant nodes of the current node are all processed, current states are popped from the stack. Meanwhile, if there is a branch state in the current state set, the algorithm resets information

related to this state and continues. In this way, the algorithm can find out which parts of existing queries are covered by a new query. But there is no solution in [59] on how to find existing queries containing the new query in the other direction.

Along this direction of detecting containment based on homomorphism, Placek et al. [86] propose a heuristic approach for checking containment of partial tree-pattern queries (PTPQ). A PTPQ allows either keyword-style queries with no structure or strictly tree-structured query specified by XPath. A PTPQ can query text data or XML data. A PTPQ p can have no precedence relationship in it. More tree-structured queries can be derived from p for containment detection. A necessary and sufficient homomorphism-based condition is provided between PTPQs and the set of derived tree-structured queries. A heuristic approach checks the containment of Q into Q_1 by checking the existence of a homomorphism from Q_1 to Q_a which is equivalent to Q. This heuristic is sound but not complete because if there is a homomorphism from Q_1 to Q_a, $Q \subseteq Q_1$, it is possible that $Q \subseteq Q_1$ and there is no homomorphism from Q_1 to Q_a. This heuristic technique equivalently adds new paths that contain precedence relationship to a PTPQ to increase the chances of having a homomorphism in this query. This increases the possibility of containment detection.

4.5 Chapter summary

This chapter presents an XPath aggregation approach which consists of containee and container algorithms. The containee algorithm is used to detect if the new query covers part of the global query tree. The container algorithm is used to detect if the new query is covered by the existing global query tree. Both of these proposed algorithms have a lower time complexity in comparison to XSearch. The experimental results show that the proposed containee algorithm can effectively reduce the processing time by 32.4%-85.2% when the number of existing queries is large. The proposed container algorithm can efficiently reduce the processing time by 18.6%-49.8% when the number of existing queries is large. Thus for the results presented in this chapter,

a significant performance improvement is achieved by the proposed algorithms over the XSearch algorithm. Except for a few cases where the number of existing queries is very small and the query paths are short, the total processing time for the new aggregation approach is lower than that of the XSearch algorithm.

Chapter 5

XML routing models

XML routing is a fundamental technology in pub/sub systems. Existing approaches for filter-based XML routing and filtering systems have to perform XML filtering and matching operations at multiple brokers in an overlay network, which limits the scalability of the system. In this chapter, different communication models for XML routing are investigated to reduce the XML filtering and matching overhead. Specifically, two new models are presented in this thesis. They are the cross-layer XML model and the peer model. This chapter investigates these two models and compares them with the traditional XML message routing model and the unicast model. In other words, four models are considered and compared:

(i). Conventional XML multicast: This is the model that has been adopted for many XML filtering and matching approaches, as explained in Section 2.5.1. Examples include Yfilter [53, 54], Afitler [28], Gfilter [47], and Bfilter [50].

(ii). Cross-layer XML message filtering and forwarding, or simply cross-layer model: The cross-layer XML model integrates the XML application layer and the IP network layer for XML message routing.

(iii). Peer model: The peer model removes the filtering operations for intermediate brokers. Instead, the filtering operation is only needed at edge brokers.

(iv). Unicast model: This is the traditional unicast approach without XML aggregation or multicast. A unicast is established for each matched subscriber.

The cross-layer XML model is discussed in Section 5.1 and the peer model is discussed in Section 5.2. Section 5.3 and Section 5.4 are devoted for performance evaluations for the different XML routing models.

5.1 Cross-layer XML message filtering and matching model

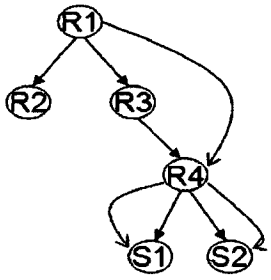
As stated, the cross-layer model makes use of both the XML application layer and the IP layer for message forwarding. Two variations are considered for the thesis. The first one is a hybrid approach of XML multicast and IP network layer unicast by considering query covering relationship. The second variation makes use of previously computed covering information from downstream brokers. Both are discussed in details in this section.

5.1.1 Cross-layer XML multicast scheme

The goal of the conventional application-layer XML multicasting scheme is to deliver a message only to those subscribers who specify their interest in it. In the worst case, all brokers in the XML pub/sub system need to receive and filter publication messages, which has a high computational and routing cost. Therefore, this thesis proposes a new design that moves subscriptions either to a provider edge (PE) broker or to brokers that have similar subscriptions along the shortest path from a PE to a customer edge (CE) broker. In this thesis, a PE is a broker that is immediately adjacent to the publisher and a CE broker is directly connected to the subscribers.

To support this idea, the IP address is appended to a subscription so that a broker knows the destination to forward a message directly. In other words, the new design integrates the application-layer multicasting and IP-layer routing. Consequently, subscriptions are stored at edge nodes and at some, but not all, intermediate brokers.

The advantage is the reduction of the number of filtering operations performed during message forwarding.



Proposed technique –

filtering operation may not be needed for intermediate brokers, e.g., R3

Application-layer routing table

Router	Subscription	Destination Node
R4 (CE)	Q1	S1
	Q2	S2
R1 (PE)	Q1	R4
	Q2	R4

Figure 5.1: Proposed cross-layer XML message filtering and forwarding model

Figure 5.1 illustrates this cross-layer XML filtering and forwarding model. Incoming queries Q_1 and Q_2 are stored either at an intermediate broker, if there is a covering relationship with Q_1 and Q_2, or at R_1, the PE node for this example. In Figure 5.1, Q_1 from S_1 and Q_2 from S_2 are stored at R_4 which is a CE. Assume that, for this simple scenario, there are no other queries. A message matching either Q_1 or Q_2 is filtered only at R_1 (PE) or R_4 (CE). There is no XML filtering and matching operation performed at intermediate routers, e.g., R_3. R_3 only conducts one trivial table lookup operation to find the IP address for R_4 at the IP layer. In other words, when broker R_3 receives the message with a destination address R_4, R_3 checks its network routing table and forwards the message to R_4. The information of neighboring nodes and reachability of other routers have already been generated and stored at the IP layer. The same IP routing table is used again in the cross-layer XML multicast scheme. Algorithm 12 and Algorithm 13 present the algorithms used in the new cross-layer model.

Algorithm 12 shows how to route a query from a CE to a PE. When an incoming query q arrives at a broker R_i (except a CE), R_i tests if there is still room available for q. The term ST_i represents subscription table. $Size(ST_i)$ is the current size of the subscription table at broker R_i and $max\_size(ST_i)$ is the maximum size of the subscription table allocated to a router. If the table is full, query q is forwarded to the next broker. Otherwise, the broker evaluates the covering relations by invoking the *queryAggregation*(q) function to detect containment relations with other existing queries. When q arrives at a CE, it is inserted into the subscription table and is forwarded to parent brokers without checking the available size of ST.

Algorithm 12: Query forwarding pseudo code for cross-layer model

```
1  When a node R_i receives a query q from its neighbor and R_i is not a CE broker:
2  if size(ST_i) ≥ max_size(ST_i) then    // ST_i is the query table at router R_i
3  |  determine a neighbor in neighbor set S and forward q to that neighbor;
   |  /* Exceed capacity, move to next broker                                  */
4  else if q comes from other brokers then
5  |  do queryAggregation(q);
   |  /* Aggregate incoming query with FSM                                     */
6  |  if ∃q' ∈ ST_i, q ⊆ q' (q' is a local query) then
7  |  |  insert q into local query table ST_i and stop;
8  |  else if ∃q' ∈ ST_i, q' ⊆ q then
9  |  |  insert q into local query table ST_i;
10 |  |  determine neighbors set S and forward query q to S;
11 |  else
12 |  |  determine neighbors set S and forward query q to S ;
   |  |  /* Forward query q to next broker                                     */
13 else
14 |  insert q into local query table ST_i and do queryAggregation(q);
   |  /* Broker is a CE and incoming query is from local subscriber           */
15 |  if ¬(∃q' ∈ ST_i, q ⊆ q') then
16 |  |  determine neighbor set S and forward query q to S;
17 |  else
18 |  |  stop;
19 return
```

If the current broker R_i finds an existing query q' which is a superset of query

q, R_i inserts q into the local subscription table and stops. In other words, q is not forwarded to its parents or upstream nodes. Next, if R_i finds the new query q covers some existing query q', the broker inserts q into the local subscription table; then forwards q to parent brokers. Otherwise, if q does not have any covering relationship with other queries at a broker, it is continually forwarded to parents brokers of R_i. In Figure 5.1, Q_1 is added to the subscription table at R_4 (CE) and then forwarded to R_3. Whether Q_1 is added to R_3 depends on if Q_1 has any covering relationship with other queries at R_3. If q is a new interest, q is simply forwarded to parents of the current broker with no addition of q to the current router. For instance, Q_1 and Q_2 are not stored in R_3.

Algorithm 13: XML messages forwarding pseudo code for cross-layer XML multicast model at broker R_i

```
1 feed x to the filter engine and generate a destination list (R_l, R_i, , R_m, S_1, S_j, , S_n);
  // R_i represents router i and S_j represents subscriber j
  /* Get a list of destination brokers or subscribers d                    */
2 for each member d in the destination list do
3 |   if d is not equal to R_i then
4 |   |   check the IP network table and get the next hop address of R_j;
5 |   |   forward message x to R_j;
6 |   else
7 |   |   send message x to local subscriber S_k;
8 return
```

Algorithm 13 describes how to forward a message x based on the aggregated queries at broker R_i. An incoming message x is filtered by the filtering engine embodying all subscriptions at broker R_i and generates a receiver list if any subscriber(s) is (are) identified. In Algorithm 13, d represents either a destination of a broker that has subscriber(s) to receive the matched message x, or a destination of a subscriber that has submitted the subscription or query.

For example, in Figure 5.1, when a message arrives at R_1, R_1 does the filtering operation and identifies matched queries, Q_1 and Q_2. Based on the R_1's application-layer routing table, the destination is identified, which is R_4. The message is forwarded

to R_4 through R_3 using the IP network layer, for this example. The intermediate router, R_3, does not perform the filtering operation.

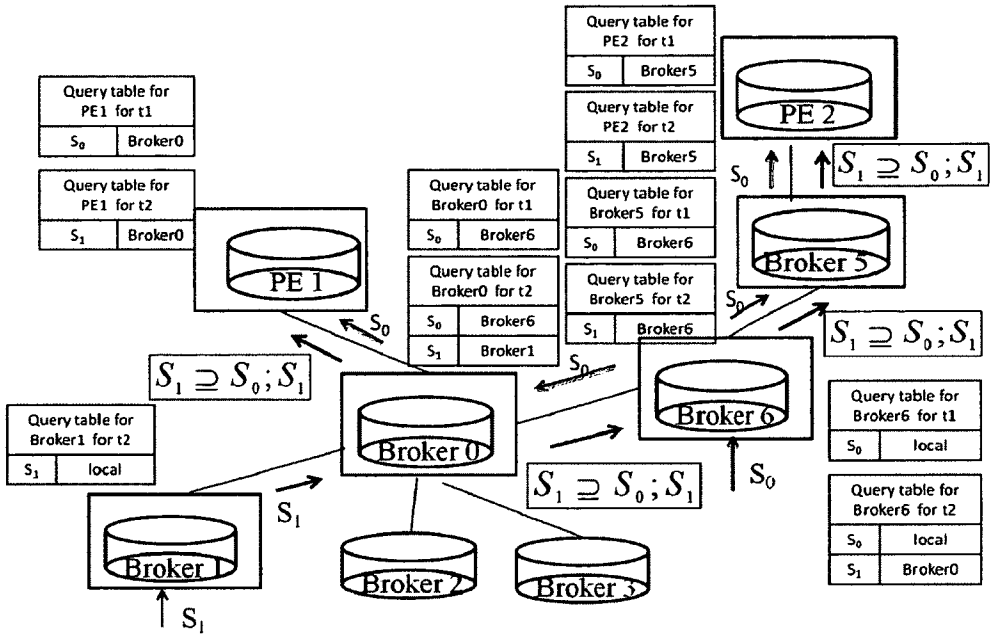
5.1.2 Query routing with covering information

The cost of aggregating XPath queries is expensive. As the complexity of supported XPath syntax increases, the cost becomes higher, as demonstrated in Chapter 4. Using aggregation, a query q sent to a parent broker may cover the queries stored in a downstream broker. In other words, q contains part of existing queries stored in a downstream broker. All messages matching q also are matched at the corresponding downstream broker. To improve the aggregation of XPath queries in a distributed environment, this thesis embeds previously computed aggregation results from a downstream broker in the message to reduce the aggregation cost in an upstream broker. To the best of our knowledge, no similar work has exploited pre-computed covering results to process subscription aggregation in such a cooperative way. The approach in [40] is for XML message filtering and matching and has some similarity with the proposed approach, but [40] concerns not query aggregation.

Table 5.1: Parent information for each broker shown in Figure 5.2

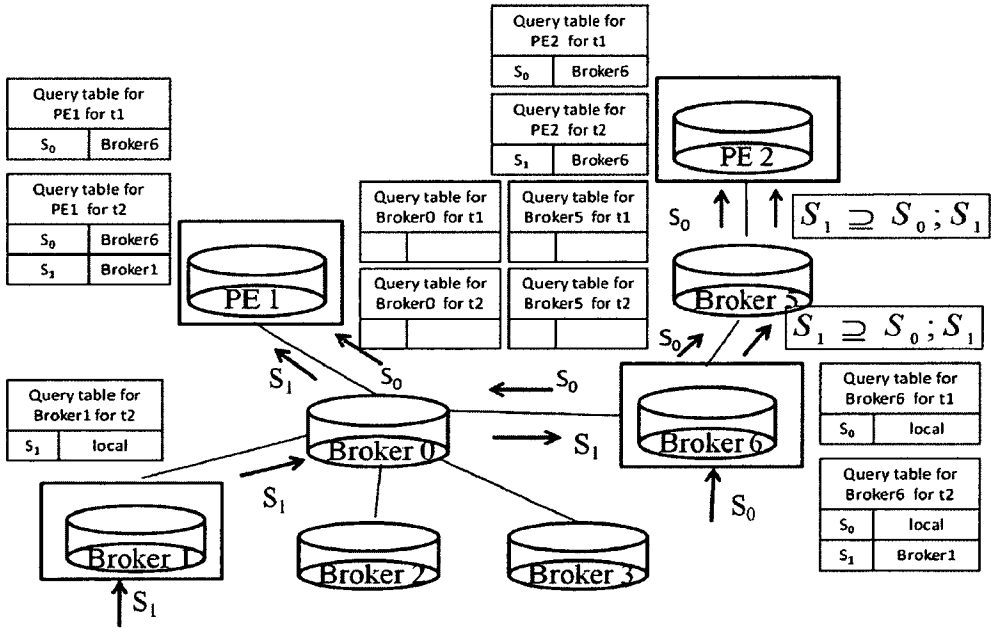
Broker	Parent Information
$Broker_0$	PE_1, $Broker_6$
$Broker_1$	$Broker_0$
$Broker_2$	$Broker_0$
$Broker_3$	$Broker_3$
$Broker_5$	PE_2
$Broker_6$	$Broker_0$, $Broker_5$
PE_1	$\emptyset$
PE_2	$\emptyset$

Figure 5.2 illustrates a query routing process with covering information using the conventional XML multicast and the cross-layer XML multicast schemes as discussed in Section 5.1.1. Assume that two queries S_1 and S_0 ($S_1 \supseteq S_0$) are injected to the network at time t_2 and t_1, respectively, and $t_1 < t_2$. A broker with a rectangle indicates



Subscriptions S_0 and S_1 arrive at t1 and t2, respectively, t1< t2 and S_1 covers S_0 $(S_1 \supseteq S_0)$

(a)



Subscriptions S_0 and S_1 arrive at t1 and t2, respectively, t1< t2 and S_1 covers S_0 $(S_1 \supseteq S_0)$

(b)

Figure 5.2: Routing with covering information: (a) conventional XML multicast model with covering results; (b) cross-layer XML message filtering and forwarding model with covering results.

that either S_1 or S_0 is stored in it. The following two subsections will explain these two schemes in more details. Parent information for each broker is listed in Table 5.1.

In the new design with query covering information, a query message contains a header and the actual query. The header contains the covering information. The concept of adding the covering information to a query message can be applied to both the conventional XML multicast and the cross-layer XML multicast schemes. Algorithm 14 and Algorithm 15 present the idea for the conventional XML multicast and the cross-layer models, respectively. The two algorithms demonstrate how a broker uses the included containment information to reduce the overhead by removing the repeated query aggregation operations, if a header contains embedded covering information.

5.1.2.1 Conventional XML multicast with covering information

First, the conventional XML multicast model with covering information included, which is defined by Algorithm 14, is considered.

Figure 5.2a demonstrates an example using the covering information scheme for the conventional XML multicast model. Each broker has two tables, a query table and a forwarding table. There are two columns for a query table. The first column represents a query; the second column shows the XML multicast destination of a query. The forwarding table is not shown in Figure 5.2a for brevity. The content of the forwarding tables for the intermediate brokers in Figure 5.2a is shown in Table 5.2. The process is described as follows.

Table 5.2: Forwarding tables for brokers at time t_1 and t_2 ($t_1 < t_2$) in Figure 5.2a and $S_1 \supseteq S_0$

Broker	t_1	t_2
$Broker_0$	S_0	S_1
$Broker_1$	$\emptyset$	S_1
$Broker_5$	S_0	S_1
$Broker_6$	S_0	S_1

1. At time t_1:

   a. $Broker_6$ receives a query S_0 from a local subscriber. S_0 is added to the query table and the forwarding table at $Broker_6$.

   b. $Broker_6$ forwards S_0 to $Broker_0$ and $Broker_5$.

   c. $Broker_0$ and $Broker_5$ add S_0 to their local query tables and forwarding tables. The XML multicast destination of S_0 is $Broker_6$. S_0 is continually forwarded to upstream brokers until S_0 reaches the PEs, e.g., PE_1 and PE_2.

2. When S_0 is received by PE_1 and PE_2:

   a. S_0 is added to the local query tables for PE_1 and PE_2. The XML multicast destination of S_0 at PE_1 is $Broker_0$; the XML multicast destination of S_0 at PE_2 is $Broker_5$.

3. At time t_2 ($t_1 < t_2$):

   a. $Broker_1$ receives a query S_1. Based on line 4 in Algorithm 14, S_1 is added to the local query table and the forwarding table at $Broker_1$. The forwarding table for $Broker_1$ is empty ($\emptyset$) at time t_1.

   b. S_1 is forwarded to $Broker_0$ which is the parent node of $Broker_1$.

4. When S_1 is received by $Broker_0$:

   a. $Broker_0$ calls the covering detection engine and identifies that $S_1 \supseteq S_0$ (line 3 in Algorithm 14).

   b. S_1 is added to the local query table of $Broker_0$ according to the steps from line 4 to line 7 in Algorithm 14. The XML multicast destination of S_1 is $Broker_1$.

   c. S_0 is replaced by S_1 in the forwarding table of $Broker_0$.

   d. The covering information $S_1 \supseteq S_0$ is constructed at $Broker_0$ and sent to its upstream nodes, $Broker_6$ and PE_1.

5. When S_1 is received by PE_1:

   a. PE_1 replaces S_0 with S_1 in its local query table. The XML multicast destination of S_1 is set to be $Broker_0$.

6. When S_1 is received by $Broker_6$:

   a. $Broker_6$ checks if there is a header containing the covering information in the message and finds out the information, $S_1 \supseteq S_0$.

   b. $Broker_6$ adds S_1 to its local query table and the XML multicast destination of S_1 is $Broker_0$.

   c. $Broker_6$ replaces the existing S_0 with the new S_1 in the forwarding table of $Broker_6$, instead of performing an aggregation operation, as the new message already conveys the containment relationship.

   d. $Broker_6$ forwards the information $S_1 \supseteq S_0$ with S_1 together to $Broker_5$.

7. When S_1 is received by $Broker_5$:

   a. $Broker_5$ checks if there is covering information in the message and finds out $S_1 \supseteq S_0$.

   b. $Broker_5$ adds S_1 to its local query table and the XML multicast destination is $Broker_6$.

   c. $Broker_5$ replaces the existing S_0 with the new S_1 in the forwarding table of $Broker_5$, without performing an aggregation operation.

   d. $Broker_5$ forwards the information $S_1 \supseteq S_0$ with S_1 together to PE_2.

8. When S_1 is received by PE_2:

   a. PE_2 replaces S_0 with S_1 in its local query table at the end. The XML mutlicast destination is $Broker_5$.

Including computed containment results in a query message can reduce the overhead incurred by repeated query aggregation operations. The message forwarding process is the same as in the conventional XML multicast model.

Continuing with Figure 5.2a, at time t_3, query S_1 is canceled. $Broker_1$ sends a message to tell $Broker_0$ that S_1 is canceled. Upon receiving the cancelation message, $Broker_0$ checks the covering information to see if S_1 has been forwarded to parents of $Broker_0$. In this example, as S_1 is a superset query which covers some existing queries at $Broker_0$. $Broker_0$ then sends two messages to $Broker_6$ and PE_1 as they are $Broker_0$'s parents. One message tells $Broker_6$ and PE_1 that S_1 is canceled; the other one indicates to register S_0 as a new representation for $Broker_0$.

5.1.2.2 Cross-layer routing with covering information

The process of cross-layer routing with covering information is described as follows. As illustrated in Figure 5.2b, queries S_1 and S_0 arrive at time t_2 and t_1, respectively, and $t_1 < t_2$. The notations for Figure 5.2b are identical to those of Figure 5.2a. The content of the forwarding tables for the intermediate brokers in Figure 5.2b is shown in Table 5.3.

1. At time t_1:

   a. $Broker_6$ receives a query S_0 from a local subscriber. S_0 is added to the query table and the forwarding table at $Broker_6$.

   b. $Broker_6$ forwards S_0 to $Broker_0$ and $Broker_5$.

   c. $Broker_0$ and $Broker_5$ call the covering detection engine (line 3 in Algorithm 15) and find out that S_0 is a new query. $Broker_0$ and $Broker_5$ forward S_0 to the parent brokers, PE_1 and PE_2 (lines 10-11 in Algorithm 15) without storing S_0 in their query tables. The forwarding tables at $Broker_0$ and $Broker_5$ are empty ($\emptyset$).

2. When S_0 is received by PE_1 and PE_2:

Algorithm 14: Conventional XML multicast model with the covering information stored in the header

```
1  first check the header field in the query message;
2  if there is no header available then
3      call covering detection engine at broker R_i;
4      if s is a new representation then
5          insert s into local query tables;
6          if R_i is not a root then
7              forward s to parent(R_i) with header;
8      if s is contained by part of existing queries in the query table then
9          insert s into local query tables;
10 if there is header indicating s ⊇ s_y in the message then
11     if s_y is a representation then
12         insert s into local query tables;
13         if R_i is not a root then
14             forward s to parent(R_i) with header;
15     if s_y is represented (or covered) by s_z in the covering table then
16         compare s_z with s for covering detection;
17         if s is represented or covered then
18             insert s into local query tables;
19         if s is a new representation then
20             insert s into the local query tables;
21             if R_i is not a root then
22                 forward s to parent(R_i) with header;
23 return
```

Table 5.3: Forwarding tables for brokers at time t_1 and t_2 ($t_1 < t_2$) in Figure 5.2b and $S_1 \supseteq S_0$

Broker	t_1	t_2
$Broker_0$	$\emptyset$	$\emptyset$
$Broker_1$	$\emptyset$	S_1
$Broker_5$	$\emptyset$	$\emptyset$
$Broker_6$	S_0	S_1

a. S_0 is added to the local query tables for PE_1 and PE_2. The destination of S_0 is $Broker_6$.

3. At time t_2 ($t_1 < t_2$):

a. $Broker_1$ receives a query S_1 from a local subscriber. Based on line 4 in Algorithm 15, S_1 is added to the local query table and the forwarding table at $Broker_1$. The forwarding table for $Broker_1$ is empty ($\emptyset$) at time t_1.

b. S_1 is forwarded to $Broker_0$ which is the parent node of $Broker_1$.

4. When S_1 is received by $Broker_0$:

a. $Broker_0$ calls the covering detection engine for S_1 (line 3 in Algorithm 15). As there is no inclusion relationship for S_1, $Broker_0$ forwards S_1 to PE_1 and $Broker_6$ without storing S_1 in its query table. The query table and the forwarding table for $Broker_0$ are empty ($\emptyset$).

5. When S_1 is received by PE_1:

a. S_1 is added to the local query table for PE_1. The destination of S_1 is $Broker_1$.

6. When S_1 is received by $Broker_6$:

a. $Broker_6$ calls the covering detection engine (line 3 in Algorithm 15) and finds $S_1 \supseteq S_0$. S_1 is added to the query table for $Broker_6$. $Broker_6$ replaces S_0 with S_1 in the forwarding table.

b. A query routing message, including S_1 and the header $S_1 \supseteq S_0$, is constructed and forwarded to the parent, $Broker_5$. (lines 12-15 in Algorithm 15).

7. When S_1 is received by $Broker_5$:

a. $Broker_5$ simply forwards the query message to PE_2 because S_0 does not exist at $Broker_5$ (lines 29-30 in Algorithm 15) without storing S_1. The query table and the forwarding table for $Broker_5$ are empty ($\emptyset$).

8. When S_1 is received by PE_2:

   a. PE_2 updates its query table by replacing S_0 with S_1. The destination of S_1 is $Broker_6$ (lines 16-18 in Algorithm 15).

Consider the next scenario where a query S_1 as shown in Figure 5.2b is canceled. $Broker_1$ sends a cancelation message to $Broker_0$, indicating that S_1 is removed. Then $Broker_0$ finds that S_1 is not stored in it. $Broker_0$ simply forwards the Cancel(S_1) message to PE_1 and $Broker_6$. When the Cancel(S_1) message is received by $Broker_6$, S_1 is removed and the covering table is checked to see if S_1 contains part of existing queries. In this example, S_1 represents S_0 at $Broker_6$. At $Broker_6$, S_1 is removed and S_0 is re-registered, and two messages are sent to $Broker_5$. One message tells that S_1 is canceled; the other one registers S_0 as a new representation for $Broker_6$. $Broker_5$ finds S_1 is not present. After the search is complete, $Broker_5$ simply forwards these two messages to its parent PE_2. At PE_2, S_1 is removed and updated with S_0 and the query forwarding process stops. The XML message processing and forwarding is the same as the cross-layer model as the cross-layer model as demonstrated in Figure 5.1.

In summary, using the covering information stored in the header in a query message can reduce the cost of query aggregation, as the covering information is already embedded in the query message. The brokers receiving the covering information do not have to repeat the query aggregation operation. The time for aggregation could vary based on the query structure and the number of existing queries. Take Q4 in Section 4.3.1 as an example. Based on results of Q4, when the number of queries is varied from 500, 5000, to 60000, the time for the aggregation operation is observed to lie between 1.38 ms and 157.6 ms. By carrying previously identified covering information, a broker can significantly reduce the processing cost incurred by repeated query aggregation operations.

Algorithm 15: Cross-layer model with the covering information stored in the header

```
1  first check the header field in the query message;
2  if there is no header available then
3      call covering detection engine at broker R_i;
4      if s is a new query that is submitted by local subscribers and s is a new
       representation then
5          insert s into local query tables;
6          if R_i is not a root then
7              forward s to parent(R_i) with headers;
8      if s is represented or covered by some existing queries in the query table and s is
       submitted by local subscribers then
9          insert s into local query tables;
10     if s is from other brokers and s is not stored in the query table then
11         forward s to parent(R_i);
12     if s covers some existing queries in the query table then
13         add s to the local query table;
14         construct a query routing message M including the covering information and
           s;
15         forward M to parent(R_i);
16 if there is header indicating s ⊇ s_y in the message then
17     if s_y is a query stored in the query table then
18         update query local tables with s;
19         if R_i is not a root then
20             forward s to parent(R_i) with header;
21     if s_y is covered by s_z in the covering table then
22         compare s_z with s for covering detection;
23         if s is represented or covered then
24             insert s into local tables;
25         if s is a new representation then
26             insert s into the local query tables;
27             if R_i is not a root then
28                 forward s to parent(R_i) with header;
29     if s_y is not present in the query table and R_i is not a root then
30         forward s to parent(R_i) with header;
31 return
```

5.2 Peer model for XML routing

The existing application-layer XML multicast model shown in Figure 5.3a has been used by Siena [35], Yfilter [53, 54], Afilter [28], Gfilter [47], and Bfilter [50], in which each broker conducts query aggregation and message filtering operations. Figure 5.3a shows that the XML routing information is stored at each broker: PE, Broker1, Broker2, CE1 and CE2.

The main problem with the conventional XML multicast model is that there is a lot of overhead for filtering and aggregation, because query aggregation and message filtering have to be conducted at each node in order for an XML message to be forwarded to the next broker.

In Figure 5.3a, for instance, queries Q2, Q3, and Q4 are aggregated at CE1; Q0 and Q1 are aggregated at CE2. CE1 and CE2 forward the aggregated queries to their upstream broker, Broker2. Broker2 aggregates the queries and builds the application-layer routing table, and it forwards the aggregated query tree to Broker1. Broker1 does the same operations. Finally, PE receives queries from Broker1 and builds its XML routing table. For this simple example, the PEs, Broker1 and Broker2 have an identical routing table. PE could be connected to multiple brokers in other scenarios. For more complicated scenarios, a PE's routing table will be different from that of the intermediate broker, e.g., the IP router, and further, the routing table at each broker may also be different.

When a message is published and arrives at a PE, the PE performs the filtering operation, e.g., using Yfilter, to identify the matched subscribers and their corresponding brokers. In this example, the published message is forwarded to Broker1 which repeats the same operation and forwards the message to Broker2. Broker2 does the same filtering task and may send the message to CE1 and/or CE2, depending on the subscriptions.

As illustrated in Figure 5.3a, query aggregation and message filtering operations

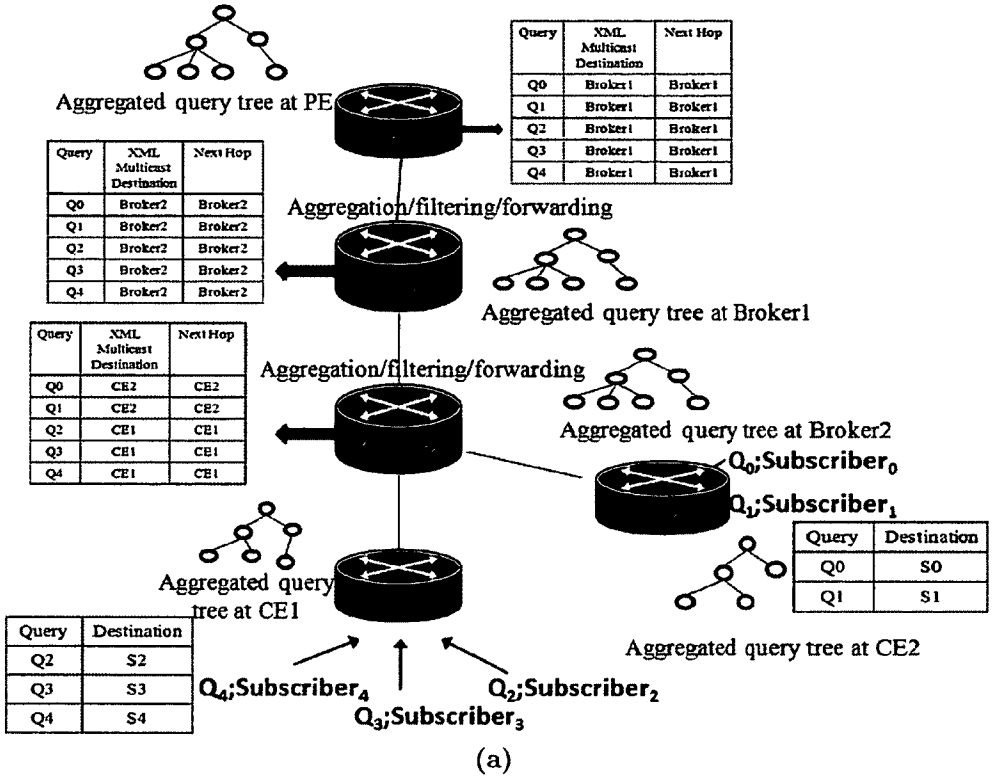


Figure 5.3: Two application-layer models: (a) conventional XML multicast model; (b) a new peer model.

are performed at each node using the conventional application-layer multicast model. The overhead of the conventional XML multicast model is high, especially, when the aggregated tree is large or the message is also large or complex.

The new peer model, as shown in Figure 5.3b, can mitigate the problem. In the peer model, intermediate brokers are simply traditional IP routers that only perform the forwarding functionality and no XML message filtering and query aggregation tasks are performed. Further, no queries are stored in IP routers that are the intermediate brokers. Subscriptions are only stored at PEs and CEs, e.g., PE, CE1, and CE2, as depicted in Figure 5.3b. The peer model relieves the intermediate brokers from performing aggregation and filtering operations repeatedly. The existing filtering engines, such as Yfilter, Afilter, Gfilter and Bfilter, can still be used together with the peer model at PEs and CEs.

However, compared to the conventional XML multicast model, the peer model may forward a message multiple times, since the peer model makes use of the unicast scheme between PE and CEs. However, the number of unicast messages is small, as the unicast messages are only sent to CEs. Compared to the peer model, the cross-layer model presented in Section 5.1.1 stores queries at selected brokers along the path with the aid of IP-layer network routing technologies. The cross-layer model still needs to aggregate queries along the path at some nodes from CE to PE. Not only aggregation is a time-consuming operation, but also complicated query table updates are needed when a query is inserted or removed.

On the contrary, the peer model presented in this section only stores queries at PE and CE nodes. The peer model design also facilities the query table update in case of an addition or a removal of a query, because only the query tables at CEs and PEs need to be updated.

The work in [26] shares some similarities with this research. The difference is that [26] uses XML pub/sub systems to deliver multicast services only within a virtual private network (VPN). The newly proposed peer model can be applied to the public

Internet without VPN services. With the new peer model, the service provider can install the XML aggregation and filtering functionalities at PEs and CEs. PEs and CEs in the new peer model perform the XML filtering and matching, and XPath query aggregation functionalities as shown in Figure 5.4. The intermediate brokers are simply the traditional IP routers performing message forwarding only.

There are two possible scenarios to consider for the peer model for message forwarding. These two scenarios are related to unicast and multicast. The following subsection describes both in more detail.

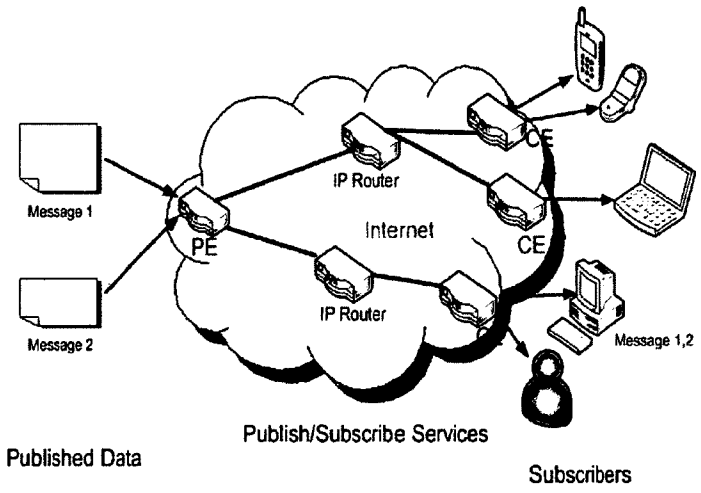


Figure 5.4: XML pub/sub with the peer model

5.2.1 Unicast versus multicast using the peer model

In this section, two scenarios of XML message delivery schemes are described using the peer model. Figure 5.5a depicts a scenario where all queries from Subscriber1 and Subscriber2 are sent to the PE node. When there is an XML document that matches both queries of Subscriber1 and Subscriber2, PE duplicates the XML document and sends it to CE1 and CE2 using two separate unicast messages. For this scenario, there are two diverse paths for these two subscribers.

Figure 5.5b depicts another scenario where in the query routing phase, when queries from subscribers are received by the CE, the CE forwards query messages

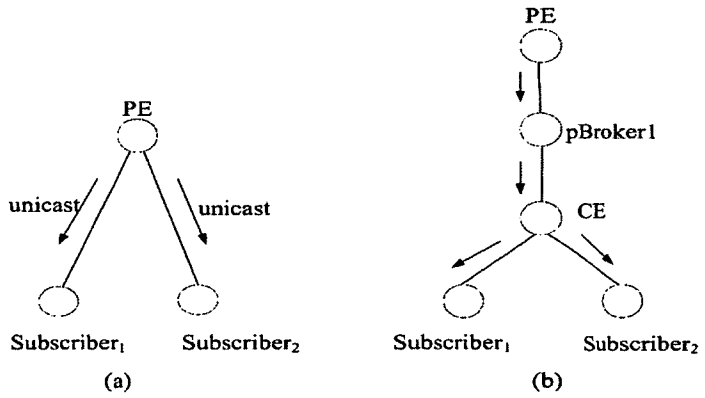


Figure 5.5: XML pub/sub deliver scenarios: (a) unicast; (b) multicast.

to PE through IP routers or IP network layer technology. In order to decrease the number of messages transmitted over the network, CEs aggregate user queries. For example, given two XPath queries Q_1: /a//b and Q_2: /a//b[d][e][c] from $Subscriber_1$ and $Subscriber_2$, respectively, they can be aggregated into one query Q':/a//b after query aggregation is applied at node CE. Subsequently, only query (Q') is transmitted to PE, instead of two separate queries (Q_1 and Q_2).

In the XML document filtering and forwarding phase, a matched XML document is delivered from the PE to CE or identified CEs in general, via IP routers. In Figure 5.5b, only one message is forwarded from PE to CE using the multicasting scheme, as aggregation was already performed for these two queries. Two unicast messages are sent from the CE to $Subscriber_1$ and $Subscriber_2$. So the peer model can guarantee system correctness, while reducing the filtering operation at each intermediate node. Compared with the unicast method, as illustrated in Figure 5.5a, the bandwidth usage for the multicast scheme is lower.

5.2.2 Discussion

Compared with the conventional XML multicast model, the peer model can effectively reduce the number of filtering/aggregation operations and subscription storage for the intermediate brokers. The peer model is more flexible and can efficiently

handle network topology changes. The limitation of the peer model is that the aggregated query tree may not be minimal. In the conventional XML multicast model, a subscription has to be aggregated at each broker so that a minimal aggregated query tree can be obtained. Compared to the conventional multicast model, the peer model may forward XML messages multiple times to different CEs. However, the number of unicast messages is expected to be small, as the number of CEs is typically small.

Furthermore, the end-to-end delay is lower for the peer model, where filtering operations are only performed at PE and some CEs. The filtering time for complex messages and a large query tree could be higher than the XML message forwarding time. Section 5.4.1 presents the experimental results in details.

5.3 Experimental setup and performance metrics

This section describes how experiments are set up and the various parameters that are used. The metrics and topologies adopted for performance evaluation are also described. All experiments are conducted on three machines which form a local network and the network speed is 100 Mbit/sec. Each machine is a system consisting of two 3.0 GHz Intel Pentium cores with 4.0 GB of RAM running under Windows XP.

5.3.1 Experimental network topologies

Two network topologies are used for the experiments:

- Topology #1 is shown in Figure 5.6a.
- Topology #2 is depicted in Figure 5.6b.

We choose the tree as demonstrated in Figure 5.6a as network topology #1 for the following reasons: (i) many subtrees that are similar to topology #1 exist in practical multicast trees; (ii) topology #1 is the left branch of the network topology tree used in [42] (see Figure 6). The tree length for message routing using topology #1 is 3; the tree length for message routing using topology #2 is 11.

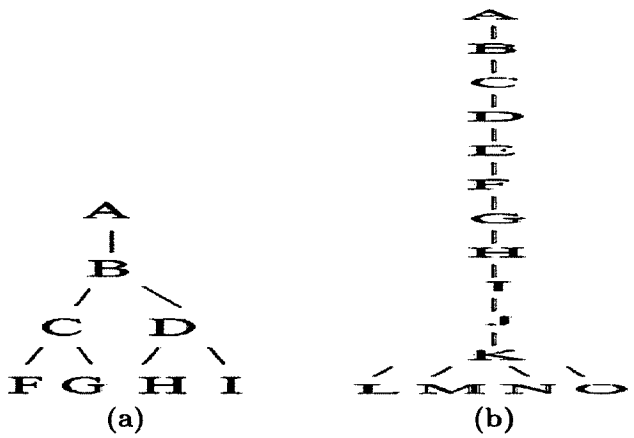


Figure 5.6: Experimental network topologies: (a) topology #1; (b) topology #2.

Next, topology #2, a longer delivery path, is considered (see Figure 5.6b). The purpose of topology #2 is to examine the impact on the performance of different models for XML routing when the length of an end-to-end delivery path is increased.

5.3.2 Experimental parameters

Table 5.4 lists the parameters and their values used in experiments for different XML routing models. A single parameter is varied in each experiment while the other parameters are held at their default values. Table 5.5 shows the default parameter values for XML messages and XPath queries. The exponential random numbers for message inter-arrival time are generated from SimJava [1].

Determination of default values for some experimental parameters. To choose the default parameter values used in our experiments, some existing works in the literature are referred. For example, in [47], the default values for the experimental parameters are: query depth=6, prob(//)=0.2, prob(*)=0.1, number of predicates=0, number of branches=0 and query population=20,000; in [42], the parameter values are listed as follows: query depth is between 3 and 10, prob(//)=0.1, prob(*)=0.1, message recursion depth=3, message arrival is characterized by a Poisson distribution with the arrival rate of 1 msg/sec, query population is between 1000

and 50,000, and message size is 22 tag pairs; in [53], the parameter values used are described as follows: query population is between 1 and 50,000, query depth is between 6 and 10, prob(//)=0.2, and prob(//)=0.2. The default values for some of the system parameters are based on values used in the research described in the referred articles [47, 42, 53]. We use the method of long run to capture the steady-state of the system. As time becomes longer, the effect of the initial conditions on later observations lessens and the observations appear to vary around a common mean. When this point has been reached, the data-collection phase can begin(See page 423 in [24]). We analyze the length of experiments and measure the average end-to-end delay as a function of the experimental run length that is given by the number of messages considered in the experiment. The experimental run lengths used include 100, 250, 500, 1000, 2000, 3000, 5000, 6000, 7000, 10000, and 12000. From the absolute time difference between two consecutive run lengths in the list, we find that after a length of 5,000, the absolute difference is smaller than 1.5% and the system reaches a steady-state. Hence, 6,000 messages are selected as the run length to be used in the experiments. This was found to be adequate for the investigation of the relative performance of the algorithms that this thesis focuses on. All reported data include outliers in the following experiments.

5.3.3 Performance metrics

The performance metrics used in the experiments are explained below.

(i). **Average E2E delay for one message:** The description of this metric is based on Definition 3 and Definition 4, depending on the model used.

Definition 3. When the conventional multicast approach or the peer model is used, let t_{is} represent the timestamp for sending the i^{th} publication message from the PE. Let t_{ij} represent the timestamp of the j^{th} ACK message of the i^{th} publication message received by the PE. Let d_{ij} be the difference in time between t_{ij} and t_{is}, $d_{ij} = t_{ij} - t_{is}$. Let k represent the total number of ACK

Table 5.4: Parameter values used in the peer model experiments

Parameters	Descriptions	Values
Message type	Test messages type	recursive messages with book.dtd
Message size	Maximum test message size	{1 KB, 5 KB, 10 KB, 15 KB, 20 KB}
λ	Message arrival rate	{0.5 msg/sec, 0.85 msg/sec, 2 msgs/sec, 4 msgs/sec, 8 msgs/sec}
Number of XPath queries	Test queries population	{10,000, 20,000, 50,000}
Prob(//)	Probability that an element has //	{0.1, 0.2, 0.3}
Prob(*)	Probability that an element has *	{0.1, 0.2, 0.3}
Prob(branching)	Probability that a query has branches	{0, 0.1, 0.2 }
Prob([ ])	Probability that a query has predicates	{0.1, 0.2, 0.3}
Query depth	The longest path length of a test query tree	{3, 6, 10}
Match ratio	The ratio of matched queries over test query population	around {0.05}
Number of CEs	The number of CEs of the test network topology	{2, 4, 6}

Table 5.5: Default parameter values of XML message and XPath query

Parameter	Value
Message size	10 KB
λ	0.85 msg/sec
XPath query population	20,000
Query depth	6
Prob(//)	0.2
Prob(*)	0.2
Prob(branching)	0.1
Prob([ ])	0.2
Query duplication	allowed
Match ratio	0.05
Number of CEs	4

Table 5.6: Closed system; 20 KB test message; 50,000 queries; and experiments comprise 1000 message arrivals:(a): based on topology #1 and (b): based on topology #2

(a)

	Multicast	Peer model
Average E2E delay for one message including outliers (ms)	918.3	179.7
Average E2E delay for one message without outliers (ms)	903.5	164.1
Maximum processing ability λ^* (message/sec)	1.1	6.1

(b)

	Multicast	Peer model
Average E2E delay for one message including outliers (ms)	3828.8	191.0
Average E2E delay for one message without outliers (ms)	3817.2	175.5
Maximum processing ability λ^* (message/sec)	0.26	5.70

messages corresponding to the same i^{th} publication message received by the PE. Then, the average E2E delay for publication message $i = \frac{1}{k}\sum_{j=1}^{k} d_{ij}$. Let R be the number of messages. The average E2E delay for one message $= \frac{1}{R}\sum_{i=1}^{R}(\frac{1}{k}\sum_{j=1}^{k} d_{ij})$.

Definition 4. When a unicast approach is used, let ts_i represent the timestamp for sending the i^{th} publication message, and t_i represent the timestamp of receiving the i^{th} ACK by the PE. Let d_i be the difference in time between t_i and ts_i, $d_i = t_i - ts_i$. Let N represent the total number of messages transmitted. Then, the average E2E delay for one message $= \frac{1}{N}\sum_{i=1}^{N} d_i$.

(ii). **Average filtering time at one node:** The average filtering time is the difference between the time a publication message leaves the filtering engine and the

time of arrival of the message at the filtering engine. The average is computed over 100 runs.

(iii). **Average round-trip time between two nodes:** The average round-trip time between two nodes is the difference between the time an ACK message is received by a sender and the time of sending the publication message to the receiver by the sender. The average is computed over 200 runs. The communication protocol between a sender and a receiver is Java non-blocking I/O (NIO).

(iv). **Average transmission time at one node:** The average transmission time is the difference between the time a publication message starting to be sent and the time when sending is complete. The average is computed over 200 runs.

(v). **Link stress on the network from PE:** This is the number of bytes transmitted over the network when one message is filtered.

(vi). **Average processing time at a broker:** The average processing time is the difference between the time at which the message is sent and the time at which the message is received by the broker. The average is computed over the total number of messages received by the broker.

5.3.4 E2E delay for different communication models

The total elapsed time for the unicast approach is significantly higher (several folds higher) than the other two approaches as shown in Table 5.7. Using unicast, the same message may be forwarded multiple times, which has a higher delay if the message is large and/or the number of unicast sessions is high. For example, in the unicast experiments the results of which are presented in Table 5.7, each message is replicated 12 times. The number in the bracket following unicast represents the number of messages published and sent by the PE. For example, when the number of messages published is 500, there are 6000 messages transmitted in total. However,

there are only 500 message transmitted using the conventional XML multicast and the peer model approaches. So in the remaining experiments, the performance of unicast is not included, as the delay is much higher than that of other approaches.

Table 5.7: Unicast, conventional XML multicast, and peer model comparision for the topology #2

	multicast	peer model	unicast (100)	unicast (500)	unicast (1000)
Average E2E delay for one message (ms)	1270.26	118.63	3222.97	86190.89	692752.08

5.4 Performance results and analysis

In this section, we present a set of experiments in order to quantify the performance of the various approaches (multicast and peer model). Some preliminary results for a small network are reported in [31, 33]. This thesis describes the performance analysis for larger networks and considers more parameters and various additional values for these parameters.

5.4.1 Experimental results for primary operations of XML filtering and matching conducted at one broker

The performance of primary XML filtering and matching operations conducted at one broker is examined. The purpose of the results presented in this section is to provide a reference to the results in other sections.

Table 5.8a records the average transmission time for sending publication messages of different sizes. Table 5.8b records the average round trip time between two nodes for simply sending a message to a client. We can observe that the network cost for our prototype is small. The average message transmission time for real core routers is even smaller. Compared to the filtering cost, the message transmission time is negligible.

Table 5.9a shows the performance of Yfilter when message size is varied and other

Table 5.8: Network measurements:(a) average transmission time;(b) average round-trip time between two nodes for different messages

(a)

Message size	Average transmission time (ms)
1 KB	0.13
5 KB	0.2
10 KB	0.26
15 KB	0.36
20 KB	0.44

(b)

Message size	Average round-trip time (ms)
1 KB	0.65
5 KB	1.07
10 KB	1.12
15 KB	1.29
20 KB	1.46

parameters are at default values as shown in Table 5.5. Table 5.9b lists the filtering performance when only one parameter of an XPath query is changed. Table 5.9c shows the filtering performance when query depth is varied. The purpose is to study the impact that some parameters have on the performance.

The *book.dtd* is used for the experiments and the DTD has one long branch and some short branches. In order to generate a query with long depth, we modify the code of *PathGenerator* class in Yfilter by allowing the long branch, containing a recursive element *section*, to be the main path. But for queries in previous experiments, every branch has the same probability of being the main path. The test XML message *book.xml* is a recursive message. The theoretical analysis in [28] indicates that the number of matched transitions from a node is an exponential function of the recursive depth of a message and queries. The results of Table 5.9c demonstrate that the filtering cost increases significantly for a recursive message as the query recursion depth increases.

5.4.2 Experimental results for topology #1

This section presents the experimental results for topology #1. The machine allocation is depicted in Figure 5.7: the PE broker A and one intermediate broker B run on machine 1, three brokers (C, F, and G) run on machine 2, and three brokers (D, H, and I) run on machine 3. These three machines form a local area network.

Table 5.9: Average filtering time:(a) average filtering time for one message when the message size is varied;(b) average filtering time for one message when only one query parameter is varied;(c) average filtering time for one message when the query depth is varied.

(a)

Message size	1 KB	5 KB	10 KB	15 KB	20 KB
Average filtering time for one message (ms)	11.66	64.69	116.33	160.83	245.18

(b)

Varying query parameter	Average filtering time for one message (ms)
default	116.33
prob(//)=0.1	77.13
prob(//)=0.3	141.55
prob(*)=0.1	99.85
prob(*)=0.3	115.14
prob(branching)=0	79.11
prob(branching)=0.2	128.06
prob([ ])=0.1	110.98
prob([ ])=0.3	116.34
query population=10,000	75.57
query population=50,000	170.39

(c)

Varying longest query depth	Average filtering time for one message (ms)
query depth=3	60.59
query depth=6	375.53
query depth=10	1351.84

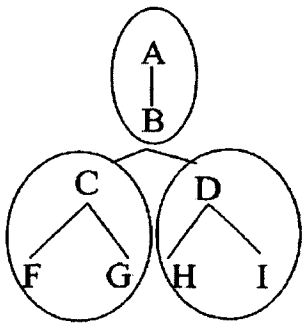


Figure 5.7: The machine allocation for topology #1 on three machines

Table 5.10 shows the performance results using the default parameter values with the two approaches (conventional XML multicast and peer model): the average E2E delay for one message using multicast is 4.98 times that of the peer model; the link stress for both approaches is 9.96 KBytes. As it can be seen, the peer model based approach significantly outperforms the multicast approach. Table 5.11 lists the average processing time at each broker of topology #1 from which we can find a big difference using different approaches for intermediate brokers. For example, the average processing times for brokers B, C, and D when the multicast approach is used are 108.4 ms, 120.2 ms, and 109.2 ms, respectively, while the average processing time when the peer model is applied are only 2.4 ms, 5.2 ms, and 5.3 ms, respectively.

Table 5.10: Performance evaluation for default parameter values with topology #1

Performance metrics	multicast	peer model
Average E2E delay for one message (ms)	316.35	63.47
Link stress (KBytes)	9.96	9.96

Table 5.11: Average processing time at each broker for default parameter values using different models for topology #1

Average processing time for one message (ms)	B	C	D	F	G	H	I
Multicast	108.44	120.20	109.18	53.06	55.52	43.43	50.53
Peer model	2.42	5.19	5.33	55.05	52.25	43.64	49.41

The purpose of the analysis presented in Figure 5.8 is to understand the effect of message parameters on the multicast and peer model approaches.

Effect of message size. Figure 5.8a shows the performance of E2E delay as a function of the message size. The peer model outperforms the multicast approach. The average E2E delays are 50.95 ms and 1050.77 ms using conventional XML multicast for a 1 KB and a 20 KB message, respectively. This shows the limitations of the conventional XML multicast: the expensive filtering operations have to be repeated

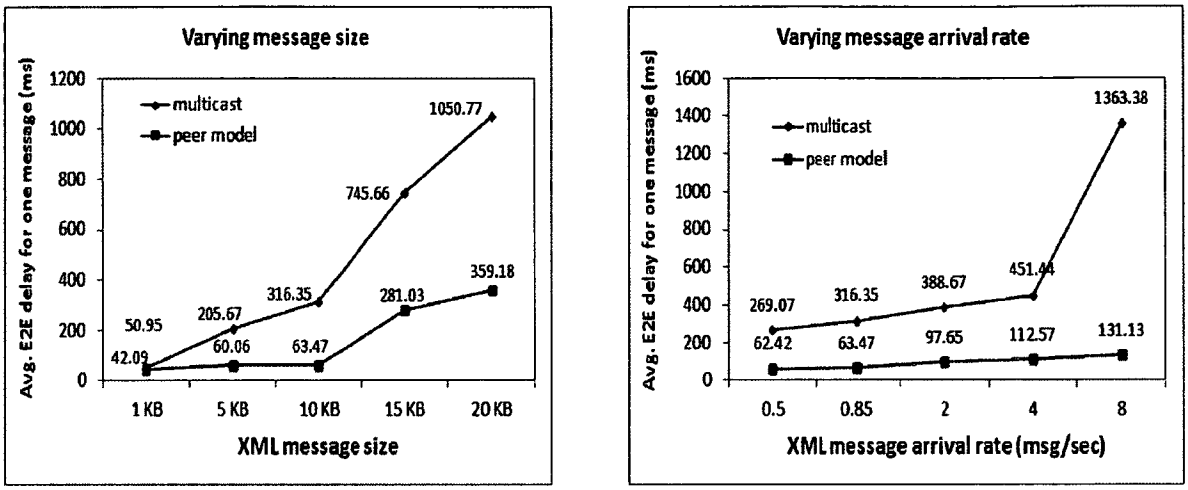


Figure 5.8: Performance evaluation for topology #1:(a) varying XML message size;(b) varying default message arrival rate

many times. For the peer model, when the message size is 5 KB and 10 KB, the average E2E delay is around 60 ms. For a 5 KB message, the average processing times for *F*, *G*, *H*, *I* are 31.04 ms, 31.23 ms, 24.78 ms, and 31.80 ms, respectively. The average processing times for *B*, *C*, and *D* are 1.57 ms, 4.47 ms, and 4.33 ms, respectively. For the 15 KB and 20 KB message, the average E2E delay is 281.03 ms and 359.18 ms, respectively. For example, for a 15 KB message, the average processing times at *F*, *G*, *H*, *I* brokers are 78.08 ms, 79.06 ms, 62.57 ms, and 73.66 ms, respectively, while the average E2E delay is 281.03 ms. The difference is caused by the waiting time when a write buffer is filled up. For a 1 KB message, the average processing times at *F*, *G*, *H*, and *I* are 5.88 ms, 5.89 ms, 5.46 ms, and 5.85 ms, respectively. But the average E2E delay for one message is 42.04 ms. The difference between the average E2E delay and the sum of average processing times along a path from a PE to a CE is because of the communication cost.

Effect of message arrival rate. Figure 5.8b shows the performance of the average E2E delay as a function of the average message arrival rate. The message with default message size and the queries generated from the default query parameters

in Table 5.5. are used in the experiments; but the message arrival rate is varied. For the multicast approach, when λ is increased from 4 msg/sec to 8 msg/sec, the average E2E delay increases from 451.44 ms to 1363.38 ms (see Figure 5.8b). However, for the peer model, the average E2E delay increases from 112.57 ms to 131.13 ms only. The socket processing time for the multicast approach is much higher than that of the peer model. Hence, when there is a large increase in the number of messages published per second, more messages are queued for the multicast approach compared with those for the peer model.

Effect of query parameters. The purpose of the analysis shown in Figure 5.9 is to understand the effect of query parameters on the performance of multicast and peer model approaches. In Figure 5.9, we vary the parameters of an XPath query, including the probability of //-operator, wildcard-operator, predicates, branching, query population, and query path. The results from Figure 5.9a to Figure 5.9d demonstrate that the peer model always outperforms the conventional XML multicast. In the experiments of varying query depths as shown in Figure 5.9d, the average message inter-arrival time is 5120 ms. The end of the next subsection explains how the average message arrival rate is determined for this experiment. Figure 5.9d depicts the effect of the depth of recursive queries on the filtering performance. For example, the end-to-end delay for one message using the conventional XML multicast approach when the query depth is 10 is 19.5 times that of the query depth at 3; the end-to-end delay for one message using the peer model when the query depth is 10 is 13.2 times that of the query depth at 3. In comparison to the peer model, the average E2E delay for the conventional XML multicast model increases more sharply with an increase in the XPath query depth.

5.4.3 Experimental results for topology #2

This section presents the experimental results for topology#2. The purpose of studying topology #2 is to understand the effect of long delivery paths on the performance

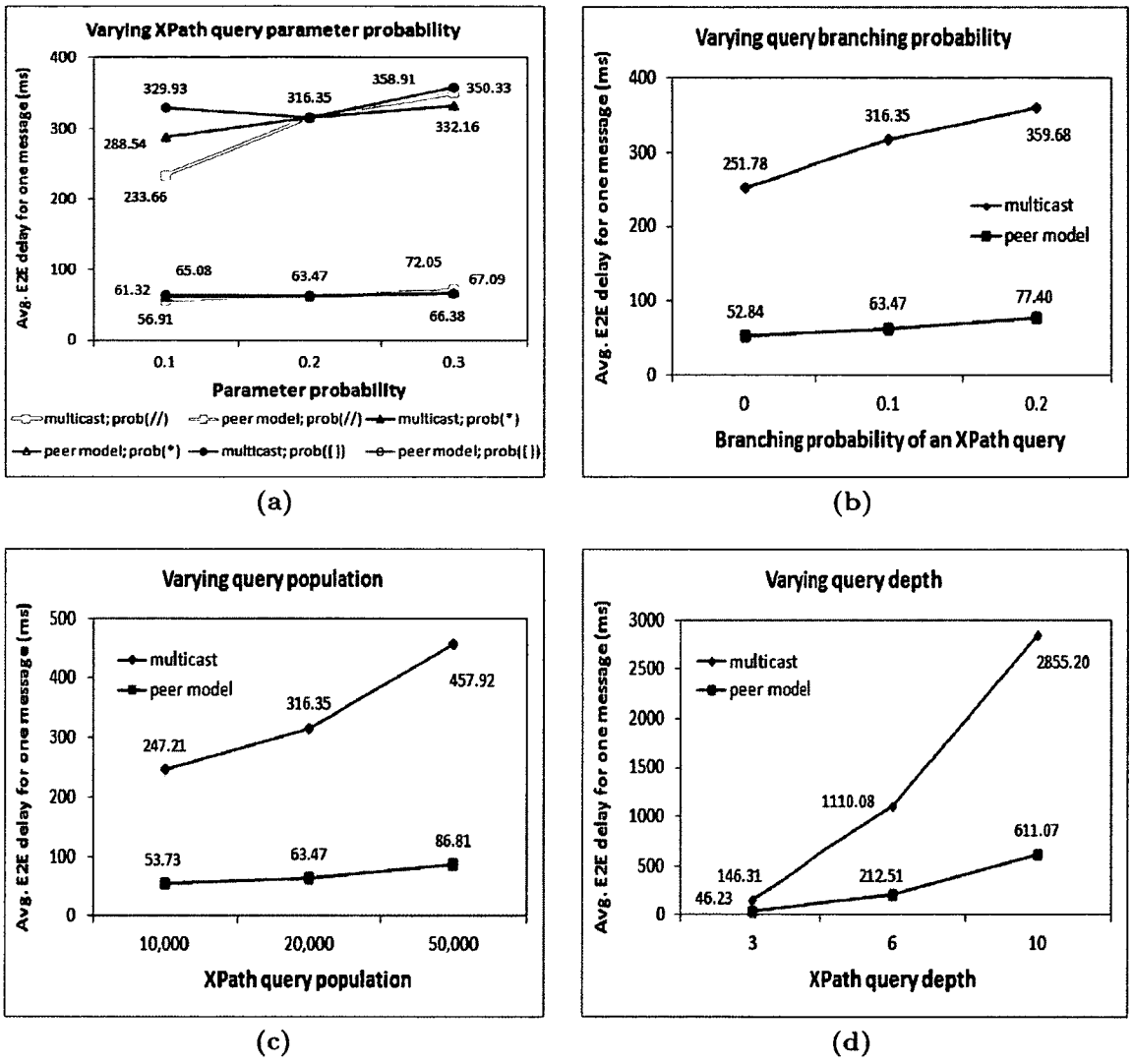


Figure 5.9: Performance evaluation for topology #1:(a) varying the probability of prob(//), prob(*), and prob([ ]), respectively;(b)varying XPath query branching probability;(c)varying XPath query population; and (d)varying XPath query depth

of the different approaches. The machine allocation for the topology #2 is shown in Figure 5.10: five brokers (A, B, C, D, and E) run on one machine, five brokers (F, G, H, I, and J) run on another machine, a broker (K) and four CE brokers (L, M, N and O) run on a third machine. These three machines form a local area network.

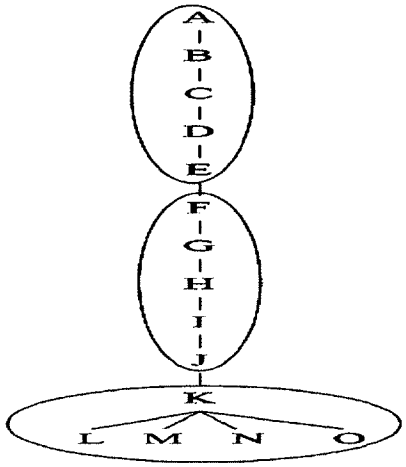


Figure 5.10: The machine allocation for topology #2 on three machines

Table 5.12 shows the performance results for default parameter values using the two approaches (conventional XML multicast and peer model). The average E2E delay for the conventional XML multicast is 10.7 times that of the peer model in terms of the average E2E delay. Table 5.13 lists the average processing time at each broker for topology #2. For intermediate brokers from B to J, the average processing times when the multicast approach is used are 63.33-80.73 times those taken by the peer model. For broker K, the average processing time using the multicast approach is 5.12 times that obtained with the peer model. The experimental results shown in both tables demonstrate that the peer model approach outperforms the conventional XML multicast in terms of efficiency and has the same link stress.

The purpose of the analysis presented in Figure 5.11 is to understand the effect of message parameters on the performance of the conventional XML multicast and the peer model approaches.

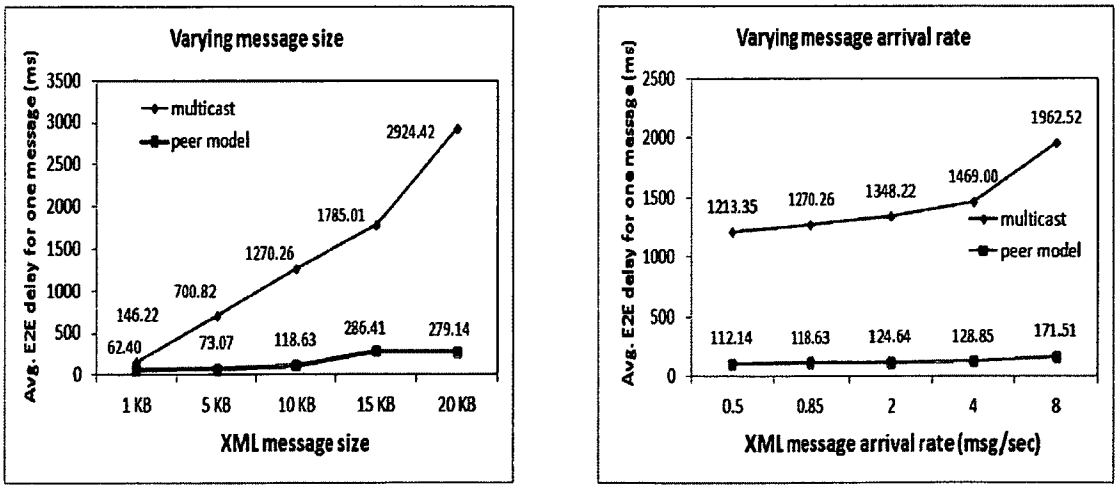
Effect of message size. Figure 5.11a shows the results when the message size is

Table 5.12: Performance evaluation for default parameter values with topology #2

Performance metrics	multicast	peer model
Average E2E delay for one message (ms)	1270.26	118.63
Link stress (KBytes)	9.96	9.96

Table 5.13: Average processing time at each broker for default parameter values using different models for topology #2

Brokers	Average processing time (ms)	
	multicast	peer model
B	122.54	1.67
C	119.06	1.88
D	120.54	1.83
E	123.41	1.63
F	115.69	1.43
G	117.08	1.74
H	117.97	1.75
I	117.44	1.83
J	117.90	1.70
K	116.53	22.77
L	92.58	95.75
M	94.35	75.31
N	64.92	66.17
O	73.77	86.19



(a) (b)

Figure 5.11: Performance evaluation for topology #2:(a) varying XML message size;(b) varying default message arrival rate

varied. We can observe that the message size can significantly affect the performance of multicast because the filtering cost increases as the message size increases. The average E2E delay for the conventional XML multicast includes the filtering time and transmission time at all brokers along the path from PE to CEs. The average E2E delay for the peer model includes the times for computing the next hop and the new destination list at pBrokers as well as the times for filtering at CEs. The number of subscriptions stored at a CE is smaller than the number of subscriptions stored at intermediate brokers using the multicast approach, because in the conventional XML multicast model a broker has to subsume subscriptions of subtree nodes rooted at it. Consider an example of 20 KB messages in the peer model: the average filtering costs at brokers L, M, N, and O are 217.49 ms, 199.47 ms, 167.98 ms, and 211.79 ms, respectively; the average processing costs at brokers from B to J are in the range of 2.72 ms to 3.12 ms; the average processing cost at broker K is 30.63 ms. Thus, the average processing times for the intermediate brokers in the peer model are smaller.

Effect of message arrival rate. Figure 5.11b shows the trend when the message arrival rate increases. If λ is smaller than 90% usage of the maximum processing ability, e.g., 0.5 msg/sec, 0.85 msg/sec, and 2 msg/sec, the change on the average E2E delay is small. However, for the multicast approach, when λ reaches a certain number, e.g., 4 msg/sec, the average E2E delay increases quickly because of the queue waiting time. The performance improvement produced by the peer model increases with an increase in λ.

Effect of query parameters. The purpose of the analysis shown in Figure 5.12 is to understand the effect of query parameters on the performance of multicast and peer model approaches. We experiment with different values for query parameters, including prob(//), prob(*), prob([ ]), prob(branching), and query population. The message and the λ are based on default values.

First of all, we can observe that the peer model outperforms the conventional XML multicast approach in different query parameter combinations. Secondly, the

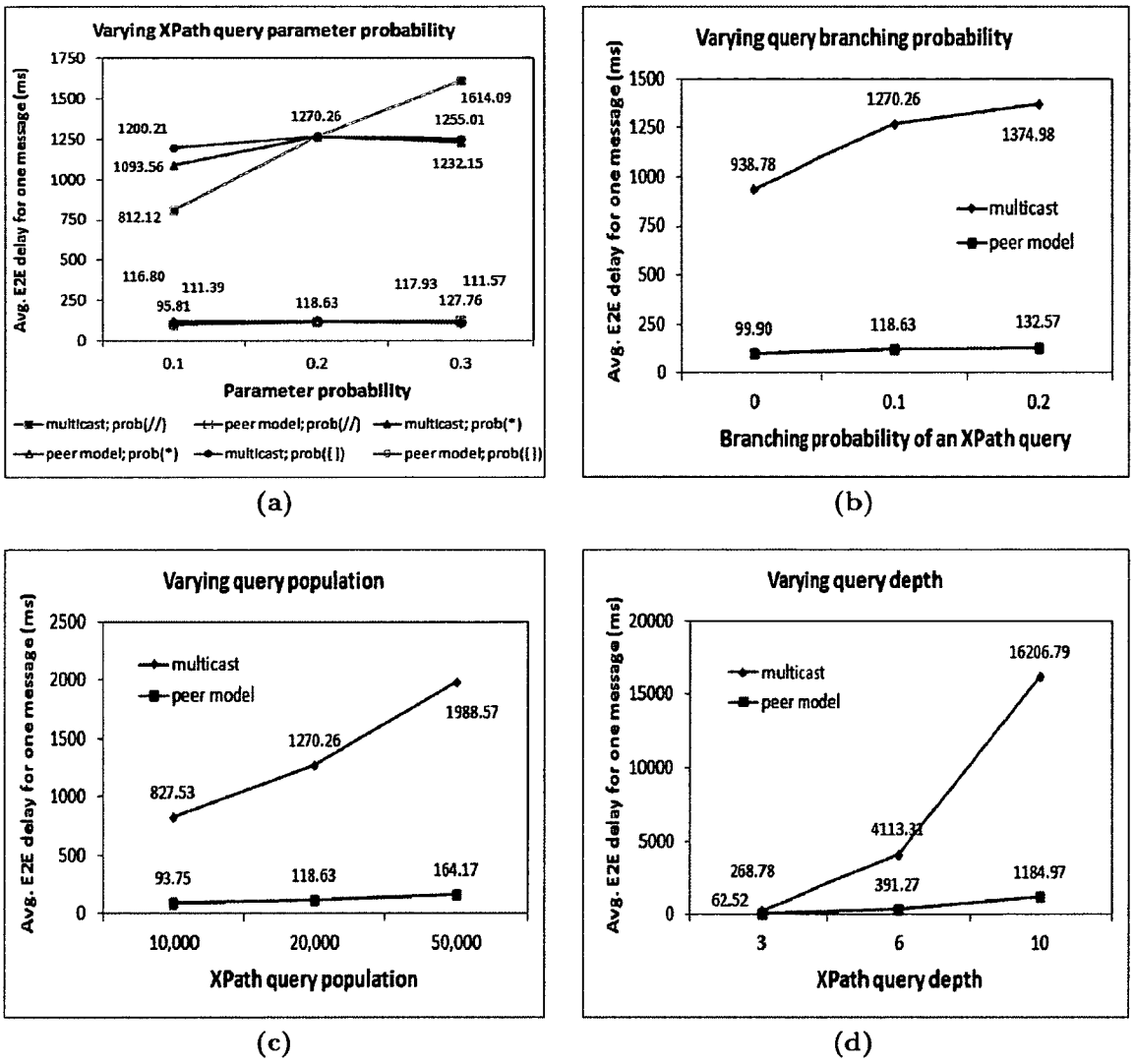


Figure 5.12: Performance evaluation for topology #2:(a)varying the probability of prob(//), prob(*), and prob([ ]), respectively;(b)varying XPath query branching probability;(c)varying XPath query population; and (d)varying XPath query depth

filtering cost increases as the complexity of XPath queries (captured in prob(//) in Figure 5.12a, prob(branching) in Figure 5.12b, and query population in Figure 5.12c) increases. But the performance for various probability values for predicates, prob([ ]), as presented in Figure 5.12a, shows minor changes due to the fact that more query predicates do not match the message.

In the results of the experiment of varying query depth as shown in Figure 5.12d, the average message arrival rate is different from the other experiments, because the query has a recursive structure. However, λ still needs to be less than 90% of the maximum processing ability of a broker. The determination of the arrival rate is described as follows. We base ourselves on the worst-case scenario when deciding the value of λ. In the worst-case analysis, a 20 KB message and 20,000 queries that have a recursive query depth of 10 are used. The average filtering time of the worst-case scenario is 1843.20 ms when repeating the experiments 1000 times. Since there are five brokers running on one machine, the sum of the processing time is 9215.98 ms. The processor of a testing machine has two cores. So the required processing time is 4607.99 ms. The maximum message arrival rate λ^* is 0.217 msg/sec. The average message arrival rate λ is 0.195 msg/sec. Thus, we can deduce that the average message inter-arrival time is 5120 ms which is used for the experiments of varying query depth.

5.5 Chapter summary

This chapter presents two new models for XML message routing. They are cross-layer XML filtering and forwarding model, and the peer model. This chapter investigates these two models and compares them with the conventional XML message multicast model and the unicast model through extensive experiments. The results demonstrate that the peer model has much lower end-to-end delay than other models. The comparison between conventional XML multicast model and the peer model in summarized in Table 5.14.

Table 5.14: Comparison between the conventional XML message multicast model and the peer model

Conventional XML Multicast		Peer Model	
☹	Long end-to-end delay	☺	Short end-to-end delay
☹	Expensive and complex for handling network topology change	☺	Efficient and flexible for handling network topology change
☺	Can produce a minimal aggregated query tree	😐	Can not guarantee a minimal aggregated query tree
☺	Forwards messages once	😐	May forward messages multiple times to different CEs Small number of unicast messages

Chapter 6

Conclusions and future work

Pub/sub systems have received a lot of attention from both academia and industry. XML continues to be a major web data source. This dissertation explores the important aspects of XML-based pub/sub systems.

6.1 Summary and conclusions

This dissertation presents three key pieces of work presented in Chapter 3, Chapter 4 and Chapter 5. The contents of these chapters are herein summarized.

Central to Chapter 3 is adding caching functionality to an XML router in an XML pub/sub system. Two caching schemes were proposed and devised for XML documents for performance improvement. The caching modules are expected to be installed on brokers. The complete message caching (YF-C-Cache) and structure-based caching (YF-S-Cache) schemes avoid repetitive filtering operations and to improve system performance. Both caching schemes can significantly shorten the processing time for XML messages. The YF-C-Cache scheme, for example, can reduce the processing time to almost 67.9% for weather data on August-5-2011 when the hit ratio was 84.74% and is better suited for systems that do not have a high frequency of message updates. The YF-S-Cache scheme takes advantage of the common structure of XML messages and lower processing time for filtering messages against queries

without predicate at internal brokers; its performance advantage is observed to increase with an increase in the number of intermediate nodes in the system.

Chapter 4 targets a new XPath query aggregation approach. XPath query aggregation is an important problem related to XML routing and XPath query caching. The proposed new algorithm takes advantage of the properties of region encoding scheme, from which the parent/child and ancestor/descendant relationships can be effectively derived. When using the region encoding scheme, the proposed algorithm does not need to traverse a tree recursively. At this stage, a question may be asked: how does one choose an XPath query aggregation approach between XSearch available in the literature and the approach proposed in this thesis? The features influencing the choice between the aggregation nalgorithm proposed in this thesis and XSearch are briefly discussed. (i) Space: the approach presented in this thesis requires extra space to build the global query tree, but the difference between each other is small as explained in Section 4.3.5; (ii) Probability of //-operator: if the probability of //-operator is high, the proposed approach is a good choice because it can efficiently compute the //-operators; (iii) Modifications of queries: if the query addition or removal is not at a high frequency, our proposed approach is recommended.

The presented approach cannot currently support the *-operator. Extending the approach presented in the thesis to handle the *-operator forms an interesting direction for future research.

Chapter 5 investigates efficient XML routing algorithms. The traditional application-level multicast approach has to filter XML messages repeatedly at each hop from a PE node to a CE node. The proposed peer model limits the XML message broker functions to be performed only at PE and CE nodes. This architecture offers several advantages: low computation cost, easy deployment and management, high network efficiency, and support for frequent user subscription changes. An XML pub/sub prototype is built and different routing approaches are evaluated on the basis of two different network topologies.

6.2 Directions for future research

A number of different directions that can be explored in the future is presented.

Future work on XPath query aggregation may include identifying the conditions under which the container should be computed before the containee and vice versa. In a pub/sub system, a new query is forwarded to upstream brokers only when it cannot be covered by the existing queries at the brokers. In the case where a new query is covered by existing queries, computing the container function first is preferred. One direction for future work is to study and analyze which function, container or containee, is good to be computed first. Future directions for XPath query aggregation may also include studying the approach for query updates and investigating their performance.

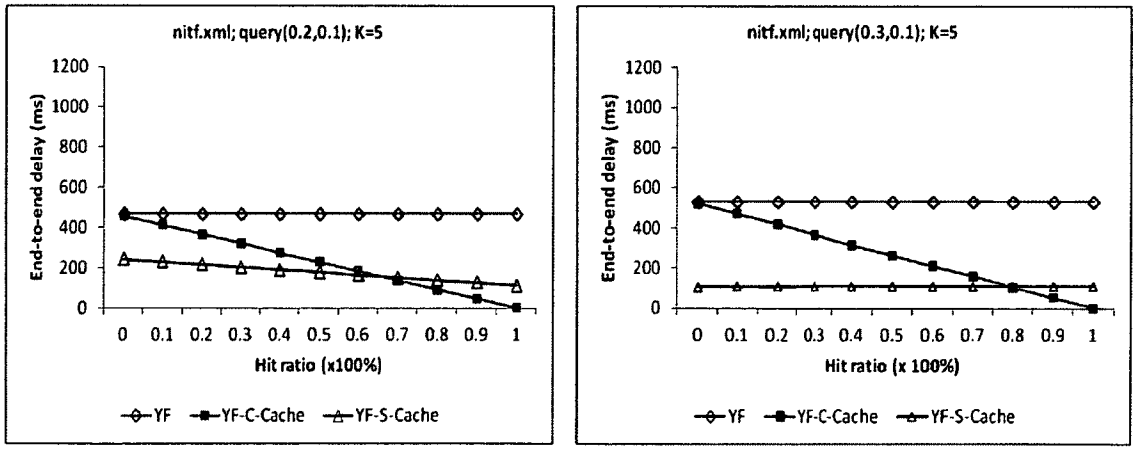
Conducting experiments using the query covering information can be another direction. Performing XML routing experiments with various other network topologies forms an important direction for future research. Integrating the three techniques - caching, query aggregation, and routing - into a prototype and evaluating its performance with real workload warrants investigation.

Appendices

Appendix A

Supplementary caching synthetic message results for K=5

A.1 Average E2E delay for nitf messages when K=5 part 1



(a) (b)

Figure A.1: E2E delay for nitf messages for K=5 part 1: (a)Query(0.2,0.1); (b)Query(0.3,0.1);

A.2 Average E2E delay for nitf messages when K=5 part 2

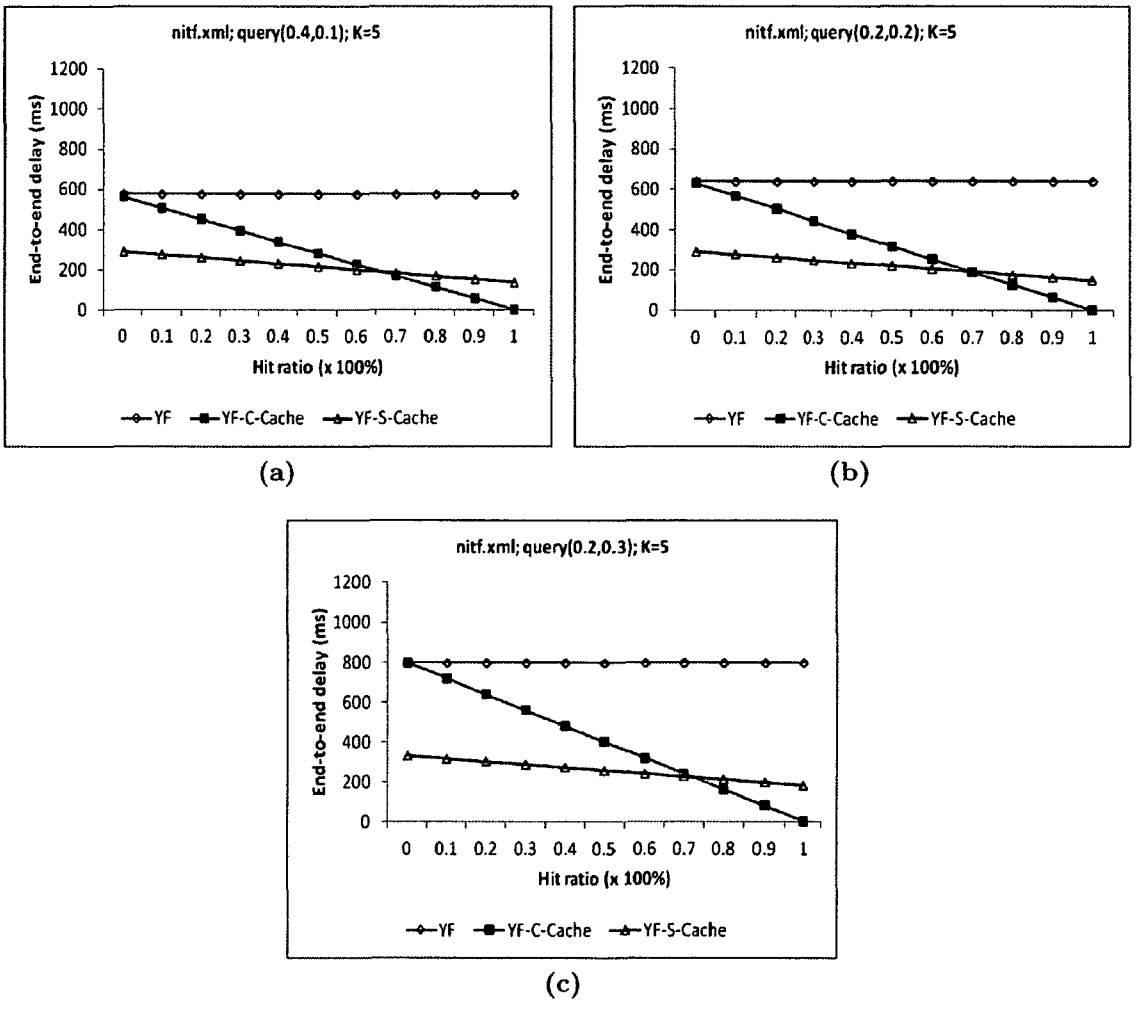


Figure A.2: E2E delay for nitf messages for K=5 part 2: (a)Query(0.4,0.1); (b)Query(0.2,0.2); (c)Query(0.2,0.3).

A.3 Average E2E delay for book messages when K=5

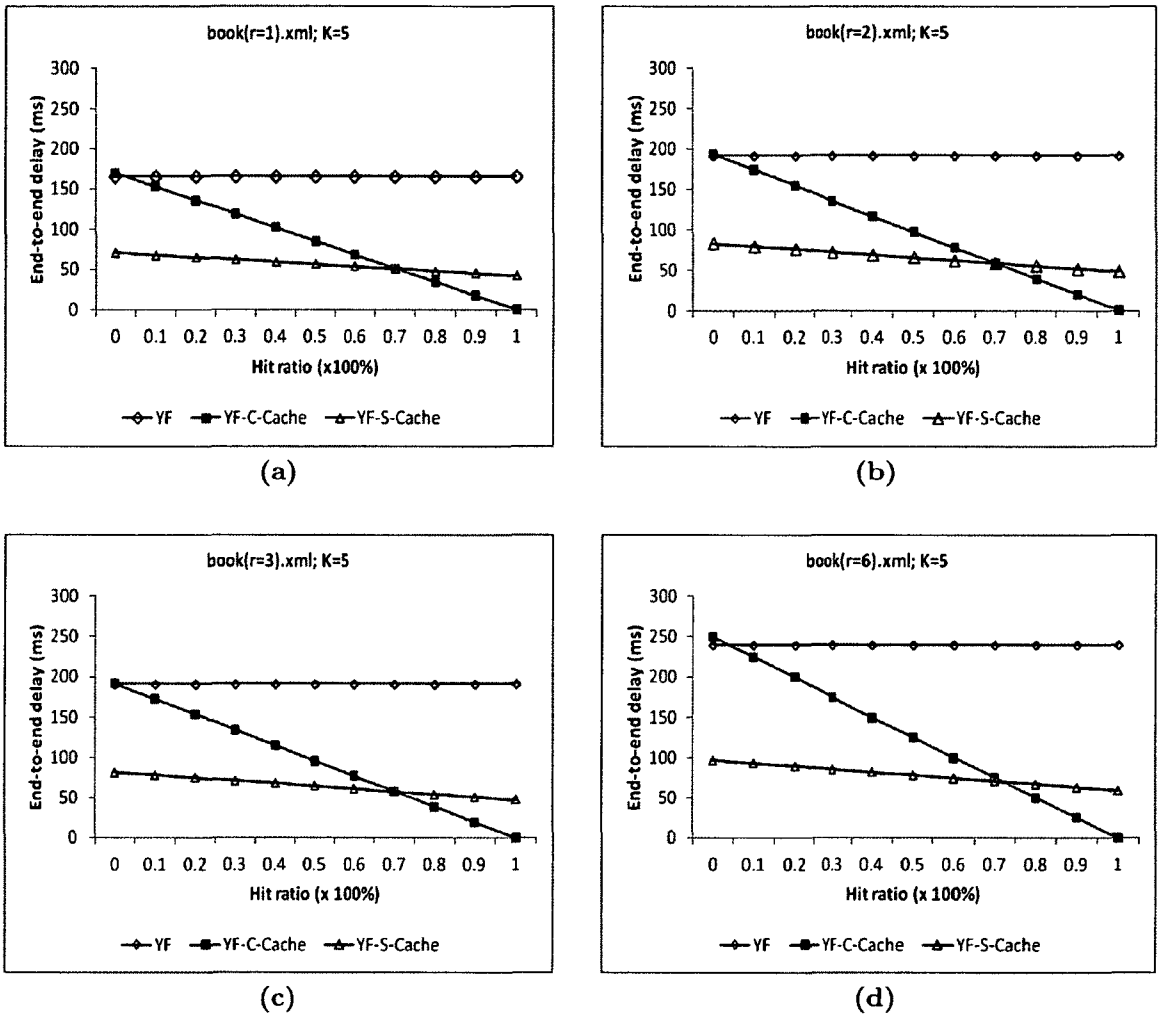


Figure A.3: E2E delay for book messages for K=5: (a)book (r=1).xml; (b)book (r=2).xml; (c)book (r=3).xml; (d)book (r=6).xml.

A.4 Average E2E delay for bib messages when K=5 part 1

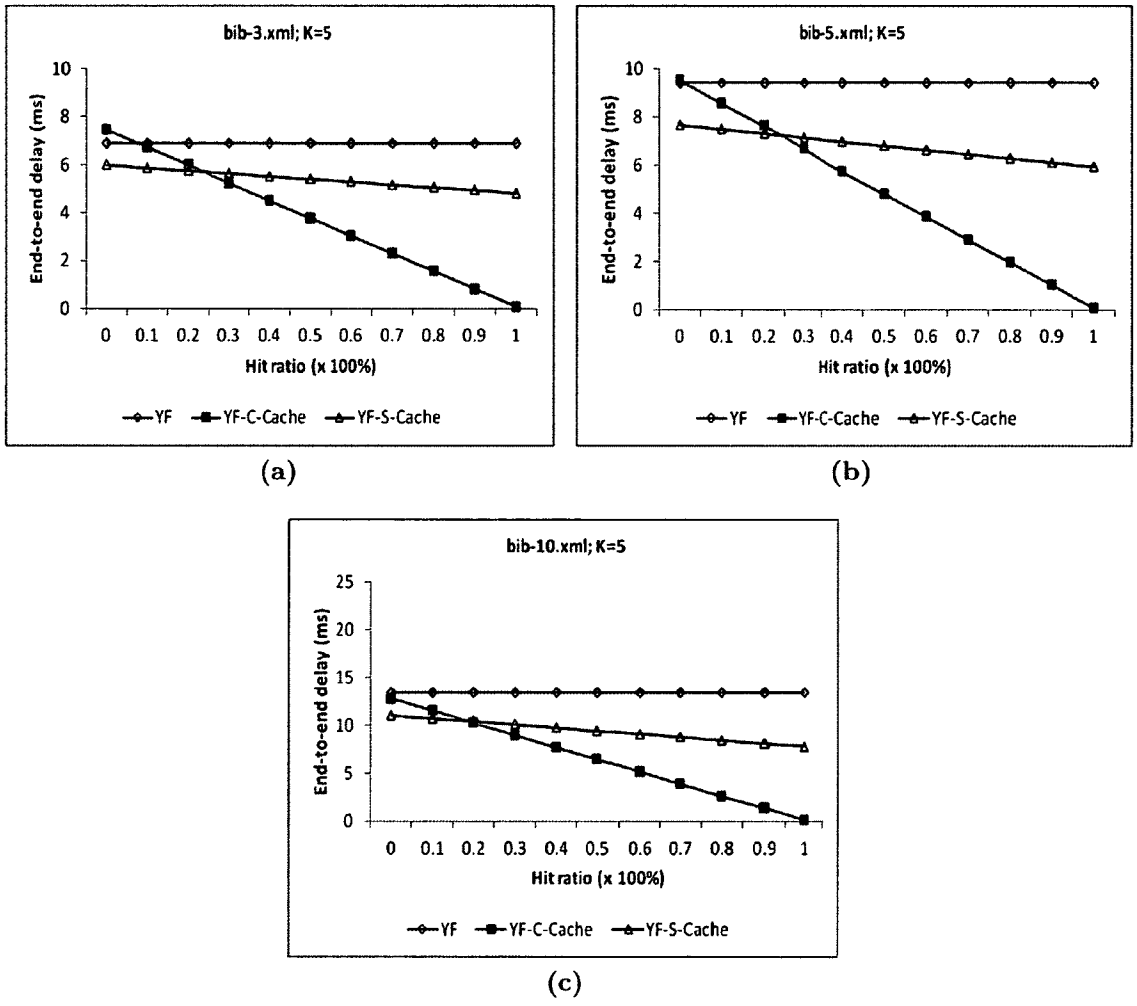


Figure A.4: E2E delay for bib messages for K=5 part 1: (a)bib-3.xml; (b)bib-5.xml; (c)bib-10.xml; (a)bib-15.xml; (b)bib-50.xml; (c)bib-200.xml.

A.5 Average E2E delay for bib messages when K=5 part 2

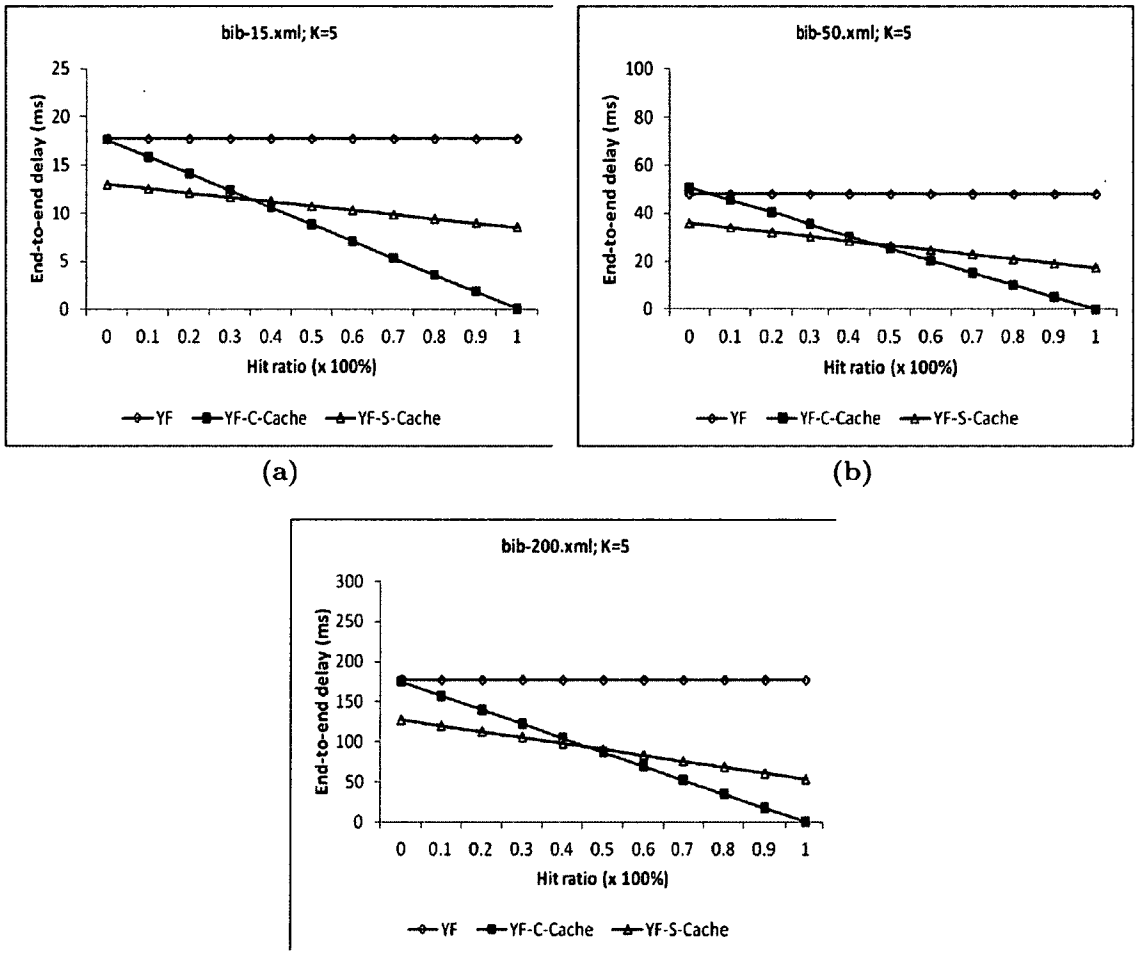
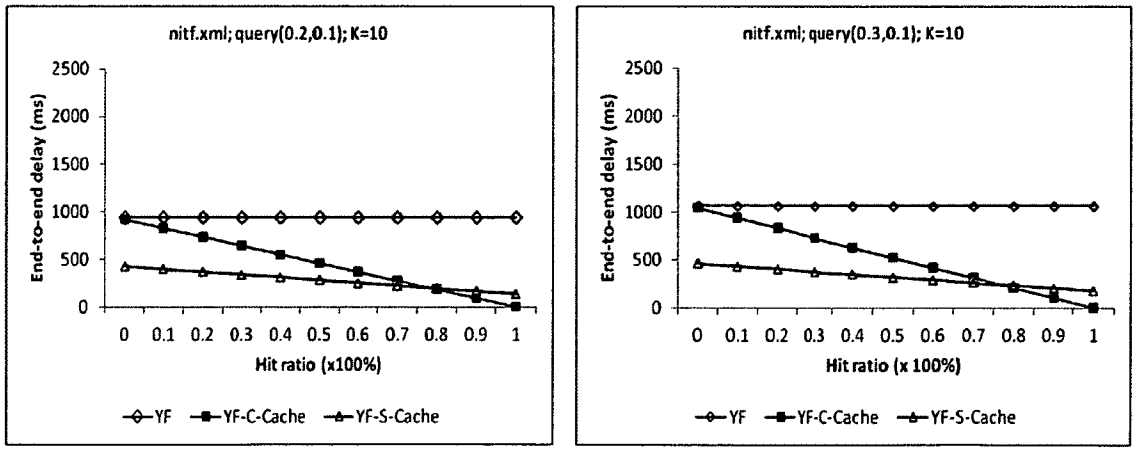


Figure A.5: E2E delay for bib messages for K=5 part 2: (a)bib-15.xml; (b)bib-50.xml; (c)bib-200.xml.

Appendix B

Supplementary caching synthetic message results for K=10

B.1 Average E2E delay for nitf messages when K=10 part 1



(a) (b)

Figure B.1: E2E delay for nitf messages for K=10 part 1: (a)Query(0.2,0.1); (b)Query(0.3,0.1);

B.2 Average E2E delay for nitf messages when K=10 part 2

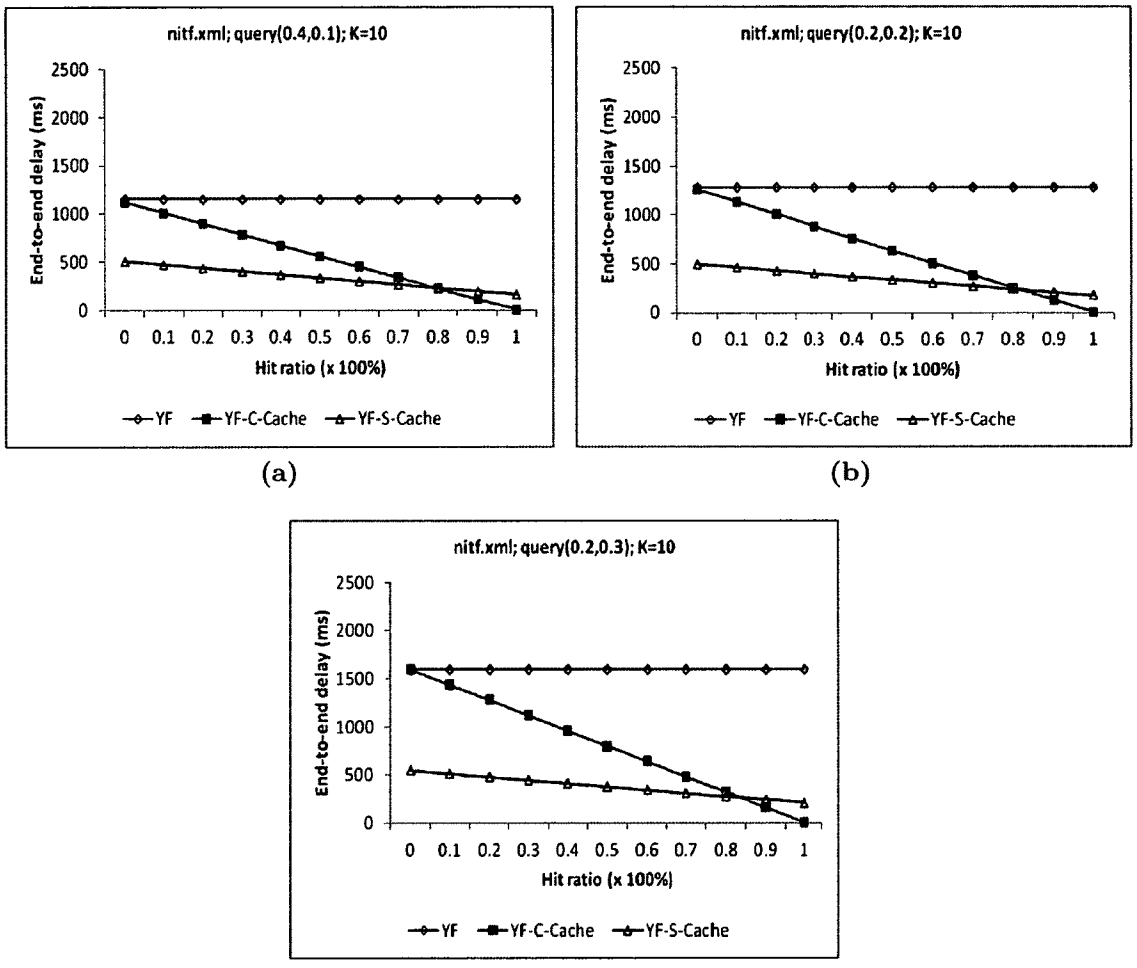


Figure B.2: E2E delay for nitf messages for K=10 part 2: (a)Query(0.4,0.1); (b)Query(0.2,0.2); (c)Query(0.2,0.3).

B.3 Average E2E delay for book messages when K=10

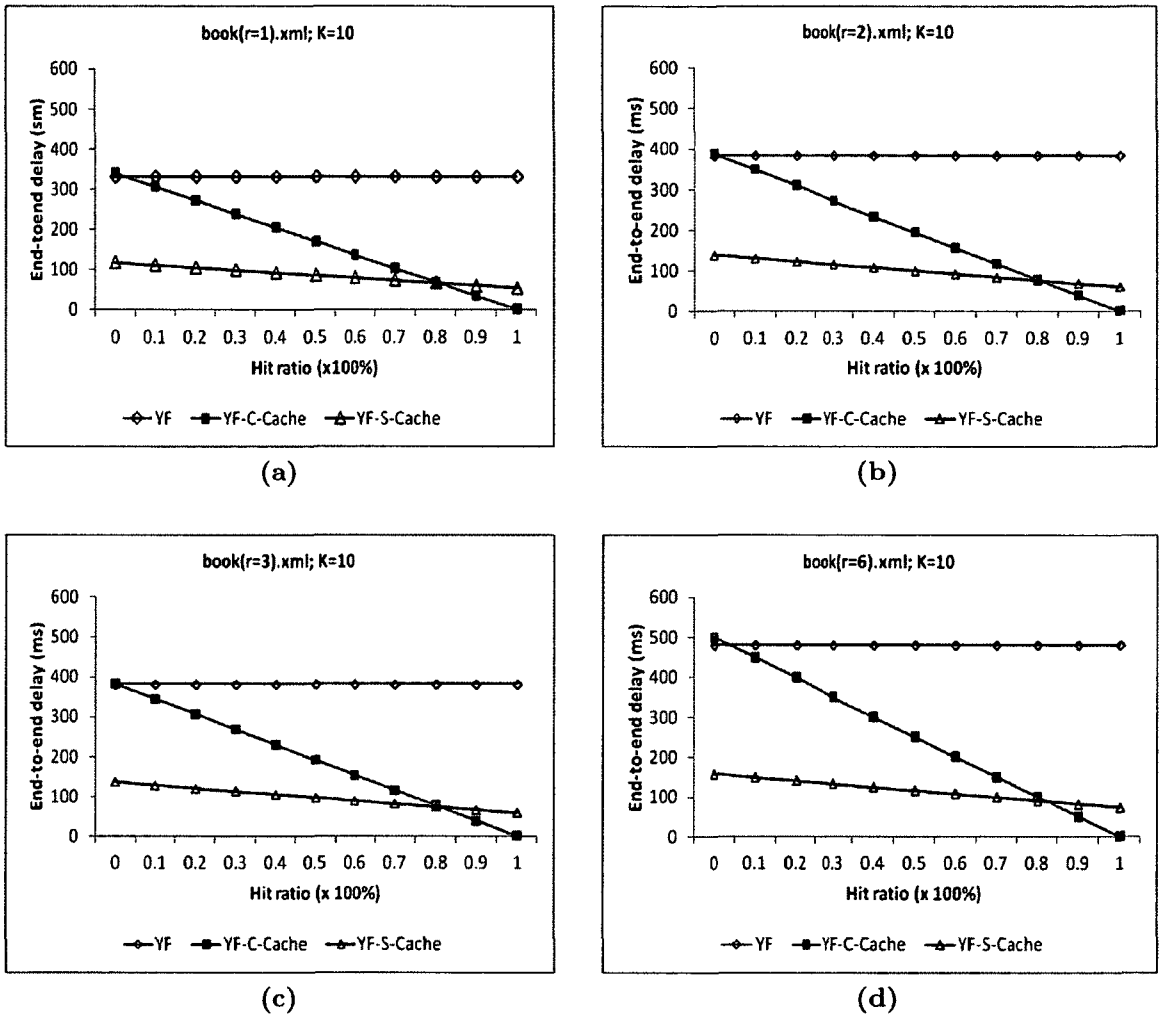


Figure B.3: E2E delay for book messages K=10: (a)book (r=1).xml; (b)book (r=2).xml; (c)book (r=3).xml; (d)book (r=6).xm;.

B.4 Average E2E delay for bib messages when K=10 part 1

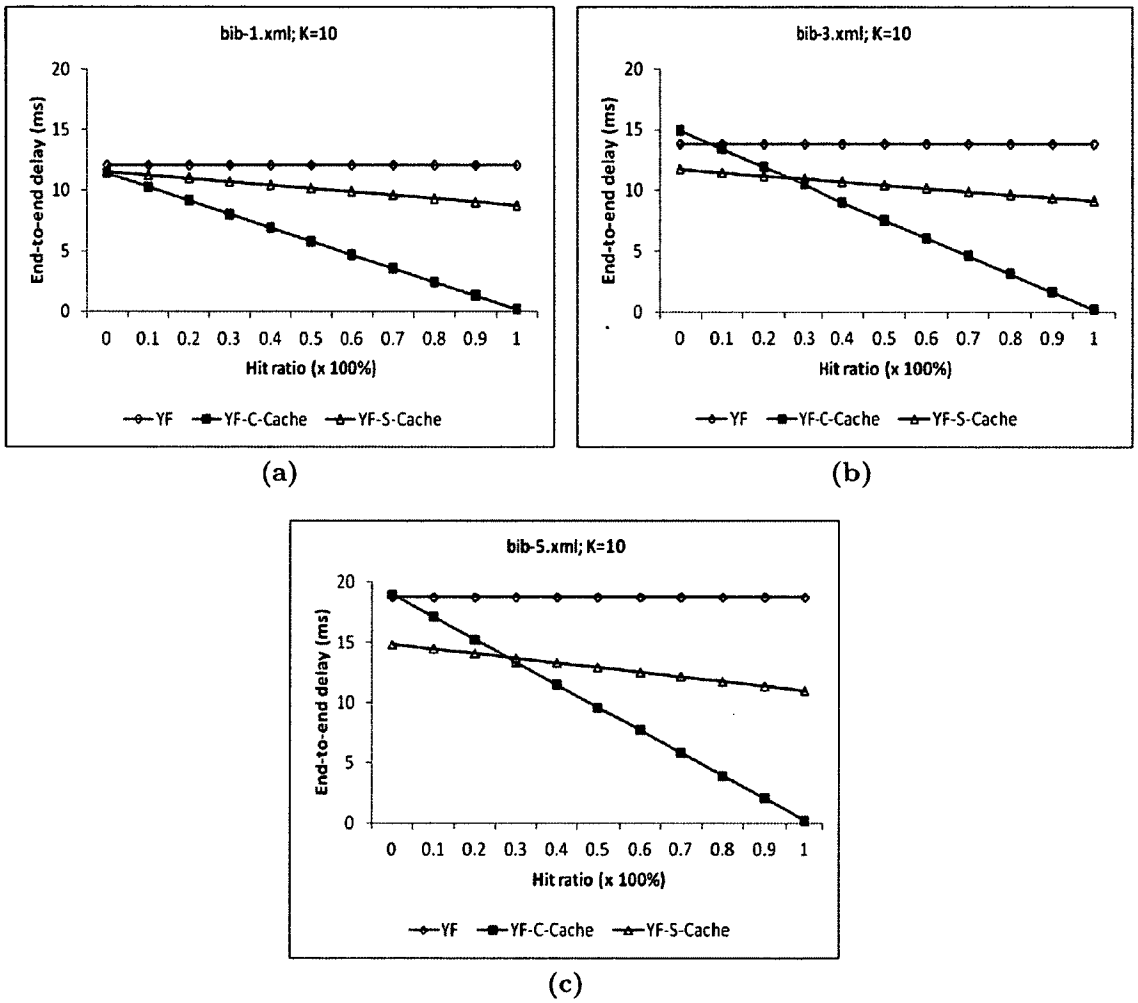


Figure B.4: E2E delay for bib messages for K=10 part 1: (a)bib-1.xml; (b)bib-3.xml; (c)bib-5.xml.

B.5 Average E2E delay for bib messages when K=10 part 2

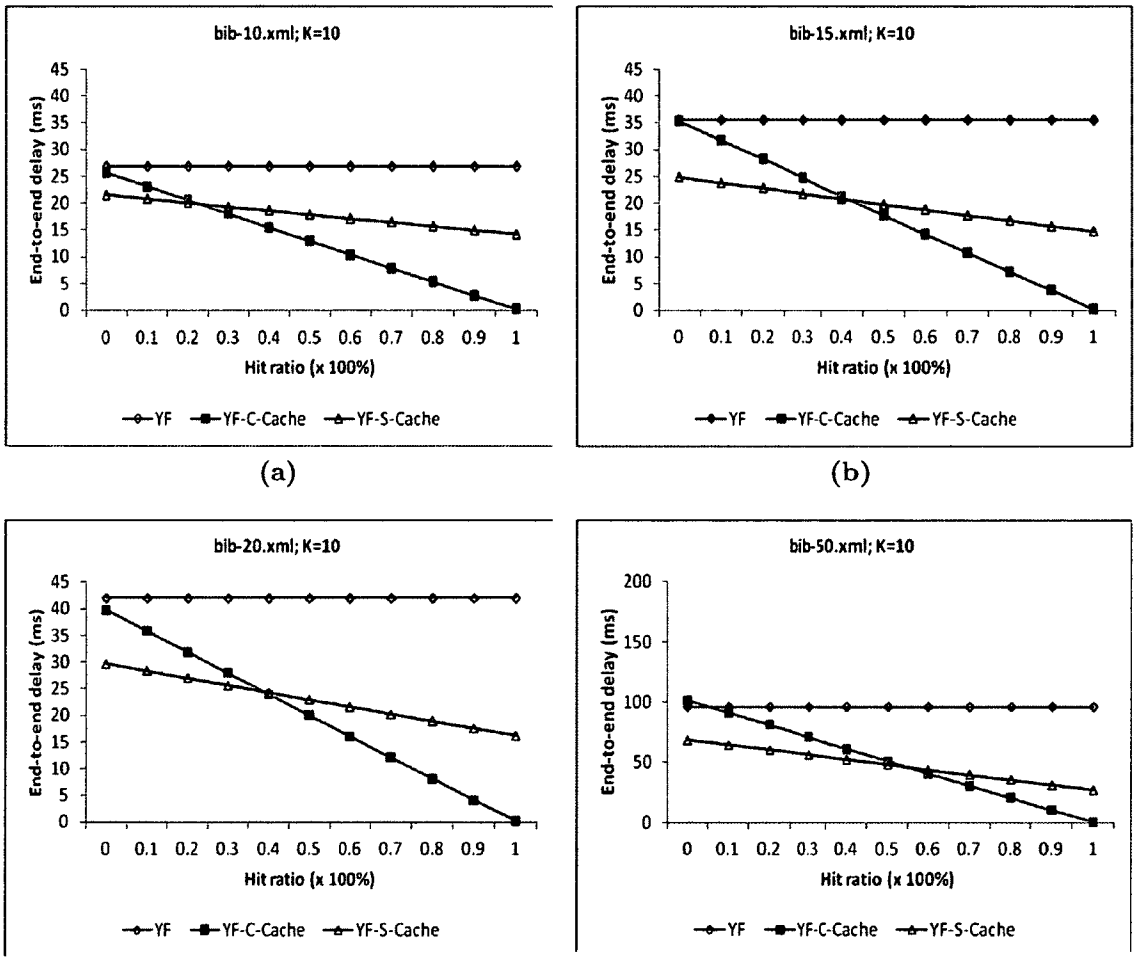


Figure B.5: E2E delay for bib messages for K=10 part 2: (a)bib-10.xml; (b)bib-15.xml; (c)bib-20.xml; (d)bib-50.xml.

B.6 Average E2E delay for bib messages when K=10 part 3

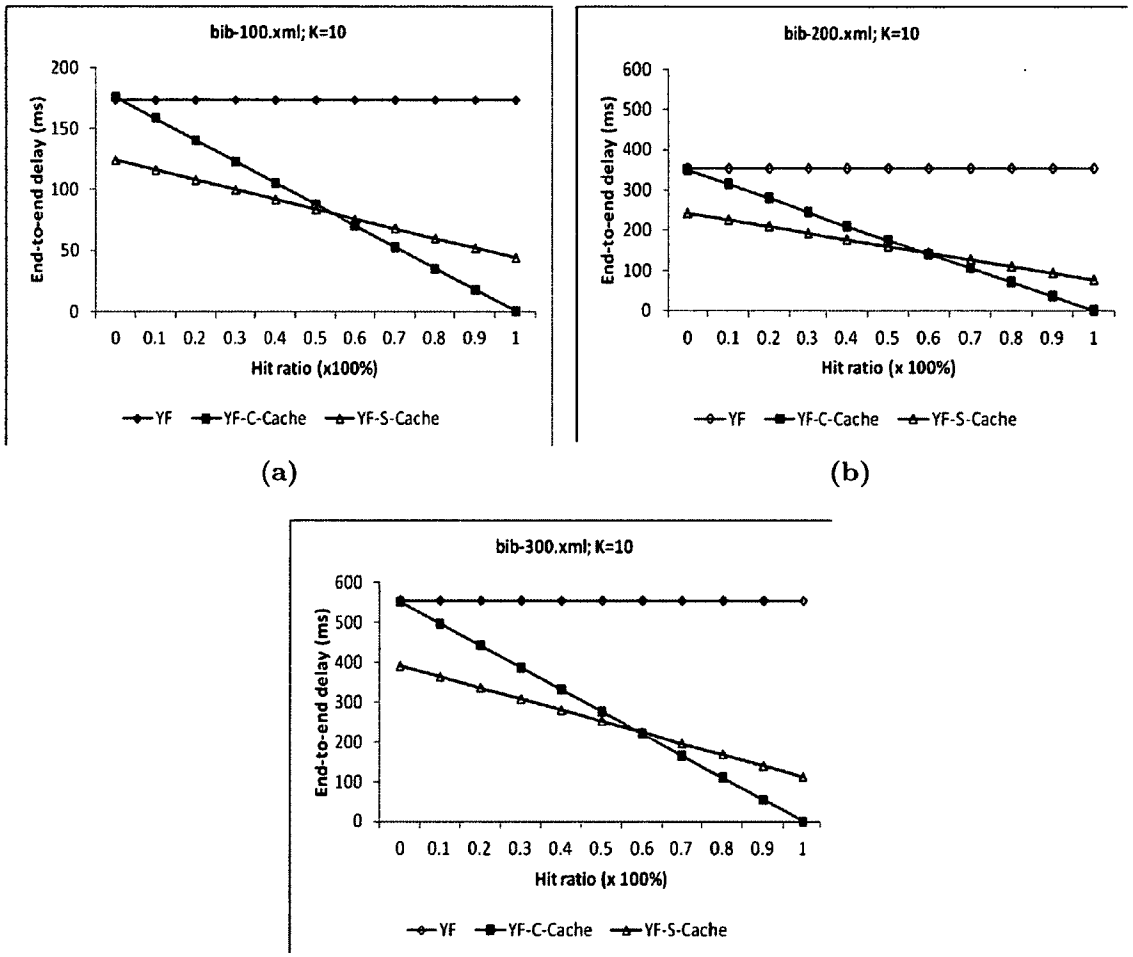


Figure B.6: E2E delay for bib messages for K=10 part 3: (a)bib-100.xml; (b)bib-200.xml; (c)bib-300.xml.

References

[1] Simjava. Available from: http://www.dcs.ed.ac.uk/home/hase/simjava/. accessed April 2012.

[2] XML tree. Available from: http://www.w3schools.com/xml/xml_tree.asp. accessed April 2012.

[3] XML path language (XPath) 1.0. Available from: http://www.w3.org/TR/xpath/, 1999. accessed April 2012.

[4] XML repository. Available from: http://www.cs.washington.edu/research/xmldatasets/www/repository.html, 2001. accessed April 2012.

[5] Extensible stylesheet language transformations (XSLT) 2.0. Available from: http://www.w3.org/TR/xslt20/, 2007. accessed April 2012.

[6] I.P.T. council. news industry text format. Available from: http://www.nitf.org/, 2007. accessed April 2012.

[7] W3C XQuery 1.0: An XML query language. Available from: http://www.w3.org/TR/xquery/, 2007. accessed April 2012.

[8] cURL groks URLs. Available from: http://curl.haxx.se/, 2009. accessed April 2012.

[9] Extensible markup language (XML). Available from: http://www.w3.org/XML/, 2009. accessed April 2012.

[10] Prüfer sequence. Available from: `http://en.wikipedia.org/wiki/Prufer_sequence`, 2009. accessed April 2012.

[11] Tib/rendezvous web site. Available from: `http://http://www.tibco.com/products/soa/messaging/rendezvous/default.jsp`, 2009. accessed April 2012.

[12] Web feed. Available from: `http://en.wikipedia.org/wiki/Web_feed`, 2009. accessed April 2012.

[13] XQuery 1.0: An XML query language (second edition). Available from: `http://www.w3.org/TR/xquery/`, 2010. accessed April 2012.

[14] Perform output caching with web services in visual c# .net. Available from: `http://support.microsoft.com/kb/318299`, September, 2003. accessed April 2012.

[15] Micah Adler, Zihui Ge, James F. Kurose, Don Towsley, and Stephen Zabele. Channelization problem in large scale data dissemination. In *Proc. of the ICNP*, pages 100–109, Riverside, California, USA, November 2001.

[16] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching events in a content-based subscription system. In *Proc. of the PODC*, pages 53–61, Atlanta, Georgia, USA, May 1999.

[17] Muath Alrammal, Gaètan Hains, and Mohamed Zergaoui. Intelligent ordered XPath for processing data streams. Technical report, Laboratory of Algorithmics, Complexity and Logic (LACL), Universitè Paris 12, France, July 2008. Available from: `lacl.univ-paris12.fr/Rapports/TR/TR-LACL-2008-4.pdf`.

[18] Mehmet Altinel and M.J.Franklin. Efficient filtering of XML documents for

selective dissemination of information. In *Proc. of the VLDB*, pages 53–64, Cairo, Egypt, September 2000.

[19] Panagiotis Antonellis and Christos Makris. XFIS: an XML filtering system based on string representation and matching. *Int. J. Web Eng. Technol.*, 4(1):70–94, 2008.

[20] Andrei Arion, Angela Bonifati, Ioana Manolescu, and Andrea Pugliese. Path summaries and path partitioning in modern XML databases. *World Wide Web*, 11(1):117–151, March 2008.

[21] James Aweya. IP router architectures: An overviews. *Journal of Systems Architecture*, 46:483–511, 1999.

[22] Roberto Baldoni. The publish/subscribe communication paradigm and its application to mobile systems. Available from:`http://www.slidefinder.net/t/the_publish_subscribe_communication_paradigm/baldoni_pub/10544754`, July 2005. accessed April 2012.

[23] Guruduth Banavar, Tushar Ch, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proc. of the ICDCS*, pages 262–272, Austin, Texas, USA, June 1999.

[24] Jerry Banks, John S. Carson II, Barry L. Nelson, and David M. Nicol. *Discrete-Event System Simulation (Third Edition).* Prentice Hall, 2000.

[25] Art Botterell. Common alerting protocol, version 1.0, committee draft. Available from: `http://www.oasis-open.org/committees/download.php/5666/emergency-CAP-1.0.pdf`, 2004. accessed April 2012.

[26] B.S. Bou-Diab. Virtual private network publish-subscribe multicast service. US patent, patent No: US7797382B2, Available from: `http://www.freepatentsonline.com/7797382.html`, Sept., 2010. accessed April 2012.

[27] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proc. of the SIGMOD*, pages 310–321, Madison, Wisconsin, USA, June 2002.

[28] K. Selçuk Candan, Wang-Pin Hsiung, Songting Chen, Junichi Tatemura, and Divyakant Agrawal. Afilter: adaptable XML filtering with prefix-caching suffix-clustering. In *Proc. of the VLDB*, pages 559–570, Seoul, Korea, September 2006.

[29] Fengyun Cao and Jaswinder Pal Singh. Medym: Match-early with dynamic multicast for content-based publish-subscribe networks. In *Proc. of the Middleware*, pages 292–313, Grenoble, France, December 2005.

[30] Fengyun Cao and Jaswinder Pal Singh. Medym: match-early and dynamic multicast for content-based publish-subscribe service networks. In *Proc. of the ICDCS Workshops (DEBS)*, pages 370–376, Columbus, Ohio, USA, June 2005.

[31] Yang Cao, Chung-Horng Lung, and Shikharesh Majumdar. A subscription coverage technique for XML message dissemination. In *Proc. of the SAINT*, pages 137–140, Seattle, USA, July 2009.

[32] Yang Cao, Chung-Horng Lung, and Shikharesh Majumdar. An XPath query aggregation algorithm using a region encoding. In *Proc. of the SAINT*, pages 27–36, Munich, Germany, July 2011.

[33] Yang Cao, Chung-Horng Lung, and Shikharesh Majumdar. A peer-to-peer model for XML publish/subscribe services. In *Proc. of the CNSR*, pages 26–32, Ottawa, Ontario, Canada, May 2011.

[34] Yang Cao, Shikharesh Majumdar, and Chung-Horng Lung. Caching techniques for XML message filtering. In *Proc. of the IPCCC*, pages 315–322, Pheonix, Arizona, USA, December 2009.

[35] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.

[36] Antonio Carzaniga, Matthew J. Rutherford, and Alexander L. Wolf. A routing scheme for content-based networking. In *Proc. of the INFOCOM*, pages 918–928, Hong Kong, China, March 2004.

[37] Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In *Proc. of the SIGCOMM*, pages 163–174, Karlsruhe, Germany, August 2003.

[38] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):1489–1499, 2002.

[39] Chee Yong Chan, Pascal Felber, Minos Garofalakis, and Rajeev Rastogi. Efficient filtering of XML documents with XPath expressions. *The VLDB Journal*, 11(4):354–379, December 2002.

[40] Chee Yong Chan and Yuan Ni. Efficient XML data dissemination with piggybacking. In *Proc. of the SIGMOD*, pages 737–748, Beijing, China, June 2007.

[41] Raphaël Chand. *Large dissemination of information in Publish-Subscribe systems.* Ph.d. thesis, University of Nice Sophia-Antipolis, 2005.

[42] Raphaël Chand and Pascal Felber. Scalable distribution of XML content with XNET. *IEEE Trans. Parallel Distrib. Syst.*, 19(4):447–461, 2008.

[43] Raphaël Chand and Pascal Felber. A scalable protocol for content-based routing in overlay networks. In *Proc. of the IEEE International Symposium on Network Computing and Applications (NCA)*, pages 123–130, Cambridge, Massachusetts, USA, April 2003.

[44] Raphaël Chand and Pascal Felber. XNET: A reliable content-based publish/subscribe system. In *Proc. of Symposium on Reliable Distributed Systems (SRDS)*, pages 264–273, Florence, Italy, October 2004.

[45] Badrish Chandramouli and Jun Yang. End-to-end support for joins in large-scale publish/subscribe systems. *In PVLDB*, 1(1):434–450, 2008.

[46] Badrish Chandramouli, Jun Yang, and Amin Vahdat. Distributed network querying with bounded approximate caching. In *Proc. of the DASFAA*, pages 374–388, Singapore, April 2006.

[47] Songting Chen, Hua-Gang Li, Jun'ichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K. Selçuk Candan. Scalable filtering of multiple generalized-tree-pattern queries over XML streams. *IEEE Trans. on Knowl. and Data Eng.*, 20(12):1627–1640, 2008.

[48] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Trans. Softw. Eng.*, 27(9):827–850, 2001.

[49] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proc. of the ICSE*, pages 261–270, Kyoto, Japan, April 1998.

[50] Liang Dai, Chung-Horng Lung, and Shikharesh Majumdar. Bfilter- a XML message filtering and matching approach in publish/subscribe systems. In *Proc. of the GLOBECOM*, pages 1–6, Miami, Florida, USA, December 2010.

[51] Ernesto Damiani and Stefania Marrara. Efficient SOAP message exchange and evaluation through XML similarity. In *Proc. of the ACM workshop on Secure web services*, pages 29–36, Alexandria, Virginia, USA, October 2008.

[52] M. Diallo, S. Fdida, V. Sourlas, P. Flegkas, and L. Tassiulas. Leveraging caching for internet-scale content-based publish/subscribe networks. In *Proc. of the ICC*, pages 1–5, Kyoto, Japan, June 2011.

[53] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter M. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.

[54] Yanlei Diao and Michael J. Franklin. Query processing for high-volume XML message brokering. In *Proc. of the VLDB*, pages 261–272, Berlin, Germany, September 2003.

[55] Yanlei Diao, Shariq Rizvi, and Michael J. Franklin. Towards an internet-scale XML dissemination service. In *Proc. of the VLDB*, pages 612–623, Toronto, Ontario, Canada, September 2004.

[56] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35:114–131, 2003.

[57] William Fenner and Divesh Srivastava. XTreeNet: Scalable overlay networks for XML content dissemination and querying. In *Proc. of the 10th International Workshop on Web Content Caching and Distribution (WCW)*, pages 41–46, Sophia Antipolis, French Riviera, France, September 2005.

[58] J. Fernandez, A. Fernandez, and J. Pazos. Optimizing web services performance using caching. In *International Conference on Next Generation Web Services Practices*, page 6, Seoul, Korea, August 2005.

[59] Ming Fu and Yu Zhang. Homomorphism resolving of XPath trees based on automata. In *Proc. of the APWeb/WAIM*, pages 821–828, Huang Shan, China, June 2007.

[60] Abdulbaset Gaddah and Thomas Kunz. Subscription aggregation for scalability and efficiency in XPath/XML-based publish/subscribe systems. Technical report, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, April 2008. Available from: `kunz-pc.sce.carleton.ca/thesis/XMLAggregation.pdf`.

[61] Xueqing Gong, Weining Qian, Ying Yan, and Aoying Zhou. Bloom filter-based XML packets filtering for millions of path queries. In *Proc. of the ICDE*, pages 890–901, Tokyo, Japan, April 2005.

[62] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata. In *Proc. of the ICDT*, pages 173–189, Siena, Italy, January 2003.

[63] John L. Guiney. Innovations and new technology for improved public weather services. Available from: `http://www.wmo.ch/pages/prog/amp/pwsp/documents/Guiney.pdf`, 2007. accessed April 2012.

[64] Rajeev Gupta and Krithi Ramamritham. Optimized query planning of continuous aggregation queries in dynamic data dissemination networks. In *Proc. of the International Conference on World Wide Web (WWW)*, pages 321–330, Banff, Alberta, Canada, May 2007.

[65] Wook-Shin Han, Haifeng Jiang, Howard Ho, and Quanzhong Li. Streamtx: extracting tuples from streaming XML data. *In PVLDB*, 1:289–300, 2008.

[66] Bingsheng He, Qiong Luo, and Byron Choi. Cache-conscious automata for XML filtering. *IEEE Trans. Knowl. Data Eng.*, 18(12):1629–1644, 2006.

[67] Michael Kay. *XSLT 2.0 and XPath 2.0 Programmer's Reference (Programmer to Programmer)*. Wrox Press Ltd., Birmingham, UK, 2008.

[68] Michael H. Kay. Saxon – the XSLT and XQuery processor. Available from: `http://www.saxonica.com/`, 2009. accessed April 2012.

[69] Georgia Koloniari and Evaggelia Pitoura. Bloom-based filters for hierarchical data. In *Proc. of the Workshop on Distributed Data and Structures (WDAS)*, pages 105–110, Toulouse, France, June 2003.

[70] Joonho Kwon, Praveen Rao, Bongki Moon, and Sukho Lee. Fist: Scalable XML document filtering by sequencing twig patterns. In *Proc. of the VLDB*, pages 217–228, Trondheim, Norway, September 2005.

[71] Guoli Li, Shuang Hou, and Hans-Arno Jacobsen. Routing of XML and XPath queries in data dissemination networks. In *Proc. of the ICDCS*, pages 627–638, Beijing, China, June 2008.

[72] Kostas Lillis and Evaggelia Pitoura. Cooperative XPath caching. In *Proc. of the SIGMOD*, pages 327–338, Vancouver, Canada, June 2008.

[73] Norman Lim, Shikharesh Majumdar, and Biswajit Nandy. Providing interoperability for resource access using web services. In *Proc. of the CNSR*, pages 236–243, Montreal, Canada, May 2010.

[74] P Mckee and I Marshall. Behavioral specification using XML. In *Proc. of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, page 53, Cape Town, South Africa, December 1999.

[75] Ralph Meijer. Publish-subscribe using jabber. In *XTech 2006: Building Web 2.0*, Amsterdam, Netherlands, May 2006.

[76] Gerome Miklau and Dan Suciu. Containment and equivalence for an XPath fragment. In *Proc. of the PODS*, pages 65–76, Madison, Wisconsin, USA, June 2002.

[77] Mirella Moura Moro, Zografoula Vagena, and Vassilis J. Tsotras. Recent advances and challenges in XML document routing. *Open and Novel Issues in XML Database Applications: Future Directions and Advanced Technologies, IGI, E. Pardede (Ed.)*, 2008. Available from: `http://www.cs.ucr.edu/~tsotras/meta-manager/Chapter-XML_DocRouting.pdf`, accessed April 2012.

[78] Mirella Moura Moro, Zografoula Vagena, and Vassilis J. Tsotras. XML structural summaries. *PVLDB*, 1(2):1524–1525, 2008.

[79] Frank Neven and Thomas Schwentick. XPath containment in the presence of Disjunction, DTDs, and Variables. In *Proc. of the ICDT*, pages 315–329, Siena, Italy, 2003.

[80] Sebastian Obermeier and Stefan Böttcher. XML fragment caching for large-scale mobile commerce applications. In *Proc. of the 10th international conference on Electronic commerce (ICEC)*, pages 1–7, Innsbruck, Austria, August 2008.

[81] Lukasz Opyrchal, Mark Astley, Joshua Auerbach, Guruduth Banavar, Robert Strom, and Daniel Sturman. Exploiting IP multicast in content-based publish-subscribe systems. In *Proc. of the Middleware*, pages 185–207, New York, New York, USA, April 2000.

[82] Olga Papaemmanouil and Uğur Çetintemel. Semcast: Semantic multicast for content-based data dissemination. In *Proc. of the ICDE*, pages 242–253, Tokyo, Japan, April 2005.

[83] Mark Perry, Christophe Delporte, Federico Demi, Animesh Ghosh, and Marc Luong. Mqseries publish/subscribe applications. Available from: `http://www.redbooks.ibm.com/redbooks/pdfs/sg246282.pdf`, 2001. accessed April 2012.

[84] Peter Pietzuch, David Eyers, Samuel Kounev, and Brian Shand. Towards a common api for publish/subscribe. In *Proc. of the ICDCS Workshops (DEBS)*, pages 152–157, Toronto, Ontario, Canada, June 2007.

[85] Peter R. Pietzuch and Jean Bacon. Hermes: A distributed event-based middleware architecture. In *Proc. of the ICDCS workshops*, pages 611–618, Vienna, Austria, July 2002.

[86] Pawel Placek, Dimitri Theodoratos, Stefanos Souldatos, Theodore Dalamagas, and Timos Sellis. A heuristic approach for checking containment of generalized tree-pattern queries. In *Proc. of the CIKM*, pages 551–560, Napa Valley, California, USA, October 2008.

[87] Aneesh Raj and P. Sreenivasa Kumar. Branch sequencing based XML message broker architecture. In *Proc. of the ICDE*, pages 656–665, Istanbul, Turkey, April 2007.

[88] Praveen Rao, Justin Cappos, Varun Khare, Bongki Moon, and Beichuan Zhang. Net-X: Unified data-centric internet services. In *Third International Workshop on Networking Meets Databases (NetDB 2007)*, pages 1–6, Cambridge, MA, USA, April 2007.

[89] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proc. of the SIGCOMM*, pages 161–172, San Diego, California, USA, August 2001.

[90] Anton Riabov, Zhen Liu, Joel L. Wolf, Philip S. Yu, and Li Zhang. Clustering algorithms for content-based publication-subscription systems. In *Proc. of the ICDCS*, pages 133–142, Vienna, Austria, July 2002.

[91] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proc. of the Middleware*, pages 329–350, Heidelberg, Germany, November 2001.

[92] Mariam Salloum and Vassilis J. Tsotras. Efficient and scalable sequence-based XML filtering. In *Proc. of the WebDB*, Providence, Rhode Island, USA, July 2009.

[93] Thomas Schwentick. XPath query containment. *SIGMOD Rec.*, 33(1):101–109, 2004.

[94] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts 7th Edition with Java 7th Edition.* John Wiley & Sons, 2006.

[95] Panu Silvasti, Seppo Sippu, and Eljas Soisalon-Soininen. Schema-conscious filtering of XML documents. In *Proc. of the EDBT*, pages 970–981, Saint Petersburg, Russia, March 2009.

[96] Vasilis Sourlas, Georgios S. Paschos, Paris Flegkas, and Leandros Tassiulas. Caching in content-based publish/subscribe systems. In *Proc. of the GLOBECOM*, pages 1–6, Honolulu, Hawaii, USA, December 2009.

[97] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of the SIGCOMM*, pages 149–160, San Diego, California, USA, August 2001.

[98] Samini Subramaniam, Su-Cheng Haw, and Poo Kuan Hoong. Mapping and labeling XML data for dynamic update. In *International Conference on Computer Research and Development*, pages 781–786, Kuala Lumpur, Malaysia, May 2010.

[99] Zografoula Vagena, Mirella Moura Moro, and Vassilis J. Tsotras. RoXSum: Leveraging data aggregation and batch processing for XML routing. In *Proc. of the ICDE*, pages 1466–1470, Istanbul, Turkey, April 2007.

[100] Zografoula Vagena, Mirella Moura Moro, and Vassilis J. Tsotras. Value-aware RoXSum: Effective message aggregation for XML-aware information dissemination. In *Proc. of the WebDB*, Beijing, China, June 2007.

[101] Yi-Min Wang, Lili Qiu, Dimitris Achlioptas, Gautam Das, Paul Larson, and Helen J. Wang. Subscription partitioning and routing in content-based publish/subscribe network. In *Proc. of the International Symposium on Distributed Computing (DISC)*, Toulouse, France, October 2002.

[102] Yi-Min Wang, Lili Qiu, Chad Verbowski, Dimitris Achlioptas, Gautam Das, and Paul Larson. Summary-based routing for content-based event distribution networks. *SIGCOMM Comput. Commun. Rev.*, 34(5):59–74, 2004.

[103] Tina Wong, Randy Katz, and Steven McCanne. An evaluation of preference clustering in large-scale multicast applications. In *Proc. of the INFOCOM*, volume 2, pages 451–460, Tel Aviv, Israel, March 2000.

[104] Derrick Wood. *Theory of Computation: A Primer*. Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusetts, USA, 1987.

[105] Peter T. Wood. Containment for XPath fragments under DTD constraints. In *Proc. of the ICDT*, pages 300–314, Siena, Italy, January 2003.

[106] Eiko Yoneki and Jean Bacon. Content-based routing with on-demand multicast. In *Proc. of the ICDCS workshops*, pages 788–793, Hachioji, Tokyo, Japan, March 2004.

[107] Sanghyun Yoo, Jin Hyun Son, and Myoung Ho Kim. An efficient subscription

routing algorithm for scalable XML-based publish/subscribe systems. *J. Syst. Softw.*, 79(12):1767–1781, 2006.

[108] Jung-Hee Yun and Chin-Wan Chung. Dynamic interval-based labeling scheme for efficient XML query and update processing. *J. Syst. Softw.*, 81:56–70, January 2008.

[109] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, pages 41–53, 2003.