Hardware Architecture of an XML/XPath Broker/Router for Content-Based Publish/Subscribe Data Dissemination Systems

Fadi El-Hassan

Thesis submitted to the Faculty of Graduate and Postdoctoral Studies

In partial fulfillment of the requirements for the degree of

Ph.D. in Electrical and Computer Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

School of Electrical Engineering and Computer Science

University of Ottawa

February 2014

© Fadi El-Hassan, Ottawa, Canada, 2014

I hereby declare that I am the sole author of this thesis.

I authorize the University of Ottawa to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Fadi El-Hassan

I further authorize the University of Ottawa to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Fadi El-Hassan

Abstract

The dissemination of various types of data faces ongoing challenges with the growing need of accessing manifold information. Since the interest in content is what drives data networks, some new technologies and thoughts attempt to cope with these challenges by developing content-based rather than address-based architectures. The Publish/Subscribe paradigm can be a promising approach toward content-based data dissemination, especially that it provides total decoupling between publishers and subscribers. However, in content-based publish/subscribe systems, subscriptions are expressive and the information is often delivered based on the matched expressive content - which may not deeply alleviate considerable performance challenges.

This dissertation explores a hardware solution for disseminating data in content-based publish/subscribe systems. This solution consists of an efficient hardware architecture of an XML/XPath broker that can route information based on content to either other XML/XPath brokers or to ultimate users. A network of such brokers represent an overlay structure for XML content-based publish/subscribe data dissemination systems. Each broker can simultaneously process many XPath subscriptions, efficiently parse XML publications, and subsequently forward notifications that result from high-performance matching processes. In the core of the broker architecture, locates an XML parser that utilizes a novel Skeleton CAM-Based XML Parsing (SCBXP) technique in addition to an XPath processor and a high-performance matching engine. Moreover, the broker employs effective mechanisms for content-based routing, so as subscriptions, publications, and notifications are routed through the network based on content.

The inherent reconfigurability feature of the broker's hardware provides the system architecture with the capability of residing in any FPGA device of moderate logic density. Furthermore, such a system-on-chip architecture is upgradable, if any future hardware add-ons are needed. However, the current architecture is mature and can effectively be implemented on an ASIC device.

Finally, this thesis presents and analyzes the experiments conducted on an FPGA prototype implementation of the proposed broker/router. The experiments tackle tests for the SCBXP alone and for two phases of development of the whole broker. The corresponding results indicate the high performance that the involved parsing, storing, matching, and routing processes can achieve.

.... وَقُل رَّبِّ زِدْنِي عِلْمًا ﴿١١٤﴾

.... **and say, "My Lord, increase me in knowledge"**

Quran:(Ta-Ha - 114)

I would like to dedicate this thesis:

To the souls of my dear parents who were my first beloved teachers in my life

and who both passed away in July 2013 before they could see this thesis

Mr. Taleb and Mrs. Hind

to whom I am so grateful

and of whom I am so proud.

To my great wife whom I love so much

Inaam

who emotionally supported me through my Ph.D. journey

and who patiently endured the considerable busy times I had.

To my two children Meera and Baseem (Junior Taleb)

who always bring wide smiles to my face

and who are the hope of the future.

Acknowledgements

First of all, I praise The Almighty Allah, Glorified and Exalted, for having bestowed on me the strength and patience, without which it would not have been possible for me to carry out my thesis.

I owe my deepest gratitude to my supervisor Dr. Dan Ionescu for his great and continuous support throughout this research. Without this vital support, I would not have been able to pursue this work.

I would like to thankfully acknowledge University of Ottawa, especially the School of Electrical Engineering and Computer Science (EECS) and the Faculty of Graduate and Postdoctoral Studies (FGPS), for the financial support that I had received and for the opportunity of gaining experience in teaching assistantship.

I am so grateful for the recommendations and valuable feedback that I have received from the committee members, Dr. Abdulmotaleb El-Saddik, Dr. Voicu Groza, Dr. Ashraf Matrawy, and Dr. Nicola Nicolici.

I am also grateful for the insight discussions with all members of NCCT lab, especially Raymond Peterkin and Mohamed Abou-Gabal.

My sincere thanks go to the professors whom I have met either during my teaching assistantship or in the courses that I have attended, especially Dr. Voicu Groza, Dr. Ahmed Karmouch, Dr. Emad Gad, Dr. Carlisle Adams, and Dr. Guy-Vincent Jourdan.

I offer my regards to the chair of the committee Dr. Jonathan Linton, the FGPS academic assistant Lise Dazé, and the retiree EECS administrative assistant Michèle Roy.

Contents

1 Introduction **1**

1.1 Motivations 1

1.2 Problems and Requirements for Solutions 4

1.3 Thesis Contributions 5

1.4 Thesis Organization 7

2 Publish/Subscribe Systems Review **8**

2.1 Definition 8

2.2 Types of Publish/Subscribe Systems 10

2.2.1 Channel-Based 10

2.2.2 Topic-Based 11

2.2.3 Type-Based 11

2.2.4 Content-Based 12

2.3 Existing Publish/Subscribe Systems 13

2.3.1 Non-XML-based Systems over non-P2P Networks 14

2.3.2 Non-XML-based Systems over P2P Networks 17

2.3.3 XML-based Filtering and Routing Systems 20

2.3.4 Other Systems 23

2.3.5 Summary of Existing Systems 25
2.4 Issues and Drawbacks of Existing Systems 26
2.4.1 Performance 26
2.4.2 Interoperability 27
2.4.3 Scalability 28
2.4.4 Other Issues 28
2.5 Proposed System versus Existing Systems 28
2.6 XML Parsing Systems 30
2.6.1 Existing XML Parsing Techniques 30
2.6.2 SCBXP versus Existing XML Parsing Techniques 33
2.7 XML/XPath Filtering Systems 33
2.7.1 Existing XML/XPath Filtering Systems 34
2.7.2 Proposed XML/XPath Filtering versus Existing Systems 35

3 Proposed XML Content-Based Publish/Subscribe System 36

3.1 XML/XPath Systems 36
3.1.1 Basic Formal Description 38
3.2 High-Level Architecture 38
3.2.1 Basic Functionality 39
3.2.2 Concurrent and Sequential Tasks 40
3.2.3 High-Level Block Diagram 41
3.2.4 Content-Based Routing 41
3.2.5 Multimedia Support 43

4 XML Parsing: The Hardware SCBXP Technique 47

4.1 Introduction to XML Parsing 47

4.2 SCBXP Overview . . . 50
4.3 XML Skeleton . . . 54
4.4 The SCBXP Internal Architecture . . . 57
4.4.1 Loading and Reading Stages: The Two Dual-Port Memory Modules 58
4.4.2 Aligning Stage: The FIFO Control Module . . . 59
4.4.3 Aligning Stage: The XML Aligning State Machine . . . 63
4.4.4 Matching Stage: The CAM Logic and Module . . . 66
4.4.5 Post-Matching Stage: The Multiple Parsing State Machines . . . 69
4.4.6 Scheduling/Writing Stage: The Parsed Words Scheduler . . . 72
4.5 Processing Time . . . 73
4.5.1 Case I: One Common Skeleton, Multiple Chunks of XML Data . . . 75
4.5.2 Case II: Multiple Skeletons, Multiple Chunks of XML Data . . . 76
5 Architecture of the Hardware XML/XPath Broker/Router 77
5.1 Introduction . . . 77
5.2 Main Broker Tasks . . . 79
5.2.1 Definitions . . . 79
5.2.2 Routing Tasks . . . 81
5.3 The Hardware Broker Architecture . . . 82
5.3.1 Basic Overview . . . 83
5.3.2 Part A: Subscription Processing Unit (SPU) . . . 87
5.3.3 Part B: XML/XPath Processing and Matching Unit (PMU) . . . 94
5.3.4 Part C: Notification Processing Unit (NPU) . . . 106
5.4 Content-Based Routing Tasks of the Broker . . . 108
5.4.1 Separation Between Matching and Routing . . . 108

5.4.2 Identity-Based Matching versus Identity-Based Routing 108
5.4.3 Content-Based Routing Mechanisms 111
5.4.4 Routing Scenarios and Corresponding Performance 142
5.4.5 Link Management 152
5.4.6 Network Scopes 154

6 Experimental Results of the Hardware SCBXP Technique 162

6.1 Implementation Note 162
6.2 FPGA Resources Utilization and Performance 163
6.3 How to Improve Performance? 163
6.4 Testing with Hardware Interface 165
6.4.1 Case I 166
6.4.2 Case II 167
6.4.3 Comparison between Case I and Case II 167
6.5 Testing with Software/Hardware Interface 169
6.5.1 Case A 170
6.5.2 Case B 170
6.5.3 Case C 171
6.5.4 Case D 172
6.6 Results Conclusion 172
6.7 Advantages and Limitations 174
6.7.1 Advantages 174
6.7.2 Limitations 175
6.8 SCBXP Conclusion 176

7 Experimental Results of the Hardware XML/XPath Broker/Router 178

7.1 Experimental Setup for Phase 1 Tests 180

7.2 Stratix FPGA Resources Utilization 182

7.3 Power Dissipation on Stratix FPGA 183

7.4 Results Discussion for Phase 1 183

7.5 Broker's Phase 1 Conclusion 187

7.6 Experimental Setup Interfaces for Phase 2 Tests 188

7.6.1 Avalon Interfaces 188

7.6.2 Hardware Interface 190

7.6.3 Software/Hardware Interface 191

7.6.4 Software/Hardware Interface with Content-Based Routing 195

7.7 Experimental Setup of Subscription Sets for Phase 2 Tests 199

7.8 Case I: Testing Through the Hardware Interface 200

7.8.1 Results of Case I 201

7.9 Case II: Testing Through the Software/Hardware Interface 210

7.9.1 Results of Case II 210

7.10 Case III: Testing Through the Software/Hardware Interface with Content-Based Routing 214

7.10.1 Results of Case III 215

7.11 Cyclone IV E FPGA Resources Utilization 222

7.12 Power Dissipation on Cyclone IV E FPGA 223

7.13 Broker's Phase 2 Conclusion 223

8 Conclusion and Future Work 225

8.1 Applicability in Wide-Area Networks 226

8.1.1 Content-Based Routing With or Without TCP/IP 227
8.2 Interfacing Multiple X2CBBRs 229
8.3 Security Challenges 230

List of Tables

2.1 Characteristics of Existing Pub/Sub Systems 27

2.2 Comparison Between X2CBBR Pub/Sub System and Some Existing Systems 29

4.1 Example of A Derived Tokenized Skeleton 54

6.1 Case I 166

6.2 Case II 168

6.3 SCBXP Performance: HW Interface vs. SW/HW Interface 173

6.4 SCBXP Performance: HW Interface vs. SAX-based Software 173

6.5 SCBXP Performance: SW/HW Interface vs. SAX-based Software 173

7.1 Broker Results 184

7.2 Worst-Case Results of the Broker with Hardware Interface 203

7.3 Regular-Case Results of the Broker with Hardware Interface 204

7.4 Best-Case Results of the Broker with Hardware Interface 205

7.5 Worst-Case Results of the Broker with Software/Hardware Interface . . . 211

7.6 Regular-Case Results of the Broker with Software/Hardware Interface . . 212

7.7 Best-Case Results of the Broker with Software/Hardware Interface 213

7.8 Results of the Broker with Software/Hardware Interface and Subscription Routing 216

7.9 Results of the Broker with Software/Hardware Interface and Subscription/Publication Routing 219
7.10 Minimum Time Consumed in the First and the Second Experiments of Case III 221
7.11 Overall Performance Comparison Between the First and the Second Experiments of Case III 221

List of Figures

1.1 Bottlenecks. 4

2.1 Overview of a Publish/Subscribe System. 9

2.2 Centralized versus Distributed. 9

3.1 Distributed Pub/Sub System with Reconfigurable Brokers. 39

3.2 The Hardware Broker and its Interfaces in the System. 40

3.3 Block Diagram 42

3.4 The Hardware Broker and its Interfaces in the System. 45

4.1 Basic Block Diagram of the SCBXP Technique. 51

4.2 Flow Diagram for the First Three Stages of the SCBXP 53

4.3 An XML Example. 54

4.4 SCBXP Internal Architecture. 57

4.5 Rules of FIFO Alignment 60

4.6 Graphical Illustration of FIFO Rules. 61

4.7 Two Examples of Strings Aligned in FIFOs 61

4.8 The Harmony Between Fifos and both the Aligning and Matching Stages 62

4.9 One State of the Aligning State Machine. 63

4.10 Aligning State Machine. 64

4.11 CAM of SCBXP. 67
4.12 Example of Well-Aligned Data 68
4.13 One of the Parsing State Machines. 70
4.14 Multiple Parsing State Machines. 70
4.15 Example of The Timing Schedule of the Multiple State Machines 71

5.1 Edge and Intermediate Brokers. 81
5.2 Block Architecture of the Reconfigurable Hardware Broker. 83
5.3 Flow Diagram 86
5.4 Processing Stages of a Subscription. 87
5.5 Simple Subscription Buffer. 88
5.6 Double Subscription Buffer. 88
5.7 Structure of an XPath Subscription. 90
5.8 Structure of the Subscription Mapping Table. 91
5.9 The FSM of the Subscription Forwarder. 94
5.10 Hardware XPath Processor Architecture. 95
5.11 CAM Internal Structure. 96
5.12 Example of Four Subscriptions Registered in the CRT. 99
5.13 Subscription IDs Memory structure. 100
5.14 Subscription Registering Algorithm (SReA). 102
5.15 XML/XPath Matching Engine. 105
5.16 Local Notification Routing Algorithm. 107
5.17 Determination of the Maximum Number of Subscriptions. 113
5.18 Operation Mode FSM. 115
5.19 Diagram Illustrating Task Concurrency. 119

5.20 Pseudo-Code for Top-Level Algorithm. 120
5.21 Structure of Local and Remote Subscriptions. 126
5.22 Local/Remote Notification/Publication Routing Algorithm. 128
5.23 Notification Re-Routing Algorithm. 130
5.24 Subscription Routing Algorithm (SRoA). 132
5.25 Storing/Registering Two Subscriptions in a Broker. 134
5.26 Matching Two Subscriptions and Notifying Corresponding Subscribers. . 135
5.27 Scenario 1-a. 143
5.28 Scenario 1-b. 144
5.29 Scenario 2. 145
5.30 Scenario 3. 146
5.31 Scenario 4. 147
5.32 Scenario 5. 148
5.33 Software/Hardware Interface and Link Management. 153
5.34 Illustration of Network Scopes. 156

6.1 Hardware-Testing Framework. 165
6.2 Graphical Representation of Throughput Results 168
6.3 Software/Hardware-Testing Framework. 169

7.1 Hardware-Testing Framework for the Broker. 180
7.2 Graphical Representation of Broker's Processing Time. 185
7.3 Graphical Representation of Broker's Throughput. 185
7.4 Avalon Read and Write Transfers with Waitrequest. 189
7.5 Broker's Hardware Interface. 190
7.6 Broker's Software/Hardware Interface. 191

7.7 The Architectural Layers of a Nios II Software Application. 192

7.8 The Broker/Router Connected to a Slave Port in the SOPC Builder Tool. 194

7.9 Triple-Speed Ethernet Driving Architecture. 196

7.10 Broker to Producer/Consumer Connectivity. 197

7.11 Broker to Broker Connectivity. 198

7.12 Broker to Broker Connectivity. 198

7.13 Clocking Strategy. 201

7.14 Processing Time in the Worst, Regular, and Best Cases. 206

7.15 Throughput in the Worst, Regular, and Best Cases. 206

7.16 Performance in the Case of One Pub. versus the First 2000 Subs. 208

7.17 Performance in the Case of One Pub. versus One Million Subs. 209

List of Acronyms

ALM	Application-Level Multicast
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASIC	Application-Specific Integrated Circuit
Avalon-MM	Avalon-Memory-Mapped
Avalon-ST	Avalon-streaming
BCL	Bluespec Codesign Language
CAM	Content Addressable Memory
CAN	Content Addressable Network
CBR	Content-Based Routing
X2CBBR	XML/XPath Content-Based Broker/Router
CORBA	Common Object Request Broker Architecture
CORONA	Cornell Online News Aggregator
CPU	Central Processing Unit
CRT	Content Routing Table
DCP	Diversity Control Protocol
DDS	Data Distribution Service
DHT	Distributed Hash Table
DMA	Direct Memory Access
DNS	Domain Name System
DOM	Document Object Model

DoS	Denial-of-Service
DPM	Dual-Port Memory
DPSS	Distributed Publish/Subscribe System
DSL	Digital Subscriber Line
DTD	Document Type Definition
FIFO	First-In First-Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
Gbps	Gigabit per second
HAL	Hardware Abstraction Layer
HCD	High Commonality Degree
HTML	HyperText Markup Language
Hz	Hertz
ICN	Information-Centric Networking
IEEE	Institute of Electrical and Electronics Engineers
IM	Instant Messaging
IMS	Internet Multimedia Subsystem
IP	Internet Protocol
IP core	Intellectual Property core
JEDI	Java Event-based Distributed Infrastructure
JMS	Java Message Service
Kbps	Kilobit per second
LAN	Local-Area Network
LCD	Liquid Crystal Display

LDAP	Lightweight Directory Access Protocol
LE	Logic Element
LightPS	Lightweight content-based Publish/Subscribe
MAC	Media Access Controller
MAN	Metropolitan-Area Network
Mbps	Megabit per second
MEDYM	Match-Early with DYnamic Multicast
MHz	Mega Hertz
MicroC/OS II	Micro-Controller Operating System
μC/OS-II	Micro-Controller Operating System
OMG	Object Management Group
ONYX	Operator Network using YFilter for XML dissemination
P2P	Peer-to-Peer
PADRES	Publish/subscribe Applied to Distributed REsource Scheduling
PCI	Peripheral Component Interconnect
PCIe	PCI Express
PCRE	PERL Compatible Regular Expression
PEM	Popularity-based Event Matching
PI	Processing Instruction
PIDF	Presence Information Data Format
PLL	Phase-Locked Loop
PSIRP	Publish-Subscribe Internet Routing Paradigm
QoS	Quality of Service
RAM	Random Access Memory

RAX	Random Access XML
RMI	Remote Method Invocation
RP	Rendezvous Point
RTL	Register Transfer Logic
RTOS	Real-Time Operating System
RTPS	Real-Time Publish Subscribe
RV	Rendezvous
SAX	Simple API for XML
SCBXP	Skeleton-CAM-Based XML Parsing
SDRAM	Synchronous Dynamic Random Access Memory
SG-DMA	Scatter-Gather Direct Memory Access
SIP	Session Initiation Protocol
SOPC	System On a Programmable Chip
SQL	Structured Query Language
SReA	Subscription Registration Algorithm
SRoA	Subscription Routing Algorithm
StAX	Streaming API for XML
TCP	Transport Control Protocol
TSE	Triple-Speed Ethernet
UCS	Universal Character Set
UDP	User Datagram Protocol
UTF	Unicode (or UCS) Transformation Format
VRT	Vicinity Routing Table
VTD	Virtual Token Descriptor

W3C	World Wide Web Consortium
WAN	Wide-Area Network
WFMS	Work Flow Management System
XAVI	XML/XPath AValon Interface
XML	eXtensible Markup Language
XPath	XML Path Language
XQuery	XML Query Language

Chapter 1

Introduction

1.1 Motivations

The sources and types of information have tremendously increased and become so diverse that new technological ways of communications are being developed. The communication technology tries to find solutions to emerging issues such as how efficiently a system can disseminate diversified data originating from many different sources and through different types of media. Future Internet needs insight envision to cope with such manifold types of information. Many of the insights for the future consider the Internet as an information-centric architecture in terms of routing and disseminating data. The publish/subscribe paradigm is an interesting promising approach in disseminating various data for such information-centric futuristic networks. For example, the Publish-Subscribe Internet Routing Paradigm (PSIRP) project is being developed by multiple European organizations aiming to "develop, implement, and validate an information-centric internetworking architecture based on the publish-subscribe paradigm, which appears to be one of the most promising approaches to solving many of the biggest challenges of the

current Internet" [2].

The publish/subscribe approach of data dissemination has been used to tackle scenarios of applications that are particularly event-based and are not easy to be handled without such systems. However, this approach has extended its presence to a broad range of applications including news, stock quotes, mobile and sensor data, alerting services, grid and cloud data, business activity monitoring, healthcare services, social media communications, and distributed system management. One of the main advantages of publish/subscribe systems is providing event processing in such applications in a total decoupling way. In other words, the end-users of such a system do not communicate directly. Instead, the system itself processes asynchronous events that are triggered as a result of matching content information originated by producers (publishers) against interests and requests originated by consumers (subscribers).

The total decoupling requirement for publish/subscribe systems draws interesting features. For example, subscribers are notified with requested information, if matching events occur, while they remain unknown from publishers (space decoupling feature). Moreover, both publishers and subscribers can submit their publications and subscriptions any time independently, and they are not required to be up online at the time of matching events (time decoupling feature). Therefore, coordination and synchronization between entities are not required, which increases the scalability of distributed publish/subscribe systems.

In general, the scalability feature is often questionable in either centralized or distributed publish/subscribe systems. Despite their great importance, the aforementioned decoupling requirements contribute to the scalability challenge. However, these requirements are not the sole reasons behind the problem. There has been an increasing demand for a *flexible* (more *expressive*) architecture of publish/subscribe systems. This

demand has led to the so-called content-based publish/subscribe systems, which offer a high degree of flexibility. In such systems, the subscriptions are more expressive; and their expressions (e.g. attribute values) have to match some content in available publications, while notifications of matched content are to be delivered - usually based on content - to corresponding subscribers. Moreover, since XML is the *de facto* system message communication, content-based publish/subscribe systems tend to utilize XML for their standard communication medium. With the additional features of flexibility (expressiveness), content-based routing, and the utilization of XML, content-based publish/subscribe systems suffer from different processing and routing bottlenecks - which affect their performance and scalability capabilities.

Content-based Networking is considered as a generalization of a distributed publish/ subscribe notification service [29]. In a Distributed Publish/Subscribe System (DPSS), a network of brokers cooperate to route the published data to interested subscribers based on the data content [129]. A DPSS architecture is, then, an infrastructure built to provide scalable content-based services to a large number of users. The routing in such an infrastructure is often referred to Content-based Routing (CBR) [29, 28]. This communication style is reminiscent of IP multicast. In particular, the receiver who subscribes to receive a specific content is analogous to the one who joins a specific multicast group in IP multicast. Also, the sender who publishes specific information is analogous to the one who sends information to a specific multicast group. The main difference resides in the addressing scheme. In publish/subscribe systems, the address scheme is content-based rather than address-based. Therefore, the communication is performed in structured messages with a known syntax, using a predicate language based on the structure of these messages. In XML-based systems, for example, XML messages can be used with selection predicates based on XML Path (XPath) expressions [25]. Therefore, users may

select a specific XML schema to send information to interested receivers who formulated XPath expressions based on the same schema. In some other systems, messages can be structured as tuples of named attributes, while receivers use Standard Query Language (SQL)-like expressions of same attribute names in their predicates. It is also possible to use the XML Query Language (XQuery) [121], which is intended to be for XML data as SQL is for databases. XQuery is an extension to XPath derived from Quilt [32].

1.2 Problems and Requirements for Solutions

The main problem of existing content-based publish/subscribe systems mainly resides in their low performance, since such systems are usually complex and need intensive processing. Moreover, even though the use of XML makes these systems interoperable, processing of XML adds more challenges toward the achievement of efficient parsing, matching, and routing mechanisms. Thus, in a content-based pub/sub system that supports both XML publications and XPath subscriptions, each broker in the system may face processing bottlenecks, as depicts Figure 1.1.

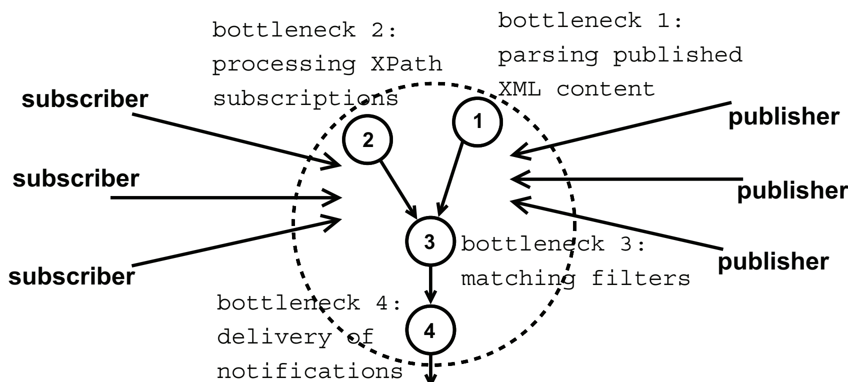

Figure 1.1: Possible Processing Bottlenecks in a Broker of an XML/XPath Pub/Sub System.

For an XML content-based publish/subscribe system architecture to operate in high-

performance, there are issues that need efficient solutions. Therefore, the development of the architecture has addressed the following requirements:

- When XML is involved, efficient XML parsing is needed. Thus, an architecture needs to address the XML parsing problem.
- Serious complications and performance issues may result from the need of (1) simultaneous processing of many subscription queries, (2) matching all these queries against XML publishers' content, and (3) delivering notifications to subscribers. Therefore, these challenges need to be appropriately transcended.
- The content-based routing feature adds complexity to the system. Therefore, an architecture has to include a mechanism that provides the content-based routing capability in an efficient manner.
- Finally, a hardware-based solution needs to be tested in terms of functionality and performance.

The proposed system has efficiently tackled these requirements throughout the development phases, and it accordingly provides a high-performance architecture. The extension of the architecture with multimedia represents a solution to the continuous strive to put more content on fixed and mobile networks, where such content has to be parsed, routed and displayed in real-time. Section 2.5 includes a comparison between the proposed system and some of the existing systems whose features may partially fall close to the proposed one.

1.3 Thesis Contributions

The main thesis contributions span four areas as follows:

- **XML Parsing:** Design and implementation of a novel hardware-based XML parsing technique (named SCBXP). This technique can accomplish high-performance parsing of XML with a rate of two bytes per clock cycle.

- **Broker:** Design and implementation of a novel efficient hardware-based XML/XPath broker that employs the aforementioned XML parsing technique, and includes a subscription queries processor and a matching engine. A network of identical hardware brokers represent a reconfigurable content-based publish/subscribe data dissemination system. To the best of our knowledge, such approach would represent the first publish/subscribe system that is based on hardware brokers.

- **Subscription Registration:** Design of mechanisms that allow for registration of subscriptions in a systematic method that efficiently exploits subscription commonalities for high-performance publish/subscribe matching.

- **Content-Based Routing:** Design and utilization of effective content-based routing algorithms that allow for efficient routing of messages based on content.

Parts of these contributions have been published in refereed journals and conferences. With regard of XML parsing, the relevant hardware technique has been introduced in [44] and subsequently detailed in [46]. The proposed broker architecture has been introduced in [45] and has been comprehensively detailed and expanded with relevant mechanisms in this thesis.

The proposed broker takes the name of X2CBBR, which stands for XML/XPath Content-Based Broker/Router.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 presents a state-of-the-art comprehensive literature review of existing publish/subscribe systems. Chapter 3 addresses the high-level design of the proposed X2CBBR. Chapter 4 describes the hardware-based XML parsing technique. The detailed architecture of the X2CBBR represents the subject of Chapter 5. The architecture description includes XML publication and XPath subscription hardware processing and matching, as well as the algorithms employed for various storing and routing tasks. The results obtained from the comprehensive testing of the parser are depicted in Chapter 6. In Chapter 7, the results obtained from testing the main components of the X2CBBR are illustrated. Finally, Chapter 8 states the thesis final conclusion, and discusses the future work.

Chapter 2

Publish/Subscribe Systems Review

2.1 Definition

The simplest form of a *Publish/Subscribe* (pub/sub) system consists of one publisher, one subscriber, and one broker. The subscriber communicates with the broker to show its interest in receiving some content of published information, whereas the publisher communicates with the broker to show interest in publishing content information that might be of interest to any subscriber. The publisher does not know who would have subscribed to receive the published information, and thus has no idea of who would receive such information. This feature is often referred to as the full decoupling characteristic of a pub/sub system in time, space, and synchronization [49]. The broker (or dispatcher) performs the role of (1) performing message-oriented communication [25] with either the publisher or the subscriber, (2) matching the interests of a subscriber against the publisher content, and (3) notifying the subscriber of any matched content that may have been found. A pub/sub system can become more complex when many publishers and subscribers are involved. An overview of such a system is shown in Figure 2.1.

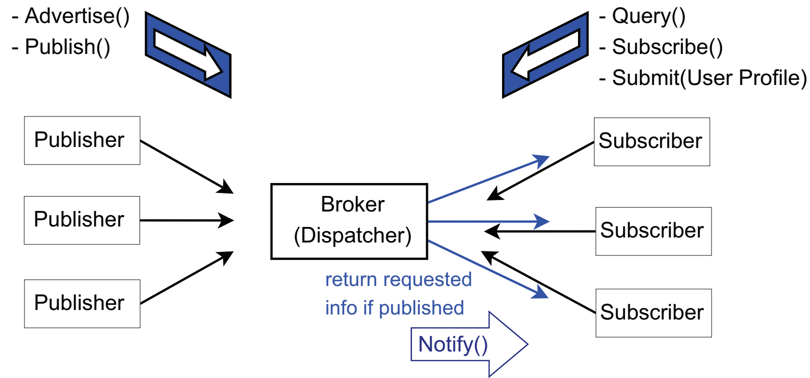

Figure 2.1: Overview of a Publish/Subscribe System.

In a centralized pub/sub system, there exists only a single broker that performs all the communication and processing tasks of all publishers and subscribers (Figure 2.2).

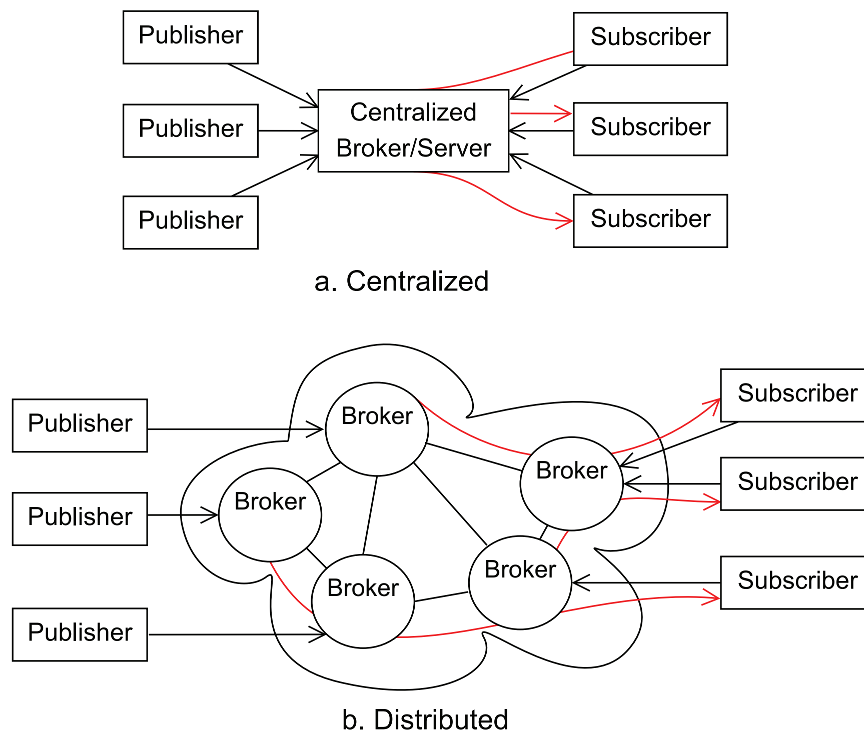

Figure 2.2: Centralized versus Distributed Publish/Subscribe System.

The downside of such a centralized system is that the processing burdens of the broker become overwhelming when the number of publishers and subscribers becomes so high.

Alternatively, research efforts suggested a distributed pub/sub system, where the tasks of communicating, matching, and notifying can be distributed among several brokers instead of just a single one. In such systems, a network of brokers cooperate to route the published data to interested subscribers based on the data content [129]. Accordingly, distributed pub/sub systems are considered more scalable and able to handle many publishers and subscribers that can be located anywhere in the world. Figure 2.2 shows a general view of the difference between a distributed pub/sub system and a centralized one.

2.2 Types of Publish/Subscribe Systems

The matching task of a broker in a pub/sub system can be based on different criteria. Some criteria are simple, while others are too complex. Reviewing the different matching criteria that may exist in pub/sub systems leads to the following pub/sub types: Channel-based, Topic-based, Type-based, and Content-based. The next sections provide descriptions for each pub/sub type.

2.2.1 Channel-Based

Producers publish events to channels, while consumers subscribe to a channel to get all the publications that exist in the selected channel [78]. The Common Object Request Broker Architecture (CORBA) [89] is often mentioned as an example of such a system. CORBA was developed by an industry consortium, known as the Object Management Group (OMG), as an architecture that enables objects to communicate even if they are written in different programming languages and run on different operating systems. This loose coupling feature allows an object that can be a publisher to communicate through

a channel with another remote object that can be a consumer. The object model of CORBA has been evolved into the so-called "component model," where components that represent "event sources" (publishers) communicate through "event channels" with components that represent "event sinks" (consumers) [87].

2.2.2 Topic-Based

Producers publish events and consumers subscribe to individual topics (or subjects), which can be identified by keywords [49]. The notion of topics extends the notion of channels with methods of event content classifications. In software-based practice, a mapping mechanism links individual topics to distinct communication channels [49].

A pub/sub system may allow for the publication of (and subscription to) one topic only. Some systems allow for the usage of wildcards or several topics whose names match a given set of keywords, such as the TIBCO Rendezvous (TIB/RV) [116]. This sort of hierarchy improves the topic-based representation, so that a URL-like tree hierarchy may represent a topic [49] [78]. Since this representation can be of XML format, TIBCO participates in some standards organizations such as W3C [119] and RosettaNet [3] in order to actively support XML.

Even though the topic-based approach is relatively simple, it suffers from limited expressiveness [49] as well as from inflexibility if slight order permutations are needed [78].

2.2.3 Type-Based

The type-based approach of pub/sub systems does not include only topics but also the notion of topic inheritance [100]. While the static nature of topics does not allow for

several supertopics, types provide the possibility of subtypes that can include supertopics [48]. For example, in a topic-based system, the topic London/Technology is different from the topic Technology/London [78]; thus, duplicated publications are needed in order for a subscription of either filtering order to be successfully matched. In a type-based system, just one publication of either order is sufficient to provide a successful matching decision for a subscription of either filtering order. This type-based concept can lead to a sort of content-based filtering [49].

2.2.4 Content-Based

In content-based pub/sub systems, events (i.e. matched publications or notifications) are not triggered based on some predefined keyword, such as a topic name, but based on matched contents that represent properties (or attributes) of the events. Examples of such systems are Siena [26][27], Gryphon [21], and JEDI [40]. Subscribers specify their interest in the form of constraints named "filters" using a custom or standard subscription language. Each filter can be a combination of one or more of name-operator-value statements (e.g. price=12 and stock<10).

The content-based pub/sub system may cover all previous approaches. The content can be just a channel, a specific topic, a specific type, or any content including complex constraints. However, a user subscribing to a topic name, instead of filter constraints, may receive irrelevant events - which results in an inefficient use of bandwidth [49]. But if the system splits a topic into subtopics, the subscriber may receive redundant events [49]. Thus, it is better to make use of the expressiveness capability of a content-based pub/sub system, so that the more expressive is the content the more precise is in receiving notifications - thereby the more efficient in utilizing bandwidth. However, the high expressiveness imposes more processing burdens, thereby adding more challenges in

achieving an efficient pub/sub system.

In order to have a standard structure of the content, the XML is an evident choice to be the content type used for content-based pub/sub systems providing the XML popularity in modern communication systems. In this thesis proposal, the proposed content-based pub/sub system utilizes XML [119] as the publication content, while the subscription language is XPath (XML Path Language) [120].

2.3 Existing Publish/Subscribe Systems

Existing content-based pub/sub systems differ in many features. For example, the type of the pub/sub system is one fundamental feature. Other main features include whether such systems are centralized or distributed, broker-based or based on Distributed Hash Tables (DHTs), whether they support XML or not, whether they are implemented over a Peer-to-Peer (P2P) or an overlay network, and whether they are filter-based or multicast-based.

The existing pub/sub systems would fall into four main categories: (1) Systems that do not directly support XML and do not usually operate over P2P networks, (2) Systems that do not directly support XML but usually operate over P2P networks, (3) Systems directly support XML especially that XML filtering and/or routing are needed, and (4) Other systems that can be either centralized or distributed.

Note that brokers may not differ from peers in a P2P network, since brokers behave as cooperating peers among themselves [74]. When no brokers exist in a pure P2P system, peers can be considered both clients and routers, thereby functioning as brokers [74]. However, this literature review simply considers a system with overlay brokers as the kind of systems that is not based on a P2P infrastructure and that does not use DHTs.

It is useful to mention that techniques used in non-XML-based pub/sub systems can conceptually apply to XML-based systems as well. However, supporting XML usually comes at considerable processing costs. Accordingly, considering the support of XML a factor in categorizing pub/sub systems seems acceptable.

2.3.1 Non-XML-based Systems over non-P2P Networks

Siena [27] is a content-based event-notification service, where event servers act as content-based routers. In Siena, publishers connect to a server in order to publish events, and subscribers connect to the server in order to subscribe and receive the events that they are interested in. Matching is performed in each server. In an originating server (router), a new subscription can be stored and forwarded to a subset of server's adjacent event servers in the network. Therefore, a subscription parades through a tree of servers. Then, the reverse path of the tree can be used to deliver notifications to the corresponding subscribers.

In a software-based centralized system, Siena performs poorer than in a software scenario modeling a network of content-based routers [30]. In order to improve routing, the authors of Siena later introduce two other cooperating routing protocols based on two combined broadcast and CBR (Content-Based Routing) layers [28]. Their intent was to perform more selective routing decisions and to avoid imperfect routing.

In Gryphon [21][112], a link-matching algorithm is utilized as a multicast technique to partially match an event in each broker of the network. This technique builds a Parallel Search Tree (PST) that consists of the filters of each subscription corresponding to a path from the root to a leaf. The algorithm exploits commonalities among subscriptions because of the existence of shared prefixes in the paths from root to leaf. This approach eases the burden of processing at each broker but increases network traffic, since it needs

to replicate all the subscriptions at all the brokers in the system.

Rebeca [77] provides a notification service, relying on a network of brokers and using filter-based routing tables. The whole broker network is considered a single acyclic graph used to disseminate data, resembling to the single spanning tree used for multicast, while the edges of the broker network form a point-to-point overlay network communicating with TCP/IP connections. Each broker maintains a filter-based routing table and supports various CBR algorithms. Rebeca supports partitioning of the broker network into separate “scopes” of overlays, so that notifications are only visible within the scope that bundles a subset of publishers and subscribers.

PADRES (Publish/subscribe Applied to Distributed REsource Scheduling) [66] makes use of composite subscriptions that consist of several traditional atomic subscriptions linked by logical or temporal operators. A composite subscription can be matched by a set of independent events that may have occurred at different locations and times. PADRES maps atomic and composite subscriptions to rules, while it inserts a publication as a fact. A rule-based broker matches the rules against the facts, where a fact may match part of a rule or several rules. PADRES builds a destination tree for each composite subscription, where each node in the composite subscription has a corresponding node in the destination tree. PADRES performs message routing through an overlay network of brokers.

PADRES is further enhanced using an adaptive content-based routing mechanism, which performs alternate publication routing paths and improves the resilience to dynamic loads [67]. This mechanism allows advertisements to be broadcast through the network to form advertisement trees. Then, subscriptions that match advertisements propagate in reverse direction along the advertisement trees, and set up one or multiple alternate routing paths for publications. Finally, publications matching subscriptions are

routed toward subscribers, without further matching, through one of the alternate paths consisting links of minimal cost.

Kyra [23] is a routing scheme aims at reducing the content-based filtering costs by partitioning the event space into small routing networks (scopes, zones, or clusters). This scheme tends to move and relocate subscriptions in different servers, which leads to the balancing of routing and storing load.

MEDYM [24] (Match-Early with DYnamic Multicast) decouples the matching and routing processes, where complex application-specific matching only occurs at the network edge, and simple generic address-based routing occurs in the network core. The matching/routing decoupling is an advantage because matching with subscriptions is needed only once, while the notification delivery is done through simple address-based routing. However, if the notification has to be delivered to entities relatively far from the source, a long list of destination addresses should accompany the notification - a matter that can adversely affect scalability. To improve scalability, the authors propose an extension to MEDYM called H-MEDYM (Hierarchical MEDYM), in which severs are clustered based on location and the content space in further partitioned into topics.

Herald [22] makes use of Rendezvous (RV) points to which subscriptions and publications are routed. The routing is performed based on content through a federation of servers. The determination of the RV server is decided by users instead of the Herald system itself. However, Herald can balance the load and move some of the traffic from one RV point to another, using replication, partitioning, and resilience to failure methods. The event notification is delivered using either unicast or multicast communications.

2.3.2 Non-XML-based Systems over P2P Networks

There are some pub/sub systems that are built on top of P2P overlay networks (e.g. Pastry [107], Tapestry [128], Chord [111], and CAN (Content Addressable Network) [104]) that facilitate Application-Level Multicast (ALM), where topics or rich contents are mapped to multicast groups.

Scribe [31], is a topic-based pub/sub system that provides an application-level multicast (ALM) infrastructure and works in P2P environments. It is built on top of Pastry [107], which is a self-organizing P2P overlay system. Pastry utilizes node ID values as host addresses, by hashing the node's public key or IP address. Pastry routes a message associated with a key to the node whose node ID value is numerically closest to the message key. In Scribe, each multicast group has a unique group ID, and the RV point for a group is determined according to the Pastry node ID value that is numerically closest to Scribe's multicast group ID. Therefore, the RV point becomes the root of the multicast tree for the multicast group.

PastryStrings [7] is a content-based pub/sub system that supports rich alphabet and operates over Pastry. PastryStrings defines string trees, where each tree is dedicated to one of the characters of the alphabet. Inside the tree, subscriptions are stored in RV nodes. PastryStrings makes use of two separate routing tables that are both based on prefix-based routing: one table is maintained by a Pastry node and used to locate the string trees along with corresponding tree paths, and the other is maintained by a string tree node to forward subscriptions towards RV nodes. All string tree nodes belonging to the located tree path may store subscriptions that match a prefix of an event value. The construction of the string tree node routing table is done based on the routing table of the Pastry node hosting the string tree node. Note that each Pastry node may host one

or more string tree nodes.

Bayeux [132] is a topic-based pub/sub system built on another P2P environment called Tapestry [128]. Bayeux takes advantage of the unicast routing nature of Tapestry in order to build an ALM system. Each multicast group consists of a tree rooted at a node that represents the destination for the tree node leaves. Once the root node advertises a new multicast session, clients can join the session if they know the session ID. A Bayeux multicast session is identified with a hash value of the semantic session name, describing the content of the multicast, in addition to a unique ID.

Ferry [130][131] supports content-based pub/sub services based on DHTs of a P2P environment such as Chord [111]. Each Chord node possesses a 160-bit identifier and maintains a routing table and a list of node successors. In Ferry, each node may take on the role of a Rendezvous Point (RP) for some subscriptions and events, and on the role of an intermediate node for event delivery. Subscriptions can be grouped along DHT links so that matching events can be aggregated in order to minimize the system traffic. However, Ferry does not construct multicast trees. Since the number of the RP nodes is proportional to the number of attributes, a large number of RP nodes may be needed. But the number of RP nodes cannot be practically too high; thus, the available RP nodes likely become overloaded - which leads to a scalability issue. Attempts to enhance Ferry's scalability result in the appearance of Eferry [123]. This enhanced version of Ferry provides mechanisms to adjust the amount of RP nodes and to evenly balance the load among those nodes.

Multicast over CAN [104] has been engineered in [105] and [51]. Each CAN node maintains a routing table whose entries consist of IP addresses and "virtual coordinate zones" for node's neighbors. The virtual coordinate zone is defined by coordinates that allow the determination of a zone adjoining the node's zone. In [105], the ALM tree

is not built for a given multicast group. Otherwise, if all the nodes in the CAN are members of a multicast group, then multicasting a message leads to flooding over the entire CAN. Instead, a subset of CAN nodes establish a separate “mini” CAN in which flooding can achieve multicasting. Meghdoot [51] is a content-based pub/sub system that is built over a DHT based on CAN. Meghdoot is able to store subscriptions and deliver matching events based on the semantics of the subscriptions and the events. Meghdoot maintains decoupling between publishers and subscribers, and adapts to highly skewed data sets using either zone splitting or zone replication.

Hermes [101][100] is a type- and content-based pub/sub system employing event brokers that form an overlay routing network. Hermes deploys an event dissemination tree, where the overlay routing operation allows brokers to find RV nodes for advertisements and subscriptions. The RV nodes consist of a subset of event brokers classified according to the hash values of events’ type. To achieve its functionality, Hermes makes use of three layers over IP: a topic-based layer, a type- and content-based layer, and an event-based layer.

Corona (Cornell Online News Aggregator) [103] is a topic-based pub/sub system for the Web, where users subscribe to Web pages using existing Instant Messaging (IM) services. The goal is to monitor and detect the updates that may occur in the subscribed Web pages, in order to asynchronously notify clients with the updates through IM. There are no required changes to the existing Web servers. The update detection is done through polling performed by cooperative Pastry peers that share the updates.

LightPS (Lightweight Content-based Publish/Subscribe) [8] is a lightweight solution for a DPSS operating over DHTs. LightPS uses the DHT routing properties to set RV nodes in the system, while avoiding the construction of a specific overlay for multicast. This technique converts multi-dimensional values of subscriptions and events into one

or more keys to define the RV nodes. In addition, the notification service is built on a DHT-generic algorithm in order to operate over different DHTs.

PEM (Popularity-based Event Matching) [127] is a pub/sub system built over a DHT overlay of a structured P2P system. PEM performs CBR using hash values of attribute names that are embedded in subscriptions and events, where both subscriptions and events are routed to multiple RP nodes. PEM distinguishes between popular and unpopular attributes, and considers the least popular attributes of the event scheme.

Other architectures of Peer-to-Peer (P2P) overlays that can be used by content-based pub/sub systems can be found such as Mirinae [39] and Kademlia [75]. For more detailed information on P2P systems, the reader may refer to the survey of Lua et al. [70].

2.3.3 XML-based Filtering and Routing Systems

The performance of pub/sub systems can be greatly enhanced when efficient XML filtering designs are enclosed. The next paragraphs state examples of existing XML filtering systems that are (or can be) used in pub/sub systems.

The ONYX (Operator Network using YFilter for XML dissemination) [42] architecture consists of an overlay network of brokers handling published data and user queries. ONYX makes use of YFilter [41], to perform XML filtering, as well as other tasks employing two layers: a control layer and a content-based processing layer. In the control layer (lower layer), an application-level broadcast tree is constructed for each broker starting from it and reaching all other brokers in the network. There are two methods in the control layer that help the content-based processing layer, one for forwarding messages down the tree, the other for reverse forwarding (i.e. upstream) of queries. In the content-based processing layer (higher layer), which consists of a data plane (for messages flow) and a query plane (for queries flow), are three main tasks that perform: content-driven

routing, incremental transformation, and user query processing.

When a subscriber requests content information, its broker (called host broker) stores the query and informs its upstream broker (parent broker) about its interest. The parent broker adds the child broker's interest to its routing table, and informs its upstream broker about its interest, and so on. The flow of forwarding queries upstream is considered the query plane, where routing tables are constructed or updated. The root broker, having publisher information and its downstream children-brokers' interests as well, matches publisher's content with its routing table entries and accordingly forwards the content information to interested downstream brokers. This content-driven routing continues to flow through interested children brokers down to the host broker. The flow of content data downstream occurs in the data plane. This routing task is called "content-driven" (or perfect routing) because when a broker forwards content data downstream through its broadcast tree, only interested children brokers (not all children-brokers in the tree) receive the content information.

XNET [35][36] provides XML filtering using decomposed XPath expressions and a trie-based index data structure called XTrie [33]. In addition, it employs XSearch algorithm to efficiently manage a large number of subscriptions. XNET's routing protocol (Xroute) performs subscription aggregation and routing, so that the size of routing tables stays manageable. In each XNET node, a routing table contains the neighbor nodes to which subscriptions should be routed. This table can be updated as a result of using a mechanism for subscription advertisement. XNET also implements fault-tolerance approaches.

In [81], rules of grammar structure are first converted into a Lex/Yacc style syntax. Then, using a custom compiler, the authors convert the Lex/Yacc style grammar to the VHDL hardware description language. The VHDL code is eventually mapped to a

pipelined logic structure. Afterwards, the grammar-mapped logic is compared to XML patterns using a pattern matcher, in order to make routing decisions. This content router delivers a maximum throughput of 3.2 Gbps when routing is based on the first character of one predetermined word. Therefore, the routing mechanism in this case can be considered as "character-based" routing. However, when updates to the grammar for XML routing are frequently required, the need for stages of software conversion for each new update would significantly drop the overall performance.

NaradaBrokering [92] is a content-based pub/sub distributed brokering system intended for web/grid services and P2P applications. NaradaBrokering utilizes a distributed XML matching engine that evaluates (1) real-time XML events against the stored XPath subscriptions and (2) stored XML advertisements against a real-time XPath query for discovery purposes. In NaradaBrokering, the broker network consists of hierarchical clusters. The links to brokers in other clusters can be alternate routes in the case of failures.

In [110], XML routing can be performed using mesh-based (in contrast to tree-based) overlay networks, where application-level XML routers perform routing of individual XML packets based on queries that describe the needs of downstream nodes. The XML content is evaluated against queries expressed in the XML Query Language (XQuery) [121]. The overlay management employs a set of algorithms to organize routers and clients into a mesh. The communication in the mesh among routers or between routers and clients adheres to a protocol called Diversity Control Protocol (DCP). In DCP, a receiver can reassemble a stream of redundant packets from diverse senders using the first copy of a packet from any sender. This XML router can achieve a throughput of 18 Mbps.

2.3.4 Other Systems

While the previous sections have mainly described distributed pub/sub systems, this section discusses centralized pub/sub systems as well as systems that are distributed but they are characterized with special features.

The Java Message Service (JMS) defines a common enterprise messaging API (Application Programming Interface) that can be supported by a wide range of enterprise messaging products [72]. JMS supports two types of messaging models: point-to-point (queuing) and pub/sub. In its pub/sub model, JMS delivers publishers' messages to topic destinations, then to the topic active subscribers. There is an option in JMS that allows a subscription to be "durable" so that inactive (offline) subscribers retrieve missed messages after reconnection.

Elvin [109] proposes a technique called "quenching" that allows publishers to get knowledge of the interests contained in the subscriptions, so that they purposely publish events that match subscriptions. This mechanism reduces traffic, since content that is not in interest would not be published. However, the system needs to report the updates in the subscription database to publishers before the "quenching" of publications occurs. Therefore, frequent updates lead to frequent reporting in order to avoid unnecessary publications.

JEDI (Java Event-based Distributed Infrastructure) [40] is used to implement an application called OPSS (ORCHESTRA Process Support System) as part of a WFMS (Work Flow Management System). ORCHESTRA stands for Open aRCHitecture for supporting Enhanced Services in inTegRAted broadband networks. JEDI has two versions: centralized and distributed. In its distributed version, JEDI exploits a hierarchical subscription strategy, where subscriptions are forwarded only upwards in the hierarchical

tree. Consequently, events may be dispatched downwards for matching subscriptions but also upwards even if it is not necessary. JEDI supports mobility through operations that allow mobile clients to disconnect from and reconnect to an event dispatcher (broker).

HYPER [126] is a hybrid pub/sub scheme that takes advantage of the features of both topic-based and content-based systems. The designers of HYPER noticed that, in many content-based pub/sub systems, expensive matching processes are usually needed in each node of the delivery tree, and that content may have to be routed to many nodes in the network, which incurs considerable delays. In HYPER, the aim is to minimize both the amount of needed matching occurrences and the amount of needed delay for delivery. Therefore, HYPER organizes subscriptions into virtual groups based on common subscriptions, where the matching process occurs once at the entry point of each group. Inside the group, messages are simply forwarded with no more matching occurrences. The matching is based on expressive content, as in content-based pub/sub systems, but the forwarding within each virtual group is similar to the delivery of content to a multicast group in topic-based pub/sub systems. In other words, HYPER implements two overlays: one consists of entry points of virtual groups and behaves as a broadcast layer, and the other overlay consists of multicast channels among the members of each virtual group.

LeSubscribe [96] is a content-based pub/sub system that provides an event notification service for the Web. LeSubscribe avoids the storage of event histories. Instead, it uses processes events (e.g. events published through web browsers and auction sites) on-the-fly to discover matching subscriptions. The event model is similar to the Lightweight Directory Access Protocol (LDAP) model. The web publications are parsed and converted into publishing events that can be sent to LeSubscribe via Java RMI (Remote Method Invocation), while subscriptions are made via HTML (HyperText Markup Lan-

guage) requests. LeSubscribe has been extended into WebFilter [95] that uses XML publishing events and XPath subscription queries.

The industry consortium OMG, which was behind the development of CORBA, has set an interoperability protocol called Data Distribution Service (DDS) that is intended for data-centric pub/sub systems [90]. This protocol promotes a "globally-accessible shared data-space" that can be accessed by system monitors and intrusion detectors, so that prompt response actions can be issued. To support the unique requirements of data-distributions systems, OMG recently developed the Real-Time Publish Subscribe (RTPS) protocol that can run over multicast and connectionless best-effort transports such as UDP/IP [88].

Naming content with name resolution has been proposed for Information-Centric Networking (ICN). The name resolution aims at matching a content name to the corresponding producer, while data is routed based on content to the corresponding consumers. However, this content routing with name resolution may not necessarily imply decoupling between producers and consumers. Moreover, the naming space can be huge, which eventually limits scalability. Examples of such ICN proposals can be found in [59] and [56].

2.3.5 Summary of Existing Systems

For the sake of clarity, Table 2.1 summarizes the different features of pub/sub systems. This table abbreviates terms as follows:

- **C/D**: Centralized or Distributed pub/sub system.
- **P/S Type**: Type of Pub/Sub system that can be content-based or topic-based.
- **B/S**: System can be formed with Brokers that may act as Servers.

- **D/H**: System makes use of DHTs or any Hash conversion.
- **X/X**: System can support either XML, XPath, or both. If there might be some kind of such support, with no very clear indication of relevant achievement, the term "pbl" indicates possible XML support.
- **S/W**: System is implemented in either Software or Hardware.
- **Im. F.**: Some identifiable Implementation Features in the system.
- **CBR**: In "Other Comments" column, the CBR term indicates that the pub/sub system utilizes some sort of a Content-Based Routing algorithm.

2.4 Issues and Drawbacks of Existing Systems

The aforementioned literature review points to many different features of existing pub/sub systems. However, all of these systems have issues, and some of them suffer from drawbacks. The following sections identify and list the main issues or drawbacks.

2.4.1 Performance

The design of any system has the goal of providing performance. In general, distributed systems are designed to enhance the performance of centralized systems. However, since matching and routing processes have to be performed in each broker of the network, and providing that these processes are mainly software-based, the system performance remains somewhat limited. In the case that a pub/sub system uses XML, the level of performance that can be reached is even more challenging.

Table 2.1: Characteristics of Existing Pub/Sub Systems

Pub/Sub Systems	C/ D	P/S Type	B/ S	D/ H	X/ X	S/ H	Im. F.	Other Comments
Siena[27]	D	Content	yes	no	no	S	-	CBR
Rebeca[77]	D	Content	yes	no	pbl	S	Scopes	CBR
Kyra [23]	D	Content	yes	pbl	no	S	Scopes	CBR Load-Balanc.
Medym[24]	D	Content	yes	pbl	no	S	Clusters	Multicast
PADRES[66]	D	Content	yes	no	pbl	S	Monitoring	CBR
Gryphon[21]	D	Content	yes	no	no	S	Link Match. Multicast	-
Herald[22]	D	Content	yes	no	no	S	Fault-toler.	CBR
Scribe	D	Topic	no	yes	no	S	P2P	ALM based on Pastry
Bayeux	D	Topic	no	yes	no	S	P2P	ALM based on Tapestry
Meghdoot	D	Content	no	yes	no	S	P2P	Based on CAN
Hermes	D	Type/Content	no	yes	yes	S	P2P	Overlay
Ferry	D	Content	no	yes	pbl	S	P2P	Based on Chord
Eferry	D	Content	no	yes	pbl	S	P2P	Based on Chord
PEM	D	Content	no	yes	no	S	P2P	Based on Chord
PastryStrings	D	Content	no	yes	no	S	P2P	On top of Pastry
LightPS	D	Content	no	yes	no	S	P2P	-
Corona	D	Topic	no	yes	yes	S	Subs.use IM	On top of Pastry
ONYX[42]	D	Content	yes	no	yes	S	Filtering	CBR
XNET[36]	D	Content	yes	no	yes	S	Filtering	CBR
Narada	D	Content	yes	no	yes	S	Clusters P2P	Web/Grid
WebFilter	D	Content	yes	no	yes	S	-	Web
TIB/RV	C/D	Topic	yes	no	yes	S	-	-
CORBA	C	Channel	yes	no	no	S	-	-
JMS	C	mainly Topic	yes	no	pbl	S	Durable Subs.	-
Elvin	C	Content	yes	no	pbl	S	Quenching	-
JEDI	C/D	Topic/Content	yes	no	no	S	Hierarchy	Mobility
HYPER	D	Content	yes	no	no	S	Virt. Groups	Multicast
LeSubscribe	D	Content	yes	no	no	S	-	Web

2.4.2 Interoperability

Many of existing pub/sub systems do not support XML, which makes such systems not capable to interoperate. Thus, for the sake of interoperability between different systems,

XML has the power of providing a practical interoperable communication standard.

2.4.3 Scalability

Even though distributed systems are more scalable than centralized systems, the scalability is often an issue when the number of publishers and subscribers greatly grow. This issue is even more important providing that matching is not separately performed away from routing.

2.4.4 Other Issues

When a failure occurs, the system should quickly report such an event and proceed to an alternative operation or to a recovery mode. This feature is often referred to the "survivability" term. As in most systems, security measures are important to prevent any abuse. In the event of any security breach, an alert has to be triggered and quick procedures should be undertaken.

2.5 Proposed System versus Existing Systems

The proposed hardware X2CBBR adopts the pub/sub approach with total decoupling between publishers and subscribers, while routing is performed based on content identification in an identified broker network. Unlike some ICN proposals, X2CBBR does not employ naming for content and name resolution.

X2CBBR can work in centralized pub/sub systems (where a single broker resides) as well as in distributed pub/sub systems (where a network of brokers interoperate). Moreover, the proposed architecture provides a content-based type of functionality, supports XML as publishers' content, and admits XPath as subscribers' queries. Moreover,

a fundamental advantage of the proposed system resides in its hardware implementation of publication processing (XML parsing), of subscription processing (XPath processing), and of the matching engine. Another advantage stems from the memory-based algorithms that allow for the simultaneous matching of many subscriptions that contain common filters against publication data. A third advantage of the proposed system resides in the separation between matching and routing, while content-based routing mechanisms are in effect. This separation is inspired by the recommendation stated in [102]. Moreover, the possibility of extending the proposed architecture with multimedia data is one more interesting advantage.

Table 2.2 compares between the pub/sub system that is based on a network of the overlay hardware X2CBBRs with "some" existing content-based XML pub/sub systems. The table restricts the comparison to the existing systems that can be the closest to the proposed system. Table 2.2 follows the same abbreviation style of Table 2.1. Moreover, in "Other Comments" column of the table, the term "Concur." indicates that the pub/sub system has some concurrent features.

Table 2.2: Comparison Between X2CBBR Pub/Sub System and Some Existing Systems

Pub/Sub Systems	**C/ D**	**P/S Type**	**B/ S**	**D/ H**	**X/ X**	**S/ H**	**Im. F.**	**Other Comments**
Rebeca	D	Content	yes	no	pbl	S	Scopes	CBR
PADRES	D	Content	yes	no	pbl	S	Monitoring	CBR
ONYX	D	Content	yes	no	yes	S	Filtering	CBR
XNET	D	Content	yes	no	yes	S	Filtering	CBR
Narada	D	Content	yes	no	yes	S	Clusters	Web/Grid/P2P
WebFilter	D	Content	yes	no	yes	S	-	Web
TIB/RV	C/D	Topic	yes	no	yes	S	-	-
X2CBBR	**C/D**	**Content**	**yes**	**no**	**yes**	**H**	**Filtering Concur. Scopes**	**CBR**

Even though the survivability and security features would be interesting add-ons to

the proposed architecture, this thesis does not consider them in its scope and therefore it does not address such issues. However, future research activities that build on this work to develop related mechanisms should fall in the right track for the proposed broker/router architecture.

2.6 XML Parsing Systems

Since this thesis presents a novel hardware-based XML parsing technique, this section provides a literature review of existing software and hardware parsing techniques. The proposed technique is called SCBXP (Skeleton-CAM-Based XML Parsing) and described in Chapter 4.

2.6.1 Existing XML Parsing Techniques

XML parsing is traditionally performed by software-based mechanisms such as the push-parsing SAX (Simple API for XML) [4], the pull-parsing StAX (Streaming API for XML) [5], and the DOM (Document Object Model) [1].

SAX and StAX represent the process of parsing XML data as a series of events. StAX saves processing time because it skips events that are not needed by an application. Therefore, the StAX technique can be considered faster than SAX. If all events in an XML document are required by an application, StAX does not provide better performance than SAX [63]. DOM, however, consists of building an in-memory tree of the XML document being parsed. Each node of the DOM tree consists of an object that stores an element's name, and corresponding pointers to element's parent, children, and siblings. DOM consumes both more memory and more processing time than SAX or StAX. However, DOM is simpler to implement.

The Virtual Token Descriptor (VTD) [122] parsing approach represents the entire XML document as (1) records of 64-bit integer arrays to indicate the positions of tokens in the document, and (2) location caches to specify the location of a parent, a child, and a sibling for the tokens. This approach outperforms DOM; and it may outperform SAX for applications that require full access to the parsed document.

The processing time reported for these software-based parsers is noted as being around 10 clock cycles per byte when parsing a simple loop, but can be degraded in some cases to 400 clock cycles per byte [60]. Moreover, the parsing cost of XML documents whose size is less than 100 KB may reach an average of 175,000 instructions per KB [85]. Improving XML processing in software is attempted by addressing special compilation techniques [60], reducing the overhead in garbage collection [117], or skipping previously-processed sequences [114]. Other techniques such as schema-specific XML parsing [38], lazy XML processing [86], double lazy XML processing [50], parallel processing using multiple cores [69] [93], and XML offload coprocessor [84] can be faster than the traditional XML parsing techniques. The double lazy parser [50] enhances the lazy parsing technique [86] by placing internal physical pointers within the XML document. In some of these software solutions [86] [93], a skeleton-like structure is employed in a pre-parsing stage, and progressive parsing follows thereafter.

In some applications, such as XML routing and XML filtering, processing XML documents in hardware is accomplished after a few stages of software conversion [81][80]. In [81], rules of grammar structure are first converted into a Lex/Yacc style syntax. Then, using a custom compiler, the authors convert the Lex/Yacc style grammar to the VHDL hardware description language. The VHDL code is eventually mapped to a pipelined logic structure. Afterwards, the grammar-mapped logic is compared to XML patterns using a pattern matcher, in order to make routing decisions. In [80], XPath queries are

first converted to PERL Compatible Regular Expressions (PCREs). The PCREs are then translated, using a custom compiler, to the VHDL hardware description language. Afterwards, the VHDL code is implemented on an FPGA. The authors of [81] and [80] do not indicate the performance of XML parsing separately. In terms of XML routing or XML filtering, these approaches show a substantial improvement over software-based XML systems. In [81], a maximum throughput of 3.2 Gbps is indicated when routing is based on the first character of one predetermined word. In [80], the authors show a throughput of 100 MB/s for their XML filtering system. However, when updates to the grammar (for XML routing) or XPath queries (for XML filtering) are frequently required, the need for stages of software conversion for each new update would significantly drop the overall performance.

There are industrial hardware accelerators [52] [65] that have appeared in the market and indeed indicate a significant speedup. Some details of these commercial products are not publicly revealed due to their proprietary nature. However, the Random Access XML (RAX) mechanism [65] accelerates XML processing in hardware after software-based steps of declaring and grouping a number of XPath statements. Then, with simultaneous processing of XPath statements, an access table is produced allowing for the use of XPaths as indices to locate and extract parsed data. However, the memory resources required are not revealed, even though an acceleration rate of at least 40% is indicated.

A preliminary study on a hardware programmable state machine shows that an average accelerating rate of one XML character per clock cycle can be achieved [71]. An example of a hardware/software XML parser that includes well-formedness checking and schema validation is found in an academic thesis [61]. In this mixed implementation, the simulation results of the hardware part reflect an achievement of 1.2 Gbps maximum throughput for a clock frequency of 151 MHz, while much of the XML schema parsing

tasks are done using software. Furthermore, the maximum reported throughput, associated with the maximum reported clock frequency, does directly translate into a parsing rate that does not exceed one byte per clock cycle.

2.6.2 SCBXP versus Existing XML Parsing Techniques

In the proposed SCBXP technique, an XML skeleton is used to configure a CAM prior to parsing XML data in hardware. The utilization of a skeleton-based CAM allows for a "tokenized" skeleton and helps in properly "aligning" XML data, as described later in Chapter 4.

The SCBXP can achieve high performance with limited memory resources, while accessing XML parsed data is simply accomplished by the external interface via a dedicated port of a dual-port memory. Note that if XPath processing is further required, this can be done independently from parsing. Thus, the SCBXP can efficiently work in parallel with a hardware-based XPath processor.

Compared to the aforementioned examples of hardware XML parsers, the SCBXP represents an efficient hardware-based XML parsing technique that provides full well-formedness checking and partial validation, where a processing rate of at least two bytes per clock cycle is achieved even if the available memory resources are limited.

2.7 XML/XPath Filtering Systems

In addition to the proposed XML parsing technique, this thesis presents an efficient XML filtering that utilizes XPath expressions and takes part of the proposed XML/XPath broker architecture described in Chapter 5. Therefore, this section provides a literature review of existing state-of-the-art XML/XPath filtering Systems.

2.7.1 Existing XML/XPath Filtering Systems

As a software-based application, XFilter [20] represents each XPath query with a Finite State Machine (FSM). Then, an indexing mechanism is used to execute FSMs simultaneously. Its drawback resides in that commonalities among different XPath expressions are not exploited. YFilter [41] combines all XPath expressions into one Non-deterministic Finite Automaton (NFA), so that common prefixes of different expressions are represented only once. To maintain path changes, YFilter uses a sort of an incremental approach at the expense of additional complexity.

In a trie-based index data structure, called XTrie [33], XPath expressions are decomposed into common substrings that contain parent-child operators. This method is improved by an approach that filters fragmented XML data [34]. The authors show optimizations as well as performance tradeoffs.

In a software/hardware application [80], XPath queries are first converted to PERL Compatible Regular Expressions (PCREs), then translated to the VHDL hardware description language using a special compiler. The VHDL code is finally implemented on an FPGA. The authors indicate improvement over software-based XML filtering systems. However, the need for stages of software conversion for new XPath queries, and for queries that need frequent updates, would significantly drop the overall performance.

In [82][83], an XPath processor is implemented onto an FPGA, where matching of queries takes place in conjunction of SAX events derived from the tags of input XML streams. In this implementation, many FPGA clusters are used as matching engines, where each cluster may host many queries. In order to reduce the fan-out, the authors limit the number of queries per cluster, and replicate both the SAX parser and the tag decoder for each cluster. This approach leads to the average throughput of 200 MB/s.

Besides the high fan-out issue and the need of intensive on-chip memory resources, the XPath architecture only focuses on comprehensive XML filtering with no exposure to external routing.

2.7.2 Proposed XML/XPath Filtering versus Existing Systems

The proposed broker architecture performs both XML parsing and XPath processing in a hardware-based environment, making careful use of memory resources that allow the broker to suitably operate in mobile networks. Moreover, the architecture performs concurrent tasks. For example, while the broker stores filters of XPath subscriptions in a Content Addressable Memory (CAM), it concurrently stores the corresponding subscription IDs in a Random Access Memory (RAM). In addition, before the broker finishes parsing of publishers' XML files, it can concurrently match stored subscription filters against already-parsed data. With regard of subscription registration, the well-organized storage of subscriptions allows the broker to simultaneously detect and efficiently exploit all commonalities that exist in stored subscriptions. Finally, with the employment of content-based routing mechanisms, the architecture smoothly integrates filtering with these mechanisms so as to operate as an efficient content-based router in publish/subscribe networks.

Chapter 3

Proposed XML Content-Based Publish/Subscribe System

This chapter presents a high-level overview of the proposed hardware solution for publish/subscribe data dissemination systems.

3.1 XML/XPath Systems

In content-based pub/sub systems, the expressiveness of subscriptions can be of a high degree so that a variety of attribute values are involved. Therefore, the processing tasks in such a system are normally so intense. With the advent of the eXtensible Markup Language (XML) [119], many systems tend to adopt XML for communication. The popularity of XML is due to its powerful features that include human-readable expressiveness, interoperability, and its extensible capabilities. Therefore, in recent years, new XML-based technologies have emerged in order to enable interoperable and extensible real-time communications. Both the interoperability and the expressiveness needed in

content-based pub/sub systems make XML a normal choice as a communication standard. Thus, publishers can use XML to advertise and publish their content. For example, a publisher may send to the pub/sub system the following XML publication:

```
<?xml version="1.0" encoding="UTF-8"?>
<stock>
    <stockitem stockid="1">
        <share val="763">
            <name>somecompany</name>
        </share>
        <qty> 60 </qty>
        <price type="USD"> 115 </price>
    </stockitem>
</stock>
```

This publication describes details of a stock item, such as the name of the company (somecompany) whose share value is 763, and the available quantity is 60 at the price of $115. The XML structure allows for the determination of relationships between XML nodes. Therefore, (share, qty, and price) are sibling element nodes and they are all children of the element node (stockitem), which is a child of the root element node (stock). Moreover, the element node (price) has the attribute node (type="USD") and the atomic attribute value "USD". Note that "atomic values" in XML are nodes that do not have any parent or children [119].

Since structured XML Path Language (XPath) expressions [120] are very useful in selecting some parts of an XML document, such expressions can often represent subscribers' interests in publishers' XML content. Possible XPath expressions matching the above XML publication are: /stock/stockitem, /stock/stockitem/share, /stock/stockitem/price, /stock/stockitem/qty, /stock/stockitem/share[@val="763"]. Accordingly, the system routes XML notifications to subscribers whose XPath expressions match some content of XML publications.

3.1.1 Basic Formal Description

One can consider a subscription S as an XPath expression that may contain one or more filters f. In the XPath subscription "/stock/stockitem/price", there are three filters $f1$ = `stock`, $f2$ = `stockitem`, and $f3$ = `price` that seek to match an XML publication. Thus, the action of subscribing to the system with this subscription can be briefly described as *sub(S,F)* where $F = \{f1,f2,f3\}$. This brief description reads that an end-user intends to subscribe to the system with a subscription S that consists of the filter set F. In general, a subscription's filter set F consisting of n filters takes the form of $F = \{f_i\}$, where i = *1,2,3,.....n*.

The action of publishing an XML publication P can be formally described as *pub(P)*. P is said to match the subscription S if P contains all filters $f_i \in F$ where F is the filter set of S, i.e. if $F \subset P$ or more precisely if $F(S) \subset P$.

As a result of successful matching, the system should send a notification to the corresponding subscriber. The action of notifying a subscriber can be described as *notify(S,N)*. This representation reads that a notification N is to be delivered to the user that had subscribed with a subscription S.

3.2 High-Level Architecture

The proposed high-level content-based XML/XPath pub/sub system represents an overlay network of brokers, where each broker comprises the proposed hardware architecture. One of the main goals of the architecture is to alleviate the possible processing bottlenecks of each broker in the system. Figure 1.1 points to such bottlenecks. The basic functionality of this distributed pub/sub system is described next.

3.2.1 Basic Functionality

Figure 3.1 shows a distributed pub/sub system that consists of publishers, subscribers, and a network of brokers. Such system can comprise the proposed reconfigurable hardware architecture in each of its brokers.

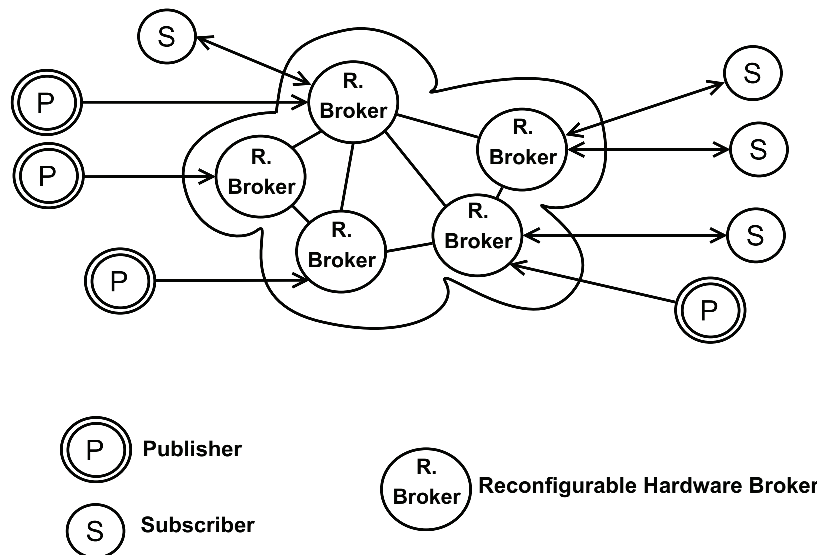

Figure 3.1: Distributed Pub/Sub System with Reconfigurable Hardware Brokers.

Subscribers can communicate with the brokers using XPath subscriptions, while publishers use XML in sending their publications. Inside each broker, all processing tasks of XML and XPath take place. These tasks include storing, parsing, matching, and routing. A broker matches available subscriptions against available publications, and it accordingly delivers (routes) notifications to corresponding subscribers. Each broker can communicate with neighboring brokers for different purposes including forwarding (routing) XPath subscriptions and XML publications/notifications. Figure 3.2 shows a high-level illustration of how two neighboring brokers would communicate with each other and with other entities in a pub/sub system. Note that two brokers *B1* and *B2* are "neighboring brokers" if they can directly intercommunicate in the overlay network

without the need of passing by a third broker.

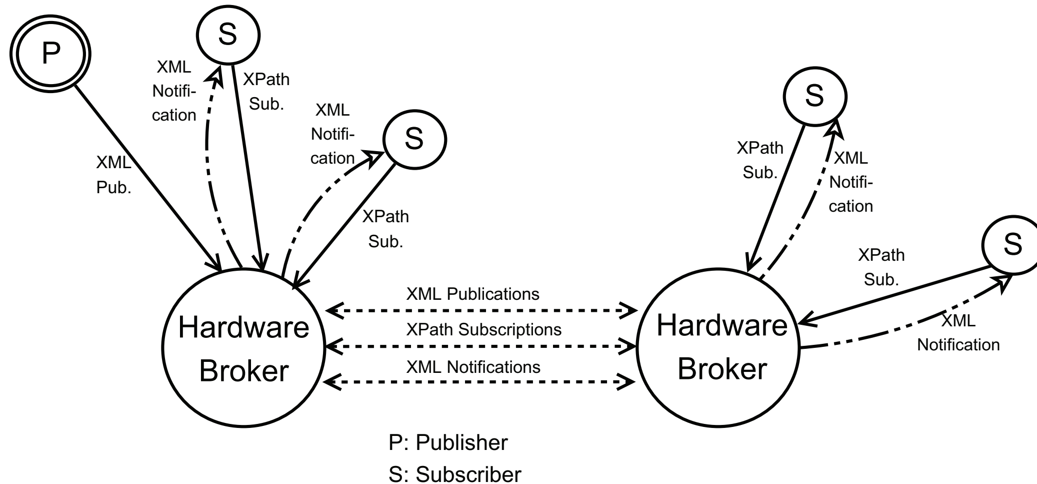

Figure 3.2: Two Reconfigurable Hardware Brokers and Their Possible Communication Messages.

3.2.2 Concurrent and Sequential Tasks

One of the great benefits of hardware solutions is providing concurrency. Therefore, we have explored the areas within the broker architecture that can run concurrently. Hence, the proposed broker carefully exploits the concurrency in its hardware design. While some of the processing tasks indeed take place in parallel, there exist some tasks that run in sequence. A broker may receive and buffer many XPath subscriptions in sequence, but it can process and register old buffered ones while it concurrently receives and buffers new ones. The matching and notification processes must take place in sequence since the notifying task is usually a result of the matching task. However, the notification process as a result of matching old subscriptions can concurrently run with the matching process for new registered subscriptions, while the broker can continue to receive and buffer newer subscriptions.

3.2.3 High-Level Block Diagram

In summary, the broker is to efficiently perform the following main tasks:

- Parsing XML publications in hardware. The broker receives each publication request *pub(P)*, stores and parses P, and then stores parsed data of P.

- Processing XPath subscriptions in hardware, while exploiting the commonalities that may exist among subscriptions. The broker receives each request *sub(S,F)*, buffers S, and registers F. In the case that common filters exist in more than one subscription, the broker registers these common filters only once to efficiently store and exploit the existing commonalities.

- Matching in hardware all available XPath subscriptions against XML publications. The broker matches all filters of registered subscriptions against available publications attempting to find any filter set $F \subset P$.

- Storing publications and registering subscriptions in small memory resources to ensure that the broker efficiently works in both fixed and mobile networks.

- Routing notifications, subscriptions, and publications based on content.

The high-level block diagram of the broker in Figure 3.3 illustrates the main processing tasks involved.

3.2.4 Content-Based Routing

In addition to its "content-based matching" role, the broker needs to decide to which link it has to route notifications for the purpose of ultimate delivery to corresponding subscribers. In the case that the broker does not find a publication matching subscriptions,

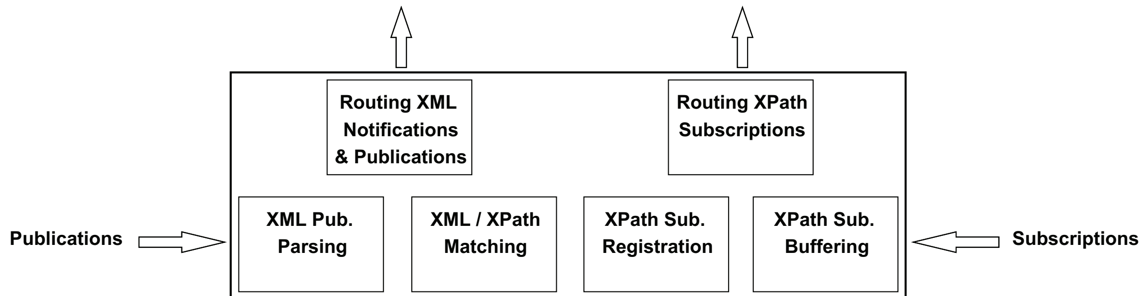

Figure 3.3: High-Level Block Diagram of the Broker.

routing of either subscriptions or publications would take place to neighboring brokers. The broker employs routing tables in order to perform the content-based routing process as a result of the matching process. The structure of these tables is fully described in Chapter 5

It is important to indicate that the routing process in the proposed broker is distinguishable from the matching process, unlike many of the existing content-based distributed pub/sub systems. In existing distributed systems, matching a part of the subscription leads to the routing task to another broker, in which matching another part of the subscription takes place and subsequently triggers the next routing task, and so on, until all parts of the subscription have matched one or more publications. This behavior implies that each subscription has been distributively stored in multiple brokers, where each of these brokers stores only a part of the subscription, and thus, the routing path actually includes the storage path of the subscription in the brokers. This association between matching and routing has the advantage of exploiting common parts that may exist among subscriptions. However, the need of storing a subscription distributively in multiple brokers adds significant latency and contributes in the excessive growth of routing tables.

In the proposed system, the whole subscription is stored in a broker, and the exploita-

tion of commonalities among multiple subscriptions is achieved thanks to the systematic storage process within the broker. Moreover, the content-based matching operation can take place in a broker and not lead to any routing operation to any other brokers. This case occurs when publications, available to the broker, match subscriptions within the same broker. Accordingly, the notification is delivered to relevant subscribers directly (or locally) communicating with the broker. The content-based routing operation takes place in one of three cases: (1) delivering a notification to a remote subscriber, (2) forwarding a subscription that has mismatched available publications, (3) and forwarding a publication to another broker. We argue that this design distinction between matching and routing has positive impact on both performance and throughput.

It should be clear that the matching operation can only find matching subscriptions within a single broker. However, the distributed routing mechanism of either publications or subscriptions can lead to finding matched subscriptions that may be only available in remote brokers in the network. Moreover, in the case of a large-scale broker network, dividing the network into "scopes" and replicating a publication in each scope can be advantageous. Note that we define a scope as a network of small number of neighboring brokers. Further related description is provided in Chapter 5.

3.2.5 Multimedia Support

X2CBBR can retrieve the multimedia data represented by an XML metadata file and made available by a publisher, providing that an end-user subscribes to receive such multimedia information. The modern term "metadata" originally appeared to identify specific data among a collection or multiple collections of data. An example of an XML metadata file, named Extensible Metadata Platform (XMP), was introduced by "Adobe Systems Inc." as a standard to describe and identify standardized and proprietary infor-

mation, thereby helping process and store content files of known formats such as PDF files or JPEG photos [54]. The same XML metadata file idea was introduced for video to support the tasks of Multimedia Information Retrieval (MIR). Thus, this type of XML metadata files contains key information about the coded multimedia content, makes it searchable similar to text, and bridges the gap between end-users and the multimedia content, thereby facilitating the MIR tasks [94].

There are many standardized XML metadata for video multimedia standards such as MPEG-7 [37] and TV-Anytime [6]. For TV-Anytime, the Content Referencing Identifier (CRID) [43] can be used to locate multimedia content, where the CRID is an IETF-registered URL prefix so as "crid://" can be recognized.

In order to support video and audio multimedia, the proposed broker needs more storage for the coded multimedia file and the corresponding metadata file. In other words, the multimedia content should exist in the proposed system of broker network instead of residing in the publisher server. The reason of this requirement is that the broker does not initiate communication with either publishers or subscribers, and then, it does not fetch content outside the broker network. Moreover, to preserve the decoupling requirement, the publishers do not communicate with subscribers and vice versa. Therefore, a publisher needs to publish its multimedia content to the system via a broker. Accordingly, the coded multimedia file and the corresponding XML metadata file should both reside in an extra storage repository of a broker, prior to matching any multimedia subscriptions. The matching process involves comparing the data of interest that exists in multimedia subscriptions with the data that exist in XML metadata files. In the event that matching occurs, the broker delivers the multimedia content to the corresponding subscriber. Figure 3.4 shows the broker when it supports and delivers both multimedia and regular content to different subscribers.

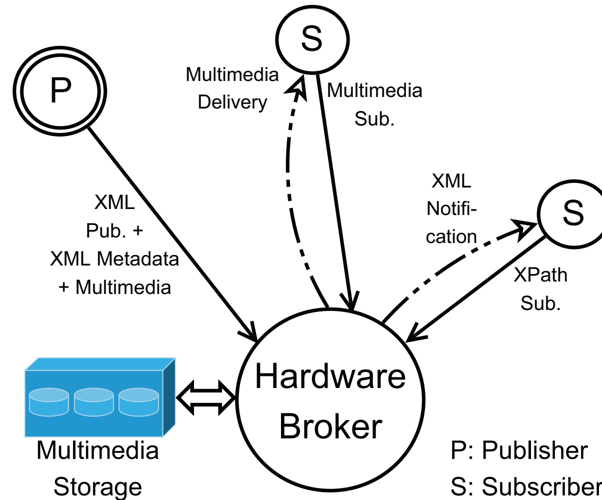

Figure 3.4: The Reconfigurable Hardware Broker that Supports Multimedia.

Given a coded multimedia content file M_m and its associated XML metadata file X_m, a user can publish the multimedia content to the system with the publication action *pub*(M_m, X_m). Receiving this multimedia publication, a broker that supports extra storage for multimedia stores M_m and parses X_m. A user can subscribe to the system with a subscription S whose request has the form of *sub(S,F)*, where F is the filter set that includes the actual multimedia locator such as the "crid" for TV-anytime. Once S matches an available XML metadata X_m, the corresponding broker routes the coded multimedia content M_m to the corresponding subscriber.

Other proposals for future Internet involve the Internet Multimedia Subsystem (IMS) that uses the Session Initiation Protocol (SIP) [106] to support multimedia. X2CBBR does not need SIP to support multimedia since it already supports XML metadata for multimedia. Moreover, processing the text-based SIP in X2CBBR would add additional processing overhead, while total decoupling between publishers and subscribers would become questionable. The reader may like to explore IMS papers including those that address hardware processing of both text-based SIP and XML-based SIP presence pack-

ets, such as [47][97][98][55][99].

The following chapters of this thesis describe the internal architecture of X2CBBR and its detailed functions.

Chapter 4

XML Parsing: The Hardware SCBXP Technique

Any publisher's data that becomes available to the X2CBBR should be of XML format. Therefore, the main task of the broker with regard of publications is XML parsing. This task of parsing involves the proposed novel hardware-based technique that this chapter thoroughly describes.

4.1 Introduction to XML Parsing

The powerful features of XML [119], such as human-readable expressiveness, interoperability, and its extensible capabilities, make it highly popular. Accordingly, new XML-based technologies have emerged in order to enable interoperable and extensible real-time communications. In addition, many web applications that are specified in XML format require efficient XML processing.

However, the advantages of XML come at the expense of intensive processing require-

ments. An XML parser has to parse the XML text; thereby identifying XML characters, checking XML well-formedness, and extracting values, as well as undertaking other optional tasks such as XML validation. In XML-based multimedia and communication systems, parsing of XML is often required in sender, receiver, and server sides during communication sessions. With regard of a single CPU core, just the XML character checking loop may occupy "more than 60% CPU cycles of the whole parsing process" [64].

With the availability of modern computer systems that include dual- or quad-core processors and large memory storage, the XML processing bottlenecks can be mitigated. However, the deployment of efficient and survivable XML-technologies is often associated with many challenges, especially in relation to mobile networks and devices where memory resources are limited and mobility time is valuable. Therefore, XML processing remains an expensive issue that has serious implications regarding systems scalability. While real-time communications must be accomplished with no downtime, software processing of XML-based messages needs to be enhanced or accelerated to meet fully real-time requirements.

In general, the basic parsing cost of XML characters that can be achieved by software ranges in the tens of clock cycles per one character. Moreover, processing other XML checks, such as character encoding validation, well-formedness, and schema validation, adds significantly more overhead to the order of hundreds of clock cycles per one character [60].

The novel hardware technique, presented in this chapter, performs XML parsing efficiently, making use of small size memory modules - a matter that is suitable for mobile networks. Even though limited memory resources are in use, the memory-efficient architecture demonstrates a scalable solution by using dual-port memory modules, a

content-addressable memory, and careful design specifications. The proposed technique takes the name of Skeleton-CAM-Based XML Parsing (SCBXP) and publicly appears in short in [44] and in details in [46].

The SCBXP is a memory-efficient technique that utilizes a skeleton as a key object to configure a Content-Addressable Memory (CAM), prior to parsing an XML document. There are two cases that involve the CAM configuration: (1) one common skeleton configures the CAM once for all XML documents; and (2) a separate skeleton configures the CAM prior to parsing each XML document. The first case can be typically applied to XML-based multimedia communications, where a predetermined XML format is constantly interchanged. The second case can be applied to XML communications that use documents of any XML format or size, including large XML documents with large skeletons. Performance analysis of these two cases and their corresponding experimental results are the subject of discussion in Section 4.5 and Chapter 6 respectively. The following sections of the current chapter provide comprehensive details of each of the different parsing stages within the technique.

In the SCBXP technique, the XML stream passes through aligning and matching stages in order to produce well-aligned data that can be processed readily in the post-matching stage. We define "well-aligned data" as the XML data aligned with specific XML characters such as '<' and '>'. For example, the XML stream sample "a<x>b</x><z" becomes well-aligned if the strings a, <x>, b, </x>, and <z become separate, so as to simplify the extraction of elements, attributes, and values. One of the advantages of the SCBXP technique is that the production of well-aligned data, at the end of the matching stage, is naturally accompanied with well-formedness checking and validation, without the need to rescan and reprocess the same XML document in terms of well-formedness and validation. From an architectural point of view, the use of

two dual-port memory modules provides parallel processing, while the use of small size memory modules is suitable for mobile devices.

The rest of this chapter is organized as follows: A high-level overview of the SCBXP technique is given in Section 4.2. Using an XML document example, a derived skeleton is defined in Section 4.3. A detailed description of the different modules and state machines of the SCBXP architecture is described in Section 4.4. In Section 4.5, analysis of the attainable processing time is discussed, in terms of configuring the CAM with either one common skeleton or multiple skeletons for parsing one or multiple XML documents. The results of implementing the technique on a Field Programmable Gate Array (FPGA), along with the results of different test cases involved, are fully stated in Chapter 6. Note that Section 2.6 provides a literature review of the related XML parsing work.

4.2 SCBXP Overview

The process of XML parsing is sequential by nature so as to keep track of starting and ending tags, of attributes and their values in relation to corresponding namespaces, and of all nested elements. Therefore, to parse an XML document in software, the processing sequence starts by loading the document (completely or partially), then reading its characters in sequence, extracting elements and attributes, writing (buffering) parsed information, and finally reading the resulting parsed data. Additional processing time is required if well-formedness and validation are to be checked.

The SCBXP hardware technique exploits each stage of the XML parsing process and speeds it up, combines different tasks simultaneously, processes certain parts concurrently, and provides a platform for highly efficient XML parsing. The block diagram of SCBXP, shown in Figure 4.1, illustrates the main stages of the technique.

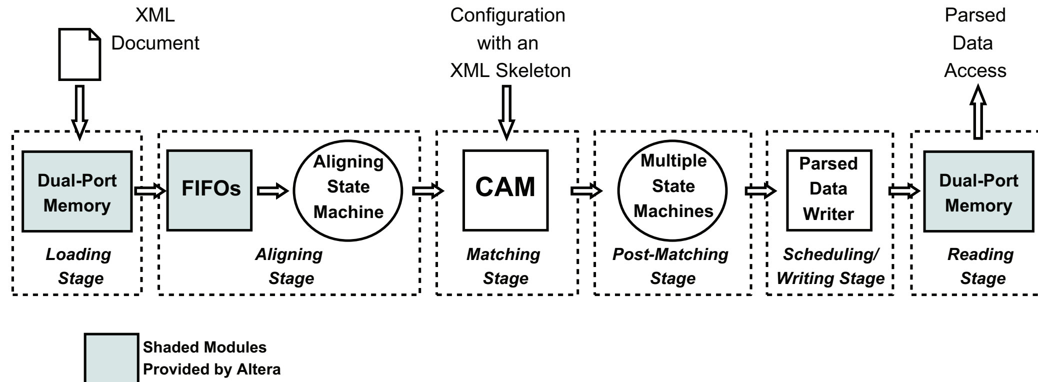

Figure 4.1: Basic Block Diagram of the SCBXP Technique.

Prior to processing a new XML document, the SCBXP must perform the task of CAM configuration. This task consists of configuring the entire CAM with a skeleton derived from the XML document to be parsed. The configuration of the CAM with an XML skeleton should have a positive effect on the overall performance of the XML parsing, because a successful match of a tagged XML string against any of the CAM contents implies that this string is well-formed and validated, due to the structure of the skeleton described in Section 4.3.

The process of loading the XML document into a dual-port memory (loading stage) is another task that must start prior to the beginning of the parsing process, but it can run afterwards in parallel with XML parsing. However, this task must terminate before the end of the parsing process in order to ensure that the entire loaded document is eventually processed.

In the following aligning and matching stages, stored XML strings (32-bit wide) are read, pushed into four FIFOs (First-In First-Out), popped out of the FIFOs in a timely-manner, matched against CAM contents, and properly aligned using an aligning Finite State Machine (FSM). The popping rate of data out of the FIFOs is based on the feedback signals and commands that originate from the matching process and the aligning FSM.

In particular, there are "pause and label" commands that control the FIFOs based on the nature of the *current* popped data, so that the *next* popped data is aligned with specific XML characters (e.g. delimiters). The well-aligned data that is produced is then sent for further processing in the post-matching stage via two 32-bit buses: V-string bus for matched and valid strings, and U-string bus for unmatched, but possibly valid, strings. Misaligned or invalid data is sent via the I-string bus and is then ignored. In the post-matching stage, multiple parsing FSMs are employed to obtain information from the well-aligned data. Subsequently, the parsed data is written in a dual-port memory that can be accessed by an external application.

Note that at the two extreme sides of the SCBXP architecture are dual-port memory modules that allow for concurrent read and write operations when both ports of each memory are used. Typically, in the loading stage, one port of the memory is utilized by the SCBXP to read loaded XML data, and the other port is employed by the outside interface to load XML data into the memory. In the reading (i.e. final) stage, one port of the memory is utilized by the SCBXP to write final parsed data, and the other port is dedicated for an external application to access parsed data.

To illustrate the sequence of the SCBXP technique functions in the first three main stages (Figure 4.1), we include the flow diagram of Figure 4.2 that clarifies the sequence of configuring the CAM, loading the XML document to be parsed, aligning the XML data with special XML characters, and matching the data against CAM contents. Essentially, loading the XML document into the memory using one port and reading it using the other port are done concurrently, with some latency to ensure that the same address of the memory is not accessed via the two ports simultaneously. We choose the latency to be five clock cycles, where 32-bit raw XML data can be written or read in each clock cycle. This latency allows at least 160 bits (20 bytes) of stored raw XML characters

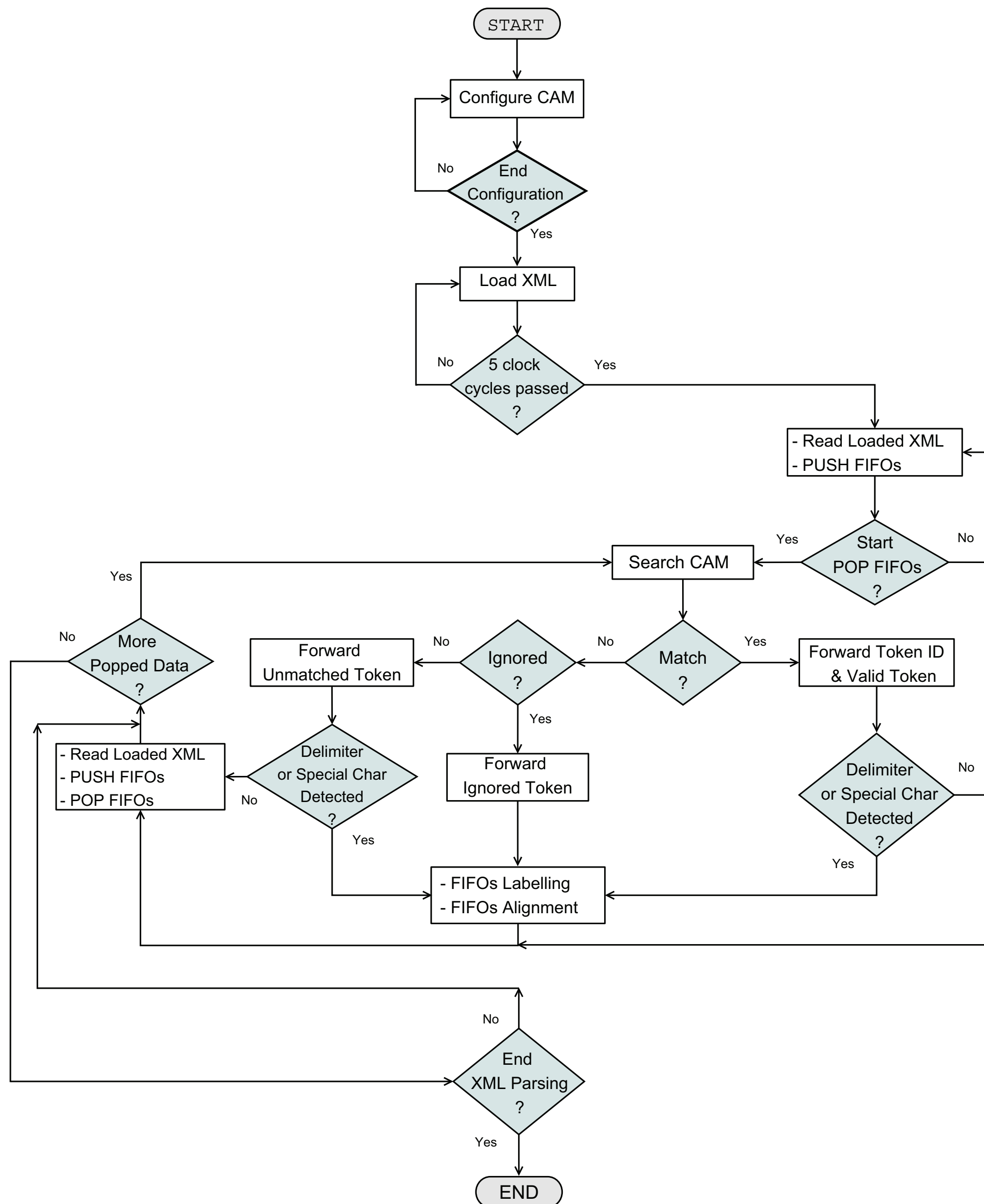

Figure 4.2: Flow Diagram Illustrating the Sequence of Main Functions in the First Three Stages of the SCBXP.

before starting the parsing process. Note that the choice of a 32-bit memory bandwidth is appropriate for standard memory interface specifications.

4.3 XML Skeleton

A possible skeleton of an XML document consists of starting and ending tags of the document. An XML example is shown in Figure 4.3, while its corresponding tokenized skeleton is illustrated in Table 4.1.

```
<stock>
   <stockitem stockid="1">
      <share val="763">
         <name>somecompany</name>
      </share>
      <qty> 60 </qty>
      <price type="USD"> 115 </price>
   </stockitem>
</stock>
```

Figure 4.3: An XML Example.

Table 4.1: A Derived Tokenized Skeleton of the Figure 4.3 XML Example.

XML String	Hex Value	Token ID		XML String	Hex Value	Token ID
<sto	0x3c73746f	1		<qty	0x3c717479	13
ck>	0x636b3e00	2		>	0x3e000000	14
ckit	0x636b6974	3		</qt	0x3c2f7174	15
em	0x656d0000	4		y>	0x793e0000	16
<sha	0x3c736861	5		<pri	0x3c707269	17
re	0x72650000	6		ce	0x63650000	18
<nam	0x3c6e616d	7		</pr	0x3c2f7072	19
e>	0x653e0000	8		ice>	0x6963653e	20
</na	0x3c2f6e61	9		</st	0x3c2f7374	21
me>	0x6d653e00	10		ocki	0x6f636b69	22
</sh	0x3c2f7368	11		tem>	0x74656d3e	23
are>	0x6172653e	12		ock>	0x6f636b3e	24

The corresponding hexadecimal values of the XML skeleton, as depicted in Table 4.1, are stored in the CAM. Note that the hexadecimal values of the English letters represent the same encoding style for both ASCII and UTF-8 standards [124]. If the width of any of these values is less than 32 bits, such value is padded with zeros (according to one method) and stored in a CAM location as a skeleton entry of 32 bits. The CAM of SCBXP is configured with this skeleton prior to parsing the XML segment example of

Figure 4.3.

Note that wildcard bits can replace the zero-padded bits, that is, if a ternary CAM is used instead of a binary CAM. However, besides extra processing cost, a ternary CAM consumes a significant amount of power - which is not desirable for battery-based mobile devices.

Once the FIFOs pop out data, the CAM logic stores the popped data into a 32-bit register. Then, it searches for the existence of such data among its contents. Based on the CAM decision (i.e. Hit or NoMatch), a state machine (the aligning FSM) sends feedback signals to the FIFOs to control the popping rate of each FIFO. This procedure updates the 32-bit register contents and aligns the next popped data with specific XML characters, such as the delimiters '<' and '>', according to the location of such characters within the popped data. If the CAM flags a hit signal, it issues a token ID for every string of 32-bit matched data. Each token ID represents the CAM location address in which the matched string was stored during configuration. More elaboration on handling the skeleton entries of the CAM, matching data against these entries, and producing aligned data are outlined in Sections 4.4.2 and 4.4.3.

In the SCBXP technique, matching a starting tag with a CAM hit and matching the corresponding closing tag with another CAM hit naturally translate into well-formedness. This natural implication is possible since the skeleton includes these tags in its structure (providing that the starting tag occurs before the ending tag; otherwise, the SCBXP flags an error either in the aligning or in the post-matching stage). For example (see Table 4.1), a matched starting tag "<stock>" leads to the generation of token IDs 1 and 2. In addition, a matched ending tag "</stock>" corresponds to the starting tag "<stock>" and leads to the generation of token IDs 21 and 24. Therefore, not only does the utilization of token IDs facilitate the post-matching parsing process, but it

also implies that these token IDs correspond naturally to matched strings that are valid and well-formed. Further well-formedness and validation checks are performed in the post-matching process. Eventually, at the end of the whole parsing process, the XML well-formedness can be fully verified, but XML validation is fully verified only for matched data. If additional data is to be validated, such data must be added to the skeleton. If full support of XML validation is needed for all contents of the XML document, including the values that attributes may have, then a DTD (Document Type Definition) or an XML schema must be used. However, the SCBXP has no support currently for either a DTD or an XML schema.

It is useful to mention that, based on CAM principles, matching XML characters against the contents of the CAM does not allow for the occurrence of a multi-hit result. Therefore, identical elements that are frequently encountered in an XML document are included in the derived skeleton only once. For example, the string "<sto" exists twice in the XML example (the 1st and 2nd lines of Figure 4.3), but it is entered only once in Table 4.1. Then, the sequence of token IDs 1,2 issued for matched "<stock>" is valid as well as the sequence of token IDs 1,3,4 for matched "<stockitem".

A single skeleton of a variety of XML documents can configure the CAM only once, if these documents contain the same elements but different attributes or values. This case is very common in XML-based communication systems that use a common special format. For example, in Voice over IP (VoIP) or Instant Messaging (IM) applications, an XML-based format for the Session Initiation Protocol (SIP) *presence* information is exchanged in a standardized common format called Presence Information Data Format (PIDF) [113] [68]. The skeleton of such an XML format can be downloaded into the CAM only once, while the SCBXP may parse many exchanged presence messages normally without the need to reconfigure the CAM.

4.4 The SCBXP Internal Architecture

The internal architecture of the SCBXP technique is depicted in Figure 4.4.

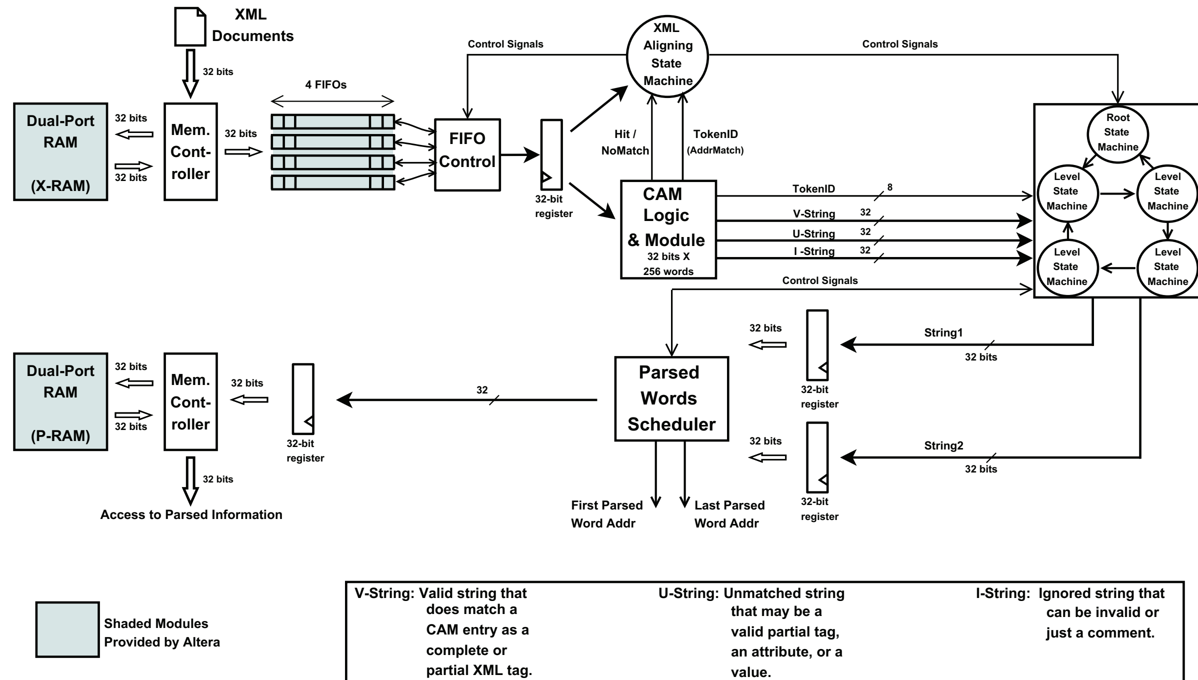

Figure 4.4: Main Components of the SCBXP Internal Architecture.

The main modules that make up the architecture are as follows:

- A dual-port memory module that is located in the first (loading) stage.
- Four 8-bit FIFOs and their associated control module that are part of the aligning stage.
- An XML aligning FSM that operates in the aligning stage.
- A CAM that is the main part of the matching stage.
- Five parsing state machines that act in the post-matching stage.

- A scheduler/writer module that operates in the scheduling/writing stage.
- A dual-port memory module that is located in the last (reading) stage.

The following sections provide the roles of each of the modules in the SCBXP stages.

4.4.1 Loading and Reading Stages: The Two Dual-Port Memory Modules

A dual-port memory, named X-RAM in Figure 4.4 (X denotes XML), stores raw XML data. The read/write width of each port is 32-bit (4 ASCII characters). In the loading stage, the X-RAM receives XML data from the external application (32 bits at a time) via one port of the memory. At the same time, the SCBXP utilizes the other port of the same memory to read the loaded XML characters (32 bits at a time). At the beginning, this reading port must not be used for a latency value of at least five clock cycles to ensure that the same memory location is not accessed via different ports at the same time. With the occurrence of simultaneous write and read operations, the SCBXP can commence the parsing process without waiting for the loading end of the entire XML document. The effect of this concurrency on the overall parsing performance is certainly beneficial.

Another dual-port memory, named P-RAM in Figure 4.4 (P denotes Parsed), stores the resulting parsed data. Again, the utilization of two ports allows for accessing the P-RAM concurrently. The read/write width of each port is 32 bits (4 ASCII characters). The P-RAM receives the parsed results at the scheduling/writing stage (32 bits at a time) via one port of the memory. The parsed information is concurrently accessed via the other port by the outside interface in the final (reading) stage. The latency rule used with the X-RAM (i.e. the minimum five clock cycles of latency) is applied with the

P-RAM as well, in order to avoid simultaneous accesses of the data stored at the same memory location.

Note that the type (e.g. element, attribute, or value) of the parsed data is also buffered in the final stage. Thus, the parsed data can be entirely or partially accessed by reading the P-RAM, depending on whether all parsed information is needed by an application or just a specific parsed portion of the document. Section 4.4.6 further elaborates upon this point.

4.4.2 Aligning Stage: The FIFO Control Module

The X-RAM delivers stored XML data (32 bits at a time) to a set of four FIFOs. Each FIFO handles 8 bits of data. At the beginning of an XML stream, the XML declaration must be included. Afterwards, the FIFO Control module performs the following three tasks concurrently:

- It verifies whether the characters popped out of each FIFO are valid according to the corresponding encoding style. Currently, UTF-16 and UTF-32 encoding forms are not supported.
- It sends every 32-bit popped data to the CAM for matching purposes.
- It identifies any specific XML characters, such as delimiters, and sends corresponding control signals to the XML aligning FSM.

Based on the signals of the FIFO Control module, and on the CAM decision signals, the aligning FSM evaluates the FIFO rule that should be applied (Figure 4.5). Subsequently, this FSM sends corresponding "pause and label" commands back to the FIFO Control module. These commands serve in controlling which FIFO must or must not

```
1: RULE A
     no change for FIFO labels
       (for FIFO1, FIFO2, FIFO3, FIFO4, original labels: 1,2,3,4 respectively)
     pop all FIFOs (for each FIFO pause = 0).
2: RULE B
     right-shift and rotate FIFO labels 1 position
       (for FIFOs labeled 1,2,3,4, their labels become 4,1,2,3).
     pop FIFO labeled 4
       (for FIFOs labeled 1,2,3, pause = 1; for FIFO labeled 4, pause = 0).
3: RULE C
     right-shift and rotate FIFO labels 2 positions
       (for FIFOs labeled 1,2,3,4, their labels become 3,4,1,2).
     pop FIFOs labeled 3,4
       (for FIFOs labeled 1,2, pause = 1; for other FIFOs, pause = 0).
4: RULE D
     right-shift and rotate FIFO labels 3 positions
       (for FIFOs labeled 1,2,3,4, their labels become 2,3,4,1).
     pop FIFOs labeled 2,3,4
       (for FIFO labeled 1, pause = 1; for other FIFOs, pause = 0).
5: RULE E
     no change for FIFO labels.
     no pop of FIFOs (for each FIFO pause = 1).
```

Figure 4.5: Rules of FIFO Alignment (FIFO Labels and Pause Commands).

pop data out. The FIFO control module accordingly executes the commands on all or some of the FIFOs. Therefore, all four FIFOs pop data out, if Rule A is applied; three FIFOs pop data out, if Rule D is applied; two FIFOs pop data out, if Rule C is applied; and only one FIFO pops data out if Rule B is applied. Figure 4.6 graphically illustrates these rules, and shows the displacement of labels on the FIFOs according to the position of the special character '<' in a 7-character sequence example (<stock>). Therefore, when '<' is in the FIFO labeled FIFO1, Rule A is applied in the next cycle, so that the next popped data will be "<sto" (simultaneously popped out of FIFOs 1,2,3, and 4 respectively); and in the following cycle, Rule A will be applied again where the new popped data will be "ck>". When '<' is in the FIFO that is not labeled FIFO1, one of the rules - except Rule A - is applied in the next cycle, while the labels change according to the applied rule. However, the data "<sto" will be popped out in the following cycle upon the application of Rule A; and in one more cycle, the popped data will be "ck>" when Rule A is applied again.

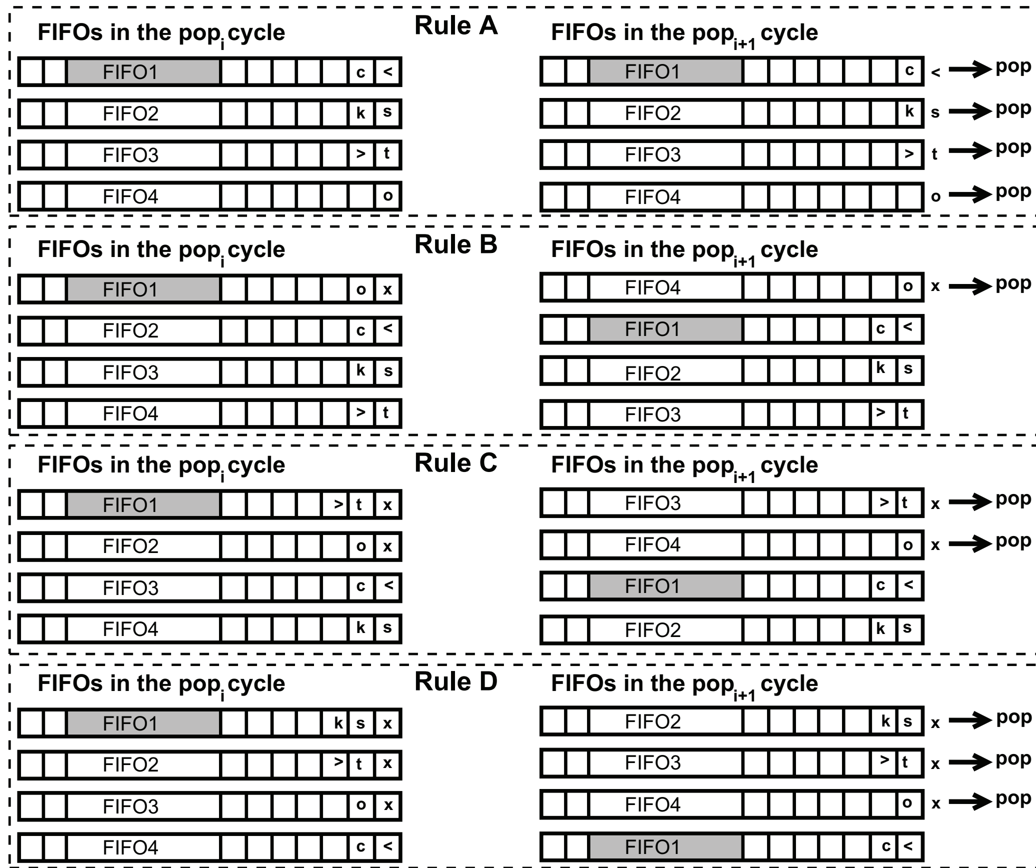

Figure 4.6: Graphical Illustration of FIFO Rules.

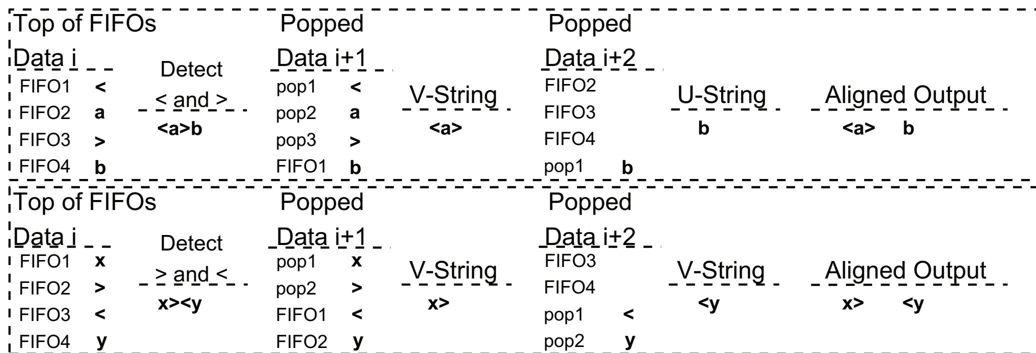

Figure 4.7: Two Examples of Strings “<a>b” and “x><y” Aligned in FIFOs.

Figure 4.7 illustrates the aligning process in the FIFOs for other two examples of strings, starting from the clock cycle during which either “<a>b” or “x><y” happens to be on the top of the FIFOs. In these two examples, the strings “<a>”, “x>”, and

"<y" are assumed to be in the skeleton configured in the CAM.

Note that Rule E is mainly applied when the CAM does not need new popped data. As an example of such a case, the CAM may need to rematch the same old data with or without whitespaces (see Section 4.4.3.1).

This process allows the CAM to receive FIFO data that are aligned with specific XML characters, so that the data can likely match one of the CAM contents. It is important to mention that the FIFO Control module, the XML aligning FSM, and the CAM work in harmony, in both the aligning and matching stages. Accordingly, the Fifos can pop out data in each clock cycle if no alignment is needed. In the case that alignment is needed, the aligning process allows the Fifos to pop the next aligned data in three clock cycles. Figure 4.8 illustrates this harmony, where both negative and positive clock edges are utilized to sample the data. Note that the timing diagram in Figure 4.8 assumes no delays in order to preserve simple illustration.

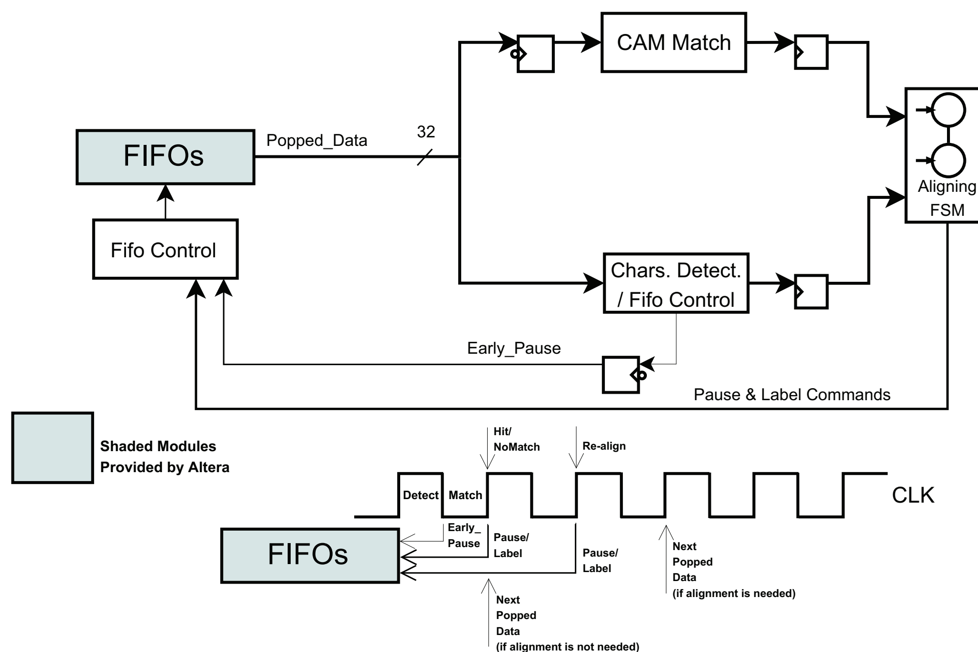

Figure 4.8: The Harmony Between the FIFOs and both the Aligning and Matching Stages.

4.4.3 Aligning Stage: The XML Aligning State Machine

In each of its states, the XML aligning FSM runs an algorithm to determine which FIFO rule (among the rules of Figure 4.5) should be applied. This rule determination is done according to certain inputs, such as the signals coming from the FIFO control module (e.g. signals produced based on the detection of characters ‘<’ and ‘>’), and from the CAM module (e.g. Hit/NoMatch and token ID). Accordingly, the aligning FSM produces its outputs, which are predominantly “pause and label” commands forwarded to the FIFO Control module for alignment as mentioned earlier. Figure 4.9 illustrates typical inputs and outputs of one of the aligning FSM states, while Figure 4.10 depicts all states of the aligning FSM.

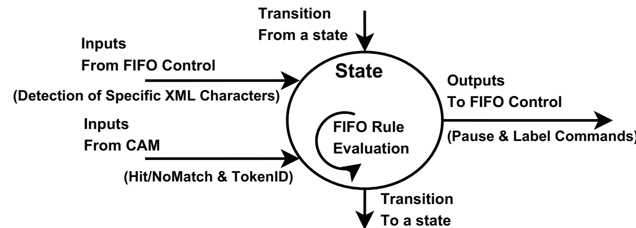

Figure 4.9: Typical Inputs/Outputs of One State of the XML Aligning State Machine.

Starting from its initial state, the state machine makes a transition to the StartPI state upon the reception of the first valid XML string “<?xml”. This state indicates the beginning of the Processing Instruction (PI) of an XML document. The FSM remains in this state until the end of the PI is detected. This detection triggers a transition to the EndPI state. Note that the XML document must be in non-canonical form. Thus, the XML declaration must appear in the beginning of a new XML stream [118]. A transition to the EndDelimiter state occurs upon the detection of the ‘>’ character, while the detection of the ‘<’ character leads to a transition to the StartDelimiter state. The occurrence of a string directly following ‘<’ allows a transition to the StartTAG state,

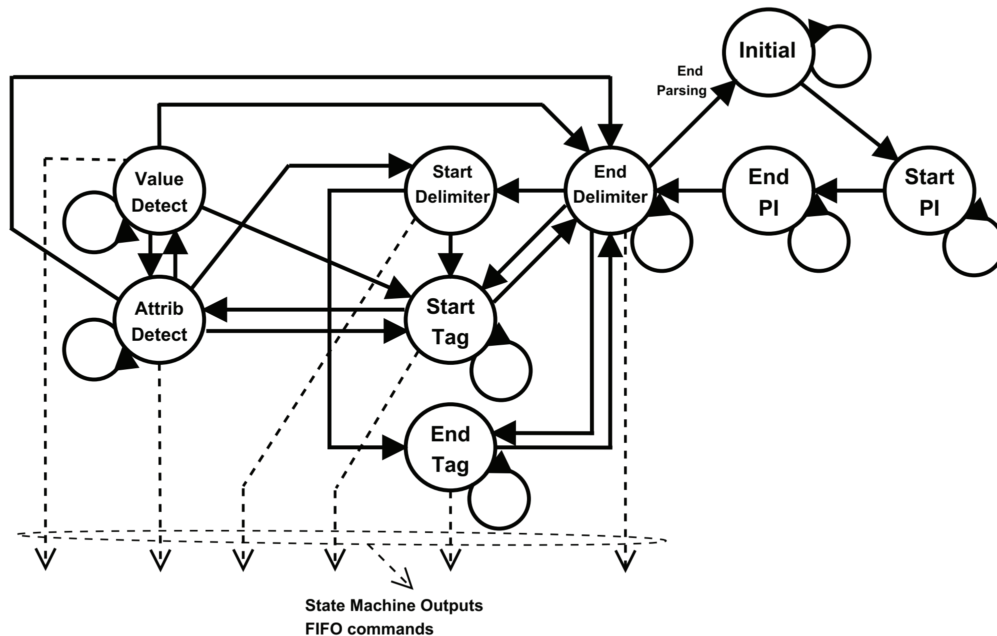

Figure 4.10: XML Aligning State Machine.

while the presence of '/' after '<' leads to the EndTAG state. If a space character (0x20) is detected after the last character of a string while the current state is StartTAG, a transition is made to the AttribDetect state, thereby indicating the presence of an attribute. Subsequently, the FSM steps to the ValueDetect state upon the detection of an attribute value that is inserted between quotation marks.

4.4.3.1 Whitespaces

The SCBXP maintains a few algorithms to handle the whitespaces that normally exist in most XML documents. In the skeleton of Table 4.1, zeros replace whitespaces that may occur in the last part of some entries. Accordingly, prior to the matching stage, the SCBXP places zeros instead of whitespaces that may be found at the end of any string to be matched with the CAM contents. If whitespaces are included at the end of some skeleton entries, the SCBXP must not replace whitespaces with zeros prior to

the matching stage. Note that a space must not directly precede an element name. Otherwise, an error occurs (e.g. "< stock>"). However, if a space is present within an element name, such as in "<sto ck>", the SCBXP treats "sto" as an element, and "ck" as an attribute. Then, a transition to the AttribDetect state is made by the aligning FSM, and an error is triggered due to the absence of the equality operator and the quotation marks that would identify the value of the attribute.

4.4.3.2 Namespaces and Attribute Uniqueness

The SCBXP picks up the name of any namespace that may be included in an XML document. If such a name is not included, and the ":" character is encountered within an attribute, as in a:b="1", the SCBXP triggers an error since the name 'a' is not assigned to a namespace. Currently, there are some limitations to the support of namespaces as follows: (1) a namespace must not be longer than 256 bits (32 ASCII characters), (2) the name of a namespace must not be longer than 32 bits (4 ASCII characters), and (3) a maximum of three different namespaces is allowed. To support too many namespaces with a higher number of characters, the SCBXP must employ a new memory module and an additional control logic circuitry - which would add more chip area and more complexity. Note that the SCBXP evaluates, bit-by-bit, the existence of two or three identical namespaces using XOR gates.

In addition to namespaces, the SXBXP checks the uniqueness of attributes within a tag. This checking occurs when the aligning FSM makes transitions from AttribDetect state to ValueDetect state back and forth.

4.4.4 Matching Stage: The CAM Logic and Module

As previously noted, the aforementioned aligning stage allows the CAM to receive FIFO data that is aligned with specific XML characters. Once the CAM locates a match for any string of the FIFO data, it provides a corresponding ID (TokenID in Figure 4.4), which can be used in further processing tasks.

In the case that a string less than 32-bit wide is to be matched against the CAM contents, the string is padded with zeros providing that the skeleton entries are padded with zeros (instead of whitespaces), as discussed in Section 4.4.3.1. For the example "<a>b" in Figure 4.7, the string "<a>" is a 24-bit ASCII character that has to be padded with 8 zeros before matching it in the CAM. For the other example "x><y" in Figure 4.7, the string "x>" is padded with 16 zeros before matching it in the CAM. However, the CAM pads the string "<y" with 16 zeros only if 'y' is identified by the aligning FSM as an element. Otherwise, the CAM pads the string with the characters that follow it, as in the example "<stock>" illustrated in Figure 4.6.

The CAM sends its decision outputs (Hit/NoMatch signals, as well as the issued token ID if a hit occurs) to the aligning FSM. Since a successful match of a string implies that the matched data is well-formed and valid, the CAM places each of the matched strings on the 32-bit V-String bus. Alternatively, if the CAM identifies data as unmatched, such data may still be valid; thus, the task of further checking the well-formed data is fulfilled in the post-matching stage. The CAM places this unmatched data on the 32-bit U-String bus.

Figure 4.11 depicts the internal architecture of the CAM and its match logic leading to the V-String and U-String buses. In the case that an invalid string is detected, the identified string must be placed on the 32-bit I-String bus and then ignored. A string

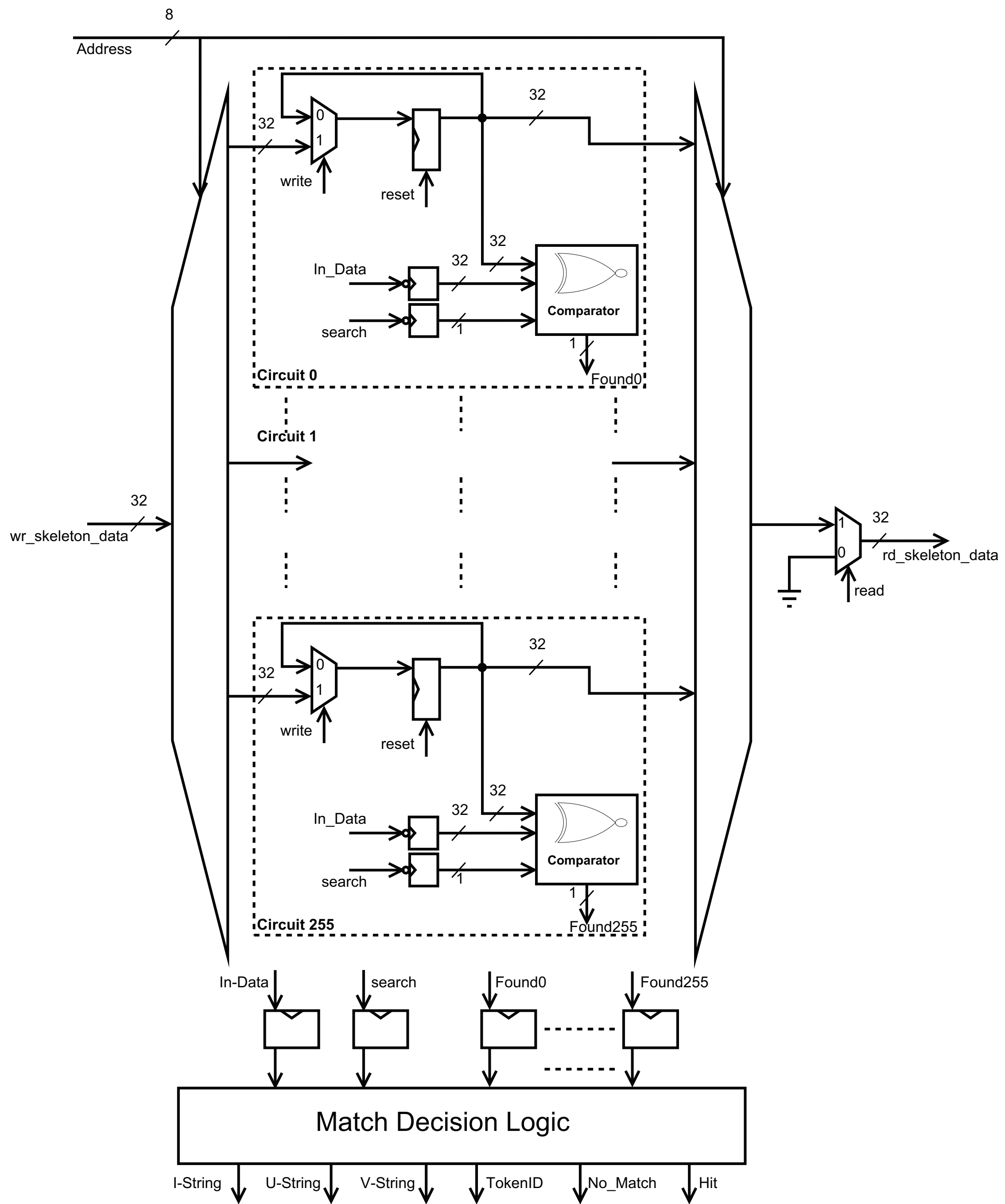

Figure 4.11: CAM of SCBXP.

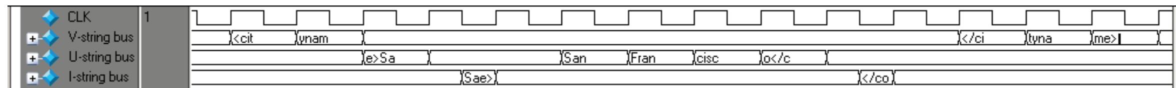

Figure 4.12: Example of Well-Aligned Data Placed on the V-string and U-string Buses, and Misaligned Data Placed on the I-string Bus.

would be considered invalid if it is just a comment that does not need to be parsed, or if it is formed temporarily during the data alignment. These cases do not violate the well-formedness required for XML data.

For example, the XML segment *<cityname>San Francisco</cityname>* appears on the three buses, as shown in Figure 4.12 using the ModelSim simulator [79]. Elements containing strings conforming to the XML skeleton appear on the V-string bus. These strings are naturally well-formed and validated. Attributes and values (and possibly partial elements) parade on the U-string bus, while the strings placed on the I-string bus are ignored. In the example illustrated in Figure 4.12, the misaligned data ("Sae>" and "</co") appears one clock cycle (on either the falling or the rising edge of the clock) after one of the specific XML characters < and > is detected in the middle of the U-String data ("e<Sa" and "o</c"). In one more clock cycle, the data is realigned and placed on either the U-String or the V-String bus.

The placement of aligned data on different buses facilitates picking up the parsed data while further checking of well-formedness can be simply performed.

It is useful to mention that it would be possible to send the V-string, U-string, and I-string data on one common 32-bit bus, instead of three separate 32-bit buses, to reduce the amount of data coming out of the CAM logic module. In this presumable case, the V-string would be identified using the token ID. However, extra-processing would be needed to differentiate between U-string and I-string data. Furthermore, this would

result in processing delays, since all the data to be processed appear in sequence on a single bus. Thus, the utilization of three buses has a positive impact on the parsing performance.

4.4.5 Post-Matching Stage: The Multiple Parsing State Machines

At the end of the matching stage, the SCBXP has accomplished its main parsing tasks by properly aligning XML strings. If a violation of well-formedness had been detected prior to the post-matching stage, an error would have been flagged. Otherwise, the well-formed and validated strings of the V-string bus are ready for pickup, while partially well-formed and validated strings of the U-string bus need to be fully checked for well-formedness.

In this stage, picking up the well-aligned data does not need "multiple" state machines. Instead, the basic mode of operation performs the post-matching tasks using the following components: one parsing FSM, a logic circuitry to pick up data directly from the buses, and a logic circuitry to check the well-formedness. In particular, multiplexers pick up the V-String and U-String data depending on the current state of the FSM and on the signal "pick_att_val" (Figure 4.13). However, to boost the performance of picking up the data in the first five nesting levels of the XML, five parsing state machines are employed. The choice of only five state machines is made to keep the implemented chip area as concise as possible. If more than five nesting levels exist, the data pickup and the well-formedness checks are done for the extra levels using multiplexers and control logic, similar to the basic mode of operation, while each of the state machines waits on the end of any of the first five levels.

Figure 4.13 illustrates the structure of one parsing state machine. The outputs of

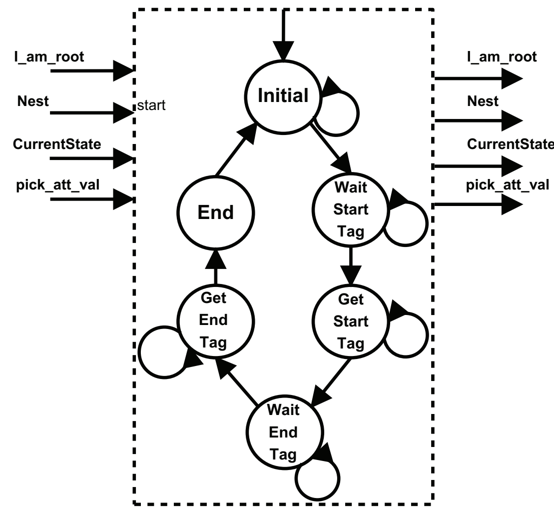

Figure 4.13: One Parsing State Machine.

such an FSM can drive another identical FSM, which in turn delivers similar outputs to a third FSM, and so on, as is depicted in Fig 4.14. The first state machine starts with the root element (e.g. "<stock>" in Figure 4.3), and asserts the FSM output "I_am_root." During its WaitEndTag state, this "root state machine" (see Figure 4.4) detects all nested elements, attributes, and values that were properly aligned in the previous stages.

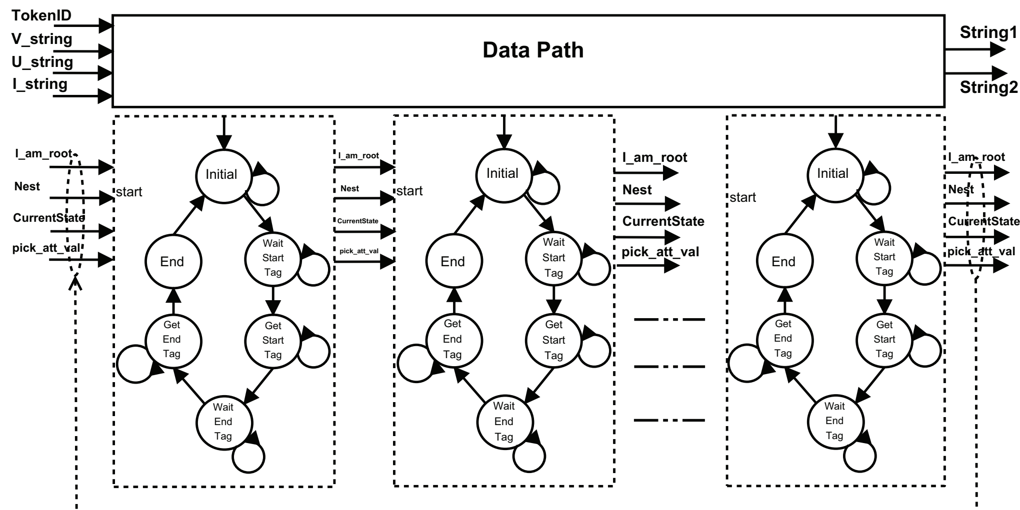

Figure 4.14: Multiple Parsing State Machines Controlling the Data Path.

Using a second FSM, processing of the nested element starts immediately once the

first FSM asserts its “nest” output (Figure 4.13 and Fig 4.14). Since the structures of both state machines are identical, the second FSM can assert its “nest” output to activate a third identical FSM, and so on. The root state machine is dedicated for the root tag; thus, it remains active during the whole parsing process. The other state machines run on other nesting levels and get the name of “level state machines” (Figure 4.4). Even though these multiple state machines start their initial operation in a round-robin fashion, some of these machines continue to run in parallel on nested elements (i.e. an FSM may operate without waiting the end of another FSM operation, as Figure 4.15 illustrates).

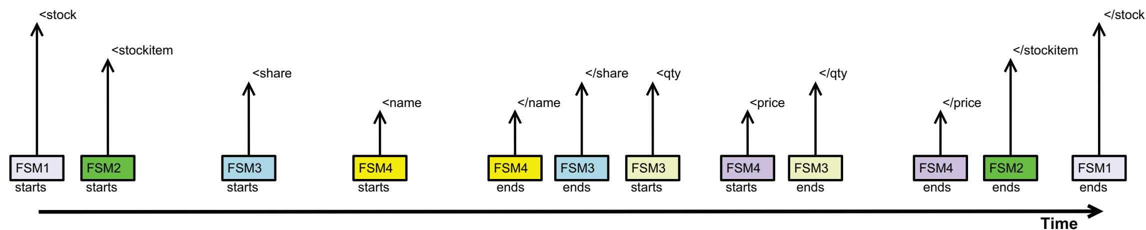

Figure 4.15: The Timing Schedule of the Multiple State Machines for the XML Example of Figure 4.3.

While each state machine is in its states GetStartTag and GetEndTag, an implemented logic circuitry checks the well-formedness of the U-string data. For example, for the well-aligned data example in Figure 4.12, the string $e{>}Sa$ is on the U-string bus; thus, it is an unmatched string that can represent a partial valid element, an attribute, or a value. The existence of the character $>$ within that string allows for an immediate pickup of the letter e, as this letter is considered a part of the starting tag *cityname*. A similar approach is applied for the string $o{<}/c$ considering the letter o a part of the value *San Francisco*. Then, the implemented logic compares between the extracted starting and ending tags to examine the well-formedness of the segment. This comparison logic is limited to 160 bits. Therefore, the number of bits for an element name should not exceed

160 bits (20 ASCII characters) in order to verify the well-formedness at this stage.

Attributes and their values appear separately on the U-string bus. Therefore, picking them up directly from the buses is possible, as well as during the transitions of one of the state machines, depending on which one is operating.

4.4.6 Scheduling/Writing Stage: The Parsed Words Scheduler

While initially activated in a round-robin fashion, the parsing state machines may run in parallel afterwards. Each state machine can later terminate its operation at the end of a nesting level, and re-operate at the beginning of a new nesting level. If two (or more) state machines happen to re-operate simultaneously (i.e. the reactivation does not adhere to the round-robin fashion in contrast to the initial activation), these machines will have to control the same nesting level simultaneously. This situation indicates an over-utilization of available resources, which may eventually lead to an efficiency decrease. In this case, the "Parsed Words Scheduler" (or simply the "Scheduler") sends a feedback signal to deactivate one machine immediately during the parsing of a certain level. Later, the deactivated machine resumes operation at the beginning of another nesting level.

If the deactivated machine is supposed to resume operation at a certain point, but it is not triggered at the proper time, a part of the parsed data may be lost. This undesirable event indicates an underutilization of available resources. In such a case, the scheduler issues a feedback signal to reactivate the delayed state machine. A state machine can have a delayed reactivation if the end of a nesting level is not reported at the proper time, while a new nesting level starts. For example, an ending tag may be too short and a new level with a starting tag occurs right after the ending delimiter of the previous level in the absence of any whitespaces.

To illustrate the parallel function of the multiple parsing FSMs, and the timely acti-

vation and deactivation of these machines, Figure 4.15 depicts the timing of the starting and ending operations of the machines using the XML example from Figure 4.3.

Another task that must be undertaken in this stage is scheduling the parsed data placed on the two 32-bit buses “String1” and “String2” of the data path (Figure 4.14). The data on one bus can be an identified “element,” while the data on the other bus can be one or more associated attributes and values. The scheduler has to handle the data on both buses using buffers.

Moreover, the scheduler reports the progressive locations of the parsed data in the memory to the external interface, including the first and last memory addresses with which contiguous parsed data can be accessed, as shown in Figure 4.4. Such address reporting permits the external application to access all parsed data, or optionally just a needed portion of contiguous parsed data, in the last reading stage. Note that the type (e.g. element, attribute, or a value) of the stored parsed data is buffered with each address of the P-RAM. Thus, when the external application reads the parsed data starting from the first reported address, the type of the read data will become available to the application. Note that the parsed information is written into the P-RAM dual-port memory using one of its two ports, while the last reading stage is performed using the other port of the P-RAM, as described earlier in Section 4.4.1.

4.5 Processing Time

While the previous sections of this chapter have described the architecture and the parsing stages of the SCBXP technique, this section analyzes the processing time that each of the stages would require, taking into consideration the concurrent features of the hardware architecture.

Assuming that all parsing stages run sequentially, we estimate the total time T_{proc} required to process an XML document as follows: $T_{proc} = T_C + T_L + T_{parse} + T_A$, where T_C is the total time needed to configure the CAM with a skeleton, T_L is the total time needed to load an entire XML document, T_{parse} is the actual time needed to read and parse the loaded XML data and to store the resulting parsed data, and T_A is the total time needed to access all the parsed data extracted from the entire XML document.

However, as mentioned earlier, parsing XML starts after a latency of five clock cycles from the beginning of loading the XML data into the X-RAM. Similarly, accessing parsed data can start after a latency of five clock cycles from the beginning of writing the parsed data into the P-RAM. Moreover, after the end of the storage of the last parsed data, the external application needs latency of at least five clock cycles to access the last parsed information. Thus, the minimum latency $T_{L_{lat}}$ before starting to read the loaded data is the actual loading time considered for the processing time (i.e. $T_{L_{lat}} = 5$ clock cycles) instead of T_L. Similarly, after the end of the parsing process, the minimum latency $T_{A_{lat}} = 5$ clock cycles should be included in the processing time instead of T_A.

Then, the minimum processing time needed to parse and access the parsed data is: $T_{proc} = T_C + T_{L_{lat}} + T_{parse} + T_{A_{lat}}$, where T_{proc} is computed for parsing one XML document, while the CAM is configured with a single corresponding skeleton that is ≤ 1 KB (which is the capacity of the CAM in Figure 4.4).

Therefore, $T_{proc} = T_C + T_{parse} + 10$ clock cycles. T_{parse} varies depending on the structure and the size of the XML document, on the skeleton entered in the CAM, and on the corresponding parsing time needed in each stage.

For parsing multiple XML documents, two cases come into consideration for computing the processing time.

4.5.1 Case I: One Common Skeleton, Multiple Chunks of XML Data

For XML-based communication systems that use a fixed XML format, the skeleton does not change for every new XML document. Accordingly, T_C remains constant in this case, because there should be no need to reconfigure the CAM for parsing new XML documents. Thus, the total processing time needed for parsing N XML documents is: $T_{procN} = T_C + (N * T_{parse}) + 10$ clock cycles, where T_{parse} is considered the "average" time needed to read and parse one XML document (loaded entirely in the X-RAM) and to write the resulting parsed data in the P-RAM. The average processing time of one XML document is: $T_{proc} = T_{parse} + [(T_C + 10)/N]$ clock cycles. When N is large, i.e. a large number of XML documents are being parsed with the same skeleton configured in the CAM, T_{proc} tends to be close to T_{parse}, where the CAM configuration time, the XML loading time, and the final access time become marginal.

If a single large XML document is to be parsed and its skeleton fits entirely in the CAM, the XML document is divided into N chunks, where each chunk is ≤ 1 KB. Then, each chunk is loaded entirely into the X-RAM. Thus, $T_{proc} = T_{parse} + [(T_C + 10)/N]$ still holds true, where T_{proc} is the average processing time of one chunk of the XML document, and T_{parse} is considered the average time needed to read, parse, and write the parsed data of one chunk of the document. Note that if the capacity of the CAM is just 256 32-bit words, a skeleton of 1 KB of data fits entirely in the CAM. A very large XML document may still have a skeleton ≤ 1 KB, if the document includes many repetitive words (especially elements).

4.5.2 Case II: Multiple Skeletons, Multiple Chunks of XML Data

In the case that the skeleton of a single large XML document does not fit in the CAM, the document must be loaded in chunks into the X-RAM, and a new skeleton must be used for parsing every chunk. This case is similar to that of parsing multiple XML documents of any XML format. The total processing time in such cases for multiple N chunks is: $T_{procN} = (N * T_C) + (N * T_{parse}) + 10$ clock cycles, and the average processing time of one chunk is: $T_{proc} = T_{parse} + T_C + (10/N)$ clock cycles. Thus, when N is large, the CAM configuration time is taken into account, while the XML loading time and the final access time remain marginal.

In Chapter 6, the results of experiments conducted for the two cases above are shown, using small and large XML documents.

Chapter 5

Architecture of the Hardware XML/XPath Broker/Router

The SCBXP technique, described in Chapter 4, makes part of the X2CBBR architecture and essentially performs the tasks of XML parsing of publications. Chapter 3 has provided an overview of the proposed broker functions along with relevant definitions and diagrams. This chapter describes the broker's internal architecture in detail excluding the SCBXP.

5.1 Introduction

A broker (or content-based router) in a content-based pub/sub system aims at efficiently matching subscribers' interests against published content, and subsequently notifying subscribers of corresponding publications. In such systems, messages are routed based on their content rather than on IP destination addresses. While publishers represent their content in XML [119], subscribers express their interests (often called filters) possibly

using XPath [120] subscriptions.

In XML-aware content-based pub/sub systems, a sluggish broker can ruin the performance, whether these systems are centralized or distributed. In particular, the processing burdens of an XML/XPath broker may lead to a few bottlenecks. The first bottleneck stems from inefficient parsing of the XML published content. The second bottleneck resides in poor processing of XPath subscriptions. The third bottleneck appears when filters of many XPath subscriptions are evaluated (matched) sequentially (rather than simultaneously) against XML published content. As a result of successful XPath matching against XML, the final bottleneck occurs when notifications are inefficiently delivered to corresponding subscribers. Figure 1.1 illustrates these bottlenecks, which need to be efficiently overridden to achieve a high performance XML/XPath pub/sub system.

Many software solutions primarily focus on the matching process, by applying means to match filters of many XPath subscriptions at once against XML content, instead of evaluating filters of each subscription individually. The simultaneous matching of filters can lead to a high-performance pub/sub system, if filters' commonalities can be efficiently exploited. However, such software processors face performance tradeoffs, due to the lack of inherent parallelism, and tend to solve these issues just partially. The problem can become even more serious in mobile devices, where memory resources are limited.

This chapter presents the hardware architecture of the X2CBBR, which can fall in an XML content-based pub/sub system and operate in mobile networks. Besides forwarding tasks, this architecture performs hardware-based XML parsing and XPath processing, as well as filter matching that is able to efficiently exploit all commonalities existing in many XPath subscriptions. Moreover, the X2CBBR architecture can undertake concurrent tasks in order to represent an effective solution that provides high performance.

This chapter is organized as follows. Section 5.2 sheds light on main broker tasks along

with related definitions. Section 5.3 describes in detail the three main parts that make up the X2CBBR architecture: The subscription processing unit, the XML/XPath processing and matching unit, and the notification processing unit. Section 5.4 extends the broker functions to include and describe in detail the content-based routing mechanisms and algorithms employed in the X2CBBR. Implementation results of the broker on an FPGA are illustrated and discussed in Chapter 7. Note that Section 2.7 provides a literature review of existing XML/XPath filtering systems.

5.2 Main Broker Tasks

In a distributed content-based pub/sub system, a broker has to perform tasks that may vary according to its location within a network of brokers (i.e. in the middle or the edge of the network), besides normal processing and matching tasks.

5.2.1 Definitions

The following items define some concepts and terms that are useful in the discussion of broker tasks.

1. An intermediate broker is a broker that takes part of a network of brokers and does not directly communicate with any subscriber. It exclusively communicates with brokers in the network.

   Definition 1. *Let B_B be a set of brokers in a network. An intermediate broker $B_m \in B_B$ is a broker that only communicates with a broker $B_i \in B_B$ identified as a neighboring broker for B_m.*

2. An edge broker is a broker that takes part of a network of brokers and directly

communicates with one or more subscribers. However, it may communicate with brokers in the network.

Definition 2. *Let B_B be a set of brokers in a network. An edge broker $B_e \in B_B$ is a broker that directly communicates with at least one subscriber S. B_e may communicate with a broker $B_i \in B_B$ identified as a neighboring broker for B_e.*

3. Two brokers are "neighboring brokers" when they take part of a network of brokers and can directly intercommunicate without the need of passing by a third broker.

   Definition 3. *Let B_B be a set of brokers in a network. Let $B_i \in B_B$. Let B_i^N be a set of brokers identified as neighboring brokers for B_i, where $B_i^N \subset B_B$. A broker B_j is a neighboring broker for B_i if B_i can directly communicate with $B_j \in B_i^N$.*

   The symmetric (or commutative) property holds. Therefore, if B_j is a neighboring broker for B_i, B_i is a neighboring broker for B_j. However, the associative property does not hold. If B_i is a neighboring broker for B_j and B_k is another neighboring broker for B_j, this does not imply that B_i is a neighboring broker for B_k. One can see however that both B_i and $B_k \in B_j^N$, i.e. they are neighboring brokers for B_j.

   Figure 5.1 illustrates a placement example of one intermediate and two edge brokers in a network. Note that, in this figure, the edge broker *B1* is not a neighboring broker to *B3*, but both edge brokers are neighboring brokers for the intermediate broker *B2*.

4. A "local subscription" available in a broker B_i is an XPath subscription sent to B_i by a subscriber that directly communicates with B_i without passing by another broker.

5. A "remote subscription" available in a broker B_i is an XPath subscription sent to

B_i by a broker $B_j \in B_i^N$, providing that it has been originally issued by a subscriber that does not directly communicate with B_i.

6. Based on Definitions 1 and 2 as well as items 4 and 5, all subscriptions available in an intermediate broker B_i are considered "remote" to B_i, while subscriptions available in an edge broker B_j may be either "local" or "remote" to B_j.

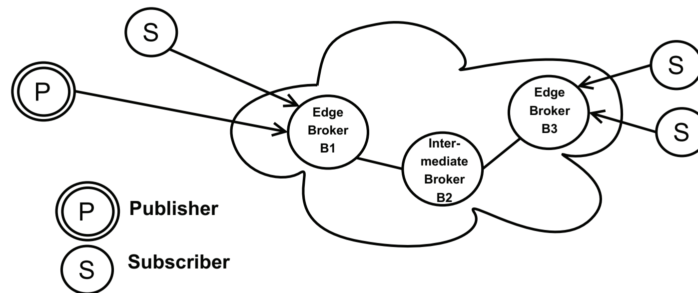

Figure 5.1: Edge and Intermediate Brokers.

5.2.2 Routing Tasks

When an X2CBBR performs the specific task of routing, it acts as a router. In a centralized content-based pub/sub system, such a broker has to perform all processing and matching tasks, but it does not need to route messages to another broker (see Section 2.1 and Figure 2.2). Details about some brokers' tasks follow.

- An interesting routing aspect that a broker may have is "content-based routing" [30], where messages are routed to other entities based on content rather than IP addresses. The messages can be notifications that have to be routed to subscribers based on subscriptions' content. This task takes place as a result of processing and matching subscriptions' content against publishers' content. An intermediate broker routes such notifications to one of its neighboring brokers. However, an

edge broker processes a local subscription and forwards the appropriate notification to the corresponding local subscriber (directly communicating with the broker). Moreover, an edge broker may need to route notifications to another broker as a result of matching a remote subscription.

- When all filters of a subscription do not match any available published content, a broker may have to forward the subscription to another broker according to some content routing criteria. Afterwards, the broker may have to either remove the subscription from its buffer or keep it for some purposes.
- When an intermediate broker receives a notification (which is ready to be delivered to a subscriber) from an up-stream broker, it must forward the notification to a down-stream broker selected according to some content routing criteria. The latter broker forwards the notification to one of its neighboring brokers or to the intended local subscriber. The forwarding tasks may have to take place with or without processing under different conditions.

The X2CBBR employs a few algorithms in order to efficiently perform the aforementioned tasks, in addition to other tasks detailed in the following sections.

5.3 The Hardware Broker Architecture

The X2CBBR architecture consists of three main units. Each of these units can communicate with external entities through two dedicated interfaces. Thus, the broker is able to handle a total of six interfaces that may be active concurrently. The broker architecture, depicted in Figure 5.2, consists of the following three main functional parts: (Part A) a Subscription Processing Unit (SPU), (Part B) an XML/XPath Processing and Matching

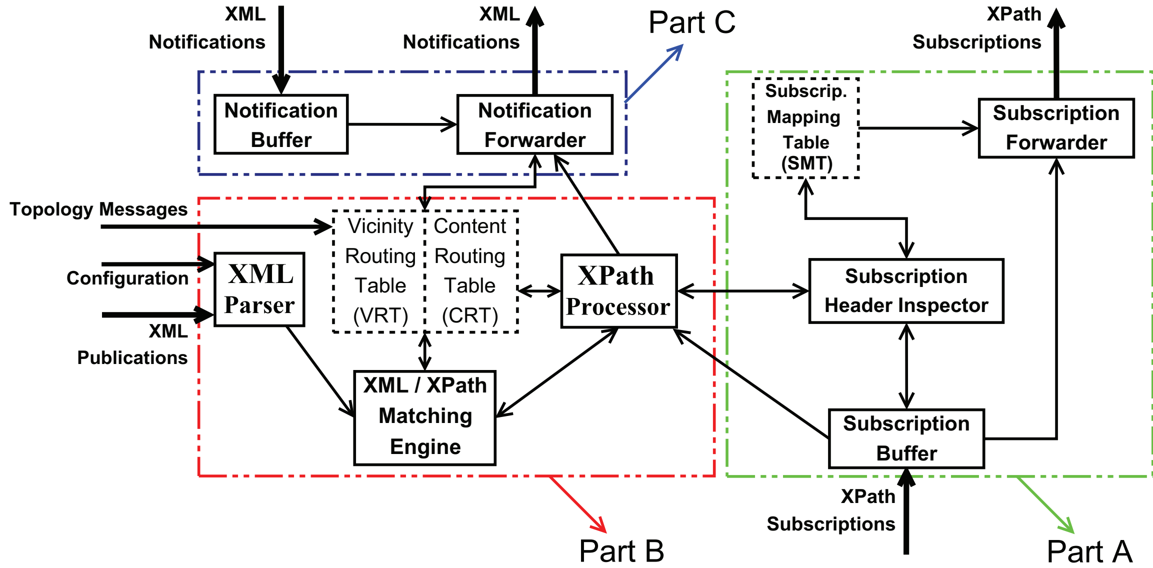

Figure 5.2: Block Architecture of the Reconfigurable Hardware Broker.

Unit (PMU), and (Part C) a Notification Processing Unit (NPU). The SPU may receive subscriptions through one interface and route out subscriptions through its other interface. The PMU may receive publications through one interface and topology messages through its other interface. The NPU may receive notifications through one interface and route out notifications through its other interface. Before discussing these units in detail, we state next a basic overview of the architecture including definitions, data flow, and some concurrency features.

5.3.1 Basic Overview

A user would send a subscription S to the SPU of the broker, via the SPU interface, as an XPath expression that may contain one or more filters *f*. For example, the XPath subscription "/stock/stockitem/price" contains three filters $f_1 = \texttt{stock}$, $f_2 = \texttt{stockitem}$, and $f_3 = \texttt{price}$ that aim to match an XML publication. Thus, the action of subscribing to the system with this subscription (S) is formulated as *sub(S,F)* where $F = \{f_1, f_2, f_3\}$.

In general, a subscription's filter set F consisting of n filters takes the form of $F = \{f_i\}$, where i = *1,2,3,.....n*. The maximum value of n must be limited (Section 5.3.3.4).

According to the XPath documentation [120], an XPath expression that matches some XML content may return a node value, a string, a number, or a boolean value. Therefore, the example of the XPath expression "/stock/stockitem/price" may return the atomic value of the node "price". Moreover, an XPath expression starting with a single forward slash selects the root node of an XML document, while a double forward slash "//" has the power of returning any atomic values of nodes that match the selection starting from the current node. For example, the expression "//price" returns all atomic values of "price" in an XML document starting from the current node.

However, the X2CBBR processes a very limited subset of XPath for simplicity and for keeping the XPath expressions in the context of subscriptions in pub/sub systems. Moreover, it treats the XPath expression in a specific way that may differ from the XPath documentation. The following **r**ules must be considered when users subscribe to the broker:

- The broker does not accept an XPath expression unless it is embedded in a subscription. The subscription must adhere to a specific format discussed later in this chapter. In this thesis, the two terms "subscription" and "XPath subscription" can be used interchangeably to express the specific format that the broker accepts.
- If a subscription that embeds an XPath expression matches some XML publication content, the returned data will not be just normal atomic values. Instead, an XML notification that includes the selected nodes and their values will be delivered to the user that has issued the corresponding subscription.
- The XPath expression of a subscription must describe a simple path with either no

operators or a few possible operators. The accepted operators are "$=, \neq, >, \geq, <, \leq$". All other operators as well as the wildcard symbol "*" cannot be recognized.

- The broker treats a single forward slash as the starting of any node even if it is not the root node. Moreover, the broker does not accept a double forward slash. Therefore, the subscription having the expression "/price" is sufficient to match the XML node "price" anywhere in the XML publication.

- The broker does not recognize built-in XPath functions such as node(), position(), last(), count(), name(), string(), and text().

In addition to the SPU interface, the architecture of the broker provides another dedicated interface for the reception of publications, so that concurrent reception of subscriptions and publications can be possible. A publisher would send an XML publication P to the XML parser of the PMU with an action formally described as *pub(P)*. P is said to match the subscription S if P contains all filters $f_i \in F$ where F is the filter set of S, i.e. if $F \subset P$ or more precisely if $F(S) \subset P$.

As a result of successful matching, the system sends a notification N to the corresponding subscriber, with an action described as *notify(S,N)*. This action reads that a notification N is to be delivered to the user that had subscribed with a subscription S. The broker may accomplish the notification process through an independent interface concurrently with the reception of publications and subscriptions.

The flow diagrams of Figure 5.3 allow for visualizing both the sequential and concurrent tasks, assuming - for simplicity - that only one XML publication is available. The buffering process of new incoming subscriptions has its sequential flow diagram, but it runs in parallel with the registration process with some latency. This latency, illustrated in the diagram by the "wait" function, allows the registration sequence of

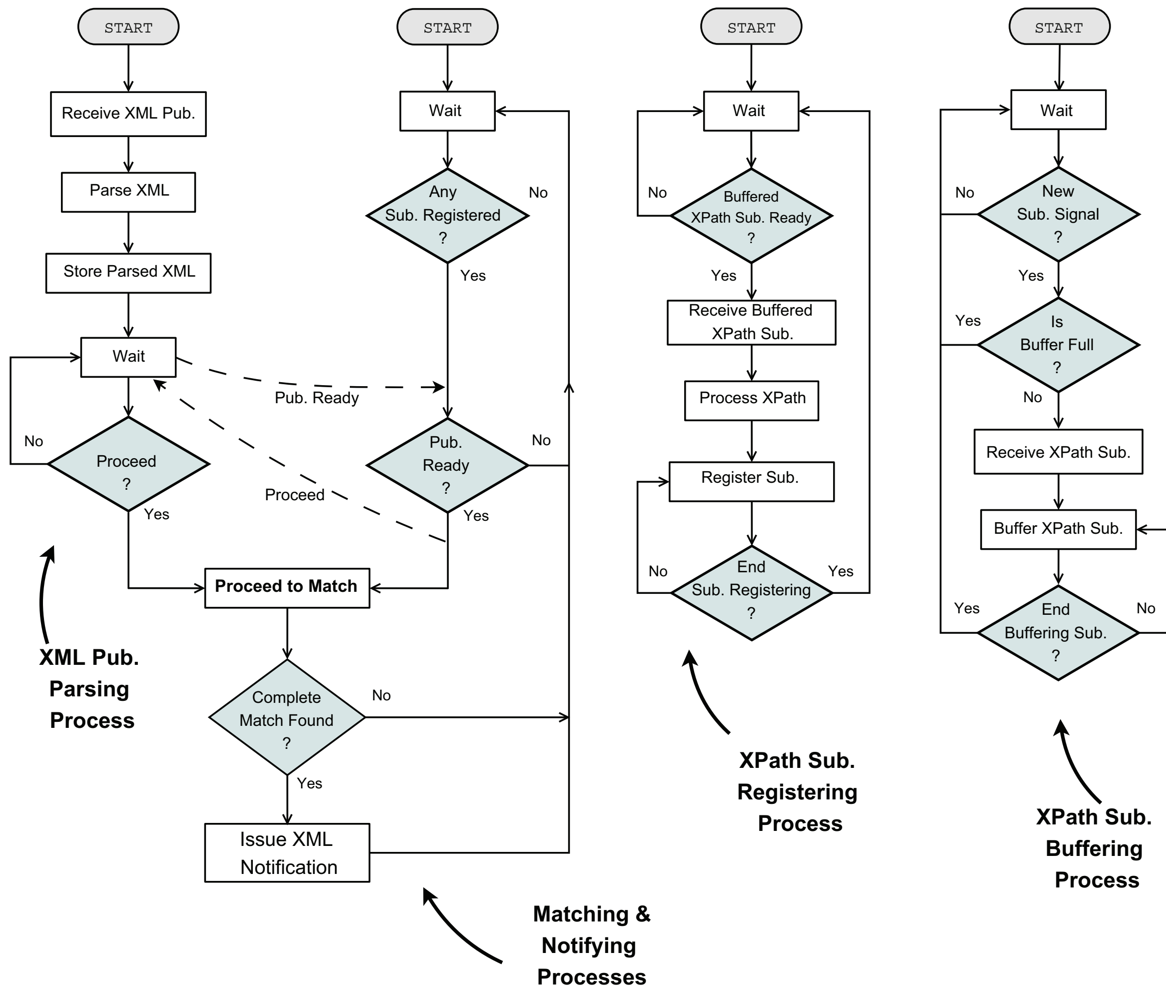

Figure 5.3: Simple Flow Diagram Illustrating the Sequence of Possible Processing Functions of the Broker in the Case that only One XML Publication is Available.

buffered subscriptions to take place while new incoming subscriptions continue to get in the buffer. An XML publication can receive its processing and parsing share in parallel with XPath subscription buffering and registration tasks. Once the parsed publication becomes ready, it signals its presence, hence the matching process takes place prior to the notification task.

Note that, for the sake of simplicity, Figure 5.3 does not show neither the routing

tasks of subscriptions, publications, and notifications nor the routing tables and the corresponding memory resources. The detailed description of all units of the broker internal architecture and their functional and routing tasks are the subject of next sections.

5.3.2 Part A: Subscription Processing Unit (SPU)

The broker processes new subscriptions in the following consecutive three stages: buffering, mapping, and registering. While the buffering and mapping stages occur in the SPU, the registering stage takes place in the PMU and will be discussed in Section 5.3.3. Figure 5.4 shows how these stages would intercommunicate.

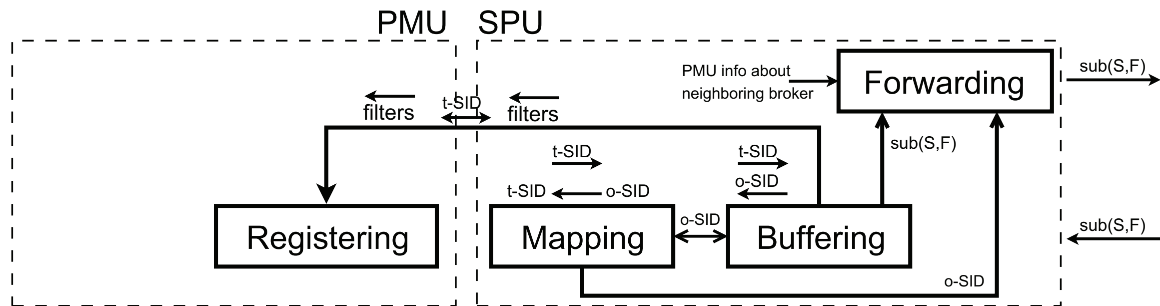

Figure 5.4: Processing Stages of a Subscription.

The SPU may also run a "forwarding process" on old subscriptions (i.e. subscriptions that have been already registered in the PMU). However, the forwarding stage may only occur as a result of the matching process executed in the PMU.

5.3.2.1 Buffering

The subscription buffering stage consists of the "Subscription Buffer" and the "Subscription Header Inspector" shown in Figure 5.2.

The "Subscription Buffer" has undergone two phases of development. In the first phase, the buffer consists of a single dual-port lightweight SRAM (32 bits x 256 words

module) that may contain up to 1 KB of raw XPath subscriptions, as shown in Figure 5.5.

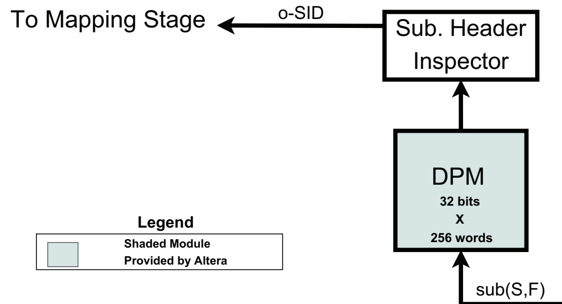

Figure 5.5: Simple Subscription Buffer.

This simple buffer works well when forwarding of raw subscriptions does not take place, since there is no need to access the buffered subscriptions after the registration process. Once registered in the PMU, the buffered subscriptions can be overwritten with new ones without any subscription data loss.

In the second phase, the "Subscription Buffer" comprises two dual-port SRAM modules, named DPM1 and DPM2, as illustrated in Figure 5.6.

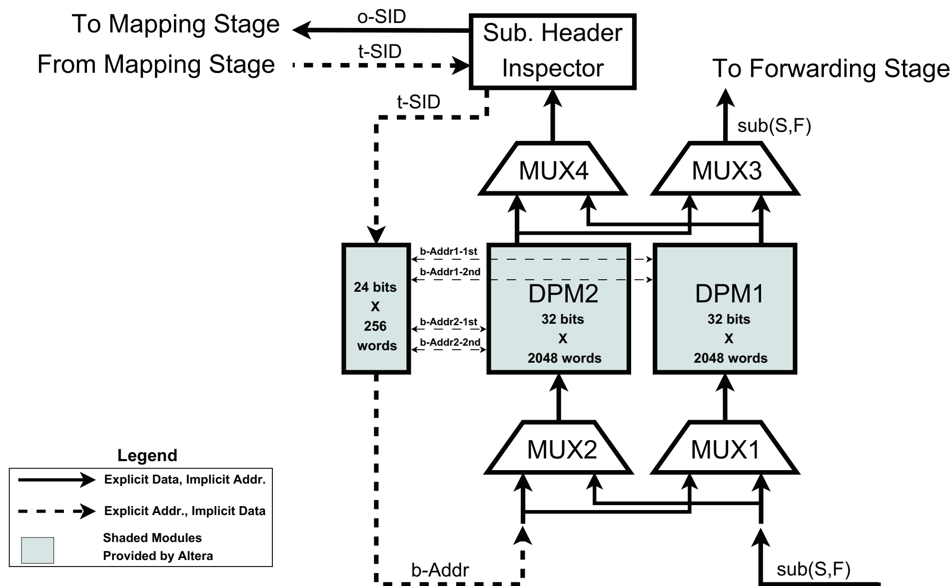

Figure 5.6: Double Subscription Buffer.

The broker loads raw (unprocessed) XPath subscriptions into either DPM1 or DPM2

but not into both at the same time. The subscriptions buffered into, for example, DPM1 will remain in it until a match decision is made and/or until these subscriptions are forwarded (routed) via the “Subscription Forwarder” to other entities. Meanwhile, the broker is able to buffer new incoming subscriptions into DPM2 without waiting for the forwarder to finish its routing task with regard of the old subscriptions that were stored in DPM1. While the subscriptions buffered in DPM2 are being processed and forwarded, the broker can load DPM1 with newly-arrived subscriptions (providing that old subscriptions are no longer needed), and so on. The selection between which memory module is to be loaded with raw subscriptions, and which one is to be accessed for forwarding purposes, is made with multiplexers (MUX1, MUX2, MUX3, and MUX4) as shown in Figure 5.6. Note that either DPM1 or DPM2 is a 32 bits x 2048 words memory module that may contain up to 8 KB of XPath subscriptions.

While the memory (either DPM1 or DPM2) stores subsequent subscriptions via one port, the “Subscription Header Inspector” reads stored subscriptions via the other port, and extracts their header information. The simultaneous writing and reading accesses allow the inspection process to run without waiting for the end of the buffering process for all raw subscriptions.

The header of each subscription consists of a data overhead that can be small or large, depending on the size of the brokers’ network and the number of potential subscribers. The subscriber first constructs the header and attaches it to the subscription’s filters, and then sends the whole subscription to a broker. In the X2CBBR present architecture, the data overhead for each subscription (Figure 5.7) is arbitrarily assumed to be 20 bytes (160 bits), including 1 to 8 bytes (8 to 64 bits) of identification data. Originally, the identification data identifies the subscriber who issued the subscription. However, a broker may later modify this data. More information about the identification data is

Figure 5.7: Structure of an XPath Subscription.

discussed in Section 5.3.2.2 and in Section 5.4.3.5.1.

The inspector passes the identification data to the mapping stage and waits for a mapped 8-bit value. Once this value is received, the inspector can use it as an address (index) to access one row of a small dual-port SRAM (24 bits X 256 words). At this specific row of the memory, the inspector stores the first and last buffer addresses (each is 8-bit wide) of the corresponding raw subscription that resides in either DPM1 or DPM2 (b-Addr1-1st and b-Addr1-2nd for DPM1, and b-Addr2-1st and b-Addr2-2nd for DPM2, in Figure 5.6). These two buffer addresses will later be used by the "Subscription Forwarder" to retrieve buffered raw subscriptions and forward them to an external entity (typically a neighboring broker). The benefit of designing the "24 bits X 256 words" memory as a dual-port SRAM is that it allows both the "Subscription Forwarder" and the inspector to access the memory concurrently. The forwarder accesses it for reading while the inspector accesses it for writing.

For example, the buffer stores a raw subscription S in addresses starting at address 1 and ending at address 10 in either DPM1 or DPM2. The inspector sends the identification data of S, say "0xABCDABCD", to the mapping stage, and then it receives a mapped value, say "20". Subsequently, the inspector utilizes the value "20" (0x14) as an address to access the small dual-port memory, and in this location stores the following 16-bit {0x01,0x0a} representing the first and last buffer addresses 1 and 10 respectively. Later, the "Subscription Forwarder" accesses the small dual-port memory via address "20",

retrieves the values of buffer addresses 1 and 10, utilizes address 1 to start reading and forwarding the corresponding raw subscription, and stops accessing the buffer after it reads and forwards the last data located in address 10.

5.3.2.2 Mapping

The "Subscription Header Inspector" extracts identification data, called original Subscription ID (o-SID), from the header information of each subscription (see Figure 5.7), and stores such data in the Subscription Mapping Table (SMT) (located in Part A of Figure 5.2). The o-SID size may vary from 1 to 8 bytes (8 to 64 bits), while the remaining header data is actually used in the determination of the beginning and the end of the header. This specific distribution of header data can change when modifications of the width of SMT is possible. For example, the o-SID may become 12 bytes or 16 bytes out of the 20-byte header. However, the current broker architecture employs a small-sized dual-port SRAM (64 bits x 256 words) for the SMT, i.e. with a maximum data width of only 8 bytes. When the broker does not forward the subscription to any external entity, the o-SID stays without modification during the lifetime of the subscription. In the case that a broker has to forward the subscription to another broker, this 8-byte o-SID will be updated so as to include the broker identification data. More information about the o-SID update is discussed in Section 5.4.3.5.1.

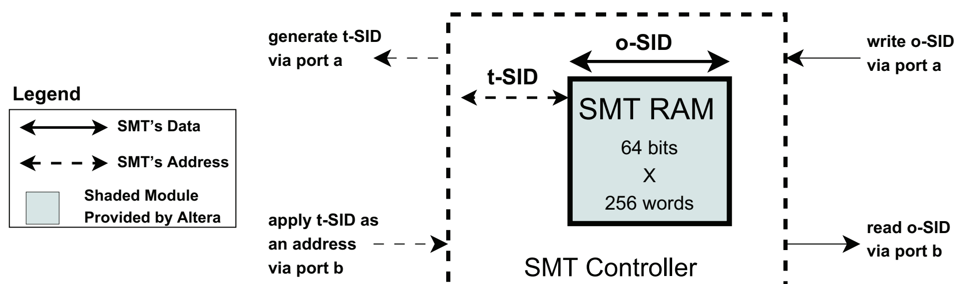

Figure 5.8: Structure of the Subscription Mapping Table.

The SMT actually maps the o-SID of each subscription to an 8-bit temporary Subscription ID (t-SID) for efficient registration and processing. The t-SID physically represents the 8-bit address of the SMT memory that stores the corresponding o-SID. Figure 5.8 shows the SMT structure. Since the t-SID must be a non-null value, the address 0x0 is not used. Therefore, the size of the SMT allows a maximum of 255 (rather than 256) local or remote subscriptions that can simultaneously be under process by the broker (Note that the feature of simultaneous processing of subscriptions is discussed in Section 5.3.3). Subsequently, the broker sends the t-SID to the XPath processor of the PMU for further processing.

The SMT clears or overwrites its entries once appropriate notifications and/or corresponding subscriptions are forwarded. Overwriting the SMT with new 255 o-SIDs of subscriptions takes place promptly, since the inspector can readily obtain the o-SIDs of the corresponding raw subscriptions that are already buffered in either DPM1 or DPM2. The benefit of designing the SMT as a dual-port SRAM is that it allows both the "Subscription Forwarder" and the inspector to access the mapping table concurrently.

5.3.2.3 Forwarding

The "Subscription Forwarder" communicates with the "Subscription Buffer" to forward raw subscriptions as described in the discussion of the buffering stage (Section 5.3.2.1). However, not every subscription (being local or remote) must be forwarded to a neighboring broker. As a result of a successfully matched subscription, the broker issues and delivers (or routes) the corresponding notification (later discussed in Section 5.3.4) without the need of forwarding such a subscription to anywhere, except for the purpose of sending the subscription to another "scope" of the network (Section 5.4.6).

Basically, the SPU forwards a subscription that has not matched any of the available

publications (thus, no notification has been issued), for the purpose that such a subscription gets the opportunity of finding matched publications that may be available in other brokers in the network.

Forwarding a raw subscription implies that its format depicted in Figure 5.7 is preserved. Therefore, the SPU forwards the subscription S in its original form *sub(S,F)* (Figure 5.6). However, the identification data in the header may change in order to include data that identifies the forwarding broker (see Section 5.4.3.5.1). The exact neighboring destination of the forwarded subscription is based on content-based routing algorithms that make use of the Content Routing Table (CRT) and the Vicinity Routing Table (VRT) (located in Part B of Figure 5.2 and later discussed in Section 5.4). Detailed information regarding subscription routing in particular can be read in Section 5.4.3.6.

Based on the aforementioned description and on the discussion of the forwarding stage in Section 5.3.2.1, we depict Figure 5.9 that illustrates the corresponding FSM (more detailed hardware algorithm for subscription routing is shown in Figure 5.24). In Figure 5.9, the transition of the FSM to its "EvalSu" state from the initial state does not take place until the "reset" signal is released and the set of all mismatched subscriptions S_u has become complete (i.e. all mismatched subscriptions have been identified and no more subscriptions would match any available publications). Once the t-SID of a mismatched subscription is retrieved, the FSM steps to its "GrabAdr" to pick up the corresponding subscription buffer addresses as described in Section 5.3.2.1. These addresses are to be used in the "GrabSub" state to retrieve the buffered subscription. Then, in the "AddBID" state, the subscription forwarder adds the identification data of the current broker to the o-SID of the subscription, and grabs the identification data of the neighboring broker to which the subscription is to be forwarded. Finally, the subscription is forwarded to the identified neighboring broker in the "FRWD" state,

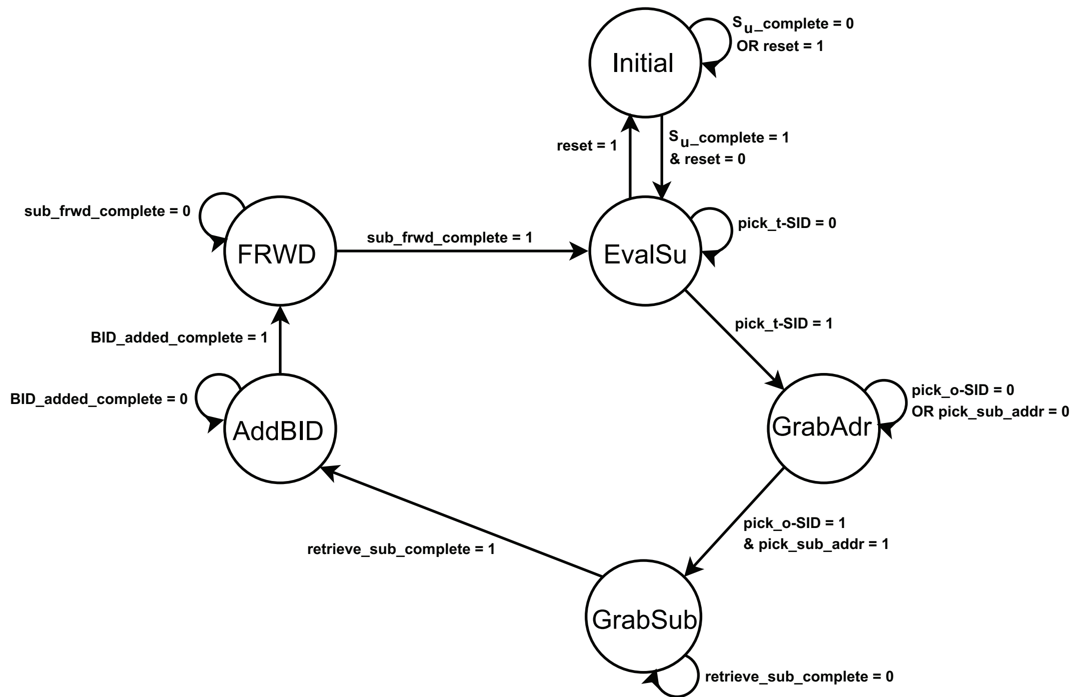

Figure 5.9: The FSM of the Subscription Forwarder.

before the FSM steps again to the "EvalSu" state in order to retrieve the t-SID of the next mismatched subscription, and so on.

5.3.3 Part B: XML/XPath Processing and Matching Unit (PMU)

The PMU architecture mainly consists of an XPath processor, an XML parser, and a matching engine (Part B in Figure 5.2).

5.3.3.1 Functional Overview

The XPath processor of the PMU has the main role of registering subscriptions. The registration process aims to systematically store the filters and the t-SID of each subscription in a table called Content Routing Table (CRT). Figure 5.10 shows the XPath processor architecture and illustrates its communication with the Subscription Buffer of

the SPU.

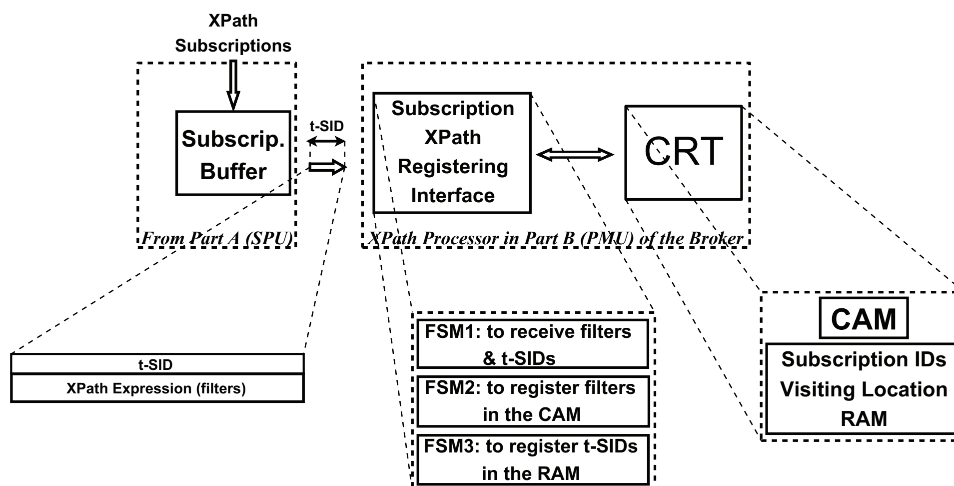

Figure 5.10: Hardware XPath Processor Architecture.

Within the CRT, a CAM stores the filters of XPath subscriptions, while a RAM (Subscription IDs Visiting Location Memory) stores corresponding t-SIDs. From a functional point of view, one can consider the CRT a recorder of all t-SIDs that have "visited" each registered filter. For example, consider two subscriptions s_1 and s_2, where each subscription has its own t-SID and filters as follows: $s_1 = \{$*t-SID1, (*f_1, f_2*)*$\}$ and $s_2 = \{$*t-SID2, (*f_1, f_3*)*$\}$. The CRT records *t-SID1* and *t-SID2* as visitors of the filter f_1, *t-SID1* as visitor of the filter f_2, and *t-SID2* as visitor of the filter f_3. Therefore, the number of CRT entries equals "at least" the number of registered filters rather than the number of registered subscriptions. Functionally, for these two subscriptions, there are three CRT entries: $\{f_1$*: t-SID1, t-SID2*$\}$, $\{f_2$*: t-SID1*$\}$, and $\{f_3$*: t-SID2*$\}$.

The XPath processor controls its main functions using the finite state machines FSM1, FSM2, and FSM3 depicted in Figure 5.10. These machines resides in the "Subscription XPath Registering Interface" to control the reception of subscription filters and corresponding t-SIDs, register the received filters in the CAM of the CRT, and register the corresponding t-SIDs in the RAM of the CRT.

Concurrently with XPath processing, the SCBXP parses the available published XML messages. The matching engine (Figure 5.2) can communicate with both the XPath processor and the XML parser. This engine attempts to find a match for each 32-bit parsed XML data among subscription filters located in the CAM. Once some publication content matches all filters of one or more subscriptions, the matching engine triggers a "match" decision for such subscriptions. Accordingly, the corresponding publication turns into a notification and is forwarded to the corresponding subscriber(s) through the Notification Forwarder component (located in Part C of Figure 5.2).

In the case that any of the filters of a registered subscription does not match any of the available XML content, the matching engine remains silent. Therefore, the absence of any "match" decision with regard of a particular subscription does not lead to a corresponding notification. Instead, the broker forwards the subscription to another broker via the "Subscription Forwarder" of the SPU.

5.3.3.2 CAM Structure

The CAM of the CRT stores the filters of subscriptions in a specific structured way so that identical filters found in multiple subscriptions can be stored only once.

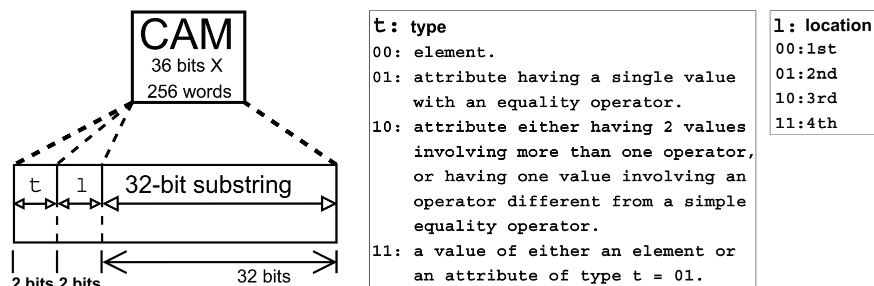

Figure 5.11: CAM Internal Structure.

Figure 5.11 shows the internal structure of CAM rows. Each CAM row hosts 36-bit of information that includes a 32-bit string representing a filter, or part of a filter, that may

exist in multiple subscriptions. The remaining 4-bit information represents two fields as follows:

- A 2-bit parameter l indicates the location of each 32-bit data within a filter. For example, the filter "stock" is to be stored in two CAM rows because the filter is wider than 32 bits. One row contains the value l = "00" (in the 2-bit location field) and the string "stoc" in the 32-bit field. The l value of "00" means that the string represents the first 32-bit of the filter. The other row contains the value l = "01" and the string "k". The l value of "01" indicates that the string represents the second 32-bit of the filter. Since l may have up to 4 different values, a simple filter can be up to 128-bit long (16 ASCII characters) stored in 4 CAM rows.

- Another 2-bit parameter t indicates the type of a string within a filter. The type identifies whether the stored string is an element with t = "00", a simple attribute with t = "01", a complex attribute with t = "10", or a simple value with t = "11".

A simple attribute may have an exact single value, while a complex attribute may have either two values involving more than one equality operator or a value involving an operator different from a simple equality operator. For example, the filter *stockid="34"* consists of a simple attribute that has to be stored in two CAM rows, and a simple value that has to be stored in one row. The first row contains t = "01" indicating an attribute, l = "00" indicating that the string resides in the first 32-bit of the filter, and the string "stoc"; the second row contains t = "01" for an attribute, l = "01" for the location in the second 32-bit of the filter, and the string "kid"; and the third row contains t = "11" indicating a value, l = "00", and the string "34".

As an example of a complex attribute, the filter *stockid* $\geq$ *"35"* can be stored in four rows. The first row contains t = "10" indicating a complex attribute, l = "00", and the

string "stoc"; the second row contains t = "10", l = "01", and the string "kid"; the third row contains t = "11" indicating a value, l = "00", and the operator $\geq$; and the fourth row contains t = "11", l = "01", and the string "35". Note that the operator $\geq$ and the value "35" are considered one value, where l = "00" for the operator and l = "01" for the next value. This specific structural design instructs the matching engine not to treat the value "35" as a simple value.

From the structure of the CAM, we can see that when a string exists in multiple subscriptions, this string will be stored only once if it has the same type t and the same location l in all these subscriptions (Figure 5.12-a).

For example, the filter "stock" is of type element (t = "00") in the four subscriptions of Figure 5.12-a, but it is stored only once in the CAM. Likewise, the filter "stockid" is of type attribute (t = "01") in the four subscriptions, and it is stored only once as well. The value of the attribute is unique for each of the four subscriptions. Therefore, each of the four values must reside in the CAM.

5.3.3.3 Subscription IDs Visiting Location Memory Structure

A subscription s, represented by its t-SID may contain one or more filters f. The subscription s is called to have visited a CAM address (*caddr*) if any of its filters $f \leq 32$ bits, or a 32-bit string in a filter $f \in s$, is stored in that *caddr*. Identical 32-bit strings occupy one common location in the CAM (if they have identical l and t). Each address *caddr* of the CAM is to be used as an index to a location *raddr* in a RAM, i.e. $caddr = raddr$. The contents of this RAM address *raddr* consist of t-SIDs of subscriptions that have visited the identical CAM address *caddr*. This RAM consists of 256 rows, where each row is 256-bit wide and has the structure illustrated in Figure 5.13.

An example of four subscriptions containing common and unique filters can be sys-

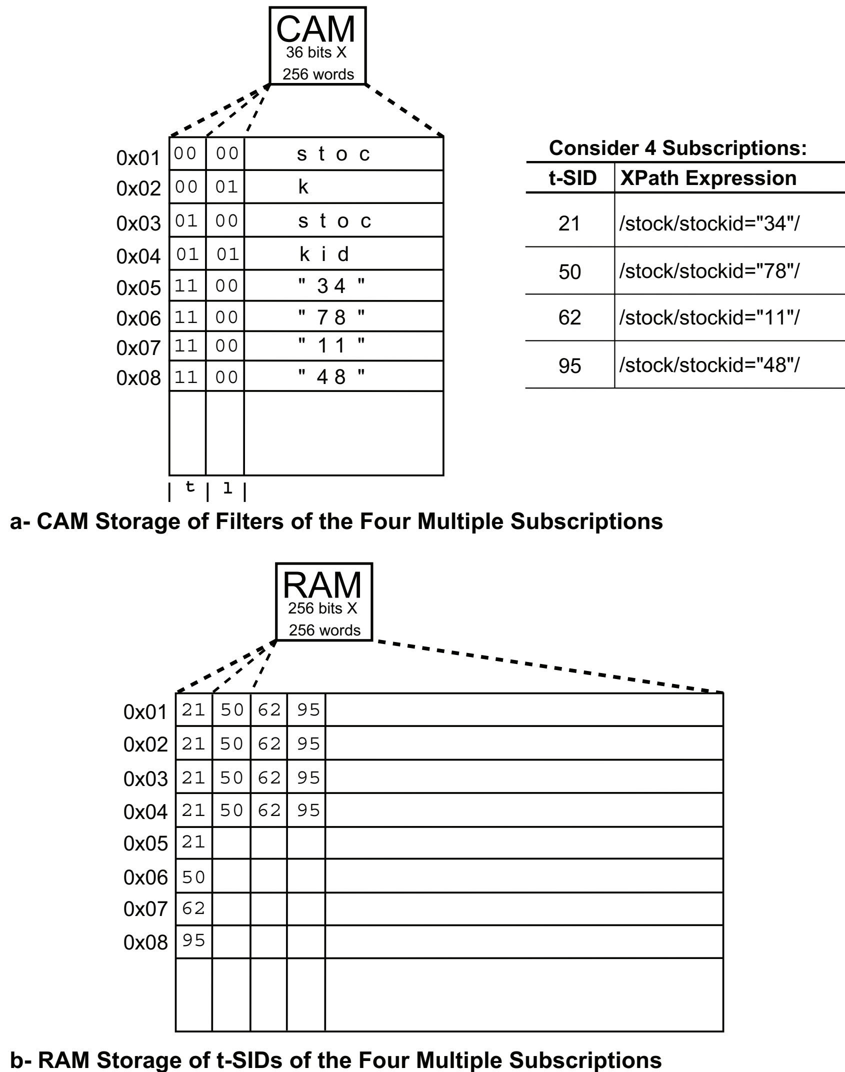

Figure 5.12: Subscription Registration in the CRT: Example of Four Subscriptions Comprising Common and Unique Filters.

tematically registered in the CRT as depicted in Figure 5.12, where Figure 5.12-b shows how the Subscription IDs Visiting Location RAM stores the t-SIDs of the four subscriptions for each filter registered in the CAM depicted in Figure 5.12-a. Since four 32-bit filters are common among the four subscriptions (thus, they are stored only once), and an additional 32-bit filter is unique for each subscription, the total number of registered 32-bit filters must be 8 rather than 20, which is the total number of 32-bit filters existing

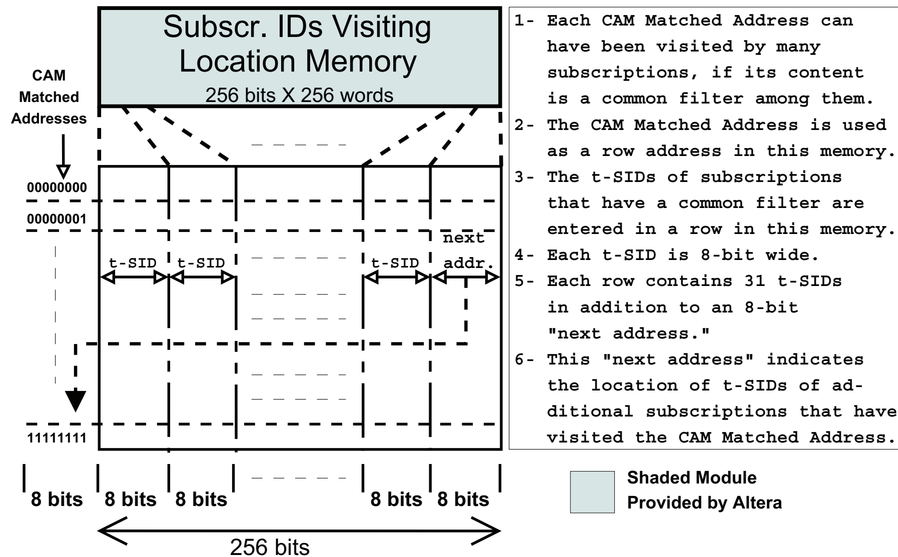

Figure 5.13: "Subscription IDs Visiting Location Memory" Structure.

in the four raw subscriptions. This exploitation of commonalities provides efficient CRT storage.

The total number of the RAM entries must equal the total number of CAM entries when no more than 31 subscriptions contain a single common filter. For example, the CAM only needs one entry, and the RAM only needs one entry as well, if we assume that only 31 subscriptions are available and ready to be registered in the broker, and if each subscription contains the same 32-bit filter *f1* and only this filter *f1*. In this simple example, the CAM stores *f1* in one of its rows (e.g. at $caddr$ = 0xAB), and the RAM stores in one of its rows (of address $raddr$ = 0xAB) the t-SIDs of all 31 subscriptions. Sine each t-SID is 8-bit wide, and the row can contain 31 t-SIDs, the RAM row can hold 8 x 31 = 248 bits of t-SIDs.

In the case that more than 31 subscriptions have a common filter, an additional RAM entry (chosen from the last addresses of the memory as shown in Figure 5.13) is needed where up to additional 31 t-SIDs can reside. In such a case, the last 8-bit of the original

RAM entry points to the address of the additional RAM entry. In other words, the row content of each *raddr* has a 256-bit data structure that consists of a maximum of 31 t-SIDs, in addition to one 8-bit “next address” value. If more than 31 subscriptions have visited a *caddr*, the value of the 8-bit next address indicates the next *raddr*, in which additional 31 t-SIDs (and another “next address” value) can be stored.

Structuring the rows of this memory in the way shown in Figure 5.13 allows for simultaneous identification of all the subscriptions that have commonalities. In Figure 5.12-b, one can see that the RAM address, for example, $raddr$ = 0x01 contains the t-SIDs of the four subscriptions. Therefore, the filter residing in the CAM address $caddr$ = 0x01 is common among these four subscriptions.

5.3.3.4 Subscription Registration Algorithm (SReA)

Based on the aforementioned description of the subscription registration, the algorithm of Figure 5.14 summarizes the corresponding process functions.

In this algorithm, the three first lines summarizes the subscription buffering stage, where each subscription may contain a maximum of 16 32-bit filters - i.e. 16 * 32 = 512 bits of filter data. This restriction is solely made so as to keep enough room to many subscriptions. With no such restriction, a too long subscription would occupy most or all of the 256-word CAM of the CRT. Consequently, the broker would not have the opportunity of simultaneously processing many subscriptions.

For example, consider an overly long subscription s containing a high number of filters f_{cam} and f_{miss}, where $f_{cam} \subset s$ and $f_{cam} \subset CAM$, and $f_{miss} \subset s$ but $f_{miss} \not\subset CAM$ – i.e. the filters f_{miss} have not had the chance to be stored in the CAM. If at least one f_{cam} filter of s does not match any available publications, no match decision will be issues. Thus, the absence of a match decision reveals that there is no need for f_{miss}. This situation

Algorithm: Subscription Registration

```
1: Consider a received subscription s
2: Consider filters (F_i ≤ 32 bits) ∈ s where i = 1, 2, ....16
3: Extract o-SID from s
4: Free SMT entry @ SMT address saddr ← o-SID (i.e. M[saddr] ← o-SID)
5: generate t-SID ≡ saddr
6: decompose s into F_i
7: if i = 0 or i > 16 {case of either no filters or too many filters in s} then
8:     clear s, F_i, t-SID, o-SID
9: else
10:    for all F_i do
11:        Match F_i against CAM entries of CRT (i.e. Search F_i in CAM)
12:        if F_i ⊄ CAM then
13:            Free CAM entry of CRT ← F_i (i.e. M[Free caddr] ← F_i)
14:            Get corresponding CAM address of CRT (caddr)
15:        else
16:            Get matched CAM address of CRT (caddr)
17:        end if
18:        RAM address of CRT raddr ≡ CAM address of CRT (caddr)
19:        Reading Register RR ← M[raddr]
20:        if RR is full then
21:            Report High Commonality Degree (HCD)
22:            for Full RR do
23:                Grab {next_address} from Reading Register RR
24:                Reading Register RR ← M[next_address]
25:            end for
26:            Append {RR, t-SID}
27:            M[next_address] ← appended {RR, t-SID}
28:        else
29:            Append {RR, t-SID}
30:            M[raddr] ← appended {RR, t-SID}
31:        end if
32:    end for
33: end if
```

Figure 5.14: Subscription Registering Algorithm (SReA).

is acceptable. However, if all filters f_{cam} match an available publication, the "match" decision occurs and the corresponding notification is issued, even though the non-stored filters f_{miss} may or may not match any publication content. This situation is an example of a potential erroneous result.

The third line of the algorithm identifies the task of the inspector that consists of extracting the o-SID of a subscription and passing it on to the mapping stage.

The fourth and fifth lines are about the generation of the t-SID of a subscription that happens in the mapping stage. The sixth line indicates the process of picking up the filters from the subscription. The lines 7 to 8 relate to the restriction of the maximum

number of filter data mentioned above, and instruct the broker to clear any subscriptions that either have no filter data or exceed the maximum allowed number of 32-bit filter data.

The remaining lines (the "for" loop from line 10 to line 32) describe the registration stage in the CAM and RAM of the CRT. Note that in the case of a full row is encountered in the RAM (see Section 5.3.3.3) the "next address" value is to be used to register the t-SID of a subscription. In such a case, the algorithm reports a High Commonality Degree (HCD) (line 21) to the SPU. This value instructs the Subscription Forwarder of the SPU to restrict its forwarding task if high mismatched subscriptions have occurred. Therefore, if many mismatched subscriptions have been identified and the HCD is reported, the "EvalSu" state of the Subscription Forwarder's FSM (Section 5.3.2.3) acknowledges the potential high degree of commonalities among these mismatched subscriptions. Accordingly, only a single subscription is forwarded instead of all mismatched subscriptions.

Referring to lines 13 and 14 of Figure 5.14, we can notice that the absence of a filter F_i in the CAM triggers a writing access to the CAM in order to register this filter. However, the existence of such a filter in the CAM does not trigger a CAM writing access (line 16). Therefore, the more commonalities that exist among subscriptions, the less registration time is needed.

5.3.3.5 XML Parser

The broker utilizes a stand-alone hardware-based XML parser, to parse XML publications and feed the matching module interface with 32-bit parsed data. This XML parser, shown in Figure 4.1, implements the proposed hardware-based SCBXP technique (see Chapter 4 for details). If the parser were software-based, the broker would become bound by the

performance of the parser. Therefore, by using a hardware-based XML parser, we avoid unexpected degeneration in the overall performance. Note that the SCBXP is capable to parse at least two XML data octets in one clock cycle (see Chapter 6). The parser stores the parsed data in its own dual-port memory using one port, while the other port of the memory can be accessed by the PMU's matching engine (Figure 5.2). Therefore, this matching engine can pick up the parsed data without interfering with the parsing process, and search for data in the CRT's CAM that matches the parsed data. This feature of concurrency has a positive impact on the processing performance.

5.3.3.6 Matching Engine

The Matching Engine (Figure 5.15) needs to search the CAM of the CRT for two purposes: (1) to verify whether a 32-bit filter is already in the CAM during the subscription registration (lines 10 to 16 in Figure 5.14) and (2) to verify whether a 32-bit string (parsed from an XML publication) matches any 32-bit XPath filter existing in the CAM. In the subscription registration stage, matching of new subscription filters against old filters takes place. In the post-registration stage, the matching process runs for parsed XML publication data against stored subscription filters.

During the subscription registration, the non-existence of a 32-bit filter in the CAM leads to writing this filter along with its parameters l and t in the CAM, and writing the corresponding t-SID in the RAM of the CRT (lines 17 to 20 in Figure 5.14). However, the existence of that filter in the CAM leads to only writing the corresponding t-SID in the RAM of the CRT. A storing counter for a specific t-SID increments with each 32-bit registered filter. The final value of a storing counter indicates the number of 32-bit filters registered in the CRT for a t-SID.

Providing that subscription filters are already registered in the CRT, the Matching

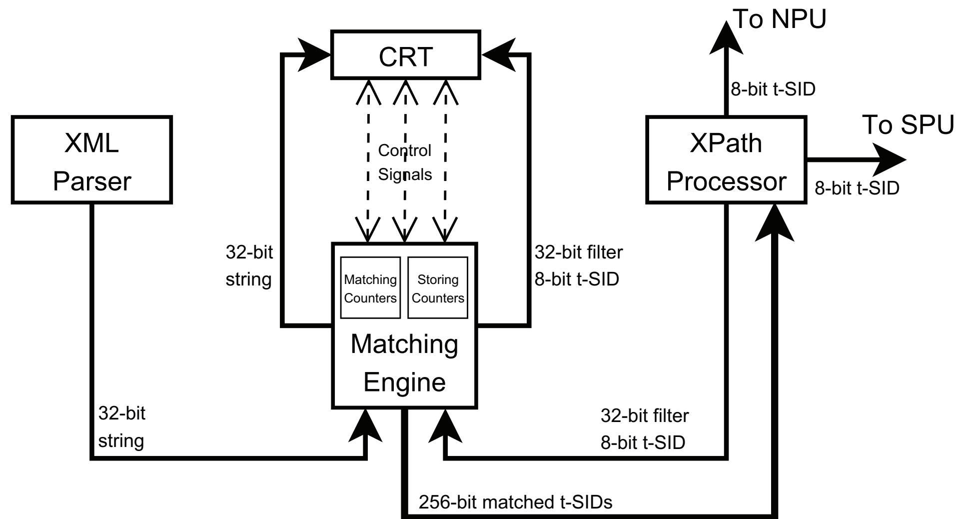

Figure 5.15: XML/XPath Matching Engine.

Engine takes 32-bit data at a time out of the XML parser, and searches for an identical string in the CAM taking into account the string's parameter l. In this search attempt, the engine masks out the string's parameter t, so that the CAM can return any type of identical strings. In the case that the searched string is found in the CAM, the returned string is recognized as a filter (or part of a filter) belonging to one or more XPath subscriptions. The CAM address *caddr* pointing to the found filter is utilized as an address *raddr* to the RAM to retrieve all t-SIDs identifying the subscriptions that contain such filter. A matching counter for each of these retrieved t-SIDs increments accordingly. The final value of a matching counter indicates the number of times a specific t-SID is retrieved as a result of a successful filter match.

At the end of the matching process, the engine verifies the matching counter and the storing counter for each t-SID. In the case that the values of these two counters are equal for a particular t-SID, the matching engine interprets that all registered filters for this particular t-SID have matched an available XML publication. Thus, a successful matched subscription should trigger a corresponding notification through the NPU.

In the case that the searched string is not found in the CAM, no further processing actions are to take place unless the searched string is expected to be a complex value. The engine expects a complex value right after it identifies a matched complex attribute. In this special case, simple identical matching must not be attempted. Instead, a sort of comparison between the parsed XML string and the complex values registered in the CAM must take place. For example, the subscription's filters "book > 120" exist in the CAM as a complex attribute "book" with a parameter $t = (10)_b$ and a value "> 120". No further actions enter into effect if the string "book" is not among the parsed XML data. When this string is one of the parsed XML data, the search outcome reveals the identical string "book" in the CAM as a complex attribute. The engine expects the next parsed XML data to be a value for this attribute. Therefore, the engine compares the next parsed XML data with the registered value "> 120" so as to decide on the corresponding matching result.

Note that the registration process for each 32-bit filter may need to access the CAM of the CRT twice: one time for searching and another time for storing. However, the matching process in the post-registration task only needs to search the CAM for any stored filter. Since the two searching processes may occur in parallel, one to register new filters and the other to search for registered filters, a permission grant mechanism is implemented, so that the two matching processes cannot simultaneously search the CAM. This mechanism ensures that match decisions related to one process are not erroneously accounted for the other process.

5.3.4 Part C: Notification Processing Unit (NPU)

In Part C of the broker architecture depicted in Figure 5.2, resides the Notification Processing Unit (NPU). The Notification Forwarder of the NPU receives the t-SIDs of

subscriptions whose all filters are successfully matched. Subsequently, it utilizes these t-SIDs as addresses to retrieve the original 64-bit identification data stored in the SMT (Section 5.3.2).

The routing algorithm of Figure 5.16 describes the generation and delivery of a notification to a local subscriber.

Algorithm: Local Notification - Generation and Delivery

1: **final successful match for a t-SID of a subscription *S***
2: **address of SMT *saddr* ≡ t-SID**
3: **o-SID ← SMT Entry @ *saddr*** (**i.e. o-SID ←** M[**t-SID**])
4: **generate corresponding notification *N* as *notify(S, N)***
5: **corresponding subscriber ← *notify(S, N)***

Figure 5.16: Local Notification Routing Algorithm.

The algorithm starts from the point that all filters belonging to a specific t-SID have matched an available XML publication. The corresponding match decision occurs in the PMU as described in Section 5.3.3. In the second line of the algorithm, the SPU receives the corresponding t-SID from the PMU and utilizes it as an address to access the SMT. In the third line of the algorithm, the SMT returns the corresponding o-SID. Regarding this exchange of t-SID and o-SID, the XPath processor acts as the coordinator that interfaces the three parts of the broker architecture: SPU, PMU, and NPU (see Figure 5.2). In line 4 of the algorithm, the NPU receives the o-SID and the matched XML publication. Accordingly, the NPU generates an XML notification N derived from the matched publication, and puts it in the form *notify(S,N)*, i.e. in the form that includes the o-SID of the matched subscription S. In line 5, the NPU actually sends the notification.

In the case that the matched subscription is not local, the NPU needs to route the notification to a neighboring broker as discussed later in Section 5.4.

Another possible case occurs when the NPU receives notifications from an up-stream

broker for the purpose of re-routing them to a down-stream broker. The NPU buffers such notifications (in the Notification Buffer of Figure 5.2) and then routes them to another broker without in-depth processing. Such notification routing will be discussed as well in Section 5.4.

5.4 Content-Based Routing Tasks of the Broker

Section 5.3 has described the internal components of the X2CBBR architecture. The current section focuses on hardware content-based routing mechanisms that operate along with other hardware brokering tasks. But first, we point to the separation aspect between matching and routing that the architecture possesses.

5.4.1 Separation Between Matching and Routing

An interesting distinction in the hardware broker architecture is the provision of matching separately from routing, even though the routing decision is made based on the matching results. This distinction is a major difference between the X2CBBR architecture and existing architectures of a broker or a content-based router. Most of existing architectures make use of "identity-based routing" and "covering-based routing" (e.g. [78]). The X2CBBR architecture supports "identity-based matching" and "covering-based matching"; and then separate routing may occur. The next section further elaborates on the difference aspect of these terms depending on either routing or matching.

5.4.2 Identity-Based Matching versus Identity-Based Routing

While the "identity" term refers to the identical filters in both routing and matching, the terms "identity-based routing" and "identity-based matching" usually point to different

things.

The **identity-based routing** mechanism assumes that each filter has an individual destination, and if this filter has matched some publication data, the notification takes the route to this particular filter's destination. Therefore, the existence of multiple filters in a single subscription leads to the construction of a logical tree (that comprises physical brokers/routers) for this subscription, and makes a path through the tree destinations (where each destination is actually a tree node). Subsequently, the corresponding notification travels through the path down (or up) to the last filter's destination of the subscription (i.e. the last node in the tree). Therefore, the "identity-based routing" is content-based, matching-based, and tree-based. Accordingly, the matching and routing aspects are so tied that distinguishing one from the other is not obvious. This kind of routing has the benefit of avoiding the forwarding of multiple notifications for redundant subscriptions.

For example, a subscription having four filters a, b, c, and d and another subscription having identical four filters are redundant, so that a notification travelling up in the tree of the first subscription from node d to node a will pass through the same nodes of the tree of the second subscription, without the need for more than one notification delivery.

The mechanism of **covering-based routing** adheres to the same concept of identity-based routing, except that the filters of a subscription are a subset of filters in another subscription. Therefore, the latter subscription "covers" the former one. For example, a subscription having four filters a, b, c, and d "covers" another subscription having two filters a and b, so that a notification travelling up in the tree of the first subscription from node d to node a will pass through the nodes b and a of the tree of the second subscription, without the need for more than one notification delivery.

However, in either one of these two mechanisms, each single filter needs to be included

in each node's routing table to identify the filter's individual destination - a matter that may lead to an excessive table growth, and the corresponding notification may need to travel through many nodes to reach the last destination - a matter that may lead to considerable delays.

The **identity-based matching** mechanism, followed by routing, aims at matching subscriptions registered in a broker, all at once, exploiting the commonalities among these subscriptions. For example, consider the set F of four 32-bit filters: $F = \{abcd$, *efgh*, *ijkl*, $mnop\}$. If each of 31 subscriptions contains F, and does not contain any filter $\notin F$, these subscriptions are redundant, even though each one has its unique t-SID identification. The CRT registration process of each of these subscriptions does not lead to the construction of any tree. Instead, the CRT saves the four filters once in four CAM rows (i.e. only 16 bytes of CAM storage), and saves the 31 t-SIDs in four RAM rows (i.e. 128 bytes of RAM storage), as discussed in Section 5.3.3. Subsequently, matching the four filters against an available XML publication leads to the simultaneous identification of all involved 31 t-SIDs. Since these subscriptions may have been originated from either local or remote subscribers, the corresponding notifications either take place immediately for local subscribers or follow a routing path through the network of brokers for remote subscribers.

The routing path traces back the original remote subscription path. Therefore, if all redundant subscriptions have reached the broker through only one neighboring broker, a single notification is to be routed to this neighboring broker, but the routed notification will be equipped with an indication of multiple deliveries. If the remote redundant subscriptions have travelled multiple paths, a single notification traces back each of the multiple paths crossed by these redundant subscriptions. In order to keep a routing path short in a very large network of brokers, it is recommended to divide the large network

into scopes, where each scope contains a few number of brokers. We further elaborate on the point of scopes later in this chapter.

The **covering-based matching** mechanism aims at matching subscriptions registered in a broker, all at once, exploiting the "coverage" of filters among these subscriptions. For example, the filter $a > 5$ belonging to a subscription "covers" the filter $a > 2$ belonging to another subscription, because matching the filter $a > 5$ implicitly matches the filter $a > 2$. This mechanism is active in the X2CBBR and stems from the structure of the CAM and RAM of the CRT, where the complex attribute type t of $(10)_b$ is registered (refer to Section 5.3.3 for more details).

5.4.3 Content-Based Routing Mechanisms

We have developed two content-based routing mechanisms: (1) Subscription Routing Algorithm, which applies to XPath subscriptions, and (2) Notification Routing Algorithm, which applies to XML notifications that take place in consequence to successful matching of subscriptions against publications. Publication routing, which applies to XML publishers' content, is combined with the Notification Routing Algorithm. Before describing these mechanisms in detail, we state relevant notations.

5.4.3.1 Notations

On top of the multiple algorithms involved in the routing mechanisms, the broker works according to a sequence of the so-called "operation modes" or "operating modes." We may use either term to exactly mean the same concept.

The operation modes of the broker can be:

- (nP, nS), in the case of accepting new Subscriptions and new Publications

- (nP, oS), in the case of accepting new Publications while keeping old Subscriptions, in condition that no timeout occurs
- (oP, nS), in the case of accepting new Subscriptions while keeping old Publications
- (oP, oS), in the case of keeping old Subscriptions and old Publications, in condition that no timeout occurs.

In the following sections and algorithms, we used some notations as described next.

- P**:** the number of Publications that have been processed for matching.
- P_{max}: the total number of Publications available in a broker (whether they have been processed or are waiting for processing). A publication $p \notin P$ is considered available in a broker if either (1) it is in the buffer of the XML parser and has not been parsed yet, or (2) it has signalled its presence from an external buffer that is interfacing the broker.
- S**:** the set of all subscriptions already registered in the CRT.
- S_{max}: the maximum number of subscriptions that the broker can register and keep in the CRT. This value is usually set to ≤ 255 as discussed below along with the algorithm of Figure 5.17.
- $S_m \subseteq S$**:** the set of all subscriptions that match at least a Publication P_i, where $i = 1, 2, ..., P$.
- $S_{m_i} \subseteq S$**:** the set of all subscriptions that match a single particular Publication P_i, where $i = 1, 2, ..., or P$. This set resets itself for each publication that has been matched against registered subscription filters.

- $S_u \subseteq S$: the set of all subscriptions that do not match any single P_i, where $i = 1, 2, ..., P$.
- $S_a \not\subset S$: the set of newly arrived subscriptions that are not yet stored (i.e. not yet registered) in the CRT. A subscription $s \in S_a$ if (1) s exists in the Subscription Buffer of the SPU and (2) all filters of s do not exist in the CRT.

The setting for S_{max} depends on the capacity of the SMT and the CAM of the CRT, as described in S_{max} Determination Algorithm (S_{max} DA) depicted in Figure 5.17. Providing that the SMT can handle up to 255 (rather than 256) t-SIDs (see Section 5.3.2.2), S_{max} may not exceed this value (line 4 of the algorithm). Therefore, $S_{max} = 255$ initially. However, S_{max} may take a lesser value when the CAM of the CRT becomes full, as illustrated in line 6 of Figure 5.17.

Algorithm: S_{max} Determination.

1: **if** reset **then**
2: S_{max} **= 0**
3: **else if** (CAM of CRT not full) **then**
4: S_{max} **= 255 (providing 0xFF is the last possible SMT address)**
5: **else**
6: $S_{max} = S$
7: **end if**

Figure 5.17: Determination of the Maximum Number of Subscriptions.

However, S_{max} can retain the value of 255 when $S = 255$ as well as the CAM is full, in the case that the last registered t-SID actually is the last possible SMT address 0xFF. For example, if each subscription contains one filter only, and the width of this filter is $\leq$ 32 bits, and these filters are all distinctive (i.e. no identical or covering filters), there will be a single CAM entry per each subscription. Accordingly, S can reach $S_{max} = 255$ while the CAM's filled entries reach $S_{max} - 1 = 254$, because the CAM may hold 256 rather than 255 entries (i.e. one entry more than the SMT's maximum number of entries). If

only one of these subscriptions has distinctive filter data that is > 32 bits and ≤ 64 bits, S reaches $S_{max} = 255$ at the same time that the CAM becomes full.

5.4.3.2 Operating Mode Mechanism

In many pub/sub applications, the subscriptions by far outnumber available publications. However, there are some applications where publications may be more frequent than subscriptions. Sometimes, as in initial conditions, the broker may be under-utilized by both subscription and publication traffic. In other situations, both publication and subscription traffic may overwhelm the broker. The basic system idea for managing the incoming and outgoing traffic is to give as many available publications as possible the chance of matching as many subscriptions as possible, while taking into account the arrival rate of both publications and subscriptions. Therefore, we introduce the concept of operation modes, so that the broker can accommodate different situations, by initially working in a mode and reverting to another mode when certain circumstances occur. Moreover, the operation mode mechanism allows administrators to effectively monitor the current status of the broker. The algorithm of Figure 5.18 illustrates the operation mode mechanism represented by an FSM, while the tasks that the broker would undertake in each mode are included in the content-based routing algorithm depicted in Figure 5.20. Note that the notations stated in Section 5.4.3.1 are very useful in reading these two algorithms.

5.4.3.3 Sequence of Operating Modes

In each of its operating modes, the broker monitors the arrival of new subscriptions and publications, measures the increasing capacity of its memory resources that hold publications and subscriptions being processed, and updates its operating mode accordingly.

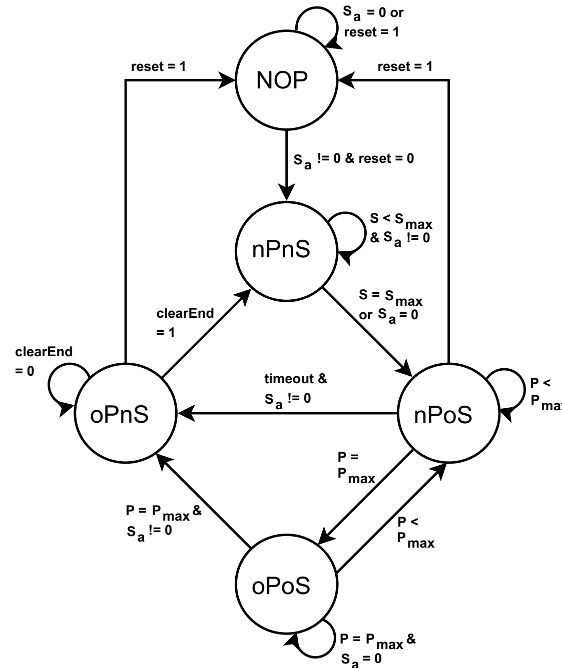

Figure 5.18: Operation Mode FSM.

Initially, the broker is in its "no operation" mode (state NOP in Fig. 5.18). Once the "reset" signal is released, the transition to a new operating mode only takes place when at least one subscription becomes available in the SPU of the broker (i.e. $S_a \neq \emptyset$). Meeting this condition, the broker transits to the (nP, nS) operation mode (state $nPnS$ in Fig. 5.18), where the CRT registers as many XPath subscriptions as possible.

However, in this mode, the broker does not allow its XML parser to process more than one XML publication (i.e. $P = 1$). Moreover, matching this single publication against registered subscriptions does not take place. The reason for only parsing one publication when the matching engine is disabled is understandable, since no need to parse more publications if they will not be exposed to the matching process. However, the reason for not enabling the matching process in the (nP, nS) mode stems from the fact that the registration process needs to access the CAM of the CRT, while the matching engine would need to access the same CAM if the matching process was enabled. Even though

we have implemented a handshake mechanism that allows the CAM to grant access permission to either the registration or matching processes (see Section 5.3.3.6), we have found that running both processes simultaneously leads to delays in the registration process. Furthermore, if the matching process had to take place concurrently with the registration process, the matching results would not be comprehensive, because the filters that are in the process of being registered (but not yet registered) would not be included in the matching process.

The broker may remain in its same (nP, nS) mode as far as $S < S_{max}$ and $S_a \neq \emptyset$. In other words, the role of the broker in this mode is primarily registering filters of as many new subscriptions as S_a contains until S reaches S_{max}.

When either S_a becomes null (i.e. when no more new subscriptions exist in the SPU) or S reaches S_{max}, the broker steps to the (nP, oS) mode (state $nPoS$ in Fig. 5.18), where the matching process immediately takes place. In this mode, no new subscriptions can be registered even if S_a becomes non-null (The "Subscription Buffer" can still store subsequent subscriptions in its other dual-port memory as shown in Figure 5.6). Instead, the broker buffers, parses, and matches as many XML publications as possible against registered subscriptions. While the broker reads parsed data of a publication P_i and matches such data against filters of registered subscriptions, it concurrently buffers a publication P_{i+1}. And while the broker matches the parsed P_{i+1}, it concurrently buffers P_{i+2}, and so on. The (nP, oS) mode stays into effect as far as $P < P_{max}$, in the condition that no timeout occurs. Therefore, in this mode, the same registered subscriptions have the chance of matching as many available publications as possible. Meanwhile, S_m and S_u are computed according to the matching results.

An additional major task undertaken by the broker during this mode is to issue and route out notifications resulting from the matched subscriptions versus publications,

referring to the computed S_m. Moreover, the broker may route out mismatched publications.

When a subscription timeout occurs, during the (nP, oS) mode, the registered subscriptions do not expire immediately. Instead, the broker inspects the status of S_a before switching to another mode. If $S_a = \emptyset$, the broker stays in its (nP, oS) mode as far as $P < P_{max}$, because the broker should give more chance to registered subscriptions to match newly-arrived publications since no new subscriptions reside in the SPU. If $S_a \neq \emptyset$, the occurrence of timeout leads the broker to transit to the (oP, nS) mode (state $oPnS$ in Fig. 5.18), where clearing of registered subscriptions takes place. Otherwise, the broker only reverts to the (oP, oS) mode (state $oPoS$ in Fig. 5.18) when the event of $P = P_{max}$ occurs.

The (oP, oS) mode illustrates a sort of "saturation" status, where all available publications and subscriptions have been processed and all pub/sub data have been compared. The broker stays in this operating mode as far as $P = P_{max}$ and $S_a = \emptyset$, even if timeout have occurred. In the event that $S_a \neq \emptyset$, the broker steps up to the (oP, nS) mode, but it switches back to the (nP, oS) mode in the event of a newly-arrived publication determined by $P < P_{max}$.

The main task of the broker during the (oP, oS) mode is to route out mismatched subscriptions, referring to the computed S_u. Switching back to the (nP, oS) mode, the broker suspends the routing process because S_u may need to be updated. However, stepping up to the (oP, nS) mode does not suspend the routing process. Instead, the clearing process of S_u suspends until all mismatched subscriptions are routed out. Once the clearing process ends up, the (nP, nS) mode takes over again.

5.4.3.3.1 Discussion. From the aforementioned discussion, referring to Figure 5.18, the sequence of the operating modes $[(nP, nS), (nP, oS), (oP, nS)]$, is more likely to happen when the publication traffic is so intense to the extent that P never reaches P_{max}, until old subscriptions expire while at least one new subscription waits for processing. A short duration of the (nP, oS) mode would reveal a short configured timeout and relatively frequent subscription traffic. However, a long duration of the (nP, oS) mode would reveal a long configured timeout or relatively infrequent subscription traffic.

The sequence of the operating modes $[(nP, nS), (nP, oS), (oP, oS), (oP, nS)]$, with short duration in the (oP, oS) mode, is more likely to happen when the subscription traffic is much more intense than the publication traffic to the extent that when P reaches P_{max} the broker quickly steps up to the (oP, nS) mode.

However, the sequence of the operating modes $[(nP, nS), (nP, oS), (oP, oS), (oP, nS)]$, with long duration in the (oP, oS) mode, is more likely to happen when both the subscription and publication traffic is not so intense to the extent that P quickly reaches P_{max} because of the low value of P_{max}, with no new publications and subscriptions have arrived or are to be processed.

While in the (oP, oS) mode, the broker may revert back to its (nP, oS) mode and forth again to the (oP, oS) mode. This scenario occurs when the broker suddenly receives a new publication after it becomes saturated with both subscriptions and publications, providing that no more subscriptions arrive.

5.4.3.4 TOP-Level Registering, Matching, and Routing Mechanism

In each operating mode (Figure 5.18), the broker may perform multiple tasks concurrently in addition to the sequential tasks. For each of the broker's operating modes, Figure 5.19 horizontally arranges the concurrent tasks, and vertically depicts the sequential tasks.

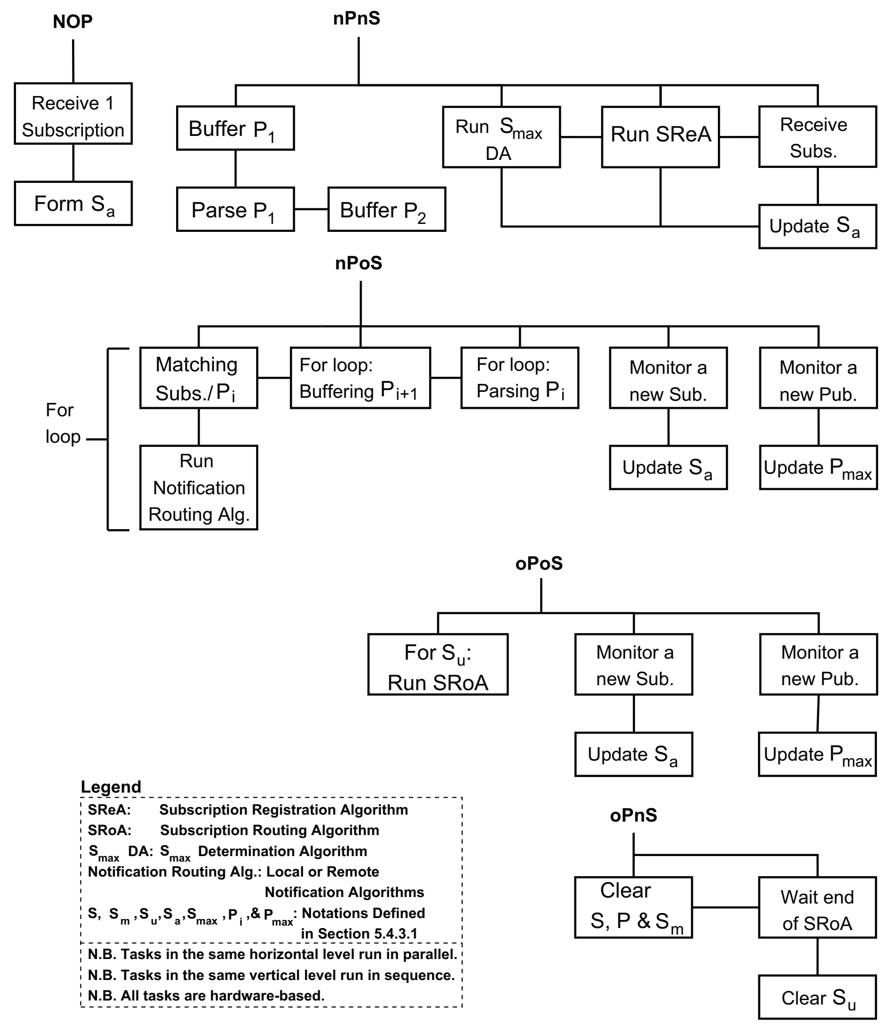

Figure 5.19: Diagram Illustrating Task Concurrency in Each Operating Mode.

Based on the FSM of Figure 5.18, and referring to the sequential and concurrent tasks shown in Figure 5.19, we depict the detailed top-level algorithm in Figure 5.20. This algorithm embeds all algorithms involved in subscription registration, matching sub/pub data, and routing notifications and subscriptions. Next, we describe the top-level algorithm referring to Figures 5.19 and 5.20. It is useful to remind the reader that we have developed and implemented all tasks of this algorithm in hardware.

Top-Level Algorithm: Content-Based Registering, Matching, and Routing

1: **if** $mode = NOP$ **then**
2: **Buffer only one new subscription; Form** S_a **& Report** $S_a \neq \emptyset$
3: **else if** $mode = (nP, nS)$ **then**
4: **Buffer only one new publication** P_i **where** $i = 1$ **(**$P = 1$**)**
5: **Parse & Report** $parsing_end$ **of** P_i
6: **Set new** P_{max}
7: **Set new** S_{max}**: Run {**S_{max} **Determination Algorithm (DA)}**
8: **Store new set** S **in CRT (Attempt** $S_a \subseteq S$**): Run {Subscription Registration Algorithm (SReA)}**
9: **Receive Subs; Update & Report** S_a**; Report High Commonality Degree (HCD)**
10: **else if** $mode = (nP, oS)$ **then**
11: **{3 FOR loops run in PARALLEL}**
12: **{1st FOR loop}**
13: **for** P_i where $i = 2 \rightarrow P_{max}$ **do**
14: **Buffer** P_i
15: **Wait & Report** $buffering_end$ **of** P_i **& Wait** $parsing_start$ **of** P_i
16: **end for**
17: **{2nd FOR loop}**
18: **for** P_i where $i = 2 \rightarrow P_{max}$ **do**
19: **Wait (**$parsing_end$ **of** P_{i-1}, $matching_end$ **of** P_{i-1}**, &** $buffering_end$ **of** P_i**)**
20: **Parse** P_i **& Report** $parsing_start$ **of** P_i **& Report** $parsing_end$ **of** P_i
21: **end for**
22: **{3rd FOR loop}**
23: **for** P_i where $i = 1 \rightarrow P_{max}$ **do**
24: **Wait** $parsing_end$ **of** P_i
25: **Match** P_i **against** S
26: **Update** S_{m_i}, S_m, **&** S_u
27: **for all** S_{m_i} **do**
28: **if** Local $s \in S_{m_i}$ **then**
29: **Run {Local Notification Routing Algorithm}**
30: **else**
31: **Run {Remote Notification Routing Algorithm}**
32: **end if**
33: **end for**
34: **end for**
35: **Store new** P_i**; Set new** P_{max}**; Update & Report** S_a**; Report** *timeout*
36: **else if** $mode = (oP, oS)$ **then**
37: **if** High S_u and High Commonality Degree (HCD) **then**
38: **Run Subscription Routing Algorithm (SRoA) for a single subscription with** ***null SID***
39: **else**
40: **for all** S_u **do**
41: **Run Subscription Routing Algorithm (SRoA)**
42: **end for**
43: **end if**
44: **Report** sub_route_end
45: **Store new** P_i**; Set new** P_{max}**; Update & Report** S_a
46: **else if** $mode = (oP, nS)$ **then**
47: **Clear** S **and** S_m
48: **if** sub_route_end **then**
49: **Clear** S_u
50: **else**
51: **Wait** sub_route_end
52: **Clear** S_u
53: **end if**
54: **Report** $clear_end$
55: **Reset (**P_i **(**$i = 0$**) &** *timeout***)**
56: **else**
57: **Shutdown**
58: **end if**

Figure 5.20: Pseudo-Code for Top-Level Algorithm.

5.4.3.4.1 Details of the Top-Level Algorithm. The initial mode of the broker is NOP, where the SPU can receive and buffer one new subscription, thereby forming the set S_a. Subsequently, the broker undertakes a cycle of operating modes starting at the (nP, nS) mode. Reaching the end of the cycle at the (oP, nS) mode, the broker becomes ready to step to a new cycle starting again at the (nP, nS) mode. It never goes back to its initial NOP mode unless a "reset" signal is applied.

Mode (nP, nS). In the (nP, nS) mode, multiple tasks/algorithms may run in parallel:

(1)- The SPU continues to receive and buffer new subscriptions upon their arrival. Consequently, the SPU updates the arrival set S_a at the end of each buffered subscription.

(2)- The broker runs the Subscription Registration Algorithm (SReA) (Figure 5.14) that registers subscription filters in the CRT. In the case that commonalities exist in many subscriptions, the algorithm reports the High Commonality Degree (HCD) signal. The SPU acknowledges the reception of this signal.

(3)- The broker runs the S_{max} Determination Algorithm (Figure 5.17) that determines the maximum number of subscriptions that the broker can register and keep in the CRT.

(4)- The XML parser of the PMU receives, buffers, and parses one new XML publication. Buffering - without parsing - a second XML publication (if it is available), concurrently with parsing the first one, is another task that may take place. Moreover, the broker monitors and updates P and P_{max}. However, no matching of parsed publication data against subscription filters would take place.

Mode (nP, oS). The matching process of parsed publication data against registered subscription filters may only take place in the (nP, oS) mode. Consequently, notification routing and delivery is a process that only runs in this mode. However, matching an XML publication p against registered subscription filters cannot start until the XML

parser flags the *parsing_start* signal for p. Once the matching process starts, the parsing process of p continues concurrently while the buffering process of the next publication takes place. In other words, the three processes: buffering, parsing, and matching run concurrently at some point after the beginning of each process.

Initially, the matching process runs for the publication $p = P_1$ that has been parsed in the (nP, nS) mode. Concurrently, the parsing process runs for the publication $p = P_2$ that has been buffered in the (nP, nS) mode. Concurrently as well, the buffering process runs for the publication $p = P_3$. Lines 10 to 35 in Figure 5.20 state three "for loops" that run in parallel indicating the synchronization of the signals *buffering_end*, *parsing_start*, and *parsing_end* through the loops. The matching process continues for all available publications up to P_{max} that is concurrently monitored and updated upon the arrival of each new publication.

The notification delivery takes place for each publication that matches a local subscription, using the Local Notification Routing Algorithm illustrated in Figure 5.16. A local matched subscription s must $\in S_{m_i}$, and can be identified (as local) by the NPU when it retrieves the corresponding o-SID from the SMT (Definitions of local and remote subscriptions were earlier stated in Section 5.2.1, and the method used by the broker to differentiate between local and remote subscriptions will later be discussed in Section 5.4.3.5.1). Routing notifications to remote subscribers also takes place according to the Remote Notification Routing Algorithm later discussed in Section 5.4.3.5. Even though lines 28 to 32 of the algorithm in Figure 5.20 point to two separate notification algorithms for the sake of clarity, there is actually one combined algorithm Figure 5.22 that includes both local and remote notification routing algorithms.

It is useful to mention that throughout this mode, the broker monitors the arrival of new subscriptions and accordingly updates S_a, even though the registration process of

new subscriptions does not take place.

Mode (oP, oS)**.** In lines 36 to 45 of the algorithm of Figure 5.20, the (oP, oS) mode becomes into effect, where the broker does not perform neither registration of subscriptions nor matching of publications versus subscriptions. Reaching this mode, the broker has to deal with subscriptions that have not matched any available publications. The broker is able to identify these unmatched subscriptions by means of the set S_u (defined in Section 5.4.3.1) that must have reached its final value for all registered subscriptions in the previous mode. Therefore, the main task of the broker in this mode is to forward these subscriptions to neighboring brokers using its "Subscription Routing Algorithm" later discussed in Section 5.4.3.6. This routing procedure is an opportunity for these subscriptions to match publications that may be available in other brokers in the network.

In the case that S_u is dense (i.e. there is a high number of mismatched subscriptions – typically at least 31 subscriptions) and the signal of High Commonality Degree (HCD) is reported by the SReA, this situation tells the broker that many mismatched subscriptions potentially have commonalities. Therefore, it would be unnecessary to route traffic potentially requesting common data. In order to reduce such homogeneous traffic, the broker only forwards a single mismatched subscription to a neighboring broker (lines 37 and 38). This subscription must contain all of its original raw information except that its SID becomes null. This *null SID* tells the broker of destination that the subscription is just a sample of many similar subscriptions in the broker of origin.

Concurrently with subscription routing, the broker monitors the arrival of new publications (to update P_{max}) and new subscriptions (to update S_a). Accordingly, the broker switches to a new operating mode (Figure 5.18).

Mode (oP, nS)**.** Once in the (oP, nS) mode, the broker applies a hardware-based self-removal mechanism that writes a series of zeroes (e.g. 0x00000000) in some of its memory

resources (e.g. CRT, SMT, and the SCBXP's memory that contains parsed XML data). This mechanism frees such resources from all processed subscriptions and publications and their respective sets, such as S, S_m, and P. In the case that the subscription routing process - which has started in the (oP, oS) mode - is still on-going, the broker waits until the end of the routing task to clear S_u (lines 51 and 52 in Figure 5.20).

However, the broker does not clear S_a and P_{max} because it needs them in the new cycle of operating modes starting from the (nP, nS) mode. The complete removal of S_a and P_{max} may only occur in the NOP mode that becomes into effect due to the application of a "reset" signal.

5.4.3.5 Remote Notification Routing

The delivery of notifications is the main task of the NPU (Section 5.3.4). When the PMU (Section 5.3.3) finds an XML publication matching a registered XPath subscription, the NPU receives the corresponding t-SID of the matched subscription and communicates with the SPU (Section 5.3.2) in order to retrieve the corresponding o-SID of the subscription. Upon reception and inspection of the o-SID, the NPU can identify whether the subscription was local or remote. In the case that the subscription is local, the NPU runs the Local Notification Algorithm that has been discussed earlier and depicted in Figure 5.16. In this section, we discuss:

- The method that allows the NPU to differentiate, through the o-SID, between a local subscription and a remote subscription.
- The mechanism of routing notifications to remote subscribers. This mechanism is called "Remote Notification Routing Algorithm".

5.4.3.5.1 Differentiation between Local and Remote Subscriptions. The subscription's o-SID consists of a minimum of one byte and a maximum of 8 bytes as discussed earlier in Section 5.3.2.2. The NPU inspects the first and second most significant bytes of the o-SID. The first most significant byte identifies the subscription, and the second most significant byte identifies the broker that originally received and forwarded the subscription. This original broker would not include its identification data in the o-SID unless it decides to forward the subscription to another broker. Therefore, if the second most significant byte is null, the NPU acknowledges the absence of broker identification - a matter that reveals that the subscription had never been forwarded to a second broker. Accordingly, the subscription must be local. Otherwise, the subscription is remote. Figure 5.21 shows the structure of the 8-byte o-SID that allows the NPU to differentiate between a local and a remote subscription.

Therefore, based on Figure 5.21, the o-SID of a local subscription can be represented as o-SID $= \{SID, 0\}$, where SID is a one-byte subscription ID. A remote subscription s for a broker $B2$ must have crossed a neighboring broker, say $B1$. Furthermore, if this subscription s was local to the neighboring broker $B1$ before reaching the broker $B2$, the o-SID of the subscription appears in $B2$ as o-SID $= \{SID, B1, 0\}$, where $B1$ is the Broker ID (BID) of the broker $B1$. In addition, if this subscription s continues its path to a third broker $B3$, the o-SID of s appears in $B3$ as o-SID $= \{SID, B1, B2, 0\}$. Since the o-SID consists of 8 bytes, there would be a maximum of 7 brokers in the path of a subscription. In this latter case, the o-SID appears in a broker $B7$ as o-SID $= \{SID, B1, B2, B3, B4, B5, B6, 0\}$. Subsequently, if s matches an available publication in the broker $B7$, this broker sends a notification through its neighboring broker $B6$ with o-SID $= \{SID, B1, B2, B3, B4, B5, B6, B7\}$.

Following the aforementioned discussion, and adding to the concept of local and

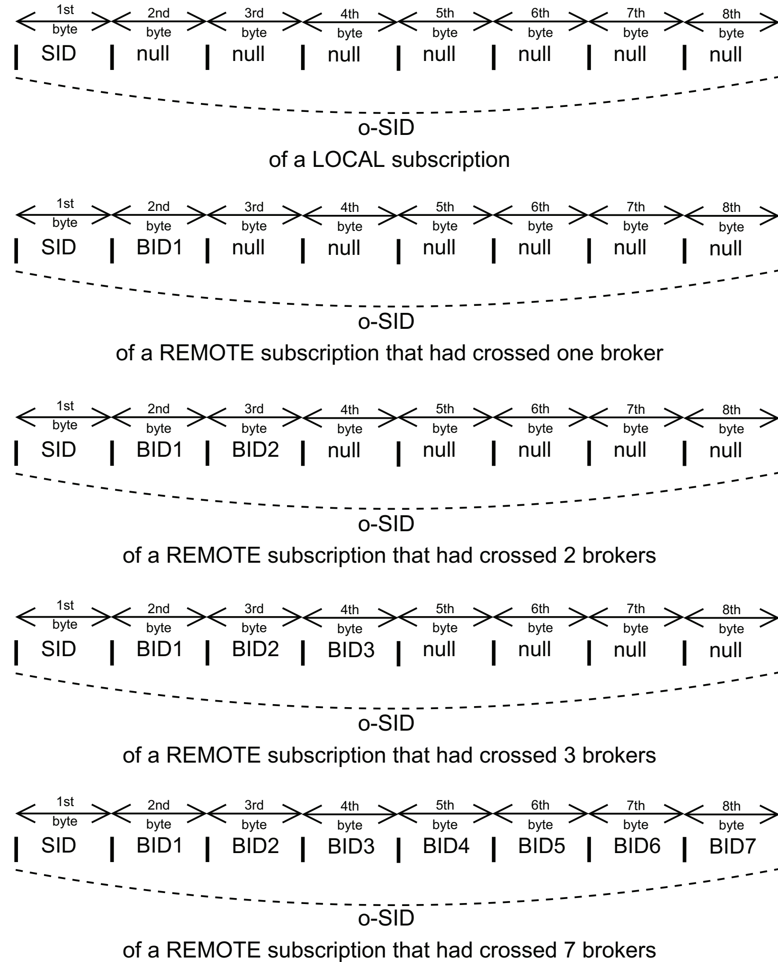

Figure 5.21: The o-SID Structure in Local and Remote Subscriptions.

remote subscriptions defined in Section 5.2.1, we can formally state that:

1. A local subscription "*s*" available in a broker B_i is an XPath subscription sent as *sub(o-SID, F)* to B_i, where *o-SID* = {*SID*,*0*}.

2. A remote subscription "*s*" available in a broker B_i is an XPath subscription sent as *sub(o-SID, F)* to B_i through at least one neighboring broker $B_j \in B_i^N$ (Definition 3 in Section 5.2.1), where *o-SID* = $\{SID, ...BID..., B_j\}$ and "$...BID...$" represents the BIDs of all brokers crossed by the subscription before reaching B_j.

Generalization in the Case of a Remote Subscription. Since BID consists of one byte (8 bits), a broker may theoretically have 255 neighboring brokers whose BID may take the value of 1 through 255. And with an o-SID of a maximum of 8 bytes, a subscription may travel through a path of a maximum of 7 brokers *(8 bytes for o-SID = (7 brokers * 1 byte) + 1 byte for SID)*. If we modify the BID length from 8 to 4 bits, a broker may have a maximum of 15 neighboring brokers whose BID may take the value of 1 through 15. In this case, with an o-SID of a maximum of 8 bytes, a subscription may travel through a path of a maximum of 14 brokers *(8 bytes for o-SID = (14 brokers * 4 bits) + 1 byte for SID)*. Furthermore, if the BID length is just 2 bits (the smallest possible length), a broker may have a maximum of 3 neighboring brokers, while a subscription may travel through a path of 28 brokers *(8 bytes for o-SID = (28 brokers * 2 bits) + 1 byte for SID)*.

Let B_s is the number of brokers traversed by a subscription s. In general, with o-SID length of l bits, a fixed SID length of 8 bits, and a BID length of b bits, a subscription s may take a path that consists of a maximum of $B_{s_{max}}$ brokers, as follows:

$$B_{s_{max}} = \frac{l-8}{b} \text{ brokers},$$

where $2 \leq b \leq 8$ and $10 \leq l \leq 64$. In order to avoid a fraction number of brokers, $(l-8)$ must be a multiple of b.

Referring to the discussion above and Definition 3 in Section 5.2.1, the maximum number of neighboring brokers that a broker B_i may have is:

$$B_i^N = 2^b - 1 \text{ brokers}.$$

5.4.3.5.2 Remote Notification Routing Algorithm. When the NPU finds that the successfully-matched subscription is remote through the subscription's o-SID, the NPU runs the Notification Routing Algorithm for a remote subscription. To better illustrate this algorithm, we combine the local notification routing algorithm of Figure 5.16 with the remote notification routing algorithm as depicted in Figure 5.22.

Algorithm: Local & Remote Notification - Generation and Routing

1: **final successful match for a t-SID of a subscription** s
2: **address of SMT** $saddr \equiv$ **t-SID**
3: **o-SID** $\leftarrow$ **SMT Entry @** $saddr$ **(i.e. o-SID** $\leftarrow$ **M[t-SID])**
4: **if** o-SID = {SID,0} {case of a local subscription} **then**
5:     **generate corresponding notification** N **as** ***notify(SID, N)***
6: **else**
7:     **if** o-SID(s) = {SID, ...BID..., B_n} and SID $\neq 0$ {case of a remote subscription} **then**
8:         **pick up the neighboring broker's BID (**B_n**) from** ***o-SID***
9:         **update** ***o-SID*****:** ***o-SID(s)*** $\leftarrow$ **{*****SID, ...BID...,*** B_n***, own BID*****}**
10:         **generate corresponding notification** N **as** ***forward(o-SID(s), N)***
11:         $B_n \leftarrow$ ***forward(o-SID(s), N)***
12:     **else**
13:         {case of SID = 0 for mutliple remote subscritptions of common filters (o-SID(s) = {0, ...BID..., B_n})}
14:         **pick up the neighboring broker's BID (**B_n**) from** ***o-SID***
15:         **update** ***o-SID*****:** ***o-SID(s)*** $\leftarrow$ **{*****0, ...BID...,*** B_n***, own BID*****}**
16:         **prepare corresponding publication** P **as** ***forward(o-SID(s), P)***
17:         $B_n \leftarrow$ ***forward(o-SID(s), P)*** **(*****Publication Routing*****)**
18:     **end if**
19: **end if**

Figure 5.22: Combined Local & Remote Notification/Publication Routing Algorithm.

In line 8 of this algorithm, the NPU picks up the BID of a neighboring broker from the o-SID of a matched subscription. Since the remote subscription may have crossed multiple brokers in its path, multiple BIDs may have been concatenated in the o-SID. Then, how would the broker know which BID belongs to the last neighboring broker crossed by the subscription? The answer to this question is straightforward: The broker simply picks the least significant non-null bits of length b, where the value of b is the encoded length of the BID of any broker in the network. Then, in line 9, the broker updates the o-SID by concatenating it with its own BID. In line 10, the broker adds the updated o-SID to the issued notification N, and in line 11, it forwards N that is

embedded in the message "*forward(o-SID, N)*" to the neighboring broker whose BID was identified in line 8.

5.4.3.5.2.1 Publication Routing. The line 13 of the algorithm points to the case of receiving a remote subscription with SID = 0. The null-value of the SID tells the broker that the received subscription is a "sample" of multiple subscriptions containing common filters. The up-stream broker decides to only send a single subscription in order to reduce unnecessary homogeneous traffic by avoiding to transmit many repetitive subscriptions that potentially consist of common filters. This traffic reduction positively impacts the overall performance of the system.

In this case, the NPU responds with the publication itself (lines 16 and 17), if matching of the involved filters occurs, instead of the corresponding notification. This response brings the publication down to the broker where the multiple subscriptions of common filters originally exist. Eventually, the local broker of such homogeneous traffic receives the routed publication and treats it as a new available publication. Subsequently, rematching this publication against subscriptions of common filters takes place in these subscriptions' local broker. As a result, corresponding notifications can locally reach their destinations.

5.4.3.5.2.2 Notification Re-routing. A broker may receive notifications as a result of running the Remote Notification Routing Algorithm in the up-stream broker. The broker does not need to process such notifications in-depth. Instead, it buffers the notification in the notification buffer of the NPU (Part C in Figure 5.2), and inspects the o-SID embedded in the received message "*forward(o-SID, N)*" in order to appropriately route the notification to the intended destination.

Algorithm: Notification Re-Routing

1: **receive a notification** N **from the up-stream broker whose BID** $= B_u$ **as** ***forward(o-SID(s), N)***
2: **strip** B_u **and own BID off** ***o-SID*****:** ***o-SID*** $\leftarrow$ **{*****o-SID*** $- B_u -$ ***own BID*****}**
3: **if** o-SID = {SID,0} {case of a local notification delivey} **then**
4:     **deliver notification** N **as** ***notify(SID, N)***
5: **else**
6:     {notification must be re-routed (*o-SID = SID, ...BID...,* B_d)}
7:     **pick up the down-stream broker's BID (**B_d**) from** ***o-SID***
8:     **update** ***o-SID*****:** ***o-SID*** $\leftarrow$ **{*****SID, ...BID...,*** B_d**,** ***own BID*****}**
9:     **route notification** N **as** $B_d \leftarrow$ ***forward(o-SID, N)***
10: **end if**

Figure 5.23: Notification Re-Routing Algorithm.

Figure 5.23 shows the steps of the notification re-routing algorithm. Upon receiving a notification from an up-stream broker, the NPU removes the BID of this up-stream broker from the o-SID (i.e. removing B_u as illustrated line 2 of the algorithm). The resulting o-SID must reveal the BID of the current broker, because the up-stream broker only sent the notification to the current broker after it had identified the BID of the current broker and had updated the o-SID with the up-stream broker's own BID (for more clarification, read the example below). At this point, the broker also removes its own BID from the o-SID. The action of stripping these two BIDs off the o-SID appears in line 2 of the algorithm in Figure 5.23.

If the resulting o-SID has the format of {*SID, 0*} after the removal of the up-stream broker's BID and the current broker's own BID, the NPU runs the local notification routing algorithm and delivers the notification N as *notify(SID, N)* (line 4 of the algorithm). Otherwise, the resulting o-SID must have the format {*SID, ...BID...,* B_d} where B_d is the BID of a down-stream broker (the symbol "...BID..." was defined in Section 5.4.3.5.1). Accordingly, the NPU updates the resulting o-SID with broker's own BID (line 8 of the algorithm) and forwards the notification with the updated o-SID to the down-stream broker of BID = B_d (line 9 of the algorithm).

For example, a subscription s may have traversed a total of three brokers whose BIDs

are $B1$, $B2$, and $B3$, and successfully matched a publication in the broker of BID = $B3$. The subscription s must have reached B_3 with *o-SID* = $\{SID, B_1, B_2\}$. Then, B_3 issues a notification to B_2 with an *o-SID* = $\{SID, B_1, B_2, B_3\}$. The broker B_2 first strips the up-stream broker's BID (B_3) off the o-SID. The BID of the current broker B_2 becomes exposed in the resulting o-SID. Then B_2 strips its own BID (B_2) from the resulting o-SID to reveal the BID of the down-stream broker which appears to be B_1. Subsequently, the broker updates the resulting o-SID with B_2 and forwards the notification to B_1 with the updated *o-SID* = $\{SID, B_1, B_2\}$. The broker B_1 strips the two BIDs off the o-SID to find out that the resulting *o-SID* = {*SID, 0*}. The absence of BIDs in the o-SID reveals that the notification should not be forwarded to any other broker. Then, B_1 delivers the notification to a local subscriber identified by the final o-SID.

Important Comment. The Notification Re-Routing Algorithm may run in any operating mode (even in the *NOP* mode if the "reset" signal is released). In other words, running this algorithm depends on the reception of a notification from an up-stream broker regardless of any of the broker's operating modes shown in Figure 5.18. Therefore, this algorithm can concurrently run with any other task that the broker may be undertaking during any of its operating modes. However, this algorithm may temporarily become on-hold if the broker is using the same external link to route other notifications or subscriptions. In this case, this algorithm resumes its tasks once the link becomes free.

5.4.3.6 Subscription Routing Algorithm (SRoA).

Having introduced and described the structure of o-SID for a subscription, we can update the formulation representing the action of subscribing to the system discussed in Section 5.3.1. In this regard, a subscription s can take the form of *sub(o-SID(s),F)* when

Algorithm: Subscription Routing

1: **final mismatch for a t-SID of a subscription** s **whose filters** $F = \{f_i\}$ **and** $i = 1, 2, 3, ...n$.
2: **address of SMT** ***saddr*** $\equiv$ **t-SID**
3: **o-SID** $\leftarrow$ **SMT Entry @** ***saddr*** **(i.e. o-SID** $\leftarrow$ **M[t-SID])**
4: **if** o-SID(s) = {SID,0} {case of a local subscription} **then**
5: **update** ***o-SID*****:** ***o-SID(s)*** $\leftarrow$ **{*SID, own BID, 0*}**
6: **issue subscription** s **as** ***sub(o-SID(s), F)***
7: **pick up a neighboring broker's BID** (B_v) **from VRT**
8: $B_v \leftarrow$ ***sub(o-SID(s), F)***
9: **else**
10: {case of a remote subscription}
11: **pick up the neighboring broker's BID** (B_s) **from** ***o-SID(s)***
12: **update** ***o-SID*****:** ***o-SID(s)*** $\leftarrow$ **{*SID, ...BID...,*** B_s***, own BID, 0*}**
13: **issue subscription** s **as** ***sub(o-SID(s), F)***
14: **pick up a neighboring broker's BID** $(B_v \neq B_s)$ **from VRT**
15: $B_v \leftarrow$ ***sub(o-SID(s), F)***
16: **end if**

Figure 5.24: Subscription Routing Algorithm (SRoA).

it reaches a broker, where *o-SID(s)* is the o-SID of the subscription s, and the subscription's filter set F consists of n filters f_i ($F = \{f_i\}$, *and* i = *1,2,3,.....n*). Based on the structure of o-SID discussed in Section 5.4.3.5.1, the subscription is local if its *o-SID(s)* = {*SID, 0*} and remote if its *o-SID(s)* = {*SID, ...BID...*}, where *...BID...* may occupy up to 7 bytes of brokers' BIDs (Figure 5.21).

As discussed in previous sections, the SPU (Section 5.3.2) buffers the subscription's filter set F and maps the subscription's o-SID to a single-byte t-SID. The PMU (Section 5.3.3) registers F and the t-SID in the CRT. In Section 5.3.2.3, we discussed that the SPU forwards a subscription to a neighboring broker when this subscription has not matched any of the available publications.

The broker can recognize a mismatched subscription via S_u that has been constructed during the broker's *(nP,oS)* mode (line 26 in Figure 5.20). In its *(oP,oS)* mode (line 41 in Figure 5.20), the broker forwards the mismatched subscription according to the Subscription Routing Algorithm (SRoA) depicted in Figure 5.24).

The SRoA runs for each subscription indicated by S_u as follows. The SPU receives from the PMU the t-SID of the mismatched subscription, retrieves the corresponding

o-SID from the SMT (line 3 of the algorithm), adds the broker's own BID to the o-SID (line 4 or line 12 of the algorithm), extracts the subscription's filters from the "Subscription Buffer," reconstructs the raw subscription as *sub(o-SID(s),F)* (line 6 or line 13 of the algorithm), and forwards the raw subscription to a neighboring broker via the "Subscription Forwarder" (line 8 or line 15 of the algorithm). The determination of which neighboring broker the broker forwards a mismatched subscription is done based on the Vicinity Routing Table discussed next.

5.4.3.6.1 Vicinity Routing Table. The Vicinity Routing Table (VRT) may receive information through a dedicated interface (Figure 5.2). This VRT information somewhat describes the topology of the network in the vicinity of a specific broker. Basically, such information represents topology messages indicating the neighboring brokers to a specific broker and the criteria that should be met to route messages to these brokers. For example, a broker B_i may receive a topology message that would be of the format *top(X, BID)*, where $BID \in B_i^N$ and X can be a filter, a topic, a data type, or any specific content. Once deciding to route a mismatched subscription out, the SPU first identifies the subscription criterion (i.e. a filter, a topic, a data type, or any specific content), consults the VRT with such criterion to retrieve a neighboring BID, and routes the subscription out to the identified neighboring broker.

The VRT includes a small RAM module provided by Altera library. It can be regularly updated through topology messages issued from an external specialized entity that has knowledge of the network topology. However, we do not describe such an external entity any further since it is out of the scope of this thesis. We manually enter the necessary vicinity routing information in this table.

5.4.3.7 Example.

Consider two subscriptions s_1 and s_2. The subscription s_1 is local to Broker 1, while the subscription s_2 is local to Broker 3 in a network of three brokers as shown in Figure 5.25 and Figure 5.26.

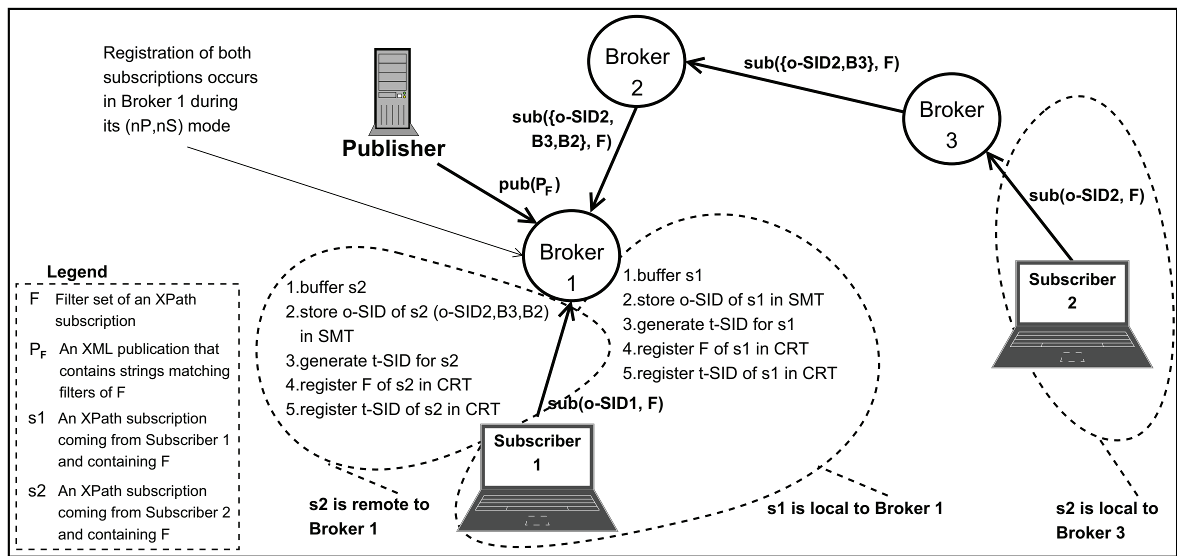

Figure 5.25: Storing/Registering Two Subscriptions (One Local and One Remote) in a Broker.

The BIDs B1, B2, and B3 belong to Brokers 1, 2, and 3 respectively. Both subscriptions contain the same filter set F. An XML publication P_F contains F as well but it is only available in Broker 1.

The Subscription Registration Algorithm (SReA) (Figure 5.14) runs for s_2 in Broker 3 and for s_1 in Broker 1. The matching process that occurs in Broker 3 does not result in a successful match for s_2. The Subscription Routing Algorithm (SRoA) of Figure 5.24 then applies to s_2 since a matching publication does not exist in Broker 3. The subscription takes the route from Broker 3 to Broker 2 and up to Broker 1 where a matching publication exists.

In Broker 2, both the SReA and SRoA run for s_2 as was the case in Broker 3.

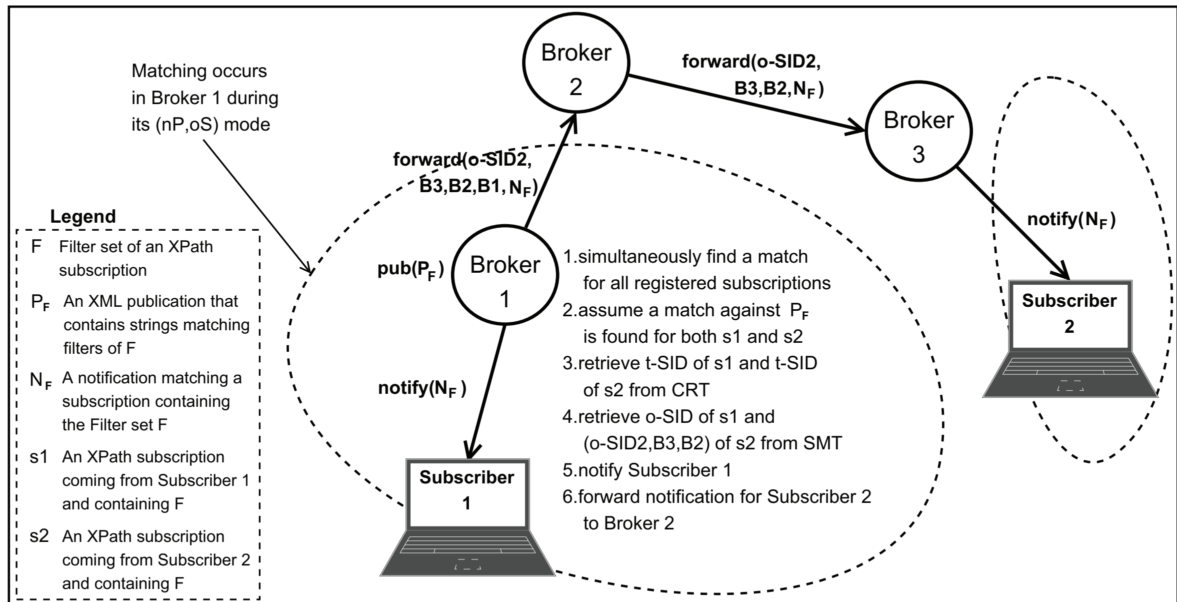

Figure 5.26: Matching Two Subscriptions (One Local and One Remote) and Notifying Corresponding Subscribers.

Finally, the SReA occurs for both subscriptions in the (nP, nS) mode of Broker 1, where exploiting commonalities between available subscriptions takes place. Thus, in Broker 1, the SReA registers the filter set F only once. Figure 5.25 illustrates the steps of the subscriptions' routing and registering processes.

Therefore, matching both subscriptions against P_F only occurs in Broker 1 that considers s_2 a remote subscription. The matching engine (Figure 5.15) initiates this matching process during the (nP, oS) mode of Broker 1. Figure 5.26 illustrates the corresponding matching and notifying steps.

5.4.3.7.1 Performance Analysis. In terms of new notations in this section, the arrow under the sub function indicates subscription routing, the arrow under the $notify$ function indicates notification delivery, and the arrow under the $forward$ function indicates notification routing.

For the **subscription** s_1, the end-to-end path (from sending the subscription until

receiving the corresponding notification) is as follows:

1. Subscriber 1 $\xrightarrow{sub(o-SID1,F)}$ Broker 1
2. Match found in Broker 1
3. Broker 1 $\xrightarrow{notify(N_F)}$ Subscriber 1

Providing that the duration of $sub(o - SID1, F)$ is T_{s_1}, that the total processing and matching time within Broker 1 is T_{p_1}, and that the duration of the relevant notification is T_{n_1}, thus, the duration of the end-to-end path T_{e_1} is as follows: $T_{e_1} = T_{s_1} + T_{p_1} + T_{n_1}$.

For the **subscription** s_2, the end-to-end path is as follows:

1. Subscriber 2 $\xrightarrow{sub(o-SID2,F)}$ Broker 3
2. No match found in Broker 3
3. Broker 3 $\xrightarrow{sub(\{o-SID2,B3\},F)}$ Broker 2
4. No match found in Broker 2
5. Broker 2 $\xrightarrow{sub(\{o-SID2,B3,B2\},F)}$ Broker 1
6. Match found in Broker 1
7. Broker 1 $\xrightarrow{forward(\{o-SID2,B3,B2,B1\},N_F)}$ Broker 2
8. Broker 2 $\xrightarrow{forward(\{o-SID2,B3,B2\},N_F)}$ Broker 3
9. Broker 3 $\xrightarrow{notify(N_F)}$ Subscriber 2

Providing that the duration of $sub(o - SID2, F)$ is T_{s_2}, that the total processing and matching time within Broker 3 is T_{p_3}, that the duration of $sub(\{o - SID2, B3\}, F)$ is T_{s_3}, that the total processing and matching time within Broker 2 is T_{p_2}, that the

duration of $sub(\{o-SID2, B3, B2\}, F)$ is T_{s_4}, that the total processing and matching time within Broker 1 is T_{p_1}, that the duration of the notification routing $forward(\{o-SID2, B3, B2, B1\}, N_F)$ is T_{n_2}, that the duration of $forward(\{o-SID2, B3, B2\}, N_F)$ is T_{n_3}, and that the duration of the relevant notification delivery is T_{n_4}, thus, the duration of the end-to-end path T_{e_2} is as follows:

$T_{e_2} = T_{s_2} + T_{p_3} + T_{s_3} + T_{p_2} + T_{s_4} + T_{p_1} + T_{n_2} + T_{n_3} + T_{n_4}$.

Since both subscriptions contain the same filter set F, the transmission time with regards of the number of bytes in each subscription can be the same. Moreover, it is quite possible that the end-user who subscribes to the local Broker 1 with the subscription s_1 utilizes the same technology that the other end-user utilizes in subscribing to the local Broker 3 with the subscription s_2. Therefore, the equation of $T_{s_1} = T_{s_2}$ is quite possible. Similarly, the notification delivery time T_{n_1} can be equal to T_{n_4}. Furthermore, T_{p_1} appears in either T_{e_1} or T_{e_2}. Therefore, T_{e_2} can be rewritten as follows:

$T_{e_2} = T_{s_1} + T_{p_1} + T_{n_1} + T_{p_3} + T_{s_3} + T_{p_2} + T_{s_4} + T_{n_2} + T_{n_3}$. Thus,

$T_{e_2} = T_{e_1} + T_{p_3} + T_{s_3} + T_{p_2} + T_{s_4} + T_{n_2} + T_{n_3}$.

With further assumptions of roughly-equal processing time in average for each subscription within any broker and roughly-equal forwarding time of either a subscription or a relevant notification, T_{e_2} can be approximately equal to $T_{e_1} + T_p + T_f + T_p + T_f + T_f + T_f$. Thus, in this case, $T_{e_2} \simeq T_{e_1} + 2T_p + 4T_f$.

These assumptions can be somewhat accurate when (1) the subscriptions have identical – or common and closely-identical – filter set, (2) the technology used is quite similar in subscribing and routing, and (3) the path distance between a broker and another one is closely equal. The path distance would include the communication media and the number of underlying hops.

5.4.3.8 General Crude Estimation of Performance.

In general, referring to the aforementioned assumptions and crudely estimating the end-to-end path delay can lead to the following general assumptions:

- **Assumption 1.** The propagation delay of a local subscription "*sub(o-SID,F)*" = propagation delay of a local notification "$notify(N_F)$" $= T_l$, with the assumption that both have the same size. In other words, T_l is the access time of a subscription to its local broker or the travel time of a notification from a broker to its local subscriber. In the case that the size of a local subscription is different form that of the relevant notification, the propagation throughput is the same for both of them, where T_l is considered the travel time per unit-size.

- **Assumption 2.** The intra-broker processing time (i.e. the average internal buffering, storing/registering, and matching time) for each subscription in each broker is T_p. Since the broker tends to initiate the matching process for many subscriptions (up to 255) simultaneously, T_p mostly consists of the average registration time for each subscription in a broker. Since the registration of a new subscription of existing (already registered) filters generally takes less time than the registration of a new subscription of new filters, the more common filters that subscriptions would contain, the smaller T_p would relatively become (Section 5.3.3.4). The processing time of a single publication and the building time of relevant notifications (prior to their routing or delivery) are also assumed to be included in T_p. In other words, the term $255T_p$ actually refers to the total processing and matching time of a single publication and 255 subscriptions within a broker, in addition to the time consumed in building relevant notifications for matched subscriptions.

- **Assumption 3.** The lasting operation of the broker in the *(nP,nS)* mode, i.e. the

time needed for the broker to switch from the *(nP,nS)* mode to the *(nP,oS)* mode, is another factor that affects T_p. This switching time depends on the number of subscriptions that are under process. In the best-case condition, the broker makes an immediate transition from the *(nP,nS)* mode to the *(nP,oS)* mode right after the registration process of a single subscription. In other words, either the condition $S = S_{max}$ or the condition $S_a = 0$ becomes into effect right after registering the subscription into the CRT (Figure 5.18). In this best-case, the total processing time of all subscriptions (actually a single subscription) is T_p. However, the value of T_p includes the processing and matching time that covers both *(nP, nS)* and *(nP, oS)* operation modes for one subscription. In the worst-case condition, the two conditions $S < S_{max}$ and $S_a \neq 0$ hold (Figure 5.18) for 254 subscriptions that do not contain any common filters. In this worst-case, the broker does not switch to the *(nP,oS)* mode upon the processing end of a single received subscription. Instead, the broker continues to operate in the *(nP,nS)* mode until $S = S_{max} = 255$ prior to switching to the *(nP,oS)* mode. Accordingly, the worst-case value of the total processing time for all subscriptions would be at least $255T_p$ – i.e. at least 255 times higher than the best-case condition.

- **Assumption 4.** The inter-broker routing time (i.e. the forwarding transmission and propagation time from broker to broker) of either a subscription or a notification is T_f, with the assumption that both have the same size. Referring to the example of Section 5.4.3.7, this assumption means that (Broker 3 $\xrightarrow{sub(\{o-SID2,B3\},F)}$ Broker 2) = (Broker 2 $\xrightarrow{forward(\{o-SID2,B3,B2\},N_F)}$ Broker 3) = (Broker 2 $\xrightarrow{sub(\{o-SID2,B3,B2\},F)}$ Broker 1) = (Broker 1 $\xrightarrow{forward(\{o-SID2,B3,B2,B1\},N_F)}$ Broker 2). In the case that the size of a local subscription is different form that

of the relevant notification, the same assumption holds for the routing throughput, where T_f is considered the inter-broker routing time per unit-size. In real content-based networks, T_f may not be equal for a subscription and a notification or for each pair of neighboring brokers, because the link delay between a broker and its neighboring one depends not only on the size of either a subscription or a notification, but also on the technology used and the number of access points crossed. In the case that content-based networks operate on top of IP networks, the inter-broker link may involve one or multiple underlying IP routers. However, this assumption can be somewhat accurate when the routing path distance (including the communication media and the number of underlying hops) between each two pairs of brokers is closely equal.

Taking these four assumptions into consideration, and referring to the aforementioned example of Figures 5.26 and 5.25 and the corresponding analysis of Section 5.4.3.7.1, one can crudely estimate the end-to-end path delay for the subscription s_1:

$$T(s_1) = (T_l + T_p + T_l)_{B1} = 2T_l + T_p$$

while the end-to-end path delay for subscription s_2 is:

$T(s_2)$ = Total subscription routing time(s_2) + Total notification routing time(s_2), where:

Total subscription routing time$(s_2) = (T_l + T_p + T_f)_{B3} + (T_p + T_f)_{B2} + (T_p)_{B1}$, and

Total notification routing time$(s_2) = (T_f)_{B1} + (T_f)_{B2} + (T_l)_{B3}$.

Accordingly,

$$T(s_2) = (T_l + T_p + T_f)_{B3} + (T_p + T_f)_{B2} + (T_p)_{B1} + (T_f)_{B1} + (T_f)_{B2} + (T_l)_{B3}.$$

Therefore,

$$T(s_2) = 2T_l + 3T_p + 4T_f.$$

5.4.3.8.1 Generalization. We can infer from this example that for a subscription s that needs to visit "four" brokers (one additional broker) until it reaches a matching publication and subsequently receives a notification,

$T(s) = (T_l + T_p + T_f)_{B4} + (T_p + T_f)_{B3} + (T_p + T_f)_{B2} + (T_p)_{B1} + (T_f)_{B1} + (T_f)_{B2} + (T_f)_{B3} + (T_l)_{B4} = 2T_l + 4T_p + 6T_f$,

where $B4$ represents the fourth broker in the network and the local broker for s. Therefore, in general, for a subscription s that needs to visit "n" brokers until it reaches a matching publication and subsequently receives a notification,

$$T(s) = 2T_l + nT_p + 2(n-1)T_f.$$

The communication between two neighboring brokers may have taken place using different technologies. For example, the communication medium can be fiber optics with a speed in the range of Terabits per second, or Ethernet with a speed of 10 Gbps, 1 Gbps, 100 Mbps, or 10 Mbps. Moreover, the communication between two neighboring brokers may involve multiple underlying physical IP routers. Therefore, T_f can be highly variable. Similarly, the communication between a broker and its local subscriber can be highly variable. For example, the access link can be DSL, wired Ethernet, fixed wireless, or mobile wireless. The communication may involve a single or multiple underlying IP routers as well. Therefore, in general, $T_f \neq T_l$. However, in special cases such as small monitoring system in a factory or a small wireless sensor network, the communication time between a subscriber and a broker or between a broker and a neighboring broker can be close. If we assume that $T_f \simeq T_l$ in such systems, we can see that (referring to s_2 in the example above),

$$T(s_2) \simeq 6T_l + 3T_p.$$

$$\text{i.e. } T(s_2) \simeq 3T(s_1).$$

This result reveals that, in such small systems, the total performance involved in processing a subscription and notifying the corresponding subscriber crudely depends on the number of network brokers visited by either the subscription or the notification, in addition to T_l and T_p. In this example, there are three brokers visited by the subscription s_2, where only one broker is visited by the subscription s_1. Accordingly, for a subscription s that visits n brokers until it matches a publication and subsequently receives a notification, $T(s) \simeq nT(s_1)$.

5.4.4 Routing Scenarios and Corresponding Performance

This section provides crude estimation of the performance for specific examples of different routing scenarios, based on the aforementioned assumptions and the crude estimation of the performance described in Sections 5.4.3.7 and 5.4.3.8.

5.4.4.1 Scenario 1

Figure 5.27 shows the simplest routing scenario, where a local subscription matches a publication in its local broker.

The crude estimation of the end-to-end path delay from sending the subscription up to receiving the corresponding notification is as follows, while referring to the assumptions of Section 5.4.3.8:

$$T(s) = T_l + T_p + T_l = 2T_l + T_p.$$

This result assumes that there was an immediate transition from the *(nP,nS)* mode to the *(nP,oS)* mode right after the registration process of the subscription. In other words, either the condition $S = S_{max}$ or the condition $S_a = 0$ becomes into effect right after registering the subscription into the CRT (Figure 5.18). Either one of these conditions

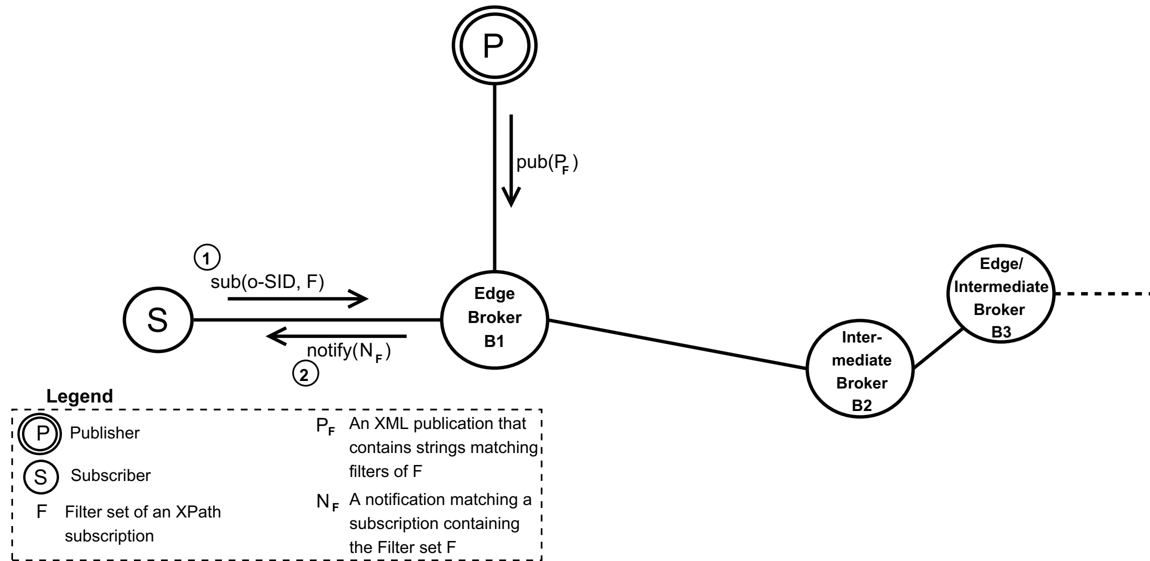

Figure 5.27: Scenario 1-a: A Subscription Matching a Publication in its Local Broker.

illustrates the best-case condition for T_p, which includes the processing and matching time that covers both *(nP,nS)* and *(nP,oS)* operation modes.

5.4.4.1.1 Scenario 1-b Figure 5.28 shows the same broker network of scenario 1-a, but 255 local subscriptions reach one broker in which they match available publications.

In this scenario, the broker operates in its *(nP,nS)* mode until $S = S_{max} = 255$ prior to switching to the *(nP,oS)* mode. Since all 255 subscriptions are local, and providing they do not contain any common filters, the conditions that lead to the worst case delay may occur.

In this worst-case, there is no saving in registration time because there are no commonalities among the subscriptions. Moreover, all these subscriptions only match the last available publication (i.e. when $P = P_{max}$) in the *(nP,oS)* mode, after they have been exposed to all available publications. In such a case, the broker has to build a relevant notification for each matched subscription, and actually notifies corresponding

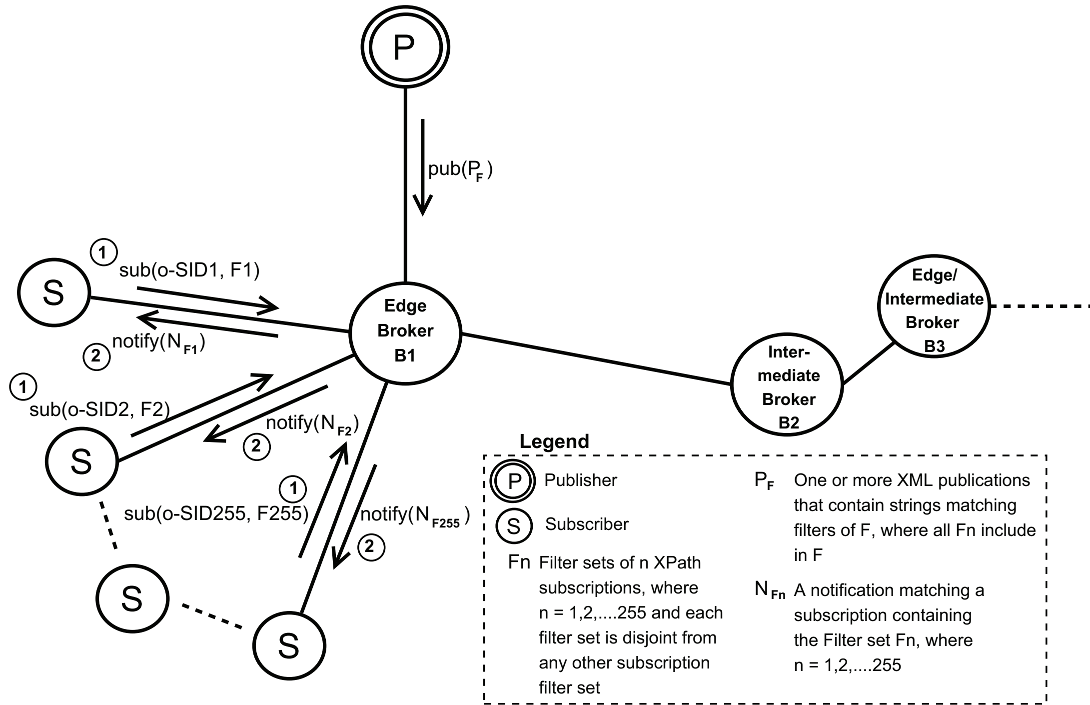

Figure 5.28: Scenario 1-b: 255 Subscriptions Matching Publications in the Local Broker.

subscribers in its *(oP,oS)* mode. Accordingly, the total end-to-end delay for all subscriptions is at least: $255(2T_l + T_p)$, which is at least 255 times higher than the best-case condition. The average end-to-end path delay for each subscription can still be approximately $2T_l + T_p$, where T_p covers the processing and matching time for each subscription during the *(nP,nS)* and *(nP,oS)* modes.

However, the first received subscription reaches the broker the first and lasts the most among all subscriptions in the *(nP,nS)* and *(nP,oS)* modes. Thus, this subscription consumes T_l to reach the local broker, but its processing time is $255T_p$. Furthermore, if matching this subscription occurs the last among all subscriptions, the corresponding subscriber receives a notification the last. Thus, the broker consumes $255T_l$ to notify this particular corresponding subscriber. Accordingly, the worst-case value of $T(s)$ occurs for

this particular subscription among all subscriptions, where:

$$T(s_{worst}) = T_l + 255T_p + 255T_l = 256T_l + 255T_p.$$

5.4.4.2 Scenario 2

Figure 5.29 shows the routing scenario for a subscription that matches a publication after it has visited two brokers in the network.

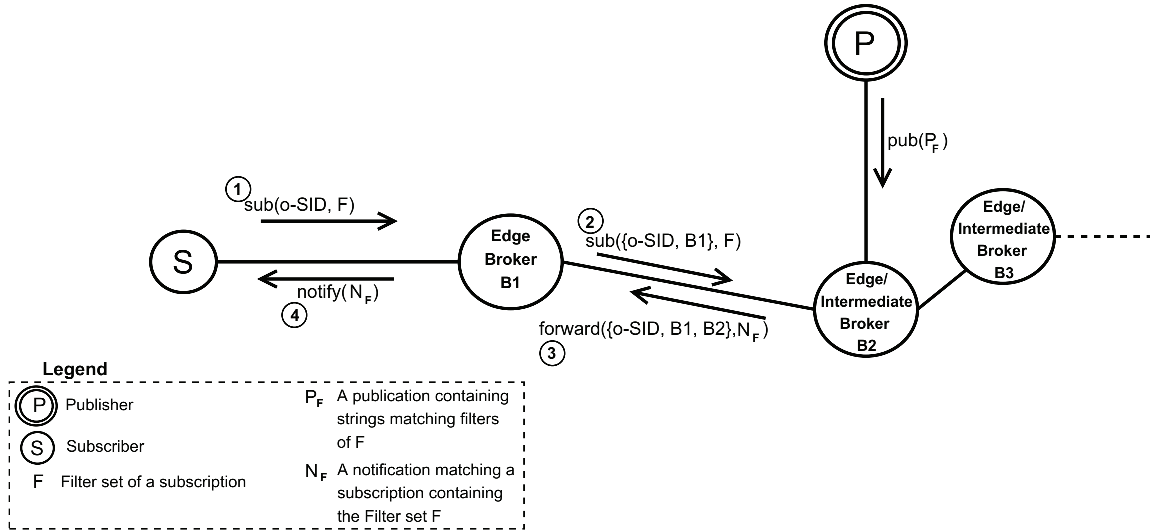

Figure 5.29: Scenario 2: A Subscription Matching a Publication After Having Crossed a Path of Two Brokers.

The end-to-end path delay from sending the subscription up to receiving the corresponding notification can be estimated as:

$$T(s) = (T_l + T_p + T_f)_{B1} + (T_p)_{B2} + (T_f)_{B2} + (T_l)_{B1} = 2T_l + 2T_p + 2T_f.$$

In small systems having $T_f \simeq T_l$,

$$T(s) = 4T_l + 2T_p = 2(2T_l + T_p).$$

In such systems, $T(s)$ of scenario 2 is twice $T(s)$ of scenario 1-a.

5.4.4.3 Scenario 3

Figure 5.30 shows the routing scenario for a subscription that matches a publication after it has visited three brokers in the network.

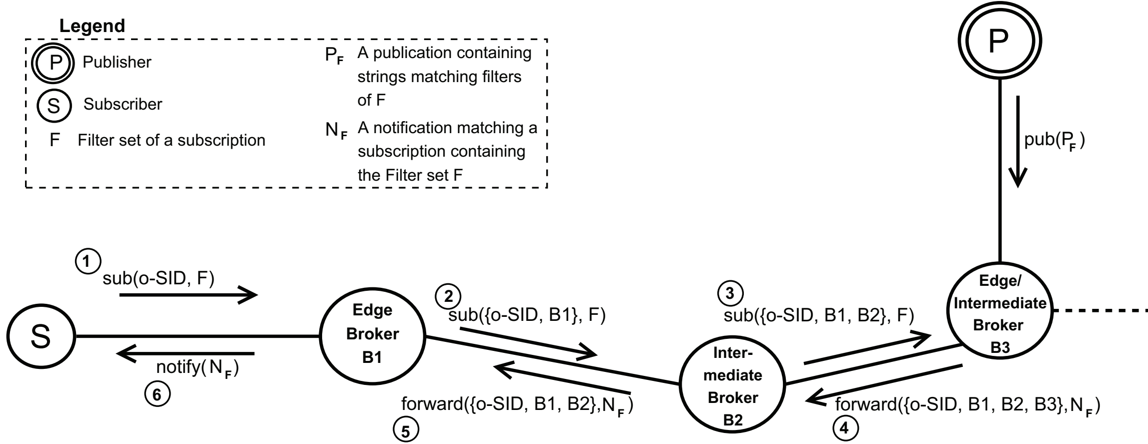

Figure 5.30: Scenario 3: A Subscription Matching a Publication After Having Crossed a Path of Three Brokers.

The end-to-end path delay from sending the subscription up to receiving the corresponding notification can be estimated as:

$$T(s) = (T_l + T_p + T_f)_{B1} + (T_p + T_f)_{B2} + (T_p)_{B3} + (T_f)_{B3} + (T_f)_{B2} + (T_l)_{B1}.$$

Therefore,

$$T(s) = 2T_l + 3T_p + 4T_f.$$

In small systems having $T_f \simeq T_l$,

$$T(s) = 6T_l + 3T_p = 3(2T_l + T_p).$$

In such systems, $T(s)$ of scenario 3 is three times $T(s)$ of scenario 1-a.

5.4.4.4 Scenario 4

Figure 5.31 shows the routing scenario for two subscriptions containing identical filters, where each matches the same publication after it has visited two brokers in the network.

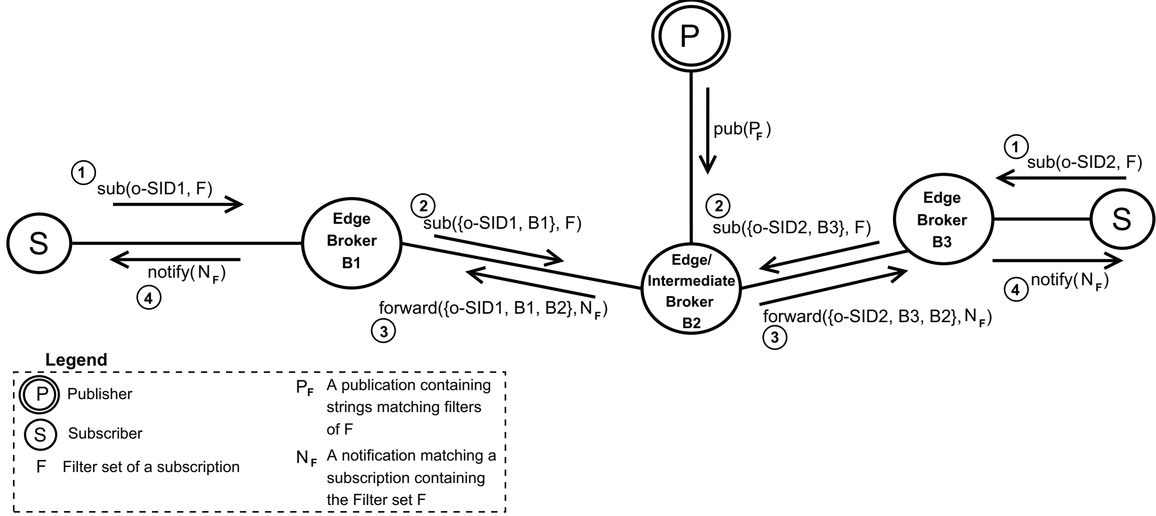

Figure 5.31: Scenario 4: Two Subscriptions Requesting Identical Data Through Two Different Brokers.

The end-to-end path delay from sending each subscription up to receiving the corresponding notification can be estimated as:

$$T(s_1) = (T_l + T_p + T_f)_{B1} + (T_p)_{B2} + (T_f)_{B2} + (T_l)_{B1} = 2T_l + 2T_p + 2T_f.$$

$$T(s_2) = (T_l + T_p + T_f)_{B3} + (T_p)_{B2} + (T_f)_{B2} + (T_l)_{B3} = 2T_l + 2T_p + 2T_f.$$

This scenario is similar to scenario 2. However, T_p in scenario 4 may be smaller than T_p in scenario 2, because one subscription contains the same filter set as that of the other subscription. Therefore, providing that the broker first registers the subscription s_1 in its CRT, the subscription s_2 takes less registration time than the subscription s_1.

In small systems having $T_f \simeq T_l$,

$$T(s_1) = 4T_l + 2T_p = 2(2T_l + T_p) = T(s_2).$$

In such systems, $T(s)$ of scenario 4 for each subscription is twice $T(s)$ of scenario 1-a.

5.4.4.5 Scenario 5

Figure 5.32 shows the routing scenario for 255 subscriptions containing identical filters, where each matches the same publication after it has visited two brokers in the network. However, only one of these subscriptions is local for the broker $B3$, while all other subscriptions are local for the broker $B1$. None of the subscriptions has matched any publication in its local broker.

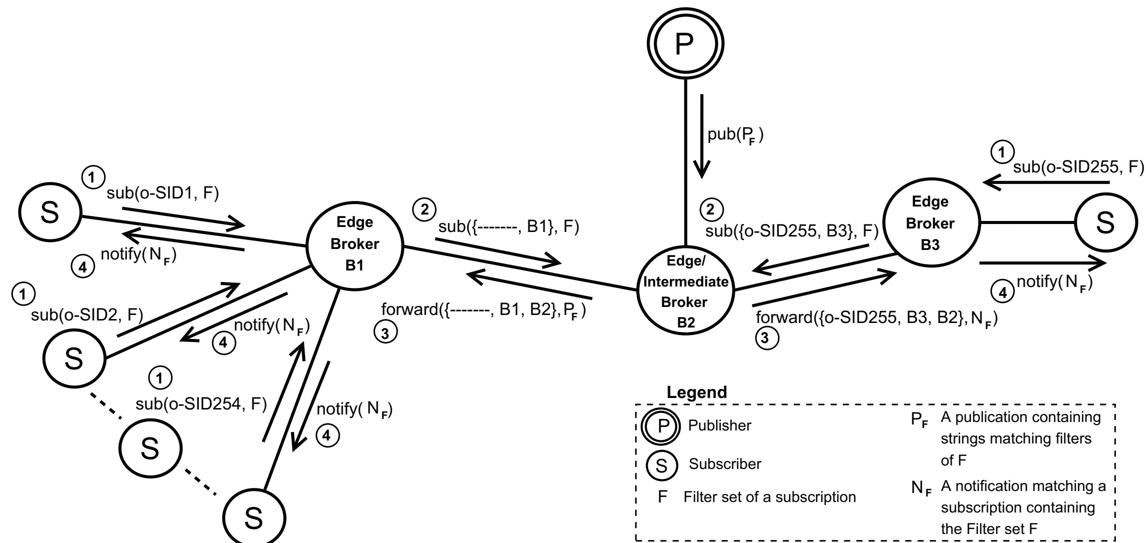

Figure 5.32: Scenario 5: Many Subscriptions Requesting Identical Data Through the Same Broker.

In this scenario, $B1$ consults its S_u to find out that 254 subscriptions have not matched any available publication. Since S_u is highly dense and all these subscriptions have identical common filters, the High Commonality Degree (HCD) must have been reported (Section 5.3.3.4 and Figure 5.14). In this case, $B1$ only forwards one mismatched subscription to the broker $B2$, instead of forwarding all the 254 mismatched subscriptions, in order to reduce unnecessary routing traffic. This single subscription contains the filter

set F but its o-SID is of the form {———,$B1$}; i.e. it has a *null SID* (see line 38 in Figure 5.20).

The publication P_F that is available in $B2$ matches the subscriptions coming from $B1$ as well as the subscription coming from $B3$. Accordingly, $B2$ issues and routes a corresponding notification to $B3$, but it routes the publication P_F itself down to $B1$ (see lines 13 to 17 in Figure 5.22 and Section 5.4.3.5.2.1).

The end-to-end delay for the path, which starts when a local subscriber sends the subscription s_{255} to $B3$ and ends when the local subscriber receives a corresponding notification, can be estimated as in scenario 2:

$$T(s_{255}) = (T_l + T_p + T_f)_{B3} + (T_p)_{B2} + (T_f)_{B2} + (T_l)_{B3} = 2T_l + 2T_p + 2T_f.$$

Regarding the other 254 subscriptions, the end-to-end path delay includes the following seven subsequent steps:

1. The local subscribers of $B1$ send their subscriptions (s_1 up to s_{254}) to $B1$.
2. $B1$ processes the received 254 subscriptions (the processes include buffering, registering, and matching).
3. $B1$ forwards a single subscription (a sample subscription) to $B2$.
4. $B2$ processes this single subscription and identifies the corresponding matched publication P_F.
5. $B2$ routes P_F out to $B1$.
6. $B1$ simultaneously matches P_F against all available subscriptions including the already-registered 254 subscriptions.
7. $B1$ notifies all local subscribers who had initiate the 254 subscriptions.

Providing that all 254 local subscriptions simultaneously reach $B1$, and all 254 notifications simultaneously reach the local subscribers, the end-to-end path delay can be estimated as follows:

$T(s_{1TO254}) = (T_l)_{B1} + (254T_p)_{B1} + (T_f)_{B1} + (T_p)_{B2} + (T_f)_{B2} + (T_p)_{B1} + (T_l)_{B1}$.

Thus, $T(s_{1TO254}) = 2T_l + 256T_p + 2T_f$.

However, the current architecture of the broker buffers new subscriptions in a subsequent way. Moreover, the NPU is able to simultaneously notify four local subscribers at most. Taking into account these limitations, we consider that the 254 subscriptions would reach the buffer of their local broker in a delay of $254T_l$, while the broker can notify its 254 local subscribers in a delay of $\frac{254}{4}T_l$. Accordingly, we can make better informed estimation for $T(s_{1TO254})$ as follows:

$$T(s_{1TO254}) = 254T_l + 256T_p + 2T_f + \frac{254}{4}T_l = 317.50T_l + 256T_p + 2T_f.$$

Note that the terms $(T_f)_{B1}$, $(T_p)_{B2}$, and $(T_f)_{B2}$ are evaluated for the single forwarded subscription. Moreover, the term $(T_p)_{B1}$ has just the value of T_p, rather than $254T_p$, because there is no need to re-register any of the already-registered 254 subscriptions. Actually, the value $(T_p)_{B1}$ includes the processing time of the publication P_F in $B1$ and the matching time of this publication against the already-registered subscriptions. Therefore, $(T_f)_{B1} + (T_p)_{B2} + (T_f)_{B2} + (T_p)_{B1}$ represents the latency needed before any subscription receives a notification. As a result, $T(s_{latency}) = 2T_p + 2T_f$. Thus,

$$T(s_{1TO254}) = 317.50T_l + 254T_p + T(s_{latency}).$$

Because of this latency, the first local subscriber receives a notification in a delay of

$$T(s) = 2T_l + T_p + T(s_{latency}) = 2T_l + 3T_p + 2T_f.$$

The term $2T_l$ comes from the subscription access time T_l to reach $B1$ and the notification travel time T_l from $B1$ to the corresponding subscriber.

The first four local subscribers may concurrently receive notifications in a delay of

$$T(s_{four}) = 2T_l + 4T_p + T(s_{latency}) = 2T_l + 6T_p + 2T_f.$$

The term $2T_l$ does not change into $4T_l$ because of the notification concurrency, providing that four local links were free of congestion prior to the notification process.

As mentioned earlier, for all 254 subscribers to receive notifications with maximum concurrency, $T(s_{1TO254}) = 317.50T_l + 254T_p + T(s_{latency})$, where $T(s_{latency}) = 2T_p + 2T_f$ providing that a total of two brokers are involved. In the case that a third broker is involved, the latency increases with $1T_p + 2T_f$, where $1T_f$ for forwarding the subscription to that broker, $1T_p$ for processing the subscription in that broker, and $1T_f$ for forwarding the notification from that broker. Therefore, the latency in a three-broker path is $T(s_{latency(3)}) = 3T_p + 4T_f$. If the path includes a total of n brokers, the total latency would be $T(s_{latency(n)}) = nT_p + 2(n-1)T_f$.

For scenario 5, with a total of two brokers in the path, the average end-to-end path delay for each of these 254 subscriptions is then estimated as:

$$T(s_{av}) = \tfrac{317.50}{254}T_l + \tfrac{256}{254}T_p + \tfrac{2}{254}T_f \simeq 1.25T_l + T_p + \tfrac{1}{127}T_f.$$

This result indicates that when many subscriptions have commonalities, the average estimated delay of a round-trip subscription-routing-notification path can be less than that of scenario 1-a (where only one local subscription is involved with $T(s) = 2T_l + T_p$), providing that T_f is not much larger than T_l. The question that we may raise here is: how large T_f can be, so that the system can stay efficient with a delay that is less or equivalent to that of scenario 1-a?

We can know the answer by equating $1.25T_l + T_p + \frac{1}{127}T_f$ to $2T_l + T_p$. In this case, $\frac{1}{127}T_f = 2T_l - 1.25T_l = 0.75T_l$. Thus, $T_f = 95.25T_l$.

Therefore, when many subscriptions have commonalities in a broker, the broker efficiently performs its tasks, comparing to scenario 1-a (even if routing to a neighboring broker is involved), as far as $T_f \leq 95.25T_l$ on average.

In small systems having $T_f \simeq T_l$, $T(s_{latency}) = 2T_p + 2T_f \simeq 2T_p + 2T_l$. Thus,

$$T(s_{1TO254}) = 317.50T_l + 254T_p + T(s_{latency}) \simeq 317.50T_l + 254T_p + 2T_p + 2T_l.$$

As a result, $T(s_{1TO254}) \simeq 319.50T_l + 256T_p$.

In such systems, the average delay of a round-trip subscription-routing-notification path can be estimated as:

$$T(s_{av}) \simeq 1.25T_l + T_p.$$

This average estimated delay is less than that of s_{255} (which is similar of scenario 2) and even less than that of scenario 1-a. This result reveals the high performance that the broker can deliver when many subscriptions have commonalities.

5.4.5 Link Management

The discussion of scenario 5 would raise questions about the number of external links through which the broker may simultaneously send or receive information, whether such information consists of receiving/routing publications, subscriptions, and notifications, or receiving topology messages. Besides the links to local subscribers, where communication can be performed as is in a Local-Area Network (LAN), the number of external links via which the broker communicates with other brokers can affect the overall performance.

The hardware architecture of the broker can simultaneously handle six external links to communicate with other brokers in the network, as follows:

1. One external link through which new publications arrive.

2. Another link through which the broker receives topology messages.

3. a link for the broker to receive remote subscriptions.

4. a link through which the broker forwards subscriptions.

5. a link through which the broker routes notifications.

6. a link for the broker to receive notifications from the up-stream broker.

Figure 5.33 shows the six external links that the software/hardware interface manage.

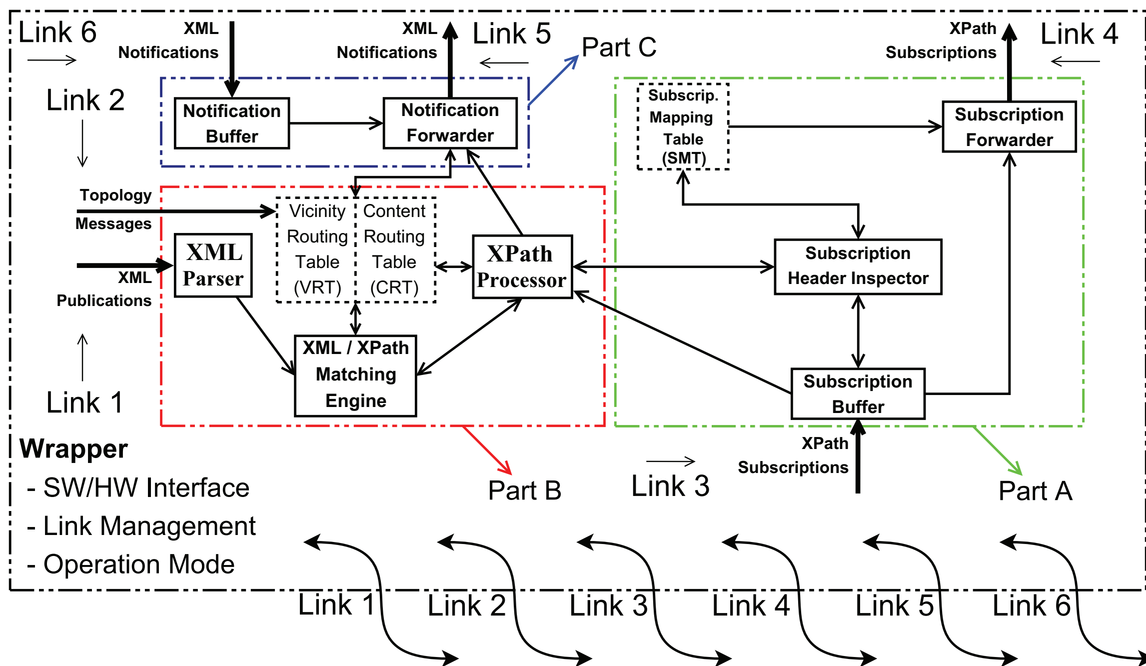

Figure 5.33: Software Wrapping and Link Management for the Software/Hardware Interface of the Reconfigurable Hardware Broker.

In the case that less than six links are physically available, and high amount of traffic is to be handled, the incoming and outgoing messages may have to compete for accessing the available links. The tasks of the software/hardware interface consist of reconfiguring

the broker, controlling the operation mode mechanism, and managing the access to available external links. In the case that only a single link exists, the software/hardware interface has no other choice but to sequentially perform these tasks.

5.4.6 Network Scopes

In small networks, the entity that knows the topology of the network can guide a mismatched subscription, by means of the VRT, toward a matching publication that is available in another broker, as discussed in Section 5.4.4. In the case that two different publications are available to two different brokers, and theoretically match the same subscription, the entity that knows the topology of the network can interfere by "asking" a broker to route a matched (rather than mismatched) subscription out to another neighboring broker. In other words, a broker may exceptionally route a "matched" subscription to another broker because of the potential possibility of the presence of additional matching publications. In such a case, the broker routes the subscription with *null SID*, as was the case in scenario 5 for a mismatched subscription having commonalities with many other mismatched subscriptions (Section 5.4.4.5).

However, in large networks, this kind of interference from the entity that knows the topology of the network may not happen. A large network may consist of many overlay brokers. Consider one broker $B1$ located near one edge of a large network, while $B2$ is a broker located far away near another edge of this network.

A subscription s_1 containing the filter set F may match a publication P_F that is available to $B1$, while another subscription s_2 containing the filter set $F1$ that covers filters of F may match a publication P_{F1} that is available to $B2$. The subscriber of s_1 may receive the corresponding notification from $B1$ where P_F is available. Since $F1$ covers filters of F, i.e. filters of $F \in F1$, s_1 matches P_{F1} as well. The subscriber of s_1

should theoretically receive a notification from $B2$ where P_{F1} is available, if s_1 has the opportunity to reach $B2$.

The large size of the network may hinder the entity that knows the topology of the network from helping s_1 recognize its way to $B2$. We assume that this possibility would not occur when the network is not so large, because the entity knowing the topology of small networks can easily help in finding the right route for a subscription. In other words, this entity is able to know at least the topology of a "scope" of the network. At this point, we state the following definition:

Definition. *Let B_B be a set of brokers in a network. A "scope" of the network consists of a subset of brokers $B_{Bs} \subset B_B$, where a topology entity is able to know the topology enclosing all brokers $\in B_{Bs}$ and their corresponding links, and this entity is able to communicate with each broker $\in B_{Bs}$.*

According to this definition, we may assume that the scope involve just a few brokers that are not geographically far from each other. Thus, a scope would be considered a small network.

Intuitively, a large network consists of multiple scopes connected together through at least one broker. Consider again the two publications P_F that is available in a broker located in one scope and P_{F1} that is available in a broker located in another scope, where filters of $F \in F1$. A subscription s_1 matching P_F can find the other matching publication P_{F1}, if inter-scope subscription routing occurs. We refer this subscription routing to "publication solicitation", since the subscription that is matched in a scope "solicits" matching publications in another scope. Accordingly, we state the following definition:

Definition. *Let B_B be a set of brokers in a large network. "Publication solicitation" is an action of routing a sample of matched subscriptions (i.e. with null SID) for the*

purpose of finding additional matching publications available in another scope of B_B.

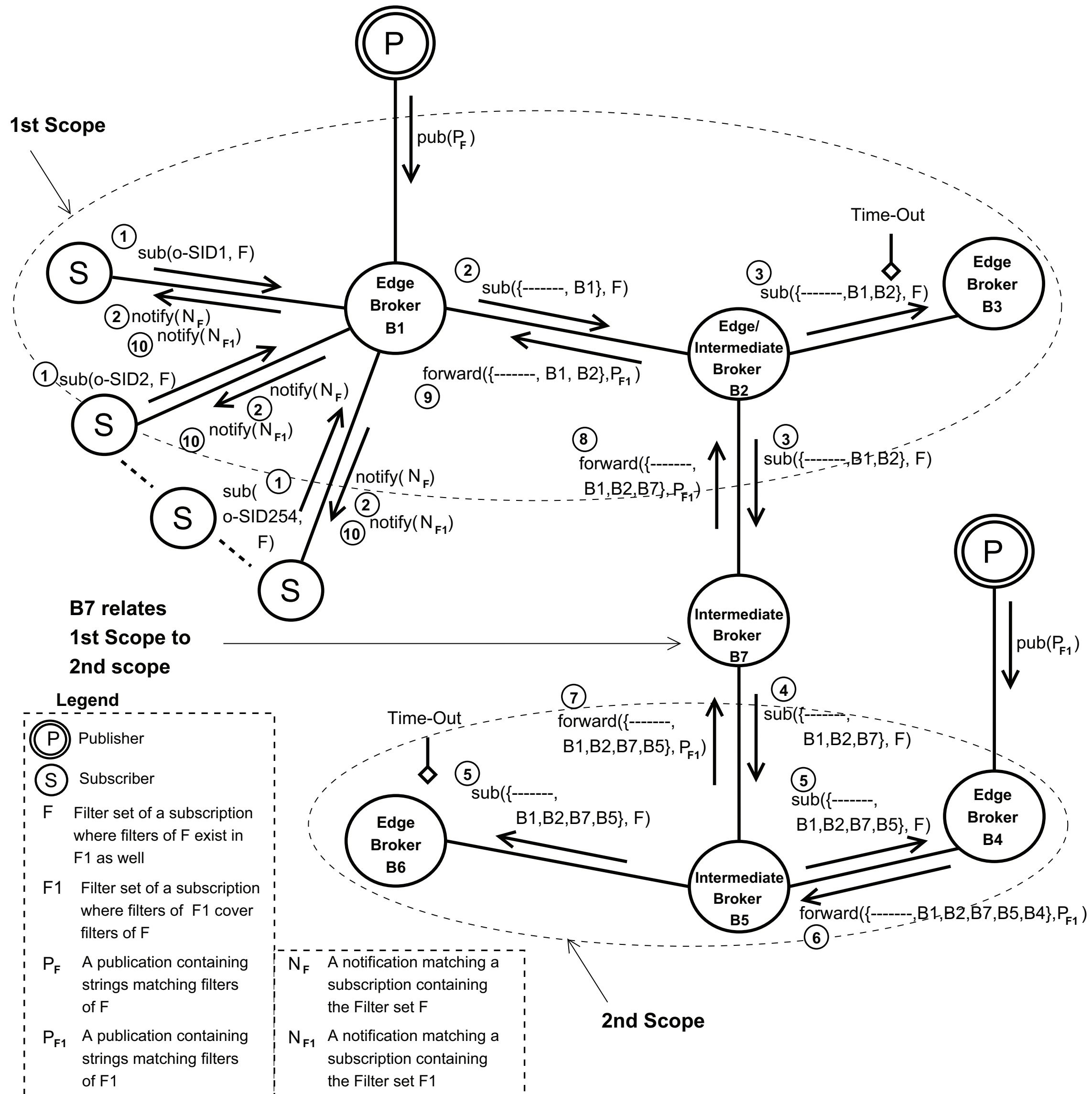

Figure 5.34: Illustration of Network Scopes: Example of Two Different Publications Located in Two Different Scopes Matching Many Subscriptions Located in One Scope.

Figure 5.34 illustrates network scopes and publication solicitation. In this figure, there is one broker ($B7$) connecting the two scopes that are presumably far from each other (e.g. from a geographical point of view). Many common subscriptions that are

available to $B1$ match the publication P_F, and their subscribers subsequently receive the corresponding notifications N_F. Even though $F \subset F1$ of the publication P_{F1} that is available in a broker ($B4$) in another scope, these users would not receive notifications issued in $B4$ unless inter-scope "publication solicitation" occurs.

Therefore, a sample subscription with a *null SID* takes the route to $B2$. The VRT of $B2$ would not have the knowledge of where to route forward the sample subscription, because the entity that knows the topology of the first scope could only identify a matching publication P_F in $B1$. Unable to route the subscription back to $B1$ due to the nature of the algorithm SRoA (line 14 in Figure 5.24), the broker $B1$ floods the sample subscription out in a publication solicitation "mission" as shown in Figure 5.34. Accordingly, both $B3$ and $B7$ receive the sample subscription as *sub(.......,B1,B2,F)*.

The VRT of $B3$ does not guide this sample subscription for any route. Therefore, the subscription will eventually be timed-out and completely removed. $B7$ connects both scopes but does not have the routing information about neither filters of F nor filters of $F1$. Since the sample subscription comes from $B2$, the broker $B7$ may only route this subscription to $B5$.

In the second scope, the VRT of $B5$ would have information about filters of $F1$ but not about filters of F as indicated in the sample subscription. Since $F \subset F1$, it is possible for $B5$ to directly route the sample subscription to $B4$. If - for any reason - this knowledge does not exist in the VRT of $B5$, this broker floods the sample subscription to both $B4$ and $B6$ as *sub(.......,B1,B2,B7,B5,F)*. Therefore, the subscription will eventually be timed-out in $B6$ and completely removed, but it effectively matches the publication P_{F1} in $B4$. Therefore, $B4$ responds to $B5$ with the publication P_{F1} rather than a corresponding notification. This publication takes the route back to $B1$ in the first scope through the brokers $B5$, $B7$, and $B2$. The broker $B1$ can now match P_{F1} against available

subscriptions, and subsequently notify all local users whose subscriptions contain the filter set F. Each of these users eventually receives two different notifications in response to a single subscription.

5.4.6.1 Scope Performance Estimation

Each of the 254 subscriptions in Figure 5.34 may receive a notification N_F in an estimated time of:

$$T(s) = (T_l + T_p + T_l)_{B1} = 2T_l + T_p.$$

This estimation of the end-to-end path delay assumes a subsequent notification occurrence when only one local link is available at a time. In the case that at least four local links are available, the broker can concurrently notify up to four local subscribers. Thus,

$$T(s) = T_l + T_p + \tfrac{1}{4}T_l = 1.25T_l + T_p.$$

When the second scope becomes involved, the additional end-to-end path delay does not take into account T_l again, because each user only subscribes with a single subscription. Thus, the additional end-to-end path delay $T(sample)$ from sending the sample subscription from $B1$ up to receiving back P_{F1} and simultaneously matching P_{F1} against subscriptions in $B1$, without taking into account the notification process for all local subscribers with the notification N_{F1}, can be estimated as:

$T(sample) = (T_f)_{B1} + (T_f)_{B2} + (T_f)_{B7} + (T_f)_{B5} + (T_p)_{B4} + (T_f)_{B4} + (T_f)_{B5} + (T_f)_{B7} + (T_f)_{B2} + (T_p)_{B1}$.

Note that the time spent in simultaneously matching P_{F1} against subscriptions in $B1$ is $(T_p)_{B1}$ rather than $(254T_p)_{B1}$, because the subscriptions were already registered before initiating the publication solicitation mission. Then, this estimated delay is:

$$T(sample) = 8T_f + 2T_p.$$

The delay $T(sample)$ is actually the latency needed before sending the notification N_{F1} to any of the local 254 subscribers.

Notifying all local subscribers with no concurrency can be accomplished in a delay of $T(n) = 254T_l$. Therefore, the total end-to-end path delay $T(s1)$ for each of the 254 subscriptions up to receiving both notifications N_F and N_{F1} is:

$$T(s1) = T(s) + T(sample) + \tfrac{1}{254}T(n).$$

Thus,

$$T(s1) = (2T_l + T_p) + (8T_f + 2T_p) + (T_l) = 3T_l + 3T_p + 8T_f = 3.(T_l + T_p + \tfrac{8}{3}T_f).$$

The total time needed for all subscribers to receive both notifications N_F and N_{F1} ($T_{s(all)}$) takes into account the latency $T(sample)$ only once. Therefore,

$$T_{s(all)} = 254.(2T_l + T_p) + 1.(8T_f + 2T_p) + 254.(T_l) = 762T_l + 256T_p + 8T_f \simeq 254.(3T_l + T_p + \tfrac{8}{254}T_f).$$

The average delay $T_{s(all)(av)}$ for each subscription is then:

$$T_{s(all)(av)} = \tfrac{762}{254}T_l + \tfrac{256}{254}T_p + \tfrac{8}{254}T_f \simeq 3T_l + T_p + 0.03T_f.$$

Notifying all local subscribers with maximum concurrency can be accomplished in a delay of $T(n) = \frac{254}{4}T_l$. Thus, in this case,

$$T(s1) = (1.25T_l + T_p) + (8T_f + 2T_p) + 0.25T_l = 1.5T_l + 3T_p + 8T_f = 3.(\tfrac{1}{2}T_l + T_p + \tfrac{8}{3}T_f).$$

Again, $T_{s(all)}$ takes into account the latency $T(sample)$ only once. Therefore, $T_{s(all)} = 254.(1.5T_l + T_p) + 1.(8T_f + 2T_p) = 381T_l + 256T_p + 8T_f \simeq 254.(1.5T_l + T_p + \frac{8}{254}T_f)$. The average delay $T_{s(all)(av)}$ for each subscription, with maximum notification concurrency, is then:

$$T_{s(all)(av)} \simeq 1.5T_l + T_p + 0.03T_f.$$

Note that the assumption of $T_f \simeq T_l$ should not apply to network scopes, because a large network is involved.

5.4.6.1.1 Scope Performance Summary. Referring to Figure 5.34, we can conclude from the discussion above the following:

1. In the first scope, each subscriber receives the notification N_F in an estimated delay of $2T_l + T_p$, with no notification concurrency.

2. In the first scope, all 254 subscribers receive the notification N_F in a total estimated delay of $254.(2T_l + T_p)$, with no notification concurrency.

3. In the first scope, each subscriber receives the notification N_F in an estimated delay of $1.25T_l + T_p$, with maximum notification concurrency.

4. In the first scope, all 254 subscribers receive the notification N_F in a total estimated delay of $254.(1.25T_l + T_p)$, with maximum notification concurrency.

5. In the case of publication solicitation that spans two scopes, each subscriber receives the notification N_{F1} in an estimated delay of $3.(T_l + T_p + \frac{8}{3}T_f)$, with no notification concurrency.

6. In the case of publication solicitation that spans two scopes, all 254 subscribers receive the notification N_{F1} in an estimated delay of $254.(3T_l + T_p + 0.03T_f)$, with no notification concurrency.

7. In the case of publication solicitation that spans two scopes, each subscriber receives the notification N_{F1} in an estimated delay of $3.(\frac{1}{2}T_l + T_p + \frac{8}{3}T_f)$, with maximum notification concurrency.

8. In the case of publication solicitation that spans two scopes, all 254 subscribers receive the notification N_{F1} in an estimated delay of $254.(1.5T_l + T_p + 0.03T_f)$, with maximum notification concurrency.

Chapter 6

Experimental Results of the Hardware SCBXP Technique

This chapter discusses the experiments that have been conducted to test the hardware-based SCBXP technique described in Chapter 4 and implemented on an FPGA device. All tests and corresponding results consider the SCBXP as a stand-alone XML parser entity that is independent of the broker.

6.1 Implementation Note

The SCBXP is implemented on Altera's FPGA Stratix EP1S40F780C5 device which is based on a 1.5-V core voltage and a 130-nm process technology [17]. The size of either the CAM, the X-RAM, or the P-RAM is 32 bits x 256 words. Thus, a 1-KB skeleton can configure the CAM for parsing at least a 1-KB segment of the XML data. We utilize small-size memory modules for the parsing technique primarily to ensure that the SCBXP functions efficiently in restricted memory resources equipment such as mobile devices.

The X-RAM and P-RAM are chosen from the dual-port memory configurable embedded modules of Altera's Stratix FPGA. However, since the Altera's original embedded CAM consumes too many resources from the Stratix chip, the SCBXP employs the CAM design described in Section 5.3.3.2. Note that Altera's recent tool versions no longer support embedded CAMs for Stratix FPGAs.

6.2 FPGA Resources Utilization and Performance

The chip area utilized as well as the performance may vary according to a number of factors, including the target device itself, the silicon process technology of the device, the clock frequency, and the severity of delay constraints assigned to the design. For example, the fitting report for the SCBXP implementation shows that the logic elements (LEs) utilized are 29002 out of 41250 i.e. 70% of the total LEs available on the Stratix chip, while the maximum clock frequency achieved is 33.37 MHz. In another attempt, the LEs utilized become 29879 out of 41250 i.e. 72% of the total LEs available on the chip, while the maximum clock frequency reached is 38.02 MHz. The value of the achieved clock frequency depends on the design and the target device technology. Note that the Stratix FPGA board does not support a clock frequency higher than 50 MHz, providing that the clock is generated from the on-board resources.

6.3 How to Improve Performance?

Looking at the critical path, we can improve the design performance (and thus obtain a higher clock frequency) if a faster CAM is achievable. As a proof-of-concept, we have resynthesized the SCBXP architecture after omitting the CAM and its associated

logic circuitry, targeting the same Stratix device. The corresponding results indicated an increase in achievable clock frequency to 66.68 MHz. In FPGAs, a CAM can be implemented using one of three different approaches [76]: Register-based, RAM-based, and Look-Up-Table (LUT)-based. While the CAM of SCBXP is mapped to registers and LUTs of the Stratix device, some transistor-level techniques can improve the CAM in terms of performance and power consumption. Such techniques can be implemented using CMOS customized hardware, as is seen in ASICs [91]. In such chips, the whole design can be improved, since the support of higher clock frequencies and less internal delays is quite possible.

Since newer FPGA boards may support higher clock frequencies, we temporarily changed the target FPGA chip from the old Stratix device to the newer Stratix IV EP4SE530H35C2ES chip whose core voltage is 1.35 V and process technology is 40 nm [19]. Providing that an FPGA board supporting this device is not available in the research laboratory that hosts the experiments, the purpose of this temporary change is simply to estimate the improvement that would be achieved. The fitting report shows in this case that the logic utilization of the SCBXP design is just 7% of the total chip area, while the maximum clock frequency that can be reached is 49.47 MHz. This achievement would lead to an approximately 30% improvement in performance.

Recently, the Terasic DE2-115 FPGA board [115] has become available in the research laboratory. This board supports the Cyclone IV E FPGA device (EP4CE115F29C7), whose core voltage is 1.2 V and process technology is 60 nm [15]. The SCBXP, successfully implemented onto this chip, could operate in this case with a clock frequency of 50 MHz without any error.

6.4 Testing with Hardware Interface

An interface capable of simultaneously accessing both memory modules can load XML data and access parsed results as well with no interruption. To emulate such an interface, we build a hardware framework for testing the SCBXP and measuring the acquired performance (Figure 6.1). The hardware interface loads 32-bit of raw XML data at a time into the X-RAM dual-port memory using a single port. Using the other port, the SCBXP reports the memory location that is being accessed to the interface.

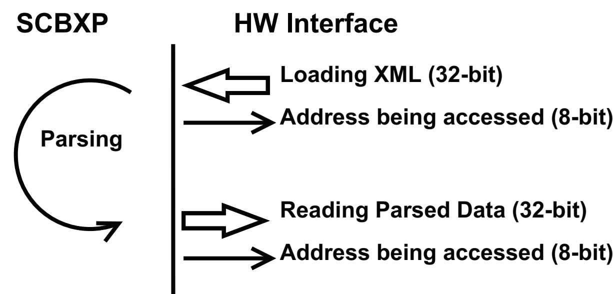

Figure 6.1: Hardware-Testing Framework.

The hardware-testing framework can calculate the processing time, starting from the time at which the X-RAM gets the first 32-bit string of the XML document, up to the end of writing all parsed data into the P-RAM. Note that the SCBXP begins the actual reading of the loaded XML stream after five clock cycles from the beginning of the XML loading process.

The size of the XML documents used for parsing ranges from 216 bytes up to 15228 bytes, where documents larger than 1024 bytes (1 KB) have to be loaded in chunks into the X-RAM of SCBXP during the parsing process. Each chunk consists of 1024-byte XML data filling the entire memory. Thus, the 15228-byte XML document is loaded in 15 chunks into the memory. During the process of chunk loading, the interface tracks the current memory location accessed by the SCBXP hardware in order to avoid overwriting

Table 6.1: Case I: CAM is configured with a single skeleton for all loaded XML documents.

XML Document Size (Bytes)	Nb. of parsed bytes per 1 clk cyc.	Clock: (tested on FPGA Time (μs.)	25MHz Stratix chip) Thrput. (Mbps)	Clock: (tested on FPGA Time (μs.)	33.33MHz Stratix chip) Thrput. (Mbps)	Clock: 50MHz (not tested) Thrput. (Mbps)	Clock: 100MHz (not tested) Thrput. (Mbps)
15228(15ch.)	2.145	283.9	429	212.9	572	858	1716
6696(7ch.)	2.140	125.1	428	93.8	570	856	1712
2904(3ch.)	2.129	54.5	425	40.9	567	851	1703
1912(2ch.)	2.131	35.8	426	26.9	568	852	1705
1324(2ch.)	2.111	25.0	422	18.8	562	844	1689
1008	2.091	19.2	418	14.4	557	836	1673
980	2.305	17.0	461	12.7	614	922	1844
932	2.099	17.7	419	13.3	559	839	1679
916	2.058	17.8	411	13.3	548	823	1646
832	2.248	14.8	449	11.1	599	899	1798
768	2.092	14.6	418	11.0	557	837	1674
684	2.171	12.6	434	9.4	578	868	1737
536	2.038	10.5	407	7.9	543	815	1630
236	2.247	4.2	449	3.1	599	899	1798
224	2.133	4.2	426	3.1	568	853	1706

the stored XML data that has not yet been processed (Figure 6.1). Using a similar method, the interface can avoid reading parsed data from a location currently accessed by the SCBXP.

The performance is measured for the two cases identified and discussed in section 4.5.

6.4.1 Case I

In this test, one skeleton only configures the CAM, while the X-RAM receives multiple chunks of XML data from the interface. As mentioned earlier, this case applies to many XML-based communication applications. As per the analysis of section 4.5.1, the processing time for each document is: $T_{proc} = T_{parse} + [(T_C + 10)/N]$ clock cycles, where N is the number of XML chunks.

Table 6.1 shows that the SCBXP can parse at least two bytes of XML data on average in each clock cycle. Using an operational clock frequency of just 25-MHz on the Stratix FPGA, the processing throughput exceeds 400 Mbps for XML documents of different

sizes. For Stratix clock frequency of 33.33 MHz, the processing throughput exceeds 543 Mbps and may surpass 600 Mbps for some XML documents. For comparison purposes, Table 6.1 includes calculations of the presumable throughput for 50-MHz and 100-MHz clock frequencies, which are available in ASICs and modern FPGAs.

6.4.2 Case II

The test of Case II employs the same XML documents of Case I; but in this test, a new skeleton reconfigures the CAM for each XML document to be parsed. From the analysis of section 4.5.2, $T_{proc} = T_{parse} + T_C + (10/N)$ clock cycles.

Table 6.2 depicts the parsing results obtained in this test case. For small XML documents, the parsing rate indicates low throughput. However, the larger the XML documents are, the higher the corresponding parsing throughput is. For XML documents larger than 15 KB, the throughput may exceed 400 Mbps for an operational clock of just 25-MHz, and can surpass 533 Mbps for a 33.33 MHz clock. For comparison purposes, Table 6.2 includes calculations of the presumable throughput for 50-MHz and 100-MHz clock frequencies, which are available in ASICs and modern FPGAs.

6.4.3 Comparison between Case I and Case II

To visually compare the parsing performance in Cases I and II, we include, in Figure 6.2, a graphical representation of the throughput measured in both cases using a 25-MHz clock. Figure 6.2 shows that the throughput in Case II gradually approaches that of Case I as the XML document gets larger. For documents larger than 15 KB, the parsing throughput in both cases tends to be steady and is presumed following roughly the same approaching curves. Thus, the graphical curve of Case II is bound by that of Case I.

Table 6.2: Case II: CAM is configured with a new skeleton for each loaded XML document.

XML Document Size (Bytes)	Nb. of parsed bytes per 1 clk cyc.	Clock: (tested on FPGA Time (μs.)	25MHz Stratix chip) Thrput. (Mbps)	Clock: (tested on FPGA Time (μs.)	33.33MHz Stratix chip) Thrput. (Mbps)	Clock: 50MHz (not tested) Thrput. (Mbps)	Clock: 100MHz (not tested) Thrput. (Mbps)
15228(15ch)	2.001	304.4	400	228.3	533	800	1601
6696(7ch)	1.839	145.6	367	109.2	490	735	1471
2904(3ch)	1.547	75.0	309	56.3	412	619	1238
1912(2ch)	1.356	56.4	271	42.3	361	542	1085
1324(2ch)	1.162	45.5	232	34.1	309	464	929
1008	1.014	39.7	202	29.8	270	405	811
980	1.045	37.5	209	28.1	278	418	836
932	0.974	38.2	194	28.7	259	389	779
916	0.957	38.2	191	28.7	255	382	765
832	0.943	35.2	188	26.4	251	377	754
768	0.873	35.1	174	26.3	232	349	698
684	0.827	33.0	165	24.8	220	330	661
536	0.691	31.0	138	23.2	184	276	553
236	0.382	24.7	76	18.5	101	152	305
224	0.363	24.6	72	18.5	96	145	290

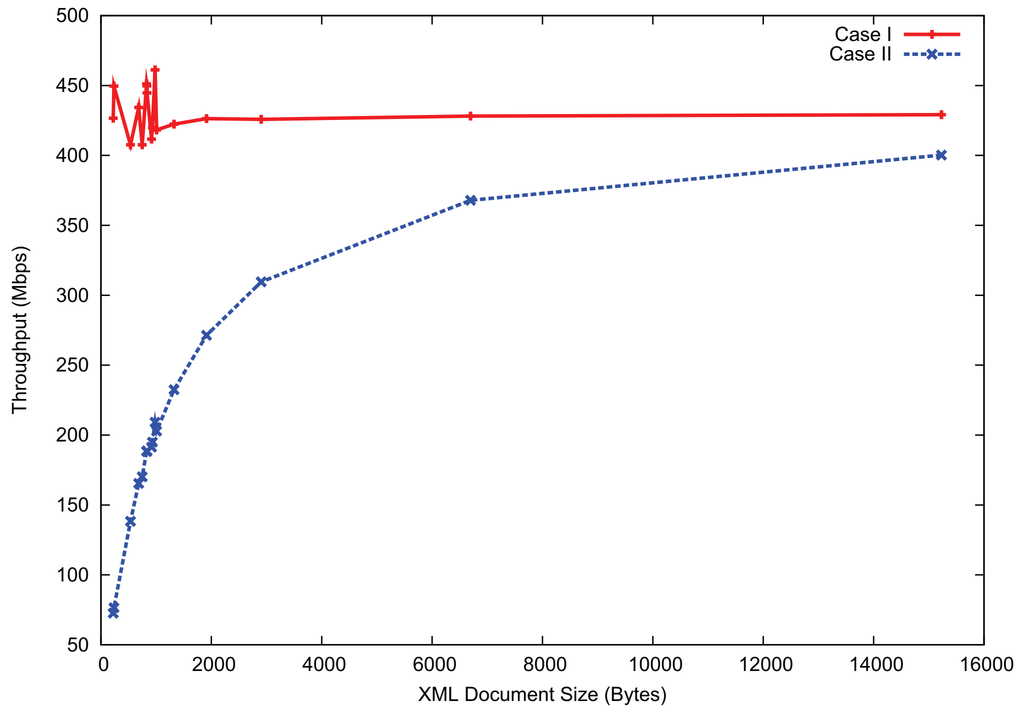

Figure 6.2: Graphical Representation of Throughput Results for Cases I and II, for a 25-MHz clock.

6.5 Testing with Software/Hardware Interface

When a software interface drives the SCBXP, some concurrency features have the potential to diminish. In particular, the software interface can either load XML data or read parsed data, but it cannot perform both tasks simultaneously. Thus, by emulating Case I and Case II above with a software/hardware interface instead of a hardware interface, we can compare the performance results collected when the SCBXP is driven by either interface. Accordingly, we have developed additional hardware and software blocks to interface the embedded Avalon bus [12] of the Stratix FPGA board. Section 7.6 states further information about the method of interfacing hardware implementations with the Avalon bus. Figure 6.3 illustrates the software/hardware framework used when testing the SCBXP.

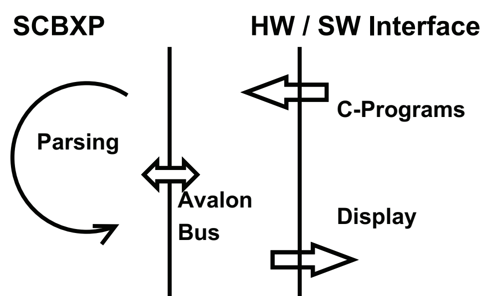

Figure 6.3: Software/Hardware-Testing Framework.

In the software test environment of Altera's Nios II [13], C-programs configure the CAM with a skeleton through the Avalon bus, load XML documents, and drive the SCBXP implemented on the Stratix FPGA. Then, a C-program reads the parsed results from the SCBXP memory (P-RAM). The total processing time of only one XML chunk should be: $T_{proc} = T_C + T_L + T_{parse} + T_A$, where T_C is the CAM configuration time, T_L is the XML loading time, T_{parse} is the actual parsing time, and T_A is the parsed data access time (see section 4.5). Then, the software/hardware framework runs four tests,

which are outlined in the following cases.

6.5.1 Case A

In this case, the software/hardware interface configures the CAM first, and then it sequentially loads the entire XML document. Afterwards, the parsing process is repeated one million times. Thus, the CAM configuration and the XML loading are performed in the first iteration only. At the end of the last iteration, the interface reads the parsed results. To calculate the parsing duration of the document, the total parsing time is divided by one million iterations. Therefore, the processing time for each document is: $T_{proc} = (T_C/N) + (T_L/N) + (N * T_{parse}/N) + (T_A/N) = T_{parse} + [(T_C + T_L + T_A)/N]$, where $N = 1$ million. This procedure marginalizes the duration of the CAM configuration, the XML loading, and the parsed results access. Thus, Case A is comparable to Case I. As an example of obtained results, an XML document of 980 bytes is parsed in Case A in 26 μs. This parsing time translates into a throughput of 301.538 Mbps for a 33.33-Mhz clock frequency. In Case I, the throughput of an XML document of the same size was 614 Mbps for the same clock. This indicates that the performance in Case A drops about 51% compared to Case I that uses a hardware interface.

6.5.2 Case B

In Case B, the software/hardware interface configures the CAM with a skeleton for every 1-KB chunk of XML data. This test runs one million times, similarly to Case A. However, the interface loads an XML chunk only in the first iteration, and reads the parsed results only by the end of the last iteration. The total parsing time (i.e. duration of mainly the two processes: the CAM configuration and the XML parsing) is divided by one million

iterations, to get the total parsing time for one iteration. Therefore, the processing time for each document is: $T_{proc} = (N * T_C/N) + (T_L/N) + (N * T_{parse}/N) + (T_A/N) = T_C + T_{parse} + [(T_L + T_A)/N]$, where $N = 1$ million. In other words, Case B marginalizes the effect of the XML loading and the parsed results access; but it takes into account the CAM configuration and the parsing process, which occur in all iterations. Therefore, Case B is comparable to Case II. The obtained results illustrate a significant decrease in performance compared to the hardware interface. For example, an XML document of 980 bytes is parsed in Case B in 1.52 ms, which translates into a throughput of 5.157 Mbps for a 33.33-Mhz clock. For the same document size, the throughput was 278 Mbps in Case II for the same clock. This indicates that the performance in Case II is 54 times higher than the performance in Case B.

6.5.3 Case C

In Case C, the software/hardware interface performs all driving tasks sequentially except for the task of accessing the parsed data. Therefore, the CAM is configured with a skeleton for every 1-KB chunk of XML data. Subsequently, each chunk is loaded entirely in the memory of SCBXP, then the actual parsing process begins. This sequence is performed one million times. At the end of the last iteration, the interface reads the parsed results. The total parsing time (i.e. duration of mainly three processes in sequence: the CAM configuration, the XML loading, and the actual XML parsing) is divided by one million iterations. Therefore, the processing time for each document is: $T_{proc} = (N * T_C/N) + (N * T_L/N) + (N * T_{parse}/N) + (T_A/N) = T_C + T_L + T_{parse} + (T_A/N)$, where $N = 1$ million. In this case, an XML document of 980 bytes is parsed in 4.227 ms, which translates into a throughput of 1.854 Mbps for a 33.33-MHz clock. This indicates that the performance in Case II is 150 times higher than that of Case C.

6.5.4 Case D

In this case, the software/hardware interface sequentially performs all driving tasks. Therefore, in addition to the sequence of tasks performed in Case C, the interface accesses all parsed data at the end of parsing each chunk. This test repeats the sequence of tasks one million times. The total parsing time (i.e. duration of four processes in sequence: the CAM configuration, the XML loading, the actual XML parsing, and the parsed data access) is divided by one million iterations. Therefore, the processing time for each document is: $T_{proc} = (N * T_C/N) + (N * T_L/N) + (N * T_{parse}/N) + (N * T_A/N) = T_C + T_L + T_{parse} + T_A$, where $N = 1$ million. In this case, a 980-byte XML document is parsed in 5.645 ms, which translates into a throughput of 1.388 Mbps for a 33.33-Mhz clock. This indicates that the performance in Case II is 200 times higher than that of Case D.

6.6 Results Conclusion

In summation, the hardware interface beats the software/hardware interface in all cases of the tests, as illustrated in Table 6.3. In particular, in Case D, the performance degrades significantly. This degradation in performance occurs even though the actual parsing process still runs in hardware, because all driving tasks of Case D are performed in sequence using the software/hardware interface - which leads to the consumption of longer processing times.

Running a SAX-based software implementation in Java over a 3.4-GHz Pentium-4 CPU, we have obtained a throughput that went from 41.67 Kbps (for an XML document of 224 bytes) to 360.53 Kbps (for an XML document of 15228 bytes). For a 980-byte document, the throughput was 125.78 Kbps. Table 6.4 illustrates the speedup of the

Table 6.3: SCBXP Performance: Hardware Interface versus Software/Hardware Interface.

Speedup of Case I (HW) over Case A (SW-HW) (Times)	Speedup of Case II (HW) over Case B (SW-HW) (Times)	Speedup of Case II (HW) over Case C (SW-HW) (Times)	Speedup of Case II (HW) over Case D (SW-HW) (Times)
2	54	150	200

SCBXP at 33.33-MHz clock over the software, using the results of Case I and Case II; while Table 6.5 illustrates the speedup of the SCBXP over the software, using the results of the software/hardware test cases.

Table 6.4: SCBXP Performance: Hardware Interface versus SAX-based Software.

Speedup of Case I (HW) over Software XML: From 224 B. to 15228 B. (Times)	Speedup of Case II (HW) over Software XML: From 224 B. to 15228 B. (Times)
13630 to 1586	2300 to 1478

Table 6.5: SCBXP Performance: Software/Hardware Interface versus SAX-based Software.

Speedup of Case A (SW-HW) over Software XML: 980 B. (Times)	Speedup of Case B (SW-HW) over Software XML: 980 B. (Times)	Speedup of Case C (SW-HW) over Software XML: 980 B. (Times)	Speedup of Case D (SW-HW) over Software XML: 980 B. (Times)
2397	41	14	11

If an FPGA/ASIC technology newer than the Stratix family of FPGAs is used, higher performance can be achieved. For example, the processing throughput can exceed 1600 Mbps (and may reach 1800 Mbps) for large documents for a clock frequency of 100 MHz in such newer technologies (Tables 6.1 and 6.2). Note that the CMOS process technology of the Stratix FPGA family is 130 nm, while 28-nm process technologies are already available for the newest FPGAs.

6.7 Advantages and Limitations

6.7.1 Advantages

The following points can be highlighted, especially in terms of scalability and concurrency:

- **Small and large XML documents, one skeleton.** All XML documents, whether small or large, are efficiently parsed with a rate of around two bytes per clock cycle, if one common skeleton for these documents does fit entirely in the CAM, with no need for further reconfiguration. The case of using one common skeleton for a large XML document implies that so many repetitive words appear in the document, and that these words can be matched against just one or a few entries in the skeleton.

- **Small and large XML documents, multiple small skeletons.** If multiple skeletons can co-locate (and entirely fit in) the CAM for parsing multiple XML chunks, CAM reconfiguration is not needed to parse these chunks. In this case, the parsing rate of around two bytes per clock cycle still holds true. However, identical entries from two (or more) different co-located skeletons are not allowed so as to avoid multiple hits in the CAM during the matching process.

- **Large XML documents, large skeletons.** If a new skeleton must configure the CAM for parsing each XML document (or each 1-KB XML chunk), XML documents need to be larger than 15 KB to be efficiently parsed with a rate of around two bytes per clock cycle.

  Note that using a larger CAM module may seem like a solution to fit many skeletons (or a large skeleton) without the need of CAM reconfiguration; but this solution

does not provide remarkable benefits for parsing XML documents larger than 15 KB, as Figure 6.2 illustrates. In addition, large CAM modules may lead to high-power dissipation, which would not be suitable for mobile devices.

- **Concurrency.** The tasks of loading XML data into memory and pushing the stored data into FIFOs are concurrently accomplished, with a latency of just five clock cycles, thanks to the utilization of a dual-port memory (X-RAM). Similarly, the tasks of writing the parsed data into memory and accessing the stored data using the outside interface are concurrently accomplished, with a latency of just five clock cycles, thanks to the utilization of another dual-port memory (P-RAM). Furthermore, a hardware interface can load XML data into the X-RAM and access the parsed data stored in the P-RAM concurrently as well.

- **Natural well-formedness and partial validation.** By using CAM-based skeletons, the SCBXP technique efficiently parses XML data, and naturally ensures that full well-formedness and partial validation are respected, with no need for extra processing.

6.7.2 Limitations

We have identified several limitations associated with the SCBXP technique as follows:

- **Encoding.** The encoding standard supported by the SCBXP can be ASCII as a subset of UTF-8. To support other encoding styles, such as UTF-16, the SCBXP technique needs to be expanded to include an appropriate encoder.

- **XML validation.** While XML well-formedness can be fully supported, XML validation is only fully supported for XML characters/strings that match any of

the CAM contents. To support XML validation for an entire XML document, an XML schema is an option that can be used, at considerable processing costs, but it is not currently supported by the SCBXP technique.

- **Frequent configuration of the CAM.** It may happen that many "too different" small XML documents (< 1 KB) are to be consecutively parsed. In this case, the CAM must be configured with a new skeleton derived from each small XML document (one possible situation in Case II). This frequent configuration of the CAM deteriorates the overall performance of XML parsing, as shown in Table 6.2, where the parsing rate indicates less than one byte per clock cycle. For documents larger than 1 KB, the parsing rate can be at least one byte per clock cycle; and for documents larger than 15 KB, the parsing process regains its normal rate of at least two bytes per clock cycle.

6.8 SCBXP Conclusion

The SCBXP, described in Chapter 4, represents an efficient hardware technique for XML parsing. The technique utilizes FIFOs, a state machine, and a CAM to align the XML data properly with special XML characters prior to storing the resulting parsed data. The parallel features of the SCBXP architecture allow various tasks to be done concurrently.

The implementation results discussed in this chapter indicate that at least two bytes of XML data can be processed in each clock cycle. The SCBXP can achieve high-performance parsing, ensuring that XML is fully well-formed and partially valid, even in the presence of limited memory resources. While processing throughput of 0.53 to 0.61 Gbps can be reached on the Stratix FPGA using a 33.33-MHz clock, higher throughput is achievable for faster clocks in newer technologies. In addition, the implementation

results reveal that the interface, through which the SCBXP is driven, has a considerable impact on the overall performance. With a hardware interface, the SCBXP technique can accelerate XML parsing several hundreds of times, compared to a software-based XML parser.

Chapter 7

Experimental Results of the Hardware XML/XPath Broker/Router

This chapter discusses the experiments and relevant results conducted to test the whole XML/XPath broker/router that is implemented on an FPGA device. The architecture of the broker, thoroughly described in Chapter 5, has undergone two phases of development and implementation. Accordingly, the test framework of the architecture in each phase is different. The next paragraphs indicate the developmental achievements for each phase and the corresponding test framework.

- **Phase 1**: The hardware development of the broker in this phase covers most of the units including the SPU, PMU, and NPU as well as the SMT and CRT (Chapter 5). Moreover, the XML publication parser makes part of the broker and utilizes the proposed hardware-based SCBXP technique (Chapters 4 and 6). However, the subscription buffer of the SPU is simple (Figure 5.5). In this phase, the broker is

able to parse XML publications, process XPath subscriptions (i.e. run in hardware all buffering, mapping, and registering tasks), match subscriptions against publications, and push local notifications to the output. The main goals of the tests in this phase consist of measuring the performance of the broker, with parallel loads of publications and subscriptions, and without any involvement of software or any specific interfacing protocol. Therefore, the hardware-testing framework for this phase aims at directly driving the broker's hardware, receiving the broker's output, and measuring the performance of subscription processing in particular as well as the total processing performance. All experiments of this phase drive and test the broker targeting the Altera Stratix FPGA device [17].

- **Phase 2**: In this phase, the VRT, the content-based routing mechanisms, and the operation mode algorithm become integrated in the hardware of the broker. However, the subscription buffer of the SPU becomes larger and readily able to contribute to the subscription routing task (Figure 5.6). In this phase, the testing data sequentially drives the broker through a hardware-based Avalon interface that emulates the Avalon bus specifications [12]. However, two different interfacing methods would drive data to and from the X2CBBR: (1) Data directly passes through this hardware interface to the broker back and forth, and (2) Software data that is based on the Nios II embedded processor drives the broker through the hardware interface. The second method effectively is a way of driving the broker through a software/hardware interface. Using each of these two methods at a time, Phase 2 experiments test the performance of the X2CBBR, where the content-based routing mechanisms and the operation mode algorithm become into effect. During all tests of this phase, the Altera Cyclone IV E FPGA device [15]

hosts the X2CBBR implementation.

In this chapter, Section 7.1 up to Section 7.5 present the tests related to Phase 1 and the relevant results. All the tests related to Phase 2 and their corresponding results come up thereafter starting from Section 7.6.

7.1 Experimental Setup for Phase 1 Tests

The experiments of Phase 1 make use of a hardware interface that exploits parallel loading. The main goal of this interface is to simultaneously load an XML publication, configure the CAM of the SCBXP with the relevant XML skeleton that is required for efficient parsing, and buffer XPath subscriptions. Moreover, the interface should be able to read notifications without affecting the loading task of subsequent data. Emulating such an interface, the hardware framework of Figure 7.1 graphically illustrates the general driving method utilized in testing the broker and measuring the acquired performance.

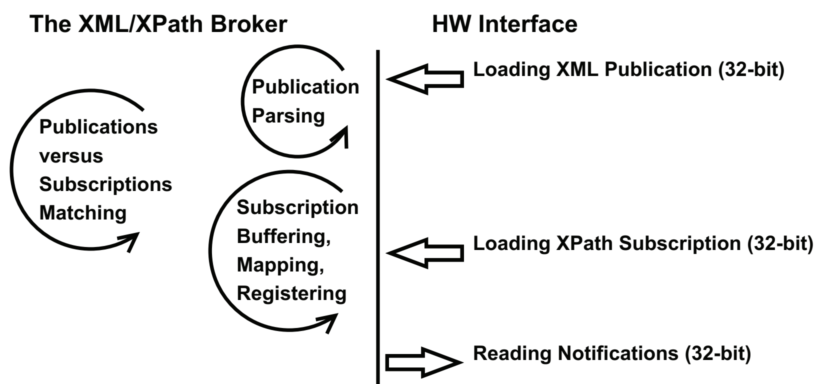

Figure 7.1: Hardware-Testing Framework for the Broker.

This hardware interface can load 32-bit of raw XML publication data at a time into the SCBXP memory, configure the SCBXP's CAM with the relevant skeleton (32-bit a time as well), and simultaneously buffers 32-bit of raw XPath subscription data into the

SPU's subscription buffer. The hardware broker performs all needed processes, utilizing the algorithms and the concurrent features described in Chapter 5. A separate link is used by the interface to read the corresponding notifications from the output of the broker. The ModelSim simulator tool [79] monitors the publication/subscription loading processes, as well as the signals resulted from all Broker's internal processes.

There are two sets of XPath subscriptions employed in the experiments of Phase 1: XPath Set1 and XPath Set2. Each subscription of XPath Set1 has relatively large-size filters, while each subscription of XPath Set2 has low-size filters. Moreover, all filters of each subscription in both sets do completely match some content in an XML published document. The inclusion of such sets in the experiments allows for measuring the impact of the filter size in subscriptions on the broker performance, where all available subscriptions have common filters that must be simultaneously detected.

XPath Set1. In this set, each filter of an XPath subscription contains at least 5 ASCII characters and at most 8 ASCII characters (between 40 bits and 64 bits inclusive). Moreover, each subscription is made of at least two filters and at most four filters (between 80 bits and 256 bits inclusive), in addition to a 20-byte (160-bit) overhead (Figure 5.7). For example, an XPath subscription */stock/share/* possesses two filters (*stock* and *share*), where each filter is 40-bit wide (5 ASCII characters). Since the dual-port memory of the "Simple Subscription Buffer" (Section 5.3.2.1 and Figure 5.5) may contain 1 KB (i.e. 1024 bytes or 8192 bits), this set contains 24 raw XPath subscriptions of different size (each of average size: $\simeq$341 bits) in order to almost fill the buffer. The interface gradually loads additional sets of 24 subscriptions to the simple subscription buffer via one port of the memory in order to test the broker with up to almost 255 subscriptions.

XPath Set2. In this set, each XPath subscription is made of just one filter, besides the 20-byte overhead, where each filter only contains 4 ASCII characters (32 bits). Thus,

this set may contain 42 raw XPath subscriptions (each of exact size: 192 bits) that almost fill the buffer. To test the broker with up to almost 255 subscriptions, the interface gradually loads additional sets of 42 subscriptions via one memory port of the simple subscription buffer.

XML. An XML published document of 980 bytes is parsed using the proposed hardware-based SCBXP technique (Chapter 4 and Section 5.3.3.5). The matching engine of the broker performs its XML/XPath matching tasks, as described in Section 5.3.3.6, for the two XPath sets. In Phase 1 experiments, only one XML document is utilized to test the broker, and all subscriptions of each set have common filters matching some content of the XML file.

7.2 Stratix FPGA Resources Utilization

The broker is implemented on Altera's FPGA Stratix EP1S40F780C5 device [17] using the Altera Quartus II synthesis and placement and route tool [16].

The corresponding implementation results indicate that the broker consume 99% of the total Logic Elements (LEs) available on this small 130-nm chip (i.e. 40,835 out of 41,250 LEs), with clock support of an achievable 50-MHz frequency. The XML parser alone consumes 22,221 LEs, which represent 53.8% of the total LEs (or 54.4% of the LEs consumed by the broker). Since the research laboratory that hosts the experiments does not physically have a larger FPGA at the testing time, a re-synthesis of the whole design targets (as a "proof of concept") the Altera's Stratix III EP3SL150F1152C2 [18] that may contain almost three times more logic elements. In this case, the logic utilization scores 37% of the total chip area, and the maximum clock frequency jumps to around 80 MHz. For more recent FPGAs, a higher clock frequency is quite possible to achieve.

Since additional hardware components have become available in Phase 2, the whole broker area became bigger. Therefore, the broker could not fit into the Stratix FPGA chip. Accordingly, the Phase 2 tests, presented later in this chapter, utilize the Altera's Cyclone IV E FPGA device that hosts the whole implementation of the X2CBBR with the support of a higher clock frequency.

7.3 Power Dissipation on Stratix FPGA

The total thermal power dissipation of the design implemented on the Stratix device is 2034.39 mW, which includes the core dynamic (579.93 mW), core static (729.90 mW), and I/O thermal power dissipation (724.57 mW). Reducing the "static" thermal power dissipation is also possible for recent FPGAs.

7.4 Results Discussion for Phase 1

Table 7.1 illustrates detailed results of processing each of the two sets of XPath subscriptions, gradually multiplied up to the near-maximum capacity of the SMT i.e. $\simeq$255 subscriptions. In particular, Figure 7.2 represents a graphical view of the XPath processing time as well as the total processing time, while Figure 7.3 represents a graphical representation of the throughput. These results highlight the following definitions and observations:

XPath Processing Time. This duration mainly is the time consumed in storing XPath subscription filters in the CAM (Section 5.3.3.2), and storing t-SIDs in the "Subscription IDs Visiting Location Memory" (Section 5.3.3.3). The results stated in Table 7.1 and Figure 7.2 show that processing XPath Set2 takes less time than process-

Table 7.1: Broker Results on Stratix FPGA

Subscrip. Buffer Loaded with	XML doc. size (Bytes)	Number of Subscr.	XPath Process. Time (μs.)	Total Proc. Time (μs.)	Throughput (Incl. Overh.) 50-MHz clk (Mbps.)
XPath Set1	980	24	026.88	035.94	227.9
XPath Set1	980	24*2=48	053.62	062.68	261.4
XPath Set1	980	24*3=72	080.30	091.96	267.2
XPath Set1	980	24*4=96	106.98	121.24	270.2
XPath Set1	980	24*5=120	133.66	147.92	276.9
XPath Set1	980	24*6=144	160.34	177.20	277.3
XPath Set1	980	24*7=168	187.02	206.48	277.7
XPath Set1	980	24*8=192	213.70	235.76	277.9
XPath Set1	980	24*9=216	240.38	262.44	280.9
XPath Set1	980	24*10=240	267.06	291.72	280.8
XPath Set2	980	42	14.66	045.74	179.0
XPath Set2	980	42*2=84	29.00	063.10	259.6
XPath Set2	980	42*3=126	43.44	083.58	294.0
XPath Set2	980	42*4=168	57.88	101.04	324.3
XPath Set2	980	42*5=210	72.32	118.50	345.6
XPath Set2	980	42*6=252	86.76	138.98	353.6

ing XPath Set1, even though the number of subscriptions in XPath Set2 is higher. This observation can be understood, because the number of filter characters needed for storing 42 subscriptions of XPath Set2 is 4*42=168 (1344 bits), while the number of filter characters needed for storing 24 subscriptions for XPath Set 1 is at least 10*24=240 (1920 bits). Thus, the total number of filter characters that need to be stored for one set is the main factor.

Total Processing Time. The total processing time includes the XPath processing time (defined above) in addition to the total time consumed in loading and buffering XPath subscriptions, matching filters of all registered subscriptions against parsed XML publication data, and subsequently initiating a corresponding notification. In other words, the total processing time is the time consumed by the whole broker, where the duration of the loading and parsing tasks of the XML publication is transparently included in the processing time since these tasks concurrently occur with other broker's

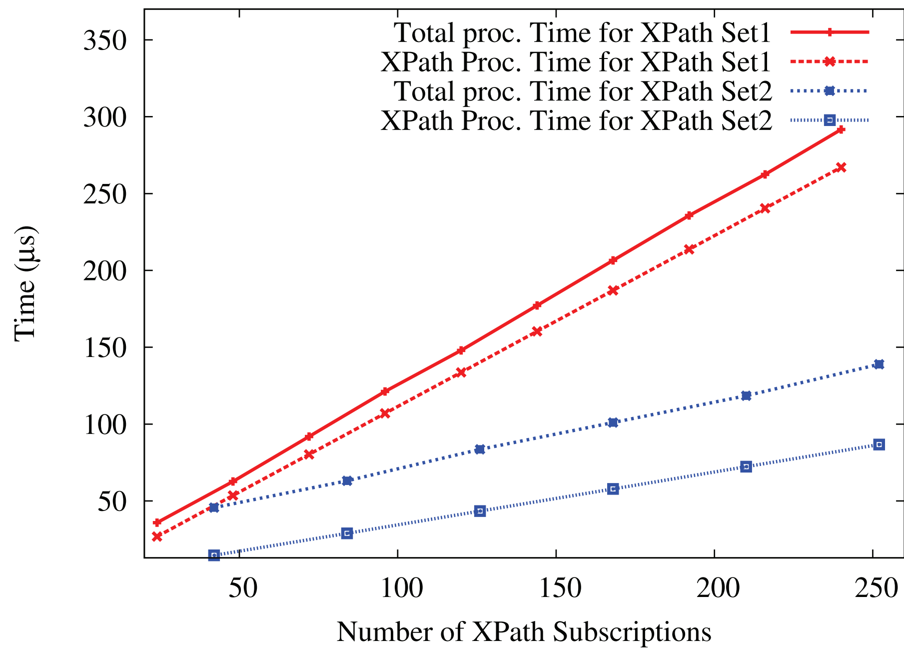

Figure 7.2: Graphical Representation of Broker's Processing Time.

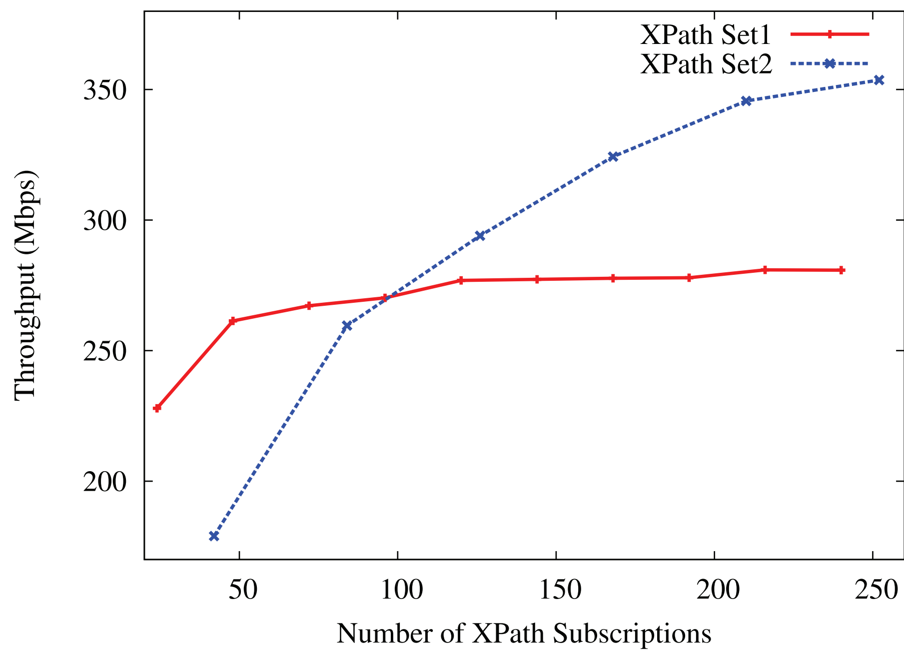

Figure 7.3: Graphical Representation of Broker's Throughput.

tasks. Table 7.1 and Figure 7.2 show the corresponding results. When the subscription buffer is loaded *only once* with either XPath Set1 or XPath Set2, the total processing time of XPath Set1 (35.94 μs) is less than that of XPath Set2 (45.74 μs), because the number of subscriptions is the main factor. When more subscriptions from either set are loaded, the total processing time of XPath Set1 becomes greater than that of XPath Set2, because the number of filter characters per subscription becomes the main factor. Thus, the XPath processing time dominates the total processing time for multiple loads of either set.

Throughput. Table 7.1 and Figure 7.3 illustrate the throughput results, where both the number of subscriptions (including their overhead) and the number of filter bits per subscription are important factors. In this phase (Phase 1), the XML parsing is not a factor under test. Rather, matching the parsed XML publication against subscriptions is a factor under test. Therefore, the number of XML bits does not take part of the shown throughput, even though the single XML publication loading and parsing processes occur in parallel with other broker's tasks. When around 96 subscriptions of either set are processed, the throughput becomes equivalent ($\simeq$270 Mbps) for either set. Prior to that point, the number of subscriptions is the main factor; thus, the throughput is higher for XPath Set1. Beyond that point, the large number of filter bits per subscription leads to larger processing time for XPath Set1; thus, the corresponding throughput does not increase sharply. However, the throughput for XPath Set2 continues to increase due to the low number of filter bits that need processing and the large number of overhead bits that do not need significant processing. These results demonstrate the impact of filter bits per subscription and the total number of subscriptions on the throughput of the broker.

7.5 Broker's Phase 1 Conclusion

Phase 1 experiments have tested the basic functionality and performance of the hardware architecture of the proposed broker in content-based publish/subscribe systems. The corresponding simulations have measured, in particular, the XPath processing time, the total processing time, and the throughput. The testing results indicate that there are two main factors that affect the overall performance: the number of filter bits per subscription, and the total number of subscriptions that must be processed. When a high number of XPath subscriptions are involved, the XPath processing time dominates the total processing time, while the matching process of all stored subscriptions is concurrently and efficiently performed.

Moreover, Phase 1 results indicate that matching common filters of more than 250 subscriptions stored in the CAM can be achieved with a throughput exceeding 350 Mbps for some cases, using a 50-MHz clock. In Phase 2, the DE2-115 FPGA board that supports Altera's Cyclone IV E FPGA device hosts the tests [115]. This device was manufactured with a process technology of 60 nm, which should allow the broker to achieve higher performance.

7.6 Experimental Setup Interfaces for Phase 2 Tests

The experiments conducted in Phase 2 examine the functionality and performance of the X2CBBR along with a new hardware interface (different from the one of Phase 1), when operating and routing algorithms are into effect. Both the X2CBBR and its hardware interface are implemented on an FPGA device. All tests of this phase utilize the Avalon bus specification [12]. In particular, the broker's hardware interface emulates the Avalon specification in hardware, while the software/hardware interface makes use of C-programs carefully developed to drive the X2CBBR along with its hardware interface through the Avalon bus. Running separate experiments using each of these two interfaces would indicate the difference in terms of performance between the X2CBBR solely interfaced by its hardware interface and the X2CBBR driven through the software/hardware interface.

7.6.1 Avalon Interfaces

The Avalon specification defines interfaces that simplify drive and control the hardware that had configured the FPGA device. For example, the Avalon-Memory-Mapped (Avalon-MM) interface supports read and write transfers, such as reading and writing registers and memory modules embedded in the X2CBBR design that have had configured the FPGA chip. This interface undertakes these transfers from a master port to a slave port, where the slave [12]:

- Receives the signals *address, byteenable, read* or *write,* and *writedata* after the rising edge of the clock. The signal *byteenable* specifies which bytes in the signal *writedata* are being written to the slave, or which bytes in the signal *readdata* the master is reading.
- Asserts the signal *waitrequest* before the next rising clock edge to hold off the

relevant transfer.

- De-asserts the signal *waitrequest* to complete the read/write transfer on the next rising clock edge. For the *read* transfer, the signal *readdata* is sampled, while the signal *writedata* is sampled for the *write* transfer.

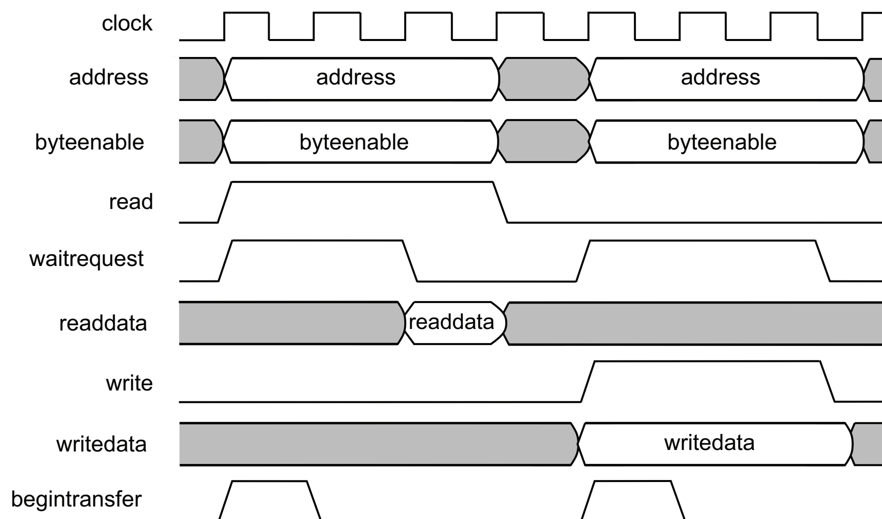

Figure 7.4: Avalon Read and Write Transfers with Waitrequest [12].

Figure 7.4 illustrates an example of the Avalon master-slave interfacing protocol. It is important to mention that the Avalon specification can handle additional signals including *begintransfer* and *chipselect*. The Avalon-MM interface asserts the signal *begintransfer* during the first clock cycle of each transfer regardless of other signals. The signal *chipselect* (not shown in Figure 7.4) does not allow the slave to read or write if it is not asserted in combination with read or write signals. This signal adds more protection against accidental transfers. However, the specification does not require that all signals appear in an Avalon-MM interface. The minimum requirements are the signal *readdata* for a read-only interface or the signals *writedata* and *write* for a write-only interface [12].

7.6.2 Hardware Interface

The hardware interface of Phase 2 emulates the Avalon-MM interface with the signals *address, read, write, writedata, readdata, begintransfer,* and *chipselect*. This interface does not need to use the signal *byteenable* because all bytes within the signals *writedata* and *readdata* are always selected. However, it utilizes the signal *chipselect* to provide protection against accidental write or read transfers.

The broker's hardware interface, named XML/XPath AValon Interface (XAVI), takes the Avalon inputs from a Verilog-based test bench and drives the X2CBBR, as illustrated in Figure 7.5.

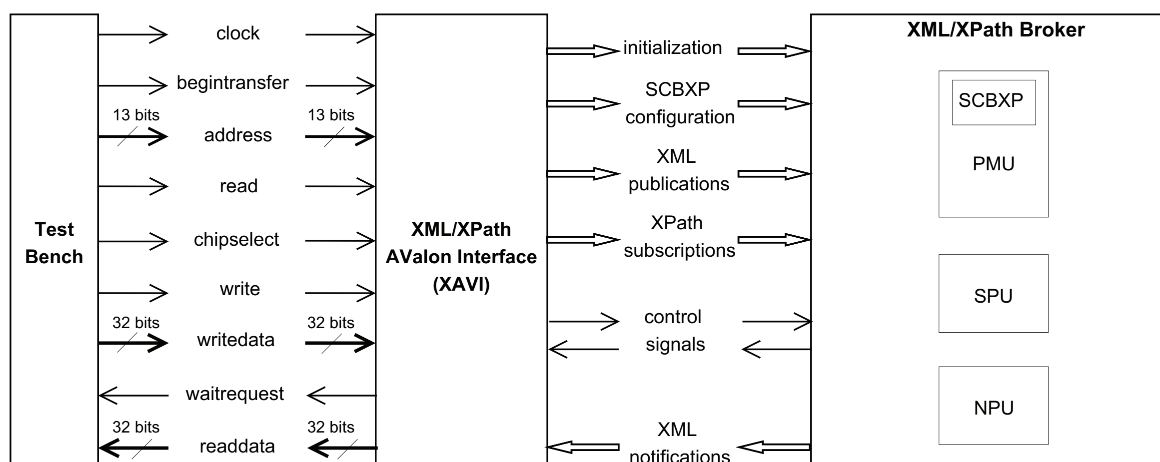

Figure 7.5: Broker's Hardware Interface.

Even though the Avalon-MM slave can have a dynamic data width of 8, 16, 32, 64, 128, 256, 512 or 1024 bits [12], either the *writedata* bus or the *readdata* bus in XAVI has a fixed 32-bit size, while the *address* bus is of 13-bit size. The choice of these sizes reflects the actual width space that the design needs to perform write and read transfers.

The hardware driving tasks include initializing all memory resources of the broker, configuring the SCBXP's CAM with an XML skeleton, loading the SCBXP's memory with XML publications, loading the SPU's subscription buffer with XPath subscriptions,

and performing various control and management activities.

The broker sends back to XAVI the relevant notifications as well as the data read from the broker's memory resources.

7.6.3 Software/Hardware Interface

The software/hardware interface consists of the hardware interface XAVI and the C-programs that are specifically developed in the software test environment of Altera's Nios II embedded processor [13]. The X2CBBR and XAVI together represent the slave module that is implemented on Altera's FPGA Cyclone IV E device. In this section, this module takes the name of $Broker_Top5$, indicating that the module is the top level being driven and that the design has been optimized up to the fifth implementation version. Through the Avalon bus master/slave ports, the C-programs drive the slave module, as depicted in Figure 7.6.

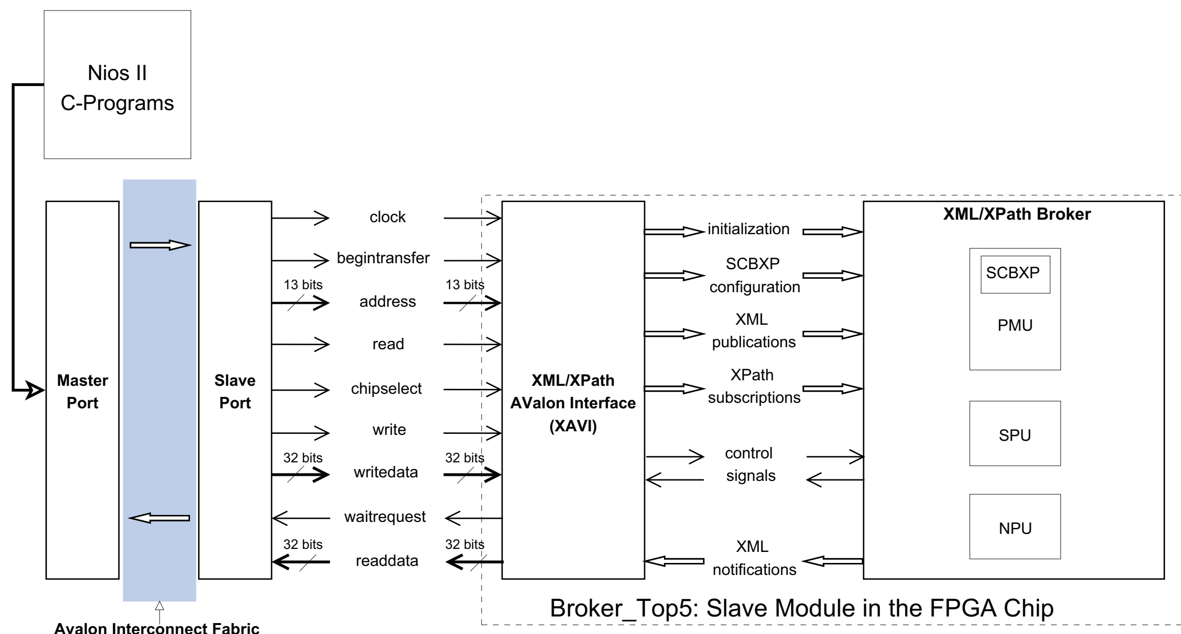

Figure 7.6: Broker's Software/Hardware Interface.

The Nios II processor embedded in the FPGA device is supported by a Real-Time Operating System (RTOS) called Micro-Controller Operating System (MicroC/OS II or simply μC/OS-II) [62]. This RTOS makes use of macros called from the developed C-programs in order to provide the Avalon bus driving signals needed to perform the communication process via the master/slave ports. A Nios II μC/OS-II software application can be considered an architecture of layers as illustrated in the onion-shape of Figure 7.7.

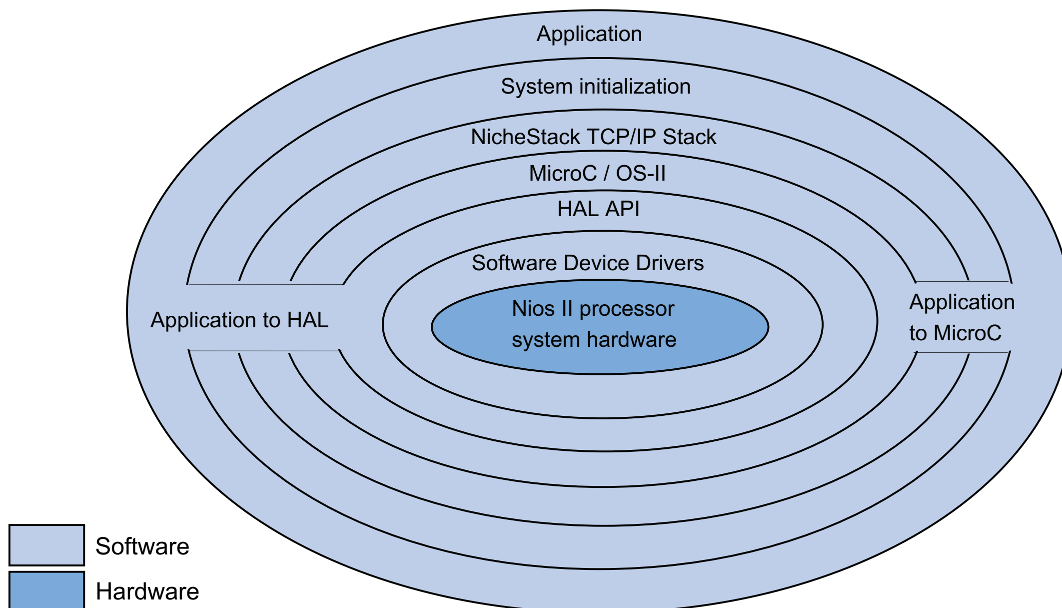

Figure 7.7: The Architectural Layers of a Nios II MicroC/OS-II Software Application. (modified after being extracted from [13])

The software application can access the hardware through a light-weight TCP/IP stack called NicheStack for networking purposes [13]. However, the application can directly access either the μC/OS-II layer or the Hardware Abstraction Layer (HAL), as shown in the right and left sides of Figure 7.7 respectively.

The μC/OS-II is a lightweight real-time multitasking kernel that is mainly written in ANSI C [62]. In order to adapt to different processor applications, the μC/OS-II includes

a small portion of assembly language code. Accordingly, many real-time applications are using the μC/OS-II besides software/hardware embedded systems. Examples of these applications include cameras, automated banking machines, medical instruments, avionics, and engine controls [62].

The HAL is a lightweight embedded runtime environment that serves as a device driver package for a Nios II processor system, and provides a simple device driver interface for C-programs to connect to the underlying hardware. The HAL Application Program Interface (API) is integrated with the ANSI C standard library [13].

The Avalon interconnect fabric (Figure 7.6) builds the necessary logic that routes the master's data intended for write operations to the signals *writedata*, *write*, and *address* of the corresponding slave. Similarly, the fabric responds to the master's command of reading data stored in a specific address, with the transfer of the address value indicated by the master to the signal *address* of the corresponding slave along with the assertion of the slave's control signal *read*. Subsequently, the fabric routes *readdata* delivered by the slave to the requesting master [12].

In general, the Avalon interconnect fabric may handle multiple masters and slaves, and its logic includes data-path multiplexing, address decoding, wait-state signalling, master/slave arbitration, write/read transfers, and interrupt request management. Altera provides a System-On-a-Programmable-Chip (SOPC) builder tool (affiliated to its Quartus II design tool) that can build the Avalon interconnect fabric interacting with the Nios II embedded processor and its peripherals [10][14].

Altera's SOPC builder tool connects the slave module "$Broker_Top5$" to the slave port of the Avalon-MM interface bus. Figure 7.8 indicates the relevant SOPC design that includes the Central Processing Unit (CPU) (that is the Nios II embedded processor), the Phase-Locked Loop (PLL) (that generates multiple clock frequencies), and the CPU's

Figure 7.8: The Broker/Router and XAVI Acting as a Slave Module Under the name of "*Broker_Top*5", which is Connected to a Slave Port in the SOPC Builder Tool.

peripherals including "*Broker_Top*5." This module and some of the peripherals reside on the configured FPGA chip, while some others are on-chip controllers that organize the communication process with off-chip modules located on the Terasic DE2-115 FPGA board [115].

On this board, the available clock is of 50-MHz frequency. However, the PLL can generate other frequencies with different phases, so that the generated clock frequency can be, for example, lower with a division factor, greater with a multiplication factor, or the same with a difference in phase. Referring to Figure 7.8, the reader can notice that,

for example, the LCD (Liquid Crystal Display) controller utilizes a clock frequency of 10 MHz generated by the PLL (*altpll_io*) in order to control the off-chip character LCD module. The *Broker_Top*5 module utilizes a clock frequency of 100 MHz generated by the PLL as *altpll_sys*. In the experiments that test the broker's performance with multiple applied frequencies, the X2CBBR has worked well with a clock frequency of 150 MHz generated by the embedded PLL.

7.6.4 Software/Hardware Interface with Content-Based Routing

To communicate with producers, consumers, or other brokers (content-based routers), the X2CBBR needs a standardized communication medium. While the broker is driven by the software programs, the Nichestack TCP/IP operating over Ethernet is the candidate that is available on the DE2-115 board. The software programs access the Nichestack layer as illustrated in Figure 7.7. The Ethernet functionality is provided as a configurable IP core (Intellectual Property core) by Altera. This IP core, named Triple-Speed Ethernet (TSE) MegaCore, complies with the IEEE 802.3 standard and possesses the features of a 10/100/1000-Mbps Ethernet Media Access Controller (MAC) [9][53].

The SOPC design shown in Figure 7.8 includes the TSE MAC that operates with other two IP cores called SG-DMA (Scatter-Gather Direct Memory Access) controllers [11]. The SG-DMA, unlike a regular DMA, is able to perform multiple transfers handled by the hardware without the intervention of a processor. The SG-DMA is able to perform the transfer between:

- A memory and another memory – An Avalon-MM master of the SG-DMA controller reads the first memory through an Avalon-MM slave. The data having been read

passes through the Avalon-streaming (Avalon-ST) source-sink pair interface [12] within the SG-DMA controller core. Then, an Avalon-MM master of the SG-DMA controller writes into the second memory through an Avalon-MM slave.

- A memory and a data stream – An Avalon-MM master of the SG-DMA controller reads the memory through an Avalon-MM slave. The data having been read passes through the Avalon-ST source of the SG-DMA controller. Then, the Avalon-ST sink of a streaming component receives the data stream.

- A data stream and a memory – The Avalon streaming source of a streaming component sends data to the Avalon-ST sink of the SG-DMA, then, the SG-DMA writes the data into memory using the Avalon-MM master-slave pair.

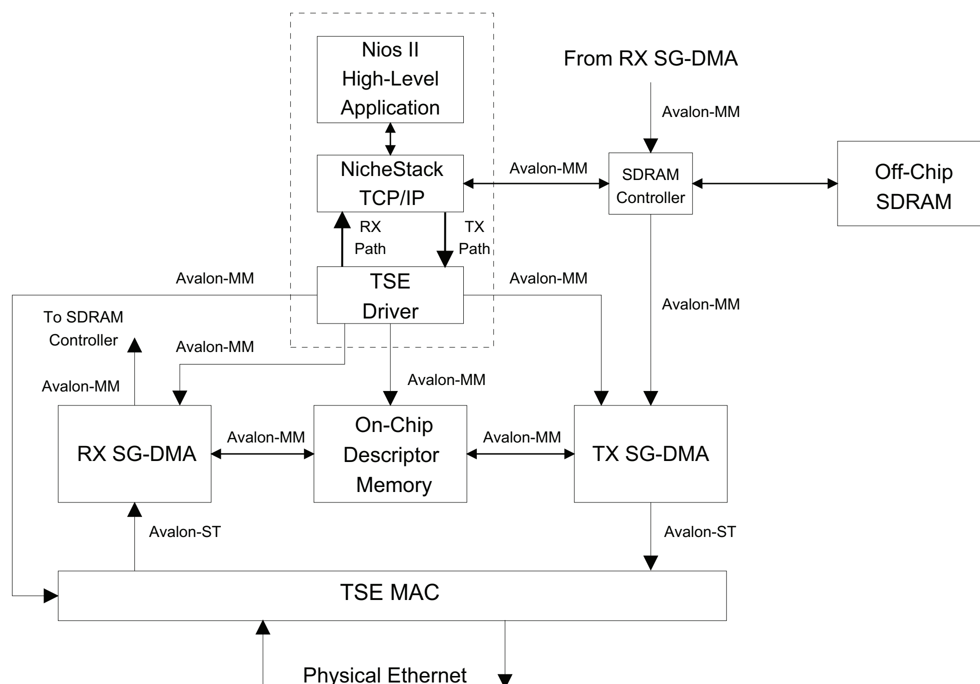

Figure 7.9: Triple-Speed Ethernet Driving Architecture, where all components are IP cores available on the DE2-115 board. (concept and basic version of this figure exist in [9])

An illustration of the TSE driving architecture [9], which reflects the TSE connections in the SOPC design of Figure 7.8, is depicted in Figure 7.9. In this figure, the top-level Nios II application drives the off-chip Synchronous Dynamic Random Access Memory (SDRAM) that is available on the DE2-115 board. Moreover, the architecture shows the TSE driver that communicates with the TSE MAC and SG-DMA controllers, and highlights as well the interaction between the SG-DMA controllers with the on-chip descriptor memory, the SDRAM, and the TSE MAC.

7.6.4.1 Broker to Producer/Consumer Connectivity

Figure 7.10 shows the connectivity between an end user (either a producer or a consumer) to the SOPC design (including the *Broker_Top*5 module as in Figure 7.8) that is implemented on the Cyclone IV E FPGA device.

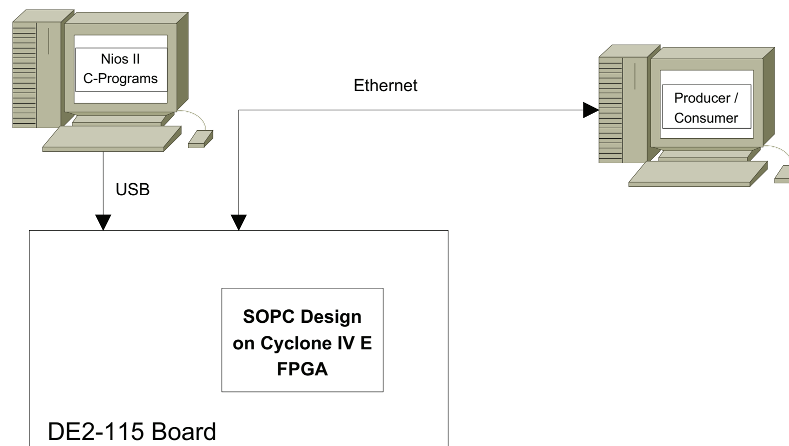

Figure 7.10: Broker to Producer/Consumer Connectivity.

7.6.4.2 Broker to Broker Connectivity

Figure 7.11 shows the broker-to-broker communication using Ethernet connectivity. In this figure, two DE2-115 FPGA boards are connected together via Ethernet.

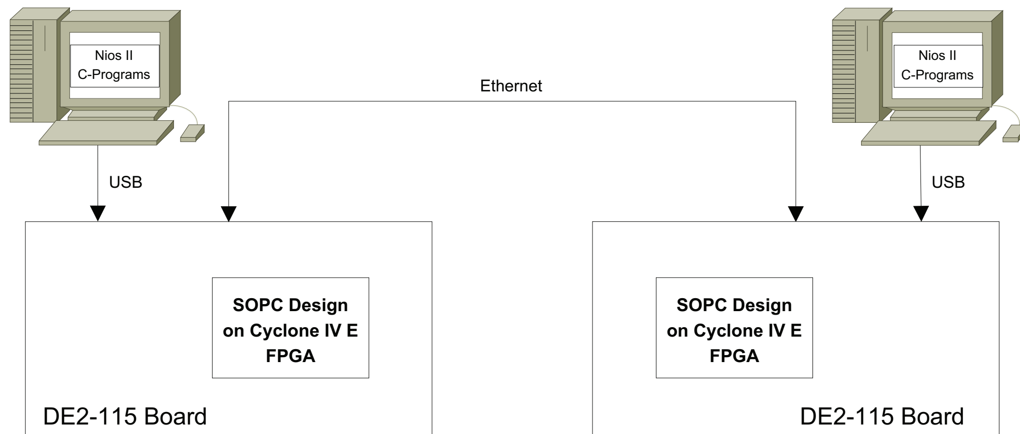

Figure 7.11: Broker to Broker Connectivity.

An alternative method of connecting a broker to another broker is illustrated in Figure 7.12, where Ethernet packets (routed subscriptions) can be collected through a linux-based socket programs running in a computer and interfacing the Nios II environment of another broker. This method allows for the measurement of the reception rate before the received data actually reaches the other broker.

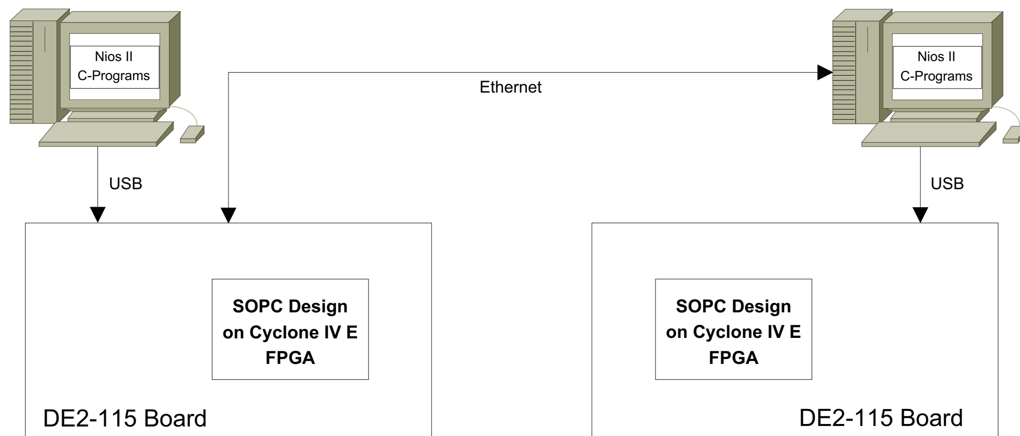

Figure 7.12: Broker to Broker Connectivity.

7.7 Experimental Setup of Subscription Sets for Phase 2 Tests

The conducted tests of Phase 2 employ three sets of subscriptions: XPath Set2 (utilized in Phase 1), XPath Set3, and XPath Set4. The next paragraphs describe these two last sets.

XPath Set3. This set consists of 42 subscriptions, and it is similar to XPath Set2 employed in the conducted tests of Phase 1 (Section 7.1) in terms of:

- The number of subscriptions (42) existing in the set, and
- The filter size (32 bits) in each subscription.

Many of the subscriptions in XPath Set3, like those in XPath Set2, match a publication that is available in the local broker. Accordingly, local notification delivery takes place for the relevant subscribers. However, the major difference between the two sets is that a few subscriptions in XPath Set3 do not match any publication that is available in the local broker. Therefore, mismatched-subscription routing followed by notification routing must take place prior to delivering the final notification to the relevant subscribers. Scenario 2 (Section 5.4.4.2) is a good example.

XPath Set4. This set of 42 subscriptions is similar to XPath Set3 in terms of the number of subscriptions existing in the set and the filter size in each subscription. However, in XPath Set4, all subscriptions have identical filters but they do not match any available publications in their local broker. Therefore, after registering multiple loads of XPath Set4, sample-subscription routing followed by publication routing must take place prior to the final matching process and notification delivery. Scenario 5 (Section 5.4.4.5) is a good example.

Three different groups of tests are to be performed with XPath Set2, XPath Set3, or XPath Set4:

- Case I: a test group that utilizes the hardware interface described in Section 7.6.2. XPath Set2 is the sole set employed in this case, while scenario 1-b (Figure 5.28) is emulated. However, 6 sets of XPath Set2 are gradually included in the test, where a reset signal resets the broker after matching each additional set.

- Case II: a test group that utilizes the software/hardware interface described in Section 7.6.3. As in Case I, XPath Set2 is the sole set employed in this case, while the same scenario is emulated with 6 sets of XPath Set2 that are gradually included in the test as well.

- Case III: a test group that utilizes the software/hardware interface with content-based routing (Section 7.6.4). In this case, the performance of subscription routing to a neighboring broker in addition to subscription processing and notification routing are to take place. This case employs both XPath Set3 and XPath Set4.

7.8 Case I: Testing Through the Hardware Interface

The procedure of tests in Case I consists of measuring the performance of the X2CBBR driven through the XAVI hardware discussed in Section 7.6.2. The embedded PLL extracted from Altera library applies the following clocking strategy: A 50-MHz clock drives the XAVI as well as the SCBXP (the XML parser), while a 150-MHz PLL-generated clock drives the core of the X2CBBR, as depicted in Figure 7.13.

This carefully-chosen strategy allows the broker to achieve its high performance in its core, while the interface and the SCBXP remain error-free and well-synchronized.

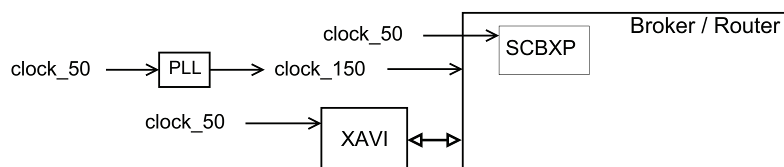

Figure 7.13: Clocking Strategy.

While the slower parser clock may initially limits the broker's performance, this impact becomes marginal on the overall performance, because the broker processes and matches many subscriptions versus one parsed XML publication at a time. Moreover, the SCBXP makes use of a stand-alone parsing technique that does not interfere with the broker's core. The output of the SCBXP goes to a dual-port memory (the reading stage in Figure 4.1), where the parsed data gets into the memory via one port with the 50-MHz clock, and this data is later accessed by the broker via the other port with the 150-MHz clock. This memory is the only module, within the broker, that must handle two different clock frequencies.

7.8.1 Results of Case I

The results of Case I represent three operational cases: The worst-case scenario that occurs in the initial operation of the broker, then the regular-case scenario, and finally the best-case scenario.

Worst-case. First, the initialization process of the broker's memory resources takes about 17.7 μs. Then, the X2CBBR sequentially receives data through its hardware interface XAVI. The main sequential broker tasks are as follows: (1) SCBXP's CAM configuration with an appropriate XML skeleton, (2) SCBXP's loading with one XML publication of 1008 Bytes, (3) SCBXP's XML parsing, (4) Buffering the 42 XPath subscriptions of XPath Set2 having a total volume of 1008 Bytes, (5) XPath subscription

mapping in the SMT and registration in the CRT (the volume of the total filter data that has to be registered is 168 Bytes, since there are four Bytes of filter data in each subscription), and (6) Initiating the matching process that involves one publication versus the registered filters of 42 subscriptions.

Since this publication is the first one that reaches the broker, the operation mode of the broker stays at the *(nP,nS)* mode up to S_{max}, then transits to the *(nP,oS)* mode (see Section 5.4.3.3). Therefore, the first five sequential tasks occur in the *(nP,nS)* mode, while the matching process occurs in the *(nP,oS)* mode. According to the simulations undertaken by the ModelSim simulator, the resulting performance can be measured.

To still emulate the worst-case, the X2CBBR has to step to the *(NOP)* mode by means of an applied reset signal. Then, the same sequence above takes place again except that the subscription loading process occurs with two series of XPath Set2 subscriptions. The resulting performance is measured thereafter.

The worst-case experiment continues with resetting the broker, initializing its memory resources, and repeating the sequence of the aforementioned tasks with additional loading of XPath Set2 subscriptions up to a total of 252 subscriptions (six loads of XPath Set2). The corresponding performance is measured after each processing round that spans the *(nP,nS)* mode, where the SCBXP only parses one XML publication, and the *(nP,oS)* mode (Section 5.4.3.3).

Table 7.2 illustrates the corresponding worst-case results delivered by X2CBBR with XAVI, where one up to six series of XPath Set2 subscriptions are applied and matched against the first XML publication of 1008 bytes in the system.

These results show that while the processing time increases with additional subscriptions, the overall throughput improves. In the case that 252 subscriptions are to be processed and matched against one XML publication, the broker completes its tasks in

Table 7.2: Worst-Case Results of the Broker with Hardware Interface on Cyclone IV E FPGA: First Pub. versus Subsequent Subs. (up to 252 Subs.)

All-Sequential Main Broker Tasks [1 Pub. (1008B.) versus Series of 42 Subs.(1008B.) up to 252 Subs.(1008B.*6)]	Oper. Mode	Clock Freq. (MHz.)	Process. Time (μs.)	Through-put (Mbps)
SCBXP's CAM Configuration (1020B.)	**(nP,nS)**	50	056.34	144.83
SCBXP's XML Pub. Loading (1020B.)	**(nP,nS)**	50	056.32	144.88
XML Pub. Parsing (1008B.)	**(nP,nS)**	50	010.54	765.08
All SCBXP Tasks for Pub. (1008B.)	**(nP,nS)**	**50**	**123.20**	**65.45**
XPath Subs. Buffering (1008B.)	**(nP,nS)**	150	046.10	174.92
Subs. Mapp.(42*8B.) & Regist.(168B.)	**(nP,nS)**	150	016.08	250.74
Matching 1 Pub. vs. Filters of 42 Subs.	**(nP,oS)**	150	020.96	448.85
All Tasks [Pub.(1008B.) / Sub.(1008B.)]		**50/150**	**206.34**	**78.16**
XPath Subs. Buffering (1008*2B.)	**(nP,nS)**	150	092.20	174.92
Subs. Mapp.(42*2*8B.) & Regist.(168*2B.)	**(nP,nS)**	150	019.50	413.53
Matching 1 Pub. vs. Filters of 42*2 Subs.	**(nP,oS)**	150	021.88	491.40
All Tasks [Pub.(1008B.) / Sub.(1008*2B.)]		**50/150**	**256.78**	**94.21**
XPath Subs. Buffering (1008*3B.)	**(nP,nS)**	150	138.30	174.92
Subs. Mapp.(42*3*8B.) & Regist.(168*3B.)	**(nP,nS)**	150	022.74	531.92
Matching 1 Pub. vs. Filters of 42*3 Subs.	**(nP,oS)**	150	022.79	530.76
All Tasks [Pub.(1008B.) / Sub.(1008*3B.)]		**50/150**	**307.04**	**105.05**
XPath Subs. Buffering (1008*4B.)	**(nP,nS)**	150	184.40	174.92
Subs. Mapp.(42*4*8B.) & Regist.(168*4B.)	**(nP,nS)**	150	026.16	616.51
Matching 1 Pub. vs. Filters of 42*4 Subs.	**(nP,oS)**	150	023.71	566.85
All Tasks [Pub.(1008B.) / Sub.(1008*4B.)]		**50/150**	**357.47**	**112.79**
XPath Subs. Buffering (1008*5B.)	**(nP,nS)**	150	230.50	174.92
Subs. Mapp.(42*5*8B.) & Regist.(168*5B.)	**(nP,nS)**	150	029.58	681.54
Matching 1 Pub. vs. Filters of 42*5 Subs.	**(nP,oS)**	150	025.54	578.85
All Tasks [Pub.(1008B.) / Sub.(1008*5B.)]		**50/150**	**408.82**	**118.35**
XPath Subs. Buffering (1008*6B.)	**(nP,nS)**	150	276.60	174.92
Subs. Mapp.(42*6*8B.) & Regist.(168*6B.)	**(nP,nS)**	150	032.10	753.64
Matching 1 Pub. vs. Filters of 42*6 Subs.	**(nP,oS)**	150	028.47	566.49
All Tasks [Pub.(1008B.) / Sub.(1008*6B.)]		**50/150**	**460.37**	**122.61**

460.37 μs (including 123.20 μs for XML parsing) with a total of 122.61 Mbps of throughput. However, the internal tasks of the broker that do not make any communication via the interface achieve higher performance than those that depend on the data passing through the interface. For example, the actual XML parsing that does not make any interface communication is just 10.54 μs. Similarly, the actual matching of one XML publication against all 252 registered subscription filters takes just 28.47 μs.

Comparing these results with those previously shown in Table 7.1, the reader would notice that the XML publication loading and parsing tasks are included in Case I of Phase 2 tests unlike the test case of Phase 1. In this phase (Phase 2), the operation mode algorithm of the broker comes into play, where the first XML publication has to be loaded and parsed through the XAVI hardware followed by other tasks. Therefore, the number of XML bits effectively takes part of the total processing time and the shown throughput. In this case, the total processing time is higher – and the throughput is less – than that of Table 7.1 because of the presence of the XAVI hardware that adds some delay in addition to the XML publication loading and parsing tasks.

Regular-case. The experiment continues with sending a new XML publication (and a relevant skeleton) at a time to the X2CBBR through the XAVI hardware, while the broker processes this publication and matches it against the 252 already-registered subscriptions. Therefore, either re-loading or re-registering of these subscriptions is not needed, and thus does not take part of the processing time. However, the processing time of each new subsequent XML publication as well as the matching process duration remain into effect. Table 7.3 shows the corresponding results. In this regular case, the broker operates in its *(nP,oS)* mode (Section 5.4.3.3).

Table 7.3: Regular-Case Results of the Broker with Hardware Interface on Cyclone IV E FPGA: Each New Subsequent Publication with a New Skeleton versus 252 Already-Registered Subscriptions.

Main Broker Tasks [Each Subseq. Pub. vs. 252 Subs.] (SCBXP Loads are All-Sequential)	**Oper. Mode**	**Clock Freq. (MHz.)**	**Process. Time (μs.)**	**Through-put (Mbps)**
SCBXP's CAM Configuration (1020B.)	**(nP,oS)**	50	056.34	144.83
SCBXP's XML Pub. Loading (1020B.)	**(nP,oS)**	50	056.32	144.88
XML Pub. Parsing (1008B.)	**(nP,oS)**	50	010.54	765.08
All SCBXP Tasks for Pub. (1008B.)	**(nP,oS)**	**50**	**123.20**	**065.45**
Matching 1 Pub. vs. Filters of 252 Subs.	**(nP,oS)**	150	028.47	566.49
All Tasks [Pub.(1008B.) / Sub.(1008*6B.)]		**50/150**	**151.67**	**372.17**

In this regular case, the broker completes its tasks in just 151.67 μs. (including 123.20 μs. for XML parsing) with a total of 372.17 Mbps. of throughput.

Best-case. The experiment continues with sending a new XML publication at a time to the X2CBBR through the XAVI hardware, where the corresponding skeleton has already configured the CAM of the SCBXP. The reader would remember that the same skeleton may be used for multiple XML publications that have common elements (Section 4.3). The broker processes this new publication and matches it against the 252 already-registered subscriptions. Therefore, the processing time of each new subsequent XML publication does not include neither the SCBXP's CAM configuration time nor the loading and registration time of the subscriptions. Table 7.4 shows the corresponding results.

Table 7.4: Best-Case Results of the Broker with Hardware Interface on Cyclone IV E FPGA: A New Subsequent Pub. and its Already-Stored Skeleton versus 252 Already-Registered Subscriptions.

Main Broker Tasks [Each Subseq. Pub. vs. 252 Subs.]	Oper. Mode	Clock Freq. (MHz.)	Process. Time (μs.)	Through-put (Mbps)
XML Pub. Loading (1020B., but No New CAM Configuration)	**(nP,oS)**	50	056.34	144.83
XML Pub. Parsing (1008B.)	**(nP,oS)**	50	010.54	765.08
All SCBXP Tasks for Pub. (1008B.)	**(nP,oS)**	**50**	**66.88**	**120.57**
Matching 1 Pub. vs. Filters of 252 Subs.	**(nP,oS)**	150	028.47	566.49
All Tasks [Pub.(1008B.) / Sub.(1008*6B.)]		**50/150**	**95.35**	**592.00**

This table states the best case during which the broker operates in its *(nP,oS)* mode. The broker completes its tasks in just 95.35 μs. (including 66.88 μs. for XML parsing) with a total of 592.00 Mbps. of throughput.

Reflecting Tables 7.2, 7.3, and 7.4, Figure 7.14 graphically shows the processing time in the worst, regular, and best cases, while Figure 7.15 graphically shows the corresponding throughput.

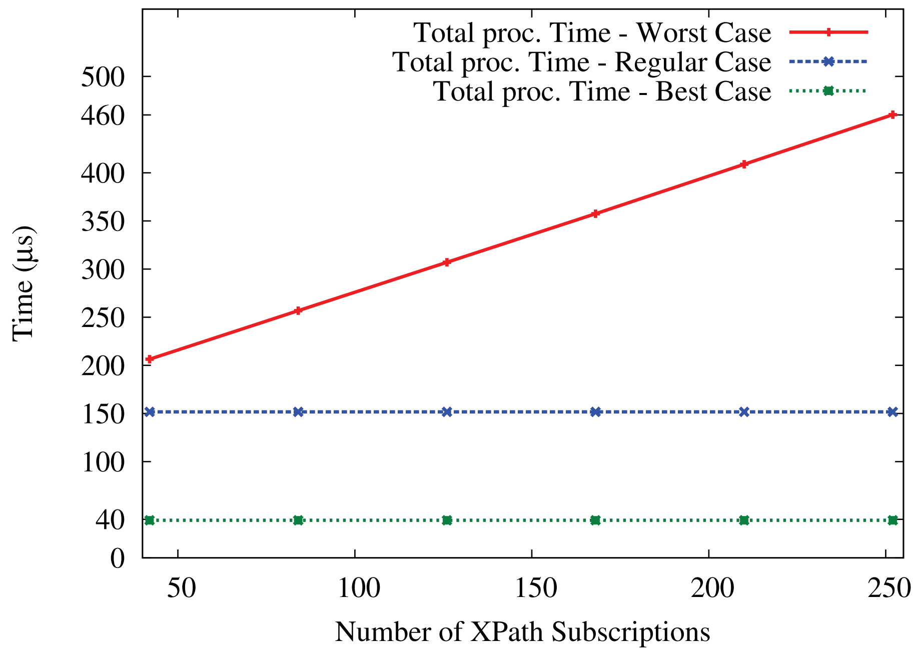

Figure 7.14: Processing Time in the Worst, Regular, and Best Cases.

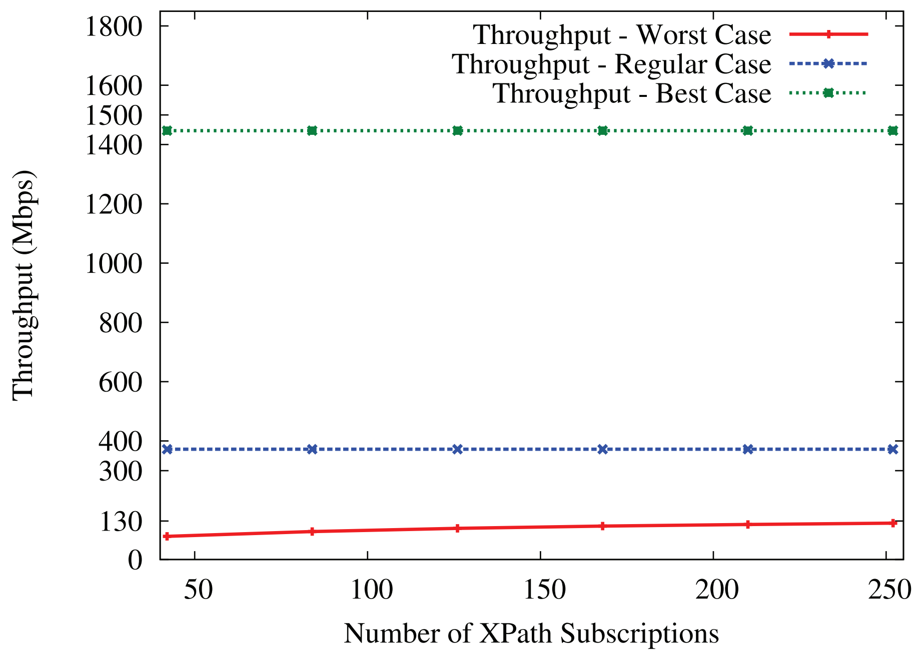

Figure 7.15: Throughput in the Worst, Regular, and Best Cases.

Completion of the Operation Mode Cycle and Reversion to the *(nP,nS)* Mode. When all available publications have been exposed for matching against registered subscriptions in the (nP, oS) mode, the broker steps to the (oP, oS) mode (Section 5.4.3.3). However, the broker is more likely to promptly step to (oP, nS) mode and back to the first (nP, nS) mode, when new subscription data is available in the buffer (Section 5.4.3.3.1). It is useful to remind the reader that, during all operating modes of the broker, the "Subscription Buffer" can still store new subscriptions in its other dual-port memory (Section 5.4.3.3 and Figure 5.6). Therefore, when the broker completes its operation mode cycle and return to its *(nP,nS)* mode, the buffer may already contain some or many new subscriptions. In this case, the buffering time needed in Table 7.2 would be reduced. Providing that the subscription buffer has already stored new 252 subscriptions prior to the broker return to its *(nP,nS)* mode, the buffering time 276.60 μs in Table 7.2 completely vanishes. In such a case, the broker completes its tasks in both *(nP,nS)* and *(nP,oS)* modes, for the first new XML publication, in just 183.77 μs (460.37 – 276.60 = 183.77). This result translates into 307.16 Mbps of throughput, which is much better than the worst-case.

Scalability. An interesting goal is to have insights about the scalability degree of X2CBBR when it is overwhelmed with only subscriptions, only publications, or with both subscriptions and publications.

Receiving one million subscriptions ($\simeq$24 MB.) with only one available publication ($P_{max} = 1$), X2CBBR processes each six sets of subscriptions (252 subs.) according to Table 7.2, with 460.37μs. Since $P = P_{max}$, the broker quickly completes the cycle of operation modes and reverts to its *(nP,nS)* mode. Subsequently, X2CBBR processes the next six sets but it does not need to load nor parse a new publication. Therefore, it saves the SCBXP time of 123.20μs. Thus, the processing time of each subsequent six sets is 460.37–

123.20=337.17μs. Since the subscription buffer consists of two memories, this buffer may already contain some or many new subscriptions. Thus, the buffering time needed in Table 7.2 would then be reduced. Providing that the buffer has already stored new 252 subscriptions, the buffering time 276.60μs in Table 7.2 completely vanishes. Accordingly, the processing time of the next six sets may become 337.17–276.60=60.57μs. However, the duration of 60.57μs would not constantly be the processing time for each new subsequent 252 subscriptions, because the buffering stage is longer than the subsequent stages. For one publication versus one million subscriptions, X2CBBR completes all tasks in a duration that ranges from 1.12 sec to 1.34 sec. The corresponding throughput ranges from 143.4 Mbps to 171.7 Mbps. Figures 7.16 and 7.17 illustrate the performance of the X2CBBR and its hardware XAVI when one publication is parsed and matched against the first 2000 subscriptions and one million subscriptions respectively.

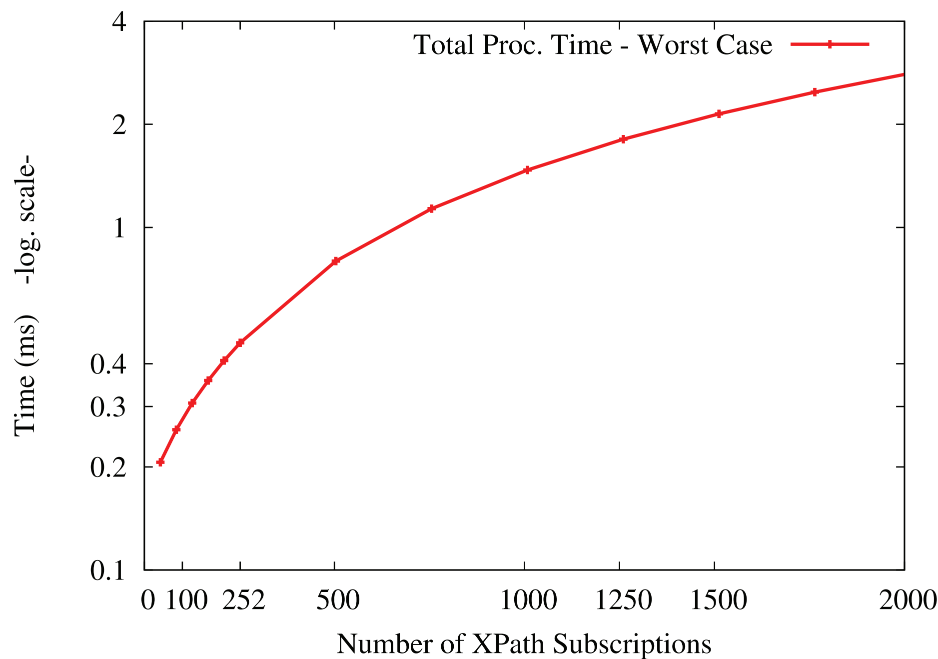

Figure 7.16: Performance in the Case of One Publication versus the First 2000 Subscriptions.

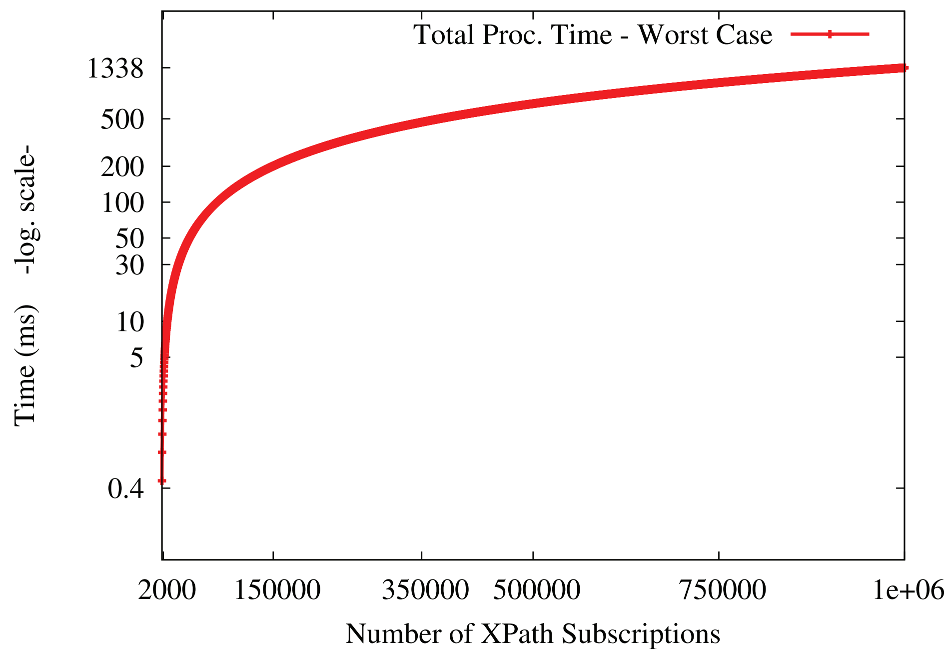

Figure 7.17: Performance in the Case of One Publication versus One Million Subscriptions.

Receiving one million publications (≃1 GB.) with only 252 (six sets) subscriptions can become time-consuming, since each publication needs to be sequentially matched against subscriptions. X2CBBR processes these subscriptions and matching them against the first publication according to Table 7.2, with 460.37μs. However, X2CBBR processes the next publication versus these already-registered subscriptions according to Table 7.3, with 151.67μs. For one million publications versus 252 subscriptions, X2CBBR completes all tasks in ≃2.53 minutes. However, when a publication and its skeleton are already loaded, X2CBBR processes such publication according to Table 7.4, with only 39.01μs. In this case, X2CBBR can complete all tasks in a duration that ranges from ≃39.01 sec to ≃2.53 minutes. The corresponding throughput ranges from 53.1 Mbps to 206.7 Mbps (average ≃130 Mbps).

When faced with a high number of subscriptions, X2CBBR limits the value of P_{max}

for publications to avoid the scenario of dealing with two growing scalability factors - a matter that may lead to a scalability problem. Therefore, when overwhelmed with subscriptions, the broker may enforce $P_{max} = 1$ to fall into the throughput range of 143.4 Mbps to 171.7 Mbps. Alternatively, it allows P_{max} to grow while limiting the number of subscriptions, so that the average throughput can be within the range of $\simeq$130 Mbps.

7.9 Case II: Testing Through the Software/Hardware Interface

When a software interface is involved in driving the broker, some concurrency features have the potential to diminish. The experiments of Case II emulate the same scenarios done in Case I using the software/hardware interface described in Section 7.6.3. Accordingly, the first experiment represents the worst-case scenario that occurs in the initial operation of the broker, while subsequent experiments represent the regular-case and best-case scenarios. By emulating Case I with a software/hardware interface instead of the sole hardware interface, the tests of Case II allow for the comparison of the performance results collected when either interface drives the X2CBBR.

7.9.1 Results of Case II

In Case II, the set of subscriptions involved in all test cases is XPath Set2 that can be multiplied 6 times, so that the total number of subscriptions can be 252 consisting of $1008 * 6 = 6048$ bytes.

Worst-case. In the software test environment of Altera's Nios II [13], C-programs first initialize the broker's memory resources. Then, the programs sequentially send data

to the broker, through the software/hardware interface.

Two loops in the programs effectively capture the processing time as follows: (1) A loop executes 10000 times the sequence of SCBXP's CAM configuration with an appropriate XML skeleton, SCBXP's loading with one XML publication of 1008 Bytes, and the initiation of SCBXP's XML parsing; (2) Another loop executes 10000 times the sequence of loading up to 252 subscriptions in the broker's subscription buffer, and ordering the broker to initiate the subscription registration and the pub/sub matching process. The processing time for one iteration of each loop is computed by dividing each loop time with the number of loop iterations (i.e. 10000). Table 7.5 shows the corresponding results.

Table 7.5: Worst-Case Results of the Broker with Software/Hardware Interface on Cyclone IV E FPGA: First Pub. versus 252 Subs.

All-Sequential Main Broker Tasks [1 Pub. (1008B.) versus 252 Subs.(1008B.*6)]	Operation Mode	Processing Time (ms.)	Through-put (Mbps)
All SCBXP Tasks for Pub. (1008B.)	**(nP,nS)**	**01.80**	**04.48**
All Tasks for 252 Subs.(1008B.*6)	**(nP,nS)/(nP,oS)**	**08.70**	**05.56**
All Tasks [Pub.(1008B.) / Sub.(1008*6B.)]		**10.50**	**05.37**

The comparison of these results with the hardware interface's worst-case of Table 7.2 shows that the processing time diminishes more than 22 times, which translates into around 95% of reduction in performance. However, achieving a worst-case performance of 10.5 ms using a software/hardware interface and the Cyclone IV E FPGA is interesting to highlight.

Regarding the SCBXP tasks, the worst-case processing time of 1.8 ms is lower than the duration of 4.227 ms achieved in the test case that was earlier presented in Section 6.5.3. This performance improvement is due to the utilization of the Cyclone IV E device instead of the Stratix device, since the newer device allows the SCBXP to safely operate with a 50-MHz clock instead of the 33.33-MHz clock.

Regular-case. The aim of this test case is to test the performance of the broker through the software/hardware interface when the subscriptions have been already registered in the CRT. Once done, this test case can be compared with the regular-case of Case I.

Therefore, two loops in the programs drive the broker as follows: (1) A loop executes 10000 times the sequence of SCBXP's CAM configuration with an appropriate XML skeleton, SCBXP's loading with one XML publication of 1008 Bytes, and the initiation of SCBXP's XML parsing; (2) Another loop executes in its first iteration the tasks of loading and processing 252 subscriptions, but in all 10000 iterations, it executes the command that initiates the pub/sub matching process. The resulting processing time for each loop is then divided by 10000 - the number of loop iterations. This test procedure marginalizes the duration of subscription loading in the buffer and the subscription registration in the CRT. Thus, the regular-case of Case II is comparable to the regular-case of Case I. Table 7.6 shows the corresponding results.

Table 7.6: Regular-Case Results of the Broker with Software/Hardware Interface on Cyclone IV E FPGA: Each New Subsequent Publication with a New Skeleton versus 252 Already-Registered Subscriptions.

Main Broker Tasks [Each Subseq. Pub. vs. 252 Subs.] (SCBXP Loads are All-Sequential)	Operation Mode	Processing Time (ms.)	Through-put (Mbps)
All SCBXP Tasks for Pub. (1008B.)	**(nP,oS)**	**01.80**	**04.48**
Matching 1 Pub. vs. Filters of 252 Subs.	**(nP,oS)**	**00.10**	**161.28**
All Tasks [Pub.(1008B.) / Sub.(1008*6B.)]		**01.90**	**29.71**

The comparison of these results with the hardware interface's regular-case of Table 7.3 shows that the processing time diminishes more than 12 times, which translates into around 92% of reduction in performance. However, the comparison of the results with the worst-case of Case II (Table 7.5) indicates that the processing time drops from 10.5

ms to just 1.9 *ms*, which means that the performance is around 82% higher.

Best-case. The aim of this test case is to test the performance of the broker through the software/hardware interface when the subscriptions have been already registered in the CRT, and the skeleton of the XML publication does not need to configure the SCBXP's CAM. Once done, this test case can be compared with the best-case of Case I.

Therefore, two loops in the programs drive the broker as follows: (1) A loop configures the SCBXP's CAM with an appropriate XML skeleton only in the first iteration, but it performs 10000 times the sequence of loading an XML publication of 1008 Bytes and parsing this publication; (2) Another loop performs the tasks of loading and processing 252 subscriptions only in its first iteration, but it executes the command that initiates the pub/sub matching process in all 10000 iterations. The resulting processing time for each loop is then divided by 10000 - the number of loop iterations. This test procedure marginalizes the duration of SCBXP's CAM configuration, the subscription loading in the buffer, and the subscription registration in the CRT. Thus, the best-case of Case II is comparable to the best-case of Case I. Table 7.7 shows the corresponding results.

Table 7.7: Best-Case Results of the Broker with Software/Hardware Interface on Cyclone IV E FPGA: Each New Subsequent Publication with an Old Skeleton versus 252 Already-Registered Subscriptions.

Main Broker Tasks [Each Subseq. Pub. vs. 252 Subs.] (XML Pub. Load, No CAM Configuration)	Operation Mode	Processing Time (*ms*.)	Through-put (Mbps)
All SCBXP Tasks for Pub. (1008B.)	**(nP,oS)**	**01.10**	**07.33**
Matching 1 Pub. vs. Filters of 252 Subs.	**(nP,oS)**	**00.10**	**161.28**
All Tasks [Pub.(1008B.) / Sub.(1008*6B.)]		**01.20**	**47.04**

The comparison of these results with the hardware interface's best-case of Table 7.4 shows that the processing time diminishes more than 12 times, which translates into around 92% of reduction in performance. However, the comparison of the results with

the worst-case of Case II (Table 7.5) indicates that the processing time drops from 10.5 ms to just 1.2 ms, which means that the performance is more than 88% higher. With regards of the comparison with the regular-case of Case II (Table 7.6), the processing time drops from 1.9 ms to 1.2 ms, which means that the performance is more than 36% higher.

7.10 Case III: Testing Through the Software/Hardware Interface with Content-Based Routing

In the experiments of Case III, the software/hardware interface is still on use with the focus on the content-based routing of either subscriptions or publications. The Ethernet connectivity between a broker and a produce/consumer or between two brokers is provided according to the description of Section 7.6.4 and Figures 7.10, 7.11, and 7.12.

The first experiment utilizes the set of subscriptions XPath Set3 multiplied six times, so that a total of 252 subscriptions (consisting of 1008 * 6 = 6048 bytes) can be figured in the test. This experiment emulates scenario 2 (Figure 5.29) in terms of routing each mismatched subscription to another broker. However, while in scenario 2 there is only one involved subscription, there are three mismatched subscriptions in XPath Set3, thus, a total of 18 mismatched ones in six sets. The communication among different entities takes place according to the actual connectivity shown in Figure 7.12. The second broker processes the received subscriptions and issues relevant notifications. The corresponding notifications take the route back to the first broker, where they are locally delivered.

The second experiment utilizes six sets of XPath Set4. Since all subscriptions of XPath Set4 does not match any publication that is available to the X2CBBR, a single mismatched subscription is routed to another X2CBBR on behalf of all mismatched

subscriptions (Section 5.4.4.5). This experiment emulates scenario 5 (Figure 5.32). The second X2CBBR processes the received subscription, and routes the corresponding "publication" to the first broker, where local subscriptions are re-processed against the received publication. Subsequently, the first broker locally delivers the corresponding notifications.

In both experiments, there is only one link between all subscribers and the first X2CBBR, and there is only one link as well between the first and second X2CBBRs. Therefore, subscriptions reach the first X2CBBR in sequence, routed subscriptions and notifications parade between both X2CBBRs in sequence, and the relevant notifications arrive to subscribers in sequence. Accordingly, the results of both experiments represent the worst-case in terms of link management. In the case of the existence of multiple links, the X2CBBR would be able to manage six external links (Section 5.4.5), which could positively impact the overall performance.

7.10.1 Results of Case III

The next paragraphs discuss the results of both experiments.

7.10.1.1 First Experiment Results

Table 7.8 shows the routing duration of each mismatched subscription as well as that of each corresponding notification. The routing duration of a single subscription (1.02 ms) applies for either the subscribing action or the subscription forwarding to another broker. The routing duration of a single notification (43.47 ms) applies for either the forwarding action or the notification delivery. This relatively-slow routing duration is due to the slow connection when the software-based Nichestack TCP/IP is used in driving the Ethernet MAC. This bottleneck could have been highly mitigated if hardware-based

TCP or UDP offload packets were used.

Table 7.8: Results of the Broker with Software/Hardware Interface and Subscription Routing

Content-Based Routing [Each Subscription 24 B.] (Each Notification 1020 B.)	**Oper. Mode**	**Routing of 1 Subscr. (ms.)**	**Through-put (Kbps)**	**Routing of 1 Notif. (ms.)**	**Through-put (Kbps)**
	(oP,oS)	**01.02**	**188.23**	**43.47**	**187.71**

Based on the discussions of Section 5.4.3.7.1, a subscription $s \in$ XPath Set3 may take the following path:

1. Subscriber of subscription $s \xrightarrow{sub(o-SID,F)}$ Local Broker.
2. No match found in Local Broker.
3. Local Broker $\xrightarrow{sub(\{o-SID,B1\},F)}$ Remote Broker, where $B1$ is the BID of the local broker.
4. Match found in Remote Broker.
5. Remote Broker $\xrightarrow{forward(\{o-SID,B1,B2\},N_F)}$ Local Broker, where $B2$ is the BID of the remote broker.
6. Local Broker $\xrightarrow{notify(N_F)}$ Subscriber of subscription s.

Therefore, referring to the definitions of T_l, T_p, and T_f of Section 5.4.3.8, the end-to-end path delay $T_1(s)$ for the subscription s is: $T_1(s) = (T_l + T_p + T_f)_{B1} + (T_p + T_f)_{B2} + (T_l)_{B1}$, where the terms of this equation denote the six items of the path in the corresponding order.

In this experiment, even though the size of the subscription is different from that of a notification, the throughput for both is almost the same ($\simeq$ 188 Kbps). Therefore, $(T_l)_{B1}$ indicates the timing per unit-size of either a subscribing action from a subscriber

to $B1$ or a notification delivery from $B1$ to the subscriber. From the table, $(T_l)_{B1} =$ 1.02 ms per 24 bytes, and $(T_l)_{B2} \simeq 1.02$ ms per 24 bytes as well (i.e. 43.47 ms per 1020 bytes). Therefore, $(T_l)_{B1} = (T_l)_{B2}$ per unit-size. This equation meets Assumption 1 of Section 5.4.3.8.

The same discussion applies for $(T_f)_{B1} = 1.02$ ms and $(T_f)_{B2} \simeq 1.02$ ms per 24 bytes as well (i.e. 43.47 ms per 1020 bytes), where $(T_f)_{B1} = (T_f)_{B2}$ per unit-size. Thus, with $T_p = 1.90$ ms (from the regular-case of Case II experiments) in each broker for each subscription, $T_1(s) = 2{*}1.02 + 2\ {*}1.90 + 2{*}43.47 = 92.78$ ms; or $T(s) = 2{*}1.02 + 2\ {*}1.90$ $+\ 2{*}1.02 = 7.88$ ms if the size of the subscription were close to that of the notification.

For XPath Set3, there are 18 mismatched subscriptions in the local broker. These particular subscriptions take the route to the remote broker, where they have matched some publication data. Therefore, the end-to-end path delay for all these 18 subscriptions $T_1(s_{18}) = 18{*}2{*}1.02 + 18{*}2{*}1.90 + 18{*}2{*}43.47 = 1670.04$ ms, which is $\simeq 1.6$ seconds; or $T_1(s_{18}) = 18{*}2{*}1.02 + 18{*}2{*}1.90 + 18{*}2{*}1.02 = 141.84$ ms if the size of the subscription were close to that of the notification. The average end-to-end path delay for each subscription that has visited two brokers $T_1(s_{av_{18}}) = \frac{1670.04}{18} = 92.78$ ms $= T_1(s)$; or $T_1(s_{av_{18}}) = \frac{141.84}{18} = 7.88$ ms $= T_1(s)$ if the size of the subscription were close to that of the notification.

Among the total 252 subscriptions that make up XPath Set3, there are 234 subscriptions that have matched some publication data in the local broker. These subscriptions have not been routed to another broker. Therefore, the end-to-end path delay $T_2(s)$ for each of these subscriptions, say s, may take the following path:

1. Subscriber of subscription $s \xrightarrow{sub(o-SID,F)}$ Local Broker.

2. Match found in Remote Broker.

3. Local Broker $\xrightarrow{notify(N_F)}$ Subscriber of subscription s.

Thus, $T_2(s) = 1.02 + 1.90 + 43.47 = 46.39$ ms; or $T_2(s) = 1.02 + 1.90 + 1.02 = 3.94$ ms if the size of the subscription were close to that of the notification. Accordingly, $T_2(s_{234}) = 234*46.39 = 10855.26$ ms $\simeq$ 10.8 seconds; or $T_2(s_{234}) = 234*3.94 = 921.96$ ms $\simeq$ 0.9 second if the size of the subscription were close to that of the notification. The average end-to-end path delay for each subscription that has visited a single broker $T_2(s_{av_{234}}) = T_2(s) = 46.39$ ms; or $T_2(s_{av_{234}}) = 3.94$ ms if the size of the subscription were close to that of the notification. The reader may notice that $T_2(s) = \frac{1}{2}T_1(s)$.

The total end end-to-end path delay for all 252 subscriptions $T(s_{252}) = T_1(s_{18}) + T_2(s_{234}) = 1670.04 + 10855.26 = 12525.30$ ms $\simeq$ 12.50 seconds; or $T(s_{252}) = 141.84 + 921.96 = 1063.8$ ms $\simeq$ 1.06 seconds if the size of the subscription were close to that of the notification. This particular result is actually the longest time that a subscriber would wait to receive the notification. This waiting time would have been much less when multiple links would exist, so that the X2CBBR could concurrently receive and send data through different links (section 5.4.5).

7.10.1.2 Second Experiment Results

Table 7.9 shows the routing duration of the mismatched subscription as well as that of the corresponding publication. The routing results in this table for either a subscription or a notification are very similar to those of Table 7.8. However, since in this experiment a single mismatched subscription is only routed to another broker (Section 5.4.4.5), it is interesting to compare between the overall performance computed in the second experiment and that of the first experiment where 18 mismatched subscriptions were routed.

Table 7.9: Results of the Broker with Software/Hardware Interface and Subscription/Publication Routing

Content-Based Routing [Each Subscription 24 B.] (Each Notification 1020 B.)	Oper. Mode	Routing of 1 Subscr. (ms.)	Throughput (Kbps)	Routing of 1 Public. (ms.)	Throughput (Kbps)
	(oP,oS)	01.02	188.23	43.47	187.71

Based on the discussions of Section 5.4.3.7.1, just a single subscription s takes the following path:

1. Subscriber of subscription $s \xrightarrow{sub(o-SID,F)}$ Local Broker.
2. No match found in Local Broker.
3. Local Broker $\xrightarrow{sub(\{-----,B1\},F)}$ Remote Broker, where $B1$ is the BID of the local broker, and no subscription ID is included.
4. Match found in Remote Broker.
5. Remote Broker $\xrightarrow{forward(\{-----,B1,B2\},P_F)}$ Local Broker, where $B2$ is the BID of the remote broker, and P_F is the routed publication.
6. Re-match found in Local Broker.
7. Local Broker $\xrightarrow{notify(N_F)}$ Subscriber of subscription s.

The local broker does not route any of the other 251 subscriptions. However, the single-subscription routing time is actually the waiting time for other subscriptions, since the broker does not issue any notifications during this period of time. Therefore, the least possible end-to-end path delay $T(s)$ for each of the 252 subscriptions of XPath Set4 is as follows: $T(s) = (T_l + T_p + T_f)_{B1} + (T_p + T_f)_{B2} + (T_p)_{B1} + (T_l)_{B1}$, where the terms of this equation denote the seven items of the path in the corresponding order.

As in the previous experiment, the throughput for routing either a subscription of a notification is almost the same ($\simeq$ 188 Kbps). From the table, $(T_l)_{B1} = 1.02$ ms per 24 bytes, and $(T_l)_{B2} \simeq 1.02$ ms per 24 bytes as well (i.e. 43.47 ms per 1020 bytes). Therefore, $(T_l)_{B1} = (T_l)_{B2}$ per unit-size.

The same discussion applies for $(T_f)_{B1} = 1.02$ ms and $(T_f)_{B2} \simeq 1.02$ ms per 24 bytes as well (i.e. 43.47 ms per 1020 bytes), where $(T_f)_{B1} = (T_f)_{B2}$ per unit-size. Thus, with $T_p = 1.90$ ms, $T(s) = (1.02 + 1.90 + 1.02) + (1.90 + 43.47) + 1.90 + 43.47 = 94.68$ ms; or $T(s) = (1.02 + 1.90 + 1.02) + (1.90 + 43.47) + 1.90 + 1.02 = 52.23$ ms if the size of the subscription were close to that of the notification.

The end-to-end path delay for all 252 subscriptions $T(s_{252})$ = 252*(1.02 + 1.90 + 1.02) + 1*(1.90 + 43.47) + 1*1.90 + 252*43.47 = 992.88 + 45.37 + 1.90 + 10954.44 = 11.994.59 ms $\simeq$ 12 seconds; or $T(s_{252})$ = 252*(1.02 + 1.90 + 1.02) + 1*(1.90 + 43.47) + 1*1.90 + 252*1.02 = 992.88 + 45.37 + 1.90 + 257.04 = 1297.19 ms $\simeq$ 1.3 seconds if the size of the subscription were close to that of the notification. This particular result represents the worst-case in terms of link management, because the X2CBBR would be able to concurrently manage six external links (Section 5.4.5) when they physically exist.

7.10.1.3 Comparison Between First and Second Experiments

Referring to the results of both experiments of Case III, Table 7.10 provides the minimum waiting time for a subscriber who sends a subscription up to the reception of the corresponding notification in the first and second experiments. There are two cases in the table for each experiment: (a) When the notification size is 1020 bytes, which is much greater than that of a subscription (24 bytes), and (2) When the notification size is very close to that of a subscription (24 bytes). In both (a) and (b) the publication size is 1020 bytes.

Table 7.10: Minimum Time Consumed for a Subscriber to Send a Subscription and Receive a Notification in the First and the Second Experiments of Case III

One Notif.(a) (*bytes*)	One Notif.(b) (*bytes*)	1st. Exper. Min.Time(a) (*ms.*)	1st. Exper. Min.Time(b) (*ms.*)	2nd. Exper. Min.Time(a) (*ms.*)	2nd. Exper. Min.Time(b) (*ms.*)
1020	**24**	**46.39**	**03.94**	**94.68**	**52.23**

The minimum waiting time for a subscriber in the first experiment (46.39 ms or 3.94 ms) is the duration needed to receive a notification when a subscription visits a single broker. In the second experiment, a single subscription visits two brokers, and the matching process runs twice in the first broker for all subscriptions. The minimum waiting time in this case is surely higher (94.68 ms or 52.23 ms). These expected results confirm that a subscriber whose subscription visits more than a broker must wait longer to receive the relevant notification.

The overall performance for all 255 subscriptions is the overall accumulated time for all 252 subscriptions to travel to the first broker and all relevant notifications reach corresponding subscribers. Table 7.11 compares between the overall performance computed in the first experiment and that of the second experiment. This table indicates the waiting time for 255 subscribers from the starting point of the first subscription issued by the first subscriber up to the delivery of the last notification to the last subscriber.

Table 7.11: Overall Performance Comparison Between the First and the Second Experiments of Case III, with Sequential Data Transmission

Total Nb. of Subs.	1st. Exper. Perform. (a) (*sec.*)	1st. Exper. Perform. (b) (*sec.*)	2nd. Exper. Perform. (a) (*sec.*)	2nd. Exper. Perform. (b) (*sec.*)
252	**12.50**	**01.06**	**12.00**	**01.30**

While the maximum waiting time in the first experiment is 12.5 sec for a subscriber to receive the relevant notification, the results of this table indicate that a subscriber

in the second experiment may never wait more than 12 sec to receive a notification, even though all subscriptions of the second experiment have not originally matched any publication. These results prove the effectiveness of the routing algorithms, where a single subscription routed on behalf of many subscriptions can save much routing time. When the notification size is close to the subscription size, the maximum waiting time in the second experiment (1.3 sec) is slighter greater that that of the first experiment (1.06 sec), even though that the matching process runs twice for all subscriptions registered in the first broker. This overall performance also reflects the efficiency of the X2CBBR in processing and matching subscriptions against publications.

However, the subsequent transmission of data in both experiments leads to these worst-case results in terms of link management. Section 5.4.5 has pointed to the X2CBBR ability of concurrently managing six external links to highly improve performance.

7.11 Cyclone IV E FPGA Resources Utilization

The X2CBBR and its hardware interface XAVI are implemented on Altera's Cyclone IV E FPGA EP4CE115F29C7 device [15], using the Altera Quartus II synthesis and placement and route tool [16].

The corresponding implementation results indicate that the whole design consumes 79,072 LEs out of the total 114,480 LEs that are available on the chip. The design area represents 69% of the total area of the chip that is manufactured with 60-nm of process technology. Moreover, the total memory bits in use reach 1,580,588 out of the maximum 3,981,312 bits that can be available on the chip, which indicates a usage percentage of 40%. Since multiple clocks are figured in different parts of the whole design, three out of four PLLs are employed.

7.12 Power Dissipation on Cyclone IV E FPGA

The total thermal power dissipation of the design implemented on the Cyclone IV E FPGA device is 1803.96 mW. This result is 11.3% lower than that of the design of Phase 1 (2034.39 mW) that was implemented on Stratix FPGA. In particular, the core static portion of the total thermal power dissipation is just 123.64 mW, which indicates 83% power reduction comparing to the design that was implemented on Stratix (729.90 mW). It is the core dynamic portion of the thermal power dissipation (1179.68 mW) that is twice higher than the one measured on Stratix (579.93 mW). However, the I/O thermal power dissipation is 500.64 mW, which indicates reduction of around 31% comparing to what was previously found on Stratix (724.57 mW).

This improvement in power dissipation occurs despite the fact that the broker becomes almost twice larger in Phase 2, with 79,072 LEs versus 40,835 LEs utilized in Phase 1. This result indicates the positive effect of the recent process technology on the power dissipation of a chip. It would be interesting to mention that a process technology of 28-nm is currently available in recent FPGA devices.

7.13 Broker's Phase 2 Conclusion

In Phase 2 experiments, the X2CBBR has undergone three main test groups, where each group runs with a different interface: (1) one with a sole hardware interface, another one with a software/hardware interface, and a third one with the software/hardware interface and Ethernet connectivity.

When the sole hardware interface is used with the Avalon bus, the broker can regularly achieve 372 Mbps of throughput and even 592 Mbps of throughput for the best case. However, when the software/hardware interface is used, the performance significantly

drops.

The experiments show that the X2CBBR can efficiently operate with content-based routing. However, the broker-to-broker connectivity has significant impact on the overall performance. Even though the Nichestack TCP/IP stack used in Case III experiments is lightweight, the Ethernet throughput does not reach high values. This particular result indicates that the provided FPGA macros for TCP/IP and Ethernet can be considered as entities having the "proof-of-concept" functionality rather than targeting a high-pitch speed value.

To better achieve higher performance, the X2CBBR would need to be interfaced with a bus more sophisticated than Avalon. For example, a hardware interface utilizing the PCIe bus specification would be a future candidate. In addition, implementing the X2CBBR on an ASIC chip would certainly lead to much higher performance, since a more advanced process technology along with a higher clock frequency can be achieved.

Chapter 8

Conclusion and Future Work

This thesis has studied in details a hardware architecture of an XML/XPath content-based broker/router for publish/subscribe data dissemination systems. The X2CBBR hardware includes an XML parser, an XPath processor, and various memory and interface modules. Moreover, the X2CBBR architecture employs various algorithms that provide efficient mechanisms to store and route subscriptions, publications, and notifications.

The hardware implementation results indicate that a reconfigurable prototype of the X2CBBR, with a hardware interface, can reach nearly 0.6 Gbps of throughput on an FPGA chip. In the future, an ASIC implementation should provide even higher performance.

The X2CBBR can efficiently operate in either centralized or distributed pub/sub systems. Since it receives publications in XML format, the X2CBBR can smoothly interoperate with any entity that accepts XML data. Moreover, the usage of the SCBXP hardware technique in the task of XML parsing allows the broker to deal with heterogeneous XML publication data sources. With regard of subscriptions, the architecture provides efficient storage that exploits their commonalities and improves subscription

processing.

The next section discusses the limitations of the X2CBBR architecture, stating how a future work may address relevant situations.

8.1 Applicability in Wide-Area Networks

In large-scale content-based pub/sub networks, this thesis has tackled the subject of "network scoping" (Section 5.4.6) that greatly enhances the large-scale network coverage of the proposed XML/XPath broker/router overlay network. However, such enhancement would only occur with the support of entities that appropriately know and manage the topology that makes up the large-scale network. As mentioned in Section 5.4.6, an entity would know the topology of the network in its scope, while other network scopes may be out of the visibility of the entity. However, would entity-to-entity communication that crosses adjacent network scopes be needed?

The scope of this thesis has not covered the inner nature or structure of such an entity. The thesis has just indicated that such entities would have the role of knowing and managing the network topology, feeding the VRT in each broker, and recommending publication solicitation. In future works, it would be interesting to develop such topology entities, so that an entity-to-entity communication in adjacent network scopes can be established. For example, a Wide-Area Network (WAN) protocol (e.g. IP combined with 10-Gbps Ethernet) can be used in the communication between two topology entities located in any two distant network scopes.

Another interesting future work would target the multimedia support (Section 3.2.5) that the X2CBBR can provide in WANs. For example, it is possible to test the ALM (application-level multicast) in the broker network along with network scoping.

8.1.1 Content-Based Routing With or Without TCP/IP

The X2CBBR utilizes content-based routing, and determines the neighboring broker to which messages are to be forwarded without taking into account any IP addresses. This routing scheme is in contrast of IP routing that forwards messages based on IP addresses resolved in the Domain Name System (DNS). The main role of the DNS in the Internet is to map names to IP network locations, making use of its globally-distributed database and its caching techniques [57].

Since the mechanism of content-based routing does not rely on IP addresses, the broker-to-broker communication can be accomplished without any role associated to IP routing. The X2CBBRs (each characterized with its Broker ID) utilize the pub/sub paradigm; and each subscription has an ID structured in a specific way that was elaborated in Section 5.3.2.1 and Figure 5.7. These subscriptions are routed based on their content, taking into account the subscription IDs and the broker IDs. Moreover, the notifications reach the relevant subscribers through content-based routers tracing back the path of the involved brokers with the support of the subscription IDs and the broker IDs (Section 5.4.3.5). Similarly, the publications are routed to the edge broker tracing back the path of brokers recognized through the broker IDs (Section 5.4.3.5.2.1). Therefore, the source broker and the destination broker can exchange, recognize, process, and route these messages without the need of IP routing.

Therefore, the following question can be raised: is it possible that two distant X2CBBRs communicate effectively without the need of the popular stack TCP/IP at all?

The response to this question is that the communication between two distant content-based routers needs a sort of stack anyway for many reasons. First, a reliable transport protocol would be needed to ensure that no errors of transmission have occurred. For

example, TCP is a reliable transport protocol that performs congestion control and packet retransmission. Second, the TCP/IP stack offers many services and QoS features. Moreover, the IP packets can be effectively encapsulated into Ethernet frames, whether Ethernet operates in LANs, MANs, or WANs. Therefore, an alternative stack needs to address the transport reliability and the ability of encapsulating content effectively in the data-link frames transferred over the physical link, such as Ethernet frames.

Recently, there has been a very ambitious attempt (Ph.D. dissertation) to engineer a transport protocol for best-effort content-based networks without the support of TCP/IP [73]. However, in this transport protocol, the congestion control mechanism borrows ideas from the reliability and congestion control protocols that are used for IP multicast. Moreover, the protocol handles content-based flows with coexistence of TCP flows, in the sense that TCP friendliness and fairness are both maintained.

In the future, it would be interesting to seek such solutions with the hope that extensive relevant research proves high performance with smooth traffic deployment.

Presently, the overlay broker network operates over the underlying IP networks, where IP is the protocol of communication between adjacent IP routers. Therefore, when a broker needs to communicate with another one, there should be a procedure to map the broker ID to the network IP address of the nearest IP router. This mapping procedure should occur in the source and the destination brokers, while normal IP routing occurs among IP routers. In the provided experiments, a static IP address is assigned to the X2CBBR for simplicity (Chapter 7). In the future, a dynamic mapping approach should enhance the smooth operation of the overlay broker network over IP networks. However, researchers have indicated that some issues might stem from the complex interaction between overlay routing and underlay routing [108][125].

8.2 Interfacing Multiple X2CBBRs

This thesis has shown that the hardware interface that drives an X2CBBR yields high performance. However, when a software/hardware interface drives the X2CBBR the performance drops. This problem commonly occurs when interfacing any hardware design with software. The reason of the occurrence of this problem is partially due to the fact that the language used in developing the software is completely different from the one used in developing the hardware. Even though some commercial tools try to compile a software-based language into RTL code, this conversion often takes place inefficiently. Furthermore, when new software data regularly has to drive the hardware, this conversion may need to be redone, which is significantly impractical. In the literature, there are recent attempts to automatically generate software/hardware interfaces using the Bluespec Codesign Language (BCL) "which permits the designer to specify the hardware-software partition in the source code" [58].

Other reasons behind the drop in interfacing performance include the bus interface and the techniques that are employed to accomplish the data transfer from the software to the hardware and vice-versa. Even though the Avalon bus, used to interface the FPGA chip in this thesis, can support burst data to accelerate data transfer, there is immense need to find innovative software/hardware interfacing solutions in the future. Such solutions would allow a highly efficient network of multiple X2CBBRs to smoothly and efficiently interoperate, even if the X2CBBRs remain in their FPGA-based implementations. However, implementing the X2CBBR in ASICs with either a PCIe or a next-generation interface would be an interesting future endeavour.

8.3 Security Challenges

The X2CBBR guarantees total decoupling between publishers and subscribers. This feature is a positive advantage for the pub/sub system made up of such brokers. Moreover, being implemented in hardware, the broker is generally less vulnerable than any alternative software implementation. For example, flooding the broker with continuous subscriptions would not lead to a Denial-of-Service (DoS) scenario, because the subscription buffer simply does not accept more than its capacity of data.

However, security issues may still exist. Since this thesis has not addressed such issues, future work can augment the current architecture with security mechanisms. Such mechanisms may include authentication to provide subscription integrity and to avoid compromises.

Bibliography

[1] Document Object Model (DOM). [Online]. http://www.w3.org/DOM, *Accessed July 2012*.

[2] PSIRP Project. [Online]. http://www.psirp.org, *Accessed July 2012*.

[3] RosettaNet. [Online]. http://www.rosettanet.org/, *Accessed July 2012*.

[4] Simple API for XML (SAX). [Online]. http://www.saxproject.org, *Accessed July 2012*.

[5] Streaming API for XML (StAX). [Online]. http://jcp.org/en/jsr/detail?id=173, *Accessed July 2012*.

[6] TV-Anytime Forum. [Online]. http://www.tv-anytime.org/, *Accessed July 2012*.

[7] I. Aekaterinidis and P. Triantafillou. "PastryStrings: A Comprehensive Content-Based Publish/Subscribe DHT Network". *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06), Portugal*, July 2006.

[8] J. P. Ahulló, P. G. López, and A. F. G. Skarmeta. "LightPS: Lightweight Content-based Publish/Subscribe for Peer-to-Peer Systems". *International Conference on Complex, Intelligent and Software Intensive Systems, Barcelona, Spain*, March 2008.

[9] Altera. "Triple-Speed Ethernet MegaCore Function User Guide". [Online]. http://www.altera.com/literature/ug/ug_ethernet.pdf, January 2013, *Accessed April 2013*.

[10] Altera. "Embedded Design Handbook". [Online]. http://www.altera.com/literature/hb/nios2/edh_ed_handbook.pdf, July 2011, *Accessed April 2013*.

[11] Altera. "Embedded Peripherals IP User Guide". [Online]. http://www.altera.com/literature/ug/ug_embedded_ip.pdf, June 2011, *Accessed April 2013*.

[12] Altera. "Avalon Interface Specifications". [Online]. http://www.altera.com/literature/manual/mnl_avalon_spec.pdf, May 2011, *Accessed March 2013*.

[13] Altera. "Nios II Software Developer's Handbook". [Online]. http://www.altera.com/literature/hb/nios2/n2sw_nii5v2.pdf, May 2011, *Accessed March 2013*.

[14] Altera. "Qsys - Altera's System Integration Tool". [Online]. http://www.altera.com/products/software/quartus-ii/subscription-edition/qsys/qts-qsys.html, *Accessed April 2013*.

[15] Altera. "Cyclone IV FPGA Family Overview". [Online]. http://www.altera.com/devices/fpga/cyclone-iv/overview/cyiv-overview.html, *Accessed July 2012*.

[16] Altera. "Quartus II - Design Entry and Synthesis". [Online]. http://www.altera.com/products/software/quartus-ii/subscription-edition/design-entry-synthesis/qts-des-ent-syn.html, *Accessed July 2012*.

[17] Altera. "Stratix FPGA Family Overview". [Online]. http://www.altera.com/devices/fpga/stratix-fpgas/stratix/stratix/overview/stx-overview.html, *Accessed July 2012*.

[18] Altera. "Stratix III FPGA Family Overview". [Online]. http://www.altera.com/products/devices/stratix-fpgas/stratix-iii/overview/st3-overview.html, *Accessed July 2012*.

[19] Altera. "Stratix IV FPGA Family Overview". [Online]. http://www.altera.com/devices/fpga/stratix-fpgas/stratix-iv/overview/stxiv-overview.html, *Accessed July 2012*.

[20] M. Altinel and M. J. Franklin. "Efficient Filtering of XML Documents for Selective Dissemination of Information". *26th International Conference on Very Large Data Bases (VLDB 2000), Cairo, Egypt*, 2000.

[21] G. Banavar, T. D. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. "An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems". *International Conference on Distributed Computing Systems (ICDCS'99), Austin, Texas, USA*, June 1999.

[22] L. F. Cabrera, M. B. Jones, and M. Theimer. "Herald: Achieving a Global Event Notification Service". *8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), Germany*, May 2001.

[23] F. Cao and J. P. Singh. "Efficient Event Routing in Content-based Publish-Subscribe Service Networks". *IEEE INFOCOM 2004*, March 2004.

[24] F. Cao and J. P. Singh. "Medym: Match-Early with Dynamic Multicast for Content-Based Publish-Subscribe Networks". *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware (Middleware 2005)*, November 2005.

[25] A. Carzaniga and C. P. Hall. "Content-Based Communication: a Research Agenda". *Proceedings of the 6th International Workshop on Software Engineering and Middleware, SEM 2006, Portland, Oregon, USA*, November 2006.

[26] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. "Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service". *In The 19th ACM Symposium on Principles of Distributed Computing (PODC2000), Portland, OR, USA*, July 2000.

[27] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. "Design and Evaluation of a Wide-Area Event Notification Service". *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[28] A. Carzaniga, M. J. Rutherford, and A. L. Wolf. "A Routing Scheme for Content-Based Networking". *INFOCOM 2004*, 2004.

[29] A. Carzaniga and A. L. Wolf. "Content-Based Networking: A New Communication Infrastructure". *Proceedings of the NSF Workshop on an Infrastructure for Mobile and Wireless Systems, Scottsdale, AZ, USA, and in Lecture Notes in Computer Science (LNCS), Springer-Verlag*, October 2001.

[30] A. Carzaniga and A. L. Wolf. "Forwarding in a Content-Based Network". *ACM SIGCOMM'03, Karlsruhe, Germany*, August 2003.

[31] M. Castro, P. Druschel, A.-M. Kermarrec, and A. I. T. Rowstron. "Scribe: A Large-Scale and Decentralized Application-Level Multicast Infrastructure". *IEEE Journal on Selected Areas in Communications*, 20(8), October 2002.

[32] D. Chamberlin, J. Robie, and D. Florescu. "Quilt: An XML Query Language for Heterogeneous Data Sources". *The World Wide Web and Databases, Third International Workshop (WebDB 2000), Dallas, TX, USA*, May 2000.

[33] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. "Efficient Filtering of XML Documents with XPath Expressions". *18th International Conference on Data Engineering (ICDE 2002), San Jose, CA, USA*, February 2002.

[34] C.-Y. Chan and Y. Ni. "Content-based Dissemination of Fragmented XML Data". *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS'06), Lisboa, Portugal*, July 2006.

[35] R. Chand and P. Felber. "XNET: A Reliable Content-Based Publish/Subscribe System". *23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04), Florianpolis, Brazil*, October 2004.

[36] R. Chand and P. Felber. "Scalable Distribution of XML Content with XNET". *IEEE Transactions on Parallel and Distributed Systems*, 19(4):447–461, April 2008.

[37] S.-F. Chang, T. Sikora, and A. Puri. "Overview of the MPEG-7 Standard". *IEEE Transactions on Circuits and Systems for Video Technology*, 11(6):688–695, June 2001.

[38] K. Chiu and W. Lu. "A Compiler-based Approach to Schema-specific XML Parsing". *The first International Workshop on High Performance XML Processing, NY, USA*, May 2004.

[39] Y. Choi and D. Park. “Mirinae: A Peer-to-Peer Overlay Network for Content-Based Publish/Subscribe Systems”. *IEICE Transactions on Communications*, E89-B(6): 1755–1765, June 2006.

[40] G. Cugola, E. D. Nitto, and A. Fuggetta. “The JEDI Event-based Infrastructure and its Application to the Development of the OPSS WFMS”. *IEEE Transactions on Software Engineering*, 27(9):827–850, September 2001.

[41] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. “Path Sharing and Predicate Evaluation for High-Performance XML Filtering”. *ACM Transactions on Database Systems*, 28(4):467–516, December 2003.

[42] Y. Diao, S. Rizvi, and M. J. Franklin. “Towards an Internet-Scale XML Dissemination Service”. *30th International Conference on Very Large Data Bases (VLDB 2004), Toronto, Canada*, August/September 2004.

[43] N. Earnshaw, S. Aoki, A. Ashley, and W. Kameyama. “The TV-Anytime Content Reference Identifier (CRID)”. *RFC 4078, IETF*, May 2005.

[44] F. El-Hassan and D. Ionescu. “SCBXP: An Efficient Hardware-based XML Parsing Technique”. *The 5th IEEE Southern conference on Programmable Logic (SPL’09), São Carlos, Brazil*, April 2009.

[45] F. El-Hassan and D. Ionescu. “A Hardware Architecture of an XML/XPath Broker for Content-Based Publish/Subscribe Systems”. *International Conference on Reconfigurable Computing and FPGAs (ReConFig 2010), Cancun, Mexico*, December 2010.

[46] F. El-Hassan and D. Ionescu. "SCBXP: An Efficient CAM-Based XML Parsing Technique in Hardware Environments". *IEEE Transactions on Parallel and Distributed Systems*, 22(11):1879–1887, November 2011.

[47] F. El-Hassan, R. Peterkin, M. Abou-Gabal, and D. Ionescu. "A High-Performance Architecture of an XML Processor For SIP-based Presence". *The 6th IEEE International Conference on Information Technology: New Generations (ITNG'09), Las Vegas, USA*, April 2009.

[48] P. Eugster. "Type-Based Publish/Subscribe: Concepts and Experiences". *ACM Transactions on Programming Languages and Systems*, 29(1), January 2007.

[49] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. "The Many Faces of Publish/Subscribe". *ACM Computing Surveys*, 35(2):114–131, June 2003.

[50] F. Farfán, V. Hristidis, and R. Rangaswami. "2LP - A Double Lazy XML Parser". *Elsevier Information Systems Journal*, 34(1):145–163, March 2009.

[51] A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi. "Meghdoot: Content-Based Publish/Subscribe over P2P Networks". *5th ACM/IFIP/USENIX International Middleware Conference (Middleware 2004), Toronto, Ontario, Canada*, October 2004.

[52] IBM. "IBM WebSphere DataPower XML Accelerator XA35". [Online]. http://publib.boulder.ibm.com/infocenter/wsdatap/v3r8m2/index.jsp?topic=%2Fdp%2Fwebgui_xa35.htm, *Accessed July 2012*.

[53] IEEE Standards Association. "IEEE 802.3: ETHERNET". [Online]. http://standards.ieee.org/about/get/802/802.3.html, *Accessed April 2013*.

[54] A. S. Inc. "Extensible Metadata Platform (XMP): Adding intelligence to media". [Online]. http://www.adobe.com/products/xmp/index.html, November 2004, *Accessed July 2012*.

[55] D. Ionescu, R. E. Peterkin, F. El-Hassan, and M. Abou-Gabal. "Reconfigurable Multimedia Collaboration System". *US Patent, US8281039*, October 2012.

[56] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. "Networking Named Content". *The 5th ACM International Conference on emerging Networking EXperiments and Technologies (ACM CoNEXT), Rome, Italy*, December 2009.

[57] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. "DNS Performance and the Effectiveness of Caching". *IEEE/ACM Transactions on Networking*, 10(5):589–603, October 2002.

[58] M. King, N. Dave, and Arvind. "Automatic Generation of Hardware/Software Interfaces". *17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012), London, UK*, March 2012.

[59] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. "A Data-Oriented (and Beyond) Network Architecture". *ACM SIGCOMM'07, Kyoto, Japan*, August 2007.

[60] M. G. Kostoulas, M. Matsa, N. Mendelsohn, E. Perkins, A. Heifets, and M. Mercaldi. "XML Screamer: An Integrated Approach to High Performance XML Parsing, Validation and Deserialization". *The 15th International World Wide Web Conference (www2006), Edinburgh, Scotland*, May 2006.

[61] R. Krishnamoorthy. "Hardware Implementation of an XML Parser". *Master's Thesis, North Carolina State University*, 2008.

[62] J. J. Labrosse. *MicroC/OS-II - The Real-Time Kernel*. *CMPBooks*, 2nd edition, 2002. ISBN 1-57820-103-9.

[63] T. C. Lam, J. J. Ding, and J.-C. Liu. "XML Document Parsing - Operational and Performance Characteristics". *IEEE Computer Magazine*, 41(9):30–37, September 2008.

[64] Z. Lei. "XML Parsing Accelerator with Intel Streaming SIMD Extensions 4 (Intel SSE4)". Intel Software Network. [Online]. http://software.intel.com/en-us/articles/xml-parsing-accelerator-with-intel-streaming-simd-extensions-4-intel-sse4, *Accessed July 2012*.

[65] M. Leventhal. "Random Access XML Programming Assisted with XML Hardware". *In Conference Proceedings of XML 2004, Washington, USA*, November 2004.

[66] G. Li and H.-A. Jacobsen. "Composite Subscriptions in Content-Based Publish/Subscribe Systems". *Lecture Notes in Computer Science (LNCS), Springer*, 3790:249–269, 2005.

[67] G. Li, V. Muthusamy, and H.-A. Jacobsen. "Adaptive Content-Based Routing in General Overlay Topologies". *ACM/IFIP/USENIX 9th International Middleware Conference (Middleware 2008), Leuven, Belgium*, December 2008.

[68] M. Lonnfors, E. Leppanen, H. Khartabil, and J. Urpalainen. "Presence Information Data Format (PIDF) Extension for Partial Presence". *RFC 5262, IETF*, September 2008.

[69] W. Lu, K. Chiu, and Y. Pan. "A Parallel Approach to XML Parsing". *The 7th IEEE/ACM International Conference on Grid Computing (GRID 2006), Barcelona, Spain*, September 2006.

[70] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. "A Survey and Comparison of Peer-to-Peer Overlay Network Schemes". *IEEE Communications Surveys and Tutorials*, 7(2):72–93, Second Quarter 2005.

[71] J. V. Lunteren, T. Engbersen, J. Bostian, B. Carey, and C. Larsson. "XML Accelerator Engine". *The 1st International Workshop on High Performance XML Processing, NY, USA*, May 2004.

[72] Q. H. Mahmoud. "Getting Started with Java Message Service (JMS)". [Online]. http://www.oracle.com/technetwork/articles/java/introjms-1577110.html, November 2004, *Accessed March 2013*.

[73] A. Malekpour. *A Transport Protocol for Best-Effort Content-Based Networks.* Ph.D. dissertation, Università della Svizzera Italiana, Switzerland, November 2012.

[74] J. L. Martins and S. Duarte. "Routing Algorithms for Content-Based Publish/Subscribe Systems". *IEEE Communications Surveys and Tutorials*, 12(1): 39–58, First Quarter 2010.

[75] P. Maymounkov and D. Mazières. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric". *Lecture Notes in Computer Science (LNCS), Springer*, 2429:53–65, 2002.

[76] K. McLaughlin, N. O'Connor, and S. Sezer. "Exploring CAM Design For Network Processing Using FPGA Technology". *AICT/ICIW'06, Guadeloupe*, February 2006.

[77] G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems.* Ph.D. dissertation, Darmstadt University of Technology, Darmstadt, Germany, Aug.-Sept. 2002.

[78] G. Mühl, L. Fiege, and P. R. Pietzuch. *Distributed Event-Based Systems.* Springer-Verlag Berlin Heidelberg, Germany, 2006. ISBN 3-540-32651-0.

[79] Mentor Graphics. "ModelSim". [Online]. http://www.model.com/, *Accessed July 2012*.

[80] A. Mitra, M. R. Vieira, P. Bakalov, W. Najjar, and V. J. Tsotras. "Boosting XML Filtering with a Scalable FPGA-based Architecture". *4th Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar, California, USA*, January 2009.

[81] J. Moscola, J. W. Lockwood, and Y. H. Cho. "Reconfigurable Content-Based Router Using Hardware-Accelerated Language Parser". *ACM Transactions on Design Automation of Electronic Systems*, 13(2), April 2008.

[82] R. Moussalli, M. Salloum, W. Najjar, and V. Tsotras. "Accelerating XML Query Matching through Custom Stack Generation on FPGAs". *International Con-*

ference on High-Performance Embedded Architectures and Compilers (HiPEAC), Pisa, Italy, January 2010.

[83] R. Moussalli, M. Salloum, W. Najjar, and V. Tsotras. "Massively Parallel XML Twig Filtering Using Dynamic Programming on FPGAs". *The IEEE 27th International Conference on Data Engineering (ICDE 2011), Hannover, Germany*, April 2011.

[84] B. Nag. "Acceleration Techniques for XML Processors". *XML 2004 Proceedings by SchemaSoft, Washington, USA*, November 2004.

[85] M. Nicola and J. John. "XML Parsing: A Threat to Database Performance". *The 12th International Conference on Information and Knowledge Management (CIKM'03), New Orleans, USA*, November 2003.

[86] M. L. Noga, S. Schott, and W. Löwe. "Lazy XML Processing". *ACM Symposium on Document Engineering (DocEng'02), McLean, Virginia, USA*, November 2002.

[87] Object Management Group (OMG). "Part 3: CORBA Component Model". *OMG Document Number: formal/2008-01-08*, [Online]. http://www.omg.org/spec/CORBA/3.1/Components/PDF/, January 2008, *Accessed October 2010*.

[88] Object Management Group (OMG). "The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification (DDS-RTPS) - Version 2.1". *OMG Document Number: formal/2010-11-01*, [Online]. http://www.omg.org/spec/DDS-RTPS/2.1/PDF/, November 2010.

[89] Object Management Group (OMG). "Common Object Request Broker Architecture (CORBA)". [Online]. http://www.corba.org/, *Accessed July 2012*.

[90] Object Management Group (OMG). "Data Distribution Intro". [Online]. http://portals.omg.org/dds/content/data-distribution-intro, *Accessed July 2012*.

[91] K. Pagiamtzis and A. Sheikholeslami. "Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey". *IEEE Journal of Solid-State Circuits*, 41(3), March 2006.

[92] S. Pallickara and G. Fox. "NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids". *4th ACM/IFIP/USENIX International Middleware Conference (Middleware 2003), Rio de Janeiro, Brazil*, June 2003.

[93] Y. Pan, W. Lu, Y. Zhang, and K. Chiu. "A Static Load-Balancing Scheme for Parallel XML Parsing on Multicore CPUs". *The 7th International Symposium on Cluster Computing and the Grid (CCGrid'07), Rio de Janeiro, Brazil*, May 2007.

[94] F. Pereira, A. Vetro, and T. Sikora. "Multimedia Retrieval and Delivery: Essential Metadata Challenges and Standards". *Proceedings of the IEEE*, 96(4):721–744, April 2008.

[95] J. Pereira, F. Fabret, F. Llirbat, H. A. Jacobsen, and D. Shasha. "WebFilter: A High-throughput XML-based Publish and Subscribe System". *27th International Conference on Very Large Data Bases (VLDB 2001), Roma, Italy*, September 2001.

[96] J. Pereira, F. Fabret, F. Llirbat, R. Preotiuc-pietro, K. A. Ross, and D. Shasha. "Publish and Subscribe on the Web at Extreme Speed". *26th International Conference on Very Large Data Bases (VLDB 2000), Cairo, Egypt*, September 2000.

[97] R. Peterkin, M. Abou-Gabal, F. El-Hassan, and D. Ionescu. “Hardware Implementation of Session Initiation Protocol Servers and Clients”. *The 14th IEEE Symposium on Computers and Communications (ISCC 2009), Sousse, Tunisia*, July 2009.

[98] R. Peterkin, F. El-Hassan, and D. Ionescu. “A Reconfigurable Architecture for IP Multimedia Subsystem Session Setup”. *The IEEE International Joint Conferences on Computational Cybernetics and Technical Informatics (ICCC-CONTI 2010), Timisora, Romania*, May 2010.

[99] R. Peterkin and D. Ionescu. “An Architecture for Heterogeneous Data Dissemination Using IMS”. *The 26th IEEE International Conference on Advanced Information Networking and Applications (AINA-2012), Fukuoka, Japan*, March 2012.

[100] P. R. Pietzuch. *Hermes: A Scalable Event-Based Middleware*. Ph.D. dissertation, University of Cambridge, Cambridge, UK, February 2004.

[101] P. R. Pietzuch and J. M. Bacon. “Hermes: A Distributed Event-Based Middleware Architecture”. *22nd International Conference on Distributed Computing Systems Workshops (ICDCSW’02), Vienna, Austria*, July 2002.

[102] C. Raiciu, D. S. Rosenblum, and M. Handley. “Revisiting Content-Based Publish/Subscribe”. *26th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW’06), Lisboa, Portugal*, July 2006.

[103] V. Ramasubramanian, R. Peterson, and E. G. Sirer. “Corona: A High Performance Publish-Subscribe System for the World Wide Web”. *3rd Symposium on Networked Systems Design and Implementation (NSDI’06), San Jose, California, USA*, May 2006.

[104] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. "A Scalable Content-Addressable Network". *ACM SIGCOMM'01, San Diego, California, USA*, August 2001.

[105] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. "Application-Level Multicast Using Content-Addressable Networks". *Lecture Notes in Computer Science (LNCS), Springer*, 2233:14–29, 2001.

[106] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. "SIP: Session Initiation Protocol". *RFC 3261, IETF*, June 2002.

[107] A. Rowstron and P. Drusche. "Pastry: Scalable, Distributed Object Location, and Routing for Large-Scale Peer-to-Peer Systems". *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), Heidelberg, Germany*, November 2001.

[108] S. Seetharaman, V. Hilt, M. Hofmann, and M. Ammar. "Resolving Cross-Layer Conflict Between Overlay Routing and Traffic Engineering". *IEEE/ACM Transactions on Networking*, 17(6):1964–1977, December 2009.

[109] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. "Content Based Routing with Elvin4". *In AUUG2K, Canberra, Australia*, June 2000.

[110] A. C. Snoeren, K. Conley, and D. K. Gifford. "Mesh-Based Content Routing using XML". *18th ACM Symposium on Operating Systems Principles, Banff, Canada*, October 2001.

[111] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications". *ACM SIGCOMM'01, San Diego, California, USA*, August 2001.

[112] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. "Gryphon: An Information Flow Based Approach to Message Brokering". *9th International Symposium on Software Reliability Engineering (ISSRE'98), Fast Abstract, Paderborn, Germany*, November 1998.

[113] H. Sugano, S. Fujimoto, G. Klyne, A. Bateman, W. Carr, and J. Peterson. "Presence Information Data Format (PIDF)". *RFC 3863, IETF*, August 2004.

[114] T. Takase, H. Miyashita, T. Suzumura, and M. Tatsubori. "An Adaptive, Fast, and Safe XML Parser Based on Byte Sequences Memorization". *The 14th International World Wide Web Conference (www2005), Chiba, Japan*, May 2005.

[115] Terasic. "Altera DE2-115 Development and Education Board". [Online]. http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=502, *Accessed July 2012*.

[116] TIBCO Software Inc. "TIBCO Rendezvous". [Online]. *Accessed March 2013*, http://www.tibco.com/products/automation/messaging/low-latency/rendezvous/.

[117] G. Wang, C. Xu, Y. Li, and Y. Chen. "Analyzing XML Parser Memory Characteristics: Experiments towards Improving Web Services Performance". *IEEE International Conference on Web Services (ICWS'06), Chicago, USA*, September 2006.

[118] World Wide Web Consortium (W3C). "Canonical XML Version 2.0". January 2012, [Online]. *Accessed July 2012*, http://www.w3.org/TR/xml-c14n2/.

[119] World Wide Web Consortium (W3C). "Extensible Markup Language (XML) 1.0 (Fifth Edition)". February 2008, [Online]. *Accessed July 2012*, http://www.w3.org/TR/2008/PER-xml-20080205.

[120] World Wide Web Consortium (W3C). "XML Path Language (XPath) Version 1.0". November 1999, [Online]. *Accessed july 2012*, http://www.w3.org/TR/xpath.

[121] World Wide Web Consortium (W3C). "XQuery 1.0: An XML Query Language". December 2010, [Online]. *Accessed July 2012*, http://www.w3.org/TR/xquery/.

[122] XimpleWare. Virtual Token Descriptor (VTD). [Online]. http://vtd-xml.sourceforge.net, *Accessed July 2012*.

[123] X. Yang, Y. Zhu, and Y. Hu. "Scalable Content-Based Publish/Subscribe Services over Structured Peer-to-Peer Networks". *15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP'07), Naples, Italy*, February 2007.

[124] F. Yergeau. "UTF-8, a Transformation Format of ISO 10646". *RFC 3629, IETF*, November 2003.

[125] H. Zhang, J. Kurose, and D. Towsley. "Can an Overlay Compensate for a Careless Underlay?". *IEEE INFOCOM 2006, Barcelona, Spain*, April 2006.

[126] R. Zhang and Y. C. Hu. "HYPER: A Hybrid Approach to Efficient Content-based Publish/Subscribe". *25th IEEE International Conference on Distributed Computing Systems (ICDCS 2005), Columbus, Ohio, USA*, June 2005.

[127] S. Zhang, J. Wang, R. Shen, and J. Xu. "Towards Building Efficient Content-Based Publish/Subscribe Systems over Structured P2P Overlays". *39th International Conference on Parallel Processing (ICPP'10), San Diego, California, USA*, September 2010.

[128] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph. "Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing". *Technical Report No. UCB/CSD-01-1141, University of California Berkeley, USA*, April 2001.

[129] Y. Zhou, K.-L. Tan, and F. Yu. "Leveraging Distributed Publish/Subscribe Systems for Scalable Stream Query Processing". *First International Workshop on Business Intelligence for the Real-Time Enterprise (BIRTE 06), Seoul, Korea*, September 2006.

[130] Y. Zhu and Y. Hu. "Ferry: An Architecture for Content-Based Publish/Subscribe Services on P2P Networks". *International Conference on Parallel Processing (ICPP'05), Oslo, Norway*, June 2005.

[131] Y. Zhu and Y. Hu. "Ferry: A P2P-Based Architecture for Content-Based Publish/Subscribe Services". *IEEE Transactions on Parallel and Distributed Systems*, 18(5):672–685, May 2007.

[132] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. "Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination". *11th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2001), New York, USA*, June 2001.