IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 15, 2018, accepted May 26, 2018, date of publication June 1, 2018, date of current version June 29, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2842706

Design and Implementation of a Hardware Versatile Publish-Subscribe Architecture for the Internet of Things

FADI T. EL-HASSAN[1], (Member, IEEE), AND DAN IONESCU[2], (Senior Member, IEEE)
[1]College of Engineering, Al Ain University of Science and Technology, Al Ain 64141, UAE
[2]School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, ON K1N 6N5, Canada

Corresponding author: Fadi T. El-Hassan (fadi.elhassan@aau.ac.ae)

ABSTRACT A variety of contemporary technologies are being framed within the Internet of Things (IoT) architecture, including publish/subscribe (pub/sub) systems. In IoT, things, such as objects, machines, vehicles, and wireless sensors, have to communicate with other things or humans and exchange information based on Internet connectivity. With the integration of pub/sub mechanism with IoT, these things can ''publish'' their presence to a specific node, which can be named a broker or router, while things that ''subscribe'' to that node are able to receive information based on publishers' content. In order to perceive a sound and efficient pub/sub structure in IoT, high performance processing and interoperability are valid requirements. This paper presents a versatile architecture of a broker, named X2CBBR, that can operate in IoT with different pub/sub systems. X2CBBR: 1) adopts XML-based publication data and XPath-based subscription data to yield interoperability; 2) processes both XML data and XPath queries in hardware (instead of software) to boost processing performance; 3) employs a hardware-based matching mechanism that exploits subscription commonalities; and 4) makes use of four different operation modes as a method for accepting or limiting acceptance of either publications or subscriptions. While maintaining the total decoupling feature between publishers and subscribers, the broker switches from an operation mode to another to keep traffic under control. Moreover, its content-based routing mechanism avoids redundant subscription and notification data. Finally, the broker can effectively operate in either centralized or distributed systems. The results obtained through a prototype hardware implementation targeting an FPGA demonstrate the high-efficiency of the broker/router in multiple scenarios.

INDEX TERMS Internet of Things, publish/subscribe, FPGA, hardware processing, interoperability, content-based routing, versatility, XML, XPath, broker.

I. INTRODUCTION

Integration of publish/subscribe (pub/sub) systems with Internet of Things (IoT) has become an interesting research topic due to many inherent advantages of such systems. In IoT, things (e.g. entities, objects, devices, machines, and wireless sensors) have to communicate with other things or humans and exchange information based on Internet wire/wireless connectivity. Many of these things need to perform their Internet communications using sensors and wireless technologies, which raises the importance of wireless sensor networks in IoT. In addition, interoperability among various things pose challenges that should be faced and optimized in IoT. With the integration of pub/sub mechanism with IoT, things can ''publish'' their presence to a specific node, which can be named a broker or router, while things that ''subscribe'' to that node are able to receive publishers' information based on content. Despite a variety of possible communication standards/protocols, just a few standards may provide interoperability. In this paper, the broker, publishers, and subscribers use and process XML and/or a subset of XPath in order to yield interoperability.

The pub/sub mechanism has been considered as a promising approach in disseminating various data in future Internet. For example, the European projects PURSUIT (Publish-Subscribe Internet Technologies) and its predecessor PSIRP (Publish-Subscribe Internet Routing Paradigm) aim to solve many of the biggest challenges of the current Internet by building a new form of internetworking [1]–[3].

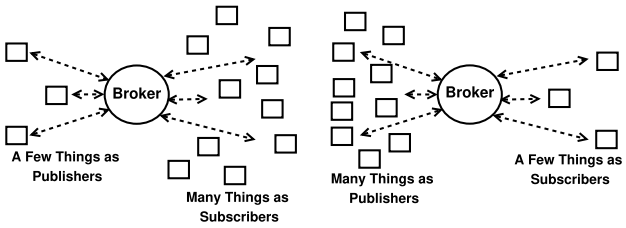


FIGURE 1. Broker handling more subscription things than publication things or vice-versa in IoT.

In IoT, content-based pub/sub networks can be overwhelmed with both publication and subscription data. Subscription data may highly dominate the network traffic, while publication data makes a small percentage of the total traffic – the other way around is certainly possible as illustrated in Fig. 1. In the case that many subscriptions contain many commonalities, initiating sequential matching processes for such subscriptions incurs significant processing cost. Therefore, there is a substantial need to effectively exploit commonalities among such subscriptions. In a pub/sub network, redundant routed traffic incurs additional unnecessary bandwidth and processing cost. Thus, a broker/router should have a mechanism to avoid routing redundant information. The broker may need to only route subscriptions or notifications for a certain pub/sub network. However, it may need to also route publications in response to specific network situations. There can be other circumstances where routed data is of different formats. In such cases, the data does not reach the destination because of the lack of interoperability among network entities. Moreover, centralized pub/sub systems may suit the network of some localized entities and small enterprises, while distributed pub/sub systems are more appropriate for large networks (Fig. 2).

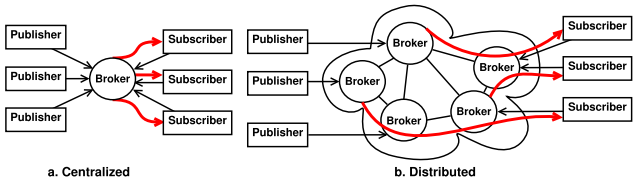


FIGURE 2. Centralized versus distributed pub/sub system.

The versatile content-based broker/router described in this paper can adapt with such diverse circumstances, thereby providing an excellent solution for IoT. It holds the name of X2CBBR (XML/XPath Content-Based Broker/Router). To yield interoperability, X2CBBR adopts XML-based publication data and XPath-based subscription data. To boost performance, it processes both XML data and XPath queries in hardware, exploiting subscription commonalities. An operation mode algorithm allows the broker to switch from a mode to another to keep subscription and publication data under control. X2CBBR makes use of a content-based routing mechanism that can route a single subscription on behalf of multiple subscriptions, or a single publication instead of multiple notifications to avoid redundant data. In addition to the aforementioned features, X2CBBR maintains the total decoupling feature between publishers and subscribers, which is a fundamental and desirable pub/sub requirement for security and privacy purposes. While some brokers may work in either a distributed or a centralized environment, X2CBBR can efficiently operate in both systems due to the inherent decoupling feature between matching and routing.

The original version of the hardware broker was introduced in our previous work [4] that described how commonalities among XPath subscriptions can be systematically stored and exploited in order to achieve high-performance matching against XML publications. With the additional aforementioned features of versatility, X2CBBR performs content-based routing and employs efficient mechanisms to cope with different pub/sub networks within IoT. The main purpose of this paper is to describe with details these versatility features, and to highlight their applicability in the context of IoT.

The rest of the paper is organized as follows. Section II presents related work. Section III describes the X2CBBR architecture and its main tasks. Section IV states the notations used in later sections of the paper. Section V discusses in detail the operating mode mechanism along with the top-level algorithm. The operating mode mechanism allows X2CBBR to switch from a mode to another to cope with different situations of traffic, while the algorithm routes the traffic accordingly. In Section VI, simulation results are depicted and discussed. Finally, Section VII concludes this paper.

II. RELATED WORK

In the literature, to the best of our knowledge, pub/sub systems have been characterized with some but not all the versatile features that distinguish our proposed system. The proposed architecture comprises interoperability, hardware design targeting an FPGA, content-based routing, total decoupling between publications and subscriptions, separate matching from routing, operation mode mechanism to cope with various traffic situations, and integration with IoT. Therefore, the structure of this section consists of subsections to describe the related work, where each subsection discusses the published papers with regard of one category of pub/sub features.

A. INTEROPERABILITY

For IoT, pub/sub systems with the interoperability feature can be based on standards related to the eXtensible Markup Language (XML) [5], such as Extensible Messaging and Presence Protocol (XMPP) [6], XPath [7], XQuery [8], Message Queue Telemetry Transport (MQTT) [9], Advanced Message Queuing Protocol (AMQP) [10], SPARQL [11], and Constrained Application Protocol (CoAP) [12].

A pub/sub architecture for IoT with the interoperability feature is proposed by [13] based on XMPP. In this work, subscribers' interests are first processed by the publisher, and then the publisher provides a customized publishing data to

a server node in order to support a lightweight architecture. It also supports periodic data transmission to save energy for battery-powered devices. However, the proposed scheme does not provide a total decoupling feature between publishers and subscribers. A pub/sub system with total decoupling in space and time is the usual expected assumption to preserve confidentiality. Due to increased complexity in the total decoupling, some architectures adopt loose or no decoupling, and even indicate that with loose decoupling better key management solutions for secure group communication can be achieved [14].

Examples of pub/sub systems and architectures for IoT have been designed or developed with the interoperability feature based on the SPARQL query language [15] and Data Distribution Service (DDS) [16]. In [15], publishers and subscribers use standard SPARQL updates and queries, where a semantic event detection algorithm updates notifications based on changes detected in previous notifications. The purpose of the algorithm is to limit the network overhead and save notification processing at the subscriber side.

In [16], Hakiri *et al.* combine Software Defined Networking (SDN) with Data Distribution Service (DDS) middleware. DDS is a middleware standard addressing pub/sub communications for real-time and embedded systems [17]. SDN decouples the control plane from the data plane by moving the control plane from routers/switches to a central software-based controller. The controller dictates the policies and rules that the data plane must use to forward packets from a flow table to another. The purpose of combining SDN with DDS is to improve service delivery of IoT systems and provide network agility and flexibility.

In [18], Triawan *et al.* propose cloud-based pub/sub architecture that can be integrated with IoT. In this approach, interoperability is handled through MQTT and JavaScript Object Notation (JSON) [19].

B. XML FILTERING WITH XPATH

Away from the context of IoT, filtering/matching XML with XPath expressions is the subject of pub/sub data dissemination systems [20]–[25], where exploiting the power of XML filtering takes place in software. In terms of the used techniques, in [20], XML matching against XPath is performed where commonalities among XPath expressions are not exploited. In [21]and [26], content-driven routing and XML filtering with additional complexity are addressed. Another software-based work, named XNET [24], uses a trie-based index data structure, called XTrie, where XPath expressions are decomposed into common substrings that contain parent-child operators [22]. An improved approach of this method filters fragmented XML data [23], [25]. In addition, XNET employs an algorithm to efficiently manage a large number of subscriptions, where its application-layer routing protocol performs subscription aggregation to keep the size of routing tables manageable. This work has interesting desirable features that include elimination of redundant subscriptions.

C. HARDWARE IMPLEMENTATIONS

Acceleration of pub/sub messaging using FPGAs has been reported in [27]. More specifically, Quigley *et al.* [28] implement the topic-based pub/sub messaging of the Robot Operating System (ROS) development platform in hardware to reduce the latency of the communication system. As the acceleration is specific to ROS-compliant, the authors did not suggest any interoperable generalization method for topic-based pub/sub systems.

In a software/hardware application [29], Mitra *et al.* indicate improvement over software XML filtering systems. However, the need for stages of software/hardware conversion for new XPath queries, and for queries that need frequent updates, would significantly drop the overall performance.

In another software/hardware application [30] that utilizes an FPGA, Moussalli *et al.* implement an XPath processor that can handle many XPath queries in conjunction of a software-based XML parser. The authors report a 200 MB/s throughput when the XML parser is replicated in software for each FPGA cluster. Besides the high fan-out in the software-hardware interface, the XPath architecture relies on intensive memory resources.

D. SPECIFIC SOFTWARE APPROACHES

Even though many of the software approaches in the literature are not close to our proposed architecture, we select a few approaches that have a special technique or methodology in dealing with pub/sub systems.

In [31], a general cloud-based pub/sub service is proposed to provide diverse data dissemination, scalability, and elasticity. With this cloud-based approach, the capacities of event matching and content distribution can adapt and increase with the growth of servers. In addition, irrelevant users can be filtered out. In terms of implementation and communication, the approach includes Java code and uses a communication framework based on Remote Procedure Calls (RPCs). In general, the cloud-based approach is interesting to consider in the future if it is coupled with IoT.

In [32], Diallo *et al.* propose a service model to meet different requirements for content-based pub/sub systems. By leveraging caching policies, the model can improve the communication efficiency as a result of selected scenarios. The caching policy could be more important if coupled with interoperability.

In [33], an adaptive software mechanism is used to perform alternate publication routing paths and improve the resilience to dynamic loads. In another improvement attempt, a hybrid approach makes use of a centralized brokering system coupled with a decentralized peer-to-peer (P2P) protocol [34]. In another perspective of a hybrid approach, subscriptions are organized into virtual groups based on common subscriptions, where the matching process occurs once at the entry point of each group [35]. Inside the group, messages are simply forwarded, without further matching, similar to the delivery of content to a multicast group in a topic-based

pub/sub system. Facing the complexity of simultaneous XML matching and routing, the authors of [36] describe an interesting software mechanism that 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 separation between matching and routing is one of our broker's features.

E. COMPARISON

In spite of the aforementioned important procedures, some applications may outperform other ones depending on the techniques used in processing/matching/filtering, traffic and network situations, and hardware support.

The versatile X2CBBR, described in this paper, inherently decouples matching from routing. The matching process exploits commonalities among available subscriptions, while the routing process focuses on the elimination of redundancy in both subscription and notification data. Moreover, X2CBBR can effectively operate in either centralized or distributed pub/sub systems, and can efficiently accommodate different content-based scenarios. In terms of hardware support, its native hardware architecture does not need stages of software conversion into hardware. In addition to interoperability, X2CBBR provides scalability (indicated in section VI-B4). This versatility of X2CBBR is so attractive for pub/sub systems, and these features are so desirable in IoT.

In order to have insights on the performance, we conducted experiments taking into the account the type of interface used in the communication between X2CBBR and external entities. The corresponding results stated in section VI show that the throughput may reach 1447 Mbps in the best-case scenario. The experiments, simulations, and results were described with details in [37].

III. XML/XPATH BROKER/ROUTER ARCHITECTURE

While the broker in [4] described hardware-based exploitation of commonalities among XPath subscriptions in order to achieve high-performance matching against XML publications, X2CBBR employs additional mechanisms including the ones related to content-based routing, and possesses therefore more versatile features. In order to accommodate different pub/sub data situations, another add-on mechanism dictates operation modes to allow the broker to switch and revert from a mode to another, as described in section V.

While the paper [4] describes the basic broker architecture and the experimental results of just the matching process (subscriptions against publications), this paper includes detailed descriptions of the broker modules and corresponding algorithms, and shows more experimental results in multiple scenarios when either a hardware interface or a software/hardware interface is employed.

The X2CBBR architecture, depicted in Fig. 3, consists of three main units that can communicate with external entities through six dedicated interfaces that may be active concurrently. Part A of the architecture is the Subscription

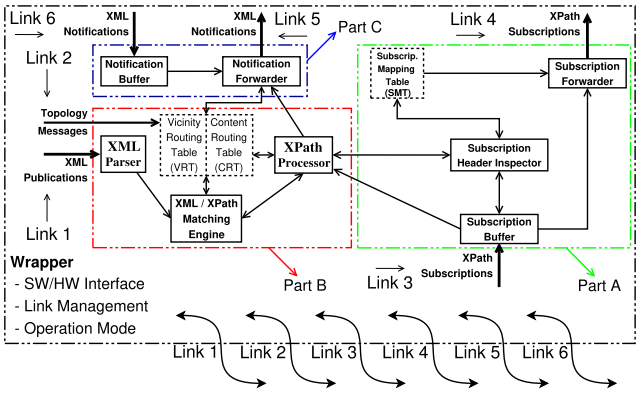


FIGURE 3. Block diagram of the X2CBBR architecture.

Processing Unit (SPU); Part B is the XML/XPath Processing and Matching Unit (PMU), and Part C is the 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 one. The NPU may receive notifications through one interface and route out notifications through its other one.

The new extensions of X2CBBR, compared to the previous work [4], are mainly related to the Vicinity Routing Table (VRT) of the PMU, the Subscription Mapping Table (SMT) and the Subscription Buffer of the SPU, and the operating mode and routing mechanisms. This paper primarily focuses on these new extensions, which highly contribute to the broker's versatility. The reader is recommended to consult the previous work for detailed description of the original architecture.

FIGURE 4. Structure of a subscription.

A. BASIC FUNCTIONAL OVERVIEW

Each subscription consists of a 20-byte header, and up to 16 4-byte filters (64 bytes). The header includes an original Subscription ID (o-SID) as illustrated in Fig. 4.

This restriction of the filter data volume 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 Content Routing Table (CRT). Consequently, X2CBBR would not have the opportunity of simultaneously processing many subscriptions.

The o-SID can be updated during the routing procedure from 1 to 8 bytes (as we shall see in section III-D). X2CBBR maps the o-SID of each subscription to an 8-bit temporary Subscription ID (t-SID). These temporary IDs are used in subsequent stages of X2CBBR to efficiently store and process subscriptions. The Subscription Mapping Table (SMT) stores (t-SID, o-SID) values until appropriate notifications

and subscriptions are forwarded. At that time, the SMT deletes or overwrites its entries and gets ready to receive new subscription IDs.

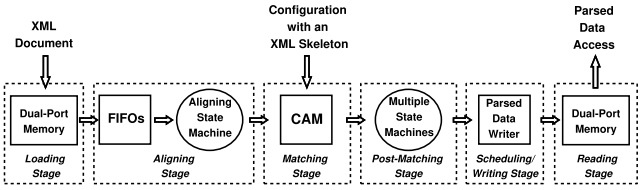


FIGURE 5. Block diagram of the SCBXP technique.

Each publication is in XML format and is parsed using our hardware-based SCBXP parsing technique [38] (Fig. 5). The CRT registers the filters belonging to subscriptions as well as the corresponding t-SIDs. Matching these filters against available XML publications is the main task of the XML/XPath matching engine. The entries of the CRT are functionally of the format (F_i,SID(F_i)) where SID(F_i) is the set of all t-SIDs whose subscriptions match filters F_i. Physically, the CRT consists of a Content Addressable Memory (CAM) and a Random Access Memory (RAM).

Once a match is found for all filters F_i of a subscription s, a corresponding notification N is either locally delivered (if s has come from a local subscriber that has a direct link to the broker) or forwarded to a neighboring broker via the "Notification Forwarder" of the NPU (Part C in Fig. 3) in order to deliver the notification to a remote subscriber. In the case that the broker receives a notification from a neighboring broker, it simply delivers it or forwards it without any processing or matching tasks, because the notification is actually ready for delivery.

The VRT describes the topology in the vicinity of a specific broker. The VRT entries are functionally of the format *(X, BID)* where *BID* is the Broker ID of a neighboring broker, and X can be a filter, a topic, a data type, or any specific content. Accordingly, the mismatched subscriptions in a broker are forwarded to the neighboring brokers whose available publications may have matching content. The VRT, which is physically a RAM, regularly updates its entries according to topology messages assumed to be issued by an external entity that knows the network topology. The structure of this particular entity is out of the scope of this paper and is not further discussed.

On top of the mechanisms of (1) subscription registration, (2) pub/sub matching, (3) notifying and routing, an operating mode mechanism (discussed in section V) maintains the number of subscriptions and publications that are available in the X2CBBR resources. Accordingly, this mechanism pushes the X2CBBR to switch from a mode to another, allowing for accepting, routing, or flushing data. Each of these mechanisms will be discussed in detail in subsequent sections of this paper.

In addition, the X2CBBR has to manage the interface data that is coming in or going out through any of its six links.

B. DEFINITIONS

The following items define some concepts and terms that are useful in the discussion of X2CBBR tasks.

1. 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.
2. A "remote subscription" available in a broker B_i is an XPath subscription sent to B_i by a broker B_j, providing that it has been originally issued by a subscriber that does not directly communicate with B_i.
3. Based on items 1, and 2 above, all subscriptions available in an intermediate broker B_i (within a network of brokers) are considered "remote" to B_i, while subscriptions available in an edge broker B_j of the network may be either "local" or "remote" to B_j.

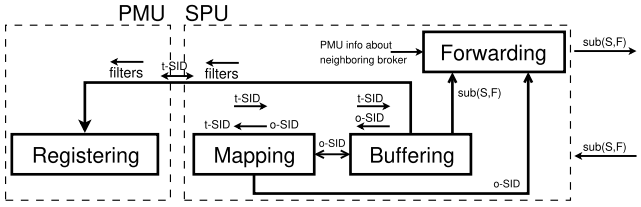


FIGURE 6. Processing stages of a subscription.

The action *sub(S,F)* (or *sub(o-SID, F)*) of subscribing to the system reads that a new subscription identified as S (or by its o-SID) and consisted of the filter set F, has reached a broker. X2CBBR 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. Fig. 6 shows how these stages would generally intercommunicate. The SPU may also run a "forwarding process" on old subscriptions (i.e. subscriptions that have been already registered in the PMU).

The next sections address the subscription registration and routing processes from an algorithmic point of view, while the architectural hardware details of each unit of X2CBBR are discussed starting from Section III-E.

C. SUBSCRIPTION REGISTRATION ALGORITHM

X2CBBR exploits commonalities among subscriptions by registering filters of all subscriptions using a method that allows simultaneous matching of the common filters that may exist in many subscriptions. The subscription registration process aims to systematically store the filters and the t-SID of each subscription in the CRT. Within the CRT, a CAM stores the filters of XPath subscriptions, while a RAM (Subscription IDs Visiting Location Memory) stores corresponding t-SIDs. Each CAM address that contains a filter corresponds to an equivalent RAM address that contains t-SIDs of subscriptions that host this filter. Each RAM row consists of 256 bits, where a maximum of 31 8-bit t-SIDs may coexist in addition to an 8-bit "next address" value. This latter value indicates the next

row address in which additional t-SIDs may be stored. Details of the physical storage structure of both memory modules can be found in our initial work [4]. 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: $s_1 = \{t\text{-}SID1, (f_1, f_2)\}$ and $s_2 = \{t\text{-}SID2, (f_1, f_3)\}$. 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*$\}$.

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 7. Subscription registering algorithm.

Fig. 7 presents the Subscription Registration Algorithm steps. In this algorithm, the first two lines summarize the subscription buffering stage, which takes into account the restriction of a maximum of 16 32-bit filters (Section III-A).

The third line 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 and 8 relate to the aforementioned restriction of the maximum number of filter data, 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.

Lines 10 to 32 describe the registration stage in the CAM and RAM of the CRT. In the case that a full RAM row exists during the registration of a filter, the algorithm reports a High Commonality Degree (HCD) (line 21) to the SPU. This signal indicates that a filter is common to at least 31 subscriptions. Therefore, if many mismatched subscriptions have been identified and the HCD is reported, only a single subscription is forwarded instead of all mismatched subscriptions in order to reduce the routed subscription traffic.

The Reading Register RR is used to temporarily store the RAM row content prior to appending the row with a new matching t-SID (lines 29 - 30). In the case that the row is full, the RR temporarily stores the ''next address'' value and later uses it in the RAM to store additional matching t-SIDs (lines 22 to 27).

Lines 13 and 14 of Fig. 7 indicate that the absence of a filter F_i in the CAM triggers a CAM writing access 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 that is needed.

D. SUBSCRIPTION ROUTING ALGORITHM

Prior to routing a subscription, the Subscription Forwarder updates the subscription's o-SID. The o-SID of a local subscription has the format of $\{SID, 0\}$, where *SID* is a one-byte subscription ID. A remote subscription s that reaches a broker, say B2, must have crossed a neighboring broker, say B1.

If the subscription s was local to B1, the o-SID of the subscription appears in B2 as o-SID $= \{SID, B1, 0\}$, where $B1$ is the BID of the broker B1. Subsequently, if this subscription s continues its path to a third broker B3, the o-SID of s appears in B3 as o-SID $= \{SID, B1, B2, 0\}$, where $B2$ is the BID of the broker B2. Since the o-SID consists of a maximum of 8 bytes, there would be a maximum of 7 brokers in the path of a subscription, providing that each BID consists of one byte. In this latter case, the o-SID appears in a broker B7 as o-SID $= \{SID, B1, B2, B3, B4, B5, B6, 0\}$. Then, if s matches an available publication in the broker B7, this broker sends a notification through its neighboring broker B6 with o-SID $= \{SID, B1, B2, B3, B4, B5, B6, B7\}$. In the case that no match occurs, $B7$ removes the subscription after a certain period of time. Fig. 8 illustrates the o-SID update structure.

Following the aforementioned discussion, one can state that the o-SID of a remote subscription has the format of $o\text{-}SID = \{SID, \ldots BID \ldots, B_j\}$ where ''$\ldots BID \ldots$'' represents the BIDs of all brokers crossed by the subscription before reaching the broker whose BID is B_j.

X2CBBR forwards each mismatched subscription according to the Subscription Routing Algorithm depicted in Fig. 9. In this algorithm, the SPU receives from the PMU the t-SID of the mismatched subscription (line 1), retrieves the corresponding o-SID from the SMT (lines 2 and 3), adds the broker's own BID to the o-SID (line 4 or line 12), extracts the subscription from the ''Subscription Buffer'' and reconstructs it as *sub(o-SID(s),F)* (line 6 or line 13), and

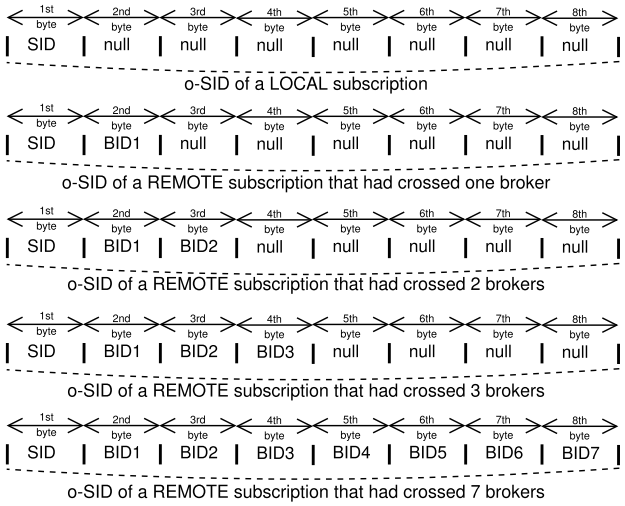


FIGURE 8. The o-SID structure in local and remote subscriptions.

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 9. Subscription routing algorithm.

forwards the subscription to a neighboring broker (identified by the VRT) via the "Subscription Forwarder" (line 8 or line 15).

E. PART A: HARDWARE SPU ARCHITECTURE

The Subscription Buffer of the SPU comprises two dual-port memory modules, named DPM1 and DPM2 (Fig. 10). The SPU loads raw (unprocessed) XPath subscriptions into either DPM1 or DPM2. The subscriptions buffered into, for example, DPM1 will remain in it until a match decision is made and/or until the Subscription Forwarder routes these subscriptions to neighboring brokers. Once DPM1 is full, the SPU buffers new incoming subscriptions into DPM2 concurrently with the forwarder's task of routing DPM1 subscriptions. While the subscriptions buffered in DPM2 are being processed and forwarded, the broker can load DPM1 with newly-arrived subscriptions, and so on. The SPU utilizes multiplexers (MUX1, MUX2, MUX3, and MUX4) to select the memory module to be loaded with raw subscriptions and the one to be accessed for forwarding purposes, as shown in Fig. 10. Note that either DPM1 or DPM2 is a 32 bits x 2048 words memory module that may contain up to

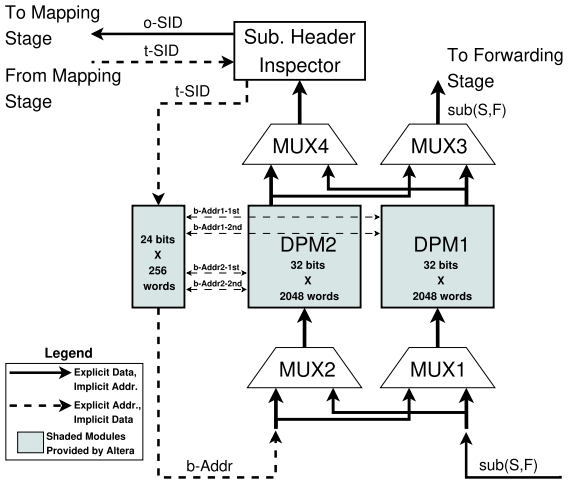


FIGURE 10. Subscription buffer.

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 o-SIDs. The simultaneous writing and reading accesses allow the inspection process to run with no discontinuation of the buffering process.

The inspector passes the o-SID to the mapping stage where the SMT sends back a mapped 8-bit t-SID value. The inspector uses the t-SID as an address (index) to access one row of a small dual-port SRAM (24 bits X 256 words). At this specific row, reside the first and last buffer addresses (each is 11-bit wide) of the corresponding raw subscription buffered 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 Fig. 10). The Subscription Forwarder later utilizes these two addresses to retrieve buffered subscriptions and route them to neighboring brokers as mentioned in Section III-D. The benefit of designing the "24 bits X 256 words" memory as a dual-port SRAM is to allow both the forwarder and the inspector to access it concurrently. The forwarder reads from it while the inspector writes into it.

Forwarding a raw subscription implies that its format depicted in Fig. 4 is preserved, with an updated o-SID (Fig. 8). Fig. 11 illustrates the FSM of the subscription forwarder. In this figure, the FSM transition to its "EvalSu" state from the initial state only takes place on the release of the "reset" signal and on the completion of the set S_u that identifies all mismatched subscriptions registered in the CRT. Once the t-SID of a mismatched subscription is retrieved, the FSM steps to its "GrabAdr" to pick up the corresponding subscription buffer addresses, then to its "GrabSub" state to retrieve the buffered subscription. In the "AddBID" state, the forwarder updates the o-SID of the subscription. Finally, it routes the subscription to the identified neighboring broker in the "FRWD" state, before the FSM steps again to the "EvalSu" state to retrieve the t-SID of the next mismatched subscription, and so on.

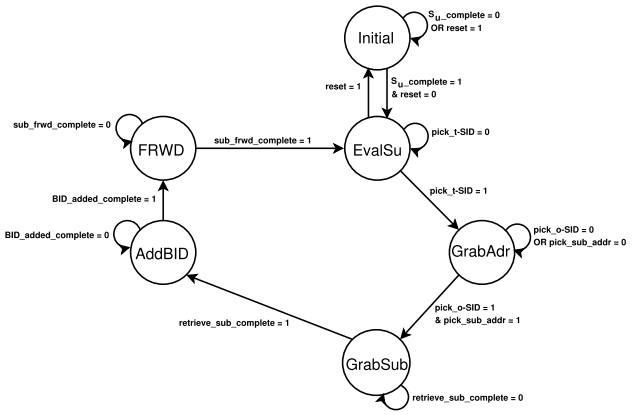


FIGURE 11. The FSM of the subscription forwarder.

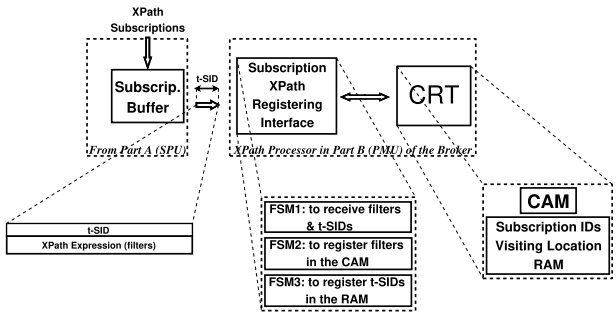


FIGURE 12. Hardware XPath processor architecture.

F. PART B: HARDWARE PMU ARCHITECTURE

The XPath processor of the PMU communicates with the SPU to perform its main role of registering subscriptions using the finite state machines FSM1, FSM2, and FSM3 (Fig. 12). These machines reside in the ''Subscription XPath Registering Interface'' to control the reception of subscription filters and corresponding t-SIDs, register the filters in the CAM of the CRT, and register the relevant t-SIDs in the RAM of the CRT. The internal structure of either the CAM or the RAM of the CRT can be found in details in the previous work.

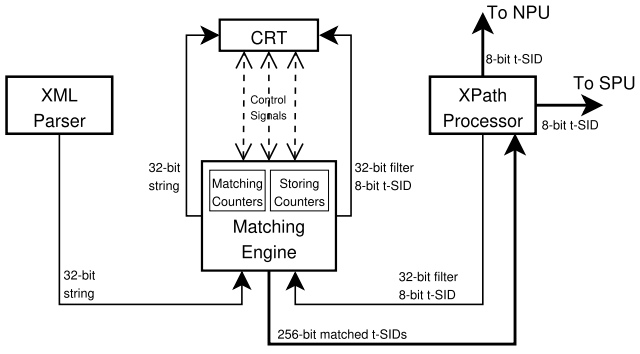


FIGURE 13. XML/XPath matching engine.

Concurrently with XPath processing, the SCBXP parses available XML publications. The matching engine (Fig. 13) communicates with both the XPath processor and the XML parser 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 NPU (Part C of Fig. 3) derives a notification from the matched publication and forwards it to the corresponding subscriber(s) through the Notification Forwarder. 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.

The engine searches the CAM of the CRT in two different stages: 1- in the subscription registration stage to verify whether a 32-bit filter is already in the CAM (lines 11 to 17 in Fig. 7) and 2- in the post-registration stage to verify whether a 32-bit string (parsed from an XML publication) matches any 32-bit XPath filter existing in the CAM. During the subscription registration, 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 particular t-SID. During the post-registration, a matching counter for a specific t-SID increments with each successfully-matched filter.

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 determines that all registered filters for this particular t-SID have matched an available XML publication. Thus, such a matching decision must trigger a corresponding notification through the NPU.

G. PART C: HARDWARE NPU ALGORITHMS

The Notification Forwarder of the NPU (Part C in Fig. 3) receives the t-SIDs of subscriptions whose all filters are successfully matched. Subsequently, it utilizes these t-SIDs as addresses to retrieve the o-SIDs stored in the SMT. A retrieved o-SID reveals whether a matched subscription is local or remote. Accordingly, the NPU delivers or routes the relevant notification using the notification routing algorithm (Fig. 14).

In the third line of Fig. 14, the SMT returns the o-SID that corresponds to the t-SID of the matched subscription. In line 4, the NPU recognizes a local subscription from the o-SID. Accordingly, the NPU includes the o-SID of the matched subscription in an XML notification N generated as *notify(SID,N)*. In line 5, the NPU actually sends this notification.

In the case of a remote subscription, the NPU picks up the BID of a neighboring broker from the o-SID (line 8). Since the remote subscription may have crossed multiple brokers in its path, multiple BIDs may have been concatenated in the o-SID (Section III-D). X2CBBR selects the appropriate BID of the last crossed neighboring broker, by picking up the o-SID's 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 with its own BID. In line 10, the broker adds the updated

Algorithm: Local/Remote Notification - Generation & 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 subscriptions 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 14. Combined local & remote notification/publication routing algorithm.

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.

1) PUBLICATION ROUTING

Line 13 of the algorithm points to the case of a remote subscription with SID = 0. The null-value of the SID tells the neighboring broker that the routed subscription is a "sample" of multiple subscriptions containing common filters. X2CBBR only routes a single subscription to reduce redundant subscriptions that potentially consist of common filters. This traffic reduction positively impacts the overall performance of the system.

As a result of successfully matching this subscription, the NPU of the neighboring broker responds with the publication itself (lines 16 and 17) instead of the 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, re-matching this publication against subscriptions of common filters takes place in the local broker of these subscriptions. As a result, relevant notifications can locally reach their destinations.

2) NOTIFICATION RE-ROUTING

X2CBBR may receive notifications from an up-stream broker. X2CBBR buffers the notification in the notification buffer of the NPU, and inspects the o-SID embedded in the received message "*forward(o-SID, N)*" to appropriately route the notification to the intended destination. The structure of the o-SID in a routed notification is the same as in a routed subscription (Section III-D). Fig. 15 shows the steps of the notification re-routing algorithm.

Upon receiving a notification from an up-stream broker, the NPU removes the BID of that broker (B_u in line 2) from

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 15. Notification re-routing algorithm.

the o-SID. As a result, the o-SID reveals the BID of the current broker. At this point, the broker also removes its own BID from the o-SID. If the resulting o-SID has the format of *{SID, 0}*, 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 $\{SID, \ldots BID \ldots, B_d\}$ where B_d is the BID of a down-stream broker (Section III-D). Accordingly, the NPU updates the 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).

IV. NOTATIONS

On top of the multiple algorithms discussed earlier, X2CBBR works according to a sequence of the so-called "operation modes" or "operating modes." Either term should 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.

Later sections and algorithms of this paper rely on 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

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 16. Determination algorithm (DA) of the maximum number of registered subscriptions.

usually set to ≤ 255 as discussed below along with the algorithm of Fig. 16.

- $S_m \subseteq S$**:** the set of all subscriptions that match at least a Publication P_i, where $i = 1, 2, \ldots, P$.
- $S_{m_i} \subseteq S$**:** the set of all subscriptions that match a single particular Publication P_i, where $i = 1, 2, \ldots,$ *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 $i = 1, 2, \ldots, P$.
- $S_a \not\subset S$**:** 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 Fig. 16. Providing that the SMT can handle up to 255 (rather than 256) t-SIDs, 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 Fig. 16.

V. OPERATING MODE MECHANISM

A. INTRODUCTION OF OPERATION MODES

In IoT, pub/sub applications can have unbalanced numbers of subscriptions and publications. A typical example of subscriptions that by far outnumber available publications is the case of a stock price publication where many users would like to readily know the price variations. There are other applications where publications may be more frequent than subscriptions. For example, multiple sensors periodically publish their presence, while sporadic events would trigger sensor alarms. 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, the introduced concept of operation modes allows the broker to accommodate different situations, by initially working in a mode and reverting to another mode when

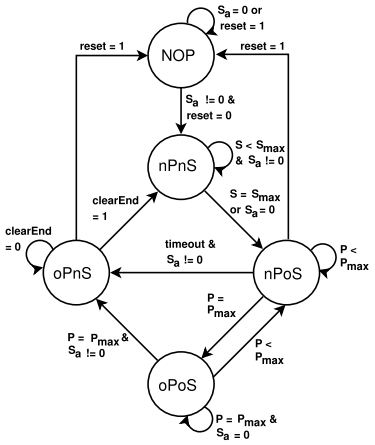


FIGURE 17. Operation mode FSM.

certain circumstances occur. Fig. 17 illustrates the operation mode mechanism as an FSM, while the top-level algorithm depicted in Fig. 18 highlights the tasks in each mode. This figure embeds all hardware algorithms involved in subscription registration, matching pub/sub data, and routing notifications and subscriptions. Note that the notations stated in Section IV are very useful in reading either Fig. 17 or Fig. 18.

B. SEQUENCE OF OPERATING MODES

In each of its operating modes, X2CBBR may perform multiple tasks concurrently in addition to the sequential tasks. It 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.

Initially, X2CBBR is in its "no operation" mode (state *NOP* in Fig. 17). 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 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.

In the (nP, nS) **mode** (state *nPnS* in Fig. 17), multiple tasks/algorithms may run in parallel (lines 3 to 9 in Fig. 18):

- The SPU continues to receive and buffer new subscriptions upon their arrival, and updates the set S_a.
- The broker runs the Subscription Registering Algorithm (Fig. 7). In the case that commonalities exist in many subscriptions, the algorithm reports the High Commonality Degree (HCD) signal.
- The broker runs the "S_{max} DA" (Fig. 16).
- The XML parser receives, buffers, and parses one new XML publication (i.e. $P = 1$). Concurrently with parsing this publication, X2CBBR may buffer - without parsing - a second XML publication (if it is available). Moreover,

Top-Level Algorithm: Embedded Mechanisms

```
1: if mode = NOP then
2:     Buffer only one new subscription; Form S_a & Report S_a ≠ ∅
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 ⊆ S): Run {Subscription
       Registration Algorithm}
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 → 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 → 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 → 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 ∈ 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 for a single subscription
           with null SID
39:    else
40:        for all S_u do
41:            Run Subscription Routing Algorithm
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 18. Pseudo-code for top-level algorithm.

the broker monitors and updates P and P_{max}. However, no matching of parsed publication data against subscription filters would take place.

X2CBBR may remain in its same (nP, nS) mode as far as $S < S_{max}$ and $S_a \neq \emptyset$ (Fig. 17)). Otherwise (i.e. when no more new subscriptions exist in the SPU or S reaches S_{max}), X2CBBR steps to the (nP, oS) **mode** (state *nPoS* in Fig. 17), 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 Fig. 10). 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. Accordingly, lines 10 to 35 in Fig. 18 state three "for loops" that run in parallel with synchronizing signals. The (nP, oS) mode stays into effect as far as $P < P_{max}$ and 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.

X2CBBR undertakes another major task during this mode, which is issuing and routing out relevant notifications according to the computed S_m. Even though lines 28 to 32 of Fig. 18 point to two separate notification algorithms for the sake of clarity, there is actually one combined algorithm (Fig. 14) for both local and remote notifications.

The (oP, oS) **mode** ((lines 36 to 45 in Fig. 18)) illustrates a sort of "saturation" status, where all the available pub/sub data has been processed and compared. The broker stays in this mode as far as $P = P_{max}$ and $S_a = \emptyset$, even if timeout has occurred. In the event that $S_a \neq \emptyset$, the broker steps up to the (oP, nS) mode. Otherwise, it may switch 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 this mode is to route out mismatched subscriptions (Fig. 9), referring to the computed S_u. 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 Subscription Registering Algorithm, X2CBBR only forwards a single mismatched subscription to a neighboring broker (lines 37 and 38) as discussed in Section III-G1.

In the (oP, nS) **mode** (lines 46 to 55 in Fig. 18), the broker clears the sets S, S_m, and P. In the case that the subscription routing process is still on-going, the broker waits until the end of the routing task prior to clearing S_u (lines 51 and 52 in Fig. 18). 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 upon the application of a "reset" signal.

C. DISCUSSION

From the aforementioned description, the sequence of the operating modes $[(nP, nS),(nP, oS),(oP, nS)]$ is likely to happen when the publication traffic is so intense to the extent that P never reaches P_{max} until old subscriptions expire. A short duration of the (nP, oS) mode indicates a short configured timeout and relatively frequent subscription traffic, while a long duration of the (nP, oS) mode reveals frequent publication traffic and either a long configured timeout or relatively infrequent subscription traffic.

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

VI. RESULTS

A. EXPERIMENTAL SETUP

The experiments examine the functionality and performance of the X2CBBR using either a sole hardware interface or a software/hardware interface, while operating and routing algorithms are into effect. Both the X2CBBR and its hardware interface are implemented on Altera's Cyclone IV E FPGA device [39] located on the Terasic DE2-115 FPGA board [40]. The broker's hardware interface emulates the Avalon bus specification [41] 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 allows for an insightful comparison of the resulting performance.

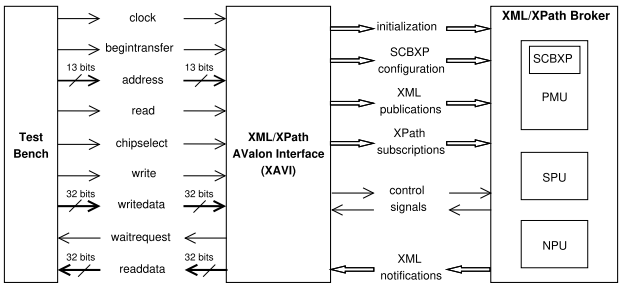


FIGURE 19. Broker's hardware interface.

1) HARDWARE INTERFACE (XAVI)

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 Fig. 19. Accordingly, the X2CBBR returns output data to the test bench. The list of Avalon inputs includes: *address, read, write, writedata, begintransfer,* and *chipselect*, while the returned outputs are: *waitrequest* and *readdata*.

The hardware driving tasks include initializing X2CBBR's memory resources, configuring the SCBXP's CAM with an XML skeleton, loading the SCBXP's memory with XML publications, loading the SPU's 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.

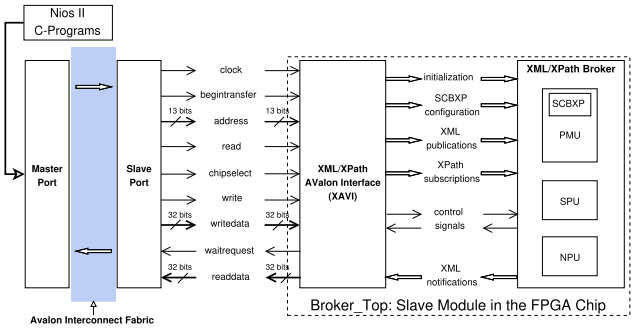


FIGURE 20. Broker's software/hardware interface.

2) SOFTWARE/HARDWARE INTERFACE

The software/hardware interface (Fig. 20) consists of the hardware interface XAVI and the C-programs that we have developed in the software environment of Altera's Nios II embedded processor [42]. The X2CBBR and XAVI together represent the slave module that is implemented on Altera's FPGA Cyclone IV E device. This module, whose name is *Broker_Top* in Fig. 20, is the top level implemented design. The C-programs drive this module via the Avalon bus master/slave ports.

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 [43], [44].

3) SOFTWARE/HARDWARE INTERFACE WITH CONTENT-BASED ROUTING

Even though X2CBBRs can efficiently intercommunicate using sole hardware interfaces, while content-based routing is into effect, the aim of this section is to use software/hardware interfaces with a standard medium of intercommunication. To communicate with producers, consumers, or other brokers (content-based routers), the X2CBBR needs a standard communication medium naturally available on the DE2-115 board. That standard is the Ethernet. The software programs drive the broker using the Nichestack TCP/IP operating over Ethernet provided as a configurable IP (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) [45], [46].

A method of connecting a broker to another broker is illustrated in Fig. 21, where Ethernet packets 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.

4) SUBSCRIPTION SETS

There are three sets of XPath subscriptions under consideration: XPath Set1, XPath Set2, and XPath Set3. Each set comprises almost 1 KB of data. Each XPath subscription is

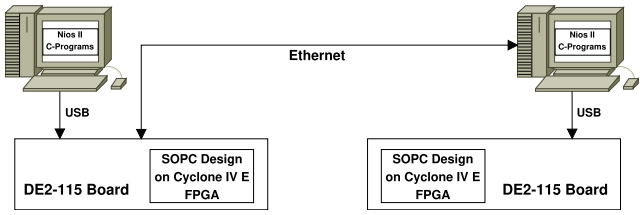


FIGURE 21. Alternative method of broker to broker connectivity.

made of just one filter for simplicity, besides the 20-byte overhead, where the filter consists of 4 ASCII characters (32 bits). Thus, with a size of 192 bits per subscription, each of the three sets contains 42 raw subscriptions. To test the broker with up to almost 255 subscriptions, the interface gradually loads six sets of 42 subscriptions via one port of the buffer's memory.

In XPath Set1, all subscriptions have common filters that match an available publication. As a result, local notification delivery to all relevant subscribers takes place. In XPath Set2, three subscriptions do not match any publication that is available in the local broker. Local notification delivery takes place for the matched subscriptions, while mismatched-subscription routing followed by notification routing must take place for each mismatched subscription prior to delivering the final notification to the relevant subscribers. In XPath Set3, all subscriptions have identical filters but they do not match any available publications in their local broker. Therefore, single-subscription routing followed by publication routing must take place prior to the final matching and notifying processes.

Three different groups of tests are to be performed with either XPath Set1, XPath Set2, or XPath Set3:

- **Case I** utilizes the hardware interface XAVI described in Section VI-A1, only employing XPath Set1. Six sets of XPath Set1 are gradually included in the test, where a reset signal resets the broker after matching each additional set.
- **Case II** utilizes the software/hardware interface described in Section VI-A2. As in Case I, XPath Set1 is the sole set employed, with the same gradual inclusion of six sets and the reset application after matching each additional set.
- **Case III** utilizes the software/hardware interface with content-based routing (Section VI-A3). This case employs both XPath Set2 and XPath Set3.

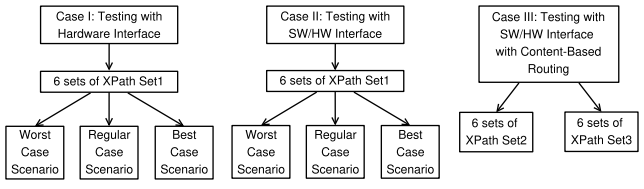


FIGURE 22. Test cases.

Figure 22 summarizes the testing strategy for the aforementioned three cases.

B. CASE I: TESTING THROUGH HARDWARE INTERFACE XAVI

The aim of this test is to measure the performance of the X2CBBR solely driven through the XAVI hardware interface, using the ModelSim simulator [47]. 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.

This clocking strategy allows the broker to achieve its high performance in its core, while the interface and the SCBXP remain error-free and well-synchronized. The SCBXP writes its parsed data output into the dual-port memory of its reading stage (Fig. 5) via one port using the 50-MHz clock, and this data is later accessed by the broker via the other port using the 150-MHz clock. This memory is the only module, within the broker, that must handle two different clock frequencies.

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 and the best-case scenarios.

1) WORST-CASE SCENARIO

First, the X2CBBR initializes its memory resources in about 17.7 μs. Then, it sequentially receives data through 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 ≈1KB, (3) SCBXP's XML parsing, (4) Buffering the 42 XPath subscriptions of XPath Set1 having a total volume of ≈1KB, (5) XPath subscription mapping in the SMT and registration in the CRT (the volume of the total filter data for XPath Set1 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.

X2CBBR transits from the *(NOP)* mode to the *(nP,nS)* mode and stays there up to S_{max}, then steps to the *(nP,oS)* mode (see Section V-B). Therefore, the first five sequential tasks occur in the *(nP,nS)* mode, while the matching process occurs in the *(nP,oS)* mode. To still emulate the worst-case, X2CBBR has to step back to the *(NOP)* mode by means of an applied reset signal after the matching process of the first set of XPath Set1 completes, and the resulting performance is measured. Then, the broker follows the same procedure after loading and matching two series of XPath Set1 subscriptions, and so on up to a total of 252 subscriptions (six loads of XPath Set1). The SCBXP only parses one XML publication in the *(nP,nS)* mode. Table 1 illustrates the corresponding worst-case results delivered by X2CBBR interfaced with XAVI.

These results show that even though more subscriptions consume more processing time, 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 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

TABLE 1. Worst-case results of the broker with hardware interface: first Pub. vs. Subseq. Subs. (up to 252).

All-Sequential Main Broker Tasks [1 Pub. (1KB) vs. 42 Subs.(1KB) up to 252 Subs.(1KB*6)]	Oper. Mode	Clock Freq. (MHz.)	Proc. Time (μs.)	Throughput (Mbps)
SCBXP's CAM Config. (1KB)	(nP,nS)	050	056.34	144.83
SCBXP's XML Pub. Load (1KB)	(nP,nS)	050	056.32	144.88
XML Pub. Parsing (1KB)	(nP,nS)	050	010.54	765.08
All SCBXP Tasks for Pub. (1KB)	(nP,nS)	**050**	**123.20**	**065.45**
XPath Subs. Buffering (1KB)	(nP,nS)	150	046.10	174.92
Subs. Mapping & Registration	(nP,nS)	150	016.08	250.74
Matching 1 Pub. vs.42 Subs.	(nP,oS)	150	020.96	448.85
All Tasks[Pub.(1KB)/Sub.(1KB)]		**50/150**	**206.34**	**078.16**
XPath Subs. Buffering (1KB*2)	(nP,nS)	150	092.20	174.92
Subs. Mapping & Registration	(nP,nS)	150	019.50	413.53
Matching 1 Pub. vs 42*2 Subs.	(nP,oS)	150	021.88	491.40
All Tasks[Pub.(1KB)/Sub.(1KB*2)]		**50/150**	**256.78**	**094.21**
XPath Subs. Buffering (1KB*3)	(nP,nS)	150	138.30	174.92
Subs. Mapping & Registration)	(nP,nS)	150	022.74	531.92
Matching 1 Pub. vs. 42*3 Subs.	(nP,oS)	150	022.79	530.76
All Tasks[Pub.(1KB)/Sub.(1KB*3)]		**50/150**	**307.04**	**105.05**
XPath Subs. Buffering (1KB*4)	(nP,nS)	150	184.40	174.92
Subs. Mapping & Registration	(nP,nS)	150	026.16	616.51
Matching 1 Pub. vs. 42*4 Subs.	(nP,oS)	150	023.71	566.85
All Tasks[Pub.(1KB)/Sub.(1KB*4)]		**50/150**	**357.47**	**112.79**
XPath Subs. Buffering (1KB*5)	(nP,nS)	150	230.50	174.92
Subs. Mapping & Registration	(nP,nS)	150	029.58	681.54
Matching 1 Pub. vs. 42*5 Subs.	(nP,oS)	150	025.54	578.85
All Tasks[Pub.(1KB)/Sub.(1KB*5)]		**50/150**	**408.82**	**118.35**
XPath Subs. Buffering (1KB*6)	(nP,nS)	150	276.60	174.92
Subs. Mapping & Registration	(nP,nS)	150	032.10	753.64
Matching 1 Pub. vs. 42*6 Subs.	(nP,oS)	150	028.47	566.49
All Tasks[Pub.(1KB)/Sub.(1KB*6)]		**50/150**	**460.37**	**122.61**

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.

TABLE 2. Regular-case results of the broker with hardware interface: each new subsequent publication with a new Skeleton vs. 252 already-registered Subs.

Main Broker Tasks [Each Subseq. Pub. vs. 252 Subs.] (SCBXP Loads are All-Sequential)	Oper. Mode	Clock Freq. (MHz.)	Proc. Time (μs.)	Throughput (Mbps.
SCBXP's CAM Config. (1KB)	(nP,oS)	050	056.34	144.83
SCBXP's XML Pub. Load (1KB)	(nP,oS)	050	056.32	144.88
XML Pub. Parsing (1KB)	(nP,oS)	050	010.54	765.08
All SCBXP Tasks for Pub.(1KB)	(nP,oS)	**050**	**123.20**	**065.45**
Matching 1 Pub. vs. 252 Subs.	(nP,oS)	150	028.47	566.49
All Tasks[Pub.(1KB)/Sub.(1KB*6)]		**50/150**	**151.67**	**372.17**

2) REGULAR-CASE SCENARIO

The experiment continues with sending a new XML publication (and a relevant skeleton) at a time to the X2CBBR through XAVI, 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 2 shows the corresponding results. In this regular case, the broker operates in its *(nP,oS)* mode (Section V-B), and completes its tasks in just 151.67 μs. (including 123.20 μs. for XML parsing) with a total of 372.17 Mbps. of throughput.

3) BEST-CASE SCENARIO

In this case, a new XML publication and its skeleton are both assumed to be already stored and ready for parsing, since X2CBBR supports two dedicated links for loading publications and configuring skeletons. The broker processes this new publication and matches it against the 252 already-registered subscriptions. Therefore, the processing time does not include neither the SCBXP's CAM configuration and publication loading time nor the loading and registration time of the subscriptions. Table 3 shows the corresponding results. The broker completes its tasks in just 39.01 μs. with a total of 1447 Mbps. of throughput. However, this case cannot hold continuously when X2CBBR needs to constantly load new publications.

TABLE 3. Best-case results of the broker with hardware interface XAVI: a new Pub. and its Skeleton already-stored vs. 252 already-registered subscriptions.

Main Broker Tasks [Each Stored Pub. vs. 252 Subs.] (Already Pub./CAM Config.)	Oper. Mode	Clock Freq. (MHz.)	Proc. Time (μs.)	Throughput (Mbps)
XML Pub. Parsing (1KB)	(nP,oS)	050	010.54	765.08
Matching 1 Pub. vs. 252 Subs.	(nP,oS)	150	028.47	566.49
All Tasks[Pub.(1KB)/Sub.(1KB*6)]		**50/150**	**39.01**	**1447**

4) 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 1, 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 1 would then be reduced. Providing that the buffer has already stored new 252 subscriptions, the buffering time 276.60μs in Table 1 completely vanishes. Accordingly, the processing time of the next six sets may become 337.17–276.60=60.57μs. For one publication versus one million subscriptions, X2CBBR completes all tasks in a duration that ranges from 0.24 sec to 1.34 sec. The corresponding throughput ranges from 143.4 Mbps to 797.7 Mbps.

Receiving one million publications ($\simeq$1 GB.) with only 252 (six sets) subscriptions can become time-consuming,

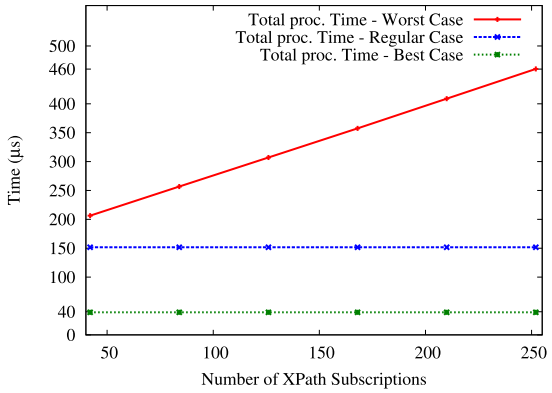


FIGURE 23. Processing time in the worst, regular, and best cases.

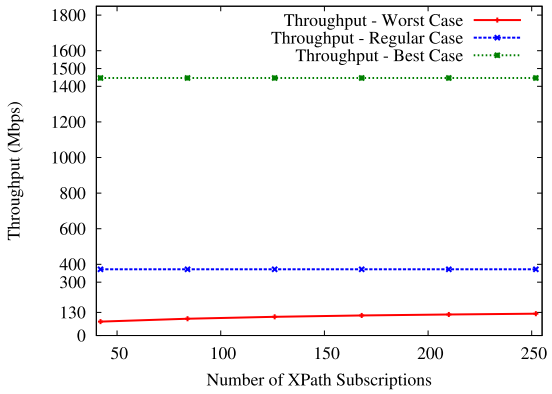


FIGURE 24. Throughput in the worst, regular, and best cases.

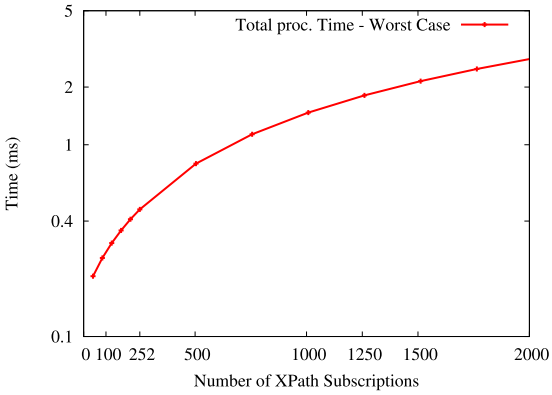


FIGURE 25. Scalability for the first 2000 subs.

since each publication needs to be sequentially matched against subscriptions. X2CBBR processes these subscriptions and matching them against the first publication according to Table 1, with 460.37μs. However, X2CBBR processes the next publication versus these already-registered subscriptions according to Table 2, with 151.67μs. For one million publications versus 252 subscriptions, X2CBBR completes all tasks in $\simeq$2.53 minutes. However, when a publication and its skeleton are already loaded, X2CBBR processes such publication according to Table 3, with only 39.01μs. In this

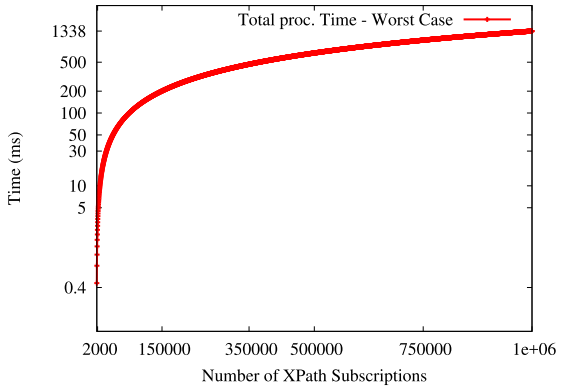


FIGURE 26. Scalability for one million subs.

case, X2CBBR can complete all tasks in a duration that ranges from $\simeq$39.01 sec to $\simeq$2.53 minutes. The corresponding throughput ranges from 53.1 Mbps to 206.7 Mbps (average $\simeq$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 43.4 Mbps to 797.7 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.

C. CASE II: TESTING THROUGH THE SOFTWARE/HARDWARE INTERFACE

When a software interface is involved in driving X2CBBR, some concurrency features have the potential to diminish. The experiments of Case II use the software/hardware interface described in Section VI-A2 and emulate the same scenarios done in Case I. This emulation allows for the comparison of the performance results collected when either interface drives the broker. The first experiment represents the worst-case scenario, while subsequent experiments represent the regular-case and best-case scenarios. In the software test environment of Altera's Nios II [42], C-programs first initialize the broker's memory resources. Then, the programs sequentially send data to the broker, through the software/hardware interface.

To emulate Case I, the software first executes 10000 times each main task intended to be taken into account, while it only executes in one loop iteration a task that should be marginalized. Then, the processing time for one loop iteration is computed by dividing the total processing time by the number of loop iterations (i.e. 10000).

For the worst-case, all tasks are sequentially executed in each loop iteration. In the regular-case, the subscriptions are already registered. Thus, the registration must be marginalized, where the tasks of loading and processing 252 subscriptions are only executed in the first iteration. All other tasks are

executed in all loop iterations. In the best-case, XML parsing and XML/XPath matching are executed 10000 times, but all other tasks are only executed in the first loop iteration.

In Case II, six series of XPath Set1 subscriptions (252 subs. consisting of ≈6KB) are involved in all test cases.

TABLE 4. Worst-case results of the broker with software/hardware interface: first Pub. vs. 252 Subs.

All-Sequential Main Broker Tasks [1 Pub. (1KB) versus 252 Subs.(1KB*6)]	Operation Mode	Proc. Time (ms.)	Throu-ghput (Mbps)
All SCBXP Tasks for Pub.(1KB)	(nP,nS)	01.80	04.48
All Tasks for 252 Subs.(1KB*6)	(nP,nS)/(nP,oS)	08.70	05.56
All Tasks[Pub.(1KB)/Sub.(1KB*6)]		10.50	05.37

1) WORST-CASE SCENARIO

Table 4 shows the worst-case results. This table indicates around 95% of reduction in performance in comparison with the results of XAVI's worst-case stated in Table 1. 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.

TABLE 5. Regular-case results with software/hardware interface: each new subsequent publication with a new Skeleton versus 252 already-registered Subs.

Main Broker Tasks [Each Subseq. Pub. vs. 252 Subs.] (SCBXP Loads are All-Sequential)	Oper. Mode	Proc. Time (ms.)	Throu-ghput (Mbps)
All SCBXP Tasks for Pub.(1KB)	(nP,oS)	1.80	04.48
Matching 1 Pub. vs. 252 Subs.	(nP,oS)	0.10	161.28
All Tasks[Pub.(1KB)/Sub.(1KB*6)]		1.90	29.71

2) REGULAR-CASE SCENARIO

This test can be compared with the regular-case of Case I. Table 5 shows the corresponding results. This table indicates around 92% of reduction in performance, in comparison with XAVI's regular-case results stated in Table 2. However, the performance is around 82% higher than the worst-case of Case II (Table 4), since the processing time drops from 10.5ms to just 1.9ms.

TABLE 6. Best-case results of the broker with software/hardware interface: a new Pub. and its Skeleton already-stored vs. 252 already-registered Subs.

Main Broker Tasks [Each Stored Pub. vs. 252 Subs.] (Already Pub. Load and CAM Config.)	Oper. Mode	Proc. Time (ms.)	Throu-ghput (Mbps)
All SCBXP Tasks for Pub.(1KB)	(nP,oS)	1.10	07.33
Matching 1 Pub. vs. 252 Subs.	(nP,oS)	0.10	161.28
All Tasks[Pub.(1KB)/Sub.(1KB*6)]		1.20	47.04

3) BEST-CASE SCENARIO

This test can be compared with the best-case of Case I. Table 6 shows the corresponding results, where around 92% of reduction in performance is measured in comparison with XAVI's best-case results stated in Table 3. However, in comparison with the worst-case of Case II (Table 4), the processing time drops from 10.5ms to just 1.2ms, which means that the performance is more than 88% higher. With regards of the comparison with the regular-case of Case II (Table 5), the processing time drops from 1.9ms to 1.2ms, which means that the performance is more than 36% higher.

D. CASE III: TESTING THROUGH THE SOFTWARE/HARDWARE INTERFACE WITH CONTENT-BASED ROUTING AND WORST-CASE LINK MANAGEMENT

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 is provided according to the description of Section VI-A3.

The first experiment utilizes six sets of XPath Set2 featuring a total of ≈6 KB of 252 subscriptions. With only 3 mismatched subscriptions per set, a total of 18 mismatched ones exist therein. In this experiment, routing of each mismatched subscription to another broker occurs (lines 40 to 42 in Fig. 18). The second broker processes the received subscriptions, then issues and routes relevant notifications back to the first broker that has to locally deliver them.

The second experiment utilizes six sets of XPath Set3 subscriptions that entirely do not match any publication that is available to the first X2CBBR. Thus, a single mismatched subscription is routed to another X2CBBR on behalf of all mismatched subscriptions (lines 37-38 in Fig. 18). The second X2CBBR processes the received subscription and finds a matched publication. Consequently, it routes this publication to the first broker where re-matching takes place (lines 13 to 17 in Fig. 14). Since now all subscriptions match this publication, the first broker locally delivers the relevant 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 both 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.

The next sections discuss the results of both experiments.

TABLE 7. Results of the broker with software/hardware interface and Sub./Notif. routing.

Sub./Notif. Routing [Each Sub. 24B] (Each Notif. 1KB)	Oper. Mode	Routing 1 Sub. (ms.)	Throu-ghput (Kbps)	Routing 1 Notif. (ms.)	Throu-ghput (Kbps)
	(oP,oS)	01.02	188.23	43.47	187.71

1) FIRST EXPERIMENT RESULTS

Table 7 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. The end-to-end duration from the subscriber site until reception of a relevant notification is therefore 2*(1.02 *ms*) + 2*(43.47 *ms*) + the processing time in each of the two brokers.

Thus, with the processing time of 1.9 ms in each broker (from the regular-case scenario of Case II experiments (Table 5)), the minimum end-to-end path duration $minT_1(s)$ for a subscription that visits two brokers: $minT_1(s) = 2*1.02 + 2*1.9 + 2*43.47 = 92.78$ ms. 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). The routing time of the notification is 43.47 ms per 1KB., which is equivalent to $\simeq$ 1.02 ms per 24 bytes - the routing time of the subscription. Therefore, the routing time indicates the timing per unit-size of either a subscribing action from a subscriber to the local broker or a notification delivery from the local broker to the subscriber. Thus, $minT_1(s) = 2*1.02 + 2*1.90 + 2*1.02 = 7.88$ ms if the size of the subscription was close to that of the notification.

With the processing time of 10.5 ms in each broker (from the worst-case scenario of Case II experiments (Table 4)), the maximum end-to-end path duration $maxT_1(s)$ for a subscription that visits two brokers: $maxT_1(s) = 2*1.02 + 2*10.5 + 2*43.47 = 109.38$ ms; or $maxT_1(s) = 2*1.02 + 2*10.5 + 2*1.02 = 25.08$ ms if the size of the subscription was close to that of the notification.

For each subscription successfully matched in the first broker, no routing to another broker takes place. Therefore, the minimum end-to-end path duration $minT_2(s)$ for a subscription that visits one broker: $minT_2(s) = 1*1.02 + 1*1.9 + 1*43.47 = 46.39$ ms; or $minT_2(s) = 1*1.02 + 1*1.90 + 1*1.02 = 3.94$ ms if the size of the subscription was close to that of the notification.

The maximum end-to-end path consumption time $maxT_2(s)$ for such a subscription is: $maxT_2(s) = 1.02+10.5+43.47 = 54.99$ ms; or $maxT_2(s) = 1.02 + 10.5 + 1.02 = 12.54$ ms if the size of the subscription was close to that of the notification.

TABLE 8. Results of the broker with software/hardware interface and subscription/publication routing.

Sub./Pub. Routing [Each Subscrip. 24B] (Each Notific. 1KB)	Oper. Mode	Routing 1 Sub. (*ms.*)	Throughput (Kbps)	Routing 1 Pub. (*ms.*)	Throughput (Kbps)
	(oP,oS)	01.02	188.23	43.47	187.71

2) SECOND EXPERIMENT RESULTS

Table 8 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/publication are very similar to those of Table 7. However, a single mismatched subscription is only routed to another broker.

The minimum end-to-end path duration $minT(s)$ for a subscription of XPath Set3 is as follows: $minT(s) = (1.02 + 1.90+1.02)_{B1}+(1.90+43.47)_{B2}+(1.90+43.47)_{B1} = 94.68$ ms; or $minT(s) = (1.02+1.90+1.02)_{B1}+(1.90+43.47)_{B2}+(1.90+1.02)_{B1} = 52.23$ ms if the size of the subscription was close to that of the notification.

The maximum end-to-end path duration $maxT(s)$ for a subscription of XPath Set3 is as follows: $maxT(s) = (1.02 + 10.5+1.02)_{B1}+(10.5+43.47)_{B2}+(1.90+43.47)_{B1} = 111.88$ ms; or $minT(s) = (1.02+10.5+1.02)_{B1}+(10.5+43.47)_{B2}+(1.90+1.02)_{B1} = 69.43$ ms if the size of the subscription was close to that of the notification.

TABLE 9. 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. T.(a) (*ms.*)	1st.Exper. Min. T.(b) (*ms.*)	2nd.Exper. Min. T.(a) (*ms.*)	2nd.Exper. Min. T.(b) (*ms.*)
1020	24	46.39	03.94	94.68	52.23

3) COMPARISON BETWEEN FIRST AND SECOND EXPERIMENTS

Referring to the results of both experiments of Case III, Table 9 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. Two cases appear in the table for each experiment: (a) When the notification size is $\simeq$1KB., 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 1KB. The results in this table confirm that a subscriber whose subscription visits more than a broker must wait longer to receive the relevant notification.

TABLE 10. Maximum 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. Max. T.(a) (*ms.*)	1st.Exper. Max. T.(b) (*ms.*)	2nd.Exper. Max. T.(a) (*ms.*)	2nd.Exper. Max. T.(b) (*ms.*)
1020	24	109.38	25.08	111.88	69.43

Table 10 provides the maximum waiting time for a subscriber who sends a subscription up to the reception of the corresponding notification in the first and second experiments. This table shows that the difference in the maximum duration is much closer than that of the minimum duration of Table 9.

The overall performance is the overall accumulated time for all 252 subscriptions to travel to the first broker until all relevant notifications reach corresponding subscribers. In the presence of only one link, sequential transmission of data occurs where all the traffic is queued. In the first experiment, there are 18 mismatched subscriptions that must be queued

TABLE 11. Overall performance comparison between the first and the second experiments of Case III, with sequential data transmission on a single link.

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

and routed and 18 relevant notifications are to be queued and routed as well. In the second experiment, only a single mismatched subscription and a single publication are routed. In the case that sequential transmission and queuing are into effect for all 252 subscriptions, Table 11 compares between the overall performance computed in both experiments. This table indicates the waiting time for 252 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.

While the maximum waiting time in the first experiment is 12.5 sec for a subscriber to receive a notification, 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 that avoid redundant subscription and notification traffic. When the notification size is close to the subscription size, the maximum waiting time in the second experiment (1.3 sec) is slightly greater than that of the first experiment (1.06 sec), even though that the matching process runs twice in the first broker. This overall performance also reflects the efficiency of the X2CBBR in processing and matching subscriptions against publications.

The subsequent transmission of data in the last two experiments leads to these worst-case results in terms of link management. However, X2CBBR has the ability of concurrently managing six external links to highly improve performance.

VII. CONCLUSION AND RECOMMENDATIONS

X2CBBR is a versatile XML/XPath Content-Based Broker/Router that provides high performance for content-based publish/subscribe systems, and can be well-integrated with IoT. It's hardware implementation on an FPGA device represents a prototype that has been tested with different interface options. The embedded hardware mechanisms yield versatility features that include the ability of both avoiding redundant traffic and coping with different pub/sub systems.

When the sole hardware interface is used with the Avalon bus, X2CBBR can regularly achieve 372 Mbps of throughput and even 1447 Mbps of throughput for the best case. However, when the software/hardware interface is used, the performance significantly drops, especially when sequential content-based routing through a single link takes place.

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, taking advantage of finer process technology and higher clock frequency. However, the reconfigurable nature of FPGAs better suits many IoT applications, where a new feature may be added to the broker hardware in the future without the need to rebuild the architecture from scratch.

X2CBBR is also very useful as a pub/sub FPGA resource in the cloud. In this case, the end-user or a "thing" (either a subscriber or a publisher) communicates with the cloud using a cloud operating system such as OpenStack [48]; and X2CBBR provides a virtualized FPGA-based pub/sub service to the user. The overall performance in such a case would depend on the efficiency of the cloud communication interface. Studying such an approach would be interesting in the future, especially that the idea of inclusion of FPGAs into the cloud is growing in the research community [49], [50].

ACKNOWLEDGMENT

The authors would like to mention that many of the experiments, simulations, and results were done at University of Ottawa labs, and they thank all labs' members who cooperated in providing and maintaining necessary tools to achieve their goals.

REFERENCES

[1] *PURSUIT Project*. Accessed: Apr. 2018. [Online]. Available: http://www.fp7-pursuit.eu

[2] *PSIRP Project*. Accessed: Apr. 2018. [Online]. Available: http://www.psirp.org

[3] N. Fotiou, P. Nikander, D. Trossen, and G. C. Polyzos, "Developing information networking further: From PSIRP to PURSUIT," in *Broadband Communications, Networks, and Systems* (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering), vol. 66. Berlin, Germany: Springer, 2012, pp. 1–13.

[4] F. El-Hassan and D. Ionescu, "A hardware architecture of an XML/XPath broker for content-based publish/subscribe systems," in *Proc. Int. Conf. Reconfigurable Comput. FPGAs (ReConFig)*, Cancún, Mexico, Dec. 2010, pp. 138–143.

[5] W3C. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Accessed: Apr. 2018. [Online]. Available: https://www.w3.org/TR/xml/

[6] P. Saint-Andre, *Extensible Messaging and Presence Protocol (XMPP): Core*, document RFC 6120, IETF, Mar. 2011.

[7] *XML Path Language (XPath)*. Accessed: Apr. 2018. [Online]. Available: http://www.w3.org/TR/xpath

[8] W3C. *XQuery 3.1: An XML Query Language*. Accessed: Apr. 2018. [Online]. Available: http://www.w3.org/TR/xquery/

[9] *MQTT Version 3.1.1*. Accessed: Apr. 2018. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

[10] OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0. Accessed: Apr. 2018. [Online]. Available: http://docs.oasis-open.org/amqp/core/v1.0/amqp-core-complete-v1.0.pdf

[11] W3C. *SPARQL Query Language for RDF*. Accessed: Apr. 2018. [Online]. Available: https://www.w3.org/TR/rdf-sparql-query/

[12] Z. Shelby, K. Hartke, and C. Bormann, *The Constrained Application Protocol (CoAP)*, document RFC 7252, IETF, Jun. 2014.

[13] H. Wang, D. Xiong, P. Wang, and Y. Liu, "A lightweight XMPP publish/subscribe scheme for resource-constrained IoT devices," *IEEE Access*, vol. 5, pp. 16393–16405, Sep. 2017.

[14] E. Onica, P. Felber, H. Mercier, and E. Rivière, "Confidentiality-preserving publish/subscribe: A survey," *ACM Comput. Surv.*, vol. 49, no. 2, p. 27, Nov. 2016.

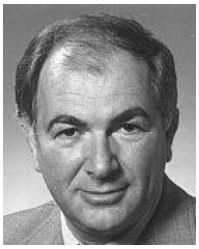
[15] L. Roffia *et al.*, "A semantic publish-subscribe architecture for the Internet of Things," *IEEE Internet Things J.*, vol. 3, no. 6, pp. 1274–1296, Dec. 2016.

[16] A. Hakiri, P. Berthou, A. Gokhale, and S. Abdellatif, "Publish/subscribe-enabled software defined networking for efficient and scalable IoT communications," *IEEE Commun. Mag.*, vol. 53, no. 9, pp. 48–54, Sep. 2015.
[17] Object Management Group. *Data Distribution Service (DDS)*. Accessed: Apr. 2018. [Online]. Available: http://www.omg.org/omg-dds-portal/
[18] M. A. Triawan, H. Hindersah, D. Yolanda, and F. Hadiatna, "Internet of Things using publish and subscribe method cloud-based application to NFT-based hydroponic system," in *Proc. IEEE 6th Int. Conf. Syst. Eng. Technol. (ICSET)*, Bandung, Indonesia, Oct. 2016, pp. 98–104.
[19] Ecma International. (Dec. 2017). *The JSON Data Interchange Syntax*. [Online]. Available: https://www.ecma-international.org/publications/standards/Ecma-404.htm
[20] M. Altinel and M. J. Franklin, "Efficient filtering of XML documents for selective dissemination of information," in *Proc. 26th Int. Conf. VLDB*, Cairo, Egypt, Sep. 2000.
[21] Y. Diao, S. Rizvi, and M. J. Franklin, "Towards an Internet-scale XML dissemination service," in *Proc. 30th Int. Conf. VLDB*, Toronto, ON, Canada, Aug./Sep. 2004, pp. 612–623.
[22] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi, "Efficient filtering of XML documents with XPath expressions," in *Proc. 18th ICDE*, San Jose, CA, USA, Feb./Mar. 2002.
[23] C.-Y. Chan and Y. Ni, "Content-based dissemination of fragmented XML data," in *Proc. 26th IEEE ICDCS*, Lisboa, Portugal, Jul. 2006, p. 44.
[24] R. Chand and P. Felber, "XNET: A reliable content-based publish/subscribe system," in *Proc. 23rd IEEE Int. Symp. Reliable Distrib. Syst. (SRDS)*, Florianpolis, Brazil, Oct. 2004, pp. 264–273.
[25] R. Chand and P. Felber, "Scalable distribution of XML content with XNet," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 4, pp. 447–461, Apr. 2008.
[26] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer, "Path sharing and predicate evaluation for high-performance XML filtering," *ACM Trans. Database Syst.*, vol. 28, no. 4, pp. 467–516, Dec. 2003.
[27] Y. Sugata, T. Ohkawa, K. Ootsu, and T. Yokota, "Acceleration of publish/subscribe messaging in ROS-compliant FPGA component," in *Proc. Int. Symp. Highly-Efficient Accel. Reconfigurable Technol. (HEART)*, Bochum, Germany, Jun. 2017, Art. no. 13.
[28] M. Quigley, B. Gerkey, and W. D. Smart, *Programming Robots With ROS*, 1st ed. Newton, MA, USA: O'Reilly Media, Dec. 2015.
[29] A. Mitra, M. Vieira, P. Bakalov, V. Tsotras, and W. Najjar, "Boosting XML filtering through a scalable FPGA-based architecture," in *Proc. Biennial Conf. Innov. Data Syst. Res. (CIDR)*, USA, Jan. 2009.
[30] R. Moussalli, M. Salloum, W. Najjar, and V. Tsotras, "Accelerating XML query matching through custom stack generation on FPGAs," in *Proc. HiPEAC*, Pisa, Italy, Jan. 2010, pp. 141–155.
[31] Y. Wang and X. Ma, "A general scalable and elastic content-based publish/subscribe service," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 8, pp. 2100–2113, Aug. 2015.
[32] M. Diallo, V. Sourlas, P. Flegkas, S. Fdida, and L. Tassiulas, "A content-based publish/subscribe framework for large-scale content delivery," *Comput. Netw.*, vol. 57, no. 4, pp. 924–943, Mar. 2013.
[33] G. Li, V. Muthusamy, and H.-A. Jacobsen, "Adaptive content-based routing in general overlay topologies," in *Proc. ACM/IFIP/USENIX 9th Int. Middleware Conf. (Middleware)*, Leuven, Belgium, Dec. 2008, pp. 1–21.
[34] M. Linderman, N. Ahmed, J. Metzler, and J. Bryant, "A hybrid publish subscribe protocol," in *Proc. ACM/IFIP/USENIX 9th Int. Middleware Conf. (Middleware)*, Leuven, Belgium, Dec. 2008, pp. 24–29.
[35] R. Zhang and Y. C. Hu, "HYPER: A hybrid approach to efficient content-based publish/subscribe," in *Proc. 25th IEEE Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Columbus, OH, USA, Jun. 2005, pp. 427–436.
[36] F. Cao and J. P. Singh, "MEDYM: Match-early with dynamic multicast for content-based publish-subscribe networks," in *Proc. ACM/IFIP/USENIX 6th Int. Middleware Conf. (Middleware)*, Paris, France, Nov. 2005, pp. 292–313.
[37] F. El-Hassan, "Hardware architecture of an XML/XPath broker/router for content-based publish/subscribe data dissemination systems," Ph.D. dissertation, Dept. Elect. Comput. Eng., Univ. Ottawa, Ottawa, ON, Canada, Feb. 2014.
[38] F. El-Hassan and D. Ionescu, "SCBXP: An efficient CAM-based XML parsing technique in hardware environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 11, pp. 1879–1887, Nov. 2011.
[39] Altera. *Cyclone IV*. Accessed: Apr. 2018. [Online]. Available: https://www.altera.com/products/fpga/cyclone-series/cyclone-iv/support.html
[40] Terasic. *Altera DE2-115 Development and Education Board*. Accessed: Apr. 2018. [Online]. Available: http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=502
[41] Altera. *Avalon Interface Specifications*. Accessed: Apr. 2018. [Online]. Available: http://www.altera.com/literature/manual/mnl_avalon_spec.pdf
[42] Altera. *Nios II Classic Software Developer's Handbook*. Accessed: Apr. 2018. [Online]. Available: http://www.altera.com/literature/hb/nios2/n2sw_nii5v2.pdf
[43] Altera. *Embedded Design Handbook*. Accessed: Apr. 2018. [Online]. Available: http://www.altera.com/literature/hb/nios2/edh_ed_handbook.pdf
[44] Altera. *Platform Designer (Formerly Qsys)*. Accessed: Apr. 2018. [Online]. Available: https://www.altera.com/products/design-software/fpga-design/quartus-prime/features/qts-platform-designer.html
[45] Altera. *Intel FPGA Triple-Speed Ethernet IP Core User Guide*. Accessed: Apr. 2018. [Online]. Available: http://www.altera.com/literature/ug/ug_ethernet.pdf
[46] *IEEE Draft Standard for Ethernet*, IEEE Standards 802.3, 2015. [Online]. Available: http://standards.ieee.org/develop/project/802.3.html
[47] Mentor Graphics. *ModelSim*. Accessed: Apr. 2018. [Online]. Available: http://www.model.com/
[48] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "OpenStack: Toward an open-source solution for cloud computing," *Int. J. Comput. Appl.*, vol. 55, no. 3, pp. 38–42, Jan. 2012.
[49] A. M. Caulfield *et al.*, "A cloud-scale acceleration architecture," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Taipei, Taiwan, Oct. 2016, pp. 1–13.
[50] N. Tarafdar, N. Eskandari, T. Lin, and P. Chow, "Designing for FPGAs in the cloud," *IEEE Design Test*, vol. 35, no. 1, pp. 23–29, Jan./Feb. 2018.

FADI T. EL-HASSAN (M'07) received the B.Sc. degree in electrical engineering from Beirut Arab University, Lebanon, the M.A.Sc. degree in electrical engineering from Carleton University, Ottawa, ON, Canada, and the Ph.D. degree in electrical and computer engineering from the University of Ottawa, Canada.

He is currently an Assistant Professor with the College of Engineering, Al Ain University of Science and Technology, UAE.

He has several years of working experience in the hardware design industry. His research interests include Internet of Things, wireless sensor networks, high-performance computing systems, hardware accelerators, network security, and embedded systems.



DAN IONESCU (SM'85) received the engineering Diploma and Ph.D. degrees in control and computers from the Polytechnic Institute of Bucharest, Romania, and the Diploma degree in mathematics from the University of Timisoara, Romania.

He was with the Polytechnic Institute Traian Vuia of Timisoara (currently Politehnica University of Timişoara) from 1973 to 1983. He has been with the University of Ottawa, Canada, since 1985. He is currently the Director of the Network Computing and Control Technologies Research Laboratory and also directs the technical activity of the NGN NCIT*net2 of the National Capital Institute for Telecommunications.

His research interests include control to real-time systems, image processing, and distributed computing with applications to NGNs and services. His research activities led to the creation of a few start-ups in the above areas.