Pub/Sub on Stream: A Multi-Core Based Message Broker with QoS Support

Zhaoran Wang[1], Yu Zhang[2], Xiaotao Chang[2], Xiang Mi[1], Yu Wang[1], Kun Wang[2], Huazhong Yang[1]
[1]Dept. of Electronic Engineering, TNList, Tsinghua University
[2]IBM Research - China
zhaoranwang@gmail.com, zhyu@cn.ibm.com, yu-wang@tsinghua.edu.cn

ABSTRACT

Publish/Subscribe (Pub/Sub) is becoming an increasingly popular message delivery technique in the Internet of Things (IoT) era. However, classical Publish/Subscribe is not suitable for some emerging IoT applications such as smart grid, transportation and sensor/actuator applications due to its lack of QoS capability.

To meet the requirements for QoS in IoT message delivery, in this paper we propose the first Publish/Subscribe message broker with the ability to actively schedule computation resources to guarantee QoS requirements. We abstract the message matching algorithm into a task graph to express the data flow, forming a task-based stream matching framework. Based on the framework, we explore a message dispatching algorithm called Smart Dispatch and a task scheduling algorithm called DFGS to guarantee different QoS requirements.

Experiments show that, the QoS-aware system can support more than 10x throughput than QoS-ignorant systems in representative Smart Grid cases. Also, our system shows near-linear scalability on a commodity multi-core machine.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: [Information filtering, Information Search and Retrieval]

General Terms

Management, Performance, Human Factors

Keywords

Publish/Subscribe; Stream Computing; Quality of Service; Internet of Things; Smart Grid;

1. INTRODUCTION

Publish/Subscribe (Pub/Sub) [25] is a promising messaging paradigm for future IoT applications because of its loosely-coupled and scalable nature [47]. In a Publish/-Subscribe communication system as shown in Fig. 1, publishers generate messages, called *events*, and send them to a machine called *message broker*; subscribers register their interests in certain patterns of messages, called *subscriptions*, onto message brokers. The message broker matches events generated by publishers with subscribers' interests, then redirect events to corresponding subscribers.

There are two categories of Publish/Subscribe systems [25], one is topic-based, the other is content-based. The difference between the two categories lies in how the message broker matches the events against the subscriptions. Events are made up of *predicates*, in the form of *"<attribute=value>"*. The topic-based Publish/Subscribe system matches messages using the attribute, while the content-based Publish/Subscribe system matches messages according to the value. For example, in a topic-based Publish/Subscribe system, the message *"<price=50>"* are redirected to the subscribers with interests in *"price"* rathefr than *"size"*. In a content-based Publish/Subscribe system, subscriptions are in the form of *"<attribute (comparison operator) value>"*. For example, the message *"<price=50>"* will be sent to subscribers interested in *"<price<60>"* rather than *"<price<40>"*. The content-based Publish/Subscribe system is an emerging alternative to the traditional topic-based Publish/Subscribe system, since it permits more flexible subscriptions along multiple dimensions. However, the content-based Publish/-Subscribe system is more compute-intensive because it inspects the content of the message. In this paper, our discussion focuses on but not limited to content-based Publish/-Subscribe.

Some major applications of IoT [28, 12], especially Smart Grid [13, 33, 14], introduce various requirements on QoS. Among multiple dimensions of QoS requirements, latency [58, 9, 42] is one of the most important metrics. For example, in the smart grid monitoring scenario, when the electrical generator is going to fail, the control center should be able to receive the message of failure within a certain latency. Otherwise, corresponding control actions trigger by this failure message may be too late to be effective, causing severe power grid failure like the Northeast blackout of 2003 [4]. In a QoS-ignorant message broker, the QoS requirements of messages with higher priority may be violated if messages with lower priority occupy too many computation resources as shown in Fig. 2. Previous efforts in sequential [26] matching algorithms, parallel matching algorithms on CMP or heterogeneous hardware such as FPGA and GPU mainly focus on achieving higher peak performance. All of

DEBS 2012, July 16–20, 2012, Berlin, Germany.

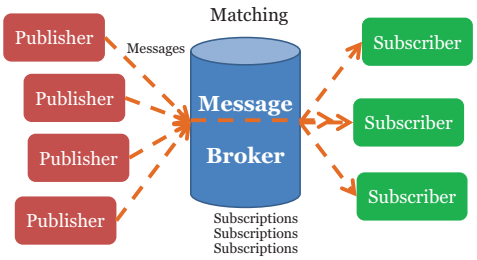

Figure 1: **How a Publish/Subscribe system works.** Publishers generate messages, called *events*, and send them to a machine called *message broker*; subscribers register their interests in certain patterns of messages, called *subscriptions*, onto message brokers. The message broker matches the events generated by publishers with subscribers' interests, then redirect events to the corresponding subscribers.

these previous works ignore the QoS requirements of different message groups.

Our work concentrates on supporting different QoS requirements from different groups of messages by leveraging commodity multi-core processors. From Section 2, we observe that the ability in two aspects of the system are essential for supporting QoS requirements. The first aspect is enabling intra-core resource allocation according to the QoS requirement. The second aspect is enabling messages to be dispatched inter-core with QoS in mind. Thus, we build a novel message broker with a two-level resource allocation mechanism. One is intra-core and the other is inter-core. With the first level mechanism called deadline-aware fine-grained scheduling (DFGS), which schedules the processing of messages with different QoS requirements in each core. The second level mechanism is Smart Dispatch, our message broker is able to smartly dispatch messages across cores with QoS considerations. To implement this mechanism, the sequential algorithm has to be decomposed into schedulable segments, which we call *tasks*. We build a system framework called *stream matching framework*, in which the message matching algorithm is abstracted into a data flow graph made up of tasks. By scheduling tasks both intra-core and inter-core, our system is able to meet different QoS requirements.

The contributions of this paper are as follows:

- Enable QoS support in a Publish/Subscribe message broker with a multi-core processor.
- Propose an intra-core fine-grained scheduling and an inter-core message dispatching mechanism to provide the QoS support in a multi-core Publish/Subscribe message broker.
- Discuss how parameters, such as the period, processing time of the message flow and the number of cores used, affect the message failure rate.
- Get more than 10x maximum throughput than QoS-ignorant Publish/Subscribe system and achieving near-linear scalability on a multi-core processor.

This paper is organized as follows. Section 2 describes the motivation case. Section 3 introduces the related work.

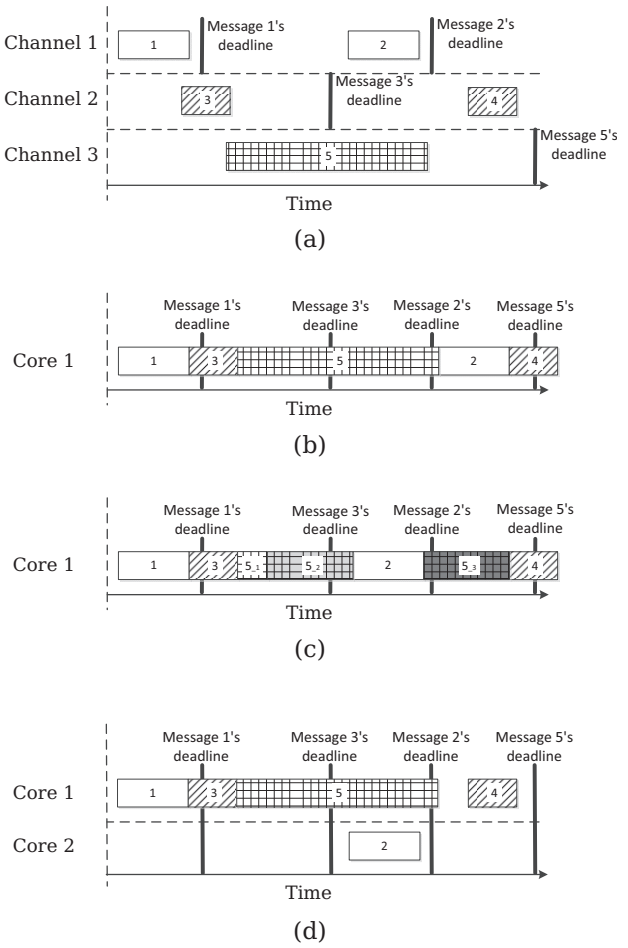

Figure 2: **(a). The arriving moment, processing time and deadline of each message.** The beginning of each bar represents the moment that the message arrives; the length of the bar in the figure represents the processing time of each message; deadlines are represented by vertical lines. **(b). The latency requirement of message 2 is violated due to inappropriate scheduling.** Messages are processed in the order they arrive at the broker. Message 2 from channel 1 won't be finished before its deadline, since message 5 of channel 3 occupies a long processing time. **(c). Resolve the QoS violation by splitting and scheduling.** Message 5 from channel 3 is processed within 3 stages. The processing of message 5 from channel 3 is interrupted by message 2 from channel 1, but resume after message 5 is finished. **(d). Resolve the QoS violation by dispatching.** Message 2 from channel 1, which has a stringent latency requirement, is dispatched to core 2 instead of core 1. Because core 1 is not able to guarantee the latency requirement of this message at this moment due to a lack of computation resources.

Section 4 describes basic event matching algorithm and proposes the partition method based on an analysis of the algorithm's execution time. Section 5 describes the stream matching framework. Experiments and analysis are given in Section 6. Finally, we conclude this paper in Section 7.

2. MOTIVATION CASE

In this section, we demonstrate a motivation case in which QoS requirements are violated because of inappropriate allocation of resources. Thus we are motivated to propose the two-level scheduling mechanism. One is the intra-core level scheduling mechanism called Deadline-aware Fine-Grained Scheduling (DFGS), while the other is the inter-core level scheduling mechanism called Smart Dispatch.

In the following discussion, we call the group of messages a *channel* if the messages have the same kind of QoS requirement. It is called a *failure* when a message's QoS requirement, especially latency, is violated. In this motivation case, the message broker with a uni-core processor is responsible for delivering messages from 3 channels.

In Fig. 2.(a), the beginning of each bar represents the moment that the message arrives; the length of the bar in the figure represents the processing time of each message. The messages of channel 1 have very stringent latency requirements. That is to say, the message broker has to finish processing a message of channel 1 in a relatively short period of time after it arrives at the broker. Messages from channel 2 and channel 3 have relatively slack latency requirements. In this case the message broker takes a long time to finish processing the message from channel 3.

Problem will occur if the messages are processed in the order they arrive at the broker. As shown in Fig. 2.(b), message 2 from channel 1 won't be finished before its deadline, since message 5 of channel 3 occupies a long processing time. Thus, the QoS requirement of message 2 is violated in this case.

2.1 Deadline-aware Fine-Grained Scheduling (intra-core)

The intra-core level mechanism called *Deadline-aware Fine-Grained Scheduling (DFGS)* is able to split the long processing stage for a message into several stages. These stages for a single message don't have to be executed successively. As shown in Fig. 2.(c), message 5 from channel 3 is processed within 3 stages. When message 2 from channel 1 arrives, it needs to be processed as soon as possible due to its stringent latency requirement. With DFGS mechanism, message 2 from channel 1 is processed before message 5 from channel 3 is finished. In other words, the processing of message 5 from channel 3 is interrupted by message 2 from channel 1, but resume after message 5 is finished. We are going to provide details of this algorithm in Section 5.2.

2.2 Smart dispatch (inter-core)

In a message broker with a multi-core processor, the inter-core level mechanism called *Smart Dispatch* is able to dispatch the message to the appropriate core, which has enough resources to guarantee the QoS requirement.

As shown in Fig. 2.(d), message 2 from channel 1, which has a stringent latency requirement, is dispatched to core 2 instead of core 1. Because core 1 is not able to guarantee the latency requirement of this message at this moment due to a lack of computation resources. We are going to provide details of this algorithm in Section 5.3.

2.3 Conclusion of the 2 level mechanism

From this motivation case, we can conclude that, to guarantee more stringent QoS requirements without any violation, the message broker with multi-core processor should be able to do both intra-core and inter-core scheduling. To enable intra-core scheduling, 1) the sequential algorithm should be partitioned into segments; 2) the segments should be scheduled according to the priority of messages. To enable inter-core scheduling, 1) the broker should be able to check the available resources on each core; 2) dispatch messages to cores with enough resources. Thus, we propose the task-based framework, in which the message matching algorithm is abstracted into a data flow graph made up of tasks. Through intra-core and inter-core scheduling of these tasks, the system is able to meet the QoS requirements of messages. This framework along with the two level mechanism are described in Section 5. To effectively partition the sequential algorithm, the original algorithm [26] is described and analyzed in Section 4.

3. RELATED WORK

In this section, we provide related works and demonstrate our novelty.

3.1 General Publish/Subscribe research

Publish/Subscribe paradigm [25] provides asynchronous event delivering and notification service in a distributed network. It is widely used in several kinds of distributed applications, such as enterprise message delivering [16, 18, 19] and event notification in mobile systems [32]. Publish/Subscribe systems can be categorized into topic-based and content-based systems [25]. The content-based Publish/Subscribe system permits more flexible subscriptions than topic-based ones while increasing the computation overhead.

Existing research works mainly focus on two aspects of content-based Publish/Subscribe systems. The first aspect is efficient event routing algorithms in a distributed network [17], which is a widely investigated topic in the network research community. The routing solutions are categorized [17] into filter-based approaches [18, 19, 46, 56] and multicast-based approaches [46, 8, 51, 59]. In the prior approach, the routing path of an event is decided by content-based matching at every node it reached. While in the latter approach, the multicast groups of destinations are determined at the moment that the event is produced. The second aspect is event processing (or event matching or filtering) in a message broker, which is gaining attention by the database and middleware research community. Performance such as throughput and processing time (latency) is the most important issue that the existing efforts try to address [10, 22, 26], which is more related to our paper. The works on event processing is described in details below.

3.2 Event processing algorithms in Publish/-Subscribe systems

According to [25], the event matching algorithm are categorized into two-phase algorithms and compilation methods. The two phase algorithms generally evaluate predicates in the events against all predicates in the subscriptions, generating an intermediate result. Then the matched subscriptions are computed according to this intermediate result. This category includes several previous works. SIFT [60] is an early Publish/Subscribe system in which the text documents are subscribed based on keywords' weights. SIFT uses the counting algorithm for the second phase. YFilter [22] uses the non-deterministic finite automation approach to match events in the form of extensible markup language

(XML). [26] improves the efficiency of the two phases algorithm by clustering the subscriptions based on selective common predicates, which are called Access Predicates (AP) in later discussion. [48, 49, 11] further improve the matching algorithm in [26] with pre-fetching, multi-attribute hash tables, access predicates and dynamic clustering. More recent works in this field include relevant subscriptions matching [53] and efficiently index Boolean expressions over a high-dimensional discrete space [54].

The other class of matching algorithms compile predicates [10, 29, 31, 35] into tree-like structure to test the event against subscriptions, which is represented as leaf nodes in the tree. The event are evaluated from the root of the subscription tree to the leaf nodes. In [10], one leaf node is corresponding to one subscription. In [29], each subscription is represented by several leaf nodes. Compared with two-phase algorithms, the compilation methods lack spatial or temporal locality and has larger overhead in storing and dynamically modifying the subscriptions. So we choose a typical two-phase matching approach in [26] for its advantages in storage and higher performance. Our system can also be applied to other two-phase matching algorithms and compilation based matching algorithms.

3.3 High performance event processing in Publish/Subscribe systems

Recent research proposed several high throughput and low latency Publish/Subscribe systems using multi-core processors [27] and heterogeneous hardware such as FPGA [52, 55, 40, 38, 41, 45, 44, 37, 24, 50] and GPU [43, 23, 39].

- **Multi-core CPU.** In [27], the event matching algorithm proposed in [26] is parallelized leveraging chip multi-processors, increasing the throughput to over 1600 events/second with eight cores and reducing the processing latency by 74%.
- **FPGA.** [55] presents fpga-ToPSS, which accelerates the two-phase matching algorithm [26] with FPGA for high-frequency and low-latency algorithmic trading. [38, 37, 41, 44, 45, 24] present content-based broker on FPGA to provide brokering service of XML metadata of publications against the XPATH subscriptions. Among them, [44, 45] propose algorithm modifications for FPGA to prevent false positives and make throughput independent from the complexity of user queries or the input XML stream. [38, 37] use FPGAs as the co-processor of a high performance computer, while [41] contains FPGA only. [24] focuses on mobile applications. Besides, [50] proposes the FEACAN system for front-end acceleration of content-aware network processing, providing both signature matching and regular expression matching.
- **GPU.** [43] presents a GPU-based XML Path Filtering system, which supports larger subscriptions and faster subscription update than FPGA. [23] builds a demo system by integrating GPU into the ZeroMQ message middleware [7]. [39] proposes CUDA Content-based Matcher (CCM) on GPU to accelerate matching in content-based Publish/Subscribe systems.

3.4 Industry products of Publish/Subscribe systems

There are also related industry products called Message-oriented middleware (MOM) [15, 6, 21, 20], which is software or hardware infrastructure supporting sending and receiving messages between distributed systems. Most MOMs provide Publish/Subscribe service as well Point-to-Point (P2P) services. Currently there are several industry products. Among them, IBM WebSphere MQ [2], Apache ActiveMQ [1] and RabbitMQ [5] are the popular commodity systems. There are also other prototypes, such as ZeroMQ [7] and Mosquitto [3], aiming at providing low-latency and high-throughput messaging service.

3.5 Scheduling in operating systems

Thread/process scheduling problem in operating systems is a widely researched subject[57]. There are First-Come, First-Served (FCFS) Scheduling, Shortest-Job-First (SJF) Scheduling, Priority Scheduling, Round Robin (RR) Scheduling, Multilevel Queue Scheduling, Real-time Scheduling etc. The scheduling algorithm in our work is similar with Real-time Scheduling methods.

3.6 Novelty of our work

To the best of our knowledge, none of the existing works mentioned the ability of QoS support in our work. Our work is completely orthogonal with existing research and industry product in Publish/Subscribe systems. Firstly, our work is not proposing any new sequential matching algorithm. Instead, our work rebuilds the system and changes the way the sequential algorithm is executed to enable QoS support taking advantage of the previous matching algorithm [26]. At the same time, our methods applies to other sequential matching algorithms as well. Secondly, compared with previous works in high performance Publish/Subscribe systems with either multi-core or heterogeneous hardware, our work focuses on providing QoS support to the applications with critical QoS requirements leveraging multi-core processor. Thirdly, industry products mainly provide topic-based Publish/Subscribe service targeting at messaging in enterprise applications. Our work is targeting at IoT applications and provides content-based Publish/Subscribe ability. Finally, even we take a similar approach in the operating system area, our novelty lies mainly in building a schedulable framework and applying the mature scheduling algorithm to the emerging real time Publish/Subscribe problem.

4. ANALYSIS AND PARTITIONING OF THE SEQUENTIAL MATCHING ALGORITHM

The broker matches the messages with the interests of the subscribers. Several sequential event matching algorithm are summarized by [25]. Among them [10, 34, 30, 36, 48, 49, 26, 26] is the most representative one, since 1). it provides high throughput and low latency compared with other algorithms; 2). its data structure is space efficient enough to support large workloads and flexible enough to support subscription changes. In this paper, we use this two stage algorithm as our baseline sequential algorithm. We introduce the matching algorithm and how to partition the algorithm into segments.

4.1 Sequential matching algorithm

4.1.1 Data structure

A subscription is made up of several predicates with the form of *"<attribute (comparison operator) value>"*, while an event is in the form of *"<attribute=value>"*.

The matching algorithm uses a predicate bit vector to represents the predicates satisfied by an incoming event. Thus each bit in the predicate bit vector is corresponding to one predicate that occurs in one or more subscriptions. If any predicate in the event satisfies a predicate in the subscription, then the corresponding bit in this event's predicate bit vector is set to "1". For example, there is an incoming event made up of only one predicate *"<price=10>"*. Predicates like *"<price>5>"*, *"<price<3>"* and *"<price>7>"* occur in one or more subscriptions. Thus, corresponding bits for *"<price>5>"* and *"<price>7>"* are set to "1" while the corresponding bit for *"<price<3>"* is set to "0". In order to evaluate the predicates in the events against every predicate occurring in the subscriptions, all of the predicates of subscriptions are stored in tables labeled with the attributes' name as shown in Fig. 3. For example, *"<price>5>"*, *"<price<3>"* and *"<price>7>"* are stored in the same table labeled with *"price"*. Each line in the table store predicates which have the same operator. For example, the first line of the table of *"price"* is for operator *">"*. For example, *"<price>5>"* and *"<price>7>"* are stored in this line.

Subscriptions are stored in a hierarchical structure. The first level is indexed by the cluster bit vector. The bits in the cluster bit vector is used as the reference to the corresponding list of subscription clusters. Each bit in the cluster bit vector is a particular predicate, which is called *access predicate (AP)*. Each subscription can only be stored once in all lists of subscription clusters. The subscriptions in the list of subscription clusters share the same AP. That is to say, subscriptions in the list of subscription clusters won't be satisfied unless the corresponding AP is satisfied, since each subscription is matched by an event only when all the predicates of it are satisfied by the the event. Thus, the list of subscription clusters associated with an AP won't be checked unless this AP is satisfied. Details in choosing the most selective predicates in subscriptions to be APs are described in [26].

4.1.2 The event matching algorithm

The algorithm has two phases as shown in Fig. 3. In the first phase, called *H Phase*, the message broker evaluates the incoming message with the predicates in subscriptions to produce the predicate bit vector described above. In the second phase, called *C Phase*, the subscription clusters are traversed and evaluated based on the predicate bit vector from H Phase.

H Phase.

H Phase is short for Hash Phase. As shown in Algorithm 1, in this stage, every predicate of an incoming event is evaluated against all the predicates in the subscriptions. The attribute name of each predicate of the event is hashed to get the entrance of the table storing corresponding predicates. Then every value in each line of the table is traversed to determine the matched subscription predicates. The predicate bit vector is set according to this result.

C Phase.

C Phase is short for Check subscription clusters Phase. As shown in Algorithm 2, in this stage, the predicate bit vector is used as the input to determine the subscriptions matched. Cluster bit vector is traversed, every entry of which is corresponding to an Access Predicate. Whether an Access Predicate is satisfied is determined by the corresponding bit in the predicate bit vector we got from the H Phase, since the predicate bit vector records the results for all predicates. If the predicate bit vector indicates that this AP is satisfied, then the subscription cluster referred by this entry is to be checked. Here we omit the details [26] of the organization of the subscription clusters and the subscription clusters checking algorithm, since the details are not important for our work. The matched subscriptions are produced after the subscription clusters are checked.

Algorithm 1: H Phase

```
Input: event
Output: predicate_bit_vector
1  for each predicate of event do
2      table_to_check = hash(predicate.attribute);
3      for each line of table_to_check do
4          for each value of the line do
5              if evaluate(event.predicate.value) ==
               satisfied then
6                  set(corresponding bit in predicate_bit_vector);
7              end
8          end
9      end
10 end
11 return predicate_bit_vector;
```

Algorithm 2: C Phase

```
Input: predicate_bit_vector
Output: subscription_matched_set
1  subscription_matched_set = NULL;
2  for each entry of cluster_bit_vector do
3      if predicate_bit_vector[entry.predicate]==1 then
4          subscription_matched_set+ =
           check(entry.subscription_cluster);
5      end
6  end
7  return subscription_matched_set;
```

4.2 Execution-time analysis and partition

4.2.1 Execution-time analysis of the sequential algorithm

We conduct a micro-benchmark using the workload described in [26, 27]. Details of the experimental settings are described in the experiment part. There are totally 1504 different predicates in the subscriptions with 238 unique attributes in the predicates. We choose workloads of 6,000 subscriptions, 60,000 subscriptions and 600,000 subscriptions and measure the time used in H Phase and C Phase. The events are randomly generated by combining the predicates with the uniform distribution. We measure the average processing time for an event. Table 1 describes the processing time of the sequential algorithm and the percentage of each phase.

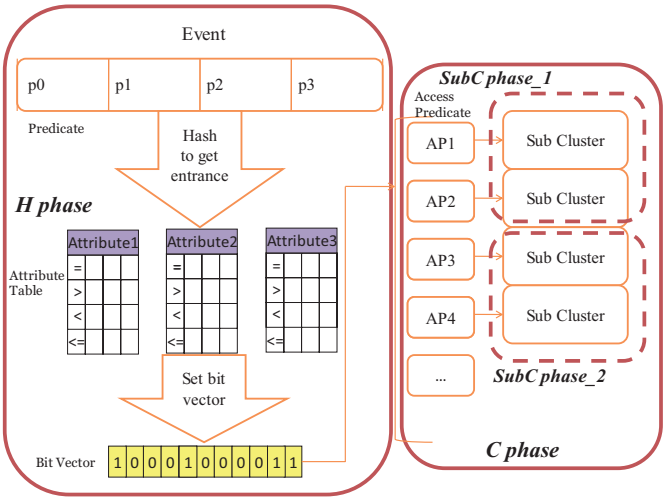

Figure 3: **Partition the matching algorithm.** The algorithm can be split into H Phase and C Phase. H Phase is short for Hash Phase, while C Phase is short for Check subscription clusters Phase. The C Phase is further decomposed into several SubC phases, each of which is responsible for processing a portion of APs.

Table 1: The percentage of each phase in the matching algorithm

Number of Subscriptions	Processing time/ms	H Phase	C Phase
6000	0.034	65%	35%
60,000	0.15	14%	86%
600,000	0.81	3%	97%

4.2.2 *Partition method*

According to Section 2, the sequential algorithm has to be split into several stages to enable scheduling. According to the analysis of the sequential execution time, we propose a 2-level partition method.

First, the algorithm can be split into H Phase and C Phase as stated above. Second, we observe that C Phase takes up a major part of the processing time when the subscription number increases. To enable fine-grained resource scheduling, we further decompose the C Phase into individual SubC Phases, each of which processes a group of the APs as shown in Fig. 3. For example, if there are 100 APs processed by 2 SubC Phases, 1-50 APs are processed by SubC Phase_1, while 51-100 APs are processed by SubC Phase_2.

5. STREAM MATCHING FRAMEWORK

5.1 Basic Framework

In order to schedule the computation resources to guarantee the QoS requirements of messages from different channels, we propose a task-based framework, called *stream matching framework*. In this section, we introduce 1). the method of mapping the original matching algorithm shown in Fig. 3 into a task graph, which describes the data flow as shown in Fig. 4; 2). how the tasks in the task graph are scheduled and executed on the cores as shown in Fig. 5.

In the framework, the program segments mentioned in Section 4 are implemented as *tasks*. In Fig. 4, every line acts as a data buffer. One task may communicate with another task using the data buffer between them if they are connected. A data buffer is called the *input buffer* of a task if the task reads data from the data buffer. Correspondingly, a data buffer is called the *output buffer* of the task if the task writes data into the data buffer. As shown in Fig. 5, when a task is executed, it reads the data from the input buffer, processes the data and then writes the result to the output buffer. For example, the Publish/Subscribe matching algorithm is partitioned as shown in Fig. 3. The H Phase processes the input message to produce the bit vector. In the stream matching framework, the H Phase is described as the H Phase task, which reads the message from the input buffer and writes the bit vector to the output buffer.

To map the segments of the sequential algorithm described in Fig. 3 onto the task graph, we need several auxiliary tasks besides the tasks for data processing as shown in Fig. 4.

- **Source**: The Source task reads the input messages. In the experiments, we use the Source task to emulate the traffic of messages.
- **Splitter**: The Splitter task dispatches the input messages to different processing units. Splitter is responsible for 1). checking the computation resources in each core and dispatch the message; 2). properly inserting the tasks for the message into the task scheduling queue as shown in Fig. 5.
- **AP Splitter**: AP Splitter helps to split the C Phase into several SubC Phases with negligible computation. As discussed in Section 4, this is done by splitting the entire AP List into several groups of APs. The AP Splitter task duplicates the bit vector of a message, which is produced by the H Phase task, and sends the bit vector to several SubC Phase tasks. Each of the SubC Phase tasks processes its corresponding group of APs.
- **AP Joiner**: The AP Joiner task merges the output results of all SubC Phase tasks into an output, which gives the matching result for a message.
- **Joiner**: The Joiner task collects the matching results produced by AP Joiners to form a serialized system output.

In the task graph, tasks are organized by *ways*.We call a group of tasks a *way*, if they are connected as a pipeline and work together to process messages from a certain channel. For example, as shown in Fig. 3, in the matching algorithm, a message needs to be processed by the H Phase and the SubC Phases. Correspondingly, in the task-based framework, as shown in Fig. 4, if a message of channel 1 is dispatched to core k, the H Phase task, the AP Splitter task, several SubC Phase tasks, and the AP Joiner task of channel 1 on core k needs to be executed successively to process the message. They are defined as tasks of way n, which is for channel 1.

On each core, the tasks are executed by the *worker threads* on their *host cores*, as shown in Fig. 4. The core, which a task or a way belongs to, is called the *host core* of the task or the way. On each core, there is a thread executing the tasks,

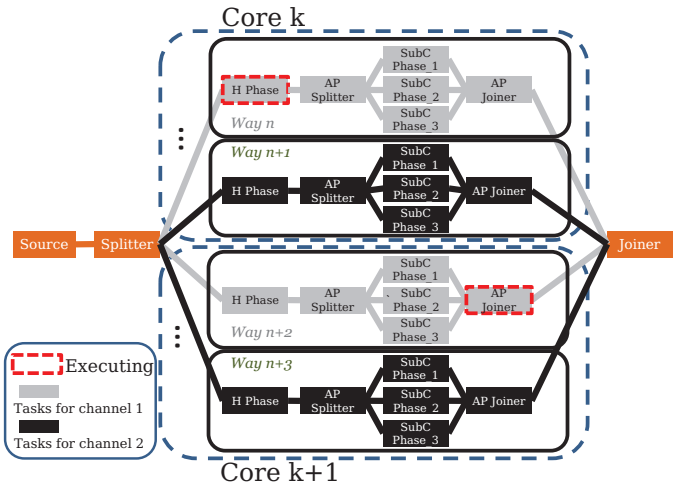

Figure 4: **Task-based framework: Stream.** Each line acts as a data buffer. The tasks are connected via the buffers. A way is a pipeline of tasks, which are responsible for processing a messsage. Each core has several ways, each of which is responsible for processing messages of one channel. For example, way n on core k is responsible for the messages of channel 1. Only one task is executed on each core at the same time. When a task is being executed (in red frame as shown in the figure), it reads data from its input buffer and writes data to its output buffer as shown in Fig. 5.

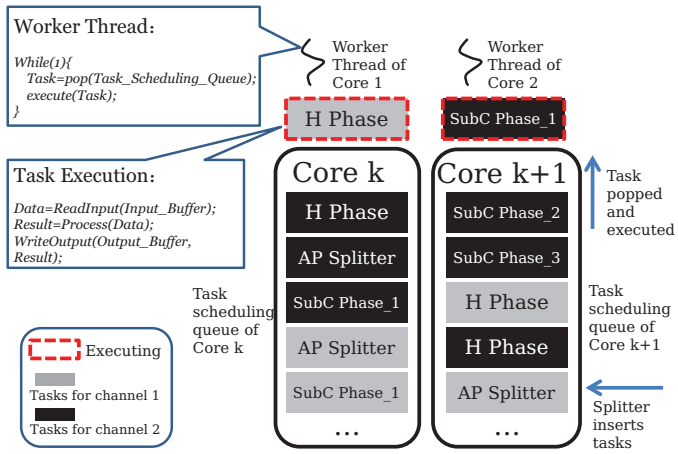

Figure 5: **How tasks are executed on each core.** Each core maintains a task scheduling queue which decides the execution order of tasks. The worker thread on each core fetches the task out of the task scheduling queue and executes it. While the Splitter inserts the tasks into the proper position of the queue. When a task is being executed (in red frame as shown in the figure), it reads data from its input buffer and writes data to its output buffer.

called the *worker thread*. As shown in Fig. 5, there is a *task scheduling queue* on each core hosting the tasks which are to be executed. The task order in the task scheduling queue represents the order in which the tasks are to be executed by the worker thread. The worker thread fetches one task out of the task scheduling queue from the head and executes one task at a time. In the task scheduling queue, the tasks for different channels interleave with each other. As shown in Fig. 5, there are tasks for two channels on each core. In core k, the H Phase task for processing messages of channel 1 is being executed by the worker thread. This task is the H Phase task of way n in Fig. 4. While in core k+1, the SubC Phase_1 task is being executed. This task is shown in Fig. 4 as the SubC Phase_1 on way n+1.

Task scheduling is accomplished by the Splitter and the worker threads on the cores. The Splitter enqueues the tasks for a message into the task scheduling queue. The worker thread dequeues the tasks out of the task scheduling queue on its host core and executes it, as shown in Fig. 5. The Splitter dispatches the message to a certain way of the message's channel. At the same time, it enqueues the corresponding tasks into the host core of that way. For example, the Splitter may dispatch a message of channel 1 to way n on core k, since way n is dedicated to processing messages of channel 1. At the same time, it enqueues the H Phase task, AP Splitter task etc. into the task scheduling queue of core k. The Splitter decides which way to dispatch a message with the Smart Dispatch algorithm. Where in the task scheduling queue to put the tasks for the message is decided by the Deadline-aware Fine-Grained Scheduling (DFGS) algorithm.

5.2 Deadline-aware Fine-Grained Scheduling (DFGS)

We create an algorithm, called Deadline-aware Fine-Grained Scheduling (DFGS), for the Splitter to check the resources of a way's host core with the time complexity of $O(n)$. The Splitter decides where in the task scheduling queue the corresponding tasks for a message should be inserted. That the tasks for the message can be inserted in the task scheduling queue indicates there are enough resources in a way's host core to guarantee the Qos requirement of the message. In this case, DFGS returns true. Otherwise, the algorithm returns false indicating there are not enough resources in this way's host core for the message.

As shown in Algorithm 3, when the Splitter dispatches a message to a way, it inserts the tasks for the message into the task scheduling queue on the host core. The tasks for the message are inserted as a block, called a *task block*. However, once the tasks are inserted, they may not be successive in the queue afterwards, since other task blocks may be inserted between them. A task block, e.g. T, can be inserted to somewhere in the task scheduling queue as long as:

- T is placed after a certain task which would otherwise fail its deadline if T was inserted before it.
- T is placed after the earlier tasks of the same channel so that the intra-channel message order is guaranteed.
- T's deadline is not to be violated.

Firstly, the Splitter traverses the task scheduling queue from the head to the tail to find a task as the mark, as shown in Algorithm 3 from line 3 to 13. The task block must be inserted after the mark to satisfy the first two of these three conditions. Correspondingly, there are two cases where we mark a task in the queue. First, if the task in the queue ("i" in Algorithm 3) would fail its deadline if the new task is inserted before it (line 4 in Algorithm 3). Second, if the task in the queue processes earlier messages of the same

channel as the new task (line 9 in Algorithm 3). If multiple tasks in the queue can be marked, the last one of them is finally set as the mark.

Secondly, according to the third condition, the Splitter then tries to insert the task block for the message right after the mark as shown in line 14. If the deadline of the message is not violated, then the tasks in the task block for the message are inserted into the queue, DFGS returns true. Otherwise, false is returned, indicating there is not enough resources on the host core of this way.

Algorithm 3: Fine-grained scheduling

```
   Input: message, current_core
   Output: whether the message can be successfully inserted
1  task_block = message.tasks_needed;
2  mark = queue.head;
3  for i = queue.head; i ≤ queue.tail; i++ do
4  |  if
   |  i will fail its deadline if task_block is inserted before i
   |  then
5  |  |  mark = i;
6  |  |  //task_block should be inserted after i
7  |  end
8  |  else
9  |  |  if i.channel = message.channel then
10 |  |  |  mark = i;
11 |  |  end
12 |  end
13 end
14 if
   task_block's deadline is violated if it is inserted after mark
   then
15 |  return false;
16 end
17 else
18 |  for j=each task in task_block do
19 |  |  insert(mark, j)
   |  |  //insert the task after the mark
20 |  end
21 |  return true;
22 end
```

5.3 Smart Dispatch

In the Splitter, we implement the algorithm called *Smart Dispatch* to decide which way to dispatch a message.

With Smart Dispatch algorithm, the Splitter tries to choose a way to dispatch the message. The way's host core must have enough computation resources to guarantee the QoS requirement of the message. Therefore, the message's deadline is not going to be violated.

After the Splitter gets its input message, it checks all the ways, which are for processing message's channel, one by one as shown from line 3 to line 10 in Algorithm 4. For example, in Fig. 4, after the Splitter gets a message of channel 1, way n and way n+2 are among the ways to be checked. The Smart Dispatch algorithm decides whether the host core of the way has enough resources using the DFGS algorithm described in next section, as shown in line 4 of Algorithm 4. If the host cores of all the ways fail to provide enough resources to guarantee the deadline of the message, the Splitter deserts the message and returns false. If the message is deserted due to lack of resources, it is counted as a failure.

The detailed resource checking algorithm used in line 4 in Algorithm 4 is explained by the DFGS in Algorithm 3. If no core has enough resources to guarantee the message's QoS requirement after a round of check as in line 13, the message is deserted and recorded as a failure, see Algorithm 4.

Algorithm 4: Smart Dispatch

```
   Input: message
   Output: whether the message is successfully dispatched
1  current_channel = message.channel;
2  ways_to_be_checked =
   all ways for processing current_channel;
3  for current_way in ways_to_be_checked do
4  |  if DFGS(message, current_way) then
5  |  |  return true;
6  |  end
7  |  else
8  |  |  current_way = current_way + 1;
9  |  end
10 end
11 if all ways fail to provide enough resources then
12 |  return false;
13 end
```

6. EXPERIMENT RESULTS

In this part, we set the parameters of message channels based on the QoS parameters in [13]. As shown in Section 5, the task scheduling queue is responsible for determine the execution order of tasks. The scheduling algorithm operates the same with any number of channels. To get a more thorough understanding, we simplify the 5 QoS levels in [13], which corresponds to 5 channels, into 2. We call the faster channel as *channel 1* and the slower one as *channel 2*.

In the experiment part, we firstly introduce the experiment settings. Then we try to discuss how the parameters affect the failure rate of messages in both QoS-ignorant system and our system. Finally we show a maximum throughput and scalability between two systems.

6.1 Experiment settings

We use a 8 core Intel Xeon CPU E5507 running at 2.27GHz. The operating system is Fedora with kernel version 2.6.31.5. We compiled the program with GCC 4.1.1 using -O3 optimization.

We use the dataset described in [26, 27]. There are 1504 independent predicates in the subscriptions, and there are 238 different attributes in the predicates. Channel 1's subscription number is 60,000 while channel 2's is 600,000. We measure the maximum event processing time and set it as the processing time for each message. The event period and deadline are set based on [13], and we'll discuss how they affect the results in the next section.

We implement the baseline QoS-ignorant Publish/Subscribe system [27], in which the Splitter dispatches the messages in a round-robin fashion. The processing of a message is indivisible. The QoS-aware system is implemented using the stream matching framework with smart dispatching and fine-grained scheduling. One core is used by the source to

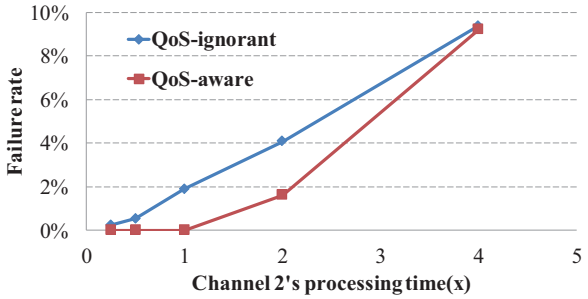

Figure 6: The processing time's effects on the failure rate

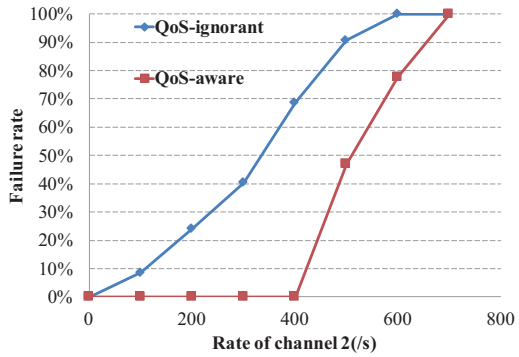

Figure 7: The event period's effects on the failure rate, 1 way

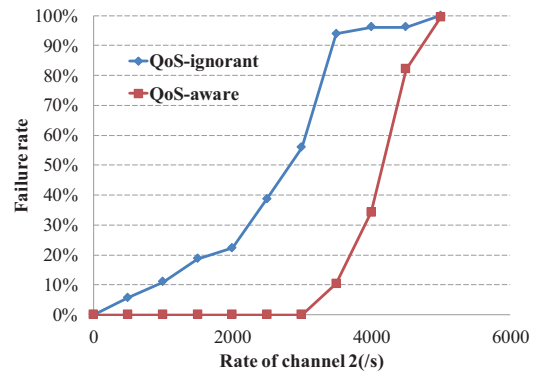

Figure 8: The event period's effects on the failure rate, 7 way

Table 2: Baseline settings of the 2 channels

property of the channel	channel 1	channel 2
subscription number	60,000	600,000
event period(s)	0.00021	0.0025
processing time(s)	0.00007	0.0013
deadline(s)	0.0005	0.006

generate the messages with certain arrive period. Thus, only the other 7 core is available in the following discussion.

We measure how many messages failed in the 20,000 messages and get the corresponding proportional, called failure rate. Two cases are taken as failures:

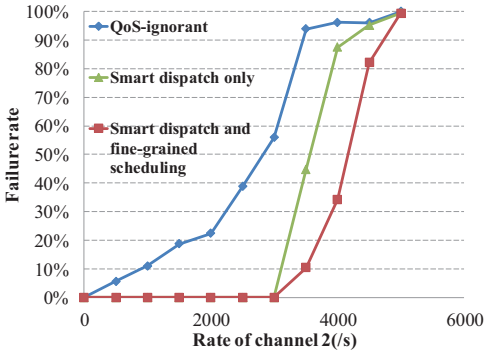

Figure 9: The event period's effects on the failure rate, 7 way

- In the QoS-aware system, the message fail to be dispatched because there are no resources.
- In both systems, the message's deadline is found to be violated after the it finishes.

There are 3 parameters we can explore for a better understanding:

- processing time,
- event period,
- computation resources, i.e. core number.

In the following discussion, we keep channel 1's parameters while changing channel 2's.

6.2 The processing time' effects on the failure rate

We keep the core number as 1. We vary the processing time of channel 2 from $\frac{1}{4}$x to 4x, we observe the failure rate as in Fig. 6.

The QoS-aware system can achieve smaller failure rate because it introduces fine-grained scheduling algorithm. As shown in the motivation case, in the QoS-ignorant system, a message of channel 2 will take up the resource for a long processing time once it starts processing. Thus, the messages of channel 1 ,which arrive at the same time, wait in queue until the message in channel 2 finishes, may result in failure. The number of these messages increase with processing time of channel 2 near linearly. Thus, the failure rate increases near linearly in the QoS-ignorant system.

The QoS-aware system can achieve a smaller failure rate because it allows channel 1 message to interrupt the processing of channel 2 message if it is very urgent. However, when the processing time of channel 2 message increases, the total resource utilization rate increases. When the utilization rate arrived at the upper limit, the failure rates of these two systems become high.

6.3 The event period's effects on the failure rate

Figures 7 and 8 are two different experiments, with 1 and 7 cores respectively. The number of cores is equal to the number of ways. We vary the event period (or event throughput)

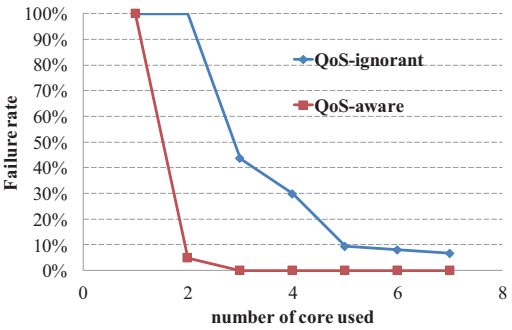

Figure 10: Failure rate decreases with additional resources

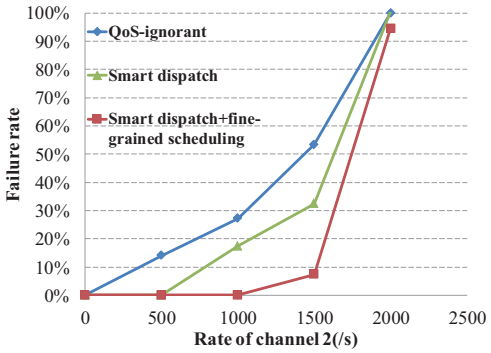

Figure 11: The event period's effects on the failure rate, 7 way

of channel 2. we observe the failure rate as in Fig. 7. When we set the core number to 7, we increase the message rate of channel 1 to 5x to fulfil the computation capacity, that is to say, the event period is set to 0.000042s.

From Fig. 7 and Fig. 8, we observed that, when the rate of channel 2 increases, the failure rate increases. The QoS-aware system can achieve a smaller failure rate than QoS-ignorant system with both 1 core and 7 cores, because it can achieve a higher resource utilization rate.

Smart dispatch and fine-grained scheduling will both affect the fail rate and contribute to the QoS-aware system differently in different settings in Fig. 9 and Fig. 11 when we use 7 cores (since we cannot use smart dispatch when we use only 1 core). In Fig. 11, the processing time of channel 2 is 2x as in Fig. 9. In Fig. 9, the smart dispatch will play an important role when the rate of channel 2 is small. This is because the resource utilization rate is low, the conflict between these two channels can be resolved by additional cores. In Fig. 9, when the rate of channel 2 is higher, the fine-grained scheduling performance an important role in resolving intra-core resource contention by efficiently scheduling. This is the same with Fig. 11, since channel 2 occupies larger portion of resources, the fine-grained scheduling shows its benefit from a smaller rate of channel 2.

6.4 Resources' effects on failure rate

The failure rate decreases with additional resources. In Fig. 10, we set the event period of channel 1 to 0.00007s and channel 2 to 0.001666s to illustrate. The QoS-aware system decreases faster than QoS-ignorant system since it utilizes the resource more efficiently.

However, even additional resources can reduce the failure rate, the QoS-ignorant system cannot reduce the failure rate under 5% with 7 cores due to the channel interference phenomenon that observed in the motivation case.

6.5 Maximum scalability

In previous experiments, we demonstrate that the QoS-ignorant system will suffer from larger failure rate. The reason is that the allocation of computation resources is improper, which leads to contention between two channels. In this experiment, we take another point of view. We control the total failure rate below 1% and watch the maximum throughput of the two systems. Compared to the previous ones, this experiment is more close to the real application case. We measure the maximum factor the throughput can reach while keeping the event period ratio between two channels.

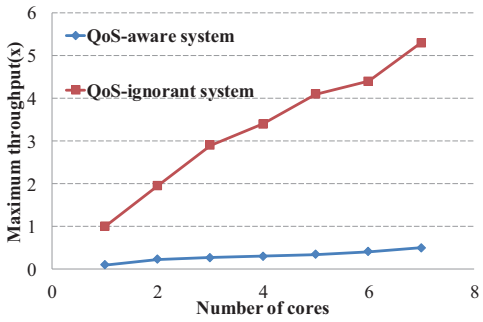

Figure 12: Maximum throughput and scalability

As shown in Fig. 12, the throughput of the QoS-aware system can achieve more than 10x throughput and a near linear scalability. The explanation is that, to achieve the same failure rate, the QoS-ignorant system has to reduce contention of computation resources (which leads to failures) by allowing a lower throughput.

7. CONCLUSION

We propose the first Publish/Subscribe message broker with the ability to actively manage and allocate computation resources to guarantee QoS requirements of messages to address the requirement for QoS in IoT message delivery. The message matching algorithm is abstracted into a task graph in the task-based stream matching framework. We explore the Smart Dispatching algorithm and the Deadline-aware Fine-Grained Scheduling algorithm to guarantee the different QoS requirements for different message channels. We show by experiments that, the QoS-aware system can support more than 10x throughput than the QoS-ignorant systems in representative Smart Grid cases. Also, our system exhibits near-linear scalability on a commodity multi-core machine.

8. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. This work was supported by IBM, National Science and Technology Major Project (2011ZX01035-001-001-002) and National Natural Science Foundation of China (No.60870001, 61028006).

9. REFERENCES

[1] Apache activemq. activemq.apache.org.

[2] Ibm websphere mq. www.ibm.com/software/integration/wmq.

[3] Mosquitto. http://mosquitto.org/.

[4] Northeast blackout of 2003. http://en.wikipedia.org/wiki/Northeast-blackout-of-2003.

[5] Rabbitmq. http://www.rabbitmq.com/blog/.

[6] Wikipedia, the free encyclopedia: Message-oriented middleware. http://en.wikipedia.org/wiki/Message-oriented-middleware.

[7] Zeromq. http://www.zeromq.org/.

[8] M. Adler, Z. Ge, J. Kurose, D. Towsley, and S. Zabele. Channelization problem in large scale data dissemination. In *Network Protocols, 2001. Ninth International Conference on*, pages 100 – 109, nov. 2001.

[9] A. Aggarwal, S. Kunta, and P. Verma. A proposed communications infrastructure for the smart grid. In *Innovative Smart Grid Technologies (ISGT), 2010*, pages 1–5. Ieee, 2010.

[10] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, PODC '99, pages 53–61, New York, NY, USA, 1999. ACM.

[11] G. Ashayer, H. K. Y. Leung, and H.-A. Jacobsen. Predicate matching and subscription matching in publish/subscribe systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, ICDCSW '02, pages 539–548, Washington, DC, USA, 2002. IEEE Computer Society.

[12] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Comput. Netw.*, 54:2787–2805, October 2010.

[13] D. Bakken, A. Bose, C. Hauser, D. Whitehead, and G. Zweigle. Smart generation and transmission with coherent, real-time data. *Proceedings of the IEEE*, 99(6):928–951, 2011.

[14] D. Bakken, R. Schantz, and R. Tucker. Smart grid communications: Qos stovepipes or qos interoperability? *Proc. 2009 Grid-Interop*, 2009.

[15] G. Banavar, T. Chandra, R. Strom, and D. Sturman. A case for message oriented middleware. *Distributed Computing*, pages 846–846, 1999.

[16] L. F. Cabrera, M. B. Jones, and M. Theimer. Herald: Achieving a global event notification service. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, pages 87–, Washington, DC, USA, 2001. IEEE Computer Society.

[17] F. Cao and J. Singh. Efficient event routing in content-based publish-subscribe service networks. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 929 – 940 vol.2, march 2004.

[18] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS)*, 19(3):332–383, 2001.

[19] G. Cugola, E. Di Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *Software Engineering, IEEE Transactions on*, 27(9):827 –850, sep 2001.

[20] E. Curry. Message-oriented middleware. *Middleware for communications*, pages 1–28, 2004.

[21] E. Curry, D. Chambers, and G. Lyons. Extending message-oriented middleware using interception. In *Third International Workshop on Distributed Event-Based Systems*, page 32. Citeseer, 2004.

[22] Y. Diao, P. Fischer, M. Franklin, and R. To. Yfilter: Efficient and scalable filtering of xml documents. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 341–342. IEEE, 2002.

[23] R. E. Duran, L. Zhang, and T. Hayhurst. Enabling gpu acceleration with messaging middleware. *Informatics Engineering and Information Science*, pages 410–423, 2011.

[24] F. El-Hassan and D. Ionescu. A hardware architecture of an xml/xpath broker for content-based publish/subscribe systems. In *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, pages 138–143. IEEE, 2010.

[25] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[26] F. Fabret, H. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *ACM SIGMOD Record*, volume 30, pages 115–126. ACM, 2001.

[27] A. Farroukh, E. Ferzli, N. Tajuddin, and H. Jacobsen. Parallel event processing for content-based publish/subscribe systems. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, page 8. ACM, 2009.

[28] C.-L. Fok, C. Julien, G.-C. Roman, and C. Lu. Challenges of satisfying multiple stakeholders: quality of service in the internet of things. In *Proceedings of the 2nd Workshop on Software Engineering for Sensor Network Applications*, SESENA '11, pages 55–60, New York, NY, USA, 2011. ACM.

[29] J. Gough and G. Smith. Efficient recognition of events in a distributed system. *Australian computer science communications*, 17:173–179, 1995.

[30] J. Gough and G. Smith. Efficient recognition of events in a distributed system. *Australian computer science communications*, 17:173–179, 1995.

[31] E. N. Hanson. Rule condition testing and action execution in ariel. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, SIGMOD '92, pages 49–58, New York, NY, USA, 1992. ACM.

[32] Y. Huang and H. Garcia-Molina. Publish/subscribe in a mobile environment. *Wirel. Netw.*, 10:643–652, November 2004.

[33] Y. Jeon. Qos requirements for the smart grid communications system. *International Journal of*

Computer Science and Network Security, 11(3):86–94, 2011.

[34] G. T. Lakshmanan, Y. G. Rabinovich, and O. Etzion. A stratified approach for supporting high throughput event processing applications. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 5:1–5:12, New York, NY, USA, 2009. ACM.

[35] G. Li, S. Hou, and H.-A. Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*, pages 447 –457, june 2005.

[36] G. Li, S. Hou, and H.-A. Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*, pages 447 –457, june 2005.

[37] C. Lin. High speed publication subscription brokering through highly parallel processing on field programmable gate array (fpga). Technical report, DTIC Document, 2010.

[38] R. Linderman. Fpga acceleration of information management services. Technical report, DTIC Document, 2005.

[39] A. Margara and G. Cugola. High performance content-based matching using gpus. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, pages 183–194. ACM, 2011.

[40] A. Mitra and W. Adviser-Najjar. *Acceleration of streaming applications on FPGAs from high level constructs*. University of California, Riverside, 2008.

[41] A. Mitra, M. Vieira, P. Bakalov, W. Najjar, and V. Tsotras. Boosting xml filtering with a scalable fpga-based architecture. *Arxiv preprint arXiv:0909.1781*, 2009.

[42] K. Moslehi and R. Kumar. A reliability perspective of the smart grid. *Smart Grid, IEEE Transactions on*, 1(1):57–64, 2010.

[43] R. Moussalli, R. Halstead, M. Salloum, W. Najjar, and V. Tsotras. Efficient xml path filtering using gpus. 2011.

[44] R. Moussalli, M. Salloum, W. Najjar, and V. Tsotras. Accelerating xml query matching through custom stack generation on fpgas. *High Performance Embedded Architectures and Compilers*, pages 141–155, 2010.

[45] R. Moussalli, M. Salloum, W. Najjar, and V. Tsotras. Massively parallel xml twig filtering using dynamic programming on fpgas. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 948–959. IEEE, 2011.

[46] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. Exploiting ip multicast in content-based publish-subscribe systems. In *IFIP/ACM International Conference on Distributed systems platforms*, Middleware '00, pages 185–207, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[47] Oracle. Complex event processing in the real world. *Oracle Whitepaper*, 2007.

[48] J. a. Pereira, F. Fabret, F. Llirbat, R. Preotiuc-Pietro, K. A. Ross, and D. Shasha. Publish/subscribe on the web at extreme speed. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 627–630, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[49] J. a. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient matching for web-based publish/subscribe systems. In *Proceedings of the 7th International Conference on Cooperative Information Systems*, CooplS '02, pages 162–173, London, UK, 2000. Springer-Verlag.

[50] Y. Qi, K. Wang, J. Fong, Y. Xue, J. Li, W. Jiang, and V. Prasanna. Feacan: Front-end acceleration for content-aware network processing. In *INFOCOM, 2011 Proceedings IEEE*, pages 2114 –2122, april 2011.

[51] A. Riabov, Z. Liu, J. L. Wolf, P. S. Yu, and L. Zhang. Clustering algorithms for content-based publication-subscription systems. In *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, ICDCS '02, pages 133–, Washington, DC, USA, 2002. IEEE Computer Society.

[52] V. Ross. Heterogeneous high performance computer. In *Users Group Conference, 2005*, pages 304–307. IEEE, 2005.

[53] M. Sadoghi and H. Jacobsen. Relevance matters: Capitalizing on less. 2012.

[54] M. Sadoghi and H.-A. Jacobsen. Be-tree: an index structure to efficiently match boolean expressions over high-dimensional discrete space. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 637–648, New York, NY, USA, 2011. ACM.

[55] M. Sadoghi, H. Singh, and H. Jacobsen. fpga-topss: line-speed event processing on fpgas. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, pages 373–374. ACM, 2011.

[56] R. Shah, Z. Ramzan, R. Jain, R. Dendukuri, and F. Anjum. Efficient dissemination of personalized information using content-based multicast. *IEEE Transactions on Mobile Computing*, 3:394–408, October 2004.

[57] A. Silberschatz, P. Galvin, G. Gagne, and A. Silberschatz. *Operating system concepts*, volume 4. Addison-Wesley, 1998.

[58] V. Sood, D. Fischer, J. Eklund, and T. Brown. Developing a communication infrastructure for the smart grid. In *Electrical Power & Energy Conference (EPEC), 2009 IEEE*, pages 1–7. Ieee, 2009.

[59] T. Wong, R. Katz, and S. McCanne. An evaluation of preference clustering in large-scale multicast applications. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 451 –460 vol.2, 2000.

[60] T. W. Yan and H. Garcia-Molina. The sift information dissemination system. *ACM Trans. Database Syst.*, 24:529–565, December 1999.