Optimal Load Balancing in Publish/Subscribe Broker Networks

Hui Zhang
Samrat Ganguly
Sudeept Bhatnagar
Rauf Izmailov
NEC Laboratories America
Email: {huizhang,samrat,bhatnagar,rauf}@nec-labs.com

Abstract—Load balancing in publish/subscribe (pub/sub) broker networks is challenging as the workload is multi-dimensional and content-dependent. In this paper we present the framework design of a middleware, called Shuffle, to achieve optimal load balancing in a pub/sub broker network. Shuffle features a suite of active workload management schemes within a single overlay topology on message parsing, matching, delivery, and forwarding, the four types of workload in a publish/subscribe service affected by two inputs - streaming events and stored subscriptions. Shuffle leverages its traffic randomization scheme and Chord, a DHT substrate, to build overlay trees for active workload aggregation and distribution, and we show the optimality property of the load balancing scheme upon any input traffic distribution on individual Shuffle aggregation trees. We also show the NP-hardness of the workload management problem when it has to be done among multiple correlated aggregation trees, and present a heuristic accordingly. Through extensive simulations we validated the design of Shuffle upon dynamic and heavy workload.

I. INTRODUCTION

Publish/Subscribe (pub-sub) model [1], [2] enables publishers to selectively disseminate information to interested subscribers. A broker network acts as an intermediary between the publishers and the subscribers. Subscribers specify their interest in terms of topic and/or predicate-based filters (called subscriptions) to the broker network. Publishers push information (called events) to the broker network without any explicit knowledge of the subscriber interest. The broker network matches the published information against subscribers’ interest and delivers it to the appropriate subscribers.

Load balancing in a broker network is challenging as the workload is hard to predict with its nature of being content dependent (i.e., decided by the real-time result of matching the published information against subscribers’ interest). Also, the workload is multi-dimensional as the potential workload can be classified into the following types: message parsing: this involves the workload of parsing the original message texts to subtract the information used for matching, and store them in the specific data structures dependent on the in-memory matching algorithm implemented for the service; message matching: this involves the workload of in-memory matching between events and subscriptions in individual nodes; message delivery: this involves the workload of notifying the interested subscribers of the matched events after the message matching procedure; message forwarding: this involves the workload of overlay message forwarding following the routing protocol in the broker network.

In this paper, we present Shuffle, the framework design of a workload management middleware. In Shuffle, load balancing issues are addressed on all four types of workloads mentioned above. Shuffle actively aggregates/distributes events and subscriptions among broker servers to avoid performance bottlenecks in terms of message parsing, and offers a simple and fast mechanism to achieve optimal load balancing on message matching and delivery with little overhead. While the active workload management incurs message forwarding overhead, the overhead is evenly distributed among all servers and actually decreases when the load balancing mechanism is applied upon high event arrival rates. The load balancing performance is insensitive to the data distribution of input requests, and optimal without introducing extra maintenance cost on the overlay topology.

The rest of the paper is organized as the following. Section II presents the background information on content-based pub/sub model and the Chord protocol. The architecture design of the Shuffle system is described in Section III and the analysis results are given in Section IV. Section V presents the simulation results. Section VI describes the related work and Section VII concludes the paper with future work.

II. BACKGROUND INFORMATION

A. Pub/Sub Model

There are two common forms of pub/sub models: topics-based and content based. In topic-based pub/sub model, events are grouped into topics by the publishers. Subscribers will receive all events published to the topics to which they subscribe, and all subscribers to a topic will receive the same messages. In content-based pub/sub model, a multi-dimensional data space is defined on $d$ attributes. An event $e$ can be represented as a set of $<a_i, v_i>$ data tuples where $v_i$ is the value this event specifies for the attribute $a_i$. A subscription can be represented as a filter $f$ that is a conjunction of $k$ $(k \leq d)$ predicates, and each predicate specifies a constraint on a different attribute, such as "$a_i = X$", or "$X \leq a_i \leq Y$". In the rest of this paper, we focus on content-based pub/sub model as it is more general than topic-based model.
B. Chord

Chord assigns each overlay node in the network a random \( m \)-bit identifier (called the node ID). Similarly, each key is also assigned an \( m \)-bit identifier (following [3], we use key and identifier interchangeably). Chord uses consistent hashing to assign keys to nodes. Each key is assigned to that node in the overlay whose node ID is equal to the key identifier, or follows it in the key space (the circle of numbers from 0 to \( 2^m - 1 \)). That node is called the successor of the key. The Chord maintains at each node a finger table having at most \( m \) entries. The \( i \)-th entry in the table for a node whose ID is \( n \) contains the pointer to the first node, \( s \), that succeeds \( n \) by at least \( 2^{i-1} \) on the ring, where \( 1 \leq i \leq m \). Node \( s \) is called the \( i \)-th finger of node \( n \).

Suppose node \( n \) wishes to lookup the node assigned to a key \( k \) (i.e., the successor node \( x \) of \( k \)). To do this, node \( n \) searches its finger table for that node \( j \) whose ID immediately precedes \( k \), and passes the lookup request to \( j \). \( j \) then recursively repeats the same operation. At the end of this sequence, \( x \)'s predecessor returns \( x \)'s identity (its IP address) to \( n \), completing the lookup.

III. SHUFFLE DESIGN

A. Architecture Overview

Shuffle supports a generic middleware framework independent of application-specific pub/sub matching algorithms. It is designed atop a set of dedicated servers residing within a LAN as broker nodes. The network usually starts with a small size (e.g., in tens) and may grow to thousands of nodes with the increasing popularity of the services.

B. Overlay Construction

Shuffle uses the Chord protocol to map both nodes and pub/sub messages onto an unified one-dimensional key space. Each node is assigned an ID and a portion (called its range) of the key space, and maintains a Chord finger table for overlay routing. Both event and subscription messages are hashed onto the key space based on one of the attributes contained in the messages, and therefore are aggregated and filtered in a coarse degree (attribute level). For each attribute in the pub/sub services, the routing paths from all nodes to the node responsible for that attribute (called the root node in the rest of the paper) naturally forms an aggregation tree, on which Shuffle manages the workload associated with that attribute.

In a Shuffle network with \( n = 2^k \) nodes and equal partition of the overlay space among the nodes, there is a specific load distribution in an aggregation tree when all nodes generate the same amount of messages to the root node. As shown in Figure 1, let us assume that every node generates one message which is then routed to the root node and some non-leaf node \( S \) has aggregated a total of \( T(S) \) unit messages in the overlay forwarding. We can show (see Section IV for the details) that \( S \) has \( \log(T(S)) \) immediate children in the tree, and the messages forwarded by the children follows the half-cascading distribution: one child forwards \( \frac{1}{2} \) of the total messages \( T(S) \), one child forwards \( \frac{1}{3} \) of \( T(S) \), and so on, until one child, which is a leaf node, contributes one message.

When the key space is not equally partitioned among the nodes, the aggregation load will not follow the exact half-cascading distribution. It is well known that the original node join scheme in Chord can cause \( O(\log n) \) stretch in key space partition. In Shuffle system, we propose a node joining/leaving scheme for equal key space partition:

Optimal Splitting (OS)

- **Joining**: a new node joins the network by finding the Shuffle node owning the longest key range and taking half of the key range from that node. When there are multiple candidate nodes with the longest key ranges, the tie is broken with a random choice.
- **Leaving**: the leaving node \( x \) finds the Shuffle node owning the shortest key range and asks it to leave its current position and rejoin at \( x \)'s position. When there are multiple candidate nodes with the shortest key ranges, \( x \)'s immediate successor node is given the priority if it is one such candidate; otherwise, the tie is broken with a random choice.

C. Message Shuffling

As a proxy for a pub/sub overlay service, the first assignment of a Shuffle node \( x \) at receiving an original subscription message \( m \) is to re-distribute it in the system. \( x \) will pick a random key for \( m \) (e.g., by hashing some subscription ID contained in the message) and send it to the node \( y \) responsible for that key in the overlay space. \( y \) is responsible for parsing \( m \), constructing a new message \( \tilde{m} \) tailored for fast overlay routing and the in-memory message matching algorithm associated with the service, and sending \( \tilde{m} \) to the destination node \( z \), which \( y \) decides by arbitrarily picking an attribute \( A \) specified in \( m \) and hashing \( A \) onto the overlay space \(^1\). We call \( z \) the root of the aggregation tree associated with the attribute \( A \), and \( y \) generates one message for the aggregation tree by sending \( \tilde{m} \) to \( z \) through the overlay routing. For an event message \( m \), each attribute specified in

1For unsubscribe consideration, the random keys for the paired subscribe/unsubscribe messages should be the same, and the choice of the attribute \( A \) has to be made consistently even if \( y \) is later replaced by another node for the key range containing the random key.
it will be used to decide a destination node to send one copy of \( m \).

The above randomization procedure (called message shuffling) achieves two goals:

- The randomization makes the distribution of the input traffic for any potential aggregation tree uniform on the key space. Combing message shuffling and Chord with OS scheme, Shuffle can construct half-cascading aggregation trees.
- The cost of message parsing on subscriptions is distributed evenly throughout the system so that Shuffle eliminates the potential performance bottleneck due to message parsing workload.

D. Optimal Load Balancing in A Single Aggregation Tree

With the half-cascading distribution, a simple pushing mechanism can be used to achieve optimal load balancing with little overhead. In a Shuffle aggregation tree, initially only the root node processes the aggregated messages and all other nodes just forward messages. We say the root node is in active state and the rest of the nodes are in hibernating state. When the processing workload is excessive, the root node re-balances the workload by activating the child forwarding half of the workload and pushing that part of the workload back to that child for processing. This equals to splitting the original aggregation tree into two with each node acting as one root. The pushing operation may happen recursively until each activated node can accommodate its assigned processing workload. With the half-cascading distribution, an active node can reduce half of its workload after each pushing operation. Therefore, the maximal number of pushing operations each node incurs is \( \log n \), where \( n \) is the network size. If the workload after \( \log n \) operations is still too high for a node, we can claim all other nodes are also overloaded. This is because the current workload this node has is due to the messages itself generates and all nodes generate the same amount of messages (workload) \(^2\).

There are two types of messages in Shuffle: subscriptions and events. Accordingly, the load balancing on an aggregation tree is classified into two kinds.

1) Subscription Balancing: In Shuffle, a subscription message \( n \) will be replicated at each hop on the routing path from the message sender \( y \) to the root node \( z \). In addition, every node in an aggregation tree will count the forwarding fraction of the subscriptions that each of its children contributes in message forwarding. When \( z \) is overloaded with too many subscriptions from an aggregation tree \( A \), it applies the replicating-holding scheme for load balancing:

i) \( z \) ranks all the children in this tree based on their forwarding fraction, and marks them as “hibernating” initially.

ii) Among all “hibernating” children, \( z \) picks the node \( l \) with the largest fraction. \( z \) will unload all subscriptions forwarded from \( l \) by sending the corresponding “unsubscribe” messages into \( z \)'s local black box, and asks \( l \) to load the cached copies from its local database into its local box.

iii) \( z \) marks \( l \) as “active”, and forwards all event messages in the aggregation tree of the same attribute \( A \) to \( l \) in the future.

iv) \( z \) repeats Step 2 if it needs to unload more subscriptions locally.

In Shuffle, the loading-forwarding scheme is applied recursively on each active node for load balancing on subscriptions.

2) Event Balancing: In a Shuffle aggregation tree, each node also counts the forwarding fraction of the events that each of its children contributes in message forwarding. When \( z \) is overloaded with too many events from an aggregation tree \( A \), it applies the replicating-holding scheme for load balancing:

i) \( z \) ranks all the children in this tree based on their forwarding fraction, and marks them as “hibernating” initially.

ii) Among all “hibernating” children, \( z \) picks the node \( l \) with the largest fraction. \( z \) will replicate all subscription messages from the same aggregation tree at node \( l \), and ask \( l \) to hold event forwarding and instead process all the events it receive at its local box in the future.

iii) \( z \) marks \( l \) as “active”, and goes to Step 2 if it needs to shed more arriving events.

In Shuffle, the replicating-holding scheme is applied recursively on each active node for load balancing on events.

3) Load Balancing Scheduling Scheme: We propose a simple scheme for scheduling the load balancing operations on events and subscriptions at an overloaded node \( n \):

- Node \( n \) determines its target event processing rate \( r \). Using the replication-holding scheme, node \( n \) stops as many of its children as required from forwarding events to itself so that its event arrival rate is at most \( r \). If node \( n \) stops its child \( n_1 \) from forwarding events, then the aggregation tree rooted at \( n \) is split into two disjoint trees rooted at \( n \) and \( n_1 \), respectively.

- Based on \( r \), \( n \) determines the upper bound on the subscriptions it should manage, (say) \( th \). It then uses the loading-forwarding scheme to off-load subscriptions to its children so that the subscription load is at most \( th \).

E. Load Balancing over Multiple Aggregation Trees

If the overloaded node \( n \) is part of multiple aggregation trees, it needs to decide along which aggregation tree should it off-load work(events/subscriptions) to its children.

Consider a pub/sub broker network with \( N \) nodes. Suppose node \( n \) is part of \( k \) aggregation trees. Let \( X^n_i \) be the total number of subscriptions that \( n \) is currently managing. Since \( n \) is part of \( k \) aggregation trees, \( X^n_i = X^n_1 + X^n_2 + ... + X^n_k \) where \( X^n_i \) is the number of subscriptions for attribute \( i \) currently being managed by \( n \). Let \( X_{\text{total}} \) and \( X_i \) be the total
number of subscriptions and the total number of subscriptions for attribute \( i \) managed by the pub/sub system, respectively, i.e. \( X_{i,\text{total}} = X_1 + X_2 + \ldots + X_k \). A threshold \( th = r \times \left( \frac{X_{i,\text{total}}}{N} \right) \) can be written as \( th = th_1 + th_2 + \ldots + th_k \) where \( th_i = r \times \left( \frac{X_i}{N} \right) \). Hence, the following always holds true in the case of multiple aggregation trees:

- If \( X_{i} > th \) then \( \exists \) at least one attribute \( i \in X_{i} > th_i \).

The above fact motivates the following heuristic, henceforth referred to as Multiple Tree heuristic for choosing an aggregation tree:

1) If the total number of subscriptions at node \( n \), \( X_{i}^n \) exceeds the threshold \( th \), it picks an attribute \( i \in X_{i}^n > th_i \) and off-load subscriptions along the aggregation tree for attribute \( i \).

When there are multiple attributes \( i \in X_{i}^n > th_i \), node \( n \) can either choose an attribute randomly or pick an attribute \( j \) for which \( (X_{i}^n - th_j) \) is the highest.

2) Repeat (1) until \( X_{i}^n \leq th \).

An overloaded node that needs to use the replication-holding scheme can use an analogous heuristic for picking an aggregation tree.

IV. Analysis

In this section we present the analytic results, and the proofs are available in the companion technical report [4].

**Power-of-2 network size:** When the Shuffle network size is a power of 2, we can show that in any aggregation tree, there is at least one child which contributes no less than \( \frac{1}{4} \) of the total load aggregated on \( x \).

Therefore, a node can shift at least \( \frac{1}{4} \) of its workload upon one single pushing operation. The upper bound of the maximal aggregation load fraction can be arbitrarily close to 1, but this will happen on either a root node or a leaf node in any aggregation tree. Therefore Shuffle load balancing still works effectively on most of the nodes in an aggregation tree.

**Hardness of Multiple Tree Load Balancing:** The problem of optimal load balancing over multiple trees can be formulated as follows: For a network of size \( N \), given \( k \) attribute trees, the number of subscriptions \( X_i \) at the root of each attribute tree \( i \) and threshold \( th \), what is the minimum number of nodes in the network to which subscriptions must be transferred to such that the number of subscriptions at any node is at most \( th? \)

We call this problem MIN-NODE-LOAD-FORWARD, and show that it is NP-hard by reduction from the MIN-COST-FIXED-FLOW problem on directed graphs with a certain structure, which is a known NP-hard problem [5].

**Theorem 3:** MIN-NODE-LOAD-FORWARD is NP-hard.

V. Evaluation

A. Methodology

We implemented the Shuffle simulator with the underlying DHT being a basic version of Chord, by extending the p2psim simulator [6].

For comparison, we consider two other possible load balancing schemes:

- **Random-Half:** In this scheme, an overloaded node picks an underloaded node with random probing, and then splits half its load with that node. The overloaded node repeats the operation until its load is reduced below a target level. This scheme is similar to the DHT load balancing scheme proposed in [7].

- **Random-Min:** Random-Min is the same as Random-Half except when an overloaded node splits its load with an underloaded node, it just delegates a bare minimum load equal to the target value to the chosen node by replicating its subscription set there and forwarding a commensurate fraction of event traffic there.

B. Results

1) **Single Tree Load Balancing:** In this set of experiments, we consider the load being incurred due to a high event rate so that the root node uses the replication-holding scheme. Each node generates the same number of events to the root node. We set the target load level as multiples of the total events generated by each node. A lower target load level implies that a large number of nodes need to active to be able to manage the work load. We present results for control messages generated.

![Control Traffic (in messages)](image)

Fig. 2. Control message overhead: 768-node network size.

For comparison, we consider two other possible load balancing schemes:

- **Random-Half:** In this scheme, an overloaded node picks an underloaded node with random probing, and then splits half its load with that node. The overloaded node repeats the operation until its load is reduced below a target level. This scheme is similar to the DHT load balancing scheme proposed in [7].

- **Random-Min:** Random-Min is the same as Random-Half except when an overloaded node splits its load with an underloaded node, it just delegates a bare minimum load equal to the target value to the chosen node by replicating its subscription set there and forwarding a commensurate fraction of event traffic there.
the aggregation tree to locate the underloaded children of an overloaded node.

2) Multiple Tree Load Balancing: In this set of experiments, the load balancing goal is to redistribute work load so that the number of subscriptions managed by any node in the network is at most the target load level. The metric of interest is nodes affected. A node is affected during subscriptions load balancing if some overloaded node off-loads subscriptions to it. The network size is 1024 nodes with 512 aggregation trees. Each node generates the same number of subscriptions. A subscription is mapped to an aggregation tree using a zipf distribution. We set the target load level as multiples of the total subscriptions generated by each node.

Using a DHT substrate for workload management in distributed systems has been presented in the literature [12], [13], [14], [15]. Also, many load balancing schemes for DHT systems have been proposed [7], [16]. Shuffle is distinguished from the previous work with its active workload management schemes: its optimal performance comes without introducing extra overlay topology maintenance cost; its load balancing operation is sensitive to the data distribution of input requests; it offers an integral load balancing solution to four types of workload, while the previous work usually addresses only one or at most two (e.g., message delivering in overlay multicasting).

VII. CONCLUSIONS

We present the design of Shuffle, an active workload management middleware to support a scalable broker network. Shuffle offers an integral solution to manage all types of the workload in a pub/sub broker network within a single overlay topology.

VI. RELATED WORK

Existing work on the design of a scalable broker network targets two objectives: a) how to minimize the subscription states maintained at each node in order to perform fast in-memory subscription matching [8] and b) how to search the subscription states efficiently in order to reduce the number of messages [9]. Recently, there has been significant effort in searching multi-dimensional subscription profiles using DHT [10], [11]. They are different from our work in that their design goal is to maximize the in-networking filtering effectiveness through overlay routing in a single pub/sub service, while our design goal is to utilize overlay routing for optimal workload management in a general pub/sub model to support diverse pub/sub services.

Fig. 3. Node affected: 1024 nodes, OS, Zipf Dist.

As shown in Fig 3, the Shuffle system affects up to 20% (70%) more nodes in compared to Random-Half (Random-Min) scheme for target load level \( r = 2.5 \). For \( r \geq 4 \), its performance is comparable to the Random-Half scheme. The reason for higher number of nodes being affected in the Shuffle system is that the Multiple Tree heuristic does loading-forwarding separately for each aggregation tree. Contrast this with the situation in case of Random-Min and Random-Half where a node sheds the workload freely without differentiating between the trees. While per-tree load balancing affects a higher number of nodes, the trade-off is the simplified overlay management in Shuffle with the existence of only one overlay - the original Chord topology. Random-Half and Random-Min create a separate overlay topology (the trees resulting from load balancing) for each overloaded node, thus increasing the network management complexity significantly.

REFERENCES