Analysis of a Matching Algorithm in a Content-Based Publish/Subscribe Peer-to-Peer Application
Brian Pine

ABSTRACT
The publish/subscribe paradigm is a very beneficial message passing system in a widespread network such as the Internet. Producers and consumers of information are de-coupled in time, space, and synchronization. If subscribers are allowed to formulate subscriptions based on the content of the messages that will be published, the publish/subscribe system must efficiently match up messages to subscriptions, while at the same time being scalable. It has been shown that peer-to-peer networks scale very well, so they can be used underneath a publish/subscribe system to route publications over a network. A matching algorithm will be tested under conditions where the load of subscriptions on network nodes range across allowed values determined by the properties of the underlying peer-to-peer network.

Keywords
Publish/Subscribe, Content-Based Routing, Peer-to-Peer Networks, Overlay Networks

1. INTRODUCTION
The Internet has a great need for distributing information efficiently from user to user, from people that generate that information to those who will use it, from one computer to another. One way of looking at this is that some users produce new events1 (publish), while others register that they have an interest in those events (subscribe).

There are several problems that are encountered when trying to disseminate information over a widespread network. How many pathways are there through the network (in other words, if there is a problem along the path, is there another way for the event to get through)? How do you efficiently match an event to the subscribers that have registered interest in that type of event? What language do you use to write subscriptions with, and how expressive is it? Some answers to these questions are explored in this paper.

The registration process in a publish/subscribe system can be taken care of by a piece of middleware (known as a broker). Middleware provides the benefit of de-coupling publishers from subscribers. Since publishers don't need to have a direct channel of communication with subscribers, they do not need to know what subscriber, or how many, is interested in their events. This could be made a part of the system if the situation calls for it, but it is not necessary. Publishers do not have to be physically located close to their subscribers, know how to contact them, or even know who they are. Also, subscribers can receive messages about events at a time that is later than when the event was published. As a result, publishers and subscribers are de-coupled in synchronization, space, and time [1].

The middle-ware can be a single node that contains all of the information about where a published event should be routed to, or the routing information can be distributed throughout a network. If the publish/subscribe system is being used in a very large network like the Internet, then using the single node approach becomes a bottleneck for the entire system, as every single event needs to pass through this node. An administrator would be required to handle all routing decisions, and keep track of nodes that leave or enter the network. This could lead to an extremely large routing table. Also, if that node fails, then the whole system ceases to work. All these reasons mean that a centralized routing system is not very scalable. Scalability is something that is very important to a system where the number of users can be very large; the Internet is a prime example.

Instead of having just one broker in charge of the entire publish/subscribe system, the workload can be distributed to a wider number of nodes. This helps relieve many of the problems that plague systems where only one network node controls the entire system.

In order for subscribers to register interest in potential messages, they must create a subscription. This subscription is phrased in a subscription query language, that is somewhat like a database query language. Expressiveness of the language is important, but this is sometimes at odds with achieving scalability of the overall system. The less expressive a language is, more messages will have to be discarded by individual subscribers rather than have them filtered out for them by their subscriptions. Siena is one system that attempts to maximize expressiveness without giving up on scalability [10]. Two types of languages are topic-based and content-based subscription languages.

In a content-based system, messages generated by publishers will be in a specific format. This format is a schema, similar to a schema in database theory. Data items to be published are associated with an attribute in the schema, where all values of a particular schema attribute are all of a given type (such as string, integer, double, etc.). So, by this definition, a message is an assignment of an appropriately typed value to some (not necessarily all) of the names in the schema. If the message assigns a value to each attribute of the schema, then it can be referred to as a total message. [8].

A generated event has to be passed along (routed) across a network to the proper subscribers, and there are several different ways of doing this. One way is called flooding, where every node that receives a message passes that message on to every other node that it is directly connected to and aware of (neighbors). This can cause a lot of network traffic as well as a message possibly being received more than once by a particular subscriber.

Another way to move messages over a network is called content-based routing. Each broker in the network becomes

---

1The words publication, event, and notification are all used to describe the same thing.
responsible for a portion of the routing job. This means that routing tables for a node are used to determine where messages will be passed to other nodes in the network.

A network can be viewed as a graph where each broker is a node, and the ways that data can be sent from one broker to another become edges in the graph. Some research has looked at only acyclic graph network representations [3]. However, these systems are prone to some of the problems of a centralized middleware system. If any node in the network breaks down, some messages will not be routed to subscribers that would normally have received that event. Also, most real-world networks are cyclic.

Peer-to-peer networks can help solve the problems with using a distributed, cyclic network. In a peer-to-peer network, no nodes are more important than any other. There is no central authority in charge of routing decisions, or where subscriptions get stored. Routing table sizes do not grow linearly with the number of nodes in the network. If a node fails, then events can still be delivered to where they belong. Peer-to-peer networks are very good at handling situations where nodes are continuously entering and leaving a network. All in all, peer-to-peer networks provide a very good base for scalable, efficient situations. A few examples of peer-to-peer systems are Chord and Pastry [11][5].

Several different publish/subscribe systems have been designed or implemented. Some examples are Rebeca [3], and Siena[10].

This paper answers some questions about a particular content-based publish-subscribe system that is designed over a peer-to-peer network [4]. A matching algorithm is tested for efficiency over the allowable range of loads for a particular node. This range is bounded by the use of a particular peer-to-peer base (Chord) [5]. What happens to the efficiency of the matching algorithm as the values range across this bound?

2. FOUNDATION

2.1 Distributed Hash Tables

One way to decrease the data strain on any one broker in a system is to use Distributed Hash Tables, or DHTs [9]. Distributed Hash Tables use a type of method called consistent hashing that allows them to distribute keys to nodes in a randomized fashion. However, consistent hashing is different from regular hashing since it guarantees that the keys will be distributed almost equally across the node space. In fact, given N nodes and K keys, a consistent hashing function will make each node contain at most $((1 + \varepsilon) *K / N)$ keys, where $e$ is at most O(log N) [5]. This property can keep a particular node from being a bottleneck, or from losing a disproportionately large portion of stored information if there is a node failure.

2.2 Chord

Chord is a peer-to-peer network lookup service, providing a way to find information efficiently in a distributed network environment [5]. In Chord, nodes are placed in a ring structure, and each node is given an identifier. Chord is based on Distributed Hash Tables, where nodes IP addresses are hashed to come up with an identifier for the node in the range from 0 to $2^m - 1$, where $m$ is the number of bits used in the identifier. Values that want to be stored in the network are also hashed, and placed at the node whose identifier equals the hashindex of the value. If there is no node corresponding to that identifier, then Chord finds that identifier’s successor, which is the identifier of the next clockwise node in the Chord ring.

By using Distributed Hash Tables, Chord can make several guarantees about the network in general and information lookup in particular. First, nodes contain location information of at most O(log N) neighboring nodes [5]. This is the case when no nodes are joining or leaving the system (steady-state). However, Chord is able to maintain this property even in the face of nodes continuously joining or leaving. Given that each node contains information about O(log N) other nodes, Chord allows information to be looked up by being routed through O(log N) nodes. Since Chord uses a consistent hashing function, each node will contain at most $(1 + \varepsilon) *K / N$ keys, as specified above.

3. Query Languages

A subscription language must be chosen to allow users to formulate a rule to describe what events they are interested in receiving. Several types of languages are available, including topic-based and content-based subscription languages. Subscribers in a topic-based subscription environment register their interest in a particular topic, and are alerted when events on that topic are published. Publishers have to determine what topic their event falls under; they may decide that their event satisfies more than topic, or that there is no available topic that matches their event. Subscribers face a similar problem: they may have to subscribe to more than one topic if what they're interested in doesn't have a topic category all to itself. They then have to locally filter out events that they're not interested in rather than completely rely on the middleware for event filtering. Topic-based subscription languages can be restrictive because they don't allow subscribers to specify filtering criterion on information that is present inside of the event.

Content-based subscription languages give the subscriber the ability to state interest in every portion of a message. As long as the messaging schema is known and followed by both publishers and subscribers, subscribers can specify interest in any (or all) of the attributes of the schema. This is done by listing an attribute’s name, followed by a predicate on the value. Different subscription languages allow different sets of operators for the predicate, but a common one is ($=, !=, <, \leq, >, \geq$). If range maximums, minimums, and precision fields of an attribute are given in the schema, then predicates that use a range are allowed for use in the system that is being tested in this paper. Also, content-based subscription doesn't require a list of pre-defined topics that publishers and subscribers are locked into using.

While content-based subscription languages allow subscribers flexibility in choosing what publications they receive, some systems allow the publisher similar flexibility. These systems are known as symmetric subscription services, one of which is the Wbsphere Matchmaking Environment [6]. This system allows a publisher to vary the content of a published event dynamically, depending on who receives that event. An example given in [6] is an insurance company sending out their latest quotes for car insurance. The same event can be received by different subscribers, but if certain criteria are met, such as the subscriber being at least a certain age or having no accidents in the past few years, then the quoted insurance price is lower than another subscriber that doesn't match those criterion. This
requires a subscriber to contain information about itself in a subscription that may not be directly used to match that subscription to potential publications.

There are several different subscription query languages that are available. Each language can have a different level of expressiveness. In general, the more expressive the language is, the harder the matching algorithm has to work to find matches for a publication. The Java Message Service (JMS) API contains a language that can be used as a query language [7]. Three other languages are the Simple Subscription Language (SiSL), the Strict Subscription Language (StSL), and the Default Subscription Language (DeSL) [8].

Each of these languages determines what types of messages are accepted by a given subscription. The Simple Subscription Language requires that all messages be total, that is, each message has to assign a value to every single attribute in the schema. The Strict Subscription Language relaxes the total message restriction, however, a message still needs to assign a value to every attribute that appears in a subscription in order to be a match. Default Subscription Languages will assign a default value to every attribute that is not specified in a given message[8].

The simulation in this paper uses a strict subscription query language.

4. SYSTEM TO BE TESTED

The system that will be tested in this paper content-based publish/subscribe system that is built on top of a Chord [4]. Chord allows assumptions to be made about the nature of the lookups for distributing subscriptions, and finding matches when notifications are created. If there are N nodes, each node in the system contains information about only O(log N) neighbors, so routing tables will only grow logarithmically with the number of nodes [5].

A subscriber sends a subscription to a neighboring broker node. This node stores the IP address of the subscriber, along with their subscription. Then, the value of every attribute in a subscription gets hashed separately, and a small piece of information called a subscriptionID gets stored at the node whose identifier equals the hashed value. It is stored in a list that is specific to that attribute. In other words, there is a separate list for each attribute in the schema at every single node. This subscriptionID contains the identifier of the node where the subscription was stored. The subscriptionID also contains a unique identifier of the subscription, and the number of attributes contained in the subscription [4].

Range predicates work almost the same way as equality predicates, with one exception. Instead of just one subscriptionID being generated for one attribute, many are generated. In the schema, each attribute has a minimum, maximum, and precision fields as well as type, name and value fields, if appropriate. One subscriptionID is hashed for every value in the range of the attribute that satisfies the range predicate[4].

When an event is created, a similar series of events takes place. The value of each attribute that the message contains is hashed. A list of matching subscriptionID's is found for each attribute, and duplicates are weeded out. This has been called the collection phase [4]. You then iterate over all these lists, and discard all subscriptionID's where the number of subscription attributes field is not equal to the number of attributes in the publication. The message is then routed to the subscriber whose number of subscriptionID's is equal to the number of attributes specified in the message.

5. SIMULATION

To answer the questions posed, a network simulator was built using Java. This simulator is an overlay network, with the assumption that a Chord peer-to-peer network rests underneath. Broker nodes are created and can receive subscriptions. A subscriptionID is stored at a node (based on where it hashes to) for every attributes value in the subscription (and potential more than one subscriptionID for range predicates). Publications are then created, matched with interested subscriptions, and delivered to the node that generated the subscription.

The simulations were run on a machine with a 2.0 Ghz Pentium 4 processor, and 256MB of RAM. Other running processes were kept consistent between the various test runs.

5.1 Schema

For my simulation, there is a schema for event notifications and subscriptions. This schema lets both publishers and subscribers know what types of information are allowed in a given publish/subscribe environment. An example schema looks like this:

Table 1: Baseball Schema

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Game Date</td>
<td>Date</td>
</tr>
<tr>
<td>VisitorTeamName</td>
<td>String</td>
</tr>
<tr>
<td>HomeTeamName</td>
<td>String</td>
</tr>
<tr>
<td>VisitorScore</td>
<td>Int</td>
</tr>
<tr>
<td>HomeScore</td>
<td>Int</td>
</tr>
<tr>
<td>Winning Pitcher</td>
<td>String</td>
</tr>
<tr>
<td>Losing Pitcher</td>
<td>String</td>
</tr>
<tr>
<td>Save</td>
<td>String</td>
</tr>
</tbody>
</table>

Subscriptions for content-based publish/subscribe show what items and values are of interest to a subscriber. A particular subscriber might be interested in all home Red Sox games, with no regard to who the opponent is. Another subscriber might only want information about all Red Sox games, home or away, where the opponent is the New York Yankees. These are examples of where the subscriber would use an equality predicate; sometimes a subscriber may want to know whether a team scored more than 6 runs, or allowed fewer than 3. These types of situations use a range predicate.
5.2 Subscriptions

A subscription (using XML-style notation) under this schema looks something like:

```xml
<HomeTeamName>"BOS"</HomeTeamName>
```

Figure 1: Sample Subscription
to register interest in all Boston Red Sox home games, or:

```xml
<HomeTeamName>"BOS"</HomeTeamName>
<WinningPitcher>"C. Schilling"</WinningPitcher>
<VisitorScore> <3 </VisitorScore>
```

Figure 2: A sample Subscription

if you want to be notified when Curt Schilling wins a Red Sox home game in which he gave up less than three runs (ignoring the fact that there might have been relief pitchers for simplicity). The value of the VisitorScore attribute is a range predicate in this case.

5.3 Matching Algorithm

The matching algorithm that I'm testing the efficiency of looks something like this:

```plaintext
// collect lists of subID
for every attribute a in an event
  hash to the correct node
  retrieve all subID's
// match events to subscriptions
for every list of subID's
  for every subID
    get the number of attributes the subscription cares about
    if the number of collected lists is equal to the number of attributes
      a match has been found, so get rid of that subID from every list
    // deliver the message to the node that subscribed
```

Figure 3: pseudocode for a matching algorithm

This is a paraphrasing of the matching algorithm found in [4].

6. RESULTS

The network simulator was run under several different circumstances. First, the number of nodes in the system, N, was kept constant at 16. A few different values for the total number of subscriptionID's, K, were used. For each combination of N and K, the placement of those subscriptionID's were moved around such that any one node contained at most \((1 + e)*K / N\), where e is in the order of \(O(\log N)\).

![Figure 4: N=16, K=16](image)

7. CONCLUSIONS
The simulations that were run show that the matching algorithm does seem to have more difficulty when subscriptionID's are bunched up. You don't see much differentiation in when the number of keys is small compared to the number of nodes, but when the keys start getting more numerous, the matching algorithm slows down. If the publish/subscribe system weren't being run over a peer-to-peer network that used consistent hashing to more evenly distribute subscriptions to all the nodes, the problem would seem to be even greater.

8. REFERENCES