Efficient Event-based Resource Discovery

Wei Yan[⋆‡], Songlin Hu[⋆], Vinod Muthusamy[†], Hans-Arno Jacobsen[†], and Li Zha[⋆]

yanwei@software.ict.ac.cn, husonglin@ict.ac.cn,
vinod@eecg.toronto.edu, jacobsen@eecg.toronto.edu

[⋆]Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[‡]Graduate school, Chinese Academy of Sciences, Beijing, China
[†]Dept. of Electrical and Computer Engineering, University of Toronto, Toronto, Canada

ABSTRACT

The ability to find services or resources that satisfy some criteria is an important aspect of distributed systems. This paper presents an event-based architecture to support more dynamic discovery scenarios, including efficient discovery of resources whose attributes can change, and continuous monitoring for resources that satisfy a set of constraints. Furthermore, algorithms are developed to optimize the discovery cost by reusing results among similar concurrent discovery requests. Detailed evaluations under various workload distributions demonstrate the feasibility of the architecture and show significant benefits of the optimizations in terms of network traffic and discovery processing time.

Keywords

Service discovery, resource discovery, content-based publish/subscribe, publish/subscribe applications, covering, subscription similarity

1. INTRODUCTION

Distributed systems make use of large sets of resources and services, often scattered throughout geographical distributed locations and stored-away in hard-to-access data centers. A basic problem in deploying applications and operating a distributed system is the static and dynamic discovery of resources and services based on a set of desired attributes, such as: *"find ten machines with 2GHz-CPU and 2GB-memory in close proximity"*. This declarative, query-based approach is much more desirable as opposed to an approach based on a naming scheme that tries to encode configuration attributes within resource name references or globally unique identifiers. Also, certain resource attributes change dynamically (such as CPU load, memory, available disk space, and QoS information of application and Web servers). Discovering and monitoring of such kinds of resources and services have become recognized problems in recent years [4]. For example, efficient resource discovery is a fundamental problem in the context of Grid computing and an important problem in emerging Cloud computing infrastructures. In this paper we adopt the definition of resources and services outlined by Ahmed *et al.* [4] and we treat resource and service discovery in the same manner; simply referring to resource discovery from here on forward.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DEBS'09, July 6-9, Nashville, TN, USA.
Copyright 2009 ACM X-XXXXX-XX-X/XX/XX ...$10.00.

Many methods for resource discovery have already been proposed. For example, Federated UDDI [24], P2P-based approaches (e.g., DHT-based approaches [26, 25, 23, 27], and Overlay-based approaches [1, 14].) However, all these approaches focus on the routing mechanism of discovery and have not paid much attention to the inherent static and dynamic characteristics of resource discovery. Resources always have *static* attributes (like the CPU and memory configuration of a server), but also have *dynamic* attributes that change frequently (e.g., available disk storage changing from 80G to 40G as the result of the creation of a new file.) Current approaches lack flexibility, thus limiting their use, as they do not consider the difference in static and dynamic attributes characterizing resources' capabilities. Federated UDDI can handle discovery of static descriptions of resources efficiently, but lacks the capability of supporting querying of dynamic QoS attributes. P2P-based approaches route resource requests to the nodes that own the matching resources, even if only static attributes are of concern.

In this paper, an event-based resource discovery approach is proposed that combines static and dynamic resource discovery, and adds a third continuous discovery model. The approach exploits a distributed content-based publish/subscribe system to achieve scalable, efficient, and real-time resource discovery. Resource registrations, discovery requests, and results are all mapped to publish/subscribe messages. The "push" capabilities of the publish/subscribe model is also what allows for a powerful continuous discovery model where users can be notified in real-time of new resources that match their criteria.

In practice, it is expected that there is a degree of similarity among resource requests, similar to the findings on Web requests following a Zipf distribution [6], the discovery approach in this paper is optimized by sharing results among concurrent requests with similar interests. Again, publish/subscribe techniques, namely subscription covering, are used to help find similar requests and share their results.

The contributions of this paper are: (1) a unified static

and dynamic resource discovery framework based on content-based publish/subscribe (2) a new continuous discovery model that allow for efficient real-time notifications of newly registered matching resources; (3) an algorithm that optimizes discovery performance by sharing results among similar discovery requests; and (4) a detailed evaluation of the algorithms under a variety of workloads.

The rest of the paper is organized as follows. Section 2 presents related work on resource discovery, followed up by background information on content-based routing in Section 3. Section 4 describes the event-based resource discovery framework, and optimizations based on exploiting similarities among concurrent resource requests are outlined in Section 5. Section 6 presents and analyzes experimental results, and finally conclusions are presented in Section 7.

2. RELATED WORK

This paper presents a framework to enable decentralized discovery of both static and dynamic service and resource attributes, and to enable the continuous monitoring of resource updates. Some of the existing resource discovery approaches and techniques are outlined in this section.

Many resource discovery schemes have been developed. This includes centralized discovery based on indexing [22], a hierarchical indexing approach [9], Federated UDDI [24], flooding-based discovery with Gnutella [1], and resource management using distributed hash tables (DHT) [8]. A comprehensive survey of resource and service discovery in large-scale multi-domain networks is given by Ahmed *et al.* who compare many prominent discovery approaches ranging from industry solutions to state-of-the-art research [4].

Condor's Matchmaker adopts a centralized architecture, where resource descriptions and discovery requests are sent to a common central matching server that performs the resource matching work [22]. This centralized approach is efficient for local area deployments for which Condor was initially designed. However, for large-scale decentralized settings, the approach requires central administration and management for the operation of the matchmaking server. While a central point of control eases administration, it is also a central point of failure and a scalability bottleneck.

Globus's MDS is a resource discovery approach based on a hierarchical architecture [9]. In MDS-2, a Grid is comprised of multiple information sources that register with index servers via a registration protocol. Resource requesters use a request protocol to query directory servers to discover resource index servers and to obtain more detailed resource descriptions from their information sources. The index servers form a hierarchical architecture. The top index server answers requests that discovery the resources registered with its child index servers. This approach limits scalability, as requests trickle through the root server, which easily becomes a bottleneck.

Federated UDDI consisting of multiple repositories that are synchronized periodically [24]. Federated UDDI is a popular and efficient solution for service discovery in distributed service networks. However, it is much too expensive to replicate frequently updated information, and, thus, it is hard to directly utilize this approach to support discovery of dynamic information.

Gnutella is an unstructured peer-to-peer network [1]. The discovery mechanism in Gnutella is based on flooding. Discovery requests are routed to all neighbor nodes of a given node. This happens until a timeout occurs or until the matched resources are retrieved. The flooding mechanism creates a large volume of traffic for networks with many nodes, connections and resources.

Alternatives have been developed where requests are more selectively propagated [14]. The proposed techniques include random walks, learning-based and best-neighbor-based propagation. Unlike in Gnutella, nodes choose their collaborating peer nodes based on expertise and preference. Resource requests are not flooded over the network, but directed to only a few selected nodes. Thus, the request-forwarding algorithms may not find all results for a request, if the matching resources are distributed among many different nodes.

Publish/Subscribe has been leveraged for service discovery [19]. In this approach, service attributes and discovery requests are translated to messages in publish/subscribe systems. This approach cannot support the combination of multiple discovery models, like the static, dynamic, and continuous models proposed in this paper. Also, it does not consider the optimizations derived from processing multiple similar discovery requests. Li *et al.* [16] have studied historic data access in distributed publish/subscribe systems, which also use subscriptions as discovery messages. The objective of their work is different from ours and Li *et al.* do not address the resource discovery problem. Also, they do not develop optimizations for the processing of multiple concurrent discovery requests.

Approaches based on DHTs such as Chord [26] and Pastry [25] have been proposed. However, DHTs only efficiently support single keyword-based discoveries. A naive approach to resolving a range query issues separate point queries to nodes that correspond to each possible value within the query range is given in [8]. This approach becomes quite expensive for a typical sized range query and thus is unfeasible for more expressive resource requests. Techniques to more efficiently process range queries and multi-attribute queries in DHTs typically build additional indexes for the data items or add layers of indirection [12, 2, 20]. For example, Mercury assigns sets of nodes to be *hubs* for each resource attribute in the system [5]. These hubs index resources containing that attribute, and handle queries with that attribute. A limitation of the Mercury algorithm is that a multi-attribute query is decomposed into a set of single attribute queries that must be processed sequentially or in parallel.

The work in this paper, on the other hand, utilizes a distributed publish/subscribe overlay that can efficiently evaluate multi-attribute and range constraints. Furthermore, the volatile message propagation paths in DHTs would prevent the similarity forwarding algorithms proposed in this paper from being employed. Unlike typical peer-to-peer environments, however, this system is not designed for high node churn rates. It is also important to point out that we are not aware of any DHT-based resource discovery protocol that supports the continuous monitoring for resources with potentially dynamically updating attributes.

The problem of resource discovery complements earlier work on automatic service composition [13] where the resources, or services, that are found are automatically composed based on certain criteria to form a composite service with a specified interface. The automatic service composition work also exploited the scalable matching capabilities

of a distributed content-based publish/subscribe system as in this paper. Therefore, the features proposed in this paper such as the ability to continuously monitor for newly registered resources and the algorithms to exploit similarity among the resources and discoveries can be utilized in the earlier work.

3. CONTENT-BASED ROUTING

The algorithms in this paper utilize a publish/subscribe messaging substrate which provides a flexible and powerful interaction model for a wide variety of large-scale distributed systems. The publish/subscribe model consists of three basic elements: subscribers, who express interest in particular information by means of a subscription language; publishers, who publish information of interest; and a broker or broker network which is responsible for matching publications with subscriptions and for routing publications to interested subscribers. The brokers in the model decouple the publishers and subscribers in space and time making the publish/subscribe paradigm particularly well-suited for large and dynamic distributed systems. The publish/subscribe system can be normally divided into three categories: channel-based, topic-based and content-based.

The content-based model is more flexible and expressive than the alternatives. In the content-based model, subscriptions specify constraints on the contents of publications. Publications are routed to subscribers with matching subscriptions. There is no interactions that requires explicit addressing. Examples of content-based systems are LeSubscribe [10], Gryphon [3], and PADRES [11]. Some of these systems use subscription-based approaches, while others introduce advertisements to optimize the performance; some are centralized, while others are inherently distributed.

In an advertisement-based publish/subscribe system, an advertisement specifies the information that the publisher may publish in the future. Advertisement messages, flooded throughout the network, build a spanning tree rooted at the publisher, and serve to direct the routing of subscriptions only towards publishers whose advertisements are of potential interest to subscribers' subscriptions [17]. Finally, publications are delivered to interested subscribers along the paths built by subscription messages.

Figure 1 shows an example of content-based routing with advertisements in the PADRES publish/subscribe system. The subscription routing table (SRT), consisting of <advertisement, lasthop>-tuples, is used to route subscriptions, and likewise, the publication routing table (PRT) stores <subscription, lasthop> tuples that are used to route publications towards interested subscribers. For example, in Figure 1, advertisement adv_1 is broadcast throughout the network and stored at each broker with the appropriate *lasthop*. Subscriptions that match adv_1 are routed according to these *lasthops*; for example, sub_1 is routed along the Path $B-C-A$. Note that the subscription sub_1 is not forwarded to Broker D since adv_1 indicates that matched publications are from Broker A. Therefore, publication pub_1 is routed along the reverse Path $A-C-B$ to the subscriber.

Many publish/subscribe systems employ the covering routing optimization [17] to remove redundant subscriptions from the network in order to obtain a more compact routing table and to reduce the subscription routing traffic. Consider the example in Figure 1 when a new subscription sub_3 is issued from a a subscriber connected to Broker D. If subscription

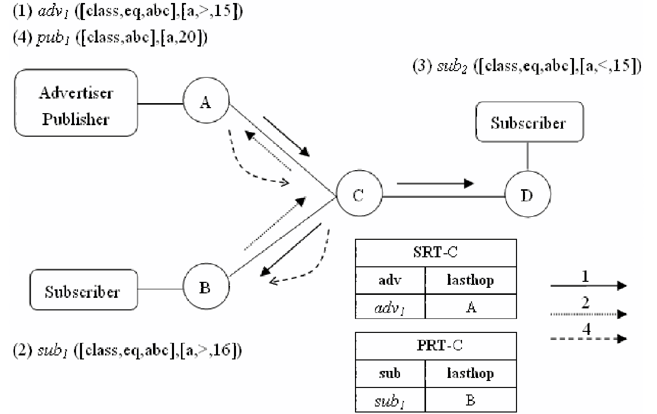


Figure 1: Content-based Routing

	One-time request	*Continuous request*
Static resource	static	static continuous
Dynamic resource	dynamic	dynamic continuous

Table 1: Supported models

sub_3 matches advertisement adv_1, sub_3 will be routed along the Path $D-C-A$. However, if subscription sub_1 covers subscription sub_3 (such as sub_3 ([class,eq,abc],[a,>,18])) at Broker C, sub_3 is not forwarded to Broker A. All publications matching sub_3 must also match sub_1. A formal definition of the covering relation is as follows: A subscription sub_1 covers sub_3, if and only if, $\mathrm{P}(sub_1) \supseteq \mathrm{P}(sub_3)$ (where $P(s)$ refers to the set of publications that match subscription s), denoted as $sub_1 \supseteq sub_3$. The covering relation defines a partial order on the set of all subscriptions with respect to $\supseteq$. The covering relations among advertisements can be defined in a similar manner.

4. RESOURCE DISCOVERY FRAMEWORK

This paper proposes a new resource discovery framework based on the publish/subscribe model in this paper. The framework supports two types of resources (static and dynamic) and two types of discovery requests (one-time and continuous) resulting in four models as summarized in Table 1. The static and dynamic *resource types* distinguish between resources whose attributes are constant or may change over time. On the other hand, the one-time and continuous discovery *request types* denote cases where requests are matched against existing resources, versus ones where requests are also continually matched against newly registered resources. The algorithms for each of the four models are described in detail in this section.

What is common across the models is that resource providers act as publishers and resource discovery clients act as subscribers. In order to allow a single system to contain resources and requests conforming to the different models, messages (including advertisements, subscriptions, and publications) are marked with a *ModelType* that specifies the model to be used. Valid *ModelType* values for discovery request messages are *static*, *dynamic*, *static continuous* (continuous query for static information), and *dynamic continuous* (continuous query for dynamic information). These

values denote the kind of information the requester wants. For the resource, the *ModelType* can only be *static* or *dynamic*, indicating whether the resource's attributes are all constant or whether some may vary dynamically.

The resource attributes and discovery constraints are specified as conjunctions of predicate constraints where each predicate is an [attribute,operator,value] tuple in which the *operator* and *value* specify a Boolean condition on the *attribute*. In this paper the attribute is a string, and predicate values may be integers, floating point numbers or strings. String types only support the equality operator, whereas the numeric types additionally support inequality operators ($<, \leq, >, \geq, =$).

For example, a resource with the static description "a Linux server with 2GB of memory and a 320GB disk" is represented by an advertisement "[system,=,Linux], [memory,<=,2], [disk,<=,320]" in the publish/subscribe system. Similarly, a discovery request for "Linux servers that have a disk larger than 120GB" is mapped to a subscription "[system,=,Linux], [disk,>,120]". As well, for dynamic resources whose attributes may vary frequently, their resource updates are conveyed with a publication. For example, a resource that currently has 1G of available memory and 200G of storage space could issue a publication such as "[system,Linux], [memory,1], [disk,200]".

In some cases, a resource may only be available during certain time periods. Such temporal availability constraints can be specified as predicates in both resource advertisements and discovery subscriptions. For example, a resource that is only available after 8 a.m. can include the following predicate in the advertisement: "[available,>,8 a.m.]".[1] Similarly, a discovery request for resources available between 7 a.m. and 3 p.m. can add the following predicates to the subscription: "[available,>,7 a.m.], [available,<,3 p.m.]".

4.1 Static model

The static model is designed to primarily handle information about static resources. The static resource description is registered with an advertisement message. This message is flooded and thus each broker caches information about all the static resources in the system. Recall from Section 3 that advertisements are cached in the SRT table in the PADRES system. To discover static resources, the discovery client simply submits a subscription message containing the desired resource constraints to a broker. Upon receiving the subscription, the broker matches the subscription with the local advertisements and returns the resulting advertisements as a package back to the discovery client. Notice that the discovery request for static resources is handled by a single broker.

Figure 2 illustrates an example of resource discovery in the static model. In a network composed of brokers A, B, C, and D, a resource connected to Broker A wants to register its resource information (ModelType=static,system=Linux, memory<=2, disk<=320). An advertisement adv_1 is generated that corresponds to these attributes and is flooded over the network. A discovery client connected to Broker C submits its request for servers with at least 1GB of memory as subscription sub_1. On receiving the subscription, Broker C queries its SRT for matching advertisements and then

[1]The time is specified in a convenient format here, but can be encoded using some mapping to an integer or floating point representation.

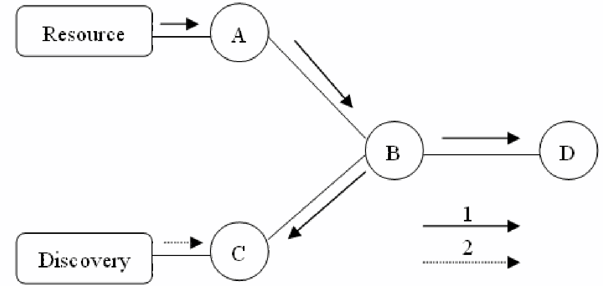


Figure 2: Static resource discovery model

delivers these matches to the client. The subscription needs not be routed to the other brokers but is processed entirely by the broker the discovery client is connected to.

4.2 Dynamic model

There are cases where some attributes of a resource description vary over time, such as the available memory or processor utilization of a server. The dynamic model is proposed to support such resources.

In the dynamic model, each broker maintains a *PubCache* structure to cache updates to dynamic resource attributes. The *PubCache* consists of pairs $<advID, pub>$, where $advID$ is the advertisement associated with a resource and *pub* is a publication that contains the latest information about the dynamic attributes of the corresponding resource.

To register a resource in the dynamic model, a resource provider connects to a broker and issues a resource description advertisement (with *ModelType* set to *dynamic*) that describes the ranges of its dynamic attributes. This advertisement is flooded across the network as in the static model. When resource attributes change, however, a publication message with the current resource attributes is generated and cached in the *PubCache* of the connecting broker.

To discover resources with dynamic attributes, the discovery client issues a subscription message to its broker (with the *ModelType* set to *dynamic*). The subscription is routed to brokers along the matching advertisement trees towards those brokers where potentially matching resources are connected. These brokers are referred to as *edge brokers.* When the subscription reaches these edge brokers, the broker reads the latest information of the resource from the *PubCache* and routes the information back to the subscriber along the path the subscription just traveled.

Figure 3 shows an example of resource discovery in the dynamic model. A resource at Broker A registers its resource (ModelType=dynamic,system=Linux, memory<=2, disk<=320) with advertisement adv_1. When the resource attributes change, the new values, say (system=Linux, memory=1, disk=200), are conveyed by issuing publication pub_1 to Broker A which then caches pub_1 in its *PubCache.* When a discovery client, connected to Broker C, wants to find servers with available disk space greater than 40GB, it issues subscription sub_1. Since adv_1 matches sub_1, sub_1 is routed along path $C - B - A$. At the last hop, Broker A

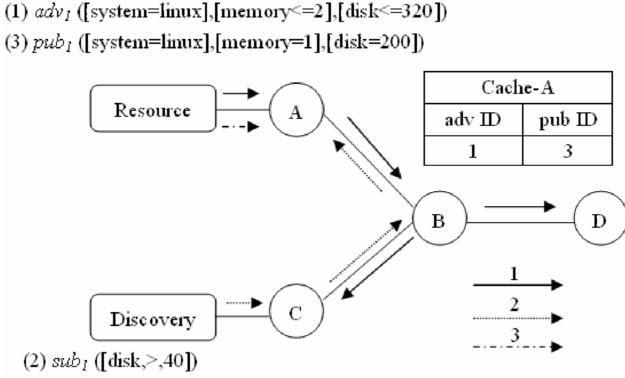


Figure 3: Dynamic resource discovery model

reads the latest attributes of resource adv_1 from its *Pub-Cache*, which is pub_1 in Figure 4, and routes pub_1 back to the discovery client along the path $A - B - C$ created by sub_1.

Notice that the algorithm does not directly send discovery results for resources hosted at Broker A to the discovery client at Broker C. This makes it possible to take advantage of the similarities among resources and requests and share these results among concurrent discovery requests, thereby reducing network traffic and message processing loads. This optimization is developed and described in Section 5.

4.3 Continuous model

In both the static and dynamic models above, the discovery request will only find resources that were registered before the request was issued. In some scenarios, a discovery client may wish to be notified of any changes to the resources in the system, including whether resources matching some criteria are added or removed from the system, as well as updates on the dynamic attributes of these resources.

The continuous model supports the ability of a client to submit a discovery request once and have matching resource information be delivered continuously as resources are added to the system or resource attributes are updated. As will be described, this model takes full advantage of the efficient matching and instantaneous message delivery capabilities of content-based publish/subscribe systems.

The continuous model is divided into two cases: a static continuous model to discover static resources, and a dynamic continuous model that is used to discover resources with dynamic attributes. The algorithms for both of these cases are outlined below.

4.3.1 Static continuous model

Discovery requests in the static continuous model are handled in a manner similar to static discovery requests as described in Section 4.1. The *ModelType* of the resource advertisements are still *static*, whereas the *ModelType* of the subscription request is now *static continuous*.

As in the static model, a broker that receives a discovery request subscription from a discovery client will query its SRT (which contains all the advertisements in the system), and return the matching resource advertisements to the client. In the *static continuous* model, however, the broker will also store the discovery request in its routing tables. When the advertisement associated with any subsequent resource registrations are received, the broker will match the new advertisement with previously issued *static continuous* discovery requests and notify the client of these new matching resources.

In this way, a client need only submit its discovery request once, and will be notified of matching resources as they are registered in real-time. A client that is no longer interested in these notifications can submit a corresponding unsubscription message and the broker will remove the indicated discovery request from its tables.

4.3.2 Dynamic continuous model

The *dynamic continuous* model fully exploits the original advertisement-based publish/subscribe routing algorithms. To begin, the resource registers its resource description with an advertisement message with the *ModelType* attribute set to *dynamic continuous* that is flooded over the network. Next, a discovery client that wants to continuously monitor some resources issues a subscription message containing the desired constraints which is then routed to brokers with matching resources. Finally, when the dynamic attributes of a resource changes, the updated publication message is generated and is routed to the interested discovery request subscribers according to the routing algorithm in publish/ subscribe systems.

Figure 4 illustrates an example of resource discovery in the dynamic continuous model. The resource connected to Broker A registers its resource description (system=Linux, memory<=2, disk<=320) with advertisement adv_1. Then a discovery client connects to Broker C as a subscriber, and indicates its interest in monitoring the status of all "Linux" machines (system=Linux) with subscription sub_1. This subscription is routed along path $C - B - A$ by tracing the reverse path of adv_1. Once the resource information changes, a new publication message is generated and routed back to the discovery client along the path $A - B - C$ traversed by sub_1.

4.4 Discussion

To summarize, the *ModelType* of the advertisements that register resources can be *static* or *dynamic*, representing the appropriate characteristics of the resource. Discovery request subscriptions, on the other hand, can have *ModelType* values of *static* or *static continuous* when the client wants to find static resources, or values *dynamic* or *dynamic continuous* to find dynamic resources.

Note that the discovery for static resources is fully handled by a single broker and these subscriptions do not need to be routed through the network. This is similar to the some of the centralized resource discovery approaches, although in this paper there are a set of brokers that can each independently service requests for static resources, allowing the system to scale better than purely centralized schemes.

Even in the case of discovery requests for dynamic resources, however, request subscriptions are only routed towards brokers with potentially matching resources. Notably, if no advertisements match the request subscription, the subscription is not forwarded.

This paper is mainly concerned with developing scalable resource discovery algorithms that support the models outlined above. While fault-tolerance is out of the scope of this paper, it is worth pointing out that the system can continue

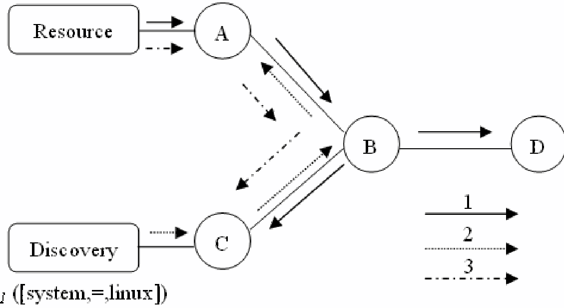


Figure 4: Continuous resource discovery model

to operate despite faulty clients. The system need not service a failed subscriber that has issued a discovery request, and the subscriber's subscriptions can simply expire after some time. Similarly, the failure of publishers is handled by expiring its resource registrations after some time. In terms of broker failures, research on reliable publish/subscribe systems can be adopted to address broker fault-tolerance concerns [7, 15, 21].

5. SIMILARITY FORWARDING

As described in Section 4, in the dynamic model resource discovery requests are routed to brokers which cache the dynamic attributes of matching resources. When the frequency of discovery requests becomes high, the routing and processing of discovery subscriptions and update publications across the network degrade the performance of the system as a whole.

One strategy to optimize this cost is based on the assumption that in a given system, some number of concurrent discovery requests may be similar. For example, most resource discovery requests on PlanetLab may simply want to find machines with sufficient memory or storage resources: "find a machine whose available storage is more than 40GB" (req_1 = "storage>40"), or "find a machine whose available storage is more than 50GB" (req_2 = "storage>50"). In this case, the relationship between the two requests is that req_1 covers req_2, that is, the set of resources that match req_1's criteria is a superset of those that match req_2.

Consider a scenario where req_2 is issued shortly after req_1. In this case, it would be desirable to reuse req_1's results by filtering out those resources that do not match req_2 and route the remaining results to the client that issued req_2. Doing so avoids the need to route req_2 to all brokers with potentially matching resources and process the request at these brokers. The rest of this section outlines algorithms to find and exploit such similar discovery requests.

5.1 Definition of discovery similarity

It is useful to define a metric that quantifies the degree of similarity among discovery requests. Among other uses, this metric is applied in Section 6 to help measure the effects of request similarity on discovery performance. Since the approach in this paper maps discovery requests to subscriptions, it suffices to define a similarity metric among a set of subscriptions.

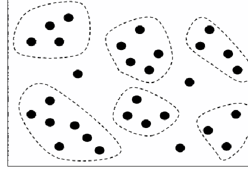


Figure 5: Subscription blocks

Consider the constellation in Figure 5 where each dot represents a subscription, and some subscriptions are grouped into blocks. Each block represents a connected covering network, in which any subscription in it will cover or be covered by some other subscription in the same block. Isolated subscriptions have no covering relationships with other subscriptions and do not belong to a block.

It can be costly to compute and consider all the covering relationships among a large set of subscriptions. As the primary use for the metric in this paper is to serve as an evaluation parameter, a relatively easy to compute metric that loosely captures the degree of similarity is sufficient.

Definition 1: The similarity of a set of subscriptions is a measure of the number of subscriptions with covering relationships compared to the total number of subscriptions.

Formally, subscription similarity is calculated as:

$$\textbf{Similarity} = \frac{\sqrt{\sum_{i=1}^{b} a_i^2}}{n}$$

where n is the total number of subscriptions in the set, b is the number of blocks, and a_i is the number of subscriptions in block i. For the example in Figure 5, the parameters are n=30, b=6, and a={4, 5, 4, 3, 4, 7}, yielding a similarity of approximately 0.38 after substitution into the formula.

One way to think about the similarity formula above is that given a set S of n subscriptions with b blocks, the numerator in the expression is trying to find a single equivalent block of size a_e that represents the similarity of the subscriptions in S. It is clear that if S only has one block of size a_1, the equivalent block should also have size a_1, and this is indeed the case in the above expression. When $b > 1$, however, it is less obvious what the equivalent block size should be, but it should be constrained by two bounds.

First, the equivalent block should be larger than any of the $b > 1$ blocks in S. Suppose this is not the case, and there is a block i such that $a_i > a_e$. This would mean that another subscription set S' also with n subscriptions but with only one block of size a_i would have a larger similarity metric than that of S: $Similarity(S') = a_i/n > Similarity(S) = a_e/n$. This is undesirable since the subscriptions in S clearly have more covering relationships than those in S' and the similarity metric should reflect this.

Second, the equivalent block should be smaller than the sum of the block sizes in S. Suppose this is not the case, and $a_e \geq a_m = \sum_i a_i$. Therefore a subscription set S'' with n subscriptions but where the b blocks in S are merged into one block of size a_m would have a smaller similarity measure than S: $Similarity(S'') = a_m/n > Similarity(S) = a_e/n$. This is undesirable since the subscriptions in S'' have more covering relationships than those in S and again the similarity metric should reflect this.

By taking the square root of the sum of squares of each

Algorithm 1 Subscription forwarding

```
Require: An incoming subscription message sub
  matchingAdvs ⇐ findMatchingAdvs(sub)
  if sub.payload = null then
    coverSub ⇐ null

    {Look for a covering sub.}
    for each srdSub in SrdSubList do
      if srdSub.covers(sub) then
        coverSub ⇐ srdSub
        break
      end if
    end for

    if coverSub = null then
      SrdSubList.insert(sub)
      route(sub, lastHops(matchingAdvs))
    else
      coverSub.waitingList.insert(sub)
      if this.broker = coverSub.DHBroker then
        route(coverSub.cachedResults, sub.lasthop)
      else if coverSub.hasResults() or coverSub.lasthop ∈
      lastHops(matchingAdvs) then
        sub.payload = coverSub
        route(sub, coverSub.lasthop)
      end if
    end if
  else
    coverSub ⇐ sub.payload
    if this.broker = coverSub.DHBroker then
      coverSub.waitingList.insert(sub)
      route(coverSub.cachedResults, sub.lasthop)
    else
      if ∃ neighbor ∈ lastHops(matchingAdvs) :
      neighbor ∉ {coverSub.lasthop, sub.lasthop} then
        coverSub.waitingList.insert(sub)
      end if
      if coverSub.hasResults() or coverSub.lasthop ∈
      lastHops(matchingAdvs) then
        route(sub, coverSub.lasthop)
      end if
    end if
  end if
```

Algorithm 2 Publication forwarding

```
Require: An incoming publication message pub
  if pub.lastHop.isClient() then
    cache(pub)
  else
    s1 ⇐ pub.relatedSubMessage
    route(pub, SubWaitingListMap.get(s1))
    route(pub, s1.lastHop)

    if SrdSubList.contains(s1) and s1.lastHop.isClient() then
      ResourceCacheMap.insert(s1, pub)
    end if
  end if
```

a_i, the similarity metric satisfies both these bounds.[2] This justifies the similarity metric above.

5.2 Similarity forwarding algorithm

As described earlier, a client discovers resources by submitting a subscription message to a publish/subscribe broker. Covering relationships may exist among different discovery subscriptions, and this section outlines these relationships that are used to optimize the discovery cost.

In the discussion below, the broker that a discovery client connects to is referred to as the discovery *host broker* (*DHBroker* for short), and the broker that a resource connects to is the resource host broker (*RHBroker*). Also, the broker or client from which a resource registration advertisement, discovery request subscription, or discovery result publication is received is referred to as the *last hop* of the associated message.

As part of the similarity forwarding algorithm, each broker maintains three additional data structures: The *SrdSubList* structure caches the discovery requests that are not covered by other requests in the current broker. The *ResourceCacheMap* structure caches the discovery results for those requests for which the broker is a host broker, that is, requests from clients directly connected to the broker. Finally, the *SubWaitingListMap* structure caches discovery requests waiting for the results from other discovery requests. The requests in this structure are covered by the ones in the *SrdSubList* structure.

When no covering discovery requests have been seen by a broker, requests are forwarded as usual towards resource host brokers with potentially matching resources, which then return information about the matching resources to the requesting client. However, when a covering request is found, the new covered request retrieves the results directly from the discovery host broker where the covering request results have been cached. In addition, to account for the case where the results for the covering request have not been delivered to the requesting client yet, the covered request is also forwarded to brokers that may potentially have outstanding results.

In the similarity forwarding algorithm, when propagating a subscription s, the first broker B that finds a subscription s' in its routing table that covers s forwards s towards the *DHBroker* of s'. This is done in order to retrieve any cached results of s' in the *DHBroker*'s *ResourceCacheMap*. Broker B also stores s in its *SubWaitingListMap* so it can intercept any new results for s'. If a broker does not find any covering subscription, it propagates s as usual based on the matching advertisements. Subscription forwarding in the similarity forwarding algorithm is detailed in Algorithm 1.

Resource update publications are cached at the resource's *RHBroker*, and forwarded hop by hop towards any matching subscription's *DHBroker* where it is also cached. As well, each broker consults its *SubWaitingListMap* to find any discovery requests that are waiting to intercept the update publication, and also propagates the publication towards these covered subscriptions. Publication forwarding is outlined in Algorithm 2.

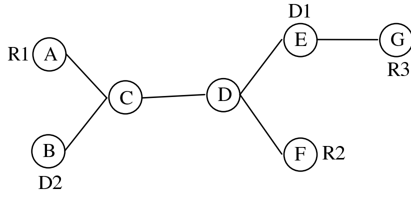
The similarity forwarding algorithm caches some information at the brokers including the discovery requests and discovery results. These cache entries expire after some globally configured expiration time (such as 100s). Longer expiration times offer more opportunities to exploit similarity among discovery requests at the expense of returning possibly stale results.

5.3 Similarity forwarding example

Figure 6 presents an example of the similarity forwarding algorithm. The example consists of a seven broker network (Brokers A to G), two discovery requests (D1 and D2) and three resources (R1, R2, and R3), where it is assumed that all three resources match both discovery requests, and that discovery request D1 covers D2.

In Figure 6, a client connects to Broker E, and submits its discovery request D1. Broker E finds no requests in its

[2] By the Pythagorean Theorem, in a right triangle, $a^2 = b^2 + c^2$, where a is larger than both b and c, but smaller than $(b + c)$.



Broker	SrdSubList	ResourceCacheMap	SubWaitingListMap
A	D1		
B	D2	D2(R1,R2,R3)	
C	D1		D1(D2)
D	D1		D1(D2)
E	D1	D1(R1,R2,R3)	D1(D2)
F	D1		
G	D1		

Figure 6: Similarity forwarding example

SrdSubList that cover D1 and so adds D1 to its *SrdSubList*. The broker then forwards D1 to neighbors from which it has received advertisements that match D1, Brokers D and G in this case. This process is repeated at each broker, including adding D1 to the *SrdSubList* if no covering requests are found, until D1 reaches all the resource host brokers with matching resources, Brokers A, F and G in this case. Next, the information about the matching resources R1, R2, and R3 are routed back to Broker E along paths $A-C-D-E$, $F-D-E$, and $G-E$, respectively. In addition, the information about resources R1, R2, and R3 are cached in Broker E's *ResourceCacheMap*. The tables in Figure 6 summarize the key state at each broker in the network, including the fact that Brokers A, C, D, E, F, G have added request D1 to their *SrdSubList*. Note that in reality these tables are distributed among the brokers and are only presented together in the figure for convenience.

Continuing the scenario in Figure 6, suppose a new client connects to Broker B, and issues discovery request D2. Broker B finds no discovery requests in *SrdSubList* that cover D2, adds D2 to *SrdSubList*, and finally routes D2 to neighbors with matching advertisements, which is only Broker C in this case. Broker C, however, finds that there is a request, D1, in *SrdSubList* that covers D2. At this point Broker C begins the process of looking for results that match D1 in two places: at those brokers that may have already received these results, and those that may still receive new results.

For the former case, D2 is forwarded to D1's last hop (which from Broker C's point of view is Broker D) until it reaches D1's host broker where any cached results are retrieved from the host broker's *ResourceCacheMap*. For the latter case, Broker C stores D2 into its *SubWaitingListMap* in order to intercept new results for discovery request D1. The new results may arrive from neighbors other than the last hop of D2 (Broker B) and the last hop of D1 (Broker C). In this scenario, Broker A is the only such neighbor, which incidentally is also the host broker of R1. Similarly, D2 is inserted into the *SubWaitingListMap* at Brokers D and E.

At Broker E, the host broker of D1, the algorithm filters the cached results for D1 preserving only those resources that also match D2, and routes the results back along the path $E-D-C-B$. When the results finally arrive at Broker B, the algorithm also caches the results here to be used by any future discovery requests that are covered by D2.

In this way, the similarity forwarding algorithm exploits the concept of subscription covering in publish/subscribe systems to find discovery requests whose results may be shared. As well, distributed publish/subscribe routing algorithms are used, with some modifications, to ensure that requests are forwarded to all brokers that potentially have already or will receive shared results. In situations with many concurrent discovery requests for overlapping sets of resources, this algorithm can provide significant benefits, as evaluations in Section 6 confirm.

6. EVALUATION

The primary objective of this section is to observe, quantify, and understand the performance of the resource discovery algorithms presented in this paper. Different workloads are evaluated in particular those with varying degrees of similarity among the discovery requests.

6.1 Setup

The algorithms in this paper have been implemented in Java over the PADRES distributed content-based publish/subscribe system [11].

The experiments are conducted on a real deployment across a cluster of machines that represent a data-center environment. The network topology consists of 20 brokers each running on a cluster node, as well as 4 other brokers that are central in the network and simply act as publish/subscribe routers. In order to accurately measure some of the metrics, all the clients for issuing the experimental workloads run on a separate node.

The evaluations measure four metrics. The *average discovery time* represents the overall time duration from when a discovery request is issued to when the results are returned. The *number of publication messages* is the number of hops traversed by publications, and likewise for the *number of subscription messages*. Recall that subscriptions correspond to discovery requests and publications to results. Finally, the *matching operations* metric counts the number of times a matching operation is performed by the brokers. In the PADRES system, which internally uses the Jess rule engine to perform the matching operations [18], the number of matching operations is simply the number of times the Jess engine is invoked.

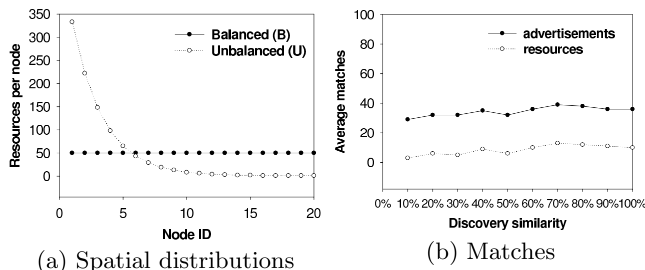
6.2 Parameters

To isolate against the effects of several parameters, the experiments start with a simplified, basic setting. It is assumed that no failures occur at either the resource or node level [14]. Also, all the resources are registered in the network before the discovery requests are issued and are not unregistered during the experiment. This avoids the influence of the fixed, known costs of resource registration from the discovery cost which is of more interest.

The details about the resources and discovery requests used in the experiments are outlined in the following sections.

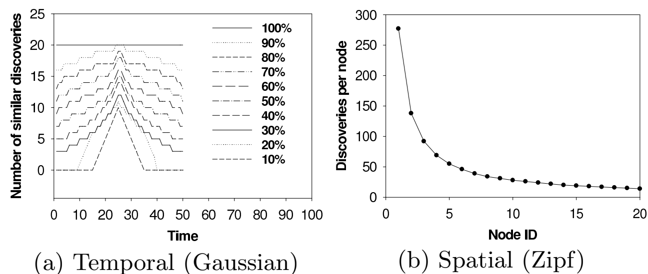
6.2.1 Resources

So as to simplify the experiments, only five tags (a, b, c, d, e) are used to represent the resource attributes. Each



(a) Spatial distributions (b) Matches

Figure 7: Resource workloads



(a) Temporal (Gaussian) (b) Spatial (Zipf)

Figure 8: Discovery workload distributions

resource randomly selects at least two attributes, and each attribute's value is randomly selected from the range 1∼100.

During the experiment, the dynamic information of these deployed resources change randomly, both in terms of the frequency of the updates, and the resource parameter value in the update, subject to being within the range that the resource advertised during registration. Recall that these updates are not propagated over the network but only refresh the cache at the resource's host broker.

Resources are registered at brokers in the network according to one of two distributions: *balanced* and *unbalanced*. In the balanced distribution, resources are uniformly registered across the network, whereas in the unbalanced distribution, the resources are deployed following a geometric distribution $P(X = n) = (1 - p)^{n-1}p$, with $p = 1/3$ in this paper. In the latter case, most of the resources are registered at a small number of brokers. In total 1000 resources are deployed across the 20 nodes. Figure 7(a) shows the number of resources deployed on each node for the two distributions.

6.2.2 Discovery requests

Ten discovery request workloads are generated, each containing 1000 discovery requests, and with each workload having similarities ranging from 10% to 100%. Each generated discovery request contains attribute a and some of the other four attributes (b, c, d, e). In order to remove the effect of different discovery result sizes on the query time, we take into account the average result size of discovery results in each workload. Figure 7(b) shows the average number of resources matched by a discovery request workload.

Among these generated discovery requests, some may match one or more resources, while others may match none. The former are referred to as valid discovery requests, and the latter invalid. The similarity of a workload is varied by replacing invalid discovery requests with ones that are covered by valid ones.

In the experiments, discovery requests are varied in both time and space. The time distribution refers to how the requests are issued over time, with the number of requests issued per unit of time following various Gaussian distributions as illustrated in Figure 8(a). The space distribution, on the other hand, is concerned with which brokers the requests are issued to, and in this case a Zipf distribution [6], as shown in Figure 8(b), is used. This means that most requests will originate from a small set of brokers.

6.3 Results

Two sets of experiments are presented here. The first set evaluates the performance of the similarity forwarding algorithm under a variety of workloads, followed by a set of experiments that quantify the costs and benefits of a decentralized resource discovery architecture compared to a centralized one.

6.3.1 Similarity forwarding

This section evaluates the effect of similarity among discovery requests on the performance of the discovery algorithms under various environments.

For comparison, both the un-optimized algorithm presented in Section 4.2 and the optimized similarity forwarding algorithm are evaluated. The un-optimized algorithm is denoted *Normal*, and the optimized one *Similarity*.

Figures 9, 10, and 11 show the results of the discovery algorithms with different workloads. In each experiment, both balanced and unbalanced resource distributions are considered. Various discovery request distributions are presented as follows: balanced in both time and space (Figure 9); balanced distribution in time and Zipf distribution in space (Figure 10); and Gaussian distribution in time and balanced distribution in space (Figure 11). In each figure, there is a chart for each of the four metrics: the overall discovery time, the publication message traffic, the subscription message traffic, and the number of times a matching operation is executed.

Overall, the results show that the similarity forwarding optimization can significantly reduce the overall discovery execution time, network traffic cost, and processing overhead. Furthermore, the benefits grow when the discovery requests exhibit increasing similarity, demonstrating the ability of the similarity forwarding algorithm to exploit workloads where the results among discovery requests can be shared.

In terms of the publication messages (the second chart in Figures 9, 10, and 11), all the experiments show that the similarity forwarding algorithm greatly reduces the number of publications. It does this by, when possible, retrieving results from the caches at discovery request host brokers instead of collecting the results from each individual resource host broker.

One seemingly odd result is that the number of publication messages (which are used to deliver discovery results) grows with increasing similarity. However, this is expected because in the workload the number of valid discoveries (which match at least one resource) increases with the similarity degree, thereby resulting in an increase in the number of matching resources, and hence more publication messages, as the discovery requests become more similar.

The number of subscription messages (the third chart in Figures 9, 10, and 11) remains relatively constant for

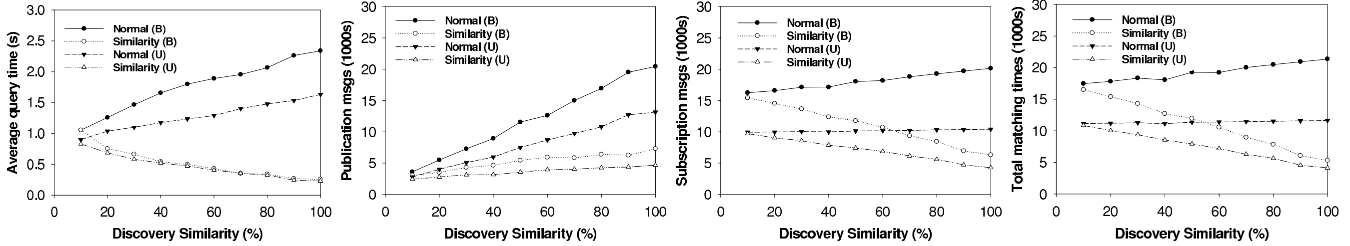


Figure 9: Uniform spatial and uniform temporal discovery distribution

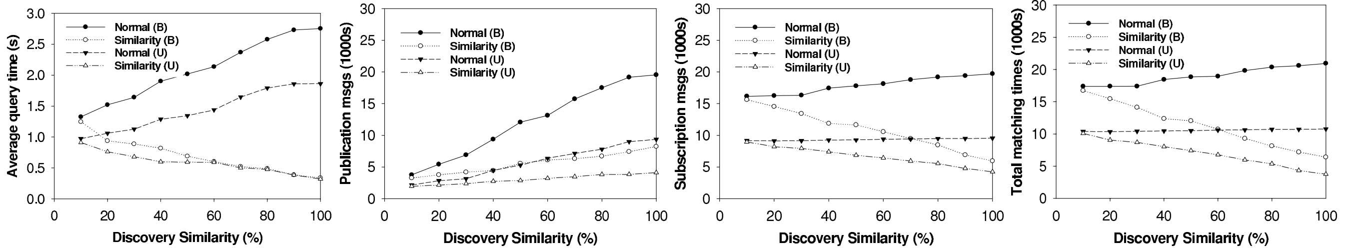


Figure 10: Zipf spatial and uniform temporal discovery distribution

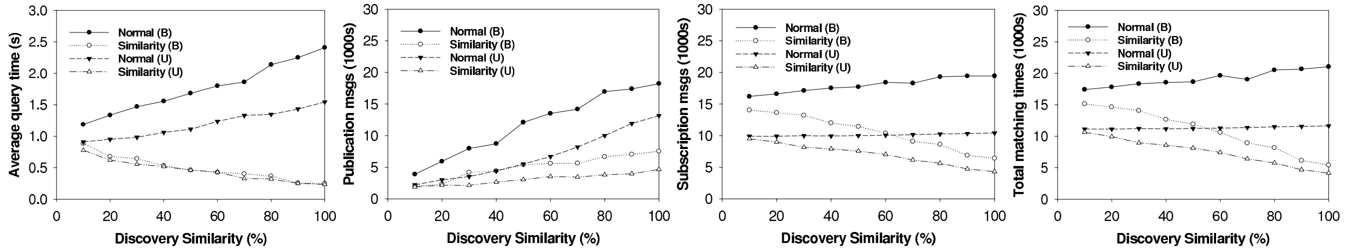


Figure 11: Uniform spatial and Gaussian temporal discovery distribution

the normal un-optimized discovery algorithm. Recall that subscriptions, which correspond to discovery requests, are routed towards potentially matching resources according to their registration advertisements. Since the average number of matching advertisements for each workload is relatively stable, as seen in Figure 7(b), it is not surprising that the number of subscription messages is also stable. With the optimized similarity forwarding algorithm, however, subscriptions sometimes only need to be routed to the host broker of a covering subscription rather than to all host brokers of all the matching resources. These savings increase when more similarity is available to be exploited.

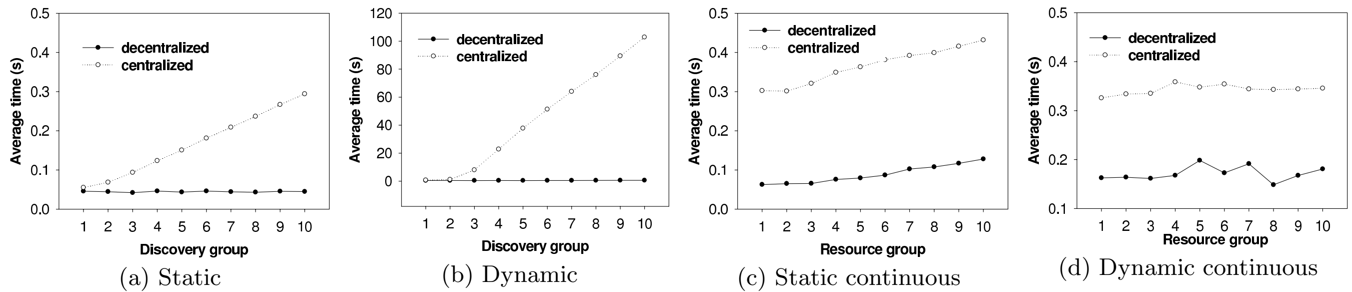
The processing time spent by brokers executing matching operations (the fourth chart in Figures 9, 10, and 11) closely tracks the number of subscriptions (the second chart in the figures) for both algorithms. This is a straightforward result of each broker attempting to match subscriptions against the advertisements in its routing tables. However, with the similarity algorithm there are cases where a subscription that is covered by another one can simply be forwarded toward a particular host broker without having to first find matching advertisements. The results do indeed confirm that the matching cost with the similarity forwarding algorithm can be less than the number of subscriptions especially when there is more similarity among the requests, and more temporal locality of similar requests (Figure 11).

There is an interesting effect of the spatial distribution of the resources in the network. In terms of the average discovery latency, messages and matching time, both the normal and similarity forwarding algorithms perform better when the resources are deployed according to the unbalanced rather than balanced distribution in Figure 7(a). The reason for this is that in the unbalanced distribution, most resources tend to be located in one part of the network and the discovery messages are more likely to be isolated to this part of the network.

6.3.2 *Decentralized architecture*

This section compares the decentralized resource discovery architecture proposed in this paper with a centralized deployment. All four models discussed in Section 4 are evaluated to measure the time it takes to find and report the matching resources for a set of discovery requests.

The decentralized deployment is the 24 broker network used in the earlier experiments, whereas the centralized deployment consists of a single broker to which all clients con-



(a) Static (b) Dynamic (c) Static continuous (d) Dynamic continuous

Figure 12: Query time for centralized and decentralized architectures

nect. The workloads for each model are as follows.

In the *static model*, 1000 resources are first registered and then 1000 discovery requests are issued. In the *dynamic model*, 1000 resources and requests are issued as in the static model, but this time the attributes of the resources are updated frequently throughout the experiment. In the *static continuous model*, 20 discovery requests are issued, one to each broker in the decentralized case but all to one broker in the centralized case. Then, 1000 resources are registered. In the *dynamic continuous model*, 20 resources are registered followed by 20 discovery requests distributed as in the static continuous case. Then, the attributes of the 20 resources are updated throughout the experiment.

In all the above cases, the resources and discovery requests follow the uniform spatial distribution for the decentralized deployments. For the two dynamic cases, each resource generates 50 resource update publications for a total of 1000 publications.

Figure 12 shows the results of evaluating the four models. In Figures 12(a) and 12(b), sets of discovery requests with increasing number of expected resource matches are issued. Each data point represents the average query time of the requests in the corresponding set of requests. The results show that unlike in the centralized architecture, the decentralized deployment, by effectively distributing the load, maintains a relatively constant overall query time despite an increasing number of matching results. The continuous model results in Figures 12(c) and 12(d) also show that overall discovery time is better when there are multiple brokers in the system.

The results in Figure 12 show that the multiple brokers in the decentralized architecture can deliver discovery results faster than the centralized resource discovery architecture in which the single broker can become a bottleneck. The tradeoff, however, is that the decentralized architecture imposes additional network traffic to propagate the discovery requests and results among the broker network.

Figure 13 quantifies this decentralization cost by plotting the total number of message hops for the experiments in Figure 12. While the message overhead may be large, it is important to note that despite these additional messages, the earlier results showed that the system as a whole is able to perform the discoveries faster and offer a more responsive and scalable service to the discovery clients.

Another set of experiments were conducted to evaluate whether it is better to send discovery results directly back to the requesting client without traversing through the brokers in the overlay. This algorithm, which was briefly mentioned at the end of Section 4.2, is denoted *Direct* and is compared to the *Normal* unoptimized discovery algorithm. The results

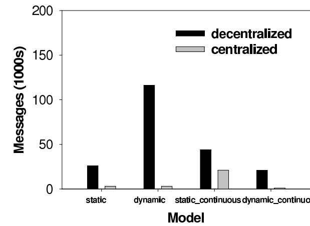


Figure 13: Decentralization message overhead

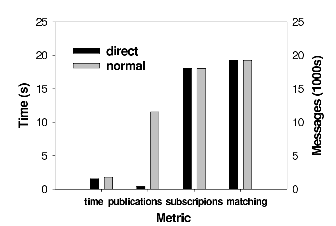


Figure 14: Direct and normal result delivery

under a balanced resource and discovery request distribution and 50% similarity are shown in Figure 14.

The experiments show that directly sending results to the requesting client can significantly reduce the publication traffic but provides negligible benefits when considering the other metrics, notably the overall discovery time. The Direct method also suffers from requiring a new connection to be established between every pair of matching resource host broker and discovery request host broker, something that may not be feasible in some environments for performance or security reasons. As well, by sending the results back, the algorithm makes it impossible to use the similarity forwarding techniques to share results among similar discovery requests.

The earlier experiments show that the similarity forwarding algorithm significantly improves all four metrics (overall discovery time, publication and subscription traffic, and matching time). Although the publication traffic costs are better with the Direct method, on balance the similarity forwarding algorithm may be the better choice.

7. CONCLUSIONS

Supporting multiple types of resources, high performance, and massive scalability are some of the most important goals in the design of a distributed resource discovery system.

This paper proposes a new resource discovery framework that leverages the properties of distributed content-based publish/subscribe systems. The framework supports three discovery models: static, dynamic and continuous. The static model is used to discover resources with fixed attributes, the dynamic model for resources with attributes that may be updated, and the continuous model allows for real-time notifications of newly registered resources. All three models can co-exist in one system and complement one another. In addition, a similarity-based optimization

algorithm is presented that utilizes publish/subscribe covering techniques to reuse the discovery results among different concurrent discovery requests.

Thorough evaluations and analyses of the algorithms are presented under a variety of workloads. The experimental results show that the similarity optimization can substantially reduce the discovery costs in terms of the time to perform discoveries, the network traffic incurred by discovery in a distributed system, and the discovery processing overhead. Moreover, these benefits improve in scenarios where the discoveries exhibit increasing similarities.

Deploying and evaluating the described framework on real-world applications in the Chinese National Grid System is one of our short-term future goals. In this context, the approach will be evaluated against many actual workloads running on the Chinese grid.

Acknowledgments

The completion of this research was made possible in part thanks to the China National 973 Program 2007CB310805, the China National 863 Program 2006AA01A106, and the NSFC Project 60752001, the Beijing Natural Science Foundation project 4092043, and the Co-building Program of Beijing Municipal Education Commission.

This research was also supported by Bell Canada's Bell University Laboratories R&D program and IBM's Center for Advanced Studies. This work builds on the PADRES research project sponsored by CA, Sun Microsystems, the Ontario Centers of Excellence, the Canada Foundation for Innovation, the Ontario Innovation Trust, and the Natural Sciences and Engineering Research Council of Canada.

8. REFERENCES

[1] Gnutella. http://www.gnutella.com.

[2] I. Aekaterinidis and P. Triantafillou. Pastrystrings: A comprehensive content-based publish/subscribe DHT network. In *ICDCS*, 2006.

[3] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *PODC*, 1999.

[4] R. Ahmed, N. Limam, J. Xiao, Y. Iraqi, and R. Boutaba. Resource and service discovery in large-scale multi-domain networks. *IEEE Communications Surveys & Tutorials*, 9(4), 2007.

[5] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. 2004.

[6] L. Breslau, P. Cao, L. Fan, G. Phillips, , and S. Shenker. Web catching and Zipf-like distributions: Evidence and implications. In *INFOCOM*, 1999.

[7] R. Chand and P. Felber. XNET: A reliable content-based publish/subscribe system. In *SRDS*, 2004.

[8] A. S. Cheema, M. Muhammad, and I. Gupta. Peer-to-peer discovery of computational resources for grid applications. In *GRID*, 2005.

[9] K. Czajkowski, C. Kesselman, S. Fitzgerald, and I. Foster. Grid information services for distributed resource sharing. *IEEE HPDC*, 2001.

[10] F. Fabret, H.-A. Jacobsen, L. F, J. Pereira, K. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *ACM SIGMOD*, 2001.

[11] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovski. The PADRES distributed publish/subscribe system. In *ICFI*, 2005.

[12] A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi. Meghdoot: Content-based publish/subscribe over P2P networks. In *ACM Middleware*, 2004.

[13] S. Hu, V. Muthusamy, G. Li, and H.-A. Jacobsen. Distributed automatic service composition in large-scale systems. In *DEBS*, 2008.

[14] A. Iamnitchi, I. Foster, and D. C. Nurmi. A peer-to-peer approach to resource discovery in grid environments. In *IEEE HPDC*, 2002.

[15] R. S. Kazemzadeh and H.-A. Jacobsen. Delta-fault-tolerant publish/subscribe systems. Tech Report, Univ. of Toronto, 2007.

[16] G. Li, A. Cheung, S. Hou, S. Hu, V. Muthusamy, R. Sherafat, A. Wun, H.-A. Jacobsen, and S. Manovski. Historic data access in publish/subscribe. In *DEBS*, 2007.

[17] G. Li, S. Hou, and H.-A. Jacobsen. Routing of XML and XPath queries in data dissemination networks. In *IEEE ICDCS*, 2008.

[18] G. Li and H.-A. Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *ACM Middleware*, 2005.

[19] D. Lynch. A proactive approach of semantically oriented service discovery. Master's thesis, Trinity College Dublin, 2005.

[20] V. Muthusamy and H.-A. Jacobsen. Small-scale peer-to-peer publish/subscribe. In *P2P Knowledge Management Workshop at MobiQuitous*, July 2005.

[21] G. P. Picco, G. Cugola, and A. L. Murphy. Efficient content-based event dispatching in presence of topological reconfiguration. In *ICDCS*, 2003.

[22] R. Raman, M. Livny, and M. Solomon. Matchmaking: distributed resource management for high throughput computing. In *IEEE HPDC*, 1998.

[23] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *ACM SIGCOMM*, 2001.

[24] P. Rompothong and T. Senivongse. A query federation of UDDI registries. In *Proc. of the 1st International Symposium on Information and Communication Technologies*. Trinity College Dublin, 2003.

[25] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *ACM Middleware*, 2001.

[26] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM*, 2001.

[27] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and and routing. Technical report, Berkeley, CA, USA, 2001.