INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery

Magdalena Balazinska, Hari Balakrishnan, and David Karger

MIT Laboratory for Computer Science
Cambridge, MA 02139
{mbalazin,hari,karger}@lcs.mit.edu

Abstract. The decreasing cost of computing technology is speeding the deployment of abundant ubiquitous computation and communication. With increasingly large and dynamic computing environments comes the challenge of scalable resource discovery, where client applications search for resources (services, devices, etc.) on the network by describing some attributes of what they are looking for. This is normally achieved through directory services (also called resolvers), which store resource information and resolve queries. This paper describes the design, implementation, and evaluation of INS/Twine, an approach to scalable intentional resource discovery, where resolvers collaborate as peers to distribute resource information and to resolve queries. Our system maps resources to resolvers by transforming descriptions into numeric keys in a manner that preserves their expressiveness, facilitates even data distribution and enables efficient query resolution. Additionally, INS/Twine handles resource and resolver dynamism by treating all data as soft-state.

1 Introduction

An important challenge facing pervasive computing systems is the development of scalable resource discovery techniques that allow client applications to locate services and devices, in increasingly large-scale environments, using detailed *intentional* [1] descriptions. Resource discovery systems should achieve three main goals: (i) handle sophisticated resource descriptions and query patterns; (ii) handle dynamism in the operating environment, including changes in resource state and network attachment point; and (iii) scale to large numbers of resources spread throughout a wide network across administrative domains. While some systems have addressed limited combinations of these properties, we address all three in this paper.

Resource discovery efforts have been largely geared toward expressive resource descriptions and complex query predicates [1–7]. These approaches differ in the details of how they name resources and how these names resolve to the appropriate network location. However, they all essentially rely on semistructured resource descriptions [8], *attribute-based* naming schemes with orthogonal attribute-value bindings, in which some attributes are hierarchically dependent on others.

Many resource discovery schemes have been designed primarily for small networks [1–4, 7], *or* for networks where dynamic updates are relatively uncommon or infrequent (e.t., DNS [9], LDAP [10]). They do not work well when the number of resources grows, *and* updates are common.

Static resource partitioning [11] and hierarchical organization of resolvers [5, 6, 12] solve scalable and dynamic resource discovery. Static partitioning relies on some application-defined attribute to divide resource information among resolvers. However, static partitioning does not guarantee good load distribution and burdens clients with selecting the relevant partitions.

Hierarchical approaches [5] organize resolvers around increasingly large domains for which they are responsible. These domains are created around particular attributes of resource descriptions, such as geographical location. Other hierarchical schemes keep data information in local resolvers and create hierarchies to filter out irrelevant queries as they travel toward the leaves [6, 12]. However, even if many hierarchies coexist, or if the hierarchies are created dynamically, root nodes may become bottlenecks. If queries are not propagated through root nodes to avoid bottlenecks, results become dependent on the origin of each query. Also hierarchies may not efficiently resolve queries that involve multiple orthogonal attributes. For example, imagine a metropolitan resource discovery system, where resolvers are hierarchically organized by institutions, neighborhoods, cities, and finally the metropolitan level. A client may be interested in locating all cameras filming main points of congestion in the metropolis, independent of location. The hierarchy described would not handle this query in a scalable manner, since it is based on location.

We describe the architecture, implementation, and evaluation of *Twine*, an approach to resource discovery that achieves scalability via hash-based partitioning of resource descriptions amongst a set of symmetric peer resolvers. Twine works with arbitrary attribute sets. It handles queries based on orthogonal and hierarchical attributes, with no content or location constraints. It also handles *partial queries*, queries that contain only a subset of the attributes originally advertised by resources (considering the other attributes as wildcards). Twine evenly distributes resource information and queries among participating resolvers. Finally, our system efficiently handles both resource and resolver dynamism. Twine is integrated with INS [1], the Intentional Naming System from MIT and now forms the core of its architecture. Therefore, we refer to INS/Twine nodes as *Intentional Name Resolvers* (INRs).

INS/Twine leverages recent work in peer-to-peer document publishing and distribution ([13–16]). Peer-to-peer systems do not rely on any hierarchical organization or central authority for distributing content or searching for documents. However, current peer-to-peer applications lack adequate querying capability for complex resource queries.

INS/Twine is designed to achieve scalable resource discovery in an environment where *all resources are equally useful*. We could imagine deploying INS/Twine throughout the Internet and letting everyone announce resources of global interest to users around the world. Such resources may be file servers, cameras showing the weather in different cities, Web services [17], and so on. Similarly we could imagine deploying INS/Twine within a city and letting users access resources such as air quality sensors, water temperature/quality indicators at public beaches, business information (e.g., number of currently available cars at a car rental company), and so on. In each deployment scenario different resources are advertised, but in both cases, all resources are potentially equally useful to clients. An important goal in INS/Twine is therefore to make all resources available to all users independent of location. However, location-dependent queries are handled well by specifying "location" as an attribute.

In both examples, the number of resources could be considerable. There are around 10^5 establishments in a city the size of Los Angeles or New York [18]. Each establishment could easily offer as many as a few thousand resources. INS/Twine should therefore scale to $O(10^8)$ resources and around $O(10^5)$ resolvers (assuming each establishment could run at least one resolver). For this, each resolver should hold only a small subset of all resource information. Most importantly, only a small subset of resolvers should be contacted to resolve queries or to update resource information.

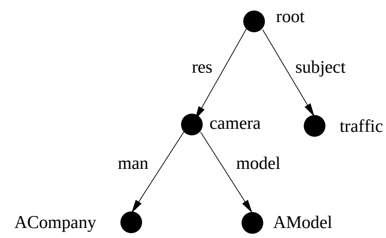
To achieve these goals, INS/Twine relies on an efficient distributed hash table process (such as Chord [15], CAN [14] or Pastry [16]), which it uses as a building block. Twine transforms each resource description into a set of numeric keys. It does so in a way that preserves the expressiveness of semistructured data, facilitates an even data distribution through the network, and enables efficient query resolution even in the case where queries contain a subset of attributes originally advertised by any resource. From a resource description, Twine extracts each unique subsequence of attributes and values. Each such subsequence is called a *strand*. Twine then computes a hash value for each strand, which constitutes the numeric keys.

A peer-to-peer approach to resource discovery creates new challenges for data freshness and consistency. Indeed, as resolvers fail or new ones join the network, the mapping from resource descriptions to resolvers changes. To maintain consistency in the face of network changes and resource mobility, resolvers treat all resource information as *soft state*. If a resource (or a proxy acting on its behalf) does not refresh its presence within a certain interval, the corresponding description is removed from the network. To achieve scalability, the refreshing frequency in the *core* of an INS/Twine network (i.e., among resolvers) is significantly lower than the refreshing frequency at the *edge* (i.e., between client applications and resolvers).

We evaluated INS/Twine by running 75 instances of the resolver and inserting various types of descriptions into the network. We find that both resource information and queries are evenly distributed among resolvers. Each resolver receives only a small subset of resource information and queries. The size of the set is proportional to the number of strands and resource descriptions but inversely proportional to the number of resolvers on the network. Resolvers associated with resource descriptions are located within $O(\log N)$ hops through the network, where N is the total number of resolvers. The query success rate is

```
<res>camera
    <man>ACompany</man>
    <model>AModel</model>
</res>
<subject>traffic</subject>
```

(a) Resource description



(b) AVTree

Fig. 1. Example of a very simple resource description and its corresponding AVTree. The resource is a camera, manufactured by ACompany and filming traffic

100% when less than $k-1$ resolvers (where k is a configurable replication level) fail or join the network within one refresh interval. Additionally, for a fraction F of failed resolvers, query failures decrease exponentially with k.

2 INS/Twine System Architecture

In this section, we first describe the details of resource descriptions. We then present the system architecture of INS/Twine and the algorithms for transforming resource descriptions into numeric keys, distributing information, and resolving queries.

2.1 Resource Descriptions

Resources in INS/Twine are described with hierarchies of attribute-value pairs in a convenient language (e.g., XML, INS name-specifiers [1], etc.). Our approach is to convert any such description into a canonical form: an attribute-value tree (*AVTree*). Figure 1 shows an example of a very simple resource description and its AVTree. All resources that can be annotated with meta-data descriptions, can be represented with an AVTree.

Each resource description points to a *name-record*, which contains information about the current network location of the advertised resource, including its IP address, transport/application protocol, and transport port number.

In INS/Twine, a resource matches a query if the AVTree formed by the query is the same as the AVTree of the original description, with zero or more truncated attribute-value pairs. For example, the device from Figure 1 would match the query: `<res>camera<man>ACompany</man></res>` or even the query: `<res>camera</res>`. This implies that an important class of queries that INS/Twine must support is *partial queries*, in addition to *complete queries* that specify the exact advertised resource descriptions.

Therefore, like resource descriptions, client queries are described using hierarchies of attribute-values and are converted to AVTrees. INS/Twine provides a way for queries to reach the resolver nodes best-equipped to handle them, based on the attributes and values being sought. The ultimate results of query matching depend on the local query processing engine attached to each resolver. Examples of this include INS's subtree-matching algorithm [1], UnQL [8] or the XSet query engine for XML [19].

Since query routing relies on exact matches of both attributes and values, it is possible to allow more flexible queries by separating string values into several attribute-value pairs. For example, <model>AModel Camcorder 123</model> could be divided into <modelw>AModel</modelw>, <modelw>Camcorder</modelw> and, <modelw>123</modelw>, allowing queries of type <modelw>123</modelw>.

2.2 Architecture Overview

INS/Twine uses a set of resolvers that organize themselves into an overlay network to route resource descriptions to each other for storage, and to collaboratively resolve client queries. Each resolver knows about a subset of other resolvers on the network.

Devices and users communicate with resolvers to advertise resources or submit queries. When a resource periodically advertises itself through a particular resolver, it is considered *directly connected* to that resolver. When a client issues a query to a resolver, it receives the response from that resolver. Communication between client applications and resolvers happens *at the edge* of the INS/Twine network. Communication between resolvers takes place in the network *core*.

The architecture of INS/Twine has three layers, as shown in Figure 2. The top-most layer, the *Resolver*, interfaces with a local AVTree storage and query engine, which holds resource descriptions and implements query processing, returning sets of name-records corresponding to (partial) queries.

When the Resolver receives an advertisement from its client application, it stores it locally using that engine. Local storage of information about directly connected resources serves for state management as described in Section 3. The resolver then splits the advertised resource description into strands and passes each one to the *StrandMapper* layer. The details of strand-splitting are discussed in Section 2.3. The StrandMapper maps the strand onto one or more numeric m-bit keys using a hash function. It then passes each key, together with the complete advertisement (the *value* corresponding to the key), to the *KeyRouter* layer. Finally, given a key, the KeyRouter determines which resolver in the network should receive the corresponding value and forwards the information to the selected peer. Hence, for data distribution, our approach boils down to inserting resource descriptions using each prefix-strand as a separate key.

As complete resource descriptions are transmitted during resource advertisement, any resolver specializing in a key computed from a query should be able to resolve the query without requiring any joins or extra data transfers. Hence, when a client submits a query, the resolver that first receives it randomly selects

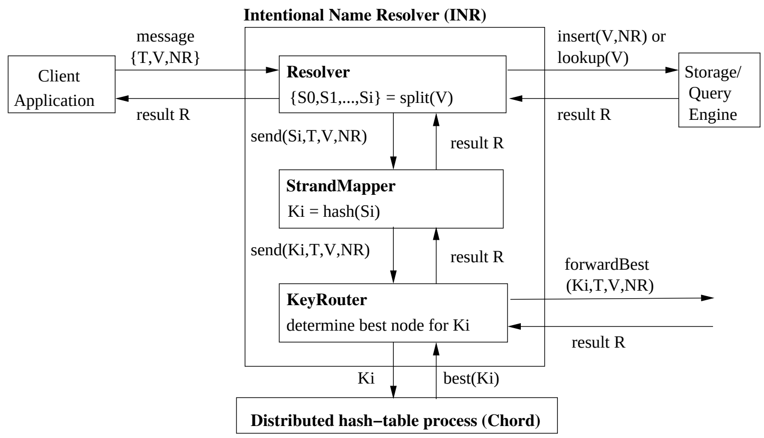


Fig. 2. An INR is composed of three layers. The Resolver receives messages from client applications. These messages include a type T (advertisement or query) an AVTree V and a name-record NR. The Resolver stores the description or performs the lookup locally. It then extracts all strands S_i from V, and for each one, the StrandMapper computes a key K_i. The KeyRouter finally maps each key onto an INR using a distributed hash table process like Chord. The message is then forwarded to that resolver. For queries, the resolver sends results back to the originating INR which in turn sends them to the application

one of the longest strands from the query AVTree. This strand serves to determine which resolver should solve the query. Query results are later returned to the originating resolver, which forwards them to the client.

A client application may also specify that it is interested in any resource matching a description. In that case, a single answer is returned. It is the resource that matches the given description and that has the lowest application-level metric. This feature comes from the original INS design [1].

Splitting resource descriptions into strands is critical to INS/Twine's ability to scale well. It enables resolvers to specialize in holding information and answering queries about a subset of resources. The choice of an adequate distributed hash table process at the KeyRouter layer is critical to achieving high query success rates while minimizing the number of resolvers contacted on each query. We discuss our choice in Section 2.5. The following sections describe each layer and its main algorithms in more detail.

2.3 Resolver Layer

At the core of an INS/Twine resolver is a strand-splitting algorithm that extracts strands from a description. The goal of the algorithm is to break descriptions into meaningful pieces so resolvers can specialize around subsets of descriptions. At the same time, the splitting must preserve the description structure and support partial queries.

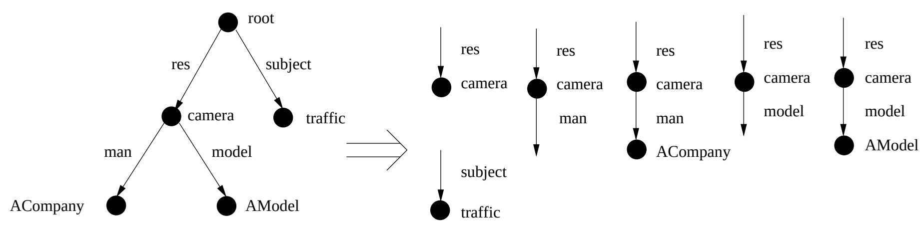


Fig. 3. Splitting a resource description into strands

A simple strand-splitting method would be to extract each attribute-value pair from the AVTree, and independently map it onto a key. However, this scheme would lose the richness of hierarchical descriptions and would not allow expressive queries to be performed. For example, *format* in the attribute sequence *printer-paper-format* would become indistinguishable from format in *video-cassette-format*.

The Twine algorithm for strand-splitting preserves the description structure and supports partial queries. It extracts each unique prefix subsequence of attributes and values from a description (advertisement or query) as illustrated in Figure 3. Each subsequence is called a *strand*. Each strand is then used to produce a separate key. For example:

```
Input strand: res-camera-man-ACompany
h1 = hash(res-camera)
h2 = hash(res-camera-man)
h3 = hash(res-camera-man-ACompany)
Output keys: h1, h2 and h3
```

We consider each top-level attribute-value pair as the minimal unit for resource descriptions and queries. We therefore omit strands composed of a single attribute and start strand-splitting after the first value. We believe that single top-level attributes would be too general to be useful in wide-area applications.

The Twine strand-splitting algorithm effectively extracts one strand for each attribute and each value in the AVTree, except for root attributes. Therefore, the number of strands depends on the number of attributes and values in the AVtree rather than its structure. More precisely, given a resource description with a attribute-value pairs, t at the root level, the total number of strands is given by:

$$s = 2a - t \tag{1}$$

With this scheme, partial queries are easily handled since each possible subsequence of attributes and values maps to a separate key, which in turn maps to a single resolver. The expectation of the storage requirement Z at each resolver is given by:

$$Z = \frac{(RSK)}{N} \tag{2}$$

where R is the number of resources in the system, S is the average number of strands per resource description, K is a configurable resource information replication level, and N is the number of resolvers in the network. This relation holds for $SK << N$.

Some strands such as `<resource>file server</resource>` may be extremely popular in resource descriptions. Advertisements can then overwhelm the node that is in charge of the popular strand. We tackle this problem by allowing each node to set a threshold (determined by the node's capacity) on the maximum number of resources for each key. When the threshold is exceeded for some key, no new resource is accepted under that key. The node could also start randomly replacing some entries with new advertisements. In both cases, this effectively renders that strand unusable for query purposes.

Since a query containing several strands is solved using one of the longest strands, the threshold restriction does not affect most queries. During query resolution, if a resolver returns an incomplete response (due to a threshold), a new strand is selected, and the process repeats until the response exhaustively lists all resources matching the description or no more strands are left in the query. In the latter case, the *partial* list of all accumulated matches is returned. It is flagged as being a partial answer, letting the application or user refine the query if necessary.

Additionally, in the rare case where a query containing only a very short and popular strand becomes extremely popular itself, edge resolvers may cache a few results to avoid flooding the node responsible for the strand. Caching is not yet implemented in INS/Twine.

2.4 StrandMapper Layer

Each strand extracted from the description is independently passed to the *StrandMapper* layer, together with the *complete* resource description or query. The StrandMapper is responsible for associating numeric keys with each strand. It does this by concatenating the attributes and values of the strand into a single string, and computing a 128-bit MD5 hash for the string.

2.5 KeyRouter Layer

The StrandMapper passes the key to the *KeyRouter* layer which uses it to determine which other resolvers should store information about the resource, or should participate in solving the query. The choice of an appropriate scheme for the KeyRouter is critical as it may easily become the limiting factor of INS/Twine's performance.

The KeyRouter may be thought of as a distributed hash table, where each node on the network keeps key-value bindings within a dynamically determined key range. Several efficient peer-to-peer algorithms have recently been proposed for this purpose: CAN [14], Chord [15], or Pastry [16]. Given a key, these systems find the node on the network that should store the corresponding value. Chord and Pastry are based on some variant of consistent hashing [20], where a node is responsible for all keys whose identifier falls between the node identifier and the closest preceding node identifier currently on the network. Hence only local disruptions occur when nodes join or leave the system. CAN uses a similar scheme, although not consistent hashing.

To achieve scalability, these systems require that each node only keeps in its routing table information about a subset of other nodes. This set is determined by the node unique identifier. Finding a node for a given key is then achieved by hopping from node to node in the appropriate direction, until the destination node is reached. This operation typically requires $O(\log N)$ hops, where N is the size of the overlay network of resolvers.

Our implementation is built on Chord, which efficiently rebuilds its overlay network in the presence of failures. INS/Twine uses Chord to efficiently identify which node, or set of k consecutive nodes, should store a given key.

3 State Management

The following consistency goals guided the design of the resource information management mechanisms in INS/Twine:

- When a resource joins a network, or moves or modifies its description, the update is propagated to the appropriate resolvers immediately. The new information replaces the old, ensuring that neither the old description nor the old location are ever returned as result of a query. While resource information propagates through the network, it is possible that both the old and new information be returned in response to a query.
- When a resource leaves or fails, its information is deleted at all resolvers.
- Query results are not affected by new resolvers joining the network, or by resolvers failing (up to a level of fault-tolerance determined by a configurable parameter k).

There are several ways to achieve consistency as defined above. If we use hard-state and require resources to always keep their information updated, we are not resilient to resources failing without prior de-registration.

Resolvers can also treat all resource information as *soft state*, requiring resources to refresh their information periodically throughout the network. If a resource does not refresh its presence within a certain interval, the corresponding description is removed from the network. A resource is free to leave the network at any time; if it does not de-register its description, the soft-state expiration mechanism will cause the resource description to be deleted. However, to keep information up-to-date, the refresh interval must be small, which imposes a high bandwidth overhead.

Periodically refreshing resource information also provides some degree of fault-tolerance by periodically sending each description not to one, but to $k > 1$ nodes per strand. This scheme relies on the capability of the underlying distributed hash table process (Chord [15] in our case) to rebuild the overlay network of interconnections as nodes join and leave. It also relies on the fact that

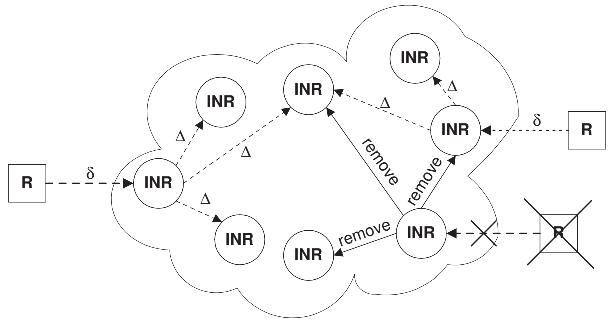


Fig. 4. State management in INS/Twine. Each resource R refreshes its information to a resolver INR at a high frequency defined by the refresh interval δ. Resolvers refresh resource information within the network at a lower rate defined by the refresh interval Δ. If a resource leaves the network without de-registering, the closest resolver detects the departure within δ and sends explicit remove messages to other nodes. Similarly when a resource updates its information. When a resolver crashes, corresponding resource information will remain in the network no longer than Δ

the set k is computed dynamically, so a failed node from the set is automatically replaced on subsequent resource advertisements.

We adopt a hybrid scheme in INS/Twine that combines the management simplicity of soft-state with the low-bandwidth requirements of hard-state. Figure 4 illustrates the approach. Each resolver at the edge of the network is responsible for resources that communicate directly with it. It acts as a proxy for these resources, keeping their states updated at the appropriate locations. Edge resolvers receive updates about the states of their resources at a fine time granularity δ (a few seconds in our implementation). They propagate any changes to appropriate resolvers. If a resource does not advertise itself within δ time units, an edge resolver assumes it has left the network and sends explicit remove messages to appropriate resolvers.

Resolvers in the network core also preserve soft-state, but they use a much longer period Δ (on the order of a few hours, for example). At every Δ time units, and for each directly connected resource, each edge resolver recomputes the set of resolvers in charge of that resource. It then transmits advertisements to every new resolver in a set. It transmits de-registration messages to resolvers no longer in a set. For resolvers that did not change, a simple ping is transmitted.

With this scheme, if a resource fails, it is de-registered from the network within δ time units. If a resolver acting on behalf of some directly connected resources fails, these resources can re-connect to another resolver while remaining available to clients. Finally, in the rare case where both the resolver and some directly connected resources fail, the resource information will be deleted from the network at most within Δ time units.

For increased fault-tolearance, each strand is mapped onto k replicas. Since we never transfer data between replicas, it is not guaranteed that each of the replicas knows about all resources matching any given strand. However, for up to $k-1$ failures within a refresh interval Δ, at least one of them does. Therefore queries are sent to all k nodes responsible for a given key, and the union of all results is returned to the client application. For larger number of failures within the interval Δ, the query failure rate becomes dependent on the *fraction* of failed resolvers. However, the query failure rate decreases exponentially with k.

Although we handle resolvers joining and leaving the network at any time, in case of network partitions and healing, information at different resolvers may become inconsistent. We currently do not handle this case, but we could use timestamps assigned by resources to their advertisements to determine which replicas hold the most recent information for each resource.

4 Evaluation

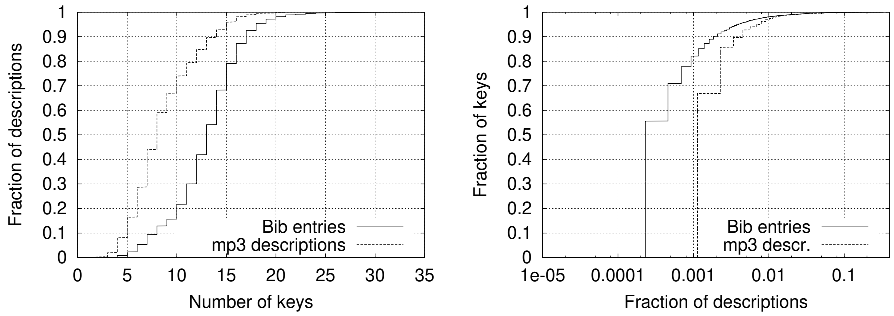
In this section, we first evaluate the strand-splitting algorithm by examining the distribution of strands from splitting real resource descriptions. We then examine how data and queries are distributed among resolvers. We finally evaluate the query success rate in the presence of failures.

4.1 Splitting Descriptions into Strands

In the first experiment, we evaluate the strand-splitting algorithm applied to real resource descriptions. Our goal is to determine how many strands are produced by such descriptions and how often the same strands come up. It is very difficult to obtain a large quantity of real resource descriptions. For our experiment, we used two sets of data. The first set contained 4318 bibliographical entries taken from latex bibliography files obtained from our own repositories as well as from Netlib [21]. The second set contained descriptions extracted from 883 mp3 files taken from our private collections. In both cases, we extracted word values from each string to enable hierarchical searches based on keywords in the title, author, and other fields. Although the data used does not describe devices, we believe it gives an intuition of the strand distribution that may appear in real data descriptions.

Figure 5(a) shows the strand distribution obtained from each data set. Bibliographical entries contain 12.9 strands on average, whereas mp3 tags produce an average of 8.7 strands. Although different resources will have descriptions of various complexity, it is interesting to note that splitting these real descriptions produced a reasonable number of strands. Figure 5(b) shows that, as expected, some strands are very popular - three strands come-up in over 10% of all resource descriptions for mp3 files. One of them is `<artist>boys</artist>`. Nine strands appear in over 10% of bibliographical descriptions, and three of them are in almost 30% of the entries. One example is `<type>article</type>`. However, this represents a very small fraction of all strands (less than 0.06%).

We evaluated INS/Twine on these two data sets as well as on workloads consisting of diverse synthetic descriptions. Our scheme mostly depends on the num-



(a) Number of strands in resource descriptions

(b) Frequency of identical strands in different descriptions

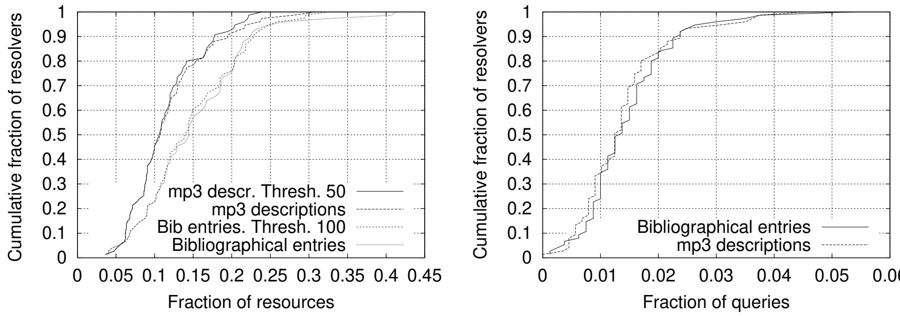
Fig. 5. (a) When splitting real descriptions into strands, most descriptions tend to contain a small amount of strands compared to our scalability goals of $O(10^5)$ resolvers. 80% of the mp3 descriptions produce 12 strands or less. 80% of the bibliographical entries generate 16 strands or less. The medians are 8 and 13 respectively. (b) Most descriptions are composed of unique strands. However, a few strands may appear in as much as one third of all descriptions

ber of strands in resource descriptions and not on the structure of the AVTrees. We therefore present only the results obtained for the real data sets.

4.2 Distributing Data among Resolvers

In the second experiment, we evaluated the quality of data distribution in INS/Twine. We ran 75 independent INR instances on 15 machines. We connected 10 client applications to 10 resolvers. Each client application inserted between 60 and 160 resource descriptions, for a total of 883 resources (from the mp3 set) and 7,668 strands. For comparison purposes, we also ran experiments on 800 of the bibliographical entries, inserted by 8 clients (100 resources per client). The main difference between the data sets is the number of strands in resource descriptions which is 30% lower on average for mp3 tags. At the Key-Router layer, each resolver advertised itself as 20 virtual nodes to create a more uniform distribution of resolvers throughout the whole key space. Since each experiment is deterministic given a set of resolvers and resources, each curve in Figure 6(a) presents the results from a single run. Different sets with the same number of resolvers give almost identical graphs.

INS/Twine dynamically specializes nodes around specific resource descriptions, so each peer stores only a small fraction of the complete directory. Figure 6(a) shows that over half of resolvers hold information about less than 15% of resources for both data sets.



(a) Resource information distribution

(b) Query distribution

Fig. 6. Cumulative distribution of resource information and queries in a network of 75 resolvers. Resource information is evenly distributed among resolvers. Increasing the proportion of number of strands to number of resolvers increases the fraction of resources known by each node. Queries are distributed evenly among all resolvers with a median of 1.2% queries per resolver, equal to $\frac{1}{N}$ where N is the number of resolvers. No resolver solves an excessive number of queries

The expected value for the fraction of resource information stored at each resolver is $\frac{(SRK)}{N} * \frac{1}{R}$, where S is the average number of strands in resource descriptions, R is the total number of resource descriptions, K is the configurable replication level, and N is the number of resolvers in the network. We compare the value from this theoretical model to the actual values obtained from the experiments. In the experiment, $K = 1$ and $N = 75$. For mp3 files, $S = 8.7$, so resolvers should know about 11.6% of all resource descriptions. This is indeed the average obtained in the experiment. The median is a little lower at 10.4%. For bibliographical entries $S = 12.9$ on average, so resolvers should know about 17.2% of all resources. The actual average and median are just slightly lower at 15.2% and 14.0%, respectively. When the number of strands is high compared to the network size, there is a higher probability that multiple strands from the same description get assigned to the same resolver. Overall, the experimental values match the model.

The long tails of the distributions are due to popular strands. To alleviate their impact, we imposed a node-based fixed threshold value for the number of resources accepted for any particular strand. We set the threshold at 50 resources for the mp3 data set and 100 resources for the bibliographical entries, since the latter contain significantly more strands than the former. In reality, these thresholds would be determined by the capacity of each node. Figure 6(a) shows that for both data sets, the overall distribution remains similar, while the tail gets significantly cut. The maximum amount of information known by any given node drops under 24% for the mp3 files and under 33% for the bibliographical entries. No node knows about more resources than twice the average.

Table 1. Query success rates function of replication level. Each resource was requested using a single randomly chosen strand, but the complete description was used for the actual lookup. Values shown are averages of three runs

Replication	Fraction failed nodes					
	0	0.013 (1 node)	0.027 (2 nodes)	0.067 (5)	0.13 (10)	0.27 (20)
None($k = 1$)	1.0	0.98	0.92	0.88	0.83	0.70
$k = 2$	1.0	1.0	0.96	0.95	0.95	0.94

Hence data is evenly distributed in INS/Twine with each resolver holding only a small subset of resource descriptions. Thresholds are also efficient at eliminating overly popular strands without changing the overall resource information distribution. Taking the city example from the introduction, with around 10^8 resources, and 10^5 resolvers, considering an average of 13 strands per description, and an additional replication level of 3, each resolver would need to know about $\frac{(SRK)}{N} = \frac{(13*10^8*3)}{10^5}$ or as few as $40*10^3$ resources.

4.3 Resolving Queries

Without failures, Twine finds any resource present in the network with the same performance as the underlying KeyRouter layer (Chord in our implementation). For all queries, $O(\log N)$ resolvers are contacted at the KeyRouter layer [15] to find the set of nodes associated with a given key. When replication is used, k resolvers then resolve the query in parallel.

To evaluate the distribution of queries among resolvers, we used 800 descriptions from each data set as queries. We submitted all queries through one randomly selected resolver. Figure 6(b) shows how queries were distributed among resolvers. The average fraction of queries solved by each resolver should be $\frac{1}{N}$ where N is the number of resolvers. For $N = 75$, this gives 1.3% of all resources. The distribution obtained shows that the queries are in fact evenly distributed with 80% of resolvers receiving less than 2% of the queries. No resolver solves significantly more queries than the average, since the maximum number of queries received by any resolver was just a little over 5%. We also find that the distribution is independent of the number of strands in resource descriptions since we obtain the same graph for both data sets.

Replicating each key onto $k > 1$ nodes allows Twine to support up to $k - 1$ nodes joining or leaving the system within an in-core refresh interval Δ. Table 1 shows the query success rate function of the fraction of failed resolvers. To show the worst case scenario, only one strand was randomly selected from each resource description to serve as query. The table shows that increasing the replication level improves success rates, as all replicas for a given key have to be down for the query to fail. For example, for 20 failed resolvers out of 75, and for $k = 2$, the probability to pick a strand whose replicas both map to a failed resolver is approximately $(\frac{20}{75})^2 = 7\%$.

The latency of query resolution (as well as resource information updates) is determined by the time taken by peers to exchange information. Therefore, INS/Twine latency and responsiveness will improve when proximity-based routing heuristics are used in the underlying key-routing system. For queries, since each description is replicated at several places in the network (at least one per prefix), there are many possible nodes that can resolve a query, and Twine may itself be able to choose a good node if it had information about network path latencies between nodes.

5 Conclusion

This paper described INS/Twine, a scalable resource discovery system using intentional naming and a peer-to-peer network of resolvers. The peer-to-peer architecture of INS/Twine facilitates a dynamic and even distribution of resource information and queries among resolvers. Central bottlenecks are avoided and results are independent of the location where queries are issued.

INS/Twine achieves scalability through a hash-based mapping of resource descriptions to resolvers. It manipulates AVTrees which are canonical resource descriptions. It does not require any a priori knowledge of attributes AVTrees may contain. Twine transforms descriptions into numeric keys in a manner that preserves their expressiveness, facilitates even data distribution and enables efficient query resolution. Resolver nodes hence dynamically specialize in storing information about subsets of all the resources in the system. Queries are resolved by contacting only a small number of nodes.

Additionally, INS/Twine handles resource and resolver dynamism responsively and scalably by using replication, by considering all data as soft-state and by applying much slower refresh rates in the core of an INS/Twine overlay network than at the edges.

INS/Twine scales to large numbers of resources and resolvers. Our experimental results show that resource information and query loads get evenly distributed among resolvers which demonstrates the ability to scale incrementally by adding more resolvers as needed.

Acknowledgments

The authors thank Godfrey Tan and Allen Miu for numerous helpful discussions. We thank Nick Feamster, Dina Katabi, Dave Andersen, and Chuck Blake for their comments on early drafts of this paper. This work was funded by NTT Inc. under the NTT-MIT research collaboration, by Acer Inc., Delta Electronics Inc., HP Corp., NTT Inc., Nokia Research Center, and Philips Research under the MIT Project Oxygen partnership, by Intel Corp., and by IBM Corp. under a university faculty award. Magdalena Balazinska is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1. Adjie-Winoto, W., Schwartz, E., Balakrishnan, H., Lilley, J.: The design and implementation of an intentional naming system. In: Proc. ACM Symposium on Operating Systems Principles. (1999) 186–201
2. Guttman, E., Perkins, C.: Service Location Protocol, Version2. RFC2608. http://www.ietf.org/rfc/rfc2608.txt (1999)
3. Sun Microsystems: Jini technology architectural overview. http://www.sun.com/jini/whitepapers/architecture.pdf (1999)
4. UPnP Forum: Understanding Universal Plug and Play: A white paper. http://upnp.org/download/UPNP_UnderstandingUPNP.doc (2000)
5. Czerwinski, S., Zhao, B., Hodes, T., Joseph, A., Katz, R.: An architecture for a Secure Service Discovery Service. In: Proc. of the Fifth Annual Int. Conf. on Mobile Computing and Networking (MobiCom), ACM Press (1999) 24–35
6. Castro, P., Greenstein, B., Muntz, R., Bisdikian, C., Kermani, P., Papadopouli, M.: Locating application data across service discovery domains. In: Proc. of the Seventh Annual Int. Conf. on Mobile Computing and Networking (MobiCom). (2001) 28–42
7. Hermann, R., Husemann, D., Moser, M., Nidd, M., Rohner, C., Schade, A.: DEAPspace – Transient ad-hoc networking of pervasive devices. In: Proc. of the ACM Symposium on Mobile Ad Hoc Networking & Computing (MobiHoc). (2000)
8. Abiteboul, S.: Querying semi-structured data. In: ICDT. Volume 6. (1997) 1–18
9. Mockapetris, P.V., Dunlap, K.J.: Development of the Domain Name System. In: Proc. of the ACM SIGCOMM Conference, Standford, CA (1988) 123–133
10. Yeong, W.: Lightweight Directory Access Protocol. (1995) RFC 1777.
11. Lilley, J.: Scalability in an intentional naming system. Master's thesis, Massachusetts Institute of Technology (2000)
12. Castro, P., Muntz, R.: An adaptive approach to indexing pervasive data. In: Second ACM international workshop on Data engineering for wireless and mobile access (MobiDE), Santa Barbara, CA (2001)
13. Gnutella: website. http://gnutella.com/ (2001)
14. Ratnasamy, S., Francis, P., M.Handley, Karp, R., Shenker, S.: A scalable content-addressable network. In: Proc. of the ACM SIGCOMM Conference, San Diego, CA (2001) 161–172
15. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for Internet applications. In: Proc. of the ACM SIGCOMM Conference, San Diego, CA (2001) 149–160
16. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: IFIP/ACM Middleware, Heidelberg, Germany (2001)
17. W3C: Web Services activity. http://www.w3.org/2002/ws/ (2002)
18. U.S. Census Bureau: United States census 2000. http://www.census.gov/ (2002)
19. Zhao, B., Joseph, A.: Xset: A lightweight database for internet applications. http://www.cs.berkeley.edu/~ravenben/publications/saint.pdf10 (2000)
20. Karger, D., Lehman, E., Leighton, T., Levine, M., Lewin, D., Panigrahy, R.: Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In: Proc. of the 29th Annual ASM Symposium on Theory of Computing. (1997) 654–663
21. Netlib: Netlib repository at UTK and ORNL. http://www.netlib.org/ (2002)