Superstring: A Scalable Service Discovery Protocol for the Wide-Area Pervasive Environment

Ricky Robinson, Jadwiga Indulska
School of Information Technology and Electrical Engineering
The University of Queensland
{ricky, jaga}@itee.uq.edu.au

***Abstract*— Arguably, the world has become one large pervasive computing environment. Our planet is growing a digital skin of a wide array of sensors, hand-held computers, mobile phones, laptops, web services and publicly accessible web-cams. Often, these devices and services are deployed in groups, forming small communities of interacting devices. Service discovery protocols allow processes executing on each device to discover services offered by other devices within the community. These communities can be linked together to form a wide-area pervasive environment, allowing processes in one group to interact with services in another. However, the costs of communication and the protocols by which this communication is mediated in the wide-area differ from those of intra-group, or local-area, communication. Communication is an expensive operation for small, battery powered devices, but it is less expensive for servers and workstations, which have a constant power supply and are connected to high bandwidth networks. This paper introduces Superstring, a peer-to-peer service discovery protocol optimised for use in the wide-area. Its goals are to minimise computation and memory overhead in the face of large numbers of resources. It achieves this memory and computation scalability by distributing the storage cost of service descriptions and the computation cost of queries over multiple resolvers.**

***Index Terms*— peer-to-peer, pervasive computing, resource discovery, service discovery**

I. Introduction

Service discovery is one of the central components of any pervasive computing environment [9]. Service discovery protocols allow services (which may be highly specialised devices such as sensors and web cams, or software services and resources such as online shops and Business-to-Business applications) to advertise themselves, and clients to issue queries against these service descriptions. In this way, clients can discover instances of services that can perform a task the client wishes to carry out. Service discovery protocols are important in pervasive systems because clients do not have a priori knowledge of services. Also, pervasive environments tend to be highly dynamic, with clients and services appearing and disappearing from the system.

Several service discovery protocols have emerged in recent times, often as a constituent in a more general platform for service discovery *and* utilisation. Bluetooth SDP [6], the Simple Service Discovery Protocol (SSDP) [11] and the discovery mechanism of Salutation [18] are examples of such discovery mechanisms. Whilst each of these address the issue of the extreme dynamism of pervasive environments, these protocols do not scale to the wide-area. Rather, they are intended for use within relatively small communities of devices. Some of these protocols, SSDP is an exception, attempt to minimise communication overhead, acknowledging the high costs often associated with the combination of wireless communication and small, battery powered devices. Whilst the cost of communication is of primary concern in small groups of devices communicating over wireless links, this cost is of less concern in the wide-area, where much of the cost is absorbed by highly capable servers, connected to high bandwidth networks. In these wide-area environments, computation costs and memory overhead are often as important, or more so, than communication costs. This is evidenced by attempts to curb routing table size in traditional network protocols such as IP [10]. To achieve high performance, these tables are kept in main memory, which is expensive. Service discovery protocols face a similar problem when the number of advertised resources is vast, as is the case in wide-area networks. At the cost of higher query latency, service descriptions can be moved to secondary storage or distributed across many machines if a high bandwidth network is available. If the second of these options is taken, then the work of processing any given query will be shared among multiple machines. This is of benefit when some queries to be more popular than others, since under this scheme, no single resolver will bear the cost of processing these numerous queries.

INS/Twine [4] and Berkeley's Secure Service Discovery Service (SSDS) [8] are two relatively new protocols for wide-area service discovery. Both of these protocols attempt to lower the communication cost involved with queries, at the expense of higher advertisement costs. However, in many circumstances, it is beneficial to minimise the advertisement costs in terms of communication and memory overhead, and to lower query computation costs for any single resolver, at the cost of slightly more communication overhead for queries, which can be achieved as outlined above.

This paper describes Superstring, a service discovery protocol for wide-area pervasive environments (environments in which highly dynamic groups of devices and services are connected by more static and more capable infrastructure such as the Internet). Superstring shifts much of the cost of querying and advertisement to the fixed infrastructure, and minimises memory and computation overhead in this infrastructure. The remainder of this paper is as follows. Section II describes related work on service discovery protocols for the wide-area.

The work reported in this paper has been funded in part by the Co-operative Research Centre Program through the Department of Industry, Science and Tourism of the Commonwealth Government of Australia.

Section III introduces Superstring, and explains how it achieves low memory and computation overhead. In Section IV, Superstring is formally compared to Twine in terms of communication overhead. A brief discussion of our future work takes place in Section V and Section VI concludes the paper.

II. Related Work

Twine is one the most recent research efforts in the area of service discovery. It builds on previous work from MIT. Twine is integrated with INS, the Intentional Naming System [1]. In INS, resources and services are named for their function rather than their location. In other words, a resource name is a service description, and these descriptions are hierarchical attribute-value trees. Twine routes service descriptions and queries using a distributed hashtable (DHT) mechanism. In theory, any of a number of existing DHTs could be used, including Pastry [16] and CAN [14]. In practice, the Twine implementation utilises Chord [17], which is also from MIT. Clients and services are said to exist at the edge of the network. The core is composed of a peer-to-peer network of Twine resolvers. These nodes consist of three layers: the *Resolver*, the *StrandMapper* and the *KeyRouter*. The Resolver layer stores a map of keys to service descriptions. It also passes *strands* extracted from service descriptions and queries to the StrandMapper layer. The StrandMapper computes a key from each strand it is given. These keys, along with the entire service description or query, are then passed to the KeyRouter layer. The KeyRouter is the distributed hashtable layer, responsible for routing between Twine nodes. Each message can be routed to the relevant node within $logN$ hops, where N is the number of resolvers.

Upon deployment, a service contacts its nearest Twine resolver. The resolver stores the description locally, and extracts each unique subsequence (from the root to each leaf, as well as each unique prefix along that path from root to leaf) from the hierarchical description. A key is computed from each of these strands. The key determines which resolvers on the network will store a copy of the service description. A client issues queries to its nearest resolver. A query takes the same form as a service description, but it may only partially match the advertised service descriptions. The longest strand from the query is selected, and its hash computed. The resulting key determines which resolver on the network can resolve the query. The query is then routed to the appropriate resolver.

Twine uses a soft state mechanism to keep service descriptions fresh, and to avoid situations where the service has disappeared but its description remains in the network. Twine can achieve fault tolerance by sending each key-service description pair to k nodes, and by routing queries to those k nodes.

The Secure Service Discovery Service (SSDS) from Berkeley is part of the Ninja project, the goal of which is to build large scale services from small devices. Unlike the resolvers in Twine, SSDS resolvers form a hierarchy. The hierarchy can be based on any arbitrary property, but usually the property chosen is location. Several hierarchies may co-exist, with each resolver being a member of multiple hierarchies. If an SSDS server finds it is being overwhelmed with queries and advertisements, it can delegate a portion of its domain to a child server. For example, if the hierarchy is location based, and a resolver is serving an entire building, it can delegate child resolvers for each floor of the building. A question now arises about the mechanism by which advertisements are propagated and queries are routed between the SSDS servers in this hierarchy. SSDS utilises Bloom filters to aggregate service descriptions in a lossy fashion. Each resolver contains a bit vector for its own service descriptions and bit vectors for each of its children. It ORs these vectors together and passes it to its parent. This process continues to the root node. Queries are hashed and this hash is checked against the resolver's own bit vector and that of its children. If all bits set in the query hash are set in the local vector, the query can be resolved locally. If there is a match for any of the child vectors, the query is sent down to the matching child. Otherwise, the query is passed up to the parent. An artifact of the Bloom filter mechanism is that it allows false positive results, but no false negatives. This results in needless forwarding of queries, but maintains correctness. As the number of services in the system increases, the more likely it is that false positives occur.

VIA [7] provides a mechanism by which resolvers organise themselves into a hierarchical cluster based on some specific application. Initially, a resolver receives all queries by listening to the multicast channel. If the resolver decides it is receiving too many irrelevant queries (queries for resources that it knows nothing about), the resolver may elect to "get behind" another resolver. That is, the resolver becomes the child of another resolver. In this fashion, a cluster of resolvers may be formed. The top level resolver must receive all queries, but it only passes relevant queries to its children. Therefore, a subset of the resolvers in the system (all the top level nodes) are involved in query resolution for all queries. If the cluster is specific to MP3 files, then the top level resolver filters out any queries that are unrelated to MP3s and passes other queries to its children. The next level of the hierarchy might filter on the artist's name. Each child resolver is responsible for a particular value of the "artist" attribute. For example, one may filter for *artist name='John Lennon'* whilst another child filters for *artist name='Khaled'*. Much of VIA's overhead comes from maintaining these hierarchies. Thus, a VIA resolver must offset the cost of receiving irrelevant queries against the cost of becoming part of a hierarchy, and maintaining its position in the hierarchy. VIA also allows the formation of a hierarchy of clusters. Again, there is a tradeoff between the cost of processing irrelevant queries, and the cost of maintaining the cluster hierarchy.

III. Superstring

Superstring is a service discovery protocol designed for environments where the services are extremely dynamic, and where queries are resolved in the core of the network, which is relatively static and composed of capable servers and high-bandwidth networks. An example of such an environment is where groups of measuring instruments are deployed in the field, and only have intermittent connectivity to an orbiting satellite or an aircraft circling the deployment zone. The client requires readings from some of these instruments and is located some distance from the deployment zone. Such scenarios are common in scientific investigations of geological phenomena and biological ecosystems [5]. There are also military scenarios which fit this description [12]. In such environments, inter-

mittent service is a common characteristic, often brought about by inclement weather conditions and other factors such as intermittent line of site to a base station. Superstring can also be used for discovering highly dynamic services or resources on the Internet.

As Twine's designers argue, peer-to-peer networks can potentially scale to larger numbers of nodes than hierarchical networks in which the root node may become a bottleneck. However, peer-to-peer protocols have their own problems. For example, Gnutella has been shown to scale poorly because of its flooding based protocol [15]. Recently, the idea of distributed hashtables (DHTs) has been put forward by several research groups. DHTs create a peer-to-peer overlay network that, given a key, can find the node at which the corresponding value is stored within $logN$ hops, where N is the number of nodes in the network. If objects are uniquely identified by a name, the name can be hashed, yielding a key for that object. If a service discovery protocol were to be built on top of a DHT, what can be used as the key for the service description? A service description is inherently more complex than a name or object identifier. It may be a hierarchy of attributes and values. Also, a query may be much smaller than a description that matches it. Queries need not contain every attribute contained in descriptions, and values may be wildcarded. An implication of this is that obtaining a single key by hashing the entire service description is not a valid option since very few queries will match the service description exactly. Twine solves the problem by computing several keys from a single service description. If S keys are produced from the description, then the service description is stored at S resolvers. A single key is extracted from a query, and this is used to route the query to the resolver responsible for that key. The query is processed at that resolver. Each message is routed to the resolver within $O(logN)$ hops. Therefore, queries can be performed relatively cheaply in terms of the number of resolvers contacted. But Twine's storage costs are high, and it is likely that some resolvers will process a much larger proportion of queries than others. This is due to the fact that, in most circumstances, queries are not normally distributed. If queries in the pervasive environment bear any resemblance to queries issued to web search engines, the distribution of query terms will be closer to a power-law distribution [2]. It cannot be assumed that queries will be targeted evenly at the various kinds of resources and services available. Undoubtedly, some services will be far more popular than others. For Twine, this means that some resolvers will process a substantially larger amount of queries than other resolvers, and this cost is not shared. Superstring acknowledges this power-law distribution of service types and queries, and seeks to minimise the burden on any single node. It distributes the cost of storage and query resolution among several nodes. While this set of nodes will collectively process more queries than other nodes in the system, a single resolver is not burdened with this cost. However, this comes at the price of extra communication overhead, since queries must be broken up and sent to a larger number of nodes.

Superstring utilises a DHT data structure, but does so in a way that optimises memory and computation overhead at the expense of potentially larger communication requirements than Twine. Upon deployment, a service sends its service description to its nearest Superstring resolver (note that this resolver may be on the same device as the service). The resolver produces a key from the top-level component of the description. This key is used to route the entire description to the resolver responsible for that key. When the description arrives at this resolver, the resolver removes the top-level component from the service description. This operation may create several sub-trees from the description. The resolver hashes each of the top-level components of the sub-trees and passes the sub-tree to one of its neighbouring resolvers. It stores the key for each sub-tree with the ID of the neighbour to which the sub-tree was passed. This process continues until the bottom of the description is reached. Each node in the hierarchy stores the name record for the service being advertised. In this fashion, a resolution hierarchy is formed that matches the service description hierarchy exactly. The query process is similar. The difference is that depending on its detail, a query need not visit every node in the hierarchy, and query results are propagated back up the tree. Query results consist of the name records of any matching service descriptions. Each internal node computes the intersection of the name record sets it receives from its children. In this way, when the results reach the top of the tree, only the name records for matching services remain. The query is resolved as part of the routing process. No resolver in the hierarchy solves the entire query (unless the query is very general and consists only of the top-level component of the description).

Figure 1 shows the cost of advertising a simple service in Superstring in terms of the number of resolvers contacted. The node responsible for the top level attribute of the description is found in $O(logN)$. The remaining cost is due to creating the hierarchy. In general, if the description has C attributes and values altogether, then the cost is $C - 1 - V$ where V is the number of values (leaf nodes).

The cost of querying in terms of resolvers contacted is at least $O(logN)$ (for the most general of queries). The cost increases for more complex queries with numerous attribute components and values. The next section provides a comparison of INS/Twine and Superstring with respect to communication, memory and computation costs.

IV. Performance

In this section, we compare the performance of Superstring to Twine. Twine is chosen for comparison because its goals are similar to that of Superstring, and, in our view it represents the most scalable resource discovery protocol to date since it has been shown via mathematical analysis to scale to an environment of 100 million resources and 100 thousand resolvers. Twine has a completely flat peer-to-peer topology. Superstring combines peer-to-peer and hierarchical topologies. It utilises a flat topology to discover top-level nodes that specialise in a particular kind of service. From this top-level node, a hierarchy is created which reflects the hierarchical structure of service descriptions. Queries are resolved by the specialised hierarchy. In some respects, this is similar to a process employed by VIA, however, Superstring does not incur costs for managing the hierarchy. Queries are filtered by the hierarchy, and name records for matching services are returned to the top-level

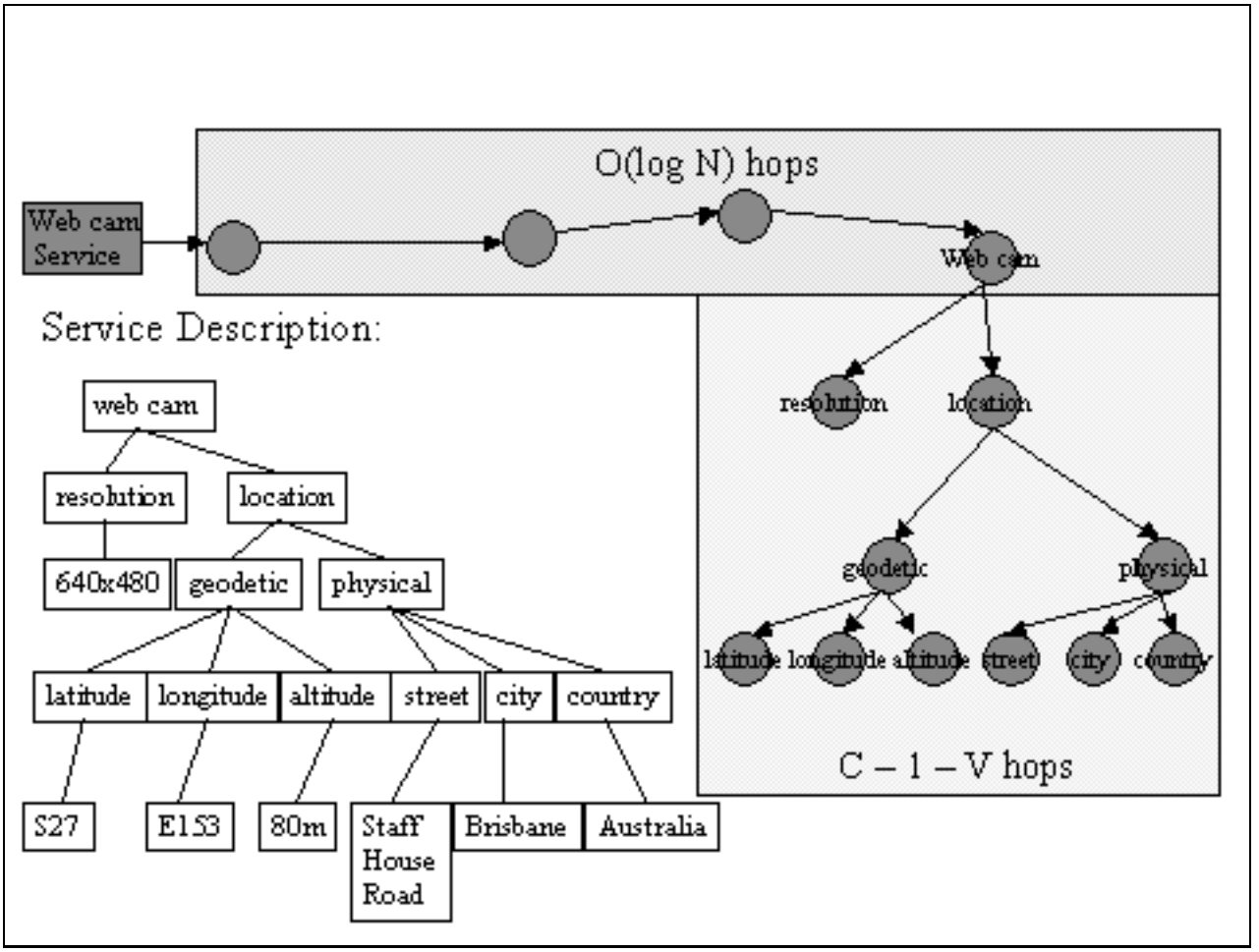

Fig. 1. Advertising a web cam

resolver where results are collated and returned to the client. Highly detailed queries may require each resolver in the hierarchy to perform some computation.

As stated, the cost in terms of the number of resolvers contacted during advertisement is $O(logN)$, plus the number of attribute components in the service description being advertised. If there are many leaf nodes in the description, then advertisement is cheaper in Superstring than it is in Twine. In general, the proportion of leaf nodes to internal nodes is quite high. For example, in a balanced binary tree, half the nodes are leaves.

Twine descriptions cannot be directly compared with Superstring descriptions. In Twine, a description is composed of attribute-value *pairs*. The attribute names correspond to the branches of the description tree, and the values correspond to the nodes. Therefore, the total number of *components* in the tree is $2A$ where A is the number of attributes. Superstring also utilises a hierarchical description model, but it is not an attribute-value tree. Rather, all internal nodes are attribute components, and all the leaves are values. An attribute is identified by the composition of attribute components from the root to a leaf. The number of attribute components in the description is given by $C - V$ where C is the total number of components in the description, including values, and V is the number of values. For Twine,

$$C = 2A \tag{1}$$

The number of strands produced from a Twine description is

$$S = C - T \tag{2}$$

where T is the number of attribute-values at the root level. Thus, for advertisement, the number of resolvers contacted is bounded by $O(SlogN)$. For Superstring, the number of resolvers contacted during discovery is bounded by

$$O((logN) + C - 1 - V) \tag{3}$$

To determine the circumstances under which Twine and Superstring perform equally well for advertisements, the above expressions can be equated and $logN$ subtracted from both sides. Then their costs are equal when

$$(S - 1)logN = C - 1 - V \tag{4}$$

Substituting for S gives

$$(C - T - 1)logN = C - 1 - V \tag{5}$$

Note that if a description has T attribute-values at the top level, then it must have at least T leaf nodes. Thus, $V \geq T$ always holds true. Advertising is cheapest for Twine when the

advertisement yields only a single strand. In this case, $V = T$. $V = T$ when $C = 2$. That is, when the description consists of one attribute component and a single value. In this case, both Twine and Superstring are bound by $O(logN)$. Therefore, Superstring can not be outperformed by Twine in terms of the number of hops required during advertisement. In most cases, however, descriptions will consist of more than just a single attribute and value. In these cases, Superstring always outperforms Twine, and this decreased cost is considerable if $V \gg T$.

The situation is different for queries. For Twine, the number of resolvers contacted is always bounded by $O(logN)$. For Superstring, it depends on the level of detail in the query, and whether the query is resolved positively (matching service found) or negatively (no matching service). In the worst case, the query cost is the same as advertisement cost, since it is possible that every branch of the tree must be traversed for a highly detailed and positively resolved query. The communication cost for querying is also higher than Twine in this case since results (name records) are propagated back up the the tree. Twine will outperform Superstring for queries by up to $C - 1 - V$ resolvers, if a query contains as much detail as the description or descriptions it matches. If the number of resolvers contacted is the only consideration in choosing one resource discovery protocol over another, then the choice comes down to the dynamism of the environment, and the frequency with which service descriptions need to be refreshed. In highly dynamic networks, the refresh interval must be set low. In this case, it is entirely possible that advertisement messages for a particular service are more frequent than queries that match that advertisement. In this case, Superstring has an advantage over Twine. The ideal environment for Superstring is one in which groups of highly dynamic devices and services are linked by a fairly static, high bandwidth network such as the Internet backbone. In this situation, resolvers only rarely fail or disconnect, since they form part of the static core. In contrast, the end nodes (services) disconnect and move regularly. The refresh interval is set low to minimise the probability of false positive query results. The ratio of queries to advertisements (and description refresh messages) approaches parity. Most of the communication takes place over high bandwidth links where the cost of communication is low. Communication costs can be reduced by caching the results of common queries on a short term basis.

The number of resolvers contacted is not the only consideration for a wide-area resource discovery protocol. Indeed, memory requirements and computation costs can often be just as important. As the pervasive environment widens to encompass an ever larger geographical area, the number of services available for discovery in that area increases. Descriptions for these services must be stored at resolvers. It is of benefit if the cost of storing these descriptions can be limited, and the computation costs involved with processing queries can be reduced for each resolver. The hierarchical structure of Superstring ensures no single resolver bears the cost of a popular query, and does so in a fashion that minimises memory overhead. To begin with, a single service description is distributed over several nodes. Thus, each resolver stores only a small part of the entire description. Twine, on the other hand, requires that S nodes store a copy of the entire description. Then, when a query is routed to one of those S resolvers, the query must be processed against all descriptions contained on that node. In Superstring, no resolver processes the entire query. Each node in the resolver hierarchy performs a lookup (which can be performed in logarithmic time) of the next attribute component in the description. The result is the address of a child resolver to which the remainder of the query should be forwarded. If the lookup yields no child, then the query fails, and an empty result is returned to the parent. Therefore, while the cumulative computation cost for any query is exactly the same as for Twine, the cost is shared among many resolvers.

Aside from these performance comparisons, Superstring has an advantage over Twine in terms of the flexibility of the query language. Upon receiving a query from a client, a Twine resolver will select the longest strand from the query and hash it to yield a key. The key determines which resolver will process the query. The strand will be comprised of a series of attribute-value pairs. Although Twine offers the possibility of issuing queries containing wildcards, indicating "don't care" for some value, it is impossible to issue queries containing relational functions such as inequalities. The presence of relational expressions in queries means that a hash of the strand containing this expression will not yield the key that is mapped to the appropriate resolver. The presence of a relational operator such as occurs in the query *resource* $\rightarrow$ *printer* $\rightarrow$ *resolution* $\rightarrow \geq$ *100dpi*, prevents the strands from hashing to the same key. Twine's attribute-value tree structure for descriptions and queries makes it hard to find a solution to this problem. Superstring, recall, stores all values at the leaves of the tree. To enable a richer query language, values are hashed neither in the advertising process nor in the querying process. Thus, queries may contain complex expressions in place of constant values.

V. Ongoing and Future Work

We are currently developing a range of different service discovery models. Superstring is the most mature of the models developed so far. Superstring takes into account that queries will usually not be distributed normally over the services that they match. Rather, some services will be matched far more often than others due to their popularity. Furthermore, over extended periods of time, some queries will be issued very frequently, whilst many queries will be issued infrequently. In almost all existing resource and service discovery protocols, this property means that a few resolvers will perform far more work than most other resolvers. Superstring acknowledges this, and seeks to distribute the computational overhead among a set of resolvers. This is a simple example of the use of complex systems theory in service discovery, but complex systems theory may yield several other enhancements to service discovery in the future. Specifically, the models we are developing make use of complex and biological systems theory. Already, there are some research groups [3, 13] making use of complex systems theory in peer-to-peer file sharing applications. We are attempting to extend the use of complex systems theory to the realm of service discovery in pervasive environments. We are also investigating the use of biological phenomena in service discovery. Ant communities, for example, exhibit properties of self-organisation as they forage for food resources. Similar

ideas can be used to build a service discovery protocol. However, so far our biologically based service discovery protocol is unable to offer the same guarantees that Superstring offers with respect to never returning false negative results in a scalable manner. Our biologically based algorithm can give this guarantee only if every node is visited. Finally, we have had some success in combining a biological approach with an approach similar to Superstring. This system can provide the same guarantees as Superstring but is more efficient with respect to communication overhead. It uses the notion of stigmergy (a method of indirect communication used by ants and other insects) to optimise query performance. Preliminary analysis suggests that this final model will form the basis of our future work into wide-area service discovery protocols.

Concurrently with this work on core service discovery algorithms, we are investigating ways in which to augment these algorithms to allow scoped discovery. Peer-to-peer protocols, such as Twine and Superstring, necessarily operate within a flat routing space, but such a topology does not accurately reflect the true nature of service organisation. Often, service advertisements and queries should be localised within a particular group or domain. This domain might reflect organisational boundaries or any other attribute relevant to a particular system of services. Local printers, for instance, should not be advertised outside of the local domain, and queries for them should be restricted to the local group. If such a service were advertised in Twine, information about the printer would be propagated to resolvers throughout the city. A query that matches the printer might be resolved by a resolver on the other side of the city. These kinds of advertisements and queries should be scoped to the local group of services, whilst other queries should be propagated to the world. This needs to be achieved without requiring an entirely separate algorithm and address space. Moreover, clients should be able to discover the nearest service within a constrained scope. Taking the example of a printer, a user would optimally like to discover the physically nearest printer that satisfies the issued query. If the nearest printer is off line or broken, the search should expand to a wider area (perhaps including printers on the floors above and below the user's floor). If location is part of the service description, then it is viable that such a system could be built by specifying an expression in the query instead of specifying exact values. For example, if the user provides the widest acceptable scope, the application could build an expression containing a list of locations in ascending order of physical distance from the user, instead of limiting the query to one specific location (returns non-optimal results in the case of failure of the printer) or not limiting the scope of the query at all (returns far too many query results). Such additions to the service discovery protocol enhances both its scalability and usability.

VI. Conclusion

We introduce Superstring, a service discovery protocol optimised for environments consisting of highly dynamic groups of services connected by more capable routing infrastructure. The set of highly dynamic devices and services such as mobile phones, laptops, hand-held computers, sensors and web-cams and the more static Internet backbone is such an environment. Superstring trades computational overhead for communication overhead, with the view that such communication overhead can be tolerated by capable nodes. In highly dynamic environments where services come and go with a high frequency, the proportion of advertisements to queries is high. Therefore, Superstring optimises advertisement costs at the expense of higher costs for queries. Superstring also acknowledges that certain kinds of services will be more popular than others. As such, certain queries are issued more frequently than others meaning that some resolvers will be required to do more work than others. Superstring distributes this extra workload over multiple resolvers so that no single resolver is overly burdened.

References

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *17th ACM Symposium on Operating Systems Principles (SOSP 99)*. M.I.T. Laboratory for Computer Science, December 1999.

[2] Amanda Spink, Dietmar Wolfram, B. J. Jansen, and Tefko Saracevic. Searching the Web: The Public and Their Queries. *The Journal of the American Society for Information Science and Technology*, 52(3):226–234, 2001.

[3] Özalp Babaoglu, Hein Meling, and Alberto Montresor. Anthill: A Framework for the Development of Agent-Based Peer-to-Peer Systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, 2002.

[4] Magdalena Balazinska, Hari Balakrishnan, and David Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *Pervasive 2002 - International Conference on Pervasive Computing*, number 2414 in LNCS, pages 195–210. Springer-Verlag, August 2002.

[5] W. A. Birkemeier, C. E. Long, and K. K. Hathaway. DELILAH, DUCK94, SandyDuck: Three Nearshore Field Experiments. In *Proceedings 25th International Conference on Coastal Engineering*, Orlando, FL, 1997.

[6] Bluetooth SIG. Bluetooth Specification version 1.1, February 2001.

[7] Paul Castro, Benjamin Greenstein, Richard R. Muntz, Parviz Kermani, Chatschik Bisdikian, and Maria Papadopouli. Locating application data across service discovery domains. *Mobile Computing and Networking*, 2001.

[8] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An Architecture for a Secure Service Discovery Service. In *Fifth Annual International Conference on Mobile Computing and Networks (Mobicom '99)*, 1999.

[9] Mike Esler, Jeffrey Hightower, Tom Anderson, and Gaetano Borriello. Next century challenges: Data-centric networking for invisible computing. In *Mobicom*, 1999.

[10] V. Fuller, T. Li, J. Yu, and K. Varadhan. RFC 1519: Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy. IETF Internet Standard, September 1993.

[11] Yaron Y. Goland, Ting Cai, Ye Gu, and Shivaun Albright. Simple Service Discovery Protocol/1.0. IETF draft specification, October 1999.

[12] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Emerging Challenges: Mobile Networking for 'Smart Dust'. *Journal of Communication and Networks*, 2(3):188–196, September 2000.

[13] Lada A. Adamic, Rajan M. Lukose, Amit R. Puniyani, and Bernado A. Huberman. Search in power-law networks. *Physical Review E*, 64, 2001.

[14] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. In *Proceedings of ACM SIGCOMM 2001*, pages 161–172, San Diego, CA, 2001.

[15] Jordon Ritter. Why Gnutella Can't Scale. No, Really. Technical report, Darkridge Security Solutions, http://www.darkridge.com/ jpr5/doc/gnutella.html, 2001.

[16] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, number 2218 in LNCS, pages 329–350, Heidelberg, Germany, November 2001.

[17] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01*. MIT Laboratory for Computer Science, August 2001.

[18] The Salutation Consortium. Salutation architecture specification (part 1) v2.0c. Specification, The Salutation Consortium, June 1999.