An Adaptive Approach to Indexing Pervasive Data

Paul Castro and Richard Muntz
Department of Computer Science
University of California, Los Angeles
{castrop, muntz}@cs.ucla.edu

ABSTRACT

In a pervasive computing world data will be scattered among millions of devices and finding the right data will be a fundamental problem. Several proposed service discovery architectures can assist applications searching for data within local boundaries but there is currently no support for global access to data. We introduce an application-level protocol VIA* for building self-organizing, distributed, hierarchical data indices that adapt to dynamic query workloads. These indices efficiently route queries to relevant devices and reduce the overall workload of the system. Adapting to the query workload, VIA* uses a "query impedance" metric to approximate the optimal hierarchy for processing the expected query workload. Distributed, "logical" nodes in the interior of the hierarchy collect information about query impedance and forward this information to "data carrying" leaf nodes that react to improve the topology of the hierarchy. We present some findings from our workload testbed that demonstrate the performance and scalability characteristics of our approach and outline our research agenda

Keywords

Pervasive computing, adaptive index, data dissemination.

1. INTRODUCTION

In a pervasive computing world, data will be scattered on millions of devices in the environment and finding the right data will be a fundamental problem. Centralized solutions are not practical; data will be mobile, dynamic, and possibly noisy. Applications, regardless of their physical location, will need access to data that can be anywhere.

Consider a photographer who uses a PDA outfitted with a digital camera to take pictures while she wanders around. Rather than store the pictures on her PDA s limited amount of memory she chooses to off-load the images via a wireless network to nearby storage elements she discovers in the environment. Later, she would like to retrieve all the images she has taken. If she maintains a history of all the storage devices she used then she can query just the relevant storage elements. However, relying on a mobile client to maintain a history may not possible and also limits the availability of data to other users. For example, suppose another user wants to find all photographs taken by the first user. Without the history he is forced to query all storage elements. Flooding queries is not practical or efficient and we desire a better approach to finding pervasive data.

Recent proposals for managing pervasive devices using *service discovery architectures* such as Jini [5], Salutation [8], UPnP [9], can assist applications in locating data within a local *service discovery domain*. For example, an application in a Jini community can consult the local "lookup service" to "discover" all services available within the community boundaries (typically an administrative subnet). With small enhancements we can use the same mechanism to locate devices that may have relevant data. Unfortunately, there is no general mechanism for an application to "query the world" for data beyond the borders of a service discovery domain. This imposed locality is common for all service discovery architectures and imposes a serious limitation on the availability of data.

Past research in wide area service discovery provides us with insight into solving this challenging problem. The goal of wide area service discovery is to provide a global index of services available in a discovery domain. For example, such an index enables an application in a discovery domain in Los Angeles to locate printer services in a remote discovery domain in Beijing. There are two main approaches for constructing a global index: distributed server hierarchies and replicated indices. In the former case, the global index is administratively partitioned among servers arranged in a hierarchy [6][4]. There can be many server hierarchies on the network. In the second approach, the global index is replicated to many distributed servers all linked together [1]. This can work well if the index is small and does not need much updating. Keeping these indices consistent in the face of more dynamic data is problematic.

In this paper we propose an adaptive index scheme guided by the following observations about indexing pervasive data:

- A fundamental problem with replication schemes for pervasive data is that the replicated index could be very large. It may be more practical to build application specific indices that limit the scope of the data they index.
- The effectiveness of an index depends on the query workload. It may not be possible to predict what kind of data applications will need beforehand. Static indices may become outdated as applications change their information needs. Re-tooling an index manually is not desirable. We require an index that can adapt to the observed workload and optimize itself to deliver data that applications need.

In this paper we introduce an application-level protocol called VIA* that addresses these issues. VIA* is an enhanced version of our Verified Information Access (VIA) protocol for global

indexing [3]. Both VIA and VIA* allow distributed servers located in different discovery domains to self-organize into a hierarchical index. VIA assumes that the topology of the hierarchy is defined *a priori* by an application data schema. In VIA* we relax this assumption and allow servers to approximate the topology of the hierarchy based on a heuristic we call *query impedance*.

The remainder of this paper is as follows. In Section 2 we provide an intuitive overview of VIA and VIA*. In Section 3 we discuss optimality consdirations for VIA*. Section 4 covers some results from our VIA testbed that illustrate the performance and scalability characteristics of VIA*. We also map out our research agenda in this section.

2. PROTOCOL OVERVIEW

In this section we list our assumptions about the topology and operation of a pervasive computing environments and outline the major features of our protocol.

2.1 Service Discovery Domains

A service discovery domain is a managed, pervasive computing environment. The boundaries of the domain can be administratively defined (e.g. a subnet in an IP network) or be enforced by some physical property such as the range of a wireless network. Devices can discover other devices or services by consulting a master directory managed by the domain infrastructure. All devices that enter a domain register with the master directory. If a device leaves, then its entry in the master directory is eventually purged.

We assume that service discovery domains exist at the edges of a wide-area network (WAN) that provides Internet "best effort" service guarantees (see Figure 1). Every service discovery domain has a dedicated *gateway* that acts as a mediator between devices in the domain and resources that are available from the WAN [7]. We assume that this gateway exists on a capable machine and primarily exists to provide management services. In a Jini community this gateway could be the same machine that runs the local lookup service. The WAN acts as the main channel of communication between gateways and under normal operation it is always possible for one gateway to communicate with another.

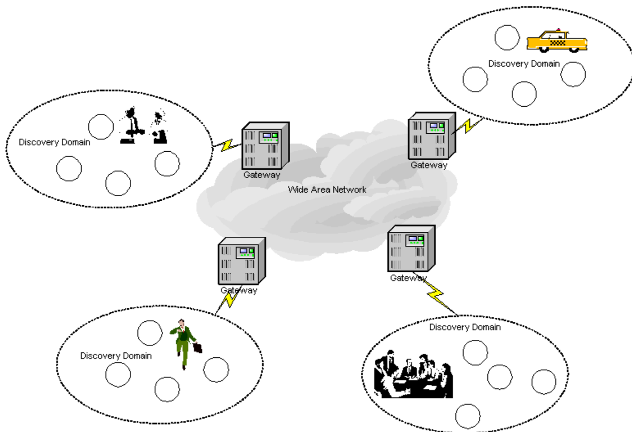

Figure 1- Service discovery domains at the edges of a wide area network

Gateways provide the fundamental infrastructure for us to build data sharing indices. We currently assume no additional machines are used to support indexing data. This may not be the most efficient infrastructure for implementing an indexing scheme but it does prevent us from relying on an installed base of special servers. However, this does not preclude us from using special servers in an actual implementation to improve the performance of our system.

Data is described by a list of attribute-value pairs contained in a metadata tag. When a device has data that it wishes to share outside the local discovery domain it sends a metadata tag describing this data to the local gateway. A query for data is another attribute-value list where attributes can have actual values or be "wildcarded", indicating a "don t care" condition. For example, a metadata tag M describing cars could look like M={tagtype="car", color="red", owner="Mani"}. If an application is looking for all cars owned by Mani then the query Q would be Q={tagtype="car", color="*", owner="Mani"}. Note that we use the tagtype attribute to uniquely defined a set of attribute names in our example. This removes ambiguity when there are two different tagtypes with different semantics for an attribute.

All queries are directed to the main channel where they are received by gateways listening on the channel. Gateways with a non-null response to a query sends a response message to the client (by way of the client s local gateway) that contains the associated data or a pointer to the data.

Any gateway listening for queries on the main channel can be flooded with irrelevant queries. Ideally, only gateways with a non-null response to a query will receive it. In VIA, we create an application-level overlay network out of the gateways on the network. The overlay network routes queries more selectively than the main channel and can adapt to changes in the query workload. Although a network level solution may be more efficient, the overlay network does not require modifications to underlying network hardware and protocols. However, any improvements to the network, e.g. content-based routing of packets, can complement our scheme.

2.2 VIA Cluster hierarchies

Data is described by metadata tags that contains attribute-value pairs defined by an application. We also view metadata tags as filters that we can use to process queries. We say a metadata tag M* subsumes another metadata tag M (M* > M) if all the corresponding values of attributes in M match those in M* (where wildcards match any value). In other words, the data described by the intersection of M and M* is non-null. We can also say that two metadata tags are equal (M*=M) if they describe the same set of data.

Assume we have three gateways G1, G2, and G3 with metadata tags M**, M*, and M, respectively and M**=M*=M as shown in Figure 2A. There is no reason for all three gateways to be on the main channel. Instead, G3 can link to G2 and G2 can link to G1. Now only G1 is on the main channel and it will forward any query Q where M**>Q to G2. G2 in turn will forward any query to G3 if M* > Q. We say that G1, G2, and G3 form a cluster (Figure 2B). The benefits of this are obvious: because G1 filters out all irrelevant queries, G2 and G3 have a vastly reduced workload. Also, the capacity requirement for the main channel is reduced since G2 and G3 no longer need to be on it. G1 still has to listen on the main channel but it does not do

significantly more work than it did before forming the cluster (though it must forward query messages).

We can repeat this process to cluster clusters. Suppose we now have a fourth gateway G4 with metadata tag M***, where M*** > M. To reduce its workload, G1 can link to G4. Although G1 will receive some irrelevant queries it still reduces its potential workload and no longer needs to be on the main channel. By linking G4 and G1, we are clustering a cluster since there can be many metadata tags that are subsumed by M***. We call this structure a cluster hierarchy (Figure 2C). Other clusters can link to G4 also. For example, a cluster with metadata tag F can get behind G4 if M***>F.

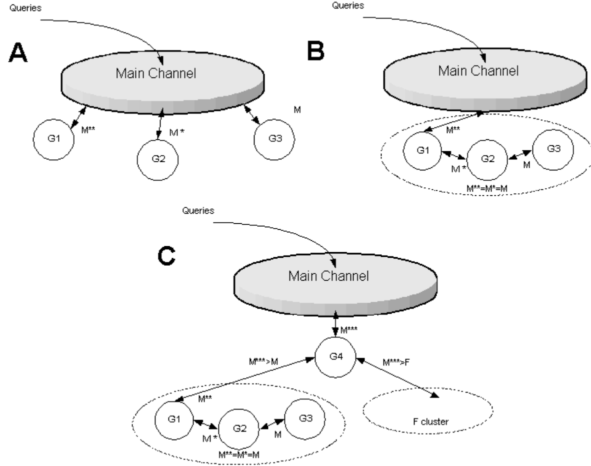

Figure 2. Steps in creating a filtering hierarchy through generalization

In VIA, gateways self-organize into cluster hierarchies using a distributed application-level protocol. The topology of the hierarchy is a spanning tree. VIA builds this tree in a bottom-up fashion. A gateway G with metadata tag M autonomously processes queries until it deems that its workload is too high. It then seeks to join a cluster that can filter out irrelevant queries. In general, it is not possible for G to know about all the clusters on the network. G instead determines internally the "best" cluster for it to join through a process called generalization. During generalization, G takes M and creates a new tag M* that is a generalized version of M. G now has two logical nodes M and M* where M* > M. VIA defines operations for G to go out on the network and find a matching cluster for M*. If one exists, then VIA defines operations for G to link into the cluster. If one does not exist then G becomes the root of a new cluster. Although G does not directly benefit from its actions, other gateways can now link to G and reduce their query workload.

Roots of clusters can repeat the clustering process again to form a multi-level hierarchy. Suppose there exists a cluster with tag M** where M** > M*. If G repeats the generalization process as the root of M* it can discover this cluster and join it. This can repeated as long as the metadata tag can be generalized.

2.3 Generalization

Generalization incrementally increases the "scope" of a metadata tag through the union operator. For any M, the generalization process replaces some number of attribute values with a wildcard to create M*. The condition M* > M is true for all possible generalizations of M, but M should also be a subset of M*. Generalization is an identification process for finding a candidate filter to "get behind." Clearly, different generalizations will create different cluster hierarchies on the network. In VIA, the order in which attributes are generalized is dictated by a hierarchical application data schema that all gateways have access to. The data schema acts like "DNA" and guides the creation of the hierarchy and it should be defined so that the hierarchy is "near optimal" for a given query workload. Multiple hierarchies can be created using VIA by changing the order of generalizing attributes. One disadvantage of this is that the data schema is statically defined and changing the topology of the cluster hierarchy requires "reseeding" the network with a new data schema. Ideally, we would like the hierarchy itself to be adaptive to changes in the query workload. In VIA* we propose a way to redefine the static definition using the query impedance of an attribute to estimate an optimal ordering of the attributes to generalize.

2.4 Query Impedance in VIA*

In VIA*, a gateway must determine an order for the attributes in M so it can generalize its filter to identify a candidate filter to get behind. In the ideal case the gateway will identify a filter that passes on the least irrelevant queries (based on the expected query workload). For example, consider the query workload:

Query$_1$= {car color = "yellow" owner="Paul"}
Query$_2$= {car color = "red" owner = "Paul"}

These two queries for "yellow cars owned by Paul" and "red cars owned by Paul" occur with equal probability.

Consider two gateways A, B with filters:

Filter$_A$= {car color="yellow" owner="Paul"}
Filter$_B$= {car color="yellow" owner="Jack"}

For A, the main problem with the possible queries is that it sometimes receives queries for red cars. It would like another filter to filter out these queries. The possible filters it can get behind (given that we only have a wildcard operator to generalize):

F_{A1}= {car color="yellow" owner="*"}
F_{A2}= {car color="*" owner="Paul"}
F_{A3}= {car color="*" owner="*"}

Clearly, the best filter is F_{A1}. Although it will pass queries for yellow cars not owned by Paul, the fact is that all queries are for owner=Paul. Thus it gains the most reduction in queries from F_{A1} and no reduction from F_{A2} and F_{A3}.

Consider gateway B in the same situation. Although it too receives irrelevant queries for red cars, the main problem for B is that all queries are for owner=Paul. Ideally, it would like to get behind a filter that would only forward queries for Jack.

The possible filters B can get behind are:

F_{B1}= {car color = "yellow" owner="*"}
F_{B2}= {car color = "*" owner = "Jack"}
F_{B3}= {car color = "*" owner = "*"}

Following the same logic as A, the best filter for B to get behind is F_{B2}. Although this will send irrelevant queries concerning red cars, it will only forward queries for owner = Jack.

Gateway A would like to generalize attribute "owner" first while B would like generalize attribute "color" first. These two situations are related because A and B are motivated to keep instantiated the attribute that will filter out the most queries.

In VIA* we measure this concept with query impedance Z for an attribute. This statistic is simply the # times an M attribute does not match a Q attribute / # queries received. To generalize, we order the attributes in increasing order of Z that could potentially create the optimal hierarchy for a given workload. However, since this is done in a distributed manner in the presence of failure, random timing issues, and each gateway only has partial knowledge about the system, the resulting hierarchy may be suboptimal.

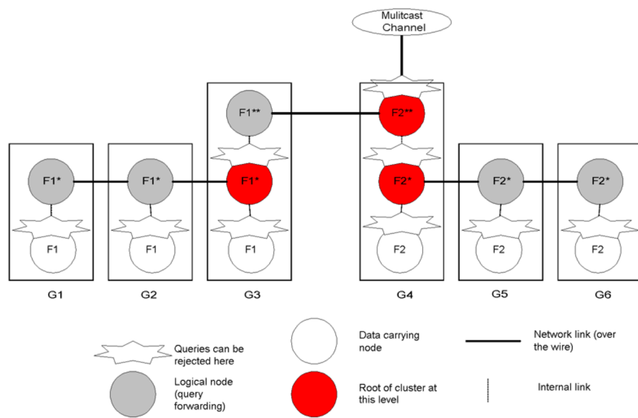

Figure 3. Multiple "logical" nodes on physical gateways

The previous discussion assumed that only a wildcard operator was available to generalize. We can refine the generalization process by allowing an "OR" operator in place of a wild card. For example, we could generalize $FILTER_A$ to be `{car = "yellow" OR "red" owner = "Paul"}`. The advantage of using an OR is that a gateway can get behind a more restrictive filter. The disadvantage is that the gateway must have knowledge about valid values for an attribute. We plan to investigate the use of OR operators as part of our research.

3. OPTIMIZING HIERARCHIES

The optimization goal of VIA and VIA* is to reduce the overall workload at each gateway but still allow data to be effectively indexed. The workload at a gateway is partly a function of the incoming messages it must process. If we assume all message types generate the same amount of work, then our optimization goal becomes one of reducing the overall communication between gateways during query processing. Reducing the cost of query processing is offset somewhat by the cost incurred by protocol overhead. VIA* must maintain a balance between the two.

3.1 Creating "good" hierarchies

In VIA and VIA*, clusters are a logical concept that represent both "real" data and query forwarding entities. From the logical perspective, the higher level clusers pass queries to lower-level clusters until they reach the appropriate terminal nodes.

Cluster hierarchies are implemented on physical gateways. In general there is not a one-to-one mapping between logical nodes and physical gateways. For example, Figure 3 shows a cluster hierarchy where several logical nodes reside on each gateway. Circles in the figure represent filters, and the enclosing rectangles are gateways that host them. Query processing occurs as follows. When a query first arrives at the root of the cluster hierarchy it is processed by gateway G4 using filter F2** and forwarded to G3 if necessary. G4 also "forwards" the query to an internal filter F2* which in turn forwards the query to G5 if necessary. This process is repeated at each gateway that receives the query until the query reaches a terminal filter. Gateways with a non-null response generate return messages to the source of the query.

An ideal hierarchy will only send queries to gateways with a non-null response. However, in the VIA* scheme gateways that contain high-level filters in the cluster hierarchy will be forced to process some queries that are irrelevant to their data carrying filters. For example, G4 must process every query that is relevant to G1, G2, and G3 even though it may have a null response to those queries.

Query impedance is a greedy heuristic that encourages gateways to get behind a gateway with the most restrictive filter that subsumes its data, thereby reducing the potential number of irrelevant queries they will receive. It is both an estimate of the query distribution and an estimate of the filtering power of a particular filter. If the estimate is correct and the data distributed on the gateways can be partitioned, the resulting filtering hierarchy will be "good." For example, the hierarchy in the previous section forwards the least number of queries to the data clusters. It is straightforward to show that any other possible hierarchy we could form will pass more queries to the data clusters for the given workload. In general, it is unlikely that using a greedy heuristic like query impedance will always result in an optimal cluster hierarchy. For example, a gateway may never find a filter to get behind if it only considers generalized filters that give it the maximum immediate benefit. A more comprehensive search strategy like dynamic programming could result in an improved hierarchy.

3.2 Towards Optimal Hierarchies

In an optimal hierarchy, gateways that expect to receive more queries should be closer to the main channel than gateways that receive fewer queries. Gateways that contain data relevant to many queries are the best candidates to host query forwarding nodes. Having relevant gateways closer to the root reduces the query response latency by minimizing the number of times a query must be forwarded. It also reduces the overall amount of irrelevant work done. For example, if most queries are for cars owned by "Jack," then it would be better for a gateway that has data about Jack’s cars to act as the root of the cluster hierarchy. If a gateway that only has data about Paul’s cars acts as the root then it will be forced to process far more irrelevant queries than if the situation were reversed.

VIA and VIA* use a "hit-ratio" statistic defined as the #relevant queries/total queries received to estimate the

relevancy of a gateway's data to the current query distribution. If the hit ratio is low, the gateway attempts to get behind a filter that will increase the hit-ratio to an acceptable level. If the gateway gets behind an ineffective filter (or becomes the root of a new higher-level cluster) then it can repeat the process. Gateways use hit-ratio to determine how far from the root of the cluster they should be. Of course, the resulting topology of the cluster hierarchy is dependant on the query distribution and the data distribution on the gateways.

3.3 Mapping Logical Hierarchies to Physical Gateways

As mentioned previously, cluster hierarchies represent logical entities that are implemented on physical gateways. "Bad" mappings will cause inefficiencies in the physical network such as duplicate queries and load imbalance. To create an optimal hierarchy, we must consider both the logical topology of the cluster hierarchy and the physical topology of the linked gateways that implement it.

For example, Figure 4 shows a mapping that results in an inefficient hierarchy. G4 has data items corresponding to F1 and F2 and is the root of two higher-level clusters that subsume F2 but do not directly subsume F1. G3 has a data item corresponding to F1 and is the root of higher-level clusters that contains F1. A query for F1 is received by G4 since it is the root of the entire cluster hierarchy. It forwards the query to G3 and internally forwards it to the logical filter node F2* where it is filtered out. G3 internally forwards the query to F1* and then to F1 where it matches. Because it matches, G3 forwards the query back to G4 where it received by the logical node F1. In this case, G4 must process the same query twice, which is undesirable.

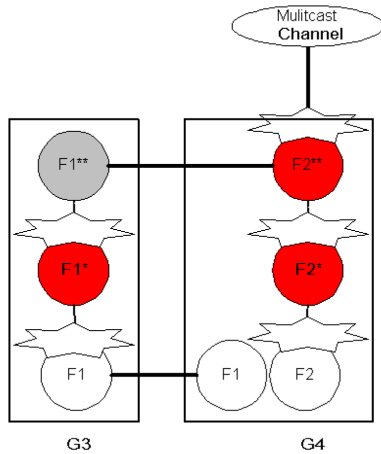

Figure 4. An inefficient mapping between a logical hierarchy and physical gateways

We can eliminate this particular inefficiency by ensuring that data nodes are always downstream of the highest-level filter on a gateway. For example, G4 could internally "create" a logical node F1* that could get behind F2**. Then G4 would no longer need to link to G3 in order to respond to queries for F1. A similar solution is to use an OR operator to create a summary filter "F1 OR F2" on G4 and use this filter as the "baseline" filter for all data managed by the gateway. This baseline filter is used in place of individual filters for F1 and F2, with the advantage being G4 would only have to process a query once. This does not penalize G3 since it can still get behind G4.

There are many questions regarding optimality and mappings for VIA*. We will investigate these questions as part of our research.

4. RESEARCH AGENDA

We have simulated VIA using the Parallel Simulation Environment for Complex Systems (PARSEC) with up to 1000 gateways as well as constructed prototype VIA gateways in our laboratory for workload experiments using Java. The purpose of the PARSEC simulation was to investigate the behavior of VIA's spanning tree formation operations. These operations are used to create and maintain a spanning tree of gateways on the network. The Java prototype testbed allowed us to investigate the workload reducing characteristics of cluster hierarchies.

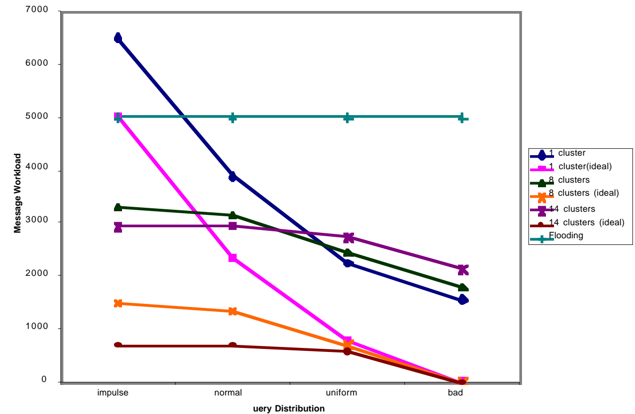

Figure 5. Message workload vs. query distribution

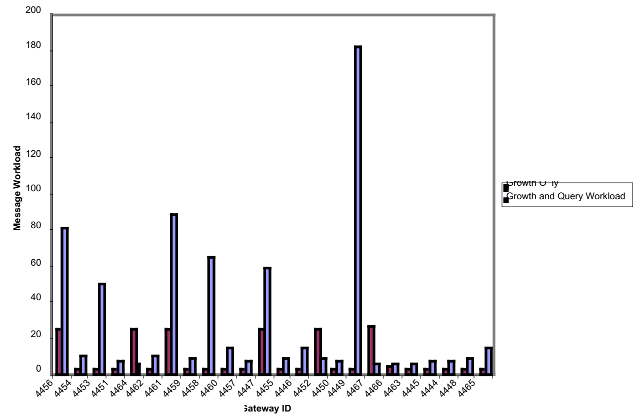

Figure 6. Message workload distribution for individual gateways

We have reported good results from our earlier VIA simulations and workload experiments [3]. For example, Figure 5 shows the effect of changing the query distribution on the message workload for different topologies, sizes, and numbers of cluster hierarchies on a network. We created a workload generator that could create a query workload at some pre-defined distribution. Changing the distribution modified the overall relevancy of queries to data on the gateways. For each workload we measured the work done by a VIA hierarchy against an ideal topology where only the relevant gateways responded to the query. We also compared VIA to flooding.

As the relevancy of the queries decreases, gateways in the ideal topology do less work (i.e. process less messages). When no queries are relevant to data on the gateways then the gateways in the ideal topology do no work. For the flooding case, each gateway does the maximum amount of work whether or not the queries are relevant. Gateways in cluster hierarchies created using VIA have the same workload characteristics as the ideal topology though there is an additional cost for using the VIA protocol. Gateways in VIA clusters do more work when queries are relevant and do less work when queries are less relevant.

Workload is not evenly distributed in a cluster hierarchy. Figure 6 shows the workload distribution for a cluster hierarchy with 24 gateways. In the figure, two different types of workload are shown. The first is the workload caused by VIA protocol overhead. The second workload is the total of protocol overhead and query processing. In the figure, most of the workload is handled by 6 gateways corresponding to root gateways of clusters in the hierarchy. The cluster hierarchy root does the most work. Non-root gateways see a significant decrease in workload when compared to query flooding.

We are currently re-tooling our prototypes to support VIA* and query impedance. There are many challenges still in place for VIA*. We are focused on characterizing optimal logical hierarchies under changing workloads and measuring VIA* s responsiveness to these changes. It is not clear how query impedance will function if changes are frequent and we intend to investigate this.

Our work in VIA* is tied closely with our on-going research in pervasive sensing infrastructures. We continue to work on a distributed, cooperative, sensor database architecture that will implement VIA*. Using our architecture, users will have the ability to "query the world" for data derived from fusion services distributed across many discovery domains [2].

5. ACKNOWLEDGMENTS

Our thanks to Akash Nanavati and Murali Mani for useful theoretical discussions regarding filtering tree optimality. Also, thanks to the reviewers for useful comments in improving this paper.

6. REFERENCES

[1] W Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J Lilley. The design and implementation of an intentional naming system. Operating Systems Review, 35(5):186-201, December, 1999.

[2] P. Castro and R. Muntz. Managing Context for Smart Spaces. IEEE Personal Communications, October, 2000.

[3] Chatschik Bisdikian, Paul Castro, Ben Greenstein, Parviz Kermani, Richard Muntz, Maria Papadopouli. Sharing Appplication Data Across Service Discovery Domains. submitted for publication. January, 2001. (available at http://mmsl.cs.ucla.edu/~castrop/via-submitted.pdf)

[4] S. Czerwinski, B. Zhao, T. Hodes, A. Joseph, and R. Katz. An architecture for a secure service discovery service. Proceedings of the fifth Annual ACM/IEEE International Conference on mobile computing and networking:24-35, 1999.

[5] JINI(tm) Connection Technology. http://www.sun.com/jini.

[6] Jonathan Rosenberg, Erik Gutman, Ryan Moats, and Henning Schulzrinne. WASRV Architectural Principles. Internet Draft. Internet Engineering Task Force, Feb 1998. Work in progress.

[7] The Open Source Gateway Initiative. http://www.ogsi.org.

[8] The Salutation consortium. http://www.salutation.org.

[9] Universal plug and play. http://www.upnp.org.