A Search-based Network Architecture for Mobile Devices

Erik Nordström, Per Gunningberg and Christian Rohner
Uppsala University, Sweden.

Abstract

This paper presents the Haggle network architecture and experimental measurements of its performance in a realistic environment. Haggle provides a search-based data dissemination framework for mobile opportunistic communication environments, making it easy to share content directly between intermittently connected mobile devices.

Haggle's novel approach is based on its identification of search as a first class operation for data-centric applications. We show how search can be used for resolution (mapping data to interested receivers) and prioritization of sending and receiving data during encounters between nodes. Haggle provides underlying functionality for neighbor discovery, resource management and resolution – thus removing the need to implement such features in applications.

Haggle has been implemented for several platforms. This paper presents experimental results, the most interesting of which demonstrates the live operation of Haggle on mobile phones in an office environment.

1 Introduction

Mobile phones are envisioned as a pervasive platform for rich mobile networking applications [12]. During the past couple of years, the emergence of increasingly powerful development environments, such as iPhone OS, Windows Mobile and Android, has also accelerated third-party application development. However, the networking applications that target these platforms are mostly mobile versions of traditional applications that access the Internet over fixed infrastructure networks. Few applications exploit opportunistic device-to-device connectivity, although there are examples, such as Ad hoc Podcasting [13], in the research community. These examples show that opportunistic networking with mobile devices is feasible and promises exciting new applications and research.

A reason for the lack of opportunistic networking applications, we argue, is that the creation of such applications is made unnecessarily difficult and inefficient by the communication frameworks and APIs on mobile phones. These frameworks and APIs are traditional in the sense that they are based on, for instance, TCP/IP and BSD sockets that embed a host-centric communication model that assumes continuous connectivity. They do not match well a data-centric, opportunistic, and intermittently connected communication environment.

To address the above situation, we propose Haggle – a novel search-based network architecture that takes an holistic approach to mobile opportunistic networking. A holistic approach provides applications with a complete architecture framework for naming and addressing, device discovery, resource management, persistent data storage and more. Applications otherwise need to implement such features themselves – often in partially overlapping and incompatible ways. A holistic approach minimizes overlaps and incompatibilities, and therefore communication can be made more efficient and resource aware, while reducing development time and incompatibilities between applications.

The most important part of this architecture framework is the communication abstraction layer that hides a lot of the details of *when*, *where*, *how*, and to *whom* data is transmitted. For instance, at the time of sending data, an application needs no connectivity to receivers and may not even know their identities, or the network interface technology eventually used to communicate. Such temporal and spatial decouplings are widely argued as a corner-stone of a modern communication architecture [6]. Haggle implements such decouplings using a data-centric communication model with a publish-subscribe (pub/sub) API [7], which spreads application data from device to device based on the data's match against a user's declared interests.

A key challenge facing data-centric communication, however, is how to resolve the mappings between the users and the data they wish to receive. In host-centric communication the mappings between data and destinations are often implicit, or known a priori through the usage of well known mnemonic identifiers such as names, email addresses, and so forth. However, in data-centric communication there is no such clear a priori mappings and centralized lookup and mapping services do not work without continuous infrastructure connectivity.

Our solution to this problem is distributed *search-based resolution*. It builds on the observation that today's computing experience is to a large extent characterized by searching, we use it to locate and order content on the Web, as well as on our local computers. The idea is to extend such searching into the opportunistic network. Normally, when a user performs a search query for data on its local computer or device, it immediately returns the local data that matches the query as a ranked list. With search-based resolution we make this searching a networking primitive within the architecture, such that matching data is also transparently received from peers as they are encountered in the network. A novelty of this approach is that the matching between data and receivers is not binary – a top ranked match is the best only relative to lower ranked ones. Each device can hence limit the amount of disseminated data to only the top ranked nodes with the most interest in the data. In contrast, other approaches to dissemination uses binary matching filters or topic channels [7, 13] that are static and lack relative matching and ranking.

In the rest of the paper, we describe the Haggle architecture and the search-based networking it embeds. We detail the implementation of Haggle and how it simplifies application development and integrates with existing networking technologies. The contributions of the paper are:

- A data dissemination framework based on search-based resolution (Section 3), which allows novel approaches to many established networking concepts (Section 4). We provide formal definitions of networking primitives based on searching (Appendix).
- The design and implementation of the Haggle architecture built around the above framework (Section 5 and 6). The architecture makes it simple to develop both new applications and extensions to the architecture itself.
- A live evaluation of Haggle, showing the functionality of the architecture and implementation (Section 7). The experiment takes place in an actual office environment where Haggle is used to disseminate pictures to mobile phones.

2 Related Work

Haggle builds broadly on two previous bodies of work. The first comprises works that incorporate searching, and we here introduce concepts from other work that have inspired the design of Haggle's search-based resolution. Then second body consists of new network architectures and paradigms with which we share similarites in architectural principles and design. We discuss these two bodies in the order mentioned.

Searching. In the context of this paper, we refer to searching as a way to extract information from an expressive but otherwise flat metadata namespace (the

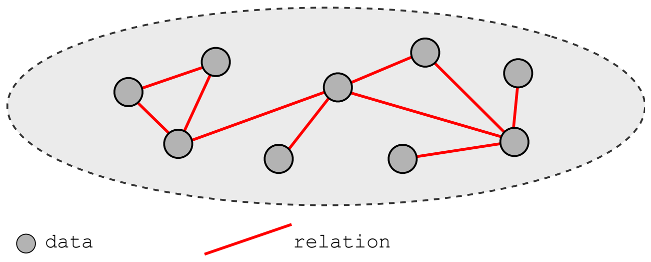

Figure 1: A relation graph consisting of data (content) and relations. Relations may have an associated weights that indicate their strength.

metadata is not structured in, for example, a hierarchy). The result from a search query is a ranked list of data that is relevant to the query. For effective searching, the data must be visible in a namespace that is common for all types of data. Thus, for Haggle we need an expressive and unifying namespace.

Early work on such namespaces was done in the context of local file searching. This was a response to the simple namespace of files and directories on a filesystem that tell little, if anything, about the content of files. Although files may embed metadata, the namespace used is often different for each application and file type.

The semantic file system [11] tries to alleviate these limitations by embedding single format attribute-based metadata (name-value pair) with the file on the filesystem. Desktop search tools, such as Google desktop [1], similarly automate metadata extraction into a searchable index, independent of filesystem. In Haggle, we embed attributes with data not only while stored on disk, but also while disseminated in the network. The attributes form the actual "header" used to forward the data.

Connections [15] enhances basic attribute-based file searching by building temporal relations between files. These relations are built when files are accessed on the filesystem in the same time slot. The relations then structure the files in a relation graph according to Figure 1. The strength of a relation is based on how many times a pair of files have been accessed simultaneously. Thus, a file that matches badly a search query may be highly ranked due to its relation to a file that matches well. PageRank [4] is also an algorithm that can be visualized in a relation graph. However, instead of using temporal relations, the edges represent hyperlinks between web pages (the vertices). The rank of a web page is determined by its degree of incoming edges and the rank of neighbor pages.

Haggle learns from the above work the concept of relations and ranking. A key addition is that both data holders (e.g., devices) and the data itself may be represented in the same namespace and this allows them to build relations between each other in the relation graph.

This idea forms the basis of search-based resolution, as we describe in Section 3.

Network Architectures and Paradigms. Publish-subscribe [7] is a communication paradigm that temporally and spatially decouple the subscribers of content from the publishers. Haggle incorporates a pub/-sub API, but differs in how data is disseminated in the network. Pub/sub systems either disseminate based on exact matching filters [5], or channels of topics [13]. In contrast, Haggle uses filters only for local demultiplexing of data to applications, and instead uses searching with ranking for disseminations. The main difference between filtering and searching is that the former does consistent matching whilst the latter does relative matching that may change over time (analogous to searching with, for example, Google). Applications prefer to receive data in a consistent manner, whilst disseminations fit better a model that accounts for what makes best sense at the time, and for the device that disseminates.

The *role-based architecture* (RBA) [8] shares similarities with Haggle in that it organizes communication in functional units called *roles* instead of layers. In RBA, packets carry metadata that consists of a number of role-specific headers (RSHs). Haggle generalizes RBA by collapsing the RSHs into a single metadata header. The roles of RBA correspond well to what we call managers (see Section 5), but we do not as rigidly bind managers to certain parts of metadata as RBA binds roles to RSHs. RBA neither incorporates searching and filter based demultiplexing. RBA uses a scheduler to determine the processing order of RSHs. Haggle instead uses an event ladder to structure processing (as explained in Section 5.1.4).

The delay tolerant network architecture (DTN) [9] provides a bundle delivery service that incorporates a late binding addressing scheme based on end-point identifiers (EIDs). The Unmanaged Internet Architecture (UIA) [10] uses similar EIDs that map to personal names, and can organize devices in groups. Haggle shares the late binding mechanisms with these architectures, but neither does bundling nor uses EIDs. Haggle does, however, use application layer framing with searching to bind data to target nodes in place of EIDs.

Su et al. presented in [16] the previous work on the Haggle architecture as part of the same project. We share with that work the basic goals and principles of the architecture and the RBA-inspired separation of roles in the form of managers. Apart from this legacy, our architecture is a complete redesign based on the lessons learned from the previous work. The major changes and new contributions are as follows. We use a flat metadata namespace with searching whilst previous work used an INS [3] inspired naming system.

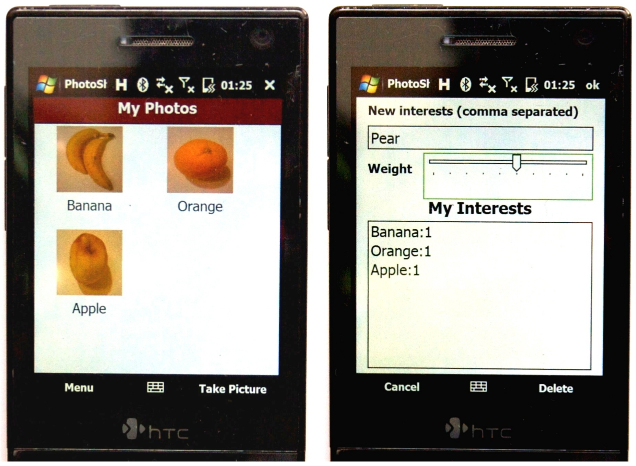

Figure 2: PhotoShare disseminates pictures based on the metadata added by the user (right). Received pictures automatically pop up in the window (left).

This improves the flexibility and late-binding mechanisms of the system. We have redesigned the architecture to be event-driven instead of task/thread-driven, which saves a lot of unnecessary processing. We use decentralized resource management with centralized policy making instead of a completely centralized system. The basic problem with the previous approach was that a central resource manager cannot well determine the resource cost of tasks that belong to other managers. Apart from these main differences, there are countless other changes in the architecture that are too numerous to list here.

3 Haggle Overview

Haggle provides a *push-based* data dissemination service that notifies applications when data matching their interests is received. Applications need not themselves implement essential mechanisms for opportunistic communication, such as neighbor discovery, interface selection, persistent data storage, and dissemination. Haggle's architecture framework provides this for applications and therefore drastically simplifies them.

In Figure 2, we show an example Haggle application called PhotoShare. This application makes it easy to share pictures that are taken with a mobile phone's camera. The pictures spread, over either Bluetooth or WiFi, according to the embedded metadata and the interest registered by users. The communication related code to achieve this is just slightly above 200 lines of C# in the application. MailProxy is another application that allows a user to send email over the opportunistic network using the phone's built in email client. The proxy translates emails into Haggle's native data format for dissemination over the opportunistic network. This proxy solution illustrates clearly that Haggle can also support legacy applications.

PhotoShare and MailProxy, as well as any other applications, may run intermittently and concurrently. When they have registered with Haggle, they can shut down while Haggle continues to receive and disseminate information on their behalf. The registered data and interests are persistent in Haggle until explicitly removed by applications. It is hence easy for users to continue dissemination while carrying their phones in their pockets.

The most important framework mechanism in Haggle, which makes this dissemination work, is its novel data-centric resolution service (i.e., mapping data to interested receivers). The interests that applications register can be seen as search queries, similarly to the keywords input into a search engine. The interests are first matched against the local data on a device, and may then also propagate to other devices for matching. In the rest of this section we will focus on how this matching and resolution work in detail.

3.1 Unified Metadata

In order to do matching between different types of data, Haggle uses a unified metadata format. The metadata format is used in place of traditional packet headers that enable addressing and multiplexing/demultiplexing based on, e.g., IP addresses and port numbers. The metadata instead itself defines the "address" and searching and filtering against interests are operations that determine how data is disseminated in the network and demultiplexed to applications, respectively.

An item of data with associated metadata is in Haggle called a *data object*, and is hence a tuple $(metadata, data)$. The data object is an application layer framing format in which the data may be, e.g., an email, an MP3 music file, a PDF document, or as in PhotoShare a JPG picture. The metadata consists of a set of attributes in the form of name-value pairs that describe the data, as illustrated by the PhotoShare example above. Attributes can be manually added or automatically extracted from the data.

3.2 The Relation Graph

Each Haggle node maintains a relation graph of their currently stored data objects, similar to the one in Figure 1. However, in Haggle a relation is determined by shared attributes between a pair of data objects. The "strength" of a relation increases with the number of shared attributes.

The purpose of search-based resolution is to bind data to interested receivers or vice versa. We call such receivers the *targets* of a specific data object. To make the target resolution work with our relation graph we also allow nodes to be represented as data objects. We call such data objects *node descriptions*, and their attributes are the combined application interests of a node. Figure 4 (a) depicts the relationships between data objects and nodes in the relation graph. In the figure, we map node descriptions to nodes in a logical node plane to help our explanation of how the relation graph works.

A node sends its node description to every newly discovered neighbor. Node descriptions typically have no auxiliary data, but may have other metadata apart from attributes. One example of such extra metadata is a Bloom filter that indicates which data objects a node has already received (see Section 5 for details on the node description). In fact, non-attribute metadata can exist in any type of data object, but it does not build relations in the relation graph.

3.3 Search-based Resolution

We define a number of search-based resolution primitives that operate on the relation graph. In the appendix, we provide formal graph theoretic definitions of the primitives while we here informally describe them.

The search primitives are invoked as the relation graph is updated. Resolutions are triggered when applications register new data objects (or interests), or when new data objects are received from neighbors. Figure 3 illustrates these two cases.

The first primitive is hence *insertion*, which simply adds a data object to the graph. Every time an insertion occurs a *demux* primitive is also triggered. This is a persistent filter operation that, given the interests of local applications, demultiplexes matching data objects.

The next primitive we define is a non-persistent *resolve*, which determines how to map data objects to nodes, and vice versa. We refer to these two resolve

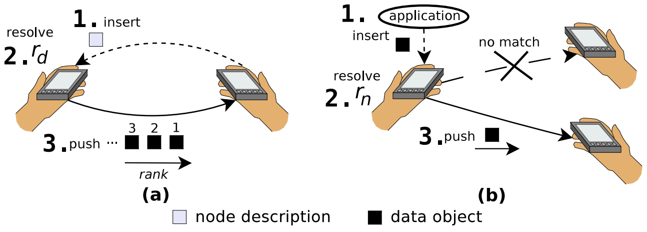

Figure 3: **Event-driven resolution:** **(a)** Two nodes meet and exchange node descriptions. This triggers r_d and the resolved data objects are pushed to the neighbor in order of rank. **(b)** An application inserts a new data object while nodes are co-located. The insert triggers r_n and the data object is pushed only to matching neighbors.

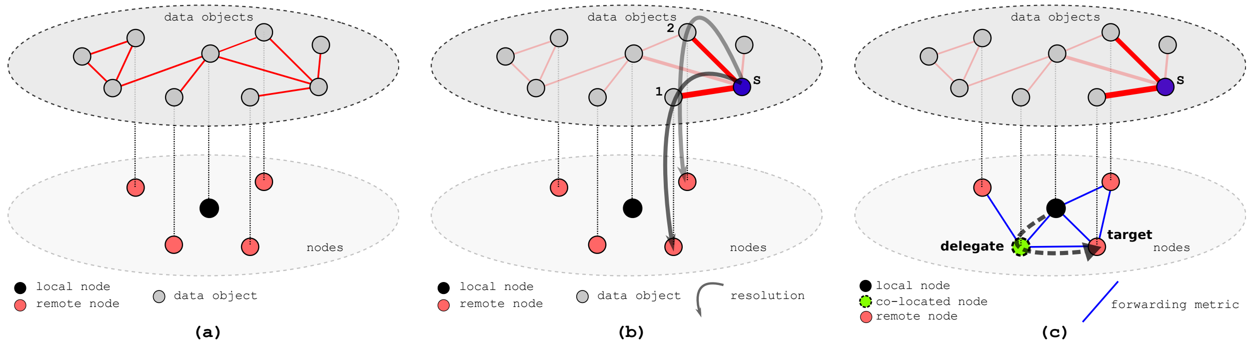

Figure 4: **Search-based resolution on a relation graph:** **(a)** Some data objects represent nodes in the network as illustrated by the logical node plane. The attributes of a node data object are the interests of the node. **(b)** A data object is matched and ranked against other data objects and subsequently nodes. This resolution step determines a ranked list of target nodes. **(c)** A forwarding algorithm may independently maintain forwarding metrics between nodes. These metrics determine *delegate forwarders* when dissemination within an interest community is not sufficient to reach the whole community.

operations as r_d (resolve data objects) and r_n (resolve nodes), respectively (again see Figure 3). In r_d one wants to resolve the data objects that match a given node description. In r_n one wants instead to find the matching node descriptions in the relation graph, given a normal data object. We refer to the resolved nodes as the *interest community* for the given data object. Note here the dual roles of node descriptions; both r_d and r_n happen for node descriptions as they are also data objects.

Figure 4 (b) illustrates r_n on our example relation graph, where the given data object s is the source of the resolution (i.e., the resolver wants to find the nodes interested in s). Using edge weighting functions, one first excludes data objects without relations to s and also those that are not node descriptions. A weighting function can be specified dynamically, i.e., we do not precalculate the weights in the relation graph. Thus, in our example, the weighting function gives zero weights for the excluded data objects, and then sets weights based on the shared number of attributes for the remaining node descriptions. In the final step, these node descriptions are ranked according to the edge weights relative s. The one performing the resolution hence determines that data object s has two target destinations to which it can disseminate in order of the targets' ranks.

In some cases, the number of resolved targets can be quite large and the node can then limit the resolution to only the top n ranked nodes. The resolver can also, through changing weighting function or setting minimal weights and ranks, exclude target nodes that do not have "enough interest" in the data object s.

Resolution r_d is analogous to r_n, only that s is a node description and the resulting data objects may be both node descriptions and normal data objects.

3.4 Forwarding

As data objects are disseminated in the network, each node must continuously make decisions on whether to forward a data object to a peer node it encounters. The default forwarding decision is simply to give data objects to interested nodes when in contact. Thus, the data ends up being disseminated epidemically within its interest community. We therefore refer to this basic forwarding as *interest forwarding*.

Because node descriptions are disseminated as any other data objects, nodes learn the interests of other nodes that they may have never met, and may never meet. When an interest community is also segmented it is not sufficient to use interest forwarding in order to reach the entire community. Even if the community is connected at some point in time it might just be slow to rely solely on interest forwarding, because one has to wait until at least one of the nodes in the interest community is encountered before forwarding can take place.

When interest forwarding is not sufficient, other forwarding algorithms can be used to improve the dissemination. Nodes then delegate data objects to nodes outside their interest communities, such that nodes carry data objects on behalf of communities they are not themselves part of. Forwarding that uses such delegation we call *delegate forwarding*.

A delegate forwarding algorithm maintains, for each pair of nodes, some metric that determines whether delegating a data object is likely to improve the dissemination (i.e., reaching the target receivers). A metric can, for example, be probability based or use history of encounters. In Figure 4 (c), we illustrate such metrics in the logical node plane. When search-based resolution determines target receivers that are not in contact, the delegate forwarding algorithm can use the metrics to compute whether a co-located neighbor is a good delegate for the data object. Haggle continuously collects statistics that delegate forwarding algorithms can use to compute metrics, and it can thus easily accommodate many different algorithms.

3.5 Scalability of Resolutions

As the relation graph grows in size over time, resolutions will span an increasingly large query space. This has implications for the scalability of search-based resolution. In order to determine if this is a major limitation within the architecture, we present benchmark results from Haggle running on a Macbook Air laptop (1.8 GHz CPU, 2 GB RAM) and an HTC Touch Diamond mobile phone (528 MHz CPU, 192 MB RAM) running Windows Mobile 6.1.

We generated and inserted a set of data objects D with randomly picked attributes copied from a pool A of size m, such that each data object $d \in D$ has a set of attributes $A_d \subseteq A$, with $|A_d| = l$. Similarly, a set of nodes N was created, where each node $n \in N$ has $|A_n| = k$.

Figure 5 shows the mean query time to return the matching data objects against each node in N, where $|N| = 100$ and the parameters $m = 1000$, $l = 10$, and $k = 100$. We hence measure the time to match a node's 100 interests against the 10 attributes of each data object in the relation graph. The important result from the graph is that a node can expect a query time around 1 second or below while storing up to 1000 data objects – even on a mobile phone. For a relation graph holding 10000 data objects we have a query time of around 20 seconds. At first glance this may seem large, but considering that only one resolution has to be done for a newly discovered neighbor, this time is not large relative the time to transmit all matching data objects. Although we believe we can further improve this result through future optimizations, we conclude that search-based resolution pose no immediate restrictions in terms of scalability.

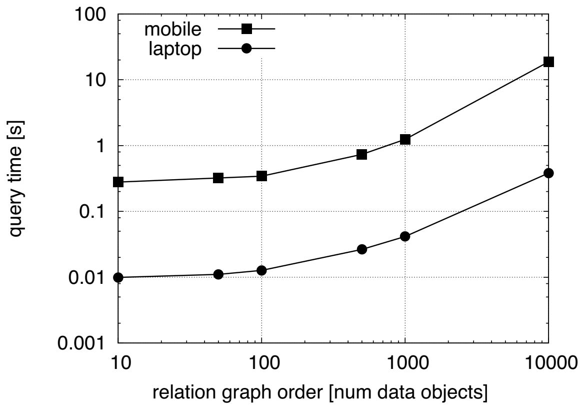

Figure 5: Query times (of r_d) for different orders ($|D|$) of the relation graph.

4 Searching as a Networking Abstraction

In this section we discuss the broader implications of our search-based architecture on a number of important networking concepts.

Naming and addressing: Search-based resolution breaks with traditional naming and addressing schemes that explicitly refer to end-points. The interests of a node (i.e., the attributes in its node description) work as both a name identifier and address. In Saltzers terminology [14], one can say that the interests provide an identity, while the positioning in the relation graph determines a (logical) location within the network, and thus an address.

Although file-sharing applications match well this naming and addressing scheme, personal communication does not as intuitively fit. However, we see point-to-point communication as a dissemination to a group consisting of exactly one node. Thus, addressing a specific node is a matter of finding a set of attributes that match well that node's interests, and thus restricts the group of nodes involved in the dissemination to only one. We therefore argue that Haggle, in combination with delegate forwarding, can also be used for traditionally host-centric communication applications.

To better facilitate restricted dissemination, we allow the interests of a node description to have associated weights. This allows nodes to express their interests more accurately by putting high weights on personal ones. For example, in the MailProxy application we can put a high weight on a personal attribute like email="john.doe@haggle.org", compared to a more generic attribute, such as subject="The meeting". John Doe can express – by weighting the interest attributes in his node description – that he is more interested in receiving data objects labeled with his email address, than those labeled with other generic attributes. The weights of interest attributes will affect the strengths of the relations in the relation graph, and hence the resolutions.

Resolution and binding: An important concept of search-based resolution is that bindings are done only at the time resolutions are performed. The search query determines the binding, although the metadata of individual data objects, of course, limits which bindings can be done. In comparison, resolutions with, e.g., DNS or ARP, are lookups that bind names and addresses already at the source and these bindings do not change during communication. Search based resolution, on the other hand, allows late and flexible bindings that occur continuously as data objects propagate among nodes.

Continuous resolution has the effect that every node that receives a data object may resolve new target nodes based on its own relation graph. Hence the accuracy of the search resolution improves as the data object propagates (at least as long as also the node descriptions of previously resolved nodes propagate with the data object). For a decentralized opportunistic network, where a node might not have a complete view of the network, this is a much better model than setting the targets only at the source.

A downside of continuous resolution is the cost in terms of time and battery power. However, we have shown that the resolutions scale well with the query space, and the time to resolve targets is very small in comparison to the time to transmit data. In Section 7, we also give results showing how battery levels are affected while Haggle is running.

Another important concept of search-based resolution is the ranking, which makes it possible to tune resolutions. For instance, to resolve a set of targets for a data object, it is possible to bind only nodes that share, e.g., at least 80% of its attributes with the data object, or have at least k attributes in common. It is also possible to limit the results of a resolution to only the top n results.

Demultiplexing: Filter based demultiplexing allows spatial and temporal decouplings of senders and receivers. A data object that is inserted into the relation graph by an application may be demultiplexed immediately to another application on the same node, or when an application adds a matching interest, or at some time in the future to an application on another node. A data object can also be demultiplexed to several applications at once. In comparison, port number demultiplexing synchronously binds communication to a single remote application or service, already at the source node. This is, of course, more efficient for small packet streams, but also less flexible.

Forwarding: Determining *delegate forwarders* is a task for forwarding algorithms. We anticipate that there is no single forwarding scheme that is suitable for all environments. We do not address specific delegate forwarding algorithms in this paper, but it is easy to integrate a number of forwarding algorithms found in the literature.

Most of these forwarding schemes do not decide the order in which messages are forwarded. This is because they commonly assume that all messages can be forwarded during node co-locations, or that they have no concept of which messages are more important than others. Search-based resolution allows data objects to be forwarded in order of rank against the target nodes. Ordered forwarding can better utilize time limited node contacts, by sending the most highly ranked data objects first.

Resource and congestion control: The disseminations in Haggle can be tuned according to the available resources of nodes in terms of disk space, bandwidth and battery power, and so forth. By expressing resource polices in node descriptions, a node can signal a neighbor that it should tune its resolutions according to the policies in the received node description. For example, when battery or storage is low, a node sets a restrictive policy that limits its own dissemination – and through the signaling – also the amount of information others will try to send it. With ranking, data objects are affected by the restrictions in order of lowest rank first. This is hence a *congestion control* scheme that automatically limits the dissemination in a way that is more sophisticated than, e.g., random drop schemes.

Security: Search-based resolution relies on the willingness of nodes to share their interests and the metadata of data objects with each other. In separate work, ways to do secret attribute matching without revealing the semantics of the attributes are being developed.

For authentication and data integrity, established security mechanisms work well also in Haggle.

5 System Architecture

In this section we give a detailed description of the Haggle system architecture and how it is designed around search-based networking.

5.1 The Core System

The Haggle architecture is event-driven and modular – features that allow flexibility and scalability. Central in the architecture is the *kernel*. It implements an event queue, over which *managers* that implement the functional logic of the architecture communicate. The kernel contains, apart from the event queue, a number of shared data structures, such as active neighbors, listening sockets, and also a data store that holds the relation graph. Figure 6 depicts how the kernel, managers and applications interact in the architecture. The circular structure of the architecture illustrates its layer-less design.

The managers are responsible for specific tasks and interact only by producing and consuming events. This makes it easy to add and remove managers in the architecture as they do not directly interact. Managers can delegate processing to *modules* that do work within their domains of responsibility. Modules are depicted in the figure as small circles attached to certain managers. We describe each manager in detail later, and here we

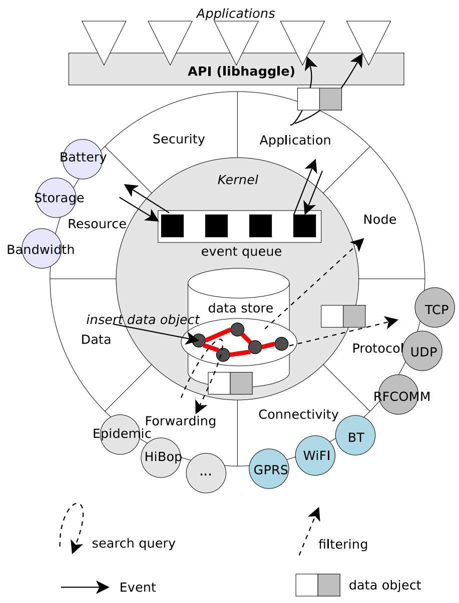

Figure 6: The Haggle architecture comprises a kernel and a set of managers.

```
<?xml version="1.0"?>
<Haggle>
        <Attr name="Haggle">NodeDescription</Attr>
        <Attr name="DeviceName">Haggle−1</Attr>
generic <Attr name="Music">Beatles</Attr>
        <Attr name="Email" weight="10">joh.doe@haggle.org</Attr>
                    ...
        <Node id="046d57ed06a0d6b78e351e6aaf38d313e5648f6b">
              <Interface type="Bluetooth">00:1b:98:9c:3b:a8</Interface>
specific      <Interface type="WiFi">00:1b:fb:05:c5:db</Interface>
              <Bloomfilter >AAAABwAAJYAAAA ... </Bloomfilter>
                                  ...
        </Node>
</Haggle>
```

Figure 7: The metadata header of a data object, in this case a node description.

instead focus on the core system and the interfaces provided to managers.

The event-driven design is essential for a search-based architecture; searching usually involves costly and lengthy I/O, and therefore requires asynchronous operation through event callbacks. Managers may register filters with the data store that generate events every time a data object matches. Similarly, they may also do search queries that return results through callback events, such that the queries do not block the system while processed. The event-driven approach was also chosen because it fits nicely with opportunistic communication. Node encounters are external events that drive the system and this triggers disseminations to occur. If the neighborhood is static and no applications generate new data, the system sits idle, thus preserving resources.

5.1.1 Data Types

To achieve efficient dissemination of data objects, Haggle is concerned with (1) matching and ranking of data objects, and (2) deciding to which nodes it should disseminate, and (3) how it can interface with nodes once they are encountered. Three data types help with these tasks internally: *Attributes* are part of the metadata of nodes and data objects. *Nodes* represent communication peers, and when a binding between a data object and a node is made, the node can be attached to the data object as a means to address the peer. *Interfaces* represent a way to interface with the peer – it can be a physical interface, for instance an Ethernet or WiFi card, or a logical interface provided by an IPC mechanism, such as a local socket or a pipe.

The metadata of data objects is expressed in XML. The managers may add any valid XML structure to the metadata of data objects, but only attributes can be searched and filtered. The metadata hence has a *generic* part consisting of attributes, and a *specific* part consisting of other XML structures. In order to receive data objects with a specific type of metadata, a manager also needs to add an attribute that signals the presence of specific metadata, such that the same manager on another node can filter with that attribute.

Figure 7 shows the structure of a *node description*, in which the attributes are interests and the specific metadata within the node tag has been added by the node manager. The node manager on the device that receives the node description can demultiplex it based on the Haggle=NodeDescription attribute. The presence of this attribute signals to the manager that there is manager-specific metadata in the data object that it can use to create a node object representing the peer node.

5.1.2 The Data Store

The data store holds data objects and implements the relation graph and search primitives. The backend is pluggable but is typically based on a relational database that is stored on disk.

Every data object is timestamped and may age, which is useful on devices with limited storage. A user can set an age threshold, after which data objects are deleted (their data may still be on disk). This limits the amount of information disseminated. Generally, only delegate data objects are deleted due to age as other data objects represent data that belong to local applications. A popularity counter also measures how often data objects match queries. Popular data objects in this way age slower. The architecture can accommodate several aging and popularity algorithms, but the development of such algorithms is out of the scope of this paper. The data store also collects context information, for example, node encounters and their durations and other statistics that may be useful for resolutions and forwarding.

The managers register filters with the data store in order to demultiplex the data objects they are interested in. The data store also implements a query interface for the r_n and r_d resolve primitives. For example, the interface for r_d is `resolve(node, max, min_match, ratio, callback);`. The `max` parameter sets the an upper limit on the number of data objects returned, `match` specifies a lower limit on the number of attributes in a data object that have to match the node's interests, whilst the `ratio` instead sets a lower limit on the percentage of the node's interests that must be matched. The `callback` parameter is the function used to return the result of the query.

5.1.3 Events

Haggle specifies three event types: *public*, *private* and *callback* events. Public events are predefined events that managers can register interest in. The most important events are listed in Table 1, along with producers and consumers and data type passed with the events.

Private events are registered by managers and are persistent until unregistered. They are used to implement timer based operations, such a garbage collection and beaconing. Each demultiplexing filter is also associated with a private event that is used to return the matching data to the manager that registered the filter.

Callback events are one-time events that are non-persistent and happen in response to a previous function call. The callback context is passed as a parameter in the function call. Callback events are typically used to return the results of search queries that are explicitly initiated by managers.

Event	**Producers**	**Consumers**	**Data**
Received Data Object	Protocol	Security, Any	Data object
Verified Data Object	Security	Data, Any	Data object
New Data Object	Data	Any	Data object
Local Interface Up	Connectivity	Protocol	Interface
Local Interface Down	Connectivity	Protocol	Interface
New Contact	Node	Forwarding	Node
End of Contact	Node	Forwarding	Node
Send Data Object	Any	Protocol	Data object
Resource Policy	Resource	Any	Policy
Data Object Targets	Data Store	Forwarding	Data object

Table 1: Example public event types, with producers, consumers and associated data.

5.1.4 Data Paths and Processing Order

Any manager can independently of others register a filter and therefore there is no specific order of processing for data objects that match multiple filters. Thus, without some way to indicate the processing state of data objects, a manager cannot know what type of processing a data object has been subjected to. The Security manager should, for example, verify the integrity of data objects before they are processed by other managers.

Our public event system solves this problem by defining events that implicitly specify the state of an object. For instance, managers that rely on security processing only listen to events that indicate security has already been considered. A data object hence climbs an *event ladder* as it is processed by different managers. The first three events in Table 1 illustrate this ladder: the Protocol manager issues the first event as a data object is received. The Security manager listens to this event and verifies the data object passed in it, after which it issues a new event indicating the data object has been verified. The Data manager in turn inserts this data object in the data store and issues an event that indicates the data object will be considered in resolutions. Thus, the designer of a manager need to account for the state of the data passed in the events it processes, and in the events it generates.

5.2 The Managers

After having described the core system we now turn to detailing the managers and the tasks they are responsible for.

Resource Manager: The Resource manager issues resource policies, based on measurements of, e.g., battery level, disk space, and bandwidth. How to act on the policy is a local decision made by each manager, since they best know the cost of its tasks and how to deal with its resources. Under resource constraints, this may include restricting disseminations and neighbor discovery, and choosing power efficient ways to transmit data. The Resource manager can also append resource control metadata to the node description (e.g., within a `<resource>` tag), in order to signal its policy to neighbors, as discussed in Section 4.

From previous work [16] we learned that a distributed policy implementation is crucial for efficient resource management. With the previous centralized resource management, managers registered tasks with the Resource manager, which then scheduled them based on the current policy. This system had two major drawbacks. First, the resource manager did not have a good understanding of the relative importance of tasks issued by different managers. Second, the other managers did not know the resource policy in effect, which meant they often registered tasks that never ran (or ran too late). This effectively wasted resources instead of preserving them.

Connectivity Manager: The Connectivity manager discovers local and remote network interfaces in order to determine connectivity to other nodes. Local interfaces are monitored for configuration changes, and whenever connectivity is established, an event is issued and neighbor discovery on the interface is started. The event informs other managers of a new connectivity opportunity. For instance, the Protocol manager starts listening servers on interfaces that become active, so that incoming data objects can be received.

The remote discovery is specific to the type of local interface used. For instance, on a Bluetooth interface regular device inquiry scans are performed. In the case of WiFi or Ethernet, beacons are sent instead. The rate of discovery can be varied depending on the policy set by the Resource manager.

Node Manager: The Node manager collects information about nodes that are encountered. Every time a new neighbor interface is discovered, the Node manager tries to exchange node descriptions with the node associated with that interface. From received node descriptions, the Node manager creates internal node objects that it inserts into the data store and into a list of active neighbors it maintains.

Protocol Manager: The Protocol manager is responsible for sending and receiving data objects reliably. With each interface type is associated a set of protocols that can be used for data object transfer. Although Haggle does not depend on legacy communication stacks, it can make use of them transparently. The Protocol manager can therefore be seen as a convergence layer for different types of transport protocols. TCP is normally used for Ethernet and WiFi, whilst RFCOMM is used for Bluetooth (with Bluetooth, Haggle runs entirely without relying on the Internet stack). UDP or UNIX sockets are used for local inter process communication (IPC). Because the means of transfer is transparent to other managers, the Protocol manager can also support protocols such as BitTorrent, network coding schemes, and bundling protocols. Choosing the best protocol and interface for transfer is a *just-in-time* decision, which depends on the current policy issued by the Resource manager. The Protocol manager delegates the actual dispatching of data objects to one of its modules that implements the chosen protocol.

Application Manager: The application manager acts on behalf of applications inside the architecture. It implements a signaling protocol based on data objects with control attributes, which it uses to communicate with applications and implement the application API. The Application manager uses filters to demultiplex the applications' data objects based on the interests they register using the signaling protocol. Data objects for applications are thus first demultiplexed to the Application manager, which then relays them to the applications. Internal events can also be passed to applications that are interested in feedback on, e.g., neighbors that are discovered.

Data Manager: The data manager inserts data objects into the data store, and must first make sure they are valid. It performs checksum verification on data objects that contain checksum attributes. Applications may optionally attach checksum attributes when they generate data objects. This allows end-to-end detection of data corruptions, something which otherwise cannot be guaranteed, as data objects can become corrupted whilst stored in-between transfers.

Forwarding Manager: During node encounters, the Forwarding manager determines the data objects to disseminate to co-located neighbors. First it does non-delegate dissemination using the r_d and r_d resolve primitives. The decision to delegate data objects to neighbors is left to specific forwarding algorithms. They exist as manager modules that are invoked depending on the choice of forwarding algorithm. The Forwarding manager tunes the resolution queries to fit the resource policy and to achieve congestion control, as previously described in Section 4.

Security Manager: The security manager provides authentication of neighbors and performs integrity checks on incoming data objects, and may encrypt and decrypt data objects. The Security manager may insert a public key in the local node description. The keys acquired from received node descriptions are used for standard security functions. If a node can acquire a certificate for a neighbor from a trusted authority, it can be used with a public key to authenticate incoming node descriptions before they are accepted.

6 Implementation

We have created a cross-platform implementation of the Haggle architecture that runs on Linux, Windows, Mac OS X and Windows Mobile. There is also ongoing work to port it to Symbian. The code is written in C/C++, consists of about 20000 lines of code (excluding applications).

Haggle runs as a user space process with a main thread in which the kernel and managers run. Managers may run their modules in separate threads when they need to do work that requires significant processing time. This may include sending and receiving data objects, computing checksums, doing neighbor discovery, and so forth. New managers and modules can be added to Haggle with little effort, and that provides a straightforward path to extending Haggle with extra functionality that does not fit in applications. In terms of connectivity we so far support Bluetooth, Ethernet, and WiFi.

The data store is based on an SQLite [2] backend, which is suitable for small embedded devices. It runs in a separate thread since disk operations involve I/O that may take a relatively long time to complete, and

```
get_handle() → handle_t h
free_handle(handle_t h);
publish_dataobject(handle_t h, dataobject_t *dobj);
register_interest(handle_t h, char *name, char *value, int weight);
register_event_interest(handle_t h, int eventId, callback_t handler);
event_loop_run(handle_t h); → dataobject_t *dobj
event_loop_run_async(handle_t h); → dataobject_t *dobj
event_loop_stop(handle_t h)
```

Figure 8: Haggle application programming interface. Returned data is indicated with →.

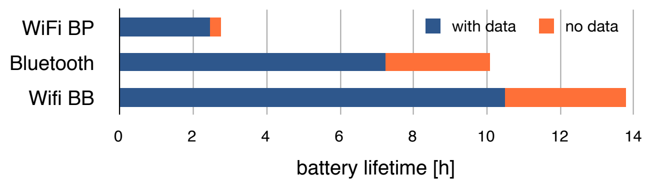

Figure 9: Battery lifetime of the HTC Touch Diamond.

would otherwise block the event queue. We currently do not implement aging, although we collect the necessary information, such as timestamps.

Applications interact with Haggle using IPC provided by a C-library, called *libhaggle*, which also exposes the pub/sub inspired API shown in Figure 8. The API is minimal and makes it easy and straightforward to write applications. The libhaggle library can be wrapped in various other programming languages, such as C# and Java.

7 Experimental Evaluation

In this section we provide an experimental evaluation of Haggle in a typical office environment. We evaluate the ability of Haggle to disseminate Garfield comic strips from a single laptop data source to seven mobile phones using Bluetooth. The phones run PhotoShare and are carried by research colleagues during a day of normal office activities. These include meetings, office work, and lunch outside the offices. Each participant has entered the attribute `Picture=garfield` into PhotoShare to express an interest in the Garfield strips, which they can also view within PhotoShare when received. Every time a phone encounters another phone, or the laptop, they both exchange node descriptions and perform search-based resolutions to determine if they have any strips to further exchange. As every participant has entered the same interest, the strips should ideally spread to all mobile phones; either directly from the source, or via continuous resolutions over the phones. We limit resolutions to include only the first ten results (i.e., strips) that have not yet been received by a peer. Thus, a node will never receive more than ten strips at a time.

7.1 Power Consumption

A limiting factor in our experiment is the battery lifetime of the mobile phones. We use three HTC Touch Diamonds and four HTC S-620. The Diamond is a more advanced phone than the S-620, but its battery lifetime is also significantly shorter, lasting only the duration of the live experiment (the S-620 lasted more than 24 hours).

The first thing we hence examine is Haggle's effect on a mobile phone's power consumption. We benchmark a single HTC Touch Diamond mobile phone using both Bluetooth and WiFi (although we do not use WiFi in the live experiment). There are three power mode settings for WiFi on the Diamond: best battery (BB), best performance (BP), and what we call auto mode. Auto mode adjusts the power against signal quality, and for coherent results we show only BB and BP. Bluetooth only has one mode of operation.

As a baseline, we run a phone isolated with only neighbor discovery on either Bluetooth or WiFi. With Bluetooth, Haggle performs a device scan every 60 ± 45 seconds and with WiFi it sends a broadcast beacon every 5 seconds. This gives us an estimate of the impact of neighbor detection. To compare with the baseline we add the laptop which generates a new 30-60 KB comic strip every minute. The strip is transferred to the phone that is placed close to the laptop. The application data rate is quite high for the scenario, but it gives us an estimate of the impact of data traffic in a busy environment.

Figure 9 shows the results from the power benchmarks. The most striking result is the short battery lifetime of WiFi BP. The 2-3 hours of running time is clearly too short for any realistic network scenario. WiFi BB, on the other hand, has the best lifetime of all modes, including Bluetooth. However, the BB mode operates at very low power output and the mobile phone and laptop have to be placed very close to each other for reliable data transfer. This limits the usage of BB mode in practice, but in combination with BP it effectively shows the upper and lower bound of auto mode.

Bluetooth costs more in terms of neighbor detection compared to WiFi BB. On the other hand, we actually found Bluetooth to be more useful for data transfer due to increased range over the WiFi BB mode. Neighbor detection with Bluetooth is not as reliable as WiFi, because a device cannot be detected while scanning. Therefore, the scan collisions increase with the density of the network. We find that Bluetooth provides a reasonable trade-off between device longevity and service provided, lasting up to seven hours with data. This is enough to last a normal working day, without frequent recharging. Bluetooth is therefore our technology of choice for prolonged experiments.

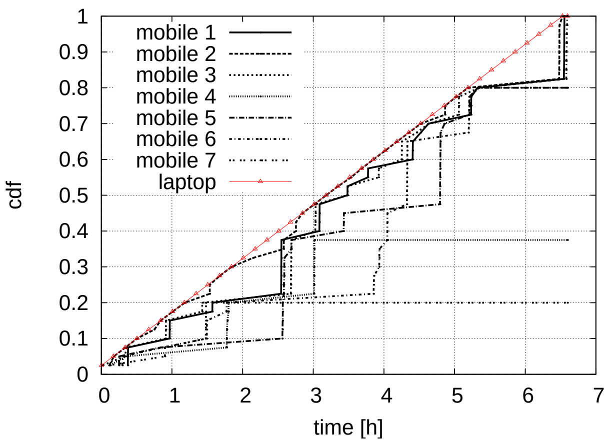

Figure 10: Fraction of data objects received over time.

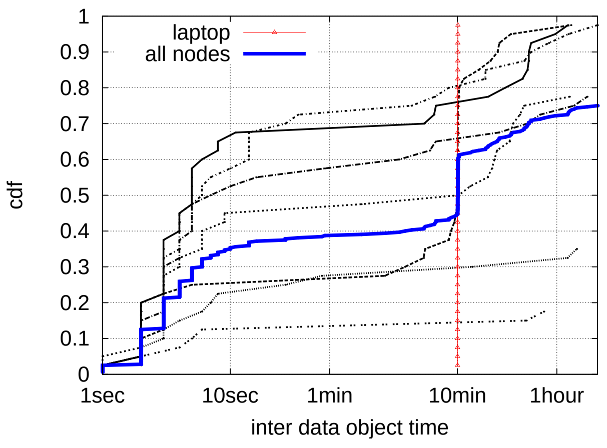

Figure 11: Distribution of inter data object times.

7.2 Live Experiment

For the live experiment we lowered the intensity of the comic strips to one every 10 minutes and increased the Bluetooth scan interval to 80 ± 60 seconds. We chose this longer interval such that devices are less likely to scan at the same time when we have several phones.

In terms of battery lifetime our reference phone managed 6.5 hours, which is just slightly shorter than the static setup with data. Although we use lower data and scan rates, the live experiment have frequent neighborhood changes that cause failed transfers and hence retransmissions. By involving several phones, we also increase the cost of neighbor detection since the phones must also respond to scans.

During previous experiments we noticed a problem with the Windows Mobile Bluetooth stack in that it sometimes goes down and resets to OFF mode. Thus, for the experiments we instructed our colleagues to be observant of any changes in the Bluetooth settings, and to turn Bluetooth back on if found in OFF mode. However, sometimes the phones may still indicate ON mode after the stack goes down, and this is hard to detect. The only solution in this case is to turn Bluetooth off and on again. This problem have an effect on our results in that phones on rare occasions miss contact opportunities or, in worst case, are unconnected for prolonged periods of time.

7.2.1 Delivery Fraction

In Figure 10, we see the fraction of received data objects over time, up until 6.5 hours when the experiment ended. The laptop is the reference as it gets the data objects directly from the application that generates them. Ideally, all phones should have the same delivery fraction as the laptop. The figure shows the different connectivity of the mobile phones. For example, mobile 2 is a phone that we know was co-located in the same room as the laptop for the most part of the experiment. Mobile 7, on the other hand, was isolated in another office and see very little progress (only at the beginning of the experiment). Three phones receive all data objects by the end of the experiment. Mobile 7 and mobile 4 only receive 20% and 30% of the data objects, respectively. However, we do not believe isolation is the only explanation for these low delivery fractions, but instead we suspect that the above mentioned Bluetooth problems also play a role.

7.2.2 Traffic Pattern

Figure 11 shows the distribution of inter data object receive times. The source has the expected constant 10 minute interval. Most of the mobile phones receive around 50% or more of their data objects with an interval less than 10 seconds. This shows that many data objects are sent and received in bursts when a phone co-locates with a new neighbor. With search-based resolution, the cost of a resolution is independent of the number of data objects resolved (it is only related to the size of the relation graph). Therefore, we argue that search-based resolution fits well the intermittent connectivity of opportunistic networks, where bursty traffic is common.

7.2.3 Hop-count and Delay

A question is how often a mobile phone gets its data objects from the laptop, or another mobile phone that forwards the data objects? The answer we find in Figure 12 (left), where the data object hop count distribution is shown (we measure the hop count once for each receiver of the same data object). 65% of the received data objects are forwarded over multiple hops, showing that Haggle provides a multihop dissemination service

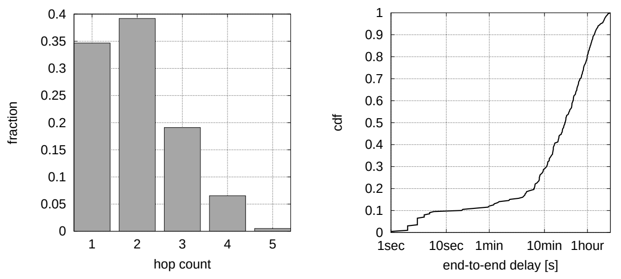

Figure 12: Hop distance and end-to-end delay.

for the majority of data. In terms of delay (Figure 12 right), 30% of the received data objects are delivered within 10 minutes, while the rest are delivered within 2 hours.

From these results we conclude that Haggle's continuous resolution system works well for providing basic interest forwarding.

8 Conclusion and Future Work

We have presented the Haggle architecture, which we have evaluated through live experiments. From the results we conclude that Haggle can be deployed on mobile phones in order to provide them with an opportunistic communication capability. Our implementation shows that Haggle has a relatively small impact on battery life, compared to the basic cost of neighbor detection, when running live and disseminating data. We plan to release the Haggle source code to the public under a free software license so that it can be reused by other researchers and application developers. We believe this significantly lowers the bar to experimentation with opportunistic network applications.

We believe Haggle promises exciting future research. For example, an interesting problem related to Haggle is how to do more flexible matching. Inexact keyword matching and automatic searches on related keywords are mechanisms that could greatly enhance the Haggle experience. Another example is how to opportunistically make use of infrastructure access when available. This provides an opportunity to fetch information from the Internet that can be further disseminated by Haggle. In this context, it is also important to have efficient ways to automatically extract relevant metadata.

These examples, and many other ones, are topics that we may explore in the future.

References

[1] Google desktop. http://desktop.google.com/.

[2] SQLite database engine. http://www.sqlite.org/.

[3] Adjie-Winoto, W., Schwartz, E., Balakrishnan, H., and Lilley, J. The design and implementation of an intentional naming system. In *ACM Symposium on Operating System Principles* (December 1999).

[4] Brin, S., and Page, L. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems 30*, 1-7 (1998), 107–117.

[5] Carzaniga, A., and Wolf, A. L. Forwarding in a content-based network. In *SIGCOMM* (August 2003).

[6] Demmer, M., Fall, K., Koponen, T., and Shenker, S. Towards a modern communications API. In *Sixth Workshop on Hot Topics in Networks (HotNets-VI)* (November 2007).

[7] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. The many faces of publish/subscribe. *ACM Computing Surveys 35*, 2 (June 2003), 114–131.

[8] Faber, R. B. T., and Handley, M. From protocol stack to protocol heap - role-based architecture. In *HotNets-I, Princeton, NJ* (October 2002).

[9] Fall, K. A delay-tolerant network architecture for challenged internets. In *ACM SIGCOMM'03* (August 2003).

[10] Ford, B., Strauss, J., Lesniewski-Laas, C., Rhea, S., Kaashoek, F., and Morris, R. Persistent personal names for globally connected mobile devices. In *USENIX Symposium on Operating System Design and Implementation (OSDI)* (November 2006).

[11] Gifford, D. K., Jouvelot, P., Sheldona, M. A., , and Jr., J. W. O. Semantic file systems. In *ACM Symposium on Operating System Principles* (1991), pp. 16–25.

[12] Keshav, S., Chawathe, Y., Chen, M., Zhang, Y., and Wolman, A. Panel: Cell phones as a research platform. MobiSys Panel discussion, June 2007. http://www.sigmobile.org/mobisys/2007/mobisyspanel.pdf.

[13] Lenders, V., Karlsson, G., and May, M. Wireless ad hoc podcasting. In *IEEE Communications Society Conference on Sensor, Mesh, and Ad Hoc Communications and Networks (SECON)* (June 2007).

[14] Saltzer, J. On the naming and binding of network destinations. IETF RFC 1498, August 1993.

[15] Soules, C. A. N., and Ganger, G. R. Connections: Using context to enhance file search. In *ACM Symposium on Operating Systems Principles* (October 2005).

[16] Su, J., Scott, J., Hui, P., Crowcroft, J., Diot, C., Goel, A., de Lara, E., Lim, M. H., and Upton, E. Haggle: Seamless networking for mobile applications. In *Proceedings of UbiComp 2007* (September 2007).

A Appendix

A.1 Search-based Resolution Primitives

Let V denote the set of vertices that correspond to data objects in a relation graph (we may use vertices and data objects interchangeably). A denotes the set of attributes over all $v \in V$, and $A_v \subseteq A$ is the set of attributes in the metadata of data object v. The relation graph is a multigraph $G_R = (V, E)$, where an edge $e = uv$ between vertices $u, v \in V$ is part of E if and only if $|A_u \cap A_v| \geq 1$. In other words, G_R is a graph of data objects V, which are pair-wise connected via edges in E only if a pair of data objects share at least one attribute. We define three basic primitives on G_R:

Insert: We define an *insertion* $i : G_R \to G''_R$ such that G''_R is a relation graph, $G_R \subseteq G''_R$ and $|G''_R| - |G_R| = 1$.

Filter: Let $A_f \subseteq A$ be a set of filter attributes. We define *filtering* $f : G_R \to G'_R$, such that $G'_R = (V', E')$ is an *induced subgraph*[1] of G_R and $\forall\, v \in V' : A_f \subseteq A_v$.

Query weighting: We define *query weighting* $q : G_R \to G_Q$, such that $G_Q = (G_R, \omega)$ is a weighted relation graph defined by the map $\omega : \vec{E} \to \mathbb{R}$, which we call a weighting function. Note that ω is defined independently for the two directions of an edge. In this paper we use a weighting function

$$\omega(\vec{uv}) = \sum_{a_k \in (A_u \cap A_v)} \alpha(w_k^u),$$

where $\vec{uv}$ is the directed edge from u to v, and $w_k^u \in \mathbb{N}$ is the k:th weight in a set W_u associated with u, where $|W_u| = |A_u|$. The function $\alpha \to \mathbb{R}$ on w_k^u is defined by the resolver. Figure 13 illustrates two example query weightings q_1 and q_2 on a relation graph.

We use the basic primitives defined so far to define the higher level primitives demultiplexing and resolution.

Demux: Let $G_R = (V, E)$ be a relation graph and C a set of filter owners where $\forall c \in C : A_c \subseteq A$ is the set of interest attributes of c. We define *demultiplexing* $d : G_R \to D$, such that $D = V(f(G_R, A_f))$ for some filter f with associated attribute set A_f, and $\forall c \in C : A_f \subseteq A_c$.

Resolve: Let $G_Q = (V, E, \omega)$ be a weighted relation graph, $s \in V$ a fixed vertice, and $(S, \overline{S})$ a cut in G_Q, where $s \in S$. We define a *resolution* $r : G_Q \to D^<$, such that $D^< = S \setminus s$ is an ordered set and $\forall v \in S : \{\delta(v) > \psi, rank(\delta(v)) \leq \rho\}$. The function $\delta : V \to \mathbb{R}$ is

[1]Given a graph $G = (V, E)$, then G' is an induced subgraph of G if $G' \subseteq G$ and G' contains all edges $uv \in E$ with $u, v \in V'$.

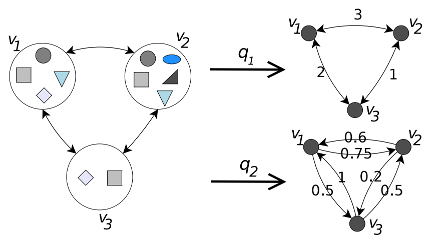

Figure 13: Two different query weightings q_1 and q_2 on G_R, using $\alpha_{q1}(w_k^u) = 1$ and $\alpha_{q2}(w_k^u) = \frac{1}{|A_u|}$, respectively.

a map, which we call a vertice weighting function, and $rank : \mathbb{R} \to \{1, 2, \ldots, |V|\}$ is a map that ranks vertices in V in order of decreasing weight. The parameters ψ and ρ are constants that decide the cut $(S, \overline{S})$.

A.2 Resolution Example

Here we formally describe the resolution example from Section 3.3. The resolution r_d with a given data object s (which is a node description) is used in a **resolve** with ω based on

$$\alpha(w_k^u) = \begin{cases} 1 & \text{if } u \in N(s), \\ 0 & \text{otherwise.} \end{cases}$$

where $N(s)$ is the set of node descriptions that are neighbors to s in the graph. On the resulting graph we weight the vertices using a vertice weighting function

$$\delta(v) = \sum_{e \in E_{\vec{uv}}} \omega(e),$$

where $E_{\vec{uv}}$ is the set of incoming edges to v (i.e., we compute the sum of incoming edge weights). The resulting vertice weights are the ranks of each data object relative s. The parameter ψ determines the minimal edge weights required to make the cut. Figure 14 illustrates this resolution on a relation graph of six data objects with $\psi = 0.3$ and $\rho = 3$. In this case the result will be an ordered list (v_1, v_2). If the vertice weighting function is applied to outgoing edges $E_{\vec{vu}}$ instead, the result is the list (v_2, v_1).

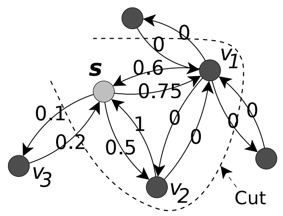

Figure 14: A resolution is a cut in the relation graph.