A Search-based Network Architecture for Content-oriented and Opportunistic Communication

Erik Nordström, Per Gunningberg and Christian Rohner
Uppsala University

Abstract

We describe Haggle, a network architecture that disseminates content among mobile users based on the match between their interests and the metadata of content. Central to Haggle's design is a set of search based networking primitives, which draw inspiration from web and desktop searching. These primitives enable, for instance, content to be disseminated in order of relevance to receivers, and to be asynchronously demultiplexed to applications. Other areas in which they enable novel solutions and abstractions include: naming and addressing, resolution, forwarding, and resource management. We describe how this is accomplished, and then illustrate the feasibility of our architecture through an implementation and evaluation. The results show that Haggle scales to hundreds of mobile devices, and thousands of pieces of content.

1 Introduction

Today's computing experience is to a large extent characterized by searching; it is used to efficiently locate and structure content on the web, as well as on our local computers. Despite the central role of searching, there is no explicit support for it in network architectures. In this paper, we therefore argue that it is beneficial to deeply embed search abstractions in architectures. Not only does this make it simpler to develop content oriented applications, but it also enables novel solutions to many networking problems, such as resolutions, demultiplexing, forwarding, and resource management. To support our claims, we present Haggle – a network architecture for opportunistic and content-oriented communication.

The first steps toward defining Haggle were taken in [17], and this paper provides a good introduction to the scenarios that Haggle targets, and the problems of the existing Internet architecture. These include the tight temporal and spatial coupling between applications and the underlying network [7], and the inflexible selection of transport methods and network interfaces. Here we take the next steps in the development of Haggle by incorporating search based resolution, which provides novel solutions to the problems above, as well as other ones. To describe these problems and the solutions provided by Haggle, we give a short overview of how Haggle operates.

Haggle devices persistently store content and metadata that they spread through disseminations. The contents should only be dissimenated to users that wish to receive it, according to their interests. Thus, Haggle devices that encounter each other first exchange interests. Once this is complete, they start to exchange the matching content. However, all the content can rarely be exchanged during a single interaction, and there may be several devices to consider simultaneously. A challenge is thus to decide which content to send, whom to send it to, and in which order it shall be sent.

We see these decisions as determining *bindings* between items of content and devices. The bindings should be deferred as long as possible due to the changing context (i.e., the stored content, metadata and in-situ state), which is important for making relevant bindings. There is hence a need to *resolve* the bindings dynamically as late as possible, and to order them according to the number of matching interests.

Traditional approaches to resolution, such as DNS and ARP, retrieve persistent bindings using *lookups*, whilst the context itself is non-persistent and forgotten once the lookup is complete. These approaches hence work inefficiently in Haggle, where the context is stored persistently, and bindings are resolved dynamically. The traditional approaches further do not provide the ability to order bindings.

Our solution is *search based resolution*. We draw inspiration from desktop search systems [1, 2]. They, intend to replace traditional file and path based name lookups with search based resolutions, in which files are located by matching keywords against their metadata.

We extend this approach by allowing local files to also represent other devices in the network. Only the associated metadata makes the actual distinction between devices and content. In the case of devices, the metadata represents, e.g., the interests of the device owner, and searching thus maps and ranks the relationships[1] between devices and content. We make these search primitives native to Haggle, and in the paper we describe how they work, and the advantages and challenges they bring. The new contributions we offer are as follows:

- We define a set networking primitives based on search based resolution (Section 2.3). These allow flexible and late binding between content and receivers.
- We describe a new architecture designed around search based resolution (Section 3). It embeds asynchronous and content-oriented communication, which is suitable for opportunistic networking environments.
- We implement Haggle and show the feasibility of the architecture through an evaluation (Section 4 and 5). The results show that search based resolution scale on constrained devices to hundreds of nodes and thousands of pieces of content.

The rest of the paper...

2 Design Overview

In this section we start by giving a high level overview of Haggle, followed by formal definitions of the search based networking primitives that underly its design. We then continue with describing in detail how the primitives are used in Haggle, and how they enable novel solutions to many networking problems.

2.1 Desktop Searching

Today's operating systems store files in a namespace of directory and file names that tell little, if anything, about the content of the files. The structure and format of any metadata in the files are proprietary to each application and file type, and therefore many disparate namespaces exists. There is no unified way to access content via metadata and searching is hence made unnecessarily difficult. Desktop search applications alleviate this problem by extracting metadata in a common format that can be easily indexed and searched. This metadata namespace is however optional and is not native to applications. As soon as data is transferred over the network it leaves this namespace, although it may perhaps enter it once again on another node. The data hence traverses several namespaces in which its metadata is not exposed. This hides potentially useful information that could otherwise be exploited in, e.g., forwarding. Online content is also indexed and searched by popular search engines like Google. But this is yet another metadata format and hence namespace that exists in parallel and is incompatible.

2.2 Unified Metadata Namespace

Haggle's metadata namespace aims to unify the disparate namespaces described above – application local ones as well as those in networks. By bridging traditionally separated namespaces we can build *relations* between entities, which otherwise would be difficult and also less flexible. A unified namespace connects spatially separated, but otherwise identical, namespaces (e.g., file systems on different physical devices), allowing relations to be established in a way that is indifferent to location. Note, however, that locality can also be expressed in metadata. But whether this metadata has a an impact on how relations are resolved is a late binding decision, as we show later.

The different entity types exist in the unified metadata namespace as *data objects* that are tuples $(metadata, data)$. The metadata describes an entity, whilst the actual data is optional. Although an entity could potentially be many different things, we limit ourselves here to two types; *content*, e.g., MP3-files, PDF documents, or JPGs, etc., and *nodes* that represent, e.g., computing devices, such as smartphones or laptops.

In the case of content, the metadata describes the content and may be extracted from the file when the data object is created. The metadata then travels with the content as the data object is forwarded in the network. In the case of nodes, the metadata instead expresses the interests that the node announces. In most situations, there is no data in data objects that represent nodes.

The data object is the single format in the namespace that penetrates the entire architecture, the network, and also applications. This makes Haggle a layer-less design. The metadata of data objects contain *attributes* in the form of name-value pairs. When two data objects share an attribute they have a *relation* in the namespace. Relations arrange data objects in a flat non-hierarchical way. The "strength" of a relation increases with the number of shared attributes.

An *actor* is an abstraction of a data object processing entity that act on the namespace. Actors may correspond to, e.g., users, applications, or processing threads, which may or may not be spatially or temporally co-located, i.e., on one node or on separate nodes. Actors

[1] Relations are, in this context, used synonymously with bindings.

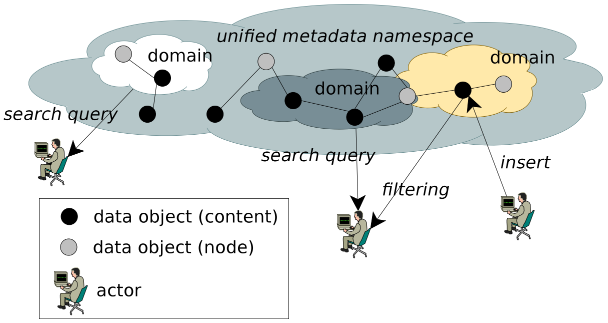


Figure 1: Metadata namespace with logical domains defined by searching and filtering.

can insert new data objects in the namespace and can also resolve the relations of data objects in the namespace. Data objects can have different relations depending on the "point-of-view" of an actor. Actors have the flexibility to choose this view, and also change it when they wish. We refer to a point-of-view as *domain*, which incorporates a subset of the data objects in the namespace, as illustrated in figure 1. Actors that insert and filter data objects in the same domain can signal each other, similarly to how layers signal each other in the TCP/IP stack. In order to do this, actors need primitives that allow them to resolve and filter the information in the namespace, such that they can structure it in a way that makes sense to them.

In the following section we formally define relations between data objects in the namespace and the primitives available to resolve and interpret information in it. We then look at how these primitives can be implemented in reality and how they provide novel solutions to many networking problems.

2.3 Search-based Networking Primitives

Let V denote the set of vertices that correspond to data objects in a unified metadata namespace (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. We define a *relation graph* over the metadata namespace as 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* $i : G_R \rightarrow G''_R$ 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* $f : G_R \rightarrow G'_R$, such that $G'_R = (V', E')$ is an *induced subgraph*[2] of G_R and $\forall v \in V' : A_f \subseteq A_v$.

Query: We define a *query* $q : G_R \rightarrow G_Q$, such that $G_Q = (G_R, \omega)$ is a weighted *query graph* defined by the map $\omega : \vec{E} \rightarrow \mathbb{R}$, 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 $\alpha \rightarrow \mathbb{R}$ on w_k^u is defined by the actor performing the query.

The *insertion* $i(G_R, v)$ inserts one new data object in the metadata namespace. We assume for now that data objects are never removed from the namespace.

The *filtering* $f(G_R, A_f)$, defines a domain in the namespace described by G_R, and which incorporates only those data objects that have all attributes of A_f. Figure 2 illustrates filtering. A relation graph G_R contains three data objects with associated attributes. A filter applied on G_R generates a subgraph that incorporates only those data objects matching all attributes in the filter's attribute set A_f.

The *query* $q(G_R, \omega)$ generates a weighted query graph. Figure 3 illustrates two example queries that weight a relation graph differently. The query graph generated by q_1 is acquired using a constant weighting function, where the weight of the edge uv in both directions will be equal to the number of shared attributes between u and v. The second query q_1, generates a query graph with edges $\vec{uv}$ weighted by the ratio between the number of shared attributes between u and v and the total number of attributes of u.

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

Demux: Let $G_R = (V, E)$ be a relation graph and C a set of actors where $\forall c \in C : A_c \subseteq A$ is the set of interest attributes of actor c. We define *demultiplexing* $d : G_R \rightarrow D$, 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 query graph, $s \in V$ a fixed vertice, and $(S, \overline{S})$ a cut in G_Q, where $s \in S$. We define a *resolution* $r : G_Q \rightarrow D^{<}$, such that

[2] 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 $u, v \in V'$.

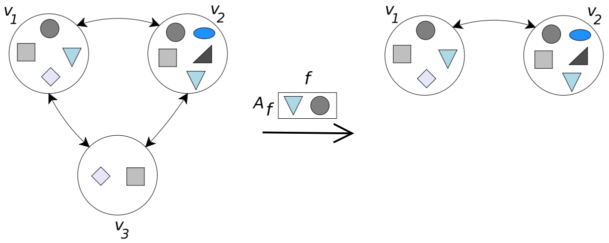


Figure 2: A filter operation f on a relation graph G_R, where vertices v_1, v_2, v_3 correspond to data objects with different, but overlapping, sets of attributes.

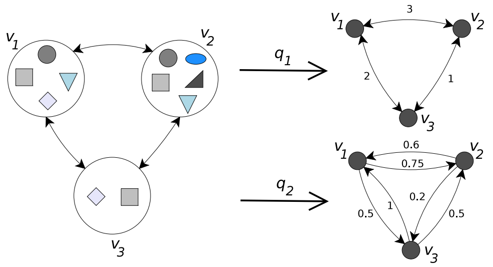


Figure 3: Two different query operations, q_1 and q_2, on G_R from figure 2, using $\alpha_{q1}(w_k^u) = 1$ and $\alpha_{q2}(w_k^u) = \frac{1}{|A_u|}$, respectively.

$D^< = S \setminus s$ is an ordered set and $\forall v \in S : \{\delta(v) > \psi, rank(\delta(v)) \leq \rho\}$. The function $\delta : V \rightarrow \mathbb{R}$ is a map, which we call a vertice weighting function, and $rank : \mathbb{R} \rightarrow \{1, 2, \ldots, |V|\}$ is a map that ranks vertices in V in order of decreasing weight. The parameters ψ and ρ are constants that decide the cut $(S, \overline{S})$.

The *demux* operation $d(f(G_R, A_f), C)$ demultiplexes data objects in the relation graph G_R to actors that have interests matching the filter attributes A_f.

The *resolve* operation $r(q(G_R, \omega), s, \delta, \psi, rank, \rho)$ returns an ranked list of data objects. The query graph determines the "strengths" of relations between data objects in a relation graph and the resolution r resolves and ranks particular relations from the point of view of s. The function δ determines (through weighting) the "importance" of each data object. The constants ψ and ρ determine the *cut* in the graph relative the data object s, by respectively eliminating data objects with too low weights, and setting an upper limit on the number of data objects returned.

The primitives defined allow a lot of flexibility, and to make things a bit more easy to grasp, we describe two common resolutions that actors execute: (1) given a data object, they wish to resolve the nodes interested in the data object, and (2) given a node, they wish to resolve the data objects matching the node's interests. We will refer to these resolutions as r_n and r_d, respectively. As nodes are data objects in the metadata namespace, albeit treated specially, these two cases correspond to essentially the same resolve operation; namely one that finds the data objects in the namespace with direct relations to a given data object, under some condition on the strengths of the relations.

Thus, given a data object s representing a node or a piece of content we can use a **resolve** with ω based on

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

and 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 and ψ a minimal edge weight. Figure 4 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 we change the vertice weighting function to use the set of outgoing edges $E_{\vec{vu}}$ instead, we get the list (v_2, v_1) as a result. Actors can hence themselves decide how resolutions are executed.

After having formally defined primitives that can be applied to the metadata namespace, we now instead turn to describing how these can be implemented and used in reality.

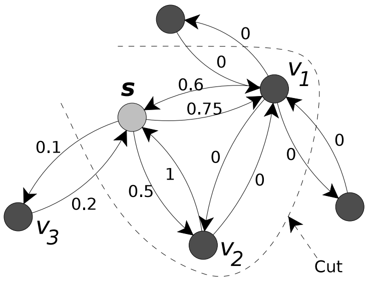


Figure 4: A resolution is a cut in the relation graph.

2.4 Persistent and Searchable Storage

To enable search-based resolution, each Haggle node maintains a *data store* that stores data objects *persistently* in a searchable state. The state of the data store is updated as nodes encounter each other in the network and and exchange data objects. A node's data store thus represents its current view of the global namespace, which of course at most times is incomplete. This does, however, not affect our search based networking primitives – it only limits the scopes of resolutions.

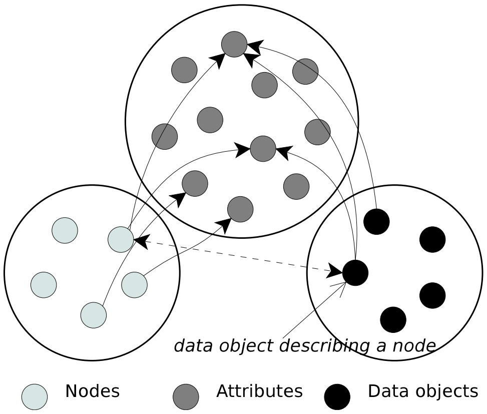


Figure 5: A logical overview of the searchable data store, showing example mappings between nodes and data objects via shared attributes.

Figure 5 shows a logical overview of how information is structured in the data store once data objects have been inserted. Separate data structures are used to represent nodes, attributes and data objects. As data objects are received, their metadata attributes are inserted into the attribute data structure and links are made to the actual data objects. If a data object represents a node, which is indicated by special metadata, a node object is also created. The node object is linked, via the shared attributes, to the original data object, as well as other data objects. Attributes thus connect nodes and data objects in the data store.

The **demux** and **resolve** primitives correspond in the data store to *persistent queries* and *non-persistent queries*, respectively. Persistent queries insert permanent *filters* into the data store that demultiplex data objects to actors *asynchronously*, at any point in time. Non-persistent queries, on the other hand, can only be applied to the data store *synchronously*, and immediately (as soon as the queries finish) return results. We now describe how these two networking primitives are used to implement the fundamental functions of Haggle.

2.5 Dissemination and Event Notification

The functionality of Haggle is split into two planes; the *event notification plane* that notifies actors of events they are interested in, and the *content dissemination plane* that disseminates content among the nodes in the network. Search based networking primitives are integral parts of both planes.

The event notification plane is a means for actors to signal each other – locally or remotely, and to subscribe to certain events. Persistent queries are part of this plane and they generate events when data objects match a filter. Actors insert data objects in the metadata namespace (i.e., in the data store), and at some point in the future the data objects might match a persistent query made by another actor – on the same node or on another node. The events generated by matches also pass on the matching data objects. Other types of events can also occur in the event notficiation plane, e.g., low battery power events from the operating system, and a external events; such as the discovery of a new neighbor node. We describe the event model in detail in section 3.

The content dissemination plane is based on non-persistent queries (i.e., resolve primitives). Dissemination resembles multicasting; every node data object with a relation to a specific data object defines a dissemination (or multicast) group. Note that the bindings between a data object and receivers are not determined only by the originator of the data object – they are resolved anew by every node that receives the data object as it is disseminated in the network. There are no "destinations" in the data object. The resolution that binds a data object to receivers can vary with the state of each node's data store, and the parameters they use in their resolve queries. But, if nodes exchange all their node data objects, the possible bindings will improve continuously as a data object is disseminated, i.e., more receivers will be resolved. However, it is up to each node to decide how much to disseminate.

Dissemination can involve forwarders with interests that do not match the data objects forwarded. We refer to these nodes as *delegate forwarders*, to distinguish them from nodes that only forward data objects that they also wish to receive themselves. A node may, of course, act as both a delegate and receiver at the same time, depending on the data objects forwarded. However, the basic dissemination that the plane carries out is non-delegate forwarding. We specifically discuss delegate forwarding in the next section.

To determine how to disseminate, two nodes that meet first exchange their so called *node descriptions*. These are the node data objects that represent them in the metadata namespace, and they contain, among other things, their interests that establish relations to other data objects. Once a node has received a neighbor's node description, and has inserted it into the data store, it can apply r_d to decide which data objects to forward. These may include other node descriptions it has received, as they are also data objects. Note that co-located nodes can rarely exchange all the data objects that match their interests, due to the duration of co-locations, bandwidth, and available storage. In this situation, an important concept of search based resolution is that data objects are ranked and exchanged in order of their rank. We call this *ordered forwarding*. If new data objects are inserted in the data store during co-locations, r_n can be used to determine if it is of interest to any current neighbors. Figure 6 illus-

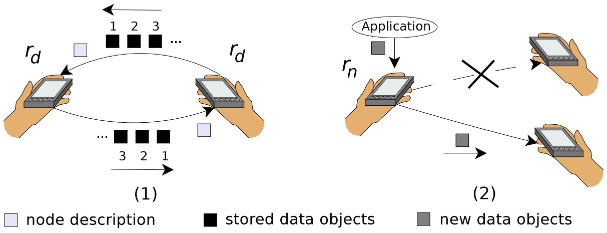


Figure 6: Resolutions in the content dissemination plane. (1) Node descriptions are exchanged, followed by data objects in order of rank (r_d). (2) A new data object is exchanged whilst nodes are co-located (r_n).

trates the different resolve operations that occur during co-locations.

2.6 Searching as a Networking Abstraction

The primitives of search based resolution enable novel solutions and abstractions to problems of mobility, intermittent connectivity, and data dissemination. In this section we discuss the broader implications of searching as a networking abstraction.

Naming & addressing: We break with "traditional" naming and addressing schemes that explicitly refer to, e.g., services or end-points. We instead use metadata to bind physical (or logical) entities through search primitives. The interests of a node (i.e., the attributes in its metadata) work as both a name identifier and address. The combined interests provide an identity, while the relations they establish determine a (logical) location, and thus an address.

Although file-sharing applications fit this naming and addressing scheme well, personal communication by default does not. However, point-to-point communication is really a dissemination to exactly one node. Addressing a specific node is hence a matter of compiling a set of attributes that match well that node's interests, and thus restricts the group of nodes involved in the dissemination (delegate forwarders may facilitate dissemination, although they are not part of it). The weights w_k^u, which are used in the **query** primitive, allow nodes to express their interest in metadata namespace more accurately. Imagine, for example, a data object that contains an email message destined for a specific node. A personal attribute like email="john.doe@haggle.org", should in this case weigh more than generic attributes, 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.

Note that only attributes in data objects that represent nodes are weighted. It does not make sense to weight attributes in data objects in general, since different nodes have their own view on their importance. Thus, if the email data object, in the example above, is a vertice v in a relation graph and u is a vertice representing John Doe, then only the edge $\vec{uv}$ is affected by the weighting, and not $\vec{vu}$.

Resolution & binding: An important concept of search based resolution is that bindings are done only at the time resolutions are performed. The 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 just fetch semi-static bindings that are predetermined. Search based resolution, on the other hand, allow late and flexible bindings that occur continuously as data objects are disseminated.

Another important concept is the ranking provided by search based resolution. It makes it possible to tune resolutions. For instance, to resolve a set of receivers of 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 n attributes in common.

Demultiplexing: Flexible filter based demultiplexing allow spatial and temporal decoupling of senders and receivers. For example, a data object that is inserted into the data store by an application may be demultiplexed immediately to another application on the same node, or whenever 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, as done in TCP/IP, binds a packet to one single remote application or service, already at the source node.

The comparison to port based demultiplexing is, however, not entirely fair. Port demultiplexing works on small application data segments and has to be very fast, and makes it easy to stream data directly to applications. The demultiplexing of a data object is not done until the entire application data unit is received. However, data objects can also be streamed if the metadata header is seen as an initiator for a stream session. The header is then received first, to determine demultiplexing, and then the stream is internally redirected to the receiver(s).

Forwarding: Determining *delegate forwarders*, i.e., nodes that carry data objects on behalf of others, is a task for forwarding algorithms. We anticipate that there is no single forwarding scheme that is suitable for all environments. However, search based resolution makes it possible to integrate a number of different schemes. We have defined the r_n and r_d **resolve** queries that determine direct relations in a relation graph. Similar *resolve* queries, e.g. r_f, can be defined to find delegate forwarders over several hops in the graph, although at the cost of increased resolution time. We do not address specific delegate forwarding algorithms in this paper.

Ordered forwarding is also an important concept enabled by search based resolution. In the literature, forwarding schemes often do not decide the order in which messages are forwarded, because they commonly assume that all messages can be forwarded during node co-locations. Ordered forwarding can better utilize time limited node contacts, by sending the most important data objects first.

Resource & congestion control: The content dissemination plane can be tuned according to the available resources of nodes; in terms of disk space, bandwidth and battery power. By expressing resource polices in node descriptions, a node can signal a neighbor, which then tunes its resolutions related to the signaling node, 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 it receives. Data objects are affected by the restrictions in order of least important 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. We believe it is possible to develop ways to do secret attribute matching, i.e., without revealing the semantics of the attributes. But, this is something we have to investigate further.

In a broader scope, we see that well established principles to establish trust and authentication, and guaranteeing data integrity, work well also in Haggle.

3 The Haggle Architecture

In this section we give a detailed description of the Haggle architecture and how it is designed around the search based networking primitives that we have defined.

3.1 The Core System

The Haggle architecture is data centric, event-driven and modular – features that allow it to scale from constrained systems to well provisioned ones. Central in the architecture is the *kernel*. It implements an event queue, over which processing entities, called *managers*, communicate. The kernel contains, apart from the event queue, a number of shared data structures, such as active neighbors, listening sockets, and also the data store. Figure 7 depicts how the kernel, managers and applications are located in the architecture, and how they can interact.

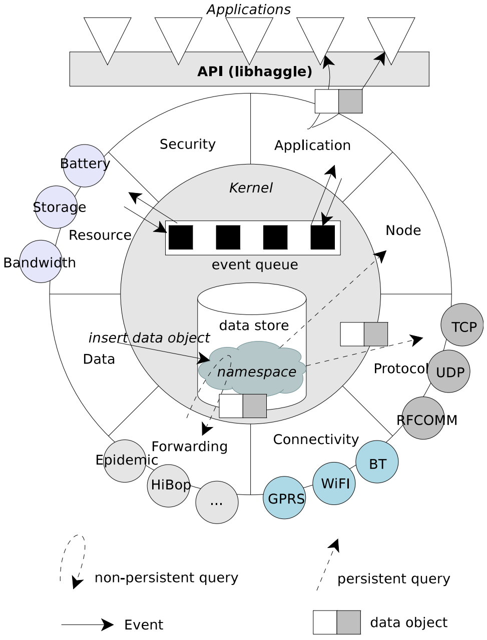


Figure 7: The Haggle architecture comprises a kernel and a set of managers.

Managers are instantiations of actors that operate within a domain of responsibility. They implement the functional logic of Haggle and interact only directly with the kernel. A manager is oblivious to anything outside its domain of responsibility and hence cannot directly call another manager. It is limited to producing and consuming events, and to interaction with the data store. This strict isolation makes it easy to add new managers and to plug them in dynamically – potentially even at run-time.

The circular structure illustrates the layer-less design; no manager is logically positioned on top of another one. Data objects are passed among managers unmodified, i.e., without additional headers, and to native applications as well. A manager might be dependent on other managers that generate events it consumes, but it knows nothing about how these events are generated or from where they originate. Managers can delegate processing tasks 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 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 I/O and therefore requires asynchronous operation through event callbacks. Both persistent and non-persistent queries are events in the event notification plane. A persistent query generates an event every time a data object matches its associated filter in the data store. Similarly, the result of non-persistent queries are returned in events, such that the queries do not block the system while being processed. The event model also fits nicely with the opportunistic nature of Haggle; physical node encounters are events that drive the system internally. Events are generated when devices are discovered in the neighborhood, and this triggers disseminations to occur. If the neighborhood is static and no applications generate new data, the system sits idle, thus preserving resources.

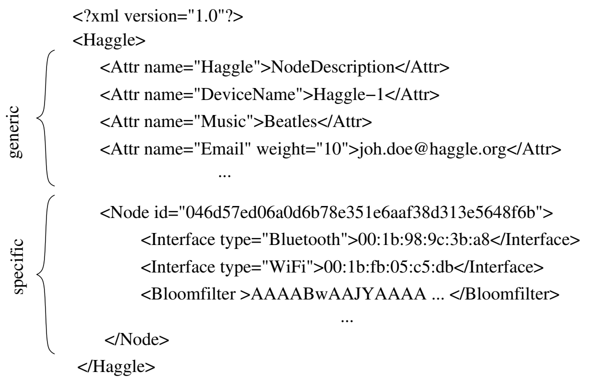
3.1.1 Data Centric

The data object is the single format of information exchanged in Haggle, and is also what defines the metadata namespace. Data objects spread among nodes as they encounter each other in the network. They enter and leave a node through a single point, and there are no layers that define an up or down, i.e., whether data objects come from applications or the network. There is hence no distinction made between data objects received locally (from applications) or those received from other nodes in the surroundings. Whether delivery is local or over the network is transparent.

To achieve efficient dissemination of data objects, Haggle is concerned with (1) how data objects relate to one another in the metadata namespace, 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 bindings internally: *Attributes* are part of the metadata of nodes and data objects, and can therefore establish relations between them, whilst also allowing filtering and demultiplexing. *Nodes* internally 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.

Whilst attributes are part of the metadata of data objects and nodes, the metadata itself is expressed in XML. This makes the metadata easy to parse, filter and search, using technologies such as XPath. Nodes and interfaces can also be expressed in metadata, and are transformed into internal objects when the data object is parsed. However, only attributes build relations between data objects in the metadata namespace, and other parts of the metadata are not visible in filtering and searching. These parts have to be exposed by attributes in case managers want to access the information. The metadata hence has a *generic* part, consisting of attributes, and a *specific* part consisting of other XML structures.

Figure 8 shows the structure of a *node description*, which represents a node in the metadata namespace.



```
<?xml version="1.0"?>
<Haggle>
    <Attr name="Haggle">NodeDescription</Attr>
    <Attr name="DeviceName">Haggle−1</Attr>
    <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>
          <Interface type="WiFi">00:1b:fb:05:c5:db</Interface>
          <Bloomfilter >AAAABwAAJYAAAA ... </Bloomfilter>
                                    ...
     </Node>
 </Haggle>
```

Figure 8: A data object metadata header in XML format, in this case a node description. The generic part describes the nodes interests and builds relations in the metadata namespace, whilst the specific part describes the node.

Data objects that contain node descriptions are demultiplexed based on the Haggle=NodeDescription attribute, and once acquired, the specific part can be accessed and transformed into an internal node object that is inserted into the data store. Managers on different physical devices can thus signal each other using a predetermined attribute, instead of using header encapsulations as layers do in the TCP/IP stack.

Data objects that contain actual content normally do not have a specific part, but applications may use the same principles as managers to implement signaling themselves. Or, they may just attach additional metadata that they do not want to be searchable.

3.1.2 The Data Store

The data store provides an interface that implements the search primitives described in section 2.6. The backend used to implement the primitives is hidden, such that several backends can be supported. A typical backend uses a relational database.

The data store does not store the actual data in data objects, only pointers to where the corresponding files are located 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. A popularity counter also measures how often data objects match queries. Popular data objects in this way age slower. The architecture can accommodate several ageing 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 more accurate resolution.

The data store manages the filters that are associated with persistent queries, and provides means to insert and remove new ones. The non-persistent query interface implements the r_n and r_d resolve primitives. As they are both very similar, we here only describe r_d. The resolve query interface is as follows: `resolve(node, max, match, ratio, callback);`. The `max` parameter sets the an upper limit on the number of data objects returned, `match` sets a lower limit on the number of attributes that have to match in a data object, and the `ratio` specifies the percentage of attributes in a data object that must match the node's interest. The `callback` parameter is the context used to return the result of the query. The result is a ranked list of matching data objects.

3.1.3 Events

Haggle specifies three event types: *public*, *private* and *callback* events. Public events are predefined events that managers can register interest in. These are listed in Table 1, along with producers and consumers and data type passed. Any data passed with public events is protected so that managers cannot change it. This is because several managers may be interested in the same public event and they therefore cannot manipulate the data that another manager may rely on. They can keep a private copy of the event data if necessary. Private events are, on the other hand, registered dynamically with the kernel, and are only known to the registering manager. This means that the data is also private and may be changed or kept for processing. Private events are used to implement timer based operations, such a garbage collection and beaconing. They also implement persistent queries, and each filter registered with the data store maps to at least one private event associated with a specific manager.

Callback events are one-time events that are non-persistent, i.e., they occur exactly once. As they are also private to managers, they can be seen as a type of private event. The difference is that they require no registration and the callback context (handler function) is passed along with the event instead of being stored persistently. Callback events are typically used to implement non-persistent queries, and occur only once per query made.

3.1.4 Data Paths & Processing Order

An implication of flexible demultiplexing is the lack of predetermined processing paths for data objects. They can be demultiplexed to several different managers, depending on the metadata, and in no specific order. This has two important consequences. First, a manager does not know what type of processing a data object has been subjected to. However, sometimes ordered processing is necessary. The Security manager should, for example, verify the integrity of data objects before they are processed further by other managers. Second, as several managers may show interest in the same public event, and may demultiplex the same data object, there will be many copies of data objects passed around[3]. This is obviously less efficient than a layered stack approach which passes data between layers in strict order. We hence trade some efficiency for flexibility.

The way we address these issues is to define public events that indicate the "state" of data objects. For instance, managers that rely on the Security manager only process data objects that are issued in events that indicate security has been considered. The first three events in Table 1, is an example of an *event ladder* that a data object climbs as it is received. The Protocol manager issues the first event as the data object is just 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's new state. The Data manager in turn inserts this data object in the data store and issues an event that clears it for "general processing". Managers that process any of these events need to account for the state of the data passed in them.

3.2 The Managers

After having described the core system we now turn to detailing the managers and the functionality they provide.

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 how to deal with resources in their domains of responsibility. Under resource constraints, this may include restricting disseminations, neighbor discovery, and choosing power efficient ways to transmit data objects. The Resource manager can also append resource control metadata to the node description, in order to signal its policy to neighbors, as discussed in section 2.6.

We learned early on that a distributed policy implementation is crucial for efficient resource management. Earlier experience with centralized policy control [17], involved a system where managers registered tasks with the Resource manager, which then scheduled them based on the current policy. Such a system has two major drawbacks. First, a centralized resource manager does not have a good understanding of how to efficiently schedule tasks that belong to another manager's domain of

[3] Only the metadata header is passed around, as the data is stored on disk.

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 object
Local Interface Down	Connectivity	Protocol	Interface object
New Contact	Connectivity	Node, Forwarding	Node object
End of Contact	Connectivity	Node	Node object
Send Data Object	Any	Protocol	Data object
Resource Policy	Resource	Any	Policy object
Data Object Targets	Data Store	Forwarding	Data object

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

responsibility. Second, managers have equivalently bad understanding of the policies in effect, which means they might register tasks that never run (or run too late) under a certain policy. This effectively wastes 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 a WiFi or Ethernet, beacons are instead sent. 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. Figure 8 shows an example node description. From received node descriptions, the Node manager creates internal node objects that it inserts into the data store. The node description contains a bloomfilter [5], which encodes the data objects a neighbor has in its data store. This is used to avoid sending data objects that a neighbor node already has received.

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. TCP is normally used for Ethernet and WiFi, whilst RFCOMM is used for Bluetooth. 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.

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. Internally, applications are represented by node objects[4], and can therefore be addressed as any other nodes. This method of signaling is hence oblivious to whether applications are running on the local device or on remote devices. The Application manager uses filters to demultiplex incoming signaling data objects sent by applications. It also demultiplexes the applications' data objects based on filters 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. The Data manager may also implement end-to-end acknowledgements for data objects.

Forwarding Manager: During node encounters, the Forwarding manager determines the data objects to dis-

[4] Application node objects are not inserted into the data store, and are therefore not exposed in the metadata namespace. The node description aggregates the interests of applications instead.

seminate to co-located neighbors. First it does non-delegate dissemination using the r_d and r_d resolve primitives. The decision to delegate other 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, and the current resource policy. The Forwarding manager tunes the resolution queries to fit the resource policy, and to achieve congestion control, as described in section 2.6.

Running several forwarding schemes in parallel may, however, lead to problems. First, the type of forwarding a data object is subjected to needs to be consistent throughout the data object's life-time in the network. Otherwise, different algorithms may counter-act one another. Second, algorithms that account for resources in their forwarding decisions, for instance bandwidth, may work badly if they do not have exclusive control over the resources. Greedy algorithms may starve other ones, or make them behave inconsistently. We are investigating solutions to these problems.

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 inserts a public key in the local node description as an attribute, and demultiplexes incoming node descriptions based on the same attribute. The keys acquired from received node descriptions are used for standard security functions. If a node can acquire a certificate for a neighbor in some other way, it can be used xwith the public key to authenticate incoming node descriptions before they are accepted.

4 Implementation

We have created a reference implementation of the Haggle architecture that runs on several platforms, including Linux, Windows, Mac OS X and Windows Mobile. There is also ongoing work to port it to Symbian, as our main target platform is mobile phones. The code is written in C/C++, consists of about 11000 lines of code (excluding applications), and implements all the basic managers, except the Security and Resource manager. We intend to implement them as well, but they are not essential for illustrating the feasibility of a search based network architecture.

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, etc.

Applications interact with Haggle using IPC provided by a C-library, called *libhaggle*, which also exposes the publish-subscribe inspired API shown in Figure 9. This API provides a clean embodiment of temporal and spatial decoupling [7]. Applications can publish data objects and add interests, which are then appended to the *node description*. The applications asynchronously receive data objects that match their interests, and can also register to receive other events. The libhaggle library makes it easy to write applications in various programming languages, and we have so far written applications in C/C++ and C#.

```
get_handle() → 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, event_handler_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 9: Haggle application programming interface. Returned data is indicated with →.

The current implementation supports Bluetooth and WiFi connectivity, as this is a prevalent technology on smartphones. Bluetooth is, however, preferred, as we found the current WiFi technologies on smartphones to be too power inefficient to be a viable option. We currently do not implement any forwarding algorithms that do delegate forwarding.

The data store is based on an SQLite [3] 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 would otherwise block the event queue. We currently do not implement ageing, although we collect the necessary information, such as timestamps.

We have developed two native applications for Haggle; *FileDrop* and *PhotoShare*. FileDrop is a file sharing application that monitors a directory on the file system. When files are put in the directory, e.g., through *drag-n-drop*, they are converted into data objects and inserted into Haggle. Metadata is automatically extracted from the files and they spread to other nodes according to this metadata. Received data objects also end up in the shared directory. PhotoShare is a Windows Mobile application that allows a user to share pictures taken with the phone's camera. The user can add its own metadata to the picture before it is turned into a data object that spreads to other Haggle devices. Neither FileDrop nor PhotoShare rely on other devices to run the same applications. Any device that exposes matching interests in its node description will receive the data objects.

Legacy applications can be supported by Haggle through application proxys. In the email scenario, which was discussed in section 2.6, a proxy application creates data objects from emails, using the email header as metadata. The proxy also registers the user's email address as a weighted interest attribute. Any data objects that contain the user's email address will hence be received by the proxy application, which can then pass on the enclosed email to an email client.

5 Evaluation

[THIS SECTION IS WORK IN PROGRESS]
We investigate how search based resolution scales with the amount of information in a node's data store. The query time should not be a significant factor relative the time of node encounters, and should scale well with the size of the data store. Queries should neither consume too much resources, in terms of memory and CPU, such that they make search based networking infeasible on constrained platforms.

To perform the measurements, we designed a Benchmark manager that inserts fake data objects and nodes into the data store, and then runs a set of queries that try, for each node, to resolve the matching data objects. We measure the time to complete each query, and the number of data objects returned. The Benchmark manager randomly picks attributes from a pool A of size m, which it then copies into a set of data objects D, 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 is created, where each node $n \in N$ has $|A_n| = k$. Note that we never remove attributes from the pool, such that nodes and data objects may share attributes between them. We do, however, make sure that attributes are not duplicated within individual nodes and data objects. The probability $P^*_{nd}(x)$ that a node n shares x attributes with a data object d is

$$P^*_{nd}(x) = \frac{\binom{l}{x}\binom{m-l}{k-x}}{\binom{m}{k}}$$

and the probability P_r that they have a relation is therefore

$$P_r = 1 - P^*_{nd}(x = 0) = 1 - \frac{\binom{m-l}{k}}{\binom{m}{l}} \approx 1 - \left(\frac{m-k}{m}\right)^l$$

We run benchmarks on two hardware configurations. The first is a MacBook Air laptop with a 1.8 GHz Intel Core 2 Duo CPU, 2 GB memory and 80 GB hard drive. The second is a Samsung SGH-i600 Windows Mobile 6 smartphone, with 64 Mb RAM and a 220 MHz Texas Instruments OMAP 1710 CPU. We use two attribute pool sizes m set at 1000 and 10000 attributes, whilst we keep the number of nodes fixed at 100, with each node having $k = 100$ interest attributes. We use either 5 or 10 attributes in each data object and then vary the number of data objects between 10 and 10000.

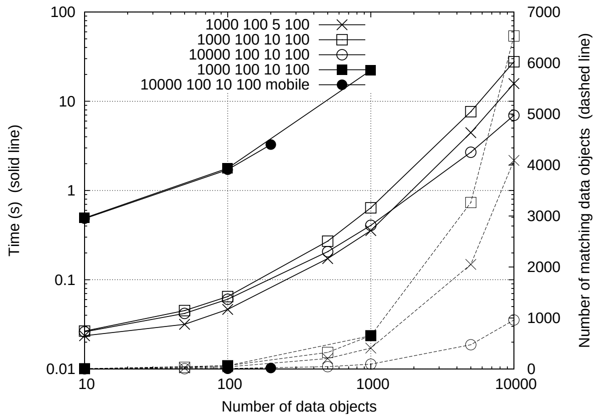


Figure 10: Query time scaling with size of the data store (log-log scale).

Figure 10 shows the results from the different runs. Solid lines indicate query time and dashed line the number of data objects returned. Note that we did not limit the number of data objects returned, since we wanted to see how it scaled with the amount of information in the data store.

The results show that the smartphone is an order of magnitude slower than the laptop. We believe a reasonable sized data store lies between 100 and 1000 data objects, based on the power and amount of storage on a mobile phone. We then have an upper bound on the query time at around 20 seconds with 100 known nodes. This might seem a lot, but for most scenarios the demands will probably be less, and the query time around a couple of seconds or more. Also note that there is only one query to determine all data objects to be exchanged with a neighbor node. The initial cost in time is hence amortized over all the data objects exchanged. We see that with increased attribute pool size, the query time decreases. This is because it reduces the likelihood of a match between a node and a data object. With 5000 and 10000 data objects the query time is reduced by several seconds, and the number of data objects returned is similarly reduced.

We also tried to limit the number of data objects returned in the query, but this did not significantly affect the query time, since each data object has to be matched in the data store anyway. We therefore conclude that it is the likelihood of a match between nodes and data objects that determines the query time.

The evaluation shows the feasibility of search based resolution. Although resolution time is significant on large namespaces and constrained devices, we think that it is not large relative the time it takes to discover neighbors and transmit the content. We also believe there is room for optimizations in the future as we develop our implementation.

6 Related Work

Haggle builds broadly on two previous bodies of work. The first comprises works that incorporate metadata based namespaces, and search primitives. The second body consists of new network architectures and paradigms. We discuss them in the order mentioned.

The semantic file system [13] uses an attribute based namespace to improve the structure and organization of files, making them more easily searchable. Desktop search tools, such as Google desktop [2], represent an alternative approach that builds search indices based on metadata extracted from files. Connections [16], enhances this type of searching by building temporal relations between files. It defines a relation-graph that resembles the one we use. EnsemBlue [15] is a network file system that uses a spanning namespace to organize data on ensembles of consumer devices. They use persistent queries to notify applications of new files added to the ensemble. INS [4] defines a hierarchical attribute namespace that uses *route-by-name* lookups in an overlay network. Haggle shares similarities with all these namespace approaches. However, it differs in one or more of the following ways. 1) it builds relations not only between content, but also mobile devices that are not co-located at all times, 2) its metadata namespace is flat and supports both content metadata and contextual metadata, 3) resolution is done using searching instead of name-to-location lookups.

Publish-subscribe (pub/sub) [8] is a broad term for systems and network architectures that temporally and spatially decouple the subscribers of content from the publishers. Linda [12], is a programming language for distributed and parallel computing that pioneered this approach. Although Haggle incorporates these concepts, along with a pub/sub inspired API, it differs in how it disseminates content. Pub/sub systems either disseminate based on exact matching filters [6], or channels of topics [14]. We use filters only for local demultiplexing and instead use in-exact searching with ranking in disseminations.

The *role-based architecture* (RBA) [9] organizes communication in functional units called *roles* instead of layers. Packets carry metadata in the form of role-specific headers (RSHs). Haggle generalizes RBA by collapsing the RSHs into metadata headers without predefined structure. The roles of RBA correspond well to our managers, but we do not as rigidly bind managers to certain parts of data objects, 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.

The delay tolerant network architecture (DTN) [10] defines a bundle layer that incorporates a late binding addressing scheme based on end-point identifiers (EIDs). The Unmanaged Internet Architecture (UIA) [11], also uses EIDs which map to personal names that can organize devices in groups. Haggle shares the late binding mechanisms with these architectures, but neither does bundling nor uses EIDs.

Su et al. presented in [17], the previous Haggle architecture. It used INS-inspired naming and was not designed around search based resolution, which is our main contribution. Although we are inspired by concepts and terminology from that work, our architecture is a clean slate design that differs significantly. For example, our architecture is event-driven, use a different data object format, and use decentralized resource management.

7 Discussion and Conclusions

We have presented Haggle, a network architecture designed around search based networking primitives. These primitives enable spatial and temporal decoupling of senders and receivers and provide flexible resolution mechanisms. Most importantly, the resolved data is ranked, giving the resolver better control of, e.g., how to disseminate the data.

These search primitives allow novel solutions to several issues, which so far, have received limited attention in related work, such as ordered forwarding and congestion control. In future work we aim to investigate the details of these mechanisms, as well defining resolutions for delegate forwarding algorithms.

We also want to find ways to interface mobile Haggle devices with infrastructure networks. It is clear that Haggle devices will move in areas with infrastructure connectivity, such as cellular networks. Haggle complements that type of networking by leveraging social networking aspects, for example, the fact that data exchanges are usually correlated with physical proximity. Infrastructure access can be used to transmit small end-to-end acknowledgements, whilst bulk data transfer is performed peer-to-peer, without involving expensive and slow infrastructure networks.

References

[1] Apple spotlight. http://www.apple.com/macosx/features/300.html#spotlight.

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

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

[4] ADJIE-WINOTO, W., SCHWARTZ, E., BALAKRISHNAN, H., AND LILLEY, J. The design and implementa-

tion of an intentional naming system. In *ACM Symposium on Operating System Principles* (December 1999).

[5] BLOOM, B. Space/time trade-offs in hash coding with allowable errors. *Communication of ACM 13*, 7 (July 1970), 422–426.

[6] CARZANIGA, A., AND WOLF, A. L. Forwarding in a content-based network. In *SIGCOMM* (August 2003).

[7] 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).

[8] 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.

[9] FABER, R. B. T., AND HANDLEY, M. From protocol stack to protocol heap - role-based architecture. In *HotNets-I, Princeton, NJ* (October 2002).

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

[11] 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).

[12] GELERNTER, D., AND BERNSTEIN, A. J. Distributed communication via global buffer. In *Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing* (1982), pp. 10–18.

[13] 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.

[14] 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).

[15] PEEK, D., AND FLINN, J. Ensemblue: Integrating distributed storage and consumer electronics.

[16] SOULES, C. A. N., AND GANGER, G. R. Connections: Using context to enhance file search. In *ACM Symposium on Operating Systems Principles* (October 2005).

[17] 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).