Haggle: Relevance-Aware Content Sharing for Mobile Devices Using Search

Erik Nordström
Princeton University
enordstr@cs.princeton.edu

Per Gunningberg
Uppsala University
perg@it.uu.se

Christian Rohner
Uppsala University
chrohner@it.uu.se

ABSTRACT

In this paper we present **Haggle**—an *ad-hoc* content sharing system for mobile devices. We describe the challenges with designing Haggle in face of unpredictable mobility and time-limited node contacts, and present the design choices made to cope with this environment. In particular, we focus on the importance of prioritizing the sharing of *relevant* content when node contacts are time limited.

Haggle leverages *search* to disseminate content based on its *relative relevance* to interested parties, by matching locally stored content against the interests of users. Search enables dissemination of content in order of how strongly users desire it, offering delay and resource savings by prioritizing the content that matters.

We evaluate Haggle through a real-world mobile phone experiment, complemented by trace-based emulations. Our results show that a content item whose *interest group* (the nodes that desire it) has strong interests is delivered with lower delay, and to a higher fraction of members, than an item with a similar sized group with weak interests.

1. INTRODUCTION

Mobile devices, such as smartphones and tablets, have increased the opportunities to access and share content while on the move. Yet, most sharing schemes rely on intermediary services, such as Facebook and Twitter (for sharing status updates, pictures, and videos), and infrastructure for the underlying communication. The increasing dependence on online services puts strain on wireless access networks (*e.g.*, 3G), forcing providers to offload traffic onto local networks (*e.g.*, WiFi)—which may also be overloaded or inaccessible. Even with flat wireless tariffs, providers are increasingly implementing data caps for their wireless plans, forcing restraints on heavy data users.[1]

Complementary *ad-hoc* approaches to content sharing can offer attractive alternatives to cloud services and can work *together* with infrastructure or *independent* of it, allowing users to share content without having to worry about overrunning their monthly data caps. Additionally, such sharing may open up new possibilities for social networking, file-sharing, games, and other applications that take advantage of user proximity and social interactions. In situations where infrastructure is out of reach, or where services are costly, slow, or inaccessible—as in many developing countries or disaster areas—ad-hoc content sharing can enable communication where it otherwise would not be possible. As recent events have shown, people under the rule of dictatorial regimes cannot always rely on infrastructure to get their "tweets" and status updates out to the world.[2]

However, despite this potential, ad-hoc content sharing faces challenges in delivering the most *relevant* content to users, and doing so efficiently in a way that minimizes strain on resources, such as battery and storage. Unfortunately, existing content sharing systems [17, 11, 16] offer coarse definitions of relevance and view every delivered content item as contributing the same "value" to the network, expending resources on delivering sometimes low-relevance content. These systems do not reflect the reality that all content items are *not* created equal and instead have *varying* relevance to users.

In this paper we present Haggle, a system for ad-hoc content sharing, focusing on how search can be leveraged for *relevance-aware* dissemination. Unlike previous mobile content sharing systems, Haggle prioritizes content based on how relevant it is to users, and allows highly desirable content to propagate faster than less desirable content—even when the group of interested users is small (*i.e.*, the content is not so popular, but those that desire it have strong interests). While other systems optimize for high delivery ratios and low average delay for *all* content—irrespective of its relevance to users—Haggle can discourage the dissemination of low-relevance content in order to save battery, storage, bandwidth, and to reduce the spreading of "junk".

Several applications have been built on top of Haggle: PhotoShare shares pictures taken with a mobile phone's camera, MailProxy allows emailing without infrastructure, MobiClique [23] and Opportunistic Twit-

[1]For instance, AT&T caps data at 2 gigabytes per month.

[2]On Jan. 27, 2011, the Egyptian regime shut down all Internet traffic in the country in a response to a public uprising.

ter [25] enable ad-hoc "facebooking" and "tweeting", while Haggle-ETT [20] provides an electronic triage tag for disaster areas. Haggle allows these applications to transparently share and disseminate content over ad-hoc networks (using, *e.g.*, WiFi or Bluetooth), or over existing local area networks (LANs)—without requiring intermediary service providers. Devices in contact exchange interests and then *push* matching content items in order of rank, based on a local search. Received content items are stored locally until another dissemination opportunity occurs. Thus, content items propagate based on their relative ranks, without the need for endhost names or addresses.

Push-based search dissemination—as a primary mechanism for content sharing—is one of the main contributions of Haggle when compared to related systems, such as DTN [10], PodNet [17] and traditional pub/-sub systems [9]. Another contribution is a new way to use forwarding algorithms to compute *content delegates*, *i.e.*, nodes that carry content although they may have no interest in it. Comparable content-centric systems that we know of only disseminate among interested nodes. The lessons learned from designing Haggle's architecture, and then implementing it, comprise our third contribution; figuring out the right abstractions and primitives and how they fit together into a system is a significant research undertaking. Our architecture and prototype represent several years of research and experimentation that provide valuable insights into the most important aspects of building a real system. Finally, our evaluation of Haggle is first to explore how content relevance and delegation can achieve a more efficient dissemination in real-world settings.

In the rest of the paper, we begin with a discussion of the challenges and concepts underlying Haggle's design in §2. We then address related work in §3, before describing the Haggle design in §4, focusing on content search and dissemination, how to avoid unnecessary transmissions, and how to deal with security and resources. §5 then continues with describing our implementation, while §6 presents our evaluation. In §7, we round off the paper with our conclusions.

2. CHALLENGES AND DESIGN CONCEPTS

There are significant challenges to overcome in order to build a system for ad-hoc content sharing; here we discuss the most important ones.

First, *naming and addressing* determine the content desires in the network, *i.e.*, how content items map to nodes. A challenge is to design naming and addressing such that users have the ability to express their content desires in flexible ways, allowing them to distinguish highly relevant content from less relevant. Traditional host-centric naming schemes fit poorly with applications that access and share content [13]. For example, host-centric naming has no concept of relevance and may lead to considerable overhead because multiple copies of the same content are labeled differently, leading to duplicates in the network. Name or address labeling also bind content to specific locations early in the communication process, making it difficult to support mobility and long periods of disconnection. Further, in an opportunistic network, the location and replication of content may change between the initiation of a content request and the resolution of it.

Second, a major challenge with ad-hoc communication is to detect and fully *exploit communication opportunities* when they arise. Devices must sense each other, and make efficient use of short-lived contacts by selecting the best means to communicate (*e.g.*, WiFi or Bluetooth) and prioritizing the most relevant content.

Third, mobile devices have limited resources in terms of storage, battery and bandwidth. A content sharing system must have mechanisms for *resource management* that judiciously decide which content to replicate, keep, or drop, in order to optimize the perceived "value" gained from the sharing system.

Fourth, depending on the structure of the network and interests of users, content may not reach the nodes that desire it without leveraging "data mules" that carry content although they have no interest in it. Such *content delegation* might be crucial for disseminating content to groups of interested nodes that are partitioned.

Finally, *privacy and security* are crucial to any architecture. With ad-hoc content sharing, new strategies for securing content rather than hosts are needed. Nodes rely on each other for dissemination, and must therefore ensure that content can be shared securely without risk of flooding nodes with "junk" or having malicious nodes disrupt the network. Schemes for learning of trusted content sources need to be devised, as well as mechanisms to set up trusted and secure groups based on social interaction and physical proximity.

2.1 Design Concepts

To meet the above challenges, Haggle rests on five main architectural design concepts:

Search resolution. Haggle foregoes traditional naming and addressing and instead uses search to resolve a content item's *interest group* by matching the interests of users against the item's metadata (or vice versa). An interest group is thus defined for each content item, and its members constitute the nodes that have (some) interest in the content. Because the interest group of a content item may change during a content's lifetime, the binding between the content and its interest group cannot only happen at the source—it must continuously be refined as the content propagates in the network.

Content prioritization. Search resolution ranks content and allows dissemination in *order of rank*, en-

suring the most relevant content is prioritized when nodes are constrained by time-limited contacts, bandwidth and device resources. Search enables more flexible matching than the binary "on/off"-matching of addresses and names; the *relative rankings* of a search change depending on the pool of information searched. In contrast, traditional pub/sub systems use either *exact matching* topic channels or filters [9] to resolve their content and subscribers without ranking.

Content replication. Unlike related systems [17, 11], Haggle disseminates a node's interests as a persistent search query (§4.2) that can propagate to nodes never encountered directly (§4.4). This persistent search allows content to be *pushed* back to the interested nodes, leveraging *content delegates* that carry content although they have not interest in it. Haggle does not mandate a specific algorithm for content delegation, instead it uses existing algorithms from the literature (§3).

Resource management. With search, Haggle can adapt to current resources by limiting the scope of search and dissemination when resources are scarce. Content can age and eventually be deleted based on its relevance and/or its degree of replication in the network.

Privacy and security. Haggle provides built-in mechanisms to sign and verify content that allow nodes to reject items they do not trust. When privacy is a concern, encryption if preferred over avoiding giving content to untrusted nodes, as they may still be good content delegates. With secure content sharing, Haggle can provide an attractive local-storage alternative to cloud services when privacy and legal issues (*i.e.*, content ownership) are of concern.

3. RELATED WORK

We discuss related works in terms of how they approach four of the main design challenges discussed in the previous section. These challenges relate to decisions made during time-limited node contacts and determine: **(i)** which content items are desired by the other node, **(ii)** the order in which items are transferred, **(iii)** which *undesired* items to replicate/delegate (in order to increase the likelihood of reaching other interested nodes), and **(iv)** which items to drop (due to finite storage). Previous works address mainly (i), (iii) and (iv), while (to our knowledge) Haggle is first to offer a scheme for (ii). This scheme is enabled by Haggle's unique approach to (i), using ranked searches that allow for ordered transfers. Algorithms for (iii) and (iv) are complementary to Haggle, and are not the focus of this work, although we have studied (iv) previously [3].

Every content sharing system needs a mechanism for (i). Most host-centric schemes [10] label content with a destination and match these against the identifiers of other nodes. This may lead to content duplicates when many nodes desire the same content (§2). Pub/sub [9] is a more general content-centric matching scheme that traditionally uses either filters or topic channels. Filters provide fine-grain content matching while channels are more coarse; neither approach rank content. A fine-grain matching scheme is more likely to give nodes highly relevant content, while coarse schemes risk delivering also less relevant content.

7DS [22], PodNet [17] and related systems [11, 16] use channels for (i), while [21] is a content distribution protocol adopting the filter approach. Haggle differs from these systems by adopting search instead of channels or filters, and by disseminating interests in the network to enable push-based rather than pull-based dissemination. These two mechanisms make it straightforward to implement prioritized dissemination (ii) and content delegation (iii), which the other systems lack.

A broad body of work on forwarding and replication strategies address (iii), and to some extent (iv). These strategies typically calculate utilities for delegating (undesired) content items to other nodes. For instance, Prophet [18] uses a history of encounters to determine the probability that replication moves an item closer to its destination, while [4, 12, 8] use context and social information. Other schemes [24, 1] take a resource allocation view of (ii) and (iv), determining the optimal items to replicate or drop, while [19, 14, 15, 3] exclusively study buffer management. Haggle can use any of these replication schemes for *content delegation* (§4.4) or dropping strategies for storage management (§4.6). Numerous other works on forwarding and replication exist but, since these are complementary to Haggle, we do not further discuss them here.

Prior work in the EU Haggle project has explored opportunistic networking in both theory and practice. Theoretical aspects of forwarding (iii) have been studied in [7, 4, 12, 8]. In contrast, we describe and evaluate a system for content dissemination based on relevance, incorporating lessons from that earlier work. The initial vision of a Haggle system was presented in [27].[3] Our clean-slate system design takes new approaches to (i), (iii) and (iv), and, unlike the prior work, also addresses (ii). We also propose solutions for duplicate data transmissions and security not addressed earlier, and provide an in-depth evaluation of Haggle.

4. THE HAGGLE DESIGN

Haggle's dissemination relies on users sharing interests. Depending on application, these interests are manually input or can be inferred from the user's content consumption patterns. The content items that users share are indexed into a *data store* based on their metadata; such a data store exists on each device and is updated with content and the interests of other users

[3]We inherited the task of designing Haggle when Intel Research Cambridge suddenly closed in 2006.

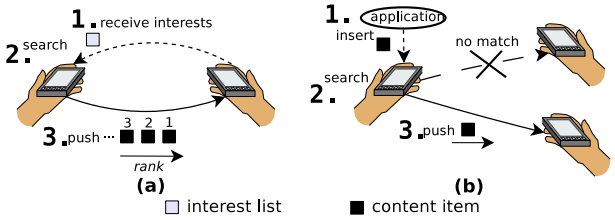

Figure 1: Content sharing in rank order, triggered by receiving new interests (a) or adding new content (b).

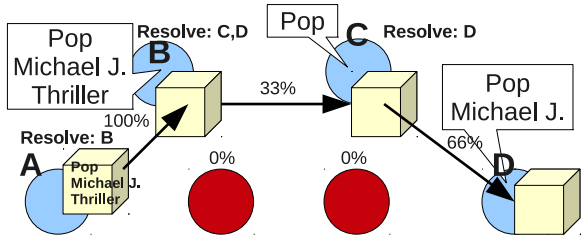

Figure 2: A content item's interest group is search-resolved as the item propagates, excluding nodes with no interest.

as they are encountered.

As shown in Figure 1 (a), two devices in contact share their interests (as detailed later, interests are actually content items that all nodes are interested in, obviating the need for a separate interest-sharing mechanism). The interests query the data store for matching content items, which are then pushed back in order of their search rank down to a configurable limit (compare this to how search engines only return the top 10 hits in a search). Received interests are stored and can further propagate to third-party nodes that repeat this search, pushing back content via *delegate nodes* (§4.4). Similarly, new content items may also be pushed to neighbors when added by applications (Figure 1 (b)).

When transmitted, content items are *not* labeled with their interest group, because the content holder's view of the group may not reflect the current membership. The group may also change as the content propagates because nodes can change interests or "disconnect" for long periods of time. Instead, search resolution is repeated the next time a receiver of the content meets another device, and so forth. This *repeating search* is illustrated in Figure 2. Node **A** initially only knows node **B** as part of the group, but new nodes are resolved as the content propagates. As illustrated, nodes resolve differently depending on their current information. This late (re)binding is crucial for opportunistic networks where nodes have no global knowledge and update their views slowly. Thus, if **A** had labeled the content with only **B**, and the other nodes had disseminated according to this label, the content would never have reached the entire group.

In the rest of this section we take a detailed look at the concepts, ideas and the functionality of Haggle. We begin, however, by describing the data formats that underlie search resolution.

4.1 Metadata-Labeled Data Objects

A content item, once published into Haggle, becomes a *data object*—a unified data format for application layer framing used by both applications and Haggle. A data object, as illustrated in Figure 3, consists of an XML-based metadata header and (optional) content, such as a picture or an mp3-file. The header comprises an arbitrary number of name-value pairs that we call *attributes* (highlighted area in the figure) and an optional number of *extensions*. The attributes typically describe the content of the payload; for instance, if the content is an mp3-file the attributes are taken from the file's ID3 tag (*e.g.*, `Artist="Michael Jackson"`, `Album="Thriller"`, and `Genre="Pop"`). A user may also add its own attributes; a camera picture can be tagged with the location and the people in the picture.

Extensions hold optional metadata useful to Haggle or applications, and are processed by a corresponding *manager*. For example, a `<Data>` extension holds information about an object's payload; its file name, length and checksum, and is processed by a "data manager" that verifies the payload. Applications can similarly add an `<Application name="MyApp">` extension that holds application-specific metadata.

Search resolution only acts on attributes, while extensions can be added and removed without affecting the dissemination of an object. The presence of an extension can, if necessary, be signaled by adding a specific attribute. For example, `MyApp` can make other nodes aware of its own extension by adding its application-specific attribute. Extensions have proved an important part of Haggle's design, as they allow control messages and application information to be disseminated as data. Thus, unlike other systems [17, 11], Haggle needs no additional mechanism, side channel or solicitation protocol to spread such information.

A data object is unique by its objectID, a SHA-1 hash calculated from its attributes and the payload checksum (if any) and an optional creation time. The objectID ensures content is never transferred more than once between the same pair of nodes.

4.2 Interest-Carrying Node Descriptions

A node description, as illustrated in Figure 4, is a special data object that carries, among other information, the interests of a node (in the form of the attributes) and the data objects it has stored (as discussed in §4.5.1). Node descriptions can be distinguished from regular data objects by their `<Node>` extension and `NodeDescription` attribute. A node (or, specifically, its node manager) can hence single out node descriptions to represent peers

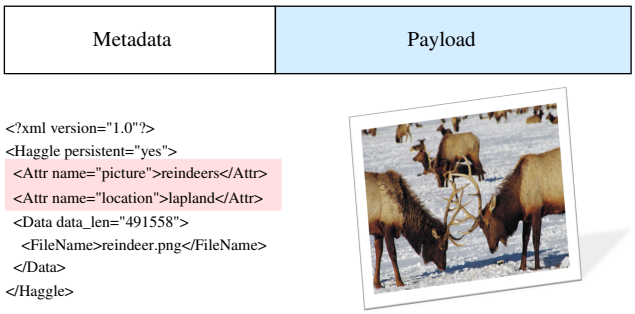

Figure 3: A data object consisting of a metadata header (XML) with two attributes and payload.

```
<?xml version="1.0"?>
<Haggle create_time="1266422654.615827" persistent="yes">
 <Attr name="location">sweden</Attr>
 <Attr name="picture" weight="4">reindeers</Attr>
 <Attr name="NodeDescription">f86edd013a18f2e85602b83c810be9fafad559cd</Attr>
 <Node id="+G7dAToY8uhWArg8gQvp+vrVWc0=" name="Haggle-19" threshold="10" limit="10">
  <Interface identifier="ABfyK8se" type="wifi">
   <Address>ip://192.168.0.19</Address>
   <Address>eth://00:17:F2:2C:CB:1E</Address>
  </Interface>
  <Inventory type="bloomfilter">...</Inventory>
 </Node>
</Haggle>
```

Figure 4: The metadata header of a node description that represents a node rather than a content item.

and their interests. For instance, when receiving the node description in Figure 4, a node learns that the owner of the description has one interest that exactly matches the data object in Figure 3. Interests may be weighted according to how strong they are (defaulting to 1 if missing). In comparison, weights in regular data objects are always 1, because the content creator does not determine the object's relevance to other nodes. Thus, in our example, the node description owner cares four times as much about pictures of reindeers as of their location.

Keeping node descriptions up-to-date. A node has a unique nodeID which others use to recognize it from previous meetings and to keep that node's description in their data stores up-to-date. The nodeID exists in the `<Node>` extension tag, which also contains information such as interfaces and parameters relevant to search resolution (§4.3). The nodeID is, however, never used for naming and addressing purposes when disseminating content.

Interest dissemination via node descriptions. It is important to realize that a node description spreads over multiple hops as any other data object, *i.e.*, based on how its attributes match the interests of other nodes. The notable exception is when two nodes meet; during contacts, nodes *always* exchange their node descriptions, irrespective of whether they share interests or not (they cannot know until after the exchange). Like regular data objects, a node description (and thus the node's interests) can propagate via third parties to a peer never encountered. This peer can then push content back to the node via the same or other parties (§4.4).

4.3 Search Resolution in a Relation Graph

We now describe how search decides which data objects to disseminate during a node contact.

Conceptually, search resolution is an operation on a *relation graph*, as illustrated in Figures 5 and 7. A relation graph shows how data objects relate to each other in a node's data store based on their mutually shared attributes. When two data objects share at least one attribute they have a *relation*, *i.e.*, they share an edge in the relation graph. Each edge is bidirectional and assigned a rating by applying a *weighting function*. For example, Haggle may use a version of the well known PageRank [6]. In this work, however, we assume that relations simply grow stronger with the number of shared attributes and their weights.

Resolving data objects. To illustrate a typical search resolution, consider the case when a new node description is received, as in Figure 1 (a). The incoming node description is matched against stored data objects in order to find the ones that the peer wishes to receive. The result of this search is indicated by the shaded area in Figure 5, and may include regular data objects and node descriptions. Relation weights are dynamically calculated on the relation edges that the inserted node description creates. These weights give a data object a rating that ranks it relative other objects that share an edge with the node description. The resolution may exclude data objects that rate below a certain threshold, and may also limit how many data objects to match. These parameters are set in node descriptions (Figure 4), and thus a resolution occurs according to the parameters of the incoming node description, rather then those of the resolving node. Nodes can hence signal how much data they are willing to receive, and how "narrow" a search should be.

Resolving an interest group. If a regular data object (*i.e.*, not a node description) is inserted in the graph (for example, when applications add new content) the resolution is conceptually the same as when receiving a node description. However, the weighting function in this case zero-weights the edges to data objects that do not contain the *NodeDescription* attribute (*i.e.*, the inserted data object should match only nodes). Such a resolution is illustrated in Figure 7 and represents the known interest group for the object. This interest group resolution is an optimization to quickly disseminate objects generated during a node contact, as shown in Figure 1 (b). However, inserted objects may be disseminated before already stored objects that are higher ranked (but were not resolved due to a search limit). This might be undesired when prioritization is important, making interest group resolutions optional.

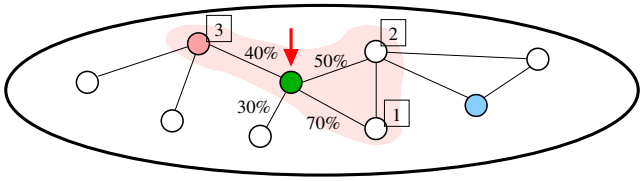

Figure 5: The relation graph of a node's data store. The colored circles are node descriptions (*i.e.*, they represent nodes) and the white circles are regular data objects (*i.e.*, content). This example shows a search that maps a node description (arrow) to stored data objects. The search has a rating threshold $> 30\%$, with resulting ranks shown in boxes.

4.4 Increasing Delivery Probability with Content Delegation

Since node descriptions propagate as data, nodes eventually learn the interests of peers that they may have never met, and may never meet. When a data object is resolved for a remote peer (irrespective of whether it is known from before or not), the data object can (by default) only reach the peer by propagation within its interest group, or by meeting the peer directly. If the interest group is partitioned, small, or consists of only one other member, it may be insufficient to disseminate only within the group to reach all members. In such cases, a *content delegation* algorithm can increase the probability of delivering an item by picking a *delegate* among current neighbors to carry the content. Such delegation is illustrated in Figure 6.

A content delegation algorithm maintains, for each known node pair, a metric of how "good" delegates they are for each other. Haggle does not specify a specific algorithm to use, but existing host-centric forwarding schemes, such as Prophet [18], can easily be adapted.

When enabled, the delegation algorithm determines (according to its metrics) the remote "target" nodes that a neighbor is a good delegate for. Thus, following the normal content search, additional searches are performed for each target, and the resolved data objects are then pushed to the delegate. The number of delegates and targets to compute is configurable. Nodes may express their willingness to act delegates in their node descriptions, along with the parameters involved.

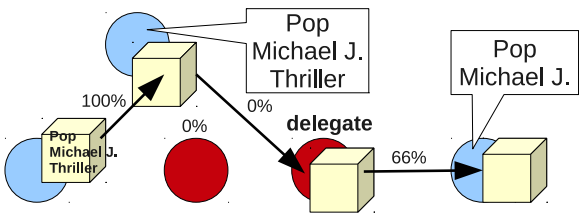

Figure 6: Content delegation where the delegate's interests do not match the content.

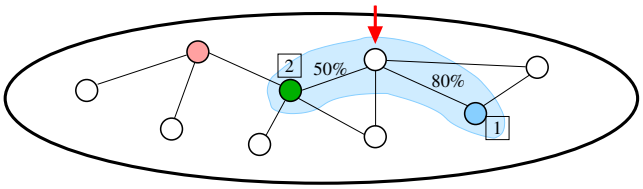

Figure 7: Resolution of the interest group matching a new data object (arrow).

4.5 Event-driven & Push-based Dissemination

Until now, we have only discussed how Haggle figures out *where* to disseminate data (or to which nodes), but not *how* the actual dissemination works. In this section, we describe push-based dissemination and why it provides an advantage over pull-based approaches.

The main advantage of push-based dissemination is that it temporally decouples the act of sending from the act of receiving; the source of a dissemination can push data towards targets without having ever met them, and without having an end-to-end path. Pushing makes sense as interests and stored content may continuously change and co-located neighbors should not have to poll each other for new content (as pulling would require). Further, when nodes want to delegate data objects, only the source knows which objects to delegate. To achieve similar functionality, a pull-based scheme, as prevalent in related systems [17, 11], would require a node to first download lists of content (or channels) in order to determine what content to pull for itself and what to act as a delegate for.

Node descriptions that contain interests are key to enabling data pushing and delegation; they represent the (potentially distant) targets of a dissemination. Pull-based systems typically do not share interests and therefore have no understanding of what other nodes are interested in. A node in such systems simply pulls the content of interest when in direct contact with a peer, and then forgets that peer when the contact ends.

4.5.1 Handling Content Duplicates and Interruptions

With push-based dissemination, receivers do not themselves select the data objects to transfer, which otherwise would make it easy to ensure that content is not received multiple times. Thus, there is a risk that several nodes try to push the same content to the same target nodes.

To avoid unnecessary pushing, a node encodes the data objects it has stored in a data structure called the *inventory*. The inventory could be implemented either as a list of data objectIDs or as a more compact bloomfilter[4], for example. A node shares its inventory in its node description, allowing pushing nodes to vet

[4]A bloomfilter is a fixed-size probabilistic data structure used to test whether an element is a member of a set.

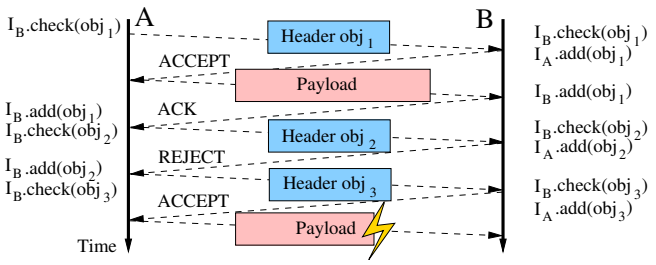

Figure 8: Data object transactions with a rejection, failure and related inventory operations.

objects against the inventory, significantly reducing the amount of unnecessary transmissions. [5] A drawback of the inventory, however, is the potential for collisions, either in objectIDs or in a compact data structure like a bloomfilter. However, using a cryptographic hash function makes collisions in objectIDs highly unlikely, and bloomfilters can also be tuned to minimize the likelihood of collisions. Optionally, node's may also choose to bypass inventories altogether, or only use them when delegating content.

In addition to the inventory, a transaction protocol aims to capture duplicate pushing when inventories are out-of-date. For instance, the target of a data object push may have received the object from a third party after it shared its node description. In such a case, the data objects not captured by inventories can be *rejected* by the receiver before the payload is sent. Rejections are common when a node meets several peers simultaneously and all try to push it the same content.

When a node has successfully sent a data object to another node, it adds the data object to its copy of the receiver's inventory to avoid sending it again. However, before adding the object, the sending node must know that the peer successfully received it. If a transfer fails, the sender does not know if the data object last written to the socket (in its entirety) has left the socket's send buffer. Therefore, data objects need to be acknowledged when fully received.

Figure 8 illustrates the transaction protocol used when a node tries to push three data objects to a peer. Node **A** starts by checking its copy of **B**'s inventory (which it received at the beginning of their encounter) and sends the header of the first data object. When received, node **B** checks its inventory for a previous reception of the object. **B** does not have the object and sends back an `ACCEPT` message, which causes node **A** to proceed with sending the payload. **B** also updates its copy of **A**'s inventory because the data object may have been generated by **A** (or received from a third party) after the initial node description exchange (thus not existing in **B**'s copy of **A**'s inventory). When also the payload has been received, **B** adds the data object to its own inventory and acknowledges the object with an `ACK` telling **A** to add the object to its copy of **B**'s inventory. The next data object is rejected after the header using a `REJECT` message, because **B** already had it in its inventory. **B**, however, still updates its copy of **A**'s inventory since it now knows that **A** has the object. **A** similarly adds the rejected object to its copy of **B**'s inventory, as **B** already has the object (or does not want it). The third data object is accepted, but the stream is interrupted during the payload transfer. Due to the missing `ACK`, **A** knows that the interruption occurred during transmission of the third data object, and therefore does not add it to its copy of **B**'s inventory. Node **A** may try to resend the failed data object at the next opportunity.

A further use of the `ACCEPT` message is to act similar to a `HTTP` range request. The accepting node attaches the byte offset of a partially received data object to the `ACCEPT` and the payload transfer starts from where the previous one failed. This benefits large data objects as they are more likely to fail during time-limited encounters. With byte offsets, such objects can be sent over several encounters, possibly retrieved chunk-wise from different nodes.

4.6 Content and Resource Management

As nodes exchange content, data stores grow, searches slow down, and storage may become exhausted. Fortunately, search resolution provides novel mechanisms to manage content and resources.

First, limits and thresholds in node descriptions allow a node to express its willingness to receive content. The willingness can adapt to current resources, such as storage and battery, and a node may also reject incoming data objects when resources are scarce. Second, data object rankings can be used to devise sophisticated strategies to age content (*e.g.*, low-ranked content ages faster than high-ranked).

By default, Haggle ages only data objects in which it has no interest. Such data objects include delegated objects, node descriptions from peers it shares no interests with, or objects it has simply lost interest in. The aging of node descriptions is important because a node should not carry outdated node information. Carrying many (and sometimes outdated) node descriptions slows down search and may cause unnecessary dissemination. Therefore, a node may also age node descriptions it shares interests with. In previous work [3], we have explored different aging strategies and studied their trade-offs. The results from that study indicate that objects should age based on how many times they have been replicated.

[5] In previous work [2], we have explored the trade-off of disseminating node descriptions often (to push up-to-date inventories) versus to cost of doing so, and found that (in common cases) it is worthwhile to be reasonably aggressive.

4.7 Content-Centric Security

Haggle provides a security framework to manage cryptographic keys and automatically sign and verify data objects. This functionality is based on standard public-key cryptography, which is already extensively used on smartphones. Newly created data objects are signed by Haggle and receivers choose to trust them or not depending on embedded signatures. It is not mandatory to sign data objects, but nodes can reject unverifiable ones to thwart the attempts of malicious nodes to, *e.g.*, pollute the network with "junk". Several nodes can endorse content by attaching their own signatures to existing data objects, hence creating signature lists [28]. Such lists can be used to elevate data object rankings based on the endorsements from trusted nodes (*e.g.*, by devising a weighting function that accounts for the number of trusted signatures).

To verify data objects, nodes must acquire and verify the authenticity of the public keys of trusted peers. Haggle supports several ways to do this by embedding keys and/or certificates in the `<Security>` extension of the node description. For instance, if a device comes pre-installed with a public key signed by a global certificate authority (CA), nodes may provide their own keys in certificates. Alternatively, the authenticity of a key can be verified through human interaction when people meet. For instance, a user can display its QR-code encoded public key (or the key's hash in the form of a self-certifying nodeID) on its mobile phone's screen. The other user can then take a picture of the QR code using its own phone's camera and automatically decode it into the public key (or, in case it is a self-certifying nodeID, it is used to verify the full key found in the node description labeled with the nodeID). Nodes can also establish communities and reputation systems using PGP-style *webs-of-trust*. Thus, new certificates (and public keys) can be accepted via already trusted third parties. Naturally, any other out-of-band mechanisms to securely acquire public keys can be used as well.

Applications encrypt their sensitive payloads using *symmetric key encryption* for efficiency. The symmetric key is public-key encrypted and embedded within the `<Security>` extension of a data object. The key can be embedded multiple times encrypted with different public keys, and only the receivers holding the matching private key(s) can decrypt the symmetric key and hence the payload. Thus, encryption works similar to, for example, email systems.

Because nodes share interests, they could potentially reveal private information through them. To alleviate this concern, secret matching, in which encrypted interests are matched without revealing their clear-text information, can be used [26]. However, the details of such secret matching we leave to future work.

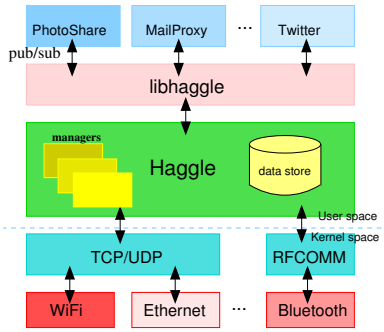

Figure 9: Implementation overview.

5. IMPLEMENTATION

We have implemented Haggle for smartphone platforms such as Android, iOS and Windows Mobile, but it runs also on Windows, Linux or Mac OS X computers. The implementation comprises around 20,000 lines of C and C++ code, excluding applications. The initial choice to develop the implementation in a native language has shown crucial for cross-platform development and for access to lower layer APIs. Higher-level languages require virtual machines and libraries that are incompatible between platforms and they seldom expose complete APIs for, *e.g.*, Bluetooth and WiFi, which are required for an event-driven design.

Figure 9 gives a logical overview of our implementation. Applications link against a library, called *libhaggle*, which implements an asynchronous pub/sub API. This library communicates with Haggle using local IPC. The API makes application development straightforward and has, despite its simplicity, enabled a wide variety of applications, as described in the introduction.

Haggle itself runs as a user space process with a main thread in which a kernel and a set of managers run. The kernel implements a central event queue, while managers divide responsibility in areas such as security, node management, content dispatching and integrity. Managers create and consume events and may run tasks in separate threads when they need to do work that requires extended processing. This may include sending and receiving data objects, computing checksums, doing neighbor discovery, and so forth. Due to the modularity of our design, managers and task modules can be added with little effort, which makes it easy to extend Haggle with extra functionality.

The data store is based on an SQLite backend and runs in a separate thread due to disk operations that take a relatively long time to complete. The data store provides an internal interface to managers that allow them to query the data store, perform searches, and add, remove, and retrieve data objects.

The implementation can leverage Bluetooth, Ethernet, and WiFi for communication. The implementation can also support multiple pluggable content dele-

gation algorithms, but we have so far only implemented Prophet [18] and epidemic-style protocols.

6. EVALUATION

In this section we evaluate Haggle's relevance-aware dissemination through a set of experiments running on mobile phones and on an emulation testbed. We focus on evaluating the effect of content prioritization, *i.e.*, do users receive content in order of relevance? The evaluation also aims to validate our system design and implementation, looking at overhead and battery cost.

6.1 Experimental Setup and Methodology

Studies have shown that simulations do not accurately capture the performance of opportunistic networks [25]. Our evaluation therefore focuses on real-world experiments with a complete Haggle implementation, running a real application that mimics photo sharing on mobile phones. Each phone carries a set of photos tagged with metadata, which are disseminated to users based on their interests. Running a real system and application accurately captures system interactions and environment characteristics, but also imposes limitations on the experiment's scale and repeatability. Fortunately, the community structure of opportunistic networks—where nodes move between "islands" of connectivity and social interaction (*e.g.*, work and friend communities)—allows a small-scale experiment to accurately capture the interactions within a single community. Thus, such an experiment can give us valuable insight into Haggle's performance. We believe intra-community behavior is also indicative of the functionality across communities—although the exact characteristics of inter-community dissemination we leave to future work. By complementing our real-world experiment with trace-based emulations, we can also accurately repeat experiments in a controlled environment, trading some realism for repeatability.

Experiment scenarios. First, we report on a real-world experiment involving 10 mobile phones that we handed out to colleagues in our office corridors. The phones ran Haggle and were pre-seeded with content and interests (as we describe below). The experiment lasted around six hours and people attended their normal work schedules; they sat in their offices, chatted with colleagues, had lunch, and participated in meetings. Second, we complement the mobile experiments with virtual machine (VM) emulations, where traces from the mobile phones decide the connectivity between VM-nodes. In the emulation testbed, we can repeat experiments to study the effect of delegation and compare to other systems—something which would be impractical in the real world. Although limited in scale, the phone experiment captures the type of content sharing opportunities a user might experience within a single community, such as work.

Studying content categories in terms of relevance and interest group size. To study relevance-aware dissemination, we look at how characteristic categories of data objects propagate in a Haggle network. We postulate that data objects whose interest group members all have a *strong* interest in their object should propagate faster than those objects whose members all have a *weak* interests. In other words, two characteristic categories can be defined by members that either give a data object a very *high rating* or give it a very *low rating*. Haggle's design should favor highly desired data objects over less desired ones. However, the size of an interest group also matters, because it is indicative of the object's popularity (although a popular object may not necessarily be strongly desired). We therefore further categorized high-relevance objects and low-relevance objects by their interest group sizes, according to small size (3-5 members) and large size (8-10 members); we thus obtained four main categories of data objects that we study. We now describe how we generated data and interests to match these categories.

Interest and (meta)data generation. Internet content popularity has been shown to be Zipf-like [5], and we therefore generate interests and data for our content categories according to a Zipf distribution. To this end, we first created a set A consisting of 100 attributes, and the probability to pick an attribute a_i from this set was configured according to a Zipf distribution with $p_i = i^{-s} / \sum_{n=1}^{100} n^{-s}$ and skewness $s = 0.5$, which ensured a good variance in interest group sizes. Every data object was assigned 10 attributes $a_i \in A$. Similarly, every node was assigned 10 random interests $a_i \in A$ with a weight $w_i = \omega \cdot p_i$, where ω was chosen such that the sum of the 10 weights was 100. Thus, any node that matches all attributes of a data object gives it a 100 point rating, and those with a non-zero rating (by definition) belong to the object's interest group.

To ensure we only have data objects that match our four categories, we first generated a large pool of 100,000 data objects using the Zipf distribution, and for each category we then picked the 200 objects from the pool that best matched the category. For example, for the category with high ratings and small groups we picked objects with 3-5 interest group members and with the *highest lowest* ratings within their groups. For the category with large groups and low ratings we picked the 200 objects with *lowest highest* rating within groups of 8-10 members. The two other categories were picked analogously. For comparison purposes, we also picked 200 objects at random to form a fifth *random* category with any group sizes and interest strengths.

The categorization of data objects allows us to compare how objects in different parts of the distribution are disseminated. Figure 10 shows how the 200 data

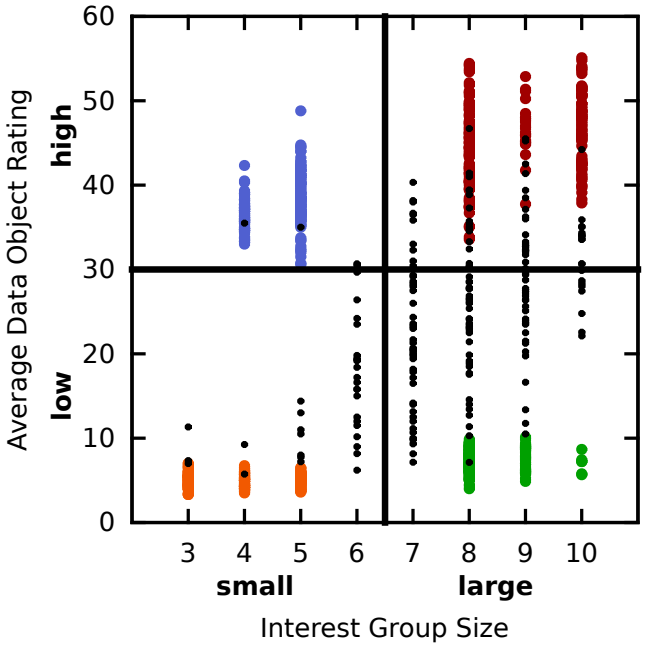

Figure 10: The data objects in our experiment shown by their interest group size and average rating. There are 200 objects of each category.

objects from each category are distributed according to interest group size and ratings. Each object has a different rating for each member of its interest group and we therefore show the average rating of the group in the figure. As shown, each of our four main categories, small/low, small/high, large/low and large/high fits comfortably within their respective quadrant of the graph. The random category follows the Zipf distribution; there are some attributes that frequently occur in both interests and data objects, causing the skewness towards large group sizes.

Hardware and experimental setup. For the real-world experiment we used 10 Nexus One phones (Android 2.2), which we pre-seeded with 10 interests and 100 data objects (20 objects from each category by uniform distribution). In total, there were 1000 objects in the network and each node had some degree of interest in around 600 objects. Each data object's payload consisted of the same ∼700 kB picture taken with a phone's camera. The phones were configured to use WiFi in adhoc mode, and to send a "hello" beacon every ∼5 s to detect each other. We configured a 2 dBm transmit power in order to reduce the power consumption. Haggle was configured to resolve 10 data objects per encounter and we enabled Prophet [18] delegation with restrictive settings; one delegate and one target could be computed per delegation opportunity. Prophet's parameters were set according to its draft specification. The emulations ran on a Lenovo M58p 3GHz dual core machine with 8GB RAM and an Intel X25-M SSD disk, with Debian 5.0.8 and Xen 3.2.

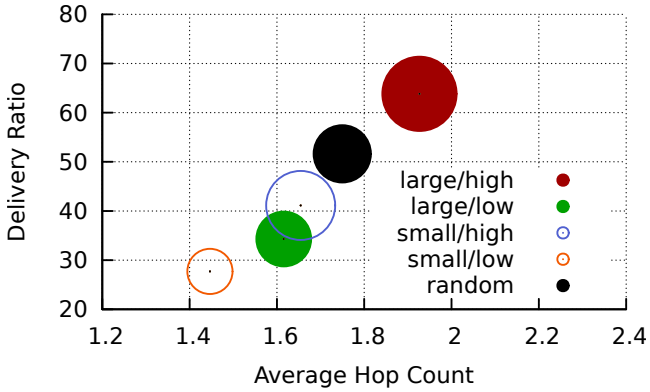

Figure 11: Delivery ratio and hop count of all data object categories. The circle sizes illustrate the relative number of delegations.

6.2 Results

In this section we present the results of our experiments, focusing on the delivery ratio, delay and overhead of our five categories of data objects. The main results are from the phones, and we discuss the emulation results later when we specifically look at delegation, content filtering and compare to other systems.

Relevance-aware data delivery. We calculate the delivery ratio for each object as the fraction of interest group members reached. The network total delivery ratio is the average of each object's ratio. With this metric, the total delivery ratio for the mobile phone experiment was ∼43%. This may at first seem low, but it only reflects the delivery the phones managed before running out of battery. The capabilities of the phones (in terms of I/O and CPU) also limit the dissemination, as we show later when looking at testbed results.

Figure 11 breaks down the total delivery ratio based on our data object categories and correlates it with the average hop count. This breakdown gives a better indication of Haggle's ability to prioritize content based on relevance, while also giving a measure of the degree of delegation in each category (indicated by the relative sizes of the circles). We first note that there is a significant difference in delivery ratio and delegation between the two extreme categories; small/low versus large/high. *This clearly illustrates Haggle's ability to distinguish highly relevant content from less relevant content, and to prioritize dissemination accordingly.*

Looking at small/low versus small/high, we see that high-relevance contents seem to be able to overcome the limitations of small groups. Small/high achieves a significantly higher delivery ratio than the comparably sized small/low and even overcomes the larger group sizes of large/low. At least part of this higher ratio can be attributed to delegation, as small/high has

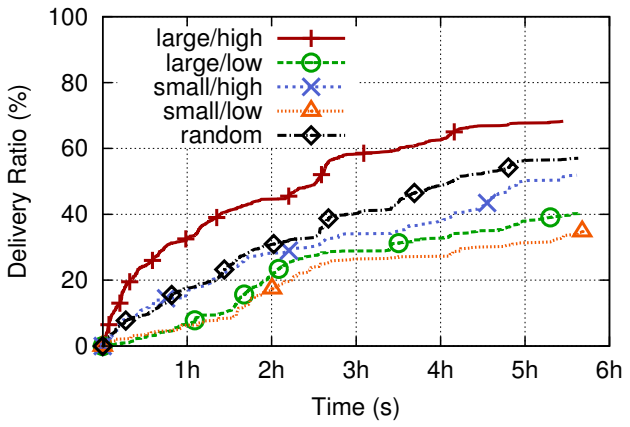

Figure 12: Cumulative delivery ratio (CDR) for each data object category in the mobile phone experiment, illustrating the prioritization of relevant objects and the role of interest group size.

more delegated objects than the other two groups. This shows that Haggle can prioritize dissemination within small groups that have strong, but narrow interests—something which might be important for niche content that appeals to few but enthusiastic people.

Correlation between relevance and delay. Under ideal conditions without time constraints, all data object categories in our phone experiment should achieve 100% delivery. Thus, the time it takes to deliver data objects of different categories might prove a more interesting metric than delivery ratio by itself. With Haggle, nodes should receive their most relevant content before the less relevant.

Figure 12 shows the cumulative fraction of delivered data objects over time (since data objects were preseeded, time also equals delay). Each data point represents 100 delivered objects (we count each member as one delivery and, therefore, larger groups have more data points). We clearly see the priority of high-relevance objects in the beginning, in particular the ones in the large/high category. *This shows that Haggle, on average, delivers high-relevance objects faster than low-relevance ones, even if their interest groups are small.* This raises the question whether the random category—which has both high- and low-relevance objects—also has its high-relevance objects delivered before the low-relevance ones (for those having the same size group)? Although not shown here in a graph, our analysis confirms that this is indeed the general trend, as expected.

An interesting event occurs at around two hours; people from one research group attended a meeting and therefore had less contact with the other people. At this time, the rate of delivered objects in the two high-rating categories flattens while the two low-relevance categories see increased dissemination. The people in the meeting thus ran out of high-relevance objects to share between this subset of nodes and therefore started

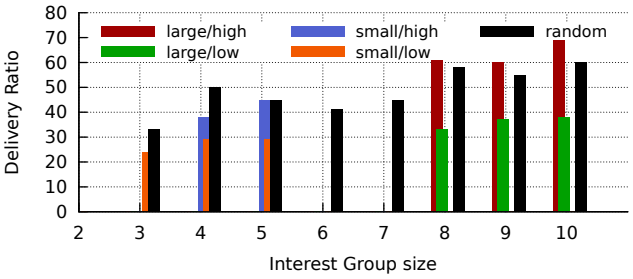

Figure 13: The effect of interest group size on delivery ratio in all data object categories.

to share also low-relevance ones. *Thus, times of network segmentation may offer opportunities for low-relevance content items to increase their share of the dissemination, as high-relevance items reach saturation.* When the meeting finishes (at around 2.5 hours), the large/high category resumes a higher rate of dissemination as the interaction with other nodes increases.

Effect of interest group size (popularity). Data objects that both have a high rating and a large interest group see the best delivery ratio, which is consistent with Haggle's design. The random category fares comparably well, which is reasonable considering it represents the performance of an average object. Since the Zipf distribution is skewed toward large groups (Figure 10), the average object is quite popular, contributing to more dissemination. Small/high, however, sees a proportionally larger fraction of delegation than random—despite smaller interest groups—due to the tendency of high-relevance objects to be delegated. Random still has a higher hop count due to its larger groups. Comparing large/high and small/high further emphasizes the effect of group size, as these two categories both have interest groups with roughly equally strong interests. In a small group, an unreached member will have a more significant impact on delivery ratio than in a large group. Further, small groups are more sparse and are therefore likely to suffer from partitioning. At least in this scenario, large groups are more densely connected and therefore also disseminate more.

In Figure 13, we further break down the categories according to interest group size. The questions we seek answer to when doing this breakdown is if "bigger is better", or whether there is one optimal interest group size, or if there is one size that is large enough, and if some categories benefit more than others? *The results point to "bigger is better", as all four main categories (in most cases) benefit from increased group size.* This trend is not as clear for the random category, which might be explained by the small number of objects in that category having small interest groups. Thus, the delivery ratios for such objects are not statistically reliable. Only at six members and beyond do we see a large number of random objects (Figure 10), and in this sub-

	#sent	#reject	#desired	Metadata size (bytes)
Data	5818	1981	2860	836
ND	2286	190	-	5139
Routing	1347	-	-	1165

Table 1: The number of transmissions of different data object types, along with average header size. (ND = Node Description.)

segment "bigger is better" generally holds. There does not seem to be any categories that benefit significantly more than others from large interest groups.

The cost of dissemination. There are four main types of overhead in Haggle: the metadata header of relevant (application-delivered) data objects, node descriptions, duplicate/rejected transmissions and routing control information. Table 1 summarizes these as the number of transmissions per type. All in all, when looking at the total number of bytes, the overhead per useful byte delivered is 35%. This high figure is due to delegation; nodes carry content which is not of interest to them, thus inflating the overhead figures. If we remove this undesired content from the calculations, we end up with a overhead of 1%. Node descriptions are disseminated to make sure peers have updated inventories, and this comes at the cost of more overhead. On the other hand, having more up-to-date inventories reduces the number of misdirected delegations (as mentioned in Section 4.5.1, this seems to be a worthwhile tradeoff). *Data object rejects occur as a result of duplicate transmissions (often when many nodes are co-located), and the high number of rejects (although overhead itself) indicates the importance of this mechanism.* Without a reject mechanism, we would also have to add payloads to the overhead and that would be significant. To reduce rejections, Haggle's transaction protocol could be extended with a "request-to-push" message that includes only a data object's ID, allowing a receiver to reject a data object before the header is received. Another option is to compress the headers, which would drastically reduce byte overhead.

Prophet accounts for the routing data objects that are exchanged at every encounter. The benefit gained at this overhead cost is difficult to judge without the ability to exactly repeat the experiment without delegation enabled. This is why we specifically look at delegation in our emulation testbed.

Power consumption. The test application we used on the mobile phones acquires a wake lock that ensures the CPU stays awake during an experiment. Without this lock, Haggle runs only while a phone is actively used—something that makes sense for everyday use, but is not ideal for our experiment. Naturally, the wake lock reduces the battery life, but in return a phone can make use of every dissemination opportunity. Unfortunately, we discovered a problem in Android that occurs after

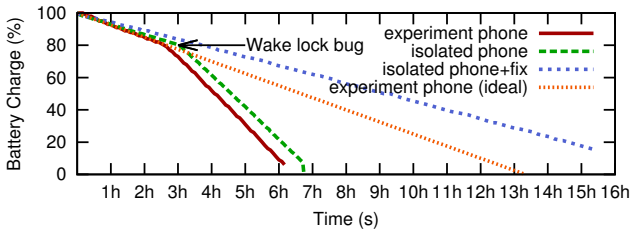

Figure 14: Battery drain of an experiment phone compared to one running in isolation, showing the effect of a wake lock bug in Android.

holding a wake lock for three hours, significantly increasing the battery consumption. Figure 14 shows the battery charge of a typical phone during the experiment (all phones had a battery life within 30 minutes of each other), clearly illustrating the increased battery drain.

To investigate the battery drain caused by node contacts, data processing and exchanges, we ran a phone in isolation with the same configuration as in the experiment. Figure 14 shows that the isolated phone only manages around 30 extra minutes, although it is not exchanging any data. Thus, data processing and exchanges have a relatively small effect on battery life. We then ran the isolated phone with a fix that releases the wake lock for five minutes every 30 minutes. The fix more than doubles the battery life to around 16-17 hours, indicating that the wake lock problem is the main cause of battery drain. If we extrapolate the initial consumption rate of the experiment phone we find that it would likely manage around 13 hours with the fix, while also exchanging data. We did not have the opportunity to re-run the live experiment, but, fortunately, six hours gave us enough data for the purpose of our study. Despite bugs, the power consumption analysis indicates that Haggle's processing cost is relatively small in comparison with WiFi in general.[6] Even when the phone is kept awake it can manage a full day.

Effect of delegation. To study the effect of delegation, we ran experiments on our testbed with delegation both enabled and disabled. The VM-nodes were configured with the same settings, interests and data objects as the mobile phones, and ran our experiment connectivity trace. Not unexpectedly, the delivery ratio in the testbed was 100% with delegation enabled and almost as high when disabled, explained mainly by more powerful computers and faster communication. *This illustrates the hardware limitations of phones, reducing the number of data objects exchanged during node contacts and further emphasizing the importance of the content prioritization.* Although the emulations are not directly comparable to the real-world experiment, the results give us confidence that the lower results of the phone experiment is not due to limitations in Haggle's design.

[6] [11] reports that WiFi gives ∼75% reduction in battery life.

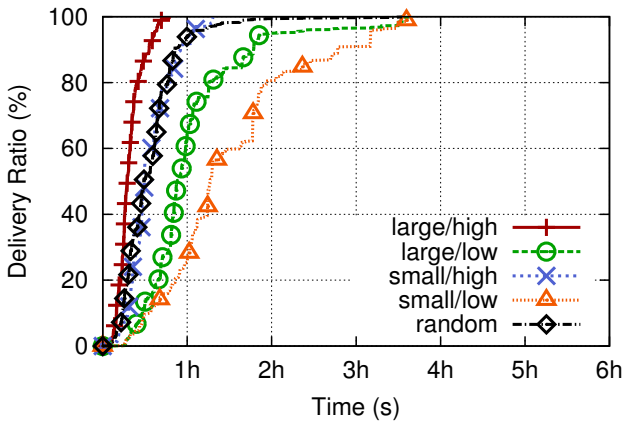

Figure 15: Cumulative delivery ratio (CDR) on the testbed with delegation.

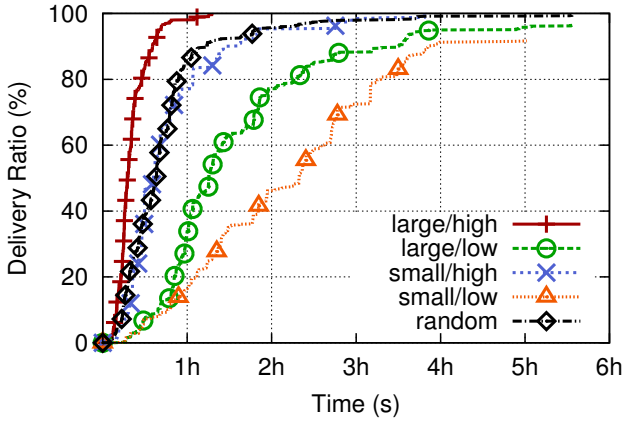

Figure 16: CDR on the testbed with no delegation.

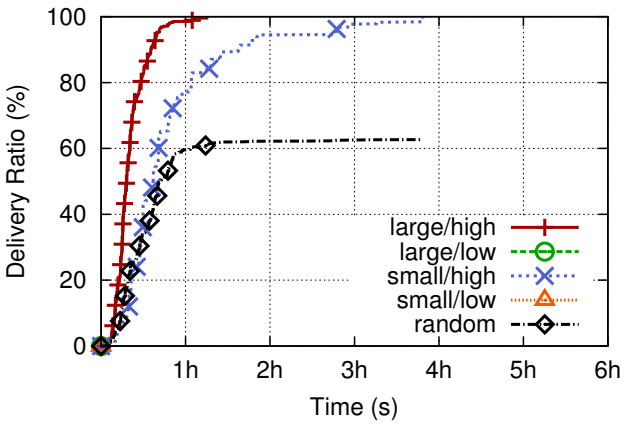

Figure 17: CDR on the testbed with no delegation and filtering of low-relevance content.

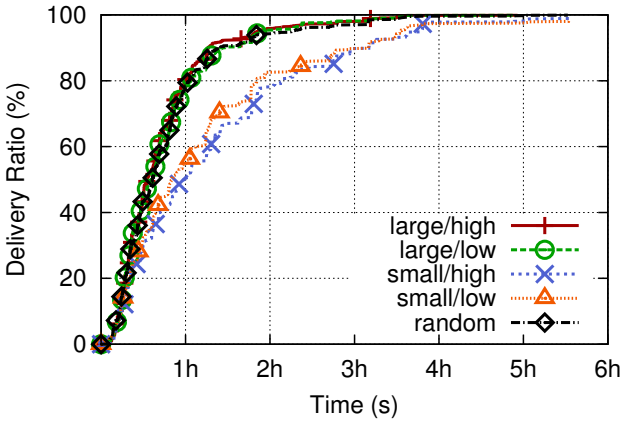

Figure 18: CDR on the testbed with relevance-agnostic dissemination and no delegation.

Optimizations and more powerful mobile phones will likely improve performance in the future.

Figures 15 and 16 show the delay graphs for the emulation experiments with delegation enabled and disabled, respectively. *It is apparent that delegation has a clear effect on delay, especially if we consider the low-relevance categories.* The benefit for low-relevance categories can be explained by a delegation "side effect"; when high-rating categories finish faster, the low-relevance ones account for the remaining disseminations. This side effect will likely be diminished in scenarios where new (high-relevance) contents continuously appear in the network.

As Haggle performs more resolutions per encounter with delegation enabled, we need to verify that the lower delay cannot only be explained by the higher amount of dissemination. Therefore we also examined the average hop count of each category and noticed a higher hop count for the two low-relevance categories, as well as the small/high one, when delegation is enabled compared disabled. The higher hop count indicates that objects took longer but lower delay paths, which would likely not have been available without delegation.

In future work we want to study the effect of delegation with highly partitioned interest groups as we suspect delegation will prove even more beneficial in such scenarios. We conclude, however, that delegation can significantly reduce delay and increase delivery ratio.

Resource savings through prioritization. An important feature of Haggle is the ability to tune search. To save resources, rating thresholds can be set to limit the dissemination to content above a certain relevance threshold. Figure 17 shows testbed results where we used a rating threshold of 20/100, which effectively filters out the low-relevance categories of data objects. The random category achieves only ~60% delivery since its low-relevance objects are not resolved. With this threshold, the total number of data object transfers is only 67% of that with no threshold in Figure 16. Thus, users receive their most relevant content at 2/3 the dissemination cost, saving both battery and storage.

Comparing to other systems. Our first intuition was to evaluate Haggle directly against related systems [10, 17, 11, 16], but it was soon apparent that

this would be an "apples-to-oranges" comparison. Haggle has a fluid definition of interests groups, based on search with thresholds, while other systems use topic channels [17, 11, 16] or destination labels [10]. Having such different definitions of what nodes should receive a specific content item makes designing comparable experiments impractical. However, since other systems differ mainly in that they are *relevance agnostic*, we can make a general comparison to other systems by simply disabling Haggle's relevance-aware dissemination. With relevance-awareness disabled, Haggle resolves interest-matching data objects, but disseminates them in any order. We should expect the average delivery ratio to remain largely unaffected, but there should be no prioritizing of specific data object categories.

Figure 18 shows the cumulative delivery on the testbed with relevance-agnostic dissemination. It is apparent that the only factor distinguishing one category of objects from another is group size. Consistent with our earlier findings, the random category tends toward large interest groups and therefore achieves delivery comparable to the other large groups. Comparing the results to Figure 16 makes the benefit of relevance-aware dissemination clear: *Haggle delivers all large/high objects in one third of the time compared to a relevance-agnostic scheme, while the small/high category increases delivery ratio and lowers delay to levels comparable to objects with average group sizes (random category).*

7. CONCLUSIONS

We have presented Haggle—a system that allows mobile phone users to share content based on its relevance, allowing prioritization of content when node contacts are time limited. Our evaluation shows that search with *relative* rankings provides more flexibility than the simple on/off-matching of comparable systems, while content delegation proves useful when interest groups are partitioned and can be bridged by non-members.

Haggle's source code is available online[7] through an open source license, and provides a readily available platform for researchers to build and study mobile applications for ad-hoc content sharing. A wide variety of applications have already been built on top of Haggle, and we expect more to be implemented in the future.

Many challenges remain, such as how to design efficient aging schemes, leverage infrastructure, and further enhance privacy and security. We also want to investigate inter-community dissemination and study other weighting and ranking schemes for search resolution.

8. REFERENCES

[1] Balasubramanian, A., Levine, B. N., and Venkataramani, A. DTN routing as a resource allocation problem. In *SIGCOMM* (2007).

[2] Bjurefors, F., Gunningberg, P., Nordström, E., and Rohner, C. Interest dissemination in a searchable data-centric network. In *European Wireless* (2010).

[3] Bjurefors, F., Gunningberg, P., Rohner, C., and Tavakoli, S. Congestion avoidance in a data-centric opportunistic network. In *SIGCOMM ICN* (2011).

[4] Boldrini, C., Conti, M., Jacopini, J., and Passarella, A. HiBOp: a history based routing protocol for opportunistic networks. In *WoWMoM* (2007).

[5] Breslau, L., Cao, P., Fan, L., Phillips, G., and Shenker, S. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM* (1999).

[6] 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.

[7] Chaintreau, A., Hui, P., Crowcroft, J., Diot, C., Gass, R., and Scott, J. Impact of human mobility on opportunistic forwarding algorithms. In *INFOCOM* (2006).

[8] Erramilli, V., Chaintreau, A., Crovella, M., and Diot, C. Delegation forwarding. In *MobiHoc* (2008).

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

[10] Fall, K. A delay-tolerant network architecture for challenged internets. In *SIGCOMM* (2003).

[11] Helgason, O. R., Yavuz, E. A., Kouyoumdjieva, S. T., Pajevic, L., and Karlsson, G. A mobile peer-to-peer system for opportunistic content-centric networking. In *MobiHeld* (2010).

[12] Hui, P., Crowcroft, J., and Yoneki, E. Bubble rap: Social based forwarding in delay tolerant networks. In *MobiHoc* (2008).

[13] Jacobson, V., Smetters, D. K., Thornton, J. D., Plass, M. F., Briggs, N. H., and Braynard, R. L. Networking named content. In *CoNEXT* (2009).

[14] Krifa, A., Barakat, C., and Spyropoulos, T. Buffer management policies for delay tolerant networks. In *SECON* (2008).

[15] Krifa, A., Barakat, C., and Spyropoulos, T. An optimal joint scheduling and drop policy for delay tolerant networks. In *WoWMoM* (2008).

[16] Krifa, A., Barakat, C., and Spyropoulos, T. MobiTrade: trading content in disruption tolerant networks. In *CHANTS* (2011).

[17] Lenders, V., Karlsson, G., and May, M. Wireless ad hoc podcasting. In *IEEE SECON* (2007).

[18] Lindgren, A., Doria, A., and Schelén, O. Probabilistic routing in intermittently connected networks. In *SAPIR* (2004).

[19] Lindgren, A., and Phanse, K. S. Evaluation of queueing policies and forwarding strategies for routing in intermittently connected networks. In *Comsware* (2006).

[20] Martín-Campillo, A., Crowcroft, J., Yoneki, E., Martí, R., and Martínes-García, C. Using haggle to create an electronic triage tag. In *MobiOpp* (2010).

[21] Moghadam, A., and Schulzrinne, H. Interest-aware content distribution protocol for mobile disruption-tolerant networks. In *WoWMoM* (2009).

[22] Moghadam, A., Srinivasan, S., and Schulzrinne, H. 7DS - a modular platform to develop mobile disruption-tolerant applications. In *NGMAST* (2008).

[23] Pietiläinen, A.-K., Oliver, E., LeBrun, J., Varghese, G., and Diot, C. MobiClique: Middleware for mobile social networking. In *WOSN* (2009).

[24] Reich, J., and Chaintreau, A. The age of impatience: Optimal replication schemes for opportunistic networks. In *CoNEXT* (2009).

[25] Ristanovic, N., Theodorakopoulos, G., and Boudec, J.-Y. L. Traps and pitfalls of using contact traces in performance studies of opportunistic networks. In *INFOCOM* (2012).

[26] Shikfa, A., Önen, M., and Molva, R. Privacy-preserving content-based publish/subscribe networks. In *IFIP SEC* (2009), pp. 270–282.

[27] 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 *UbiComp* (2007).

[28] Walsh, K., and Sirer, E. G. Experience with a distributed object reputation system for peer-to-peer filesharing. In *NSDI* (2006).

[7] http://haggle.googlecode.com.