Technical Report

UCAM-CL-TR-680
ISSN 1476-2986

Number 680

Haggle: Clean-slate networking for mobile devices

Jing Su, James Scott, Pan Hui, Eben Upton, Meng How Lim, Christophe Diot, Jon Crowcroft, Ashvin Goel, Eyal de Lara

January 2007

ISSN 1476-2986

Haggle: Clean-slate networking for mobile devices

Jing Su[†△], James Scott[†], Pan Hui[†‡⋆], Eben Upton[†], Meng How Lim[†],
Christophe Diot[⋆], Jon Crowcroft[‡], Ashvin Goel[△], Eyal de Lara[△]

[†] Intel Research Cambridge [△]University of Toronto
[‡] Cambridge University [⋆] Thomson

Abstract

Haggle is a layerless networking architecture for mobile devices. It is motivated by the infrastructure dependence of applications such as email and web browsing, even in situations where infrastructure is not necessary to accomplish the end user goal, e.g. when the destination is reachable by ad hoc neighbourhood communication. In this paper we present details of Haggle's architecture, and of the prototype implementation which allows existing email and web applications to become infrastructure-independent, as we show with an experimental evaluation.

1 Introduction

Miniaturization, Moore's Law and convergence have had a profound impact on portable devices, such as smart phones, notebooks, and PDAs. The result is that people are able to carry their previously desktop-based computing environments with them, with the aim of having ubiquitous access to applications such as email and web browsing in an always-on, always-available fashion. However, the low speed, high price and constrained availability of wireless Internet access (through any means, e.g. 802.11, GPRS, etc) means that these devices are often disconnected from the Internet, or have only a slow or expensive connection. These devices use the same, OSI-layered, IP based networking approach as desktop PCs, which assume a fixed network. This fixed network design sometimes perform badly or not at all in the environment that mobile devices find themselves in, which can bear more resemblance to Pocket Switched Networks [6] or Delay Tolerant Networks [5].

Motivating Examples

For instance, let us consider two users, Alice and Bob, on a train going towards London. Alice wishes to send Bob a document that she wants Bob to review for her. In order to accomplish this, Alice might imagine that she can rely on what she would normally use, i.e. sending an email with the attached document. However, in this situation using an email application might (a) not work, failing "silently" as the message waits in the Outbox, or (b) work, but very slowly and expensively, as GPRS is used to transfer data to Bob's email server, and Bob's device has to get it from the server in turn.

As trained experts, we can clearly see that the use of email infrastructure is the wrong way for the message to be sent in this example — it should ideally go directly from Alice's device to Bob's device, using some mutually-supported networking technology such as 802.11. Unfortunately, as untrained users, Alice and Bob may be very confused as to how they can proceed, since they would have to know about their mutual networking technologies, how to configure them, and which applications work over them. In fact, even trained professionals in this situation would likely give up and use a USB memory key to physically transfer the data. (While USB keys may be an acceptable solution sometimes, they are by no means ideal, requiring users to carry around another device, manage the files on it, and they only help when users can easily see/reach one another.)

A second motivating example is also set on a train to London. Charlie wants to find out about restaurants in London using his laptop. He does not have any GPRS connection (and may not be willing to pay for it if he did, or may be out of coverage range, or the train might actually be a plane). Currently, Charlie would not bother even trying to perform this task, as he probably knows that his web browser (which is the obvious application to end users for obtaining information) only works when he has a connection to a wireless access point (AP). In this case, the frustrating thing is that the data is highly likely to be present on many other devices within wireless range of Charlie, since others going to London may well have looked up restaurants before they departed, or on the train (if they did have GPRS access etc). However, with the existing architecture, that information is not available to Charlie.

The Underlying Problem

The root cause of the deficiencies highlighted above lies in the current network architecture for mobile devices (the IP suite of protocols and the Berkeley sockets API), which presents applications with a synchronous, end-to-end connectivity model using numeric addresses for endpoints. In order to satisfy user-level tasks such as messaging and web-browsing, applications are effectively forced by this model to act in ways that include inherent reliance on networking infrastructure, i.e. Internet connectivity.

Due to use of a synchronous model, applications are forced to become aware of the connectivity state of the node and to handle changes in this state, or (typically) simply assume always-on connectivity and avoid solving the problem. By employing end-to-end connections, applications are prevented from making use (without extensive and explicit support) of network routes that may involve non-contemporaneous connectivity. By requiring numeric addresses, user-memorable endpoint identifiers such as user@domain.org and www.server.com must be translated before the interface can be used, forcing a reliance on the presence of DNS.

In reality, while our devices may often have cheap, fast Internet connectivity for some periods (e.g. when we are at home or work), at other times they are disconnected from the Internet, or only connected through an expensive and/or slow network (e.g. GPRS, pay-to-use 802.11 APs). However, while they are disconnected, devices may often be connected to other devices in the neighbourhood, and, as described in the motivating examples above, this limited connectivity may often be enough to provide significant value to users if it could be put to work.

Paper Contributions and Structure

In this paper, we present the Haggle architecture, a ground-up redesign of networking for mobile devices, to support the mobile user scenario. Our prototype implementation of Haggle allows existing email clients and web browsers to seamlessly operate in infrastructure-less network conditions.

The contributions of this paper are as follows. After an overview of the core concepts behind Haggle, (Section 2), we present a detailed description of the Haggle architecture (Section 3). While many of the ideas that are integrated into Haggle are built on existing research[1], a key contribution of this paper is their organization into a coherent architecture. Other contributions include the open source prototype implementation (Section 4), the description of how existing email and web applications can be supported with Haggle (Section 5) and an experimental evaluation of the prototype and applications (Section 6). In building Haggle, we identified a number of research problems that it brings up, both in terms of optimizing the algorithms used and in terms of new avenues of research that Haggle enables or makes easier to pursue (Section 7).

2 Core Concepts of Haggle

In previous work [13] we explored the principles behind the design of Haggle (though we had not built a working prototype at that time). In this section we reiterate the key concepts before diving in to the architectural description that follows.

The key idea behind Haggle is to have a data-centric architecture [2] where applications do not have to concern themselves with the mechanisms of transporting data to the right place, since that is what has made them infrastructure-dependent. By delegating to Haggle the task of propagating data, applications can automatically take advantage of any connection opportunities that arise, both local neighbourhoood opportunities and connectivity with servers on the Internet when available. We identify four design decisions for Haggle that follow on from this.

2.1 Data Persists inside Haggle

The data on each node in Haggle must be visible to and searchable for by other nodes (with appropriate security/access restrictions applied). This facilitates operation of our motivating web example, in that the public webpage needed by one person can be found despite it being in another person's device. In practice, this means that Haggle must manage persistent data storage for applications, instead of applications storing data in a separate file system.

2.2 Networking Protocols inside Haggle

Any application-layer networking protocol includes implied assumptions about the type of network available. For example, client-server protocols such as SMTP, POP and HTTP assume that the Internet-based servers are contact-able. With Haggle, we place networking protocol support inside Haggle itself, allowing us to present a data-centric rather than connection-centric abstraction to applications.

[1]we provide references in the main body of the text rather than in a separate section

2.3 Name Graphs Supporting Late Binding

Since Haggle aims to be infrastructure-independent, it must be able to use protocol-independent names for delivery (since many protocols imply infrastructure of one sort or another). Since we are in an environment where we cannot predict the best path for data a priori, we must perform late binding from protocol-independent names such as a person's name to protocol-specific names such as MAC addresses or email addresses. Haggle therefore maintains its own naming repository (it obviously cannot rely on remote look-up of this data), with mappings from user-level names to protocol-specific names specifying the various ways to get to the user-level name. Furthermore, the whole set of mappings (the "name graph") is transmitted along with the data, allowing even intermediate (i.e. non-source) nodes to bind to protocol-specific names as late as possible.

2.4 Centralized Resource Management

One role of the networking architecture on every device is to decide what to do with each of its network interfaces *now*. In the current architecture, this decision does not take into account resource management — the decision to spend resources on something is taken by applications individually. This makes it very hard for applications to be proactive, since they must make sure themselves that only a suitable level of resources is used. Haggle therefore contains a centralized resource management component, which decides on a cost/benefit comparison basis what tasks it chooses to perform on each network interface at a given moment.

3 Haggle Architecture

The Haggle architecture is shown in Figure 1. Haggle is at a macro-scale comprised of six *Managers*, the Data, Name, Forwarding, Protocol, Connectivity and Resource Managers. In addition, many of the Managers themselves have well-defined abstractions for their contents, as shown in italics/parentheses on the diagram - e.g. the Protocol Manager encapsulates a number of "Protocol" objects.

Haggle is a layerless architecture, in that we do not pass data and control signals up and down between layers as for the current network architecture. Instead, all Managers provide interfaces which other Managers can communicate with. In terms of the current model, Haggle spans the link layer through to the application layer, inclusive. Link layer functionality in Haggle includes, for example, the choice of whether the 802.11 interface is in infrastructure mode or ad hoc mode, while Haggle's Protocols include application-layer protocols such as SMTP and HTTP. Rather than present a "cross-layer design" where layering is deliberately broken, we instead acknowledge that this model is not appropriate for Haggle. The key value of layering, in that between layers there are well-defined abstract interfaces facilitating modularity, is kept: the six Managers provide abstract interfaces and are modular in that they can be replaced independently.

As there is no top layer, the API that Haggle provides to applications is composed of a subset of the APIs that each Haggle manager provides to each other. In this paper we do not list the APIs explicitly due to an excessive level of detail — interested readers are referred to http://cvs.sourceforge.net/projects/haggle/ where these interfaces are available.

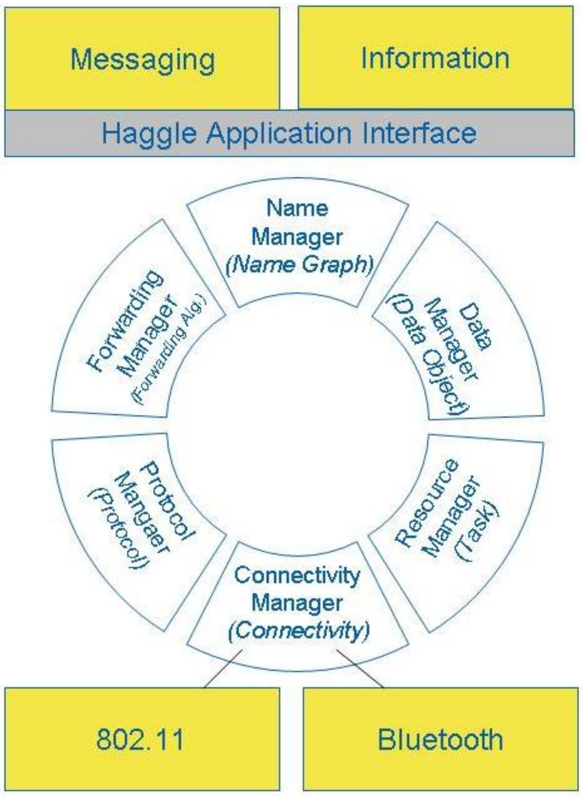
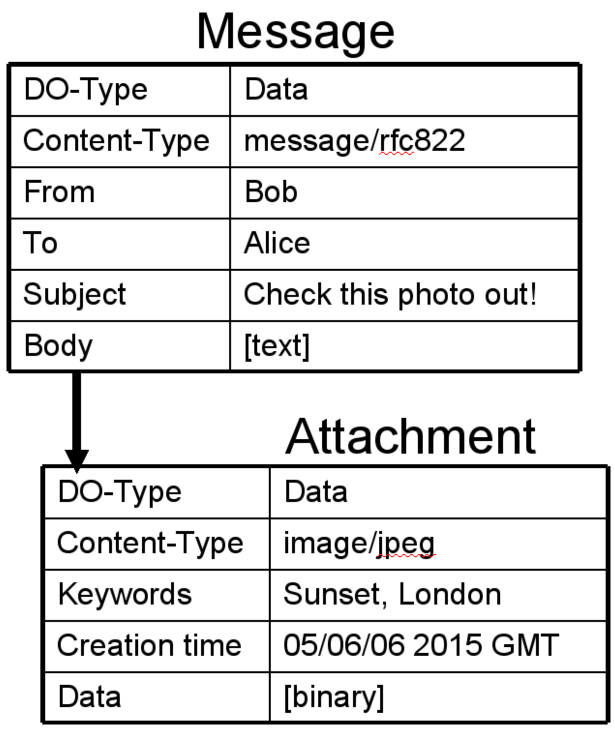

Messaging
Information
Haggle Application Interface
Name Manager (Name Graph)
Data Manager (Data Object)
Resource Manager (Task)
Connectivity Manager (Connectivity)
Protocol Mangaer (Protocol)
Forwarding Manager (Forwarding Alg.)
802.11
Bluetooth

Figure 1: Haggle overview

We now describe the design of each of the managers, including the key data abstractions and components, and how they communicate to perform networking tasks. In this section we restrict ourselves to describing architectural decisions, and do not discuss specific implementations (e.g. the SMTP protocol, or the 802.11 connectivity) — this is left to later in the paper where we discuss the prototype that has been built and evaluated. At the end of this section, we discuss potential security and privacy issues that Haggle raises.

3.1 Data Manager

As stated previously, Haggle maintains users' data persistently rather than relying on a separate file system. Haggle's data format is designed around the need to be *structured* and *searchable*. In other words, relationships between application data units (e.g. a webpage and its embedded images) should be representable in Haggle, and applications should be able to search both locally and remotely for data objects matching particular useful characteristics. We draw inspiration from desktop search products (e.g. Google Desktop) which have changed the way that many users file and access their data [4], allowing us to avoid having to methodically place data in a file/directory structure. We propose that applications can use a combination of structured data and search, with the former providing the kind of capabilities expected of a traditional file/directory system, and the latter allowing applications to easily find and use data that they themselves did not store.



Message

DO-Type	Data
Content-Type	message/rfc822
From	Bob
To	Alice
Subject	Check this photo out!
Body	[text]

Attachment

DO-Type	Data
Content-Type	image/jpeg
Keywords	Sunset, London
Creation time	05/06/06 2015 GMT
Data	[binary]

Figure 2: Example DOs: Message and Attachment

3.1.1 Data Objects

Our data format is simple. A Data Object (DO) comprises many *attributes*, each of which is a pair consisting of a *type* and *value*. Types and values are typically strings, though some values may also be binary packed representations, e.g. the data in an mp3 file. We encourage and expect applications to expose as much *metadata* as possible about an item of application data using attributes, including application data. Two example DOs are shown in Figure 2, representing a message from Bob to Alice, and a photo of sunset. Note that we do not require users to enter more metadata about their objects than applications would require themselves; the value of exposing metadata is in searchability using DO filters described later.

In order to facilitate multi-application environments and to avoid cache consistency issues, DO attributes are immutable after creation. (Haggle itself may mutate attributes for internal record-tracking, but applications may not). Applications must create a new DO instead of modifying a DO, and cause their existing links and claims to point at the new DO. This provides useful guarantees for applications that their data will not be modified “under their feet”, although the disadvantage is that if they wish to use the most up-to-date version of data, they must be proactive, and use the search functionality described later to get updates either proactively or reactively. Another potential disadvantage of copying DOs, that of the time and storage costs of data replication, can be minimised by using standard copy-on-write techniques present in many filesystems.

3.1.2 Links between DOs

DOs can be linked into a directed graph. Links can take two forms. The first is to link data to embedded or prerequisite data, e.g. a photo album’s metadata can link to the set of photos in the album, a webpage can link to its embedded objects, or (as shown in Figure 2) an email can link to its attachments. This provides applications with a way to structure data, akin to the way that some applications use the placement of files in a common directory but more explicit. It allows Haggle to keep track of the prerequisite objects that must be shared alongside a top-level object in order to properly transmit a given application data unit. The second purpose of linking is for applications to themselves link to the DOs which they require for their operation, which can be regarded as an "ownership claim." In this way, many applications can claim the same DO, e.g. a photo gallery application can claim a photo that is linked to by a message (which brought it into the node) which is in turn claimed by the messaging application. Linking and claiming are accomplished using the same mechanism, we use the two terms to differentiate between the parent being another DO or a different entity.

Since Haggle allows many applications to claim DOs, it does not have a "delete" call, instead just an "unlink". When the last link is removed from a DO, it becomes eligible for garbage collection, though this is not necessarily performed immediately since the node may have plenty of persistent storage space. The data remains searchable even in its unclaimed state, which is an advantage since data is not lost unless space is actually required for new data.

3.1.3 DO Filters

In addition to the ability to retrieve DOs via a unique ID provided at creation time, the Data Manager also supports searching of DOs using a "DO filter". This comprises a set of regular-expression-like queries about the attributes of an object, e.g. ”mime-type” EQUALS ”text/html” AND ”keywords” INCLUDES ”news” AND ”timestamp” $\geq$ (now() - 1 hour) would return DOs matching recent news webpages. A filter can be made persistent, and since it is itself stored in a DO, it can be sent remotely. This flexibility allows a single mechanism to be used for multiple purposes: a non-persistent local filter is a search on local data, a persistent local filter is a registration of interest in incoming data of a particular type (which functions analogously to a TCP socket "listen"), a non-persistent remote filter is a request for data which is sent across the network as appropriate (depending on the Forwarding Algorithms, Protocols and Neighbours available), and a persistent remote filter allows "subscriptions" to particular data to be registered with other nodes (e.g. a home PC registers interest in receiving all photos generated on a mobile phone).

The Data Manager is responsible for matching DO filters to DOs, and performs this whenever new data appears (which may match an existing persistent filter) or whenever a new filter appears (which may match existing DOs). If there are matches, then the source of the filter (whether local or remote) is notified. The ability of Haggle to unilaterally, without invoking application processes, answer remote queries is a key feature. It facilitates sharing of information between nodes beyond what we have today, since once the information is provided to Haggle, any and all connection opportunities that the node sees can result in the sharing of that information, given appropriate security concerns such as encryption and access control.

3.2 Name Manager using Name Graphs

Endpoint descriptions for data transmissions in Haggle are not performed in the usual method of the nested headers found on the front of current physical-layer packets (e.g. Ethernet address, IP address, TCP port, and SMTP's RCPT TO field describing the endpoint for an email message). This is because we aim to be able to make use of any available ad-hoc or infrastructure

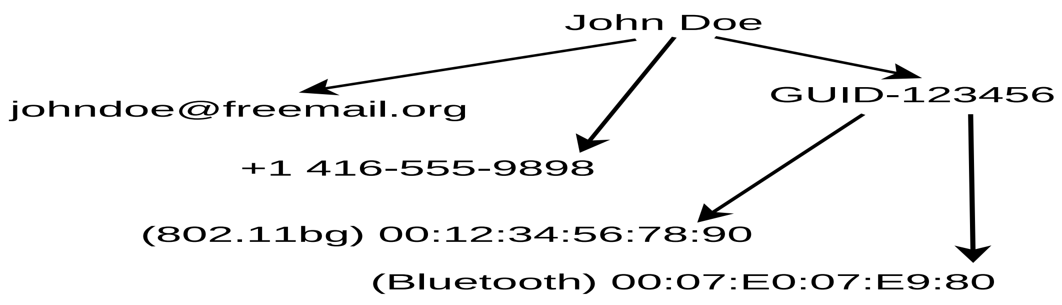


Figure 3: Example of a name graph

connectivity opportunity. Since we cannot assume knowledge of the best end-to-end path, either when a communication is generated or even at an intermediate node once the communication is in-transit, we cannot perform the ahead-of-time directory lookups that are currently used to map a user-level endpoint, e.g. "Bob Smith", to the SMTP's RCPT TO field (b@a.org) and so on, in order to construct those addresses ahead of time. Not only is it the case that some of these lookups require infrastructure services such as DNS that may not be available, but even more importantly it is not possible to perform the initial name-to-email-address mapping that is implicit in the users' choice of an email client rather than an Instant Messaging or mobile phone text message (SMS) client. The choice of client program by the user is currently equivalent to making them choose a networking protocol (e.g. email implies use of SMTP) and may be equivalent to making them choose a device on which the receiver will receive the message (e.g. text message implies use of a particular phone).

We require a more general form of naming notation that allows late-binding of user-level names, independent of the lower-level addressable name, as proposed in i3 [18]. We achieve this by using *name graphs*, inspired by INS [1], which are hierarchical descriptions of all known mappings from a user-level endpoint to lower-level names (which may imply particular protocols/connectivity methods), and by using the whole name graph as the "recipient" for a message, both on the source node and at intermediate nodes for the message. This is one of the "layerless" aspects of Haggle, and it contrasts with the existing Internet architecture where names are only meaningful at particular layers of the protocol stack.

3.2.1 What's in a Name?

An example name graph is shown in Figure 3. These graphs span from top-level nodes such as personal names through to leaves comprising persistent methods of reaching them (e.g. an email address), but not transient addressing data for those methods (e.g. an IP address for the email server, or MAC address for the next-hop). Let us first discuss the choice of this partition [8]

In Haggle, we regard all of the nodes in a name graph as "names", and *any* of these names can be an "address" if there exists a suitable protocol on the node which understands that name. For example, an SMS-capable device regards a phone number "name" as an "address", but a non-SMS-capable device would not. This allows for the fact that, as a message moves between nodes, different methods of mapping names to transmission methods become available. Thus a layered model of name to address mapping is not always appropriate.

We note that while a given node may need to "resolve" a name further in order to effect sending to that name, e.g. taking an email address, discovering the SMTP server suitable, and using IP routing and a next-hop MAC address to send towards that SMTP server, it is not sensible to regard these looked-up values as members of the name graph, since another node with the same message would need completely different values, and would have to resolve them independently.

While persistent information is stored in name graphs, transient information is captured using the notion of a Neighbour, which identifies the next-hop in the path for the data in order to reach the name. Neighbours are discovered by Connectivities (this is discussed in the next section), and their properties are used by Protocols to establish what names a given communication can be forwarded to.

3.2.2 Name Objects

Haggle represents name graphs using DOs with a particular attribute containing the name as a string, which are known as Name Objects (NOs). These are linked using the normal DO linking function to provide name graphs. The use of DOs for naming allows names to be easily managed and made persistent.

Before Haggle can send data to a name graph, the NOs and links must be constructed somehow. There are many potential sources for NOs. Firstly, although the name graph concept seems at odd with existing user devices, actually much of what is proposed is simply a consolidation of naming information from disparate sources already present on a device. For instance, name to email address mappings, name to instant messaging ID mappings, etc are already kept by the respective applications. In addition, names can also be gathered from Connectivities such as Bluetooth or 802.11, as MAC addresses of nearby Haggle-capable nodes are regarded as Names. A third discovery method follows on from this, whereby the Name Manager can detect nearby Haggle nodes (via the existence of the neighbours) and send them directly a message containing both the NO graph corresponding to the node, and a DO filter requesting the recipient's own NO graph (analogous to the the node saying "Hi stranger, I'm Bob, who are you?"). A fourth discovery method is in the receipt of messages where the node is acting as a courier, or is the destination. These messages' NO graphs can be mined for information. A fifth method may be for name graphs to be maintained and distributed during periods of well-connectedness with a trusted server on the Internet.

Finally, it is worth noting that a name graph used as an address for a message need not remain static. Intermediate nodes could potentially add to this graph, either adding hints for forwarding algorithms to perform better routing, e.g. "This MAC address was seen recently", or filling in missing sections of the graph. As an extreme example, a user might be told the name of someone that they wish to send a message to, but not have any other information such as their email address. By simply using the person's name, a message can be created which can only be delivered by (controlled) broadcasting. However, one of the nodes in the room might have the name graph corresponding to that name, and could add the necessary NOs as destinations for the message so that it can be delivered properly.

3.3 Connectivity Manager and Connectivities

Haggle aims to support and embrace the use of many different networking technologies at the same time. Networking technologies differ by their range, latency, bandwidth, infrastructure available, cost of using the infrastructure, battery consumption, availability, and so on. It is therefore appropriate for different Connectivities to be used depending on the particular type of communication being sent, e.g. a small but urgent message could use (relatively) expensive GPRS, while large, non-urgent data could wait until a free connection opportunity arises (either locally or via a “free” access point).

The job of the Connectivity Manager in Haggle is simply to encapsulate a number of Connectivity objects and to initialize the appropriate number at start-of-day. Each Connectivity must support a well-defined interface including functionality for neighbour discovery, opening/using/closing communications channels, and estimating the costs (in terms of money, time and energy) of performing network operations. The Connectivity must interface with the underlying hardware to provide this functionality.

There will be one Connectivity instance per instance of a network interface on a node (so there were two 802.11 interfaces there would be two Connectivities created during initialization). This is because a Connectivity is regarded by the Resource Manager as a schedulable resource, so it must be clear exactly how many resources there are. Since the Resource Manager expects to schedule the network interface, all operations that result in network activity, including operations initiated by the Connectivity’s code itself, must be passed to the Resource Manager as “Tasks” (to be discussed in Section 3.6).

Connectivities also interact with Protocols, providing them with Neighbour lists gained during neighbour discovery. A Neighbour is a potential next-hop by which particular Protocols may know how to send data of particular types to particular NOs. We differentiate between “non-Internet” Neighbours which are direct next hops running Haggle, and “Internet” Neighbours which are next hops supporting IP for accessing the Internet. Typically, each Protocol will only be interested in one type of Neighbour.

Neighbour discovery can take various forms. In 802.11, any node with reception turned on can see beacons from access points (APs) which announce their existence. For Bluetooth, neighbour discovery is an active (and time-consuming) process. For GPRS, neighbour discovery is implicit in that when base station coverage is present, an Internet Neighbour is visible.

3.4 Protocol Manager and Protocols

The Protocol Manager is only responsible for encapsulating a set of Protocols, and initializing that set at start-of-day. A Protocol is a method by which DOs can be forwarded to Names, e.g. SMTP, HTTP, a direct peer-to-peer protocol etc. This highlights an architectural difference between Haggle and traditional network stack, since these protocols are normally at the application layer and forwarding decisions are normally considered to be taken at the IP layer underneath.

Each Protocol monitors the Neighbours visible through the Connectivities, and using these Neighbours it can determine which NOs it can deliver to. This decision can also take into account the type of data being forwarded, e.g. an SMTP protocol can send a message to an email NO, but it may refuse to accept non-message data (e.g. application signaling) since that is not suitable to appear in an inbox.

For Protocols which must accept incoming connections, e.g. a direct peer-to-peer protocol, they must provide each Connectivity with enough information so that it can redirect incoming data to that protocol. This is akin to listening sockets in the existing architecture. Some Protocols do not accept incoming connections; typically, all Internet-using Protocols (HTTP, SMTP, POP) act as clients to existing servers and so must initiate connections themselves. While seemingly simpler, this proves a source of additional work due to polling requirements — for example, the POP Protocol must use Resource Manager “Tasks” in order to request that email accounts be checked (if Internet connectivity is available).

3.5 Forwarding Manager and Algorithms

The Forwarding Manager provides an API to applications to cause data to be sent remotely, encapsulates a number of Forwarding Algorithms, and sends the Forwarding Tasks that are produced by them to the Resource Manager.

Applications request data transfers by specifying a set of DOs (the heads of a larger set of linked DOs) and a set of NOs (the heads of name graphs). The Forwarding Manager constructs a Forwarding Object (FO) which is a DO with metadata about the forwarding operation, and which is linked to the destination NO graphs, to the DOs, and to an NO graph describing the sender (useful for replies either from applications or from Haggle’s internal replies with DOs matching DO filters). The metadata can include expiry times and expiry hop counts, security information and routing hints for forwarding algorithms (described below), as well as a list of NOs to which the data has already been sent. Other metadata can also be present — the DO format allows for simple extensibility, and unknown fields can be easily ignored across different implementations/instantiations of Haggle.

3.5.1 Forwarding Algorithms

Once an FO is created, it is the job of one or more Forwarding Algorithms to determine suitable next hops. In Haggle, we precisely define the role of a Forwarding Algorithm as: for each FO, propose a set of {Protocol, NO, Neighbour} tuples which this FO could be sent to, and a scalar “forwarding benefit” associated with each tuple, which is an estimate of the probability that sending it that way would result in successful end-to-end delivery. The tuples and benefit levels change continuously, depending on the available connection opportunities, the known information about the FO (e.g. if it expires or has already been delivered), etc.

Haggle has the useful feature of allowing many Forwarding Algorithms to be in use *simultaneously*. Note that we do *not* mean that traffic is generated according to the wishes of all Forwarding Algorithms, since the Resource Manager will be responsible for accepting or denying the proposed actions of every algorithm. The simplest algorithm is a direct forwarding algorithm, which only proposes to send an FO if it can directly reach an NO which is present in its graph of destinations (i.e. it does not make use of any multi-hop communication), with a forwarding benefit of 100% by definition. Another algorithm is the epidemic forwarding algorithm [19], which sends the data to every Name that is reachable, i.e. it floods the data, but with a correspondingly low forwarding benefit. Haggle can also make use of MANET algorithms such as geographic [11] or distance-vector [12], as well as opportunistic store-and-forward [16, 21] such as mobility-based [6, 9, 10] algorithms. Haggle is able to use many of these algorithms simultaneously, obtaining the “best of many worlds” in that for each forwarding operation, a different Forwarding Algorithm may prove best, due to availability or not of per-algorithm state information. Such state information can be exchanged in Haggle by Forwarding Algorithms themselves creating Forwarding Tasks targeted at nearby nodes.

For each FO, and each Protocol, Name, Neighbour that an FO is proposed to be sent to (with associated benefit), the Forwarding Manager creates a “Forwarding Task”, to be executed when the Resource Manager decides on doing so. When executed, the Forwarding Task causes the associated Protocol to send the FO to the NO, via the Neighbour.

3.6 Resource Manager using Tasks

As referred to many times above, all outgoing or incoming network operations in Haggle are proposed to the Resource Manager and executed only if/when the Resource Manager chooses; they are not necessarily executed in order or at all. A Task comprises a method of accomplishing the work, the benefits of achieving the Task, and the costs of performing the Task. This definition is deliberately abstract so that the Resource Manager can compare between different possible actions while knowing little about the actual mechanisms or details of formulating or carrying out Tasks.

Both the costs and benefits of Tasks are re-evaluated by the Resource Manager each time a Task is considered for execution, using callback functions provided by at Task creation time. A Task's cost describes measurable, true costs to the node, expressed in terms of energy, money and time-on-network. Time-on-network refers to connectivity-specific nature of the Tasks being scheduled, and represents the opportunity cost of not doing something different with that time. Task benefits describe the estimated utility to end users of executing a Task. This is not a simple calculation to make. Components of this benefit include forwarding benefit which, as described above, is the likelihood that this action will result in a successful end-to-end transmission, but also application benefit (how worth it to the application is that transmission), and user benefit (what priority is the action to the user). We would also like to be able to take into account priorities specified by the owner of the devices, e.g. "I don't want to spend money on others' traffic, but I will allow Haggle to share a limited percentage of my battery")

Tasks can be either asynchronous or immediate. Asynchronous Tasks are the norm, and (as the name suggests) the Resource Manager is provided with a callback to asynchronously call when it wants the Task performed. Asynchronous Tasks can be "persistent", i.e. once they are executed they persist to be executed again later — otherwise, a Task is only called at most once. Immediate Tasks are used when a particular operation must either be done now or not at all, and they are used to deal with events such as incoming network connections which must either be accepted or rejected.

Benefits and Costs for asynchronous Tasks are often varied by their owner over time. For persistent Tasks, the benefit of a Task that has just been executed will be low, e.g. checking for new email on a server or checking for new Neighbours is not that beneficial if the operation was last performed 1 second ago, and the benefit would rise over time. On the other hand, an FO with an imminent application-set expiry time becomes less and less beneficial to forward in a multi-hop fashion, since the likelihood of reaching the destination in time for the application purpose becomes low. Costs are calculated with the assistance of the Connectivity that the Task will be using — typically, the Task owner would provide an estimate of the number of bytes to be sent/received via a particular Neighbour, and the Connectivity can translate that into the expected money, energy, and time that this transmission will take.

Once a Task is being executed, the Resource Manager can also be asked for a "continuation", i.e. if the scope of the work being done by the Task increases over the initial cost/benefits specified, then the Resource Manager can be synchronously polled for permission (with additive cost/benefit over the existing Task) to authorize work on the extended Task. This is useful for circumstances such as email checking, which may discover a large email waiting for download.

The Task model is in marked contrast to the traditional network stack, where networking operations proposed by applications or operating system functions are always attempted. The centralization of decision-making about what Tasks are worth doing at all, and which are more important at any time, allows Haggle to have a number of advantageous features. First, Haggle can execute periodic Tasks, such as email checking, on a dynamic schedule instead of at fixed intervals. These periodic Tasks can be performed more often when bandwidth and energy are abundant, and less often otherwise or when there are more important Tasks. Second, Haggle can allow applications to request network operations of varying priorities, including speculative Tasks which are often not possible or worthwhile, but which would be executed as and when an opportunity arises. Third, Haggle can implement device-wide policy about resource conservation and consumption, and not rely on individual applications to do so (e.g. to throttle low-priority Tasks at particular times).

The Resource Manager is a key illustration of how Haggle is "layerless" since Tasks come from many different managers. We have already described examples of Tasks generated by the Name Manager (querying nearby Neighbours for name information), Forwarding Manager (exchanging state information), Protocols (email checking), and Connectivities (Neighbour discovery).

3.7 Security and Privacy

In the current version of Haggle, security and privacy have not been addressed as key concerns; we chose to narrow the scope of the problem to exclude them, so as to allow us to make progress. We intend to introduce security primitives as a core concern in future versions of Haggle. However, we have made an initial analysis of the potential security threats that Haggle raises, discussed below, though this has not been acted upon in the prototype implementation.

Many data security issues in Haggle can be handled using standard security techniques such as encryption, access control, and data signing. Haggle merely makes it more likely that there will be a man-in-the-middle attack. One proviso is that many security techniques rely on access to a trusted third party, e.g. a certificate signing authority. This access may be available less often when using Haggle. One interesting approach would be to accept data which is uncheckably signed, but somehow mark it (both internally and to the user) as "untrusted" until the signature can be checked through infrastructure access.

There are particular security and privacy issues in the use of name graphs. A name graph can contain sensitive information, e.g. a user's email address and/or phone number, or the number and type of a user's devices (and hence how worthwhile it is to rob the user). Since Haggle potentially exposes the full graph to everyone who can see an FO with the graph, this could prove to be a breach of privacy. One solution might be to restrict trust to particular groups of users, e.g. the personnel of a company, and avoid sending messages through untrusted nodes, except when the name graph and data have been encrypted and authenticated to the extent that those nodes could not obtain any useful information, and could only help by passing the data on to non-privacy-sensitive names (e.g. MAC addresses).

There are also privacy issues to do with neighbour discovery protocols, since one's devices essentially beacon their identity. This could allow tracking of the user. This problem is not unique to Haggle, and many devices already essentially act as beacons, e.g. laptops using 802.11 placing their MAC addresses on each frame.

Resource theft or resource denial-of-service is an interesting issue for any system in which user-owned nodes cooperate to achieve their goals, and resources are limited. In Haggle, we have a built-in mechanism to cope with this, namely the Resource Manager, which already makes judgments taking into account the utility of a given action to the user, and the device owner's preferences. On the other hand, this offers a single point of attack whereby a remote exploit might allow an attacker to take full control of the device, so securing the Resource Manager will be of particular concern.

Finally, we might ask the question of what motivates any node to spend its resources assisting any other node. An incentive to cooperate can be created in may ways — using reputation systems, micro-payments, or social kudos/disapproval. Furthermore, in some possible deployments of Haggle, e.g. within an enterprise, there is a pre-existing incentive to cooperate so this may not be a problem.

4 Prototyping Haggle

The Haggle prototype has been developed using Java, targeted initially at Windows XP (as described in the experiments later), but also ported to Linux and Windows Mobile platforms. This development has been conducted using sourceforge.net, an open source development site, under the GNU General Public License (GPL), and remains open source and available to other research groups.

We now describe some implementation details of the prototype. This implementation was demonstrated at UbiComp 2006 [14] and ACM MobiCom 2006, and is used to conduct the experiments detailed in the next section.

4.1 Data Manager using SQL

We implemented Haggle's Data Manager using Java's standard SQL interface, backed by MySQL. Although any SQL back-end will work, we chose MySQL because it is freely available. We chose SQL as the storage mechanism for the Data Manager because it provides an easy interface for persistence and search. DOFilters are implemented as regular expression-like queries, and for the most part these queries can be translated directly into SQL "select" statements, to take full advantage of SQL's optimized environment. In the future, to support Haggle on mobile phones and other low-powered devices, we are considering SQLite as a lightweight SQL server.

We use two SQL tables for Haggle, one "attributes" table with fields {DOID (int), type (String), value (String)}, and one "links" table with fields { head-DOID (int), tail-DOID (int)}. In the links table, the former DOID can be negative to indicate that the linker is not a DO but is an application or a Haggle manager.

4.2 Resource Manager

For the prototype, the Resource Manager does not consider energy or money costs, but only time-on-network. This was because we used no charged network, and our best estimate of energy consumption currently is simply proportional to the time-on-network. Ideally, we would receive energy consumption estimates from the network interface card or driver, since they have a greater knowledge of their radio characteristics and medium state. We used no external policies restricting actions, so all nodes are cooperative, and set the "user" component in the benefits to be always 100%.

To compare cost and benefit and obtain a score, we simply multiplied the "forwarding" and "application" components of the benefit (since both are expressed as percentages), and divided by the time-on-network cost. For asynchronous Tasks, we select the top scoring Task for execution, however we applied a minimum score threshold so that repetitive Tasks (which are implemented using rising-over-time benefits that reset on execution) do not continually execute. We do not execute more than one asynchronous Task at a time on each Connectivity in this initial implementation.

Immediate Tasks are scored similarly, but if they pass the threshold they are always accepted. This is because of the nature of immediate Tasks - they represent now-or-never network connections. In the prototype, the only immediate Task is caused when the P2P Protocol (described below) receives an incoming network connection. Since the presence of an incoming connection implies a significant amount of synchronization overhead already accomplished, it makes sense to accept these whenever possible (denial of service attacks are not considered in this iteration of Haggle). Note that accepting immediate Tasks can result in more than one Task executing simultaneously on a Connectivity; no new asynchronous Task would be scheduled until all the running Tasks complete.

4.3 Connectivity: 802.11

We chose to focus on 802.11 connectivity for our prototype. This is because it is a widely used wireless access network, and is available for a range of devices from laptops to mobile phones. It also offers both neighbourhood and infrastructure connections (through ad hoc mode and infrastructure mode respectively) which allow us to explore the range of Haggle capabilities using a single connectivity type.

Implementing 802.11 support requires a native driver component which communicates with the NDIS driver interface for Windows XP. We implemented this in C++ resulting in a dll file, which provides our Java code with capabilities such as putting the 802.11 interface in ad hoc mode or AP mode.

The 802.11 driver provides neighbour discovery capabilities allowing us to discover the existence of a HaggleAdhoc network (name chosen by us) when another Haggle device's 802.11 card is in ad hoc mode. We can also discover the existence of APs in this fashion. Since discovery is passive (in that it occurs through simply hearing beacon transmissions and does not involve sending by the node), this can be done frequently.

Unfortunately, for both the AP and ad hoc case the information received is not enough to construct Neighbour objects that can be used by Protocols. For the AP case, we would claim to be able to see an Internet-capable Neighbour when seeing any AP, and this is not true since many APs may be "closed" (i.e. encrypted) or not connected to the Internet. For newly-seen APs, we therefore first create a Task to check the AP for Internet connectivity, whose Benefit drops as the number of known usable nearby APs rises (i.e. if we know we have two usable APs, a third is less beneficial). We store the AP records as DOs so that we do not have to perform this check in future, though the record is reset if the AP later proves to lack Internet connectivity.

When we see the "HaggleAdhoc" ad hoc network, we know that there are one or more other Haggle devices present, but we do not know their MAC addresses. We therefore create a persistent Task which uses UDP to broadcast the node's MAC address on the ad hoc network. Its benefit rises over time and resets on execution, making it execute periodically. We also listen for the UDP broadcast packets and from these can obtain neighbour information to export to the Protocol Manager.

For 802.11, the time-on-network cost calculation takes into account the overhead of switching between AP and ad hoc modes, which can be a number of seconds when switching into AP mode due in particular to the latency of DHCP to provide an IP address. We used typical values of the switching overhead based on our experience. We also applied a per-byte time-on-network cost (i.e. taking into account the bandwidth and the size of the data being sent). We used a lower bandwidth estimate for AP mode than ad hoc mode since we expect the access link to be the bottleneck in AP mode. This would ideally be dynamically measured on a per-AP basis rather than statically estimated. This cost structure causes Haggle to avoid switching to AP mode, but amortize the cost of the switch by performing high-scoring Tasks which require AP access before switching back to ad hoc mode.

4.4 Forwarding Algorithms: Direct and Epidemic

We implemented two Forwarding Algorithms so far, namely "direct" and "epidemic". The direct algorithm only proposes to send an FO to a NO if that NO appears as a destination of the FO, with a forwarding benefit of 100% since the destination is always reached in this case. The epidemic algorithm proposes to send every FO to every NO where a Protocol says it can support that transfer, but does so with a lower forwarding benefit since it has no idea whether it will reach the destination this way. We do not need to special-case the epidemic algorithm for the event that a neighbouring NO is the destination, since the direct forwarding algorithm handles this well.

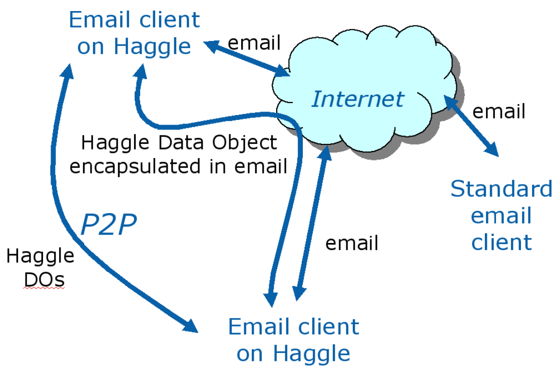
4.5 P2P protocol

We implemented a simple P2P protocol so that neighbouring Haggle nodes can send FOs to one another. This serializes and sends the FO and all DOs/NOs underneath it. The P2P protocol currently makes uses of TCP/IP, since it allowed us to code this aspect more quickly, without having to implement fragmentation/reassembly and error detection/correction. However, we do not rely on TCP/IP — this functionality could be reimplemented "raw" over link layer frames if that proved to be desirable.

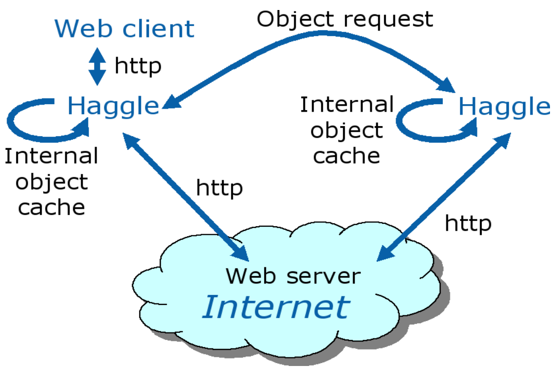
Name Graph Construction

We now discuss how Haggle creates name graphs and shares them with neighbours. We first deal with how Haggle nodes create their own name graphs. On first startup, a Haggle node creates an NO containing a GUID representing the node itself, and finds out its MAC addresses (from the connectivities) which it places as children NOs of the node NO. When the email proxy in Haggle (to be explained in Section 5) sees an outgoing email, it captures the sender email address and personal name and turns both into NOs with the name NO linking to the email NO, and also adds a link from the name NO to the node NO. Many email addresses/names can exist on the same Haggle node (and therefore point to the node NO). Also, if the node discovers that other nodes claim that email address, then more than one node can be a child NO of the name NO. An illustration of a name graph created in this way is found in Figure 3.

When a Connectivity finds a Haggle Neighbour, a corresponding NO is generated by the Name Manager for that Neighbour's MAC address. If that NO is not part of an existing name graph, it is referred to as a singleton NO. The Name Manager proactively creates FOs destined for singleton NOs containing this node's name graph, as well as a DO essentially saying "who are you?". The Name Manager uses a persistent DO filter to register an interest in incoming NOs, which it merges into its knowledge base. It also uses a persistent DO filter to register an



(a) Email Application



(b) Web Application

Figure 4: Haggle Email and Web Applications

interest in incoming "who are you?" DOs, upon arrival of which it creates a replying FO with the node's name graph information. In this way, when two nodes which have never met before meet, they are able to obtain each other's name graphs, despite the absence of any other nodes or infrastructure.

The method above does not aim to preserve privacy at all, and is insecure in many ways. However, it suffices for now to demonstrate Haggle's functionality. See Section 3.7 for a discussion of security/privacy issues.

5 Support for Existing Applications

Based on our introductory motivating examples, we have chosen to target email and web as our prototype applications. To be clear, by "email" and "web" we mean the messaging and hyperlinked-information applications, rather than the protocols that underly them.

Both of these applications enjoy huge support from the pre-existing infrastructure deployment of servers and content. It is a crucial feature of Haggle that we can take advantage of this infrastructure as well as providing new functionality (operation when infrastructure is not available). This makes Haggle much more compelling to existing users of that infrastructure, and the value added by Haggle provides motivation for it's deployment.

To provide legacy support for existing email and web applications, we implement localhost SMTP/POP and HTTP proxies alongside Haggle. This allows users to keep using the same applications they habitually use (we have tested Outlook Express, Thunderbird, Internet Explorer and Firefox) with only minimal reconfiguration. The modes of communication and operation of email and web applications using Haggle are highlighted in Figure 4. We will first describe how Haggle provides support for email, followed by the description of web support.

5.1 Email

Supporting email in Haggle consists of two components: an SMTP/POP proxy for interfacing with email clients, and SMTP and POP Protocols inside Haggle that communicate with email servers. The SMTP proxy accepts emails provided by the user's mail client and translates them into an FO with a DO for the email itself and a DO per attachment to the email. For each of the list of destination email addresses, a "top level NO" is found by seeing if that email address is in one of the Name Manager's name graphs, and following the graph up to its root, and these NOs are added as destinations for the FO. If this step were not taken, then an outgoing email would only ever be sent via its email address. As we have already described in the previous Section, the SMTP proxy also creates NOs for the personal name and email address of outgoing messages, and lets the Name Manager know that these are names by which that node is known, so that it can share naming information with Neighbours appropriately.

The POP proxy listens on port 110 for incoming connections from the user's email client. When the client checks for new mail, the POP proxy uses the Data Manager to search for DOs which represent emails and are not yet provided to the email client. From those objects returned the POP proxy reconstructs the full email message (including attachments) to return to the email client (marking the email as delivered). Because email servers require some form of authentication for receiving emails (and some require authentication for sending emails), it is necessary to pass authentication information from the email application to the SMTP/POP Protocols. To achieve this, when the user's mail client first check mails from the POP proxy, Haggle saves the authentication information the mail client sends.

When an Internet neighbour is available, the SMTP Protocol can translate FO/DO/NO-format emails to destinations that are RFC2822 formatted mail addresses. When doing so, the Protocol reconstructs the email format and adds additional Haggle metadata in a hidden attachment named "haggle.data" (in addition to any other attachments that might be present).

Similarly, the POP Protocol translates received email messages into FO/DO/NO graphs. If the "haggle.data" attachment is present, it uses this to perform a lossless translation into the original FO/DO/NOs, otherwise; if it is not present, the email has not been sent from a Haggle node and the FOs/DOs/NOs are created as described above. When the POP Protocol has the proper authentication information described above, for each Internet neighbour visible, it creates a persistent Task to periodically check for email from the server through that Neighbour. In this way, the Resource Manager can choose the correct time and connectivity by which email is checked according to the current resource constraints and demands from other Tasks. In the first instance, this Task's costs/benefits just cover checking for new email and not the downloading of emails — the POP protocol uses the Resource Manager's Task continuation functionality to request permission to download each email it finds.

Note that Haggle email users need to have an email account as normal. A "free" email account such as the service from Google (GMail) is perfectly adequate, and is what is used in our experiments later.

5.2 Web

The Haggle web proxy operates as follows. When the user's browser makes an HTTP GET request, the proxy first checks if the object requested is in a pre-existing DO, and if so returns it. If it is not, and an Internet neighbour is available, the proxy requests an immediate outgoing connection using the Resource Manager's Task interface. If the request is allowed, the proxy creates a connection to the email server, and acts as a mirror between the proxied connection and actual connection, with the proviso that it also stores the incoming web data as DOs.

If there is no Internet neighbour visible, or the immediate Task is denied, the proxy creates a DO filter which matches a webpage with the appropriate URL, and an FO encapsulating that, without any destination NOs. It returns a message to the browser allowing the user to see that Haggle is attempting to service the request, along with a refresh command so that browser checks again every few seconds for updates. The FO will eventually be “sent” either using the P2P protocol (to share the request with a neighbouring Haggle node) or using the HTTP protocol. The latter offers service only to FOs with DO filters requesting URLs, and it operates by using an Internet neighbour to connect to the web server and download the webpages, inserting them as DOs and marking the FO as handled. These DOs match the DO filter, so the results are provided to the requester. If the requester is not local (i.e. the DO filter came in using the P2P protocol, the Data Manager will itself create a FO sending the DOs to the requesting node.

When the HTML Protocol receives an HTML DO, it must also request the embedded objects for that page. We parse the HTML to find the linked objects, and again use a DO filter to obtain them as described already. The Protocol must maintain links between the DOs comprising an HTML page and its embedded objects, so that when a DO filter is responded to, the full set of embedded objects is sent along with the HTML page itself. This parsing is not easy, and our algorithm is quick rather than complete — to be complete involves great lengths such as parsing and executing javascript on the page (since some pages use javascript to load images). However, if our parser misses a linked object, all is not lost as the web client itself will request it, albeit potentially incurring more delay.

6 Experiments and Results

In this section we describe several experiments using the motivating applications described earlier. We provide qualitative results showing the new capabilities that Haggle enables, in addition to quantitatively demonstrating that Haggle’s implementation, although not optimized, has acceptable overheads.

We conducted the experiments on two laptop computers, which we will call *node1* and *node2*. Both are running Windows XP. Node1 is equipped with an Intel 3945 mini-PCI 802.11 interface, and node2 is equipped with an Intel 2200 802.11 interface.

Email

For the email experiments we created several accounts using the GMail[2] service. GMail provides POP and SMTP services over an encrypted and authenticated SSL link. This allows us to have one configuration which works from within any network that allows Internet access. However, there are limitations with using the GMail service. Though there is no limit for the size of email received, GMail restricts the size of outbound emails to be 10 megabytes or less.

For the quantitative experiments, we send emails from node1 to node2 one account to another, of varying sizes, ranging from 10 bytes (no attachment) up to a 10 megabyte attachment. Each size is sent seven times over a 3 Megabit download / 800 Kilobit upload DSL link. An automated script is used to send an email from node1 to node2, with node2 configured to check its inbox once every 5 seconds. The script ensures that for every email that node1 sends, node2 must receive it first before node1 sends the next email. This eliminates any congestion effects in the results.

Figure 4(a) shows the functionality of email application using Haggle, and Figure 5 shows the latencies for end-to-end delivery of various-sized emails under different network connectivity conditions, both with and without Haggle.

[2]http://mail.google.com

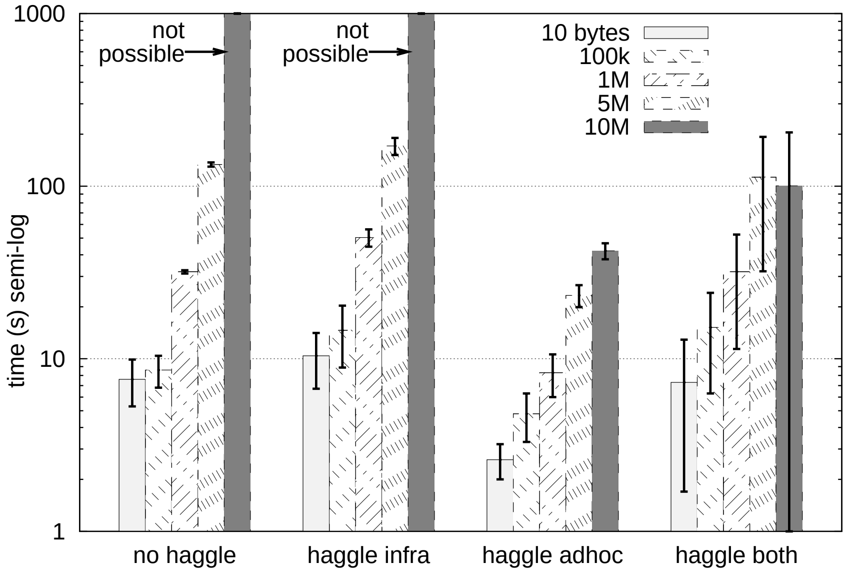


Figure 5: Mean email end-to-end delivery times, with standard deviation. Individual bars indicate attachment size. Lower values are better. Note that for the *no haggle* and *haggle infra* cases, it was not possible to send 10M emails due to server limitations.

The *no haggle* cluster shows email clients not using Haggle, i.e. using infrastructure as normal, and is our control case. The *haggle infra* cluster shows the same thing (infrastructure-based email) but using Haggle, and we can see that the overhead added by Haggle is low.

The most important result is shown in the *haggle adhoc* cluster, which shows Haggle sending and receiving emails without infrastructure present. (The equivalent graph for no-Haggle would be infinitely high bars all the way across). *haggle adhoc* also shows the fastest transfer for all email sizes, which is to be expected since this mode of operation uses only the 802.11 network and only once, while *no haggle* and *haggle infra* use the 802.11 link as well as the bottleneck DSL access link and other Internet links to the server, and does this twice (once for send and once for receive). Also, we notice that the 10Mb attachment does not get transferred in the infrastructure case (due to GMail's outgoing email size limit), while it does in the ad hoc case.

Ideally, the *haggle both* performance would be close to the *haggle adhoc* performance. This is not so (though it is still comparable with *no haggle*) and there is a larger variance in the numbers. This is due to interaction between the 802.11 Connectivity and the POP Protocol. Because an Internet Neighbour is visible in this scenario, the POP protocol is requesting Tasks to check the email account at GMail. This is additional work that Haggle is doing which it is not doing in the *haggle adhoc* case. Added to this is the fact that switching between ad hoc and AP mode in 802.11 incurs a significant overhead, since lost DHCP request packets cause long timeouts. This can be remedied by improving 802.11 performance, and also by making use of multiple network interfaces if they are available; both of these are discussed later in the paper.

Web

In our web experiments we focus on the HTTP protocol and the retrieval of static and dynamic content from content providers. For the purposes of this experiment we do not consider other non-HTTP web objects such as Flash. We chose four different webpages to cover a range of complexities, sizes, and scenarios.

- http://www.sigmobile.org/mobisys/2006/program.html
  The Mobisys 2006 technical program is relatively simple consisting mostly of text and no dynamic content. The transfer size is 64Kb. Attendees at MobiSys might want this page frequently to see what's on next; however, at conferences there are frequently failures of Internet connectivity [7].

- http://www.torontolife.com/restaurants
  Toronto Life is a popular city life and culture site with moderately complex layout. The transfer size is 500Kb, sent as 380K of gzip-enabled web traffic. This page is one that might be useful in our train-based motivating example.

- http://news.bbc.co.uk
  The BBC news site is a relatively complex website with frequently updated content. The transfer size is 370Kb, sent as 100Kb of gzip-enabled web traffic. This page is very highly viewed, so there is a reasonable likelihood of a copy being present in a group of users.

- http://www.rottentomatoes.com
  Rotten Tomatoes is a movie review site which contains a dense layout with dynamic content. The transfer size is 834Kb, sent as 230Kb of gzip-enabled web traffic. This might be looked for at a cinema while deciding what to watch, with a reasonable expectation that others in the area already looked it up.

For each of the experiments we retrieve the contents seven times, each time clearing all caches. We measure the end-to-end time as starting from the moment of request at the browser until the browser finishes loading and rendering all content on the page, using the FireFox web browser with the FasterFox plugin since it contains a built-in page load timer (we turned off all other functionality that FasterFox offers).

Figure 4(b) shows the operation of the web application over Haggle, and Figure 6 shows the performance results for retrieving the above described webpages with and without Haggle.

Between *no haggle* and *haggle infra* the comparison is less favorable than for the email case. We attribute this to the overheads of (a) the time taken to parsing the HTML pages to determine linked data objects, (b) the overhead in the proxying approach, since the web client opens and closes multiple socket connections to inform Haggle of different objects it requires, (c) inefficiencies in our Data Manager implementation in that the webpages are stored data persistently before they are transmitted to the web client.

All of these aspects can be improved with optimization work, particularly the parser which is being compared with the extremely highly-optimized parsers found in web browsers (since they have been designed and tweaked to reduce latency). Therefore we do not consider this result to be indication of any fundamental limit in Haggle.

For each web object retrieved from the Internet, the web proxy attempts to reconstruct its relation with other objects which it may have been linked from. Because web browsers make multiple simultaneous connections to the proxy and use each pipe in parallel, we must examine the headers of the objects returned in order to properly associate objects to webpages. To do this the web proxy examines the referrer tag of the HTTP response message for the retrieved object to determine from which other object the current was requested from. After finding the

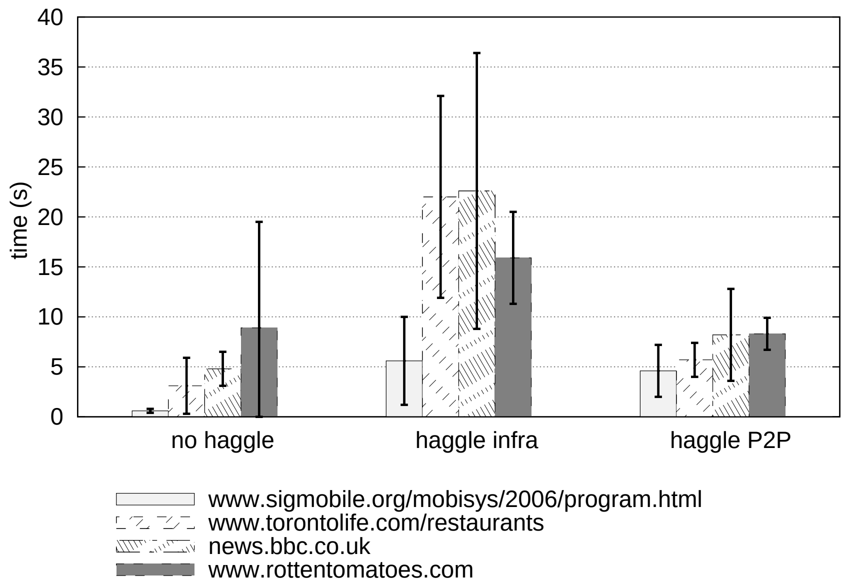


Figure 6: Mean webpage retrieval times, with standard deviations. Lower values are better.

originating object as a DO, the web protocol creates a link from it to the newly retrieved web object. This search and link requires queries to the Data Manager which adds an overhead time to each web object retrieved. We can see this overhead in the comparison between *no haggle* and *haggle infra*.

For the *haggle P2P* experiments, we have node1 configured to enable access to the access point as well as communicate in ad hoc mode. The four webpages described are then visited using the web client on node1 so that it has an cache of DOs representing those pages (and embedded objects). At this time, the access point is turned off, modeling node1 being moved to an infrastructure-free location. Node2 is only able to communicate in ad hoc mode, and is placed near node1. We request a webpage on node2 (clearing the cache each time an experiment is run); since node2 does not have Internet connectivity, it sends a DO filter requesting the webpage to node1, who returns the matching webpage, with embedded objects.

We observe that our Haggle implementation can retrieve webpages even when there is no Internet connectivity available, if another user in the environment has previously retrieved the webpage and has it cached. This is fundamentally new functionality; the equivalent bar graph without Haggle would contain infinitely high bars. The performance is comparable with the *no haggle* case, there is more initial overhead, but the faster bit-rate (since transfers proceed at the full 802.11 rate and not the bottleneck access link rate) means that for the larger rottentomatoes.com page, the latency is matched.

7 Discussion and Future Work

In this section we discuss some of the next steps for Haggle and the future research that Haggle enables. We also highlight Haggle's shortcomings.

7.1 Future Work and Improvements

Web Search

One problem with our web application is that users in an ad hoc connectivity environment must know the precise URLs of the webpages the other users have in order to obtain their cached content, at least for the entry page into a site. In reality, users seeking information often use search functionality and are happy to receive it using any number of different websites.

To address this, we are currently implementing special-case handling for web searches, allowing Haggle's exposed metadata and searching functionality with DO filters to be reused so to obtain ad hoc search functionality. This involves:

- The web proxy handles requests for search websites locally – instead of requesting the URL remotely, it provides its own search webpage (redirecting the user's browser to a new URL to avoid confusion by the user).

- When a search is specified, the web proxy generates a DO filter asking for an HTML page with "keywords = ⟨the search string⟩". It also generates a webpage containing the current results – i.e. any existing cached pages which match this DO filter.

- The HTML protocol makes sure to fill in keywords on received webpages. Simple methods to accomplish this are either to copy the keywords from the existing HTML META tag of the same name, or to heuristically extract keywords from the page title, headings, or common words/phrases. For more advanced indexing, we can use techniques from existing desktop search products.

- We add a Web Search Protocol which can process FOs containing DO filters which are web searches, by going to a web search site and downloading the top N links, via any an Internet Neighbours visible.

This illustrates how easy it is to add new functionality under Haggle; to implement this as a stand-alone application would be more complex than the four simple steps above.

Optimizing the 802.11 Connectivity

As we saw during the email experiments, the 802.11 Connectivity in Haggle carries significant overhead due to the need to switch between AP and ad hoc modes, particularly when switching to AP mode if the DHCP handshake incurs packet loss. A simple optimization would be to more aggressively retransmit DHCP packets. A more thorough solution might be based on the use of techniques like Virtual WiFi aka MultiNet [3], so that the Connectivity could maintain one or more AP associations while also being in ad hoc mode.

Multiple Connectivities

The experiments above have only used 802.11. Laptops and smart phones typically have multiple interfaces: Bluetooth, GPRS/UMTS, SMS/MMS, infrared, and in the future WiMax or Wireless USB (Ultra-wideband connectivity). Many other research projects have explored the problems of heterogeneous network interfaces [17], and struggling with how this can be done sensibly using IP routing.

Haggle is designed to be able to take advantage of multiple Connectivities, so adding support for a new Connectivity should be easy. We are currently working on support for SMS, GPRS and Bluetooth. When combined with centralized resource management and with late binding using name graphs, Haggle should be able to allow the appropriate Connectivity to be used per-Task, e.g. sending short emails over SMS, using GPRS for checking of new email existence but waiting for an 802.11 AP (or other cheaper method) to download large emails. Haggle offers a new approach to this problem, complementing existing IP-based approaches such as [20].

Resource Manager

Haggle's centralized decision making in the Resource Manager is a very powerful feature; we have not yet begun to explore the potential here. We highlight here a number of areas that need additional research.

The current implementation performs only a single Task at a time, whereas many may be parallelizable. It is not clear how to go about choosing the group of Tasks that (a) *can* be scheduled together (i.e. their demands on the underlying Connectivity do not conflict), or (b) *should* be scheduled together, i.e. it would be helpful rather than harmful.

Another limitation of the current Resource Manager is that it is reactive only, and does not attempt to predict future connectivity options (e.g. as OCMP [15]). For example, currently Haggle may epidemically send a message to a remote host when, in five minutes, the user will arrive at their place of work and have free broadband connectivity.

We do not use monetary costs or energy consumption in our current decision process, however these are key issues in device connectivity today, as they impact battery life and the potentially high cost of "always-on" connectivity. Particularly when we have multiple connectivities, we will likely be faced with situations where we have to choose between connectivity options which trade off forwarding benefit against monetary cost or energy consumption.

Native Email and Web Applications

The applications ported to Haggle so far use proxies so that the applications themselves need no modification. This is an important feature, providing backwards compatibility. However, there are a number of advantages to having these applications ported to running directly over Haggle, over and above the removal of the overhead due to proxying.

First, the applications currently do not use the Data Manager to store data persistently, instead duplicating information in a local file system. Also, the applications' naming data, e.g. the address book in the case of email, does not get shared automatically with Haggle, and we are forced to use heuristics to map outgoing emails onto the appropriate name graph destinations, as well as obtain the authentication data for POP server access.

Next, the user interface experience can be improved. At the moment, the email application shows that the email is sent as soon as the Haggle proxy receives it, however this is of course untrue. While Haggle knows more information about the status of the message, this is not presented to applications. With the web application, when a webpage is not immediately available a status page is presented to the user giving them feedback, however this is only true if the user does not navigate away from the page, so they must keep pending pages viewable in windows or tabs and keep checking them. A proactive notification may be more suitable.

Third, Haggle's additional API features over the Berkeley socket interface are not utilized. Web clients can use this API to ask Haggle for low-priority predictive downloading of webpages that the user might need, e.g. because they are often-viewed, or because they are linked to from the currently-being-viewed page. Such predictive requests are easily expressed in Haggle, using application benefit levels $<100\%$, and explicitly limited to within the user's resource constraints.

New Haggle Applications

Haggle can also support new applications, making use of its new features. One interesting application that we are targeting for future work is in the area of "resource-friendly media sharing". We observe that humans collect ever more media (photos, music, videos, etc) and wish to (a) share them easily with friends, and (b) have them transferred seamlessly between their devices, both mobile devices and those fixed at various locations.

With the current architecture, it is not possible for an application to easily express "all photos taken on my mobile phone should be sent to my home server for backup" without being at risk of consequences such as large GPRS bills when phone transmits holiday snaps over a foreign carrier, and the phone running out of batteries since it performs transfers even if there is scarce power. With Haggle, these concerns can be easily expressed, and persistent remote DO filters provide a simple yet powerful pub/sub mechanism for this kind of application.

Forwarding Algorithms

As we described in Section 3.5, in Haggle the job of a Forwarding Algorithm is well-specified, and we can support many running simultaneously. This makes Haggle an ideal test-bed for the development of new algorithms, and their comparison against existing algorithms in experimental situations.

User Issues

We have already discussed security and privacy in Section 3.7. This remains a key future work area for Haggle. In addition, there is the issue of usability of Haggle-based applications, which we have already discussed briefly above in relation to the email and web applications.

More broadly, we observe that users currently have a simple mental model of mobile networking. When they have an IP connection, their apps work, otherwise they don't. Haggle breaks this model for the good, as it provides more functionality. However, how will users mentally model Haggle? How will they understand what works under Haggle and what fails? One possible way in which users can be trained is to consider if they can see what they need in the environment. If a user can see the person that they are messaging, or they can see others who have data that they might want, then they should expect that Haggle might deliver that message or find that data.

However, it may not be that simple. We have described how the Resource Manager allows users to set preferences to limit their consumption of resources such as money and energy, and how applications use prioritized traffic. Providing a configuration interface for these preferences is no easy task. Furthermore, when something doesn't go through that the user expects to go through, assisting the user in troubleshooting (what preference should be changed, and what to?) is non-trivial. We expect that usability will be a key concern if we ever reach a situation where Haggle or Haggle-like functionality is common.

7.2 Limitations of Haggle

We now discuss some limitations of Haggle and give ideas as to how they can be overcome.

Integrating Protocols

We propose to include application-layer protocols into Haggle. This raises the issue of how new or existing applications can be integrated into Haggle. While we have shown the feasibility of doing this with two applications, this did involve the separation of the networking protocols from the applications and the manual integration of them into Haggle's codebase.

So how does this work for new or not-yet-ported applications, and does a new version of Haggle have to be installed by a user each time they need a new Protocol added? The answer is to define a plugin interface to Haggle to allow Protocols to be dynamically linked and implemented separately. The plugin interface must be secured (Haggle Protocols currently have nothing stopping them from accessing all DOs with potentially private data, or specifying arbitrarily high numbers of maximum-value Tasks to the Resource Manager to steal resources). While a similar situation would exist for Connectivities, and possibly for Forwarding Algorithms as well, those are not expected to change on the frequent basis that a user's set of applications that they use might change.

On the other hand, the ability of applications to be written without including any networking protocols, but still using networking by simply sharing data remotely, is a key advantage of Haggle. Every messaging application can be a “universal” messaging application in that all messaging protocols are supported, and the application writers can concentrate on the user interface aspects.

Streaming Data and Resource Reservation

Haggle currently does not provide support for streaming data, due to its use of asynchronous transfers of full data blocks (DOs). While streaming could be implemented by using many small DOs, Haggle currently has no method of reserving bandwidth for this other than using the Resource Manager as before. To implement streaming applications, it is possible to consider adding reservation capabilities to the Resource Manager.

Computational Overload

Haggle currently imposes significant computational overhead on the devices it runs on. The use of DO filters to search for data implies some overhead over methods such as listening on a socket, or accessing files from a well-known directory location (though DO filters also have advantages as we have already seen). The use of a Resource Manager which evaluates the cost/benefit of every possible network operation every time the network becomes free imposes significant overhead over simply picking the next packet off the queue as with IP.

For both of these cases, we note that we have not yet done any sort of optimization work on Haggle, and that this is an interesting area for future research.

8 Conclusions

Haggle is a new node architecture for mobile devices that was designed with a clean-slate approach. Haggle allows applications to become infrastructure-independent, freeing them from having to explicitly handle different and changing connectivity environments. We demonstrate Haggle using existing email and web applications, showcasing their ability to operate in ad hoc networking circumstances where they would previously have failed. This allows people to use the same application across different connectivity scenarios, something they cannot do today without manual configuration.

Haggle provides a uniform interface for exposing application-layer names and naming metadata to allow late-binding of data delivery. This allows Haggle to select the best protocols and connectivities to use, under any given network constraints, for reaching the destination. The Resource Manager provides a single informed decision point for managing the usage of network resources, allowing the node to coordinate the needs of its applications with the user's preferences.

In this paper we have described Haggle's architecture in detail, how the prototype was implemented, and quantitative experiments into Haggle's performance. We also describe a large number of research questions that implementing Haggle has caused us to consider. Haggle has been released as open source code at http://www.sourceforge.net/projects/haggle/.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proceedings of SOSP 1999*.

[2] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of PODC '99*.

[3] R. Chandra, P. Bahl, and P. Bahl. Multinet: Connecting to multiple ieee 802.11 networks using a single wireless card. In *Proceedings of IEEE Infocom 2004*.

[4] E. Cutrell, S. T. Dumais, and J. Teevan. Searching to eliminate personal information management. *Commun. ACM*, 49(1), 2006.

[5] K. Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of SIGCOMM 2003*.

[6] P. Hui, A. Chaintreau, J. Scott, R. Gass, J. Crowcroft, and C. Diot. Pocket Switched Networks and human mobility in conference environments. In *Proceedings of WDTN-05: The 2005 ACM SIGCOMM workshop on Delay-Tolerant Networking*.

[7] A. P. Jardosh, K. N. Ramachandran, K. C. Almeroth, and E. M. Belding-Royer. Understanding congestion in ieee 802.11b wireless networksrevised. In *Proceedings of IMC 2005*.

[8] M. Karsten, S. Keshav, and S. Prasad. An axiomatic basis for communication. In *Proceedings of HotNets 2006*.

[9] J. Leguay, T. Friedman, and V. Conan. Dtn routing in a mobility pattern space. In *Proceedings of WDTN-05: The 2005 ACM SIGCOMM workshop on Delay-Tolerant Networking*.

[10] A. Lindgren, A. Doria, and O. Schelen. Probabilistic routing in intermittently connected networks. In *Proc. SAPIR*, 2004.

[11] M. Mauve, A. Widmer, and H. Hartenstein. A survey on position-based routing in mobile ad hoc networks. *Network*, 15(6), Nov 2001.

[12] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc on-demand distance vector (aodv) routing. *RFC3561*, 2003.

[13] J. Scott, P. Hui, J. Crowcroft, and C. Diot. Haggle: a networking architecture designed around mobile users. In *Proceedings of IFIP WONS 2006*.

[14] J. Scott, M. H. Lim, J. Su, E. Upton, and P. Hui. Infrastructure-independent applications using haggle. In *Demo Abstract in Supplementary Proceedings of UbiComp 2006*, 2006.

[15] A. Seth, D. Kroeker, M. Zaharia, S. Guo, and S. Keshav. Low-cost communication for rural internet kiosks using mechanical backhaul. In *Proceedings of MobiCom 2006*.

[16] R. C. Shah, S. Roy, S. Jain, and W. Brunette. Datamules: Modelling a three tiered architecture for sparse sensor networks. In *IEEE SNPA 2003*.

[17] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins. Turducken: hierarchical power management for mobile devices. In *Proceedings of MobiSys 2005*.

[18] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *Proceedings of SIGCOMM 2002*.

[19] A. Vahdat and D. Becker. Epidemic routing for partially connected ad hoc networks, 2000.

[20] H. J. Wang. Policy-enabled handoffs across heterogeneous wireless networks. Technical Report CSD-98-1027, 23, 1998.

[21] W. Zhao, M. Ammar, and E. Zegura. A message ferrying approach for data delivery in sparse mobile ad hoc networks. In *Proceedings of the MobiCom 2004*.