University of Twente
Enschede - The Netherlands

Master's Thesis

Computer Science

Ahoy: A Proximity-Based Discovery Protocol

Robbert Haarman

January 18, 2007

Design and Analysis of Communication Systems
Faculty of Electrical Engineering, Mathematics, and Computer Science
University of Twente

Abstract

This report describes the design and implementation of Ahoy, a decentralized service discovery protocol based on attenuated Bloom filters. The protocol is specifically designed with mobile ad-hoc networks (MANETs) in mind: it allows the discovery of services located multiple hops away, generates little network traffic, uses no central directories or other infrastructure, and detects and handles changes in network topology. Attenuated Bloom filters are used to efficiently disseminate information that allows queries to be propagated to only those nodes likely to have knowledge of the requested service.

The implementation of Ahoy demonstrates the feasibility of a service discovery protocol based on attenuated Bloom filters. It also provides a platform for further experimentation. Measurements show Ahoy to have a very low impact on the amount of network traffic.

This report describes the Ahoy protocol and its implementation. It discusses a number of design decisions and the choices that were taken, as well as possible alternatives, and presents suggestions for further work.

Contents

1 Introduction

Ahoy is a service discovery protocol: it allows computers to discover and locate services that are being offered on the network. Nodes implementing the protocol send queries containing service identifiers, and receive zero or more responses, each response containing an address at which an instance of the requested service can be found. For example, a node wishing to send out an email message might send a query for the service `smtp` (SMTP is the mail transport protocol), and receive two responses: one for the IPv6 [1] address `fec0::1` on port 25, and one for the IPv6 address `fec0::53` on port 25. The node can then choose either of these addresses to start an SMTP session with and transmit its email message. If the chosen address does not work, the other address can be tried.

Sending an email message requires only one working SMTP service to be found. Other scenarios may require multiple service instances. For example, a node wishing to list all files being offered for download over FTP (the Internet file transfer protocol) could send out a query for the service `ftp`, connect to all addresses it receives responses for, and start an FTP session with each address to obtain the list of files being offered.

Ahoy differs from many other service discovery protocols in that it was designed specifically for mobile ad-hoc networks (MANETs). Traditional (non-MANET), computer networks consist of a number of stationary computers, connected by network cables. All computers on the same network can communicate with one another directly (at least, as far as the Internet protocol is concerned). Routing packets to and from other networks is done by one or more designated routers. The other computers on the network are leaf nodes: they send and receive packets, but they do not forward them to other nodes. Figure 1 depicts an infrastructure network.

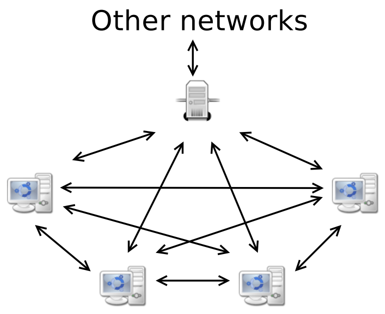


Figure 1: A traditional infrastructure network.

Mobile ad-hoc networks are a radical departure from the traditional model. Instead of having one or a few routers and many leaf nodes, and relying on their static structure and configuration to deliver network packets, prototypical MANETs consist of battery-powered mobile nodes communicating using radio. There is no static structure or configuration at all; nodes automatically configure themselves by cooperating with neighboring nodes, and the configuration will have to be changed in response to changes in the availability and reachability of neighboring nodes. Rather than being able to reach other nodes on the network directly, nodes in a MANET will often have to communicate through other nodes, meaning that intermediate nodes must be willing to forward packets. Figure 2 depicts a MANET.

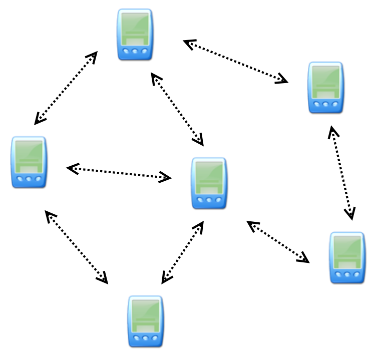


Figure 2: A mobile ad-hoc network (MANET).

MANETs pose a number of challenges for service discovery. First of all, not all nodes may be able to reach one another directly, so provisions must be made to allow for the discovery of services located multiple hops away. Secondly, it is highly desirable to minimize the generated network traffic, because nodes will often operate on limited battery power and on a shared medium with limited capacity. Thirdly, nodes may have limited memory and processing power available. This imposes limitations on the amount of information nodes can be required to maintain, and the amount of processing they may be required to perform. Fourthly, the mobility of nodes causes reachability and availability to change; a service discovery protocol for MANETs must deal with this effectively. This report describes the design and implementation of Ahoy, a decentralized service discovery protocol based on attenuated Bloom filters.

Bloom filters offer a compact encoding of service presence information with some probability of false positives. When combined with attenuation, this allows queries for services to be propagated only to those nodes likely to have information about these services. By using attenuated Bloom filters, Ahoy requires neither full service location information nor service queries to be distributed to all nodes. Each node maintains information about its own services, and the Bloom filters of nodes up to a configurable distance (measured in hops) away. This is why Ahoy is called a proximity-based discovery protocol. In Ahoy, all nodes are equal. There is no reliance on centralized directories, which avoids the associated problems in the face of mobility. Thanks to the compactness of Bloom filters, the size of the state to be maintained, and the messages to be exchanged, is relatively small. All these qualities make Ahoy a suitable service discovery for use in MANETs.

The organization of the rest of this document is as follows: Section 2 describes the goals of the project. Section 3 gives a high level overview of the Ahoy protocol, including an explanation of attenuated Bloom filters. Section 4 presents related work and highlights how Ahoy differs from it. A detailed specification of the protocol is given in Section 5. Section 6 discusses the requirements Ahoy imposes on the network. The implementation of Ahoy, the issues that were encountered, and the way they were addressed are discussed in Section 7. Section 8 describes the tests that were performed and the results that were obtained. Section 9 describes how Ahoy was integrated with the JXTA peer to peer framework. Conclusions are presented in Section 10. Section 11 provides suggestions for further work.

2 Goals

In [2], a context-discovery protocol based on attenuated Bloom filters is proposed. Based on mathematical modeling, the protocol is compared to a protocol performing full context advertisements and a protocol performing no such advertisements. The protocol based on Bloom filters is shown to have very low network overhead in many cases. In [3], the protocol is implemented in the OPNET network simulator [4]. Simulation results confirm the efficiency of the protocol for service discovery.

In the present project, the protocol introduced in [2] and [3] is further refined and a prototype implementation is created. The prototype implementation serves as a proof of concept and as a base for further experimentation.

Besides delivering a prototype implementation, this project answers the following research questions:

- Can the protocol be implemented in practice?
- What restrictions, if any, does implementability impose on the design of the protocol?
- What additional design decisions were necessary to turn the original protocol into a complete, implementable specification?
- What design choices have been made, and how would different choices affect the performance of the protocol?
- How usable is the protocol in practice?

To determine the feasibility of implementing the protocol, a prototype implementation has been made. This required choices to be made about the message formats and operation of the protocol. These choices have been identified and documented, and the decisions that were taken are explained in this report.

To demonstrate the practical usability of the protocol, a proof of concept integration with the JXTA peer to peer framework [5] has been performed. In the combined system, JXTA uses Ahoy to discover the addresses of the recipients of propagate messages. This demonstrates the successful use of Ahoy with existing applications.

Besides this report, the project delivers a prototype implementation of the Ahoy protocol, which can be built, installed and used on Unix-like systems (such as GNU/Linux). The implementation consists of Ruby source code for the Ahoy daemon that implements the protocol, a Ruby module for developing clients that use the daemon to publish, revoke, and discover services, and a number of clients using the library. Java classes for developing Ahoy clients are also provided, as well as a patch against JXTA 2.4 that modifies JXTA to use Ahoy (see Section 9).

3 Protocol Overview

This section gives a high-level overview of how the Ahoy protocol works. It is divided into a number of subsections. Section 3.1 introduces Bloom filters, which Ahoy uses to encode service availability information. Section 3.2 introduces attenuated Bloom filters, which Ahoy uses to aggregate and distribute information about services located multiple hops away. Section 3.3 describes the main message types of Ahoy and illustrates the way services are announced and discovered.

3.1 Bloom Filters

In Ahoy, service availability information is encoded in Bloom filters [6]. A Bloom filter is represented by a bit array and a number of hash functions returning indices in the array. Initially, all bits are set to 0. Bloom filters support two operations: adding an item, and testing for item presence. To add an item, it is run through each of the hash functions, and, for each result, the bit at that position is set to 1. To test if an item is present, it is run through the same set of hash functions, and the bits at each of the resulting indices are tested. If one or more of the tested bits are 0, the item is not present. If all tested bits are 1, the item is probably present, although there is a small chance of a false positive. In Ahoy, the items added to Bloom filters are strings (encoded using UTF-8 [7]) representing service names.

Figure 3 depicts an empty Bloom filter. Since the filter is empty, all bits are set to zero.

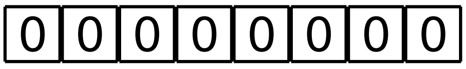


Figure 3: An empty Bloom filter.

Suppose we are using two hash functions, and running the service name `ftp` through the hash functions yields the values 1 and 5, respectively. Then, to add `ftp` to the Bloom filter, we set the corresponding bits to 1. The result is shown in figure 4.

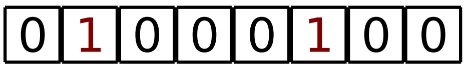


Figure 4: A Bloom filter containing `ftp`.

Testing if the filter contains `ftp` requires us to check whether bits 1 and 5 are set to 1. Since we just set these bits to 1, the test will tell us that `ftp` is present in the filter. If `http` hashes to the values 2 and 6, checking if `http` is present will tell us it is not present, because both bits are 0. If `smtp` hashes to values 2 and 5, testing whether it is present in the filter will also tell us it is not present: bit 5 is set, but bit 2 is not. However, if we add `http` by setting bits 2 and 6 to 1 (obtaining the filter depicted in figure 5), both `http` (bits 2 and 6) and `smtp` (bits 2 and 5) seem to be present in the filter, even though we have never added `smtp`. This means testing if the filter contains `smtp` yields a false positive; the test tells us it is present, although it is not.

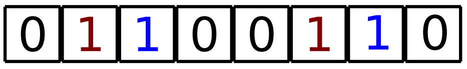


Figure 5: A Bloom filter containing `ftp` and `http`.

The probability that false positives will occur is a function of the number of services in a filter, the number of bits in the filter (called its *width*), and the number of hash functions used. In these examples, 8-bit Bloom filters have been used for the sake of simplicity. In practice, larger Bloom filters would be used to make false positives less likely. Also, the bit indices used in these examples are not the actual bit indices that Ahoy would generate for the given service names. The actual algorithm for computing bit indices is given in Section 5.2.

Bloom filters allow service availability to be represented compactly, leading to limited network traffic and memory requirements inside nodes.

3.2 Attenuated Bloom Filters

Ahoy nodes exchange information about services offered up to a configurable distance (measured in hops) away from themselves. This information is represented in attenuated Bloom filters. An attenuated Bloom filter consists of multiple layers of Bloom filters, where the first layer contains a Bloom filter representing services available at the sending node, the second layer consists of services available one hop away from the sending node, and so on, up to a configured number of hops away (this number of hops is called the **depth**). Attenuated Bloom filters allow the discovery of services located multiple hops away, and provide information about which nodes may have more information about a service. Figure 6 shows how information from incoming attenuated Bloom filters is aggregated and represented in an outgoing attenuated Bloom filter.

The bottom node receives attenuated Bloom filters from the left and right nodes. It then sends out an attenuated Bloom filter containing a layer corresponding to its own services (top row), one layer that is the bitwise OR of the first layer received from the left node and the first layer from the right node (middle row), and one layer containing the bitwise OR of the second layers received from the left and right nodes (bottom row). The third layers received from the left and right nodes are not represented in the attenuated Bloom filter sent out from the bottom node, because they represent services too many hops away.

3.3 Message Distribution

Nodes exchange service information by sending announcement messages to their neighbors. When a node wishes to use a service, it sends a query message

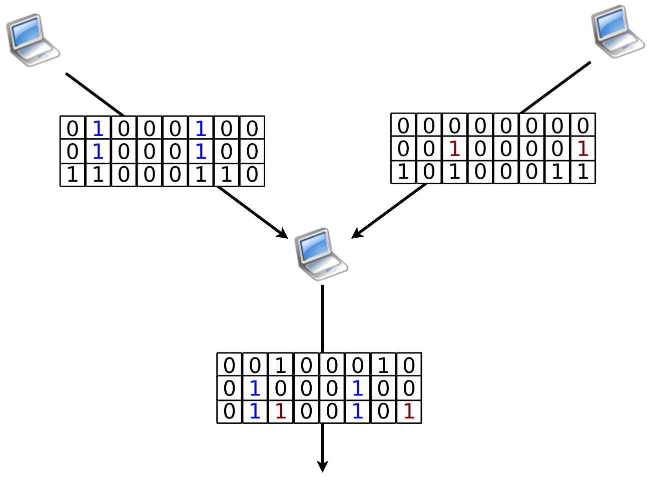


Figure 6: Aggregation of service information.

containing the name of the service it seeks information about. Nodes receiving a query for a service they provide will respond by sending a response message containing the service's address to the node the query originated from. This section illustrates the distribution of announcements, queries, and responses.

3.3.1 Announcement Distribution

Since attenuated Bloom filters only encode information about services up to a maximum number of hops away, not all services can necessarily be discovered by all nodes in the network. Figure 7 shows the distribution of a particular service in a network with a regular grid structure.

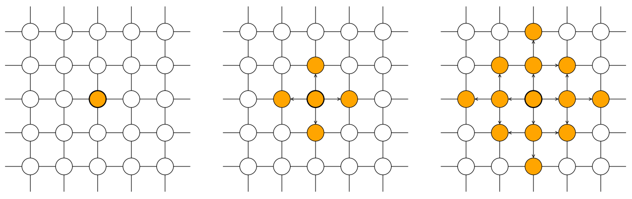


Figure 7: Distribution of service information in a network with a regular grid structure.

In this example, the **depth** is 2. The node in the center sends out an announcement containing information about a service it offers. This information is received by the sending node's neighbors, who include it in announcements they subsequently send out. The information is then received by the neighbors of these nodes, but not sent on further, because the maximum depth has been reached. As a result, all colored nodes will be able to discover the service, whereas the white nodes will not.

3.3.2 Query Distribution

Bloom filters encode only availability information. Also, they do not encode that information precisely, but with a small probability of false positives. To discover services with certainty and to discover the address at which these services can be contacted, nodes send query messages. This is the only time queries are sent; in particular, queries are not sent periodically. Besides the name of the requested service, queries carry a query id (used to avoid loops) a time-to-live (limiting the distribution of the query to **depth** hops), and the address of the node the query originated from. Queries are only sent to nodes whose Bloom filters contain the requested service, because other nodes certainly would not be able to provide useful responses. The distribution of queries is illustrated in figure 8.

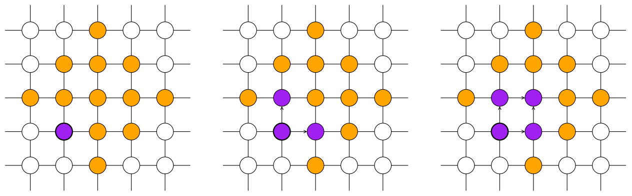


Figure 8: Distribution of a query.

Again, the **depth** is 2. The query is only distributed to nodes that have sent Bloom filters containing the right bits for the requested service. These nodes are colored (bright) orange in this image. Also, query propagation is limited to those nodes who have information about the service within the time-to-live of the query. Thus, in effect, the a node forwards the query only to those nodes that are closer to the query than itself. All nodes receiving the query are colored (medium) purple in figure 8.

3.3.3 Response Distribution

Finally, when a node receives a query for a service that it provides, it sends a response containing the IP address and port number of the requested service. Responses are addressed directly to the node the query originated from. The routing layer of the network is used to deliver responses to nodes located multiple hops away. Figure 9 illustrates the distribution of responses.

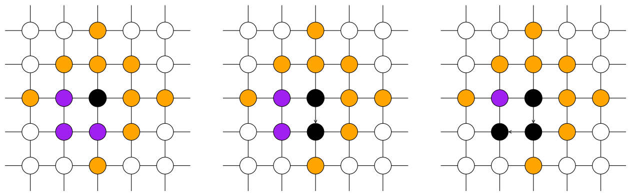


Figure 9: Distribution of a response.

As before, nodes that received the announcement for the service are colored (bright) orange, and nodes that received the query for the service are colored (medium) purple. The center node offers the service requested in the query. It sends a response to the node the query originated from. Although the response message has to pass through other nodes (colored black) to reach its destination, these nodes do not do any processing on the message besides forwarding it towards its destination. In particular, the message is not processed by Ahoy on these intermediate nodes.

3.4 Summary

It should be clear that Bloom filters play a crucial role in Ahoy. Where fully pro-active service discovery protocols broadcast full service information, Ahoy broadcasts only Bloom filters, which are often smaller. Where fully reactive protocols require queries to be distributed to all nodes, Ahoy's attenuated Bloom filters allow queries to be guided to the nodes that have the requested information, or even completely eliminate queries for non-existent services. Thus, it can be expected that Ahoy compares favorably (in terms of network traffic generation) to both pro-active and reactive protocols. This is also demonstrated in [2].

The use of Bloom filters also represents a trade-off. When a false positive occurs, a query will be sent for a service that does not exist. This wastes network traffic. One way to reduce the probability that false positives will occur is to use larger Bloom filters. However, this increases the size of announcements, and thus, network traffic. Clearly, there is a certain optimum at which network traffic is minimized. However, this optimum depends on a large number of factors (number of services, Bloom filter width, attenuation depth, number of hash functions, announcement rate, query rate), and is difficult if not impossible to determine precisely in mobile ad-hoc networks.

4 Related Work

Although Ahoy is not based on any other protocol, it shares certain features with other protocols. This section discusses some of these protocols, by mentioning their most important characteristics and how they differ from Ahoy.

4.1 ZeroConf

ZeroConf [8] defines a suite of simple protocols for zero configuration networking. One of these protocols is DNS-SD [9], which allows service discovery, and is implemented as part of Apple's Bonjour[10] framework (used extensively by various applications for Mac OS X), as well as supported by the KDE[11] and GNOME[12] desktop environments. The zero configuration service discovery mechanism is built on top of multicast DNS[13]. This is a decentralized and lightweight solution. It also offers a number of functions that Ahoy does not provide, such as enumerating all services in the network. However, queries must reach all nodes on the network, which does not scale well to larger networks.

4.2 Scalable Service Discovery for MANET

In Scalable Service Discovery for MANET [14], a service discovery protocol based on Bloom filters and directory agents is proposed. Service presence information only has to be exchanged among directory nodes, and queries only have to be made from the client node to the nearest directory node, making this protocol particularly efficient in terms of network traffic. Furthermore, Bloom filters are used for the exchange of service availability information among directory agents. Directories are automatically set up using an election algorithm.

This protocol differs from Ahoy in a number of important aspects. Most importantly, Scalable Service Discovery allows services in the whole network to be discovered, whereas Ahoy limits service discovery to a preconfigured number of hops. Secondly, where Ahoy services are described by simple strings, Scalable Service Discovery describes services using DAML [15] and allows queries to refer to attributes and values. Thus, Scalable Service Discovery is more powerful than Ahoy, but also much more complex. Also, Scalable Service Discovery relies on directories, which could be problematic in MANETs where connectivity changes frequently. By contrast, Ahoy is fully decentralized.

4.3 GSD

GSD [16] is a service discovery protocol designed specifically for MANETs. It is based on "peer-to-peer caching of service advertisements and group-based intelligent forwarding of service requests". Like Scalable Service Discovery, GSD uses DAML to describe services, thus allowing rich queries to me made. Furthermore, it uses the class/subClass hierarchy described by DAML to selectively forward queries (i.e. queries are not flooded to all nodes). In this sense, DAML descriptions are used in a similar fashion to how Bloom filters are used in Ahoy and Scalable Service Discovery.

Although GSD does not use Bloom filters, it does share a number of traits with Ahoy. Like Ahoy, GSD limits service advertisements to a preconfigured number of hops. Also, in GSD as well as in Ahoy, all nodes are equal, and all cache information from service announcements they receive. Finally, both GSD and Ahoy resolve service names by means of query messages which are limited to a certain number of hops and are selectively forwarded to neighbors likely to have information about the requested service.

Unlike Ahoy nodes, which announce all services they know of in a single attenuated Bloom filter, GSD nodes advertise each service separately, with a large amount of information contained in the advertisements. It can thus be expected that announcing services is much more expensive in GSD than it is in Ahoy.

5 Protocol Specification

A high-level overview of the Ahoy protocol has been given in Section 3. This section presents a detailed description of the protocol. Section 5.1 describes the various parameters governing the operation of Ahoy. Section 5.2 details how announcements are computed, an operation which is referred to various times in the rest of the protocol specification. Section 5.3 discusses the messages used by Ahoy. Section 5.4 describes the protocol in terms of events and responses.

5.1 Configuration Parameters

On startup, the daemon reads a configuration file to determine various parameters. Currently, the following configuration options are recognized:

announcement-min-time The minimum time that should elapse between subsequent announcements (used to prevent announcement storms, where many announcements are sent out in rapid succession). The value is in seconds, and defaults to 5.

broadcast-queries This parameter controls whether query messages are propagated using a one-hop broadcast, or by unicasting them to each of the neighbors they are intended for. It defaults to `false`, which causes unicast propagation to be used. Any other value causes broadcast propagation to be used.

depth The depth of the attenuated Bloom filters; that is, how many hops will be represented in each announcement. This value should be the same for all nodes on the network. If not specified, it defaults to 4.

hash-functions The number of hash functions to use with the Bloom filters. Hashes are computed as follows:

```
$hash \leftarrow n$
for each byte $b$ in $service\_name$:
    $hash \leftarrow 33 * (hash \text{ xor } b))$
return $(hash \bmod width)$
```

where n is the number of the hash function (starting at 0), $width$ is the value of the configuration parameter **width**. To simplify computation on 32-bit systems, all arithmetic is performed modulo 2^{32}.

The default number of hash functions is 3.

ip-address The IPv6 address to which the daemon will bind. This address is also used as the source address for queries. It is a required parameter; without it, the daemon will not operate correctly.

local-address The local address on which the daemon is to listen for user connections. If not specified, it defaults to `/tmp/ahoy/socket`.

port The port number on which to listen. This parameter must be the same for all nodes on the network, or the protocol will not work. The default value is 5000.

keep-alive-time How much time should elapse between subsequent keep-alive messages. The value is in seconds, and causes a keep-alive message to be sent that many seconds after the latest announcement or keep-alive message, with some random variation as specified by **keep-alive-jitter**. The default value is 15.

keep-alive-jitter The value of **keep-alive-time** is not followed strictly; rather, each keep-alive message is randomly scheduled within a certain margin around the designated time. This is done to prevent the situation where two or more nodes have coinciding schedules for keep-alive messages, with each node sending each keep-alive at exactly the same time as the other node, causing them to collide. The value of **keep-alive-jitter** is a percentage. The delay for the next keep-alive message is computed as **keep-alive-time** + (**keep-alive-time** $*$ ($rand - 0.5$) $*$ **keep_alive_jitter** / 100), where $rand$ is a random value between 0 and 1. The default value of **keep-alive-jitter** is 25, meaning that the jitter is 25% of the value of **keep-alive-time**.

local-service-timeout Users are required to re-announce their services periodically. This is done so that these services can be removed in the event a user program crashes or otherwise fails to revoke service announcements for services no longer offered. **local-service-timeout** specifies after how many seconds local services should be discarded. It defaults to 300.

query-timeout The number of seconds each query will remain active. Responses to queries will not be processed after **query-timeout** seconds. The default value of this parameter is 10.

query-cache-timeout The interval (in seconds) between subsequent clean-ups of the query cache. Every **query-cache-timeout** seconds, entries in the query cache that are older that **query-cache-timeout** seconds will be removed, meaning their query ids can be reused.

width The width of the Bloom filters; that is, the number of bits in the bit array for each Bloom filter. This parameter must be the same for all nodes on the network, or the protocol will not work. If not specified, it defaults to 128.

5.2 Computing Announcements

One of the most complex tasks in Ahoy is the computation of announcement messages. An announcement message consists of an attenuated Bloom filter containing **depth** layers. The first layer represents services advertised by the node sending the announcement. The second layer represents services advertised by the neighbors of that node. The third layer represents services advertised by the neighbors of these neighbors, and so on.

To compute an announcement, a node needs three things: the names of the services it advertises, the announcements received from neighboring nodes, and the parameters for the computation of the announcement (**width**, **depth**, and the number of **hash-functions**).

5.2.1 The First Layer

The first layer of the announcement represents services advertised by the node computing the announcement. The node creates an empty Bloom filter, consisting of **width** bits (initially set to 0) and **hash-functions** hash functions. The hash functions are specified by the following algorithm:

```
$hash \leftarrow n$
for each byte $b$ in $service\_name$:
    $hash \leftarrow 33 * (hash \text{ xor } b))$
return $(hash \text{ mod } width)$
```

where n is 0 for the first hash function, 1 for the second hash function and so on.

After the empty Bloom filter has been created, the node adds all services it advertises to the filter, as follows. For each of the hash functions associated with the Bloom filter, the node calls that function, passing the string representing the service (i.e. the service name) as an argument. This yields an index in the bit array of the Bloom filter (i.e. a value from 0 up to, but not including, **width**). The bit at that index is set to 1. This is done for all hash functions and all service names.

For example, suppose **width** is 8, **hash-functions** is 2, and the services to be advertised are `ftp` and `http`. First, an empty Bloom filter containing 8 bits is created. This filter is displayed in figure 10.

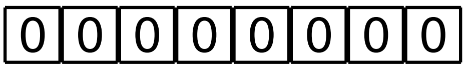


Figure 10: An empty Bloom filter of width 8.

Then, `ftp` is added to the filter. Represented as a string in UTF-8 (the character encoding used by Ahoy), this is byte sequence $\{102, 116, 112\}$. Running this through the first hash function ($n = 0$), we get 2. So we set bit 2 (the third bit) of the Bloom filter to 1. Running the same byte sequence through the second hash function ($n = 1$), we get 3. So we set bit 3 (the fourth bit) of the Bloom filter to 1, as well. The Bloom filter, after adding `ftp` to it, is shown in figure 11.

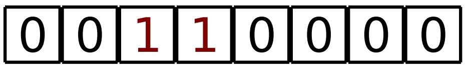


Figure 11: A Bloom filter of width 8, containing the service `ftp`.

After `ftp` has been added, we add `http` to the filter. The procedure is the same, except that the byte sequence for the string `http` is $\{104, 116, 116, 112\}$. This yields the value 0 when run through the first hash function, and the value 1 when run through the second hash function. Thus, we set bits 0 and 1 of the Bloom filter to 1, yielding the filter shown in figure 12.

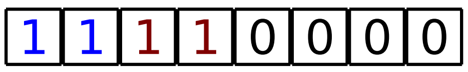


Figure 12: A Bloom filter of width 8, containing the services `ftp` and `http`.

5.2.2 The Other Layers

The other layers of the announcement are computed from the announcements received from neighboring nodes. The second layer of the announcement being computed is created by taking the bitwise OR of the first layers of all announcements received from neighbors. The third layer of the new announcement is computed from the second layers of all received announcements, and so on. The last layers of the received announcements are not used in the computation, because they will not be represented in the new announcement being computed.

The combining of attenuated Bloom filters received from other nodes into a new attenuated Bloom filter is graphically depicted in figure 13. The attenuated Bloom filter sent by the bottom node consists of three layers, where the first layer is computed from the names of services the bottom node itself advertises, the second layer is the bitwise OR of the first layers received from the top nodes, and the third layer is the bitwise OR of the second layers received from the top nodes. The third layers received from the top nodes are saved by the bottom node, but not included in the attenuated Bloom filter it sends out.

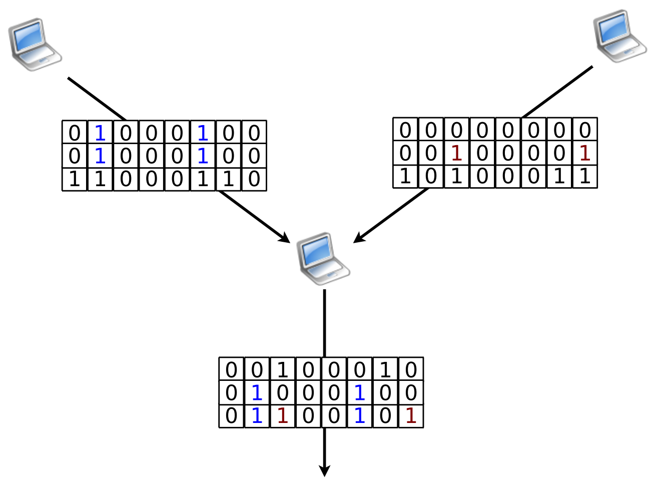


Figure 13: Aggregation of service information.

A complete description of all the fields of an announcement message is given in Section 5.3.2.

5.3 Message Formats

Message formats are specified using a table listing the fields the message contains, the offset of each field within the message, the size of the field, and a concise description of the content of the field. Offsets and sizes are in octets. Numeric fields are unsigned integers in network byte order (big-endian), and textual fields are in UTF-8. Each table is followed by a more detailed description of each of the fields.

Several messages (announcements, keep-alive messages, queries and responses) contain id fields. Each of these id fields is 4 octets wide. The id field in keep-alive messages must match that of the latest announcement sent out by a node, and the id field in a response must match that of the query the response belongs to. Id fields for announcements and queries can be generated by any algorithm, as long as care is taken that subsequent announcements from the same node have different ids, and concurrently active queries from the same node have different ids. Queries or announcements coming from different nodes do not have to have distinct ids, as they can be distinguished by sender address.

5.3.1 Addresses

Some messages (queries and responses) include addresses. In principle, any type of address could be supported, but the only address type currently defined is the IPv6 address. Because services that use IPv6 typically also need a port number, a field for the port number is included in the structure. The representation of an IPv6 address is shown in table 1.

Offset	**Size**	**Description**
0	1	size
1	1	type
2	2	port
4	16	address

Table 1: Wire format of an IPv6 address.

size The size of the address in bytes, including the size and type fields. For IPv6 addresses, this is always 20.

type The address type. For IPv6 addresses, the type is 1.

port The port number as a 16-bit unsigned integer in network byte order.

address The 128 bits of the IPv6 address proper.

5.3.2 Announcements

The format of announcement messages is shown in table 2.

type The type for announcements is 1.

generation-id A 32-bit unsigned integer which is updated each time a new announcement is sent out. This is used to detect missed announcements (see Section 5.3.5).

depth The number of layers of Bloom filters in this announcement.

Offset	Size	Description
0	1	type
1	4	generation-id
5	1	depth
6	2	width
8	variable	filters

Table 2: Wire format of an announcement message.

width The width of the Bloom filters, in bits.

filters The bits of the filters. Filter width should be a multiple of 8 bits (one octet) for maximum efficiency. However, if filter width is not a multiple of 8 bits, the remaining bits should be set to 0. Filters are sent in order from least to greatest depth (distance from the node sending the announcement). The ordering of the bits in each filter is from the least significant bit in the first octet to the most significant bit in the last octet; i.e. bit 0 of the filter is the bit with weight 1 in the first byte, bit 7 of the filter is the bit with weight 128 in the first byte, bit 8 of the filter is the bit with weight 1 in the second byte, etc.

An announcement contains multiple layers of Bloom filters, each layer corresponding to a certain hop count from the node sending the announcement. The **depth** field indicates the number of layers in the announcement, and should be the same for all announcements sent by all nodes. Nodes can discover services up to **depth** hops away from themselves. The **width** field indicates the number of bits in the Bloom filters, and must be the same for all nodes; if it is not, service discovery will not work.

Announcements are broadcast to all neighbors whenever the contents of the Bloom filters maintained by a node change. This can be in response to a service being announced or revoked on the node itself, but also in response to an announcement just received from a neighbor, or the expiry of information from a node that has not been heard from for a while. To prevent flooding the network with announcements, there is a minimum delay between announcements.

Announcements are also sent in response to update requests (see Section 5.3.6). In this case, the announcement is not broadcast to all neighbors, but sent only to the node requesting the update. There is no rate limiting for this sending of announcements.

The way announcements are computed is explained in Section 5.2.

5.3.3 Queries

The format of query messages is shown in table 3.

type The type for queries is 2.

query-id The query-id for this query as a 32-bit unsigned integer identifying the query.

Offset	**Size**	**Description**
0	1	type
1	4	query-id
5	1	time-to-live
6	2	name-length
8	variable	name
variable	variable	sender-address

Table 3: Wire format of a query message.

time-to-live The maximum number of hops this query may still be propagated. It is decremented each time the query is propagated by a node. Queries should only be sent and received if the time-to-live is at least 1.

name-length The number of bytes in the service name.

name The name of the service being queried for.

sender-address The address of the originator of this query.

A query is sent whenever a node wishes to locate service instances. The query is propagated to all neighbors that may be able to provide information about the service. This is determined by testing the Bloom filters received from neighbors for the presence of the service name. This process is repeated by neighbors receiving the query, as long as the **time-to-live** permits service propagation. Thus, the query is propagated to all nodes within the number of hops specified by the originator that have information about the requested service.

Each node receiving the query checks if it has the service being queried for. If so, it sends a response (see Section 5.3.4) to the originator of the query. The query is propagated (**time-to-live** permitting) no matter if a match is found in the current node or not. Thus, a query returns all reachable service addresses, not just the nearest ones.

5.3.4 Responses

The format of response messages is shown in table 4.

Offset	**Size**	**Description**
0	1	type
1	4	query-id
5	variable	address

Table 4: Wire format of a response message.

type The type for response messages is 3.

query-id The query-id of the query this response relates to.

address The address of the service.

A node will send a response when it receives a query for a service it provides. The response will contain the **query-id** of the query and the **address** of the service, and it will be sent directly to the **address** specified in the query message (see Section 5.3.3).

5.3.5 Keep-Alive Messages

The format of keep-alive messages is shown in table 5.

Offset	Size	Description
0	1	type
1	4	generation-id

Table 5: Wire format of a keep-alive message.

type The type for keep-alive messages is 4.

generation-id The generation-id of the latest announcement sent out by this node.

Keep-alive messages are periodically sent to all neighbors. Their purpose is twofold. First of all, keep-alive messages notify neighbors that the sending node is still present and still participating in Ahoy. This allows nodes to discard information from neighbors that have not been heard from for a while. Secondly, keep-alive message inform neighbors of the **generation-id** of the latest announcement sent out by the node sending the keep-alive. If a node missed an announcement (or was not in range when the announcement was sent), it will know its information is out of date, and it will send an update request (see Section 5.3.6).

Two parameters are important to keep-alive messages: the delay between the sending of consecutive keep-alives, and the time to wait before discarding information from nodes from which no keep-alive message has been received. The keep-alive delay is reset when a keep-alive message or an announcement is sent, and the timeout for a neighbor is reset whenever a keep-alive message or an announcement is received from that neighbor.

To prevent keep-alive messages from colliding, each keep-alive message is scheduled a random amount of time (between configurable bounds) earlier or later than the strict value of the keep-alive delay.

All nodes should have their timeouts set sufficiently large that information from nodes is not discarded needlessly often.

5.3.6 Update Requests

The format of update requests is displayed in table 6.

type The type for update requests is 5.

A node will send an update request when it receives a keep-alive message from a neighbor, but the **generation-id** in the update request does not match that of the latest announcement received from that neighbor (or no announcement

Offset	Size	Description
0	1	type

Table 6: Wire format of an update request message.

from that neighbor is recorded at the time the keep-alive message is received). The update request is sent directly to the sender of the keep-alive message. In response, the neighbor is expected to send its latest announcement directly to the node sending the update request (using unicast).

When a node first comes online, it will broadcast an update request to all neighbors, prompting them to send their current announcements to the joining node. If any services are being offered, the joining node will then send out an announcement itself, indicating its ability to relay queries.

5.4 Functional Description

This subsection describes Ahoy in terms of events and responses. There are four types of event:

1. Messages from other Ahoy nodes: announcements, queries, responses, keep-alives, and update requests
2. User actions: announcements, revocations and queries.
3. Timeouts: minimum announcement delay, keep-alive delay, query timeout, query cache cleanup timer, and service list cleanup timer.
4. Exceptions: the user sending a break to the program, the operating system sending a terminate signal, out of memory. etc.

Before these events are discussed, an overview of the state variables Ahoy maintains and an overview of the configuration parameters that govern Ahoy's operation are given.

5.4.1 State Variables

The Ahoy daemon maintains five state variables:

Query Cache The query cache is used for detecting (and subsequently discarding) duplicate queries. For each query that is received, it contains the query id (see 5.3.3), the source address of the query, and a timestamp. The query cache is cleaned up at regular intervals by purging old entries.

Neighbor List The neighbor list contains information about the direct neighbors of the node running the Ahoy daemon. For every neighbor, it contains the neighbor's address, a timestamp, and the latest announcement (generation id and Bloom filters) received from that neighbor.

Local Services The local services list contains the services announced by users. For each service, it contains the name, the address, and a timestamp. The local services list is cleaned regularly by removing old entries.

Latest Announcement The Ahoy daemon keeps the information in the latest announcement it sent out. When the neighbor list or the local services change, a new set of Bloom filters is computed, but these only need to be broadcast if they are different from the filters in the latest announcement. Keeping a copy of the information sent out in the latest announcement allows the daemon to decide whether it needs to send out a new announcement or not.

Active Queries The daemon keeps a list of active queries, i.e. queries that have been initiated by users and that the daemon is currently awaiting responses for. For each such query, the list contains the query id and the socket on which the user program is listening for responses.

5.4.2 Startup

After the configuration parameters have been set, Ahoy opens a UDP socket on the configured **ip-address** and **port**. This socket is used for sending and receiving Ahoy messages. After the socket has been opened, Ahoy broadcasts an update request message, after which it enters the idle state (see Section 5.4.3). The update request causes neighboring Ahoy instances to send their latest announcements to the new instance, allowing it to populate its neighbor list.

5.4.3 Idle State

In the idle state, the daemon waits for events to occur. The events are listed below, along with the number of the section that discusses the handling of the event.

- An announcement is received from another Ahoy node (Section 5.4.4).
- A query is received from another Ahoy node (Section 5.4.5).
- A response is received from another Ahoy node (Section 5.4.6).
- A keep-alive message is received from another Ahoy node (Section 5.4.7).
- An update-request message is received from another Ahoy node (Section 5.4.8).
- An announcement is received from a user (Section 5.4.9).
- A revocation is received from a user (Section 5.4.10).
- A query is received from a user (Section 5.4.11).
- The keep-alive timer expires (Section 5.4.12).
- The announcement timer expires (Section 5.4.13).
- The query cache cleanup timer expires (Section 5.4.14).
- The service list cleanup timer expires (Section 5.4.15).
- A query timer expires (Section 5.4.16).

- A signal is received from the operating system, causing the daemon to clean up and exit (Section 5.4.17).

These events are handled as described below. If a message is received that does not fit one of the above categories, a warning is emitted and the message is discarded.

5.4.4 Ahoy Announcements

When an announcement is received from another Ahoy node on the network, the following steps are taken:

- The current time, the content of the announcement, and the address of the neighbor from which it was received are recorded in the Neighbor List, overwriting any previous entry for the same address.
- A new announcement is computed based on the new information (see Section 5.2 for a description of how announcements are computed).
- If the new announcement differs from the Latest Announcement, it is broadcast to all neighbors and becomes the new Latest Announcement.

5.4.5 Ahoy Queries

When a query is received from the network, the following actions are performed:

- The query's id and source address are looked up in the Query Cache. If a match is found (meaning the query has been seen before), only the timestamp in the cache is updated and no further processing is performed (the query is discarded).
- The current time, the query's id, and the query's source address are inserted into the Query Cache.
- The query's service name is looked up in the Local Services. For any matching services, a response message (see Section 5.3.4) is sent to the query's source address.
- If the query's time-to-live is greater than 1, it is propagated. Propagation works as follows:
  - A new query message is created, with its id, service name, and source address equal to those of the received query, and a time-to-live of one lower than the received value.
  - The Neighbor List is cleaned up by removing neighbors from which no announcement or keep-alive message has been received in the last **neighbor-timeout** seconds. If any neighbors were removed, a new announcement is computed based on the remaining service information. If this announcement differs from the Latest Announcement, it is broadcast to neighboring nodes, becomes the new Latest Announcement, and the keep-alive timer is reset.

  - After the Neighbor List has been cleaned, a look up is performed against it, returning all neighbors who have matches for the service name in their Bloom filters, within a number of hops less than or equal to the time-to-live of the new query.
  - The new query is sent to all these neighbors, if any. If **broadcast-queries** is false, the query is unicast to each individual address. Otherwise, a single broadcast message is sent if there are any knowledgeable neighbors. No query message is sent if the look up in the previous item did not return any matches.

5.4.6 Ahoy Responses

When a response is received from the network, it is processed as follows:

- The id is looked up in the Active Queries. If no match is found, no further processing is done.
- If a match is found, the address contained in the response is sent to the user program waiting for it.

5.4.7 Keep-Alive Messages

When an Ahoy node receives a keep-alive message from another Ahoy node, it takes the following actions:

- The generation-id is extracted from the keep-alive message.
- The node the message was received from is looked up in the Neighbor List.
- If the neighbor is found in the list, and the generation id recorded in the entry in the Neighbor List matches that of the message, the timestamp for the entry is updated.
- If the neighbor is not found in the list, or if the recorded generation id does not match the one contained in the message, an update request is sent to the neighbor.

5.4.8 Update-Request Messages

Upon receipt of an update-request message, an Ahoy node sends its latest announcement to the node that the update request was received from.

5.4.9 User Announcements

Users can announce services on Ahoy by specifying the service name, IP address, and port number. User announcements are handled by taking the following steps:

- Enter the current time, the service name, and the service address in the Local Services. If an entry with the same service name and address already exists, it is overwritten (put differently, the timestamp is updated).

- Recompute the Bloom filters to be announced, based on the new information.
- If the new Bloom filters differ from those sent in the Latest Announcement, a new announcement with the updated filters is sent out (this becomes the new Latest Announcement), and reset the keep-alive timer.

Note that, since local services are automatically removed after **local-service-timeout** seconds, users have to re-announce their services periodically if they wish them to persist.

5.4.10 User Revocations

Users can revoke service announcements by specifying the service name, IP address, and port number of the announcement to be revoked. This causes Ahoy to perform the following steps:

- Look up the service name and address in the Local Services.
- If no entry is found, no further processing is done.
- If an entry is found, it is removed.
- If this removes the last entry for the given service name, meaning the service is no longer offered at all, a new announcement is computed. If the new announcement differs from the Latest Announcement, it becomes the new Latest Announcement, it is distributed to all neighbors, and the keep-alive timer is reset.

5.4.11 User Queries

Users may discover service instances by performing queries on the service name. Queries are performed as follows:

- Generate an id for the query.
- Enter the query id into the Active Queries, along with information about the user program that responses are to be sent to.
- Generate a query message containing the id, the service name, the daemon's IP address, and a time-to-live field with a value equal to **depth**.
- Search the Neighbor List for neighbors that have matches for the service name in their latest announcements.
- Send the query to any such neighbors, using unicast if **broadcast-queries** is false, using broadcast otherwise.
- Set a timeout of **query-timeout** seconds. When the timeout expires, the query information is removed from Active Queries.

5.4.12 The Keep-Alive Timer

The keep-alive timer is set after an announcement or a keep-alive message is sent, to schedule the sending of another keep-alive message. When the timer expires, a keep-alive message is broadcast to all neighbors. The keep-alive timer is then reset to **keep-alive-time** $\pm$ (**keep-alive-time** $*$ **keep-alive-jitter** / 200) (in other words: a value of **keep-alive-jitter**% around the value of **keep-alive-time**) to schedule the next keep-alive message.

As an example, if **keep-alive-time** is 15 and **keep-alive-jitter** is 25 (the default values), the keep alive timer will be set to a value between 13.125 and 16.875. The actual value is chosen at random each time the timer is set, i.e. it will be different for every subsequent keep-alive message.

Finally, when the keep-alive timer expires, the Neighbor List is also cleaned up by removing neighbors from which no announcements or keep-alive messages have been received in the last **neighbor-timeout** seconds. If any neighbors were removed, a new announcement is computed based on information available from the remaining neighbors and Local Services. If the computed announcement differs from the Latest Announcement, it becomes the new Latest Announcement, is broadcast to neighboring nodes, and the keep-alive timer is reset.

5.4.13 The Announcement Timer

The announcement timer is set whenever a new announcement (different from the Latest Announcement) has been computed, but an announcement was sent less than **announcement-min-time** seconds ago. The new announcement cannot be sent immediately, and thus is scheduled to be sent at a later time.

When the announcement timer expires, an announcement computed from the latest available information (including changes that occurred after the timer was set) is sent. The new announcement then becomes the new Latest Announcement. Also, as always when an announcement is sent, the keep-alive timer is reset (see Section 5.4.12).

5.4.14 The Query Cache Cleanup Timer

The query cache cleanup timer is used to periodically clean up the Query Cache. It is first set when Ahoy is started, and reset each time it expires, causing the Query Cache to be cleaned up every **query-cache-timeout** seconds. Cleaning up is done by discarding from the cache any entries that are older than **query-cache-timeout** seconds.

5.4.15 The Service List Cleanup Timer

The service list cleanup timer is used to clean up the Local Services list. It is set to **local-service-timeout** / 10 seconds when Ahoy starts, as well as each time it expires. When it expires, the list of Local Services is scanned for entries that are older than **local-service-timeout** seconds, and any such entries are removed. If this results in any services no longer being offered, a new announcement is computed. If the announcement differs from the Latest Announcement, it is broadcast to neighbors, and the keep-alive timer is reset.

5.4.16 Query Timeouts

Whenever a query is initiated by a user, a timeout of **query-timeout** seconds is set. When the timeout expires, the query is closed. Any responses that come in after such time will be ignored.

5.4.17 Shutdown

When the Ahoy daemon exits, it deletes the `AF_LOCAL` socket it created for communicating with user programs.

5.5 Alternatives

5.5.1 Returning Responses Along the Query Path

Ahoy uses (and thus requires) routing to return response messages to querying nodes. An alternative approach would be to send the response back along the path taken by the query; that is, the responding node sends the response to the neighbor it received the query from; that neighbor then sends the response to the neighbor it received the query from, and so on, until the response reaches the query originator.

Sending responses back along the query path has the advantage that no routing functionality has to be present in the platform. On the other hand, it essentially means that this functionality is being duplicated in Ahoy. Since routing would probably be required for using the discovered addresses anyway, it was decided to simply assume the availability of routing and use it to deliver response messages.

5.5.2 Bloom Filters in Queries

The existing protocol sends service names in queries. Nodes receiving queries check the service name against the names of the services they offer, and send a response containing the query id and the address of the service if a match is found.

An alternative would be to send Bloom filters in queries. Nodes receiving queries would then send responses using the names and addresses of any services that match the received Bloom filter. This approach was initially taken in [2] and [3]. However, this approach was not taken for Ahoy, because it leads to larger response messages. It may also lead to larger queries (if the size of the included Bloom filter exceeds the size of the service name) and more response messages (if the included Bloom filter matches that of a service other than the service being queried for, although this is extremely unlikely). An advantage of sending Bloom filters in queries, instead of names, is that the filter corresponding to the name is computed only once, at the originating node, rather than at every node that receives the query.

5.5.3 Three Message Protocol

The original design for Ahoy featured only three message types: announcements, queries, and responses. Announcements would be periodically re-broadcast to allow for the detection of changes in the network topology. Nodes becoming reachable would be detected by the reception of an announcement from an address no announcement had been recorded for. Nodes becoming unreachable would be detected by not receiving an announcement from them within a specified time interval.

Having only three messages in the protocol simplifies its specification. It also simplifies the implementation somewhat. However, periodically re-broadcasting announcements generates quite a lot of unnecessary network traffic. Using keep-alive and update-request messages to discover topology changes and handle lost announcements allows for quicker detection of changes and/or conservation of resources (a keep-alive message is 5 bytes, whereas announcements can be hundreds or even thousands of bytes).

5.5.4 Alternative Service Specifications

Ahoy identifies services by arbitrary strings. Certain other service discovery protocols (e.g. Scalable Service Discovery for MANETs [14]) allow services to be specified using key-value pairs. Technically, any type of data can be encoded into Bloom filters, so Ahoy could be made to support other types of service specifications without changing the fundamental nature of the protocol. However, for this project, the focus was on the use of Bloom filters, not on the nature of service specifications. Using alternative service specifications could be an interesting topic for further research.

5.5.5 Generalized MANET Packet/Message Format

A proposal exists within the IETF for a generalized MANET message/packet format [17]. This proposal describes a message format that, when implemented by nodes, can be used to encode messages in a service-independent way, so that nodes not participating in a service could still meaningfully process messages. For example, service discovery messages encoded in such a format could be forwarded by nodes not participating in the service discovery protocol to nodes that do participate in the protocol.

After studying the proposal, it was decided that the message format and the message processing rules were complex enough that implementing them in Ahoy would add to the complexity of the prototype, without significant benefit to Ahoy. Thus, it was decided to keep the existing, simple message formats.

5.5.6 Returned Information

Ahoy nodes respond to queries by sending a response message for every matching service that is discovered. These response messages contain the IPv6 address and port number at which the service can be contacted. This has been adequate for our purposes. However, there is certainly other useful information that could be contained in response messages, such as transport or application protocol identifiers or quality of service information. In case Ahoy is used on non-IPv6 networks, response messages will have to be adapted accordingly. Also, the information that response messages contain could be made dependent on parameters present in the query. This might be an interesting topic for further research.

5.5.7 Unicast/Broadcast Query Propagation

Ahoy works most efficiently in a network that supports both unicast and broadcast messaging. Both of these mechanisms could be used for propagating queries. Broadcast propagation requires only a single message to be sent, but that message will be received by all neighbors, not just the neighbors that needed to get it. Unicast sends the message to only those neighbors that need to get the message, but may require multiple messages to be sent.

Whether broadcast propagation or unicast propagation is more efficient is likely to depend on the specifics of the network. For example, broadcast propagation is probably a good choice for wireless networks, because any message would probably be received by several neighbors, even if it were only addressed to one of them. On the other hand, unicast messages not destined for the node receiving them can be rejected early, saving power compared to broadcast messages, which have to be processed all the way up to the application layer. Unicast messages also allow different time-to-live values to be used per destination address, depending on how many hops away a match is expected to be found.

Rather than fixing the choice of broadcast or unicast, Ahoy allows nodes to use either mechanism for query propagation. The prototype allows the mechanism to be selected using the **broadcast-queries** configuration parameter.

5.5.8 Find-Any vs. Find-All

For some applications, it is desired to find all nodes advertising a certain service. For other applications, finding just a subset, or even just one, of these nodes is enough. Ahoy propagates queries as long as the time-to-live field permits it, resulting in all (or most) nodes in range that advertise a service being discovered. If the application requires only one or a few responses, network traffic could be reduced by only propagating queries if no local match were found. This would result in a subset of all nodes with matching services being found. Ahoy offers no way to request this behavior, but it could be a useful feature to add in a the future.

6 Requirements

In order to send the various types of message used by Ahoy, the network layer must support certain features. This section discusses these requirements, how the platform chosen to implement Ahoy on meets the requirements, and the alternative platforms that were considered.

6.1 Required Features

The following features are required to implement the protocol specified in Section 5:

- To broadcast announcements, a node needs a way to send a message to all its neighbors. This can be accomplished through a broadcast mechanism, or a combination of unicast messaging and an explicit list of neighbors.
- To send queries, nodes need to be able to send messages to their neighbors. This can be accomplished through broadcast messaging or unicast messaging.
- Responses to queries may need to traverse several hops before reaching the node that originally sent the query. As specified, the protocol requires routing to deliver responses. If the alternative of returning responses along the query path (described in Section 5.5.1) is implemented, the routing requirement can be dispensed with.

It is assumed that communication is bidirectional, that is, if node A can send a message to node B, then node B can also send a message to node A. The protocol can still work in the presence of unidirectional links, but some messages will be wasted, because they will be sent to neighbors that cannot be reached. Also, unless all nodes a query would need to traverse to reach a node offering a service lie within **depth** hops from that node in both directions, the query will fail to discover the service.

6.2 Platform

The initial plan was to implement Ahoy in the context of the JXTA peer-to-peer framework [5]. If the protocol could have been used as a drop-in replacement for JXTA's built-in discovery protocol, a number of existing application written for the JXTA platform would have been readily available as test cases. Also, JXTA is programming language and platform-independent, which are desirable properties for the discovery protocol, too.

Unfortunately, it turned out that JXTA did not provide all the features required to implement Ahoy. JXTA abstracts from the physical locality of nodes and provides no way to broadcast a message to all neighboring nodes or send a message without routing. Also, JXTA's service discovery protocol is specified to make use of XML [18] messages, which are rather verbose. Since small message size is a key reason for using Bloom filters, such verbosity is undesirable.

After JXTA had been discarded as an implementation environment, it was decided to implement the protocol on top of UDP [19]. UDP is lightweight, standardized, widely implemented and used, time tested, and well documented.

UDP does not provide much of the functionality that JXTA provides, and also imposes few requirements on message formats and provided functionality. This means that the flexibility to meet the requirements set out in Section 6 is there, but additional choices must be made regarding auxiliary protocols and message formats.

UDP can be layered on top of either IP version 4 (IPv4) [20], or IP version 6 (IPv6) [1]. IPv6 was chosen for the initial work, with an option to support IPv4 later. The reason for this choice was the expectation that IPv6 will be used more and more often in future research on mobile networking and future network nodes.

There are many programming languages that provide access to UDP and IPv6. For this project, Ruby [21] was chosen as an implementation language. Ruby is a simple but powerful language that is easy to learn and use allows programs to be written in a clear and concise manner. This makes programs written in Ruby easy to understand and adapt, which was the main reason for choosing Ruby.

For testing purposes, a virtual ad-hoc network of virtual computers was set up. This was done using User Mode Linux [22]. Each of the virtual machines in this network ran a minimal installation of Debian GNU/Linux [23], augmented with Ruby and the prototype being developed.

Routing between the virtual machines was performed by the olsr.org implementation [24] of the OLSR [25] routing protocol. This implementation supports IPv6 and runs on a number of platforms.

The chosen platform meets the requirements set out in Section 6.1 as follows:

- The ability to broadcast announcements to neighbors is provided by UDP datagrams sent to the IPv6 ip6-allnodes multicast address `ff02::1`. This takes the announcements to all nodes up to one hop away from the sender, exactly as intended.

- To send queries, either broadcast messages or unicast messages could be used. UDP/IPv6 provides both of these: broadcast propagation of queries can be performed by sending them to `ff02::1` (as above for announcements), whereas unicast propagation of queries can be performed by sending UDP datagrams directly to the IPv6 address of the neighbors intended to receive the queries.

- Responses can be sent using UDP datagrams, provided that a routing protocol is available to deliver the datagrams to nodes multiple hops away. OLSR was used as a routing protocol while running the tests in this project, but other routing protocols for ad-hoc networks, such as AODV [26] or DSR [27] could have been used instead.

6.3 Alternatives

6.3.1 Lower-Layer Messaging

Ahoy nodes use UDP over IPv6 for communicating with one another. An alternative would be to send messages at a lower level. For example, raw Ethernet frames could be used for communication (in this case, responses should probably be returned along the query path, as described in section 5.5.1). This would save tens of bytes of overhead per message, which is quite significant, considering that many Ahoy messages are only a few bytes long.

Unfortunately, using raw Ethernet frames has a number of drawbacks. The programming interface for sending and receiving Ethernet frames is not standardized across operating systems, which means that using raw Ethernet frames for communication would make the prototype non-portable. Secondly, many programming languages do not contain support for raw Ethernet messaging, which would complicate the creation of alternative implementations of the Ahoy protocol, if it were to use raw Ethernet frames. Finally, sending and receiving raw Ethernet frames requires `root` privileges on most Unix-like systems, which would severely increase the security risk of running an Ahoy implementation, and should generally be avoided if possible. UDP has none of these drawbacks, and was thus seen as a better choice for implementing the prototype.

6.3.2 Non-Unix Platforms

Most of the development work on the Ahoy prototype was done on Debian GNU/Linux. Many other platforms could have been chosen, for example, Microsoft Windows (which runs on the majority of personal computers), Microsoft Windows Mobile (which runs on many PDAs and Smartphones), or Java 2 Micro Edition (which runs on many mobile phones).

Debian GNU/Linux was chosen because the author is familiar with it, and, being a Unix-like system, code developed for it is easily ported to other Unix-like systems, such as Mac OS X. Also, GNU/Linux systems can be found on computers ranging from small embedded systems to supercomputers. Thus, developing the code on a GNU/Linux system produces code that runs on a large variety of systems in various classes.

By comparison, code developed for Microsoft Windows, Microsoft Windows Mobile, or Java 2 Micro Edition is not often easy to port to different platforms, and these platforms themselves have a relatively limited scope compared to the Unix-like platforms that the present implementation targets.

7 Implementation Report

This section gives a chronological account of the design and implementation of Ahoy. It also discusses the larger problems that were encountered, and the ways these problems were solved.

During the planning stages, the implementation environment was selected and a rough design for the implementation was made. The choices that were made for the implementation environment, as well as the motivations for them, are discussed in Section 6.2.

The initial design of the protocol consisted of three messages: announcements, queries, and responses, with formats and functions similar to those discussed in Section 5.3. Announcements would have to be broadcast periodically, so that changes in the reachability of nodes could be detected and handled. Also, announcements would not be broadcast immediately, but only after a minimum amount of time had elapsed since the last announcement. This was done to prevent announcement storms.

The design of the implementation consisted of a daemon that communicates with other nodes on the network and maintains information about locally announced services, as well as a client protocol allowing clients to connect to the daemon using a local socket (using the `AF_LOCAL` address family, historically known as `AF_UNIX`). The client protocol would be implemented by a Ruby module that could be used to easily implement clients, and three basic clients would be provided: one that announces a service, one that revokes an announced service, and ones that runs a query and displays the addresses of discovered service instances.

The stable branch of Debian GNU/Linux was chosen as the main platform for implementation and testing, with additional testing being performed on OpenBSD and Mac OS X.

The first steps in implementing Ahoy consisted of an implementation of bit vectors (data structures holding an arbitrary number of bits, each individually addressable) and Bloom filters (consisting of a bit vector and an array of hash functions), and implementing the `insert!` and `contains?` methods which implement the two operations defined on Bloom filters: inserting an item, and testing if the filter contains an item.

After Bloom filters had been implemented and tested, a skeleton of the Ahoy daemon was created, supporting the sending of Bloom filters to neighboring nodes and integrating received filters with the node's own filters. Next, a client interface was added, and clients were created to enable announcing services, querying, and revocation from the command line. The sending of queries and responses over the network was implemented and tested using the clients.

Up to this point, addresses had been represented in the format used by the `struct sockaddr` data structure from the C programming language, which is easily accessible from Ruby. However, this structure differs between operating systems, which was detected using cross-platform testing. Thus, a platform-independent representation (described in Section 5.3.1) was invented. This representation was subsequently used in both the main Ahoy protocol and the client protocol used in the communication between user programs and the Ahoy daemon.

A timeout was added to queries in the client code, so that clients would have an indication that no more responses would be coming. A timeout was also added in the daemon, to prevent clients from consuming resources of the daemon indefinitely.

So far, real computers in a real network had been used for testing. However, the network being used was a single-hop network with only a few nodes on it. To enable testing on multi-hop networks and using larger numbers of nodes, it was decided to perform testing using virtual nodes and networks.

Two software packages that provided virtual nodes and networks were identified: Xen [28] and User Mode Linux[22]. Although Debian packages were available for both, they proved problematic: the packages installed without any problem, but User Mode Linux would fail to boot virtual machines, whereas the Xen packages did not support IPv6, preventing the implementation of Ahoy from working. Attempts to compile Xen from the available source code consistently failed. Eventually, it was found that Debian etch (which was, at the time, in testing, with the eventual goal of becoming the new stable Debian release) included working User Mode Linux packages, and so the main testing platform was switched to etch.

The first tests in a multi-hop environment revealed lots of unnecessary query traffic, due to queries being propagated again and again, until their time to live ran out. This was remedied by preventing nodes from propagating a query back to the neighbor it was received from, and by adding a cache for seen queries, so that queries already processed could be discarded quickly.

By now, the implementation had gotten reasonably complex, with a number of values (such as the width of the used Bloom filters and the number of hops to propagate them to) hard-coded in it. Support for loading the configuration from a file was added, and many of the previously hard-coded parameters were made configurable, with the previously hard-coded values used as defaults.

The minimum and maximum delay for announcements were implemented. The purpose of the minimum delay is to prevent overloading the network; the maximum delay allows for the removal of information about nodes that have not sent announcements for some time.

At this point, the keep-alive and update-request messages were added. The motivation for the keep-alive message is that, any time an announcement is sent because the maximum delay has been reached, unnecessary traffic is generated: the full Bloom filters are sent out, even though all the information they hold should already be present at neighboring nodes. Since the purpose of this periodic broadcast is only to inform neighbors of the presence of the sending node, a new, small message was added just for that purpose. This message includes the generation-id of the latest announcement, so that nodes that have not received that announcement will notice. The update-request message allows them to request the latest announcement to be sent to them.

A random amount of jitter (with a configurable range) was added to the time between keep-alive messages to make collisions less likely. The code for selecting generation-ids and query-ids was changed to select random initial ids, and use random positive increments for new ids. This discloses slightly less information about nodes, and may prevent certain types of attacks.

The lifetime of service announcements coming from clients was limited, so that services will be revoked even when a client fails to do so explicitly (e.g. because the client crashed). The client module was updated to automatically re-announce services. The re-announcing stops when the main program exits or explicitly requests the service to be revoked.

8 Testing Report

Testing has been performed throughout the development of Ahoy. After every feature that was added, tests were run to verify that the feature behaved as intended, and did not disrupt existing behavior in undesired ways. During some of the tests, measurements were performed on the network to establish what messages were being sent and in what order, the number of messages being sent, and the size of the messages being sent.

A number of different test setups were used. The simplest tests were run on a single machine. These tests helped catch some bugs before more complex test setups were used. These tests used actual computers running Debian GNU/Linux 3.1 (woody), OpenBSD 3.8, or Mac OS X 10.4 (Tiger).

Most tests were run on a virtual network consisting of five virtual machines. The virtual networks and virtual machines were created using User-Mode Linux, and each of the virtual machines was running Debian GNU/Linux 4.0 (etch). Some of these tests used a regular grid structure, some tests used a topology where every node could reach every other node, and some tests used a script that periodically and randomly changed reachability to simulate mobility.

A number of tests were performed using virtual networks consisting of thirteen virtual nodes; the maximum number of nodes possible within the memory constraints on the machine running the tests. The tests being performed were identical to those performed under the five-node setup. As in the five-node tests, regular grid structure, full reachability, and randomly changing topologies were used.

The results of the tests show the implementation to work as expected. Announcements are being sent when knowledge of available services changes and in response to update requests. Update requests are sent when the Ahoy daemon is started, and in response to missed announcements. Keep-alive messages are sent at regular intervals, subject to random jitter. Queries are propagated as long as the time-to-live allows it and as long as there are neighbors whose latest announcements contain matches for the requested service name. Duplicate queries being received by nodes do not result in further network traffic. Responses are sent when queries reach nodes that offer a requested service. Services that were not re-announced by user programs are properly timed out.

Besides verifying that the prototype worked as expected, some data was also gathered on the amount of network traffic generated. For each of the test setups discussed below, some graphs visualizing the network traffic are shown, as well as the average number of bytes per second generated by Ahoy and the total network traffic (byte counts include UDP, IPv6, and Ethernet headers). Virtually all network traffic not generated by Ahoy is generated by OLSR (the routing protocol used), with a negligible portion of the traffic being ICMPv6 [29] control messages. These numbers give an impression of the impact Ahoy has on the network.

For each of the test setups, three different scenarios were run. In the first scenario, Ahoy is idle, and nodes only exchange keep-alive messages, update requests, and announcements as appropriate (the latter two message types are only used when the topology changes). In the second scenario, one node announces a service, waits 60 seconds, revokes the announced service, waits 60 seconds, announces it again, etc. This causes announcements to be sent. In the third scenario, one node is offering a service, and another node performs a query for that service every 20 seconds. This scenario generates query messages.

Network packets were captured with TCPDUMP [30]. Each node participating in the test captured the packets it sent and logged them to a file. After the test, the files from the individual nodes were combined into a single file using the `mergecap` utility from Wireshark [31]. The combined network dump was then used for further analysis. The average number of bytes per second (both for Ahoy and for all traffic) was extracted using Wireshark. A combination of TCPDUMP and a Ruby script was used to convert the packet data to a table listing the number of bytes sent for each second of the test. Graphs were generated from this data using Gnuplot [32].

The tests were run with the default parameters for Ahoy, except that **broadcast-queries** was set to **true**.

8.1 Five Nodes, Full Connectivity

The first tests were run in a simulated 5-node network with full connectivity. Every node can reach every other node directly. Figure 14 depicts one possible topography of such a network. Note that, for these tests, it does not matter what the network actually looks like; only the reachability of nodes is important.

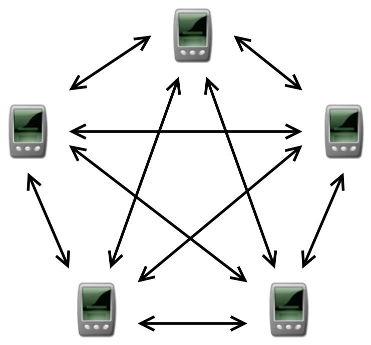
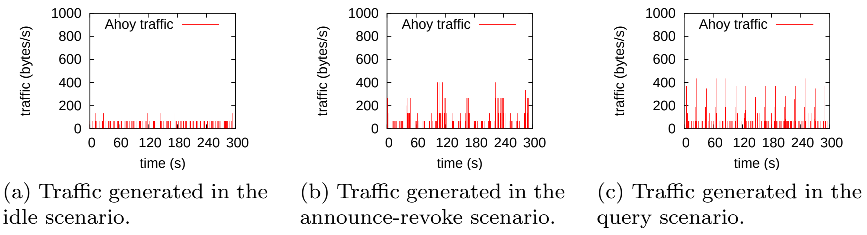


Figure 14: A five-node network with full connectivity.

Figures 15a through 15c show the network traffic generated in this setup. From left to right, they correspond to the idle scenario, the announce-revoke scenario, and the query scenario.



(a) Traffic generated in the idle scenario.

(b) Traffic generated in the announce-revoke scenario.

(c) Traffic generated in the query scenario.

Figure 15: Ahoy traffic in a five-node network with full connectivity.

The graphs clearly show that keep-alive messages generate very little traffic. In the idle scenario, the traffic generated by Ahoy is about 23 bytes per second on average (about 5 bytes per second per node). This compares to about 402 bytes per second of total traffic.

In the announce-revoke scenario (figure 15b), Ahoy traffic peaks every 60 seconds. This corresponds to announcement messages being sent when a service is introduced or removed. Also, the announcements inhibit the sending of keep-alive messages, which is expressed in the graph by the empty spaces following the spikes. In this scenario, Ahoy generated an average of 50 bytes per second (10 bytes per second per node). The total amount of traffic was 412 bytes per second on average.

In the query scenario (figure 15c), Ahoy generated an average of 52 bytes per second (about 10 bytes per second per node). The spikes that occur when queries are sent (every 20 seconds) are clearly visible. The average of all traffic was 437 bytes per second.

These measurements show the impact of Ahoy on a network to be very small. In fact, when the network is running OLSR, the extra traffic that running Ahoy generates is an order of magnitude less than the traffic already present in the network. Of course, other routing protocols may generate much less traffic, causing the relative impact of Ahoy to be greater.

8.2 Five Nodes, Grid Structure

Further tests were conducted in a simulated 5-node network with grid structure. In this setup, every node can reach every other node, but, in some cases, the message has to be forwarded by an intermediate node. The structure of this network is depicted in figure 16.

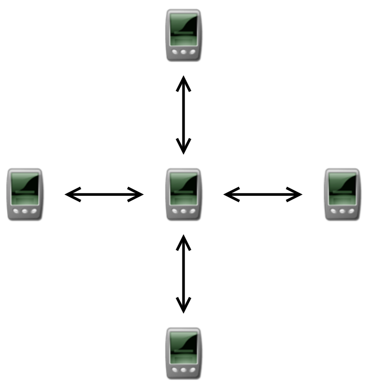
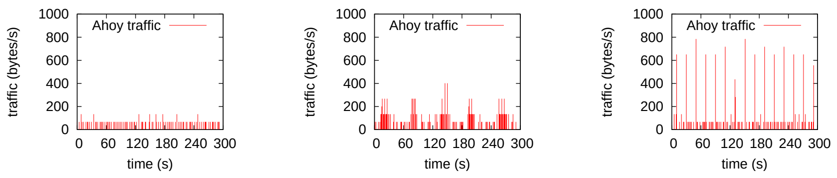


Figure 16: A five-node network with a regular grid structure. In the announce-revoke scenario, the rightmost node announces and revokes the service. In the query scenario, the rightmost node provides the service, whereas the leftmost node queries for it.

Figures 17a through 17c show the network traffic volume that Ahoy generates in this scenario. As before, the left figure corresponds to the scenario where Ahoy is idle; no services are being announced or revoked and no queries are run during the packet capturing. The middle figure shows the announce-revoke scenario, and the right picture shows the query scenario.



(a) Traffic generated in the idle scenario.

(b) Traffic generated in the announce-revoke scenario.

(c) Traffic generated in the query scenario.

Figure 17: Ahoy traffic in a five-node network with a regular grid structure.

The graphs are similar to those in the full connectivity setup, which means the same comments apply. The main apparent difference is that graph 17c has sharp, tall spikes, whereas in the corresponding graph for the full connectivity scenario (15c), the peaks are lower, but wider. This is probably a result of the way the graphs are generated rather than an actual difference in the traffic: messages that are captured in the same second will yield tall, narrow spikes, whereas messages that are captured in subsequent seconds will yield lower, wider peaks.

Ahoy generated 23 bytes per second (about 5 bytes per second per node) in the idle scenario, where the total amount of traffic was 337 bytes per second on average. In the announce-revoke scenario, Ahoy traffic averaged to 51 bytes per second (about 10 bytes per second per node), and total traffic to 337 bytes per second. The amount of Ahoy traffic in the query scenario was 57 bytes per second (about 12 bytes per second per node) on average, compared to a total of 406 bytes per second.

The results obtained from this setup are similar to the results obtained from the full connectivity setup. In particular, the amount of Ahoy traffic generated in the idle and announce-revoke scenarios is virtually the same between the two setups. This is to be expected, as these scenarios generate only broadcast messages whose number and size does not change between these two setups.

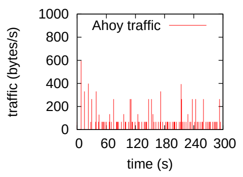
The amount of Ahoy traffic in the query scenario is higher in the grid setup (57 bytes per second) than in the full connectivity setup (52 bytes per second). The difference can be explained by the fact that response messages have to be forwarded to get from the rightmost node (the node providing the service) to the leftmost node (the node sending the queries). The size of the response message is 87 bytes (25 bytes payload plus 62 bytes lower level headers), and one is sent every 20 seconds. This generates about 5 bytes per second of extra traffic; exactly the difference observed between the two setups.

Another difference between the results from the grid setup and those from the full connectivity setup is that non-Ahoy traffic is lower in the grid setup. This is due to OLSR sending smaller messages in the grid setup than it does in the full connectivity setup.

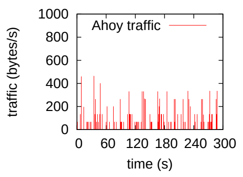
8.3 Five Nodes, Dynamic Structure

The last series of tests on networks consisting of five nodes was done with reachability randomly changing every 30 seconds. This results in update requests and announcements being sent whenever previously unreachable nodes become reachable. Consequently, the volume of network traffic generated during these tests can be expected to be higher than during the other five-node tests.

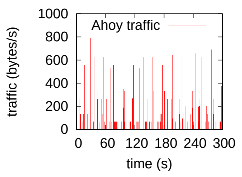
The graphs in figures 18a, 18b, and 18c represent the traffic generated in the idle scenario, the announce-revoke scenario, and the query scenario, respectively.



(a) Traffic generated in the idle scenario.



(b) Traffic generated in the announce-revoke scenario.



(c) Traffic generated in the query scenario.

Figure 18: Ahoy traffic in a five-node network with a dynamic structure.

The graphs generated by this setup have visibly more spikes than the graphs in the static setups. This is due to the unicast update-request and unicast announcement messages that are sent when a node discovers a new neighbor. Note that, although the network topology is changed every 30 seconds, a new neighbor is only discovered when a keep-alive message is received from it. Since the time at which keep-alive messages are sent is largely controlled by a randomizer, the peaks corresponding that occur in response to neighbor discovery are not evenly spaced.

In the idle scenario, the amount of Ahoy traffic was 36 bytes per second (about 7 bytes per second per node) on average. About 13 bytes per second of this was unicast traffic (update-requests and announcements in response to the discovery of new neighbors), with the remaining 23 bytes per second being the same keep-alive traffic that was also observed in the grid and full connectivity setups. The average of all traffic was 510 bytes per second in this scenario.

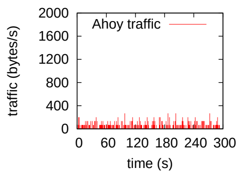
In the announce-revoke scenario, Ahoy generated an average of 49 bytes per second (about 10 bytes per second per node). As in the idle scenario, the unicast traffic was 13 bytes per second. This leaves 36 bytes per second of broadcast traffic; less than in the static setups. The difference is probably due to not all nodes being able to reach all other nodes in 4 hopsat all times. This would cause information about availability of the service not to be propagated to all nodes. The total amount of traffic was 480 bytes per second on average.

In the query scenario, the Ahoy traffic amounted to 70 bytes per second (14 bytes per second per node), of which 23 bytes per second were unicast traffic. Furthermore, an average of 3 bytes per second were due to response messages. This is less than in previous setups, because in the dynamic setup, the service cannot always be successfully discovered. The average of all traffic in this scenario was 521 bytes per second.

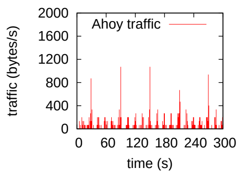
8.4 Thirteen Nodes, Full Connectivity

Besides tests with five-node networks, tests were also conducted using thirteen nodes; the maximum number feasible with available memory. The first series of tests with thirteen nodes were performed in a virtual network with full connectivity.

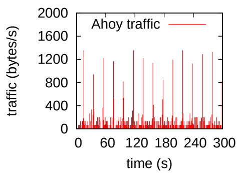
The scenarios are the same as the ones used in the five node setups. Figures 19a through 19c depict the traffic generated in each of the scenarios. Note that the graphs are on a different scale: where the top of the graphs is 1000 bytes per second in the five node setups, here it is 2000.



(a) Traffic generated in the idle scenario.



(b) Traffic generated in the announce-revoke scenario.



(c) Traffic generated in the query scenario.

Figure 19: Ahoy traffic in a thirteen-node network with full connectivity.

Unsurprisingly, the thirteen node setup generates more traffic than the five node one. In the idle scenario, 60 bytes per second of Ahoy traffic are generated. In the announce-revoke scenario, this number rises to 81 bytes per second. As with the five node setup, the query scenario generates most traffic; in this case, an average of 144 bytes per second.

Calculating how much traffic is generated per node per second in each scenario leads to the following numbers: about 5 bytes per node per second in the idle scenario, about 6 bytes per node per second in the announce-revoke scenario, and about 11 bytes per node per second in the query scenario.

The amounts of Ahoy traffic per node in this setup are similar to those computed in the five node, full connectivity setup, except for the announce-revoke scenario, which yields a lower value here. An explanation for this can be found in the ping-pong effect, where service information is propagated up and down between neighboring nodes. This happens, because, when a node announces a service that it knows about, the neighbors of that node will announce that there is knowledge of that service one hop away from them, which will cause the original node to announce that there is information about the service two hops away, and so on. The ping-pong effect ceases once the distance at which the information is found reaches the value of the **depth** parameter. This requires the same number of distinct announcements to be sent in the five-node case as in the thirteen-node case. However, in the thirteen-node case, there are more nodes, and thus, the traffic per node is lower.

Total traffic was 1809 bytes per second in the idle scenario, 1511 bytes per second in the announce-revoke scenario, and 1756 bytes per second in the query scenario. As in the five node setups, Ahoy traffic is well below total network traffic. In fact, the difference has increased: where Ahoy traffic per node has remained the same or decreased, OLSR traffic per node has increased.

8.5 Thirteen Nodes, Grid Structure

More measurements were performed in a network of thirteen nodes, with the grid structure depicted in figure 20.

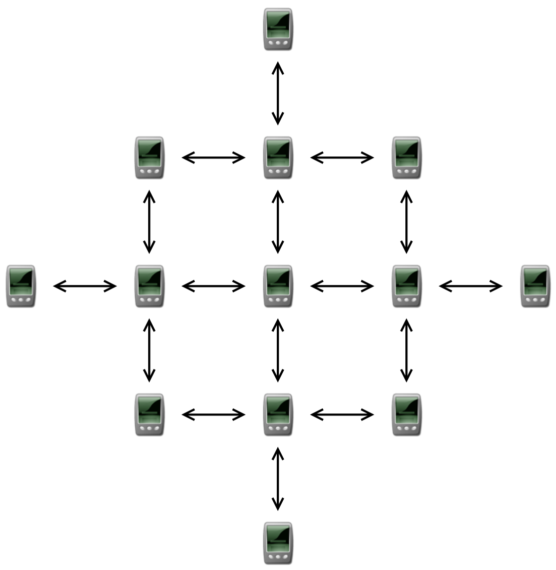
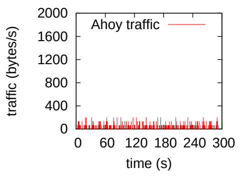
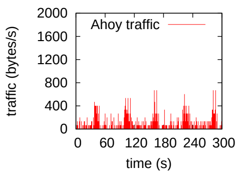


Figure 20: A thirteen-node network with a regular grid structure. In the announce-revoke scenario, the service is announced and revoked by the upper right node. In the query scenario, the service is provided by the upper right node, and queried for by the lower left node.

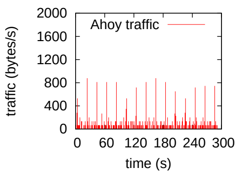
The generated network traffic is shown in figures 21a through 21c.



(a) Traffic generated in the idle scenario.



(b) Traffic generated in the announce-revoke scenario.



(c) Traffic generated in the query scenario.

Figure 21: Ahoy traffic in a thirteen-node network with a regular grid structure.

The measured average number of bytes per second of Ahoy traffic was 61 (approximately 5 per node) in the idle scenario (figure 21a), 117 (9 per node) in the announce-revoke scenario (figure 21b) and 99 (approximately 8 per node) in the query scenario (figure 21c).

Comparing the results obtained in this setup with those obtained in the thirteen node, full connectivity setup, a first observation is that traffic in the idle scenario has remained almost the same. This is in line with expectations, because an equal number of keep-alive messages should be sent in both setups.

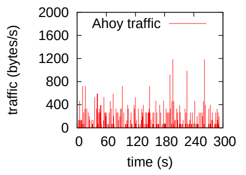
The announce-revoke scenario generates markedly more traffic in the grid setup than in the full connectivity setup. This may be somewhat surprising, given that the traffic generated in the corresponding setups using five nodes is equal. However, the difference can be explained by observing that the ping-pong effect causes more messages to be sent in a grid network than in a full connectivity network, because, in a grid network, not all messages are received by all nodes. In the five-node setups, this difference is not apparent, because of the limited depth of the network.

Compared to the full connectivity setup, the grid setup generates much less traffic in the query scenario. This is contrary to the results obtained from the corresponding five-node scenarios, where response forwarding caused the grid setup to generate more traffic than the full connectivity setup. The explanation for this observation can be found in the hop limit imposed on queries: in the thirteen-node grid setup, this causes the query not to be propagated by certain nodes, whereas in the depth of the five-node network is small enough that the query is propagated by every node.

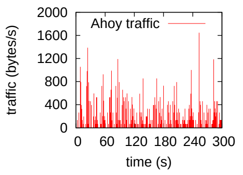
The average of all traffic generated in this setup was 1436 bytes per second in the idle scenario, 1535 bytes per second in the announce-revoke scenario, and 1560 bytes per second in the query scenario.

8.6 Thirteen Nodes, Dynamic Structure

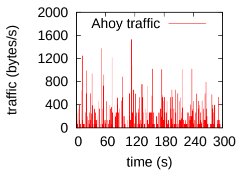
The last series of tests were performed on a thirteen node network with connectivity randomly changing every 30 seconds. The traffic graphs are shown in figures 22a through 22c.



(a) Traffic generated in the idle scenario.



(b) Traffic generated in the announce-revoke scenario.



(c) Traffic generated in the query scenario.

Figure 22: Ahoy traffic in a thirteen-node network with a dynamic structure.

Ahoy generated an average of 145 bytes per second (60 in broadcast traffic, 85 due to changes in network topology) in the idle scenario, 204 bytes per second (79 in broadcast traffic, 125 due to changes in network topology) in the announce-revoke scenario, and 226 bytes per second (127 in broadcast traffic, 4 in responses, and 95 due to changes in network topology) in the query scenario.

Translated to traffic generated per node, the numbers are 11 bytes per second (5 in broadcast traffic, 6 due to changes in topology) for the idle scenario, 16 bytes per second (6 in broadcast traffic, 10 due to changes in topology) for the announce-revoke scenario, and 17 bytes per second (10 in broadcast traffic, less than 1 in responses, and 7 due to changes in topology).

As expected, this setup generates the most traffic of all, owing to the update-requests announcements sent in response to changes in connectivity. When that traffic is discounted, traffic in the idle scenario is the same as always, as expected, and traffic in the announce-revoke and query scenarios is lower, which is also expected, because some nodes will sometimes be more than **depth** hops from an announced service, causing messages not to be propagated to them.

The total amount of traffic averaged 2938 bytes per second in the idle scenario, 2945 bytes per second in the announce-revoke scenario, and 3017 bytes per second in the query scenario. Thus, in all scenarios, Ahoy traffic is well below 10% of the total amount of traffic.

8.7 Summary

The conclusions that can be drawn from the measurements on networks of thirteen nodes are similar to those drawn after the measurements on networks consisting of five nodes. In a network running OLSR, Ahoy only adds relatively little to the traffic in the network. Moreover, the amount of Ahoy traffic each node generates is is about the same (in idle mode) or less (in the presence of announcements or queries) in thirteen-node networks as in five-node networks. This bodes well for the scalability of Ahoy.

9 Integration with JXTA

As a test case and a demonstration of the successful implementation of Ahoy, it has been integrated with JXTA. JXTA is a peer to peer framework created by Sun Microsystems. Two concepts from JXTA are key to the integration of JXTA and Ahoy: pipes and peer groups. Pipes are the abstraction JXTA peers communicate over. These pipes can be unicast pipes or propagate pipes. Unicast pipes are used for sending messages from one JXTA peer to another, whereas propagate pipes propagate messages to all peers in a peer group.

Peer groups are named groups of peers that participate in a shared service. All JXTA peers are members of the NetPeerGroup, but they can simultaneously be part of any other number of peer groups. For example, one peer could be part of the NetPeerGroup, a peer group hosting a distributed file system, and another group performing a distributed computation.

JXTA's pipe abstraction is implemented on top of a number of transports. One such transport is the TcpTransport, which listens on a TCP socket, and communicates with other JXTA peers by connecting to their TCP sockets. Transports implement methods for sending messages to specific addresses (unicast messages) and methods for sending messages to all members of a peer group (propagate messages). The integration of Ahoy and JXTA consists of altering the mechanism by which propagate messages are sent, without affecting the way unicast messages are handled. Figure 23 illustrates JXTA and the way Ahoy was integrated into it.

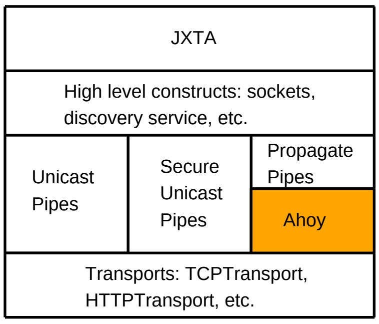

JXTA
High level constructs: sockets, discovery service, etc.
Unicast Pipes
Secure Unicast Pipes
Propagate Pipes
Ahoy
Transports: TCPTransport, HTTPTransport, etc.

Figure 23: Ahoy integrated with JXTA.

To integrate Ahoy with JXTA, an Ahoy client has been developed in Java. Because Java does not support `AF_LOCAL` sockets, the client uses TCP sockets to communicate with a small daemon called the `ahoy-tcp-interface`. This daemon relays messages between the Java client and the Ahoy daemon's `AF_LOCAL` socket.

When a JXTA peer joins a peer group, the integrated Ahoy client is used to send out an announcement to the Ahoy daemon. This announcement contains the name of the peer group and the address on which the TcpTransport of the peer is listening for incoming connections. One such announcement is sent for every IPv6 address the TcpTransport is listening on.

When a JXTA peer wants to send a propagate message, the integrated Ahoy client is used to perform a query for the destination peer group name. For every response to this query, the message is sent to the address contained in the response, using the TcpTransport. After being processed by Ahoy, the message is handed to the existing transports for the same processing that would have occurred in an unmodified version of JXTA.

The JXTA integration has been tested using the discovery tutorial shipped with JXTA and the JXTA shell [33], as well as with context publishing and subscription classes developed as part of the AWARENESS project [34].

10 Conclusions

The successful implementation of Ahoy shows that a service discovery protocol based on Bloom filters is, indeed, feasible. No part of the original protocol introduced in [2] and [3] was found in need of being altered because of implementation concerns.

To implement the protocol from [2] and [3], a number of issues had to be clarified. The original idea encompassed announcements using Bloom filters, as well as query and response messages, but did not specify the exact formats of these messages, nor how services are designated, nor what information is returned for discovered services. Ahoy addresses all of these issues, and extends the original protocol with keep-alive and update-request messages for efficiency reasons.

A number of design choices where identified. In some cases, various alternative choices were investigated. Sending service names in queries was chosen in favor of sending Bloom filters in queries, and service names in responses, because the former choice was expected to result in lower network load. Keep-alive and update-request messages were included in the protocol, even though the protocol could work without them, because they allow network topology changes to be detected more quickly and/or with generating less network traffic. A custom message format was used, rather than building on top of the Generalized MANET Packet/Message format, because it was felt to increase complexity of the message parser for no appreciable gain.

In the case of query propagation, two choices are offered as configuration options: unicast and broadcast. The expectation is that unicast is more efficient in some situations, and broadcast in others. Therefore, the decision is left to the user. The choice need not be uniform among nodes; the protocol works without any problems if different nodes use different propagation mechanisms.

11 Suggestions for Future Work

New and existing applications could be built on top of Ahoy. This could be done by integrating an Ahoy client in these applications, or by integrating Ahoy in an existing application programming interface or protocol. One possibility would be the development of an compatibility layer for existing applications using the ZeroConf protocol, such as can be found on Apple's Mac OS X [35] and the KDE [11] and GNOME [12] desktop environments. This would allow existing, unmodified applications to use Ahoy for service discovery.

Ahoy currently only works with IPv6: it uses UDP/IPv6 for sending and receiving messages, and the addresses included in queries and responses are IPv6 addresses. Support for other types of address could be added, for example IPv4 addresses or Ethernet MAC addresses. This would extend the utility of Ahoy to other types of network.

In the current version of Ahoy, responses to queries consist of an IPv6 address and a port number. This is adequate for services that are built on top of UDP or TCP and where the node requesting the service knows which one of these two to use. It could be useful to include a protocol identifier in the response, or to define completely different response types, for example URIs for services implemented on top of XML-RPC [36], or CORBA [37] identifiers.

Several useful features could be added to Ahoy. In particular, clients may be offered more control over queries by means of additional query parameters. In the current implementation of Ahoy, queries are propagated to all nodes within `depth` hops from the querying node that may offer the requested service, and no guarantees are given about the order in which results are returned. It is conceivable that clients are only interested in one response, or that clients wish responses to be in order of increasing hop count. Query parameters could be added that request nodes not to further propagate a query if a local match is found, or to perform an expanding ring search. Query parameters could also be used to match services by other attributes than service name (e.g. quality of service), or to request that information other than the service address is provided in response messages.

Another possible project would be integration of Ahoy with a reactive routing protocol such as AODV [26], where it could help reduce network traffic due to route requests.

There are several parameters to the operation of Ahoy that must be identical for all nodes in the network: the number of hops included in each announcement, the width of the Bloom filters being used, the number of hash functions being used, etc. Optimal values for these parameters exist, but depend on the properties of the network and the services being offered. Currently, there is no way for nodes to automatically agree upon these values, and no way for nodes to change the values in response to changes on the network. Work could be done to investigate the possibility of developing and integrating an agreement protocol that would have Ahoy nodes automatically decide on common values and react to changes in the network, so that correct and/or efficient operation is reached without user intervention.

Ahoy could be ported to different platforms, such as Microsoft Windows, Microsoft Windows Mobile, or Java 2 Micro Edition.

More measurements could be performed on the traffic generated by the Ahoy prototype. The measurements described in this report were performed on small networks. Measurements on larger networks could provide more insight in the scalability of Ahoy.

Appendix A Obtaining and Installing Ahoy

The prototype implementation of Ahoy is available online from
`http://ahoy.sourceforge.net/`.

Instructions for building, installing, and using the prototype are also provided there.

List of Figures

List of Tables

List of Abbreviations

AODV Ad-hoc On-demand Distance Vector.

CORBA Common Object Request Broker Architecture.

DAML DARPA Agent Markup Language.

DARPA Defense Advanced Research Projects Agency.

DNS Domain Name System.

DNS-SD DNS Service Discovery.

DSR Dynamic Source Routing.

FTP File Transfer Protocol.

GNU GNU's Not Unix!

HTTP HyperText Transfer Protocol.

IETF Internet Engineering Task Force.

IPv4 Internet Protocol, version 4.

IPv6 Internet Protocol, version 6.

KDE K Desktop Environment.

MANET Mobile Ad-hoc NETwork.

OLSR Optimized Link-State Routing.

OS Operating System.

PDA Personal Digital Assistant.

RPC Remote Procedure Call.

SMTP Simple Mail Transfer Protocol.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

URI Uniform Resource Identifier.

UTF-8 8-bit Unicode Transformation Format.

XML eXtensible Markup Language.

XML-RPC XML Remote Procedure Call.

References

[1] S. Deering and R. Hinden, “RFC 2460: Internet Protocol, Version 6 (IPv6) specification,” Dec. 1998. [Online]. Available: `http://www.faqs.org/rfc/rfc2460.html`

[2] F. Liu and G. J. Heijenk, “Context discovery using attenuated bloom filters in ad-hoc networks,” in *Proceedings 4th International Conference on Wired/Wireless Internet Communications, WWIC 2006, Bern, Switzerland*, ser. Lecture Notes in Computer Science, T. Braun, G. Carle, S. Fahmy, and Y. Koucheryavy, Eds., vol. 3970. Berlin: Springer-Verlag, May 2006, pp. 13–25.

[3] P. T. H. Goering and G. J. Heijenk, “Service discovery using bloom filters,” in *Proceedings of the twelfth annual conference of the Advanced School for Computing and Imaging, Lommel, Belgium*, B. P. F. Lelieveldt, B. R. H. M. Haverkort, C. T. A. M. de Laat, and J. W. J. Heijnsdijk, Eds. Delft, Netherlands: Advanced School for Computing and Imaging (ASCI), June 2006, pp. 219–227.

[4] “OPNET modeler,” [Online; accessed 16 December 2006]. [Online]. Available: `http://www.opnet.com/products/modeler/home-2.html`

[5] “JXTA,” [Online; accessed 16 December 2006]. [Online]. Available: `http://www.jxta.org/`

[6] Wikipedia, “Bloom filter — Wikipedia, the free encyclopedia,” 2006, [Online; accessed 16 December 2006]. [Online]. Available: `http://en.wikipedia.org/w/index.php?title=Bloom_filter&oldid=93571095`

[7] M. Rose, “UTF-8, a transformation format of ISO 10646,” June 1999. [Online]. Available: `http://www.faqs.org/rfcs/rfc2629.html`

[8] “Zero configuration networking (ZeroConf),” [Online; accessed 16 December 2006]. [Online]. Available: `http://www.zeroconf.org/`

[9] “Dns service discovery (DNS-SD),” [Online; accessed 16 December 2006]. [Online]. Available: `http://www.dns-sd.org/`

[10] “Bonjour,” [Online; accessed 16 December 2006]. [Online]. Available: `http://developer.apple.com/networking/bonjour/index.html`

[11] “K Desktop Environment,” [Online; accessed 16 December 2006]. [Online]. Available: `http://www.kde.org/`

[12] “GNOME: The free software desktop project,” [Online; accessed 16 December 2006]. [Online]. Available: `http://www.gnome.org/`

[13] “Multicast DNS,” [Online; accessed 16 December 2006]. [Online]. Available: `http://www.multicastdns.org/`

[14] F. Sailhan and V. Issarny, “Scalable service discovery for manet,” in *PERCOM ’05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 235–244.

[15] “DARPA Markup Language (DAML+OIL),” [Online; accessed 16 December 2006]. [Online]. Available: `http://www.daml.org/`

[16] D. Chakraborty, A. Joshi, T. Finin, and Y. Yesha, “GSD: A Novel Group-based Service Discovery Protocol for MANETs,” in *4th IEEE Conference on Mobile and Wireless Communications Networks (MWCN)*. Stockholm. Sweden: IEEE, September 2002.

[17] T. Clausen, C. Dearlove, J. Dean, and C. Adjih, “Generalized MANET packet/message format,” July 2006, [Online; accessed 16 December 2006]. [Online]. Available: `http://www.ietf.org/internet-drafts/draft-ietf-manet--packetbb-02.txt`

[18] "Extensible markup language (XML)," [Online; accessed 16 December 2006]. [Online]. Available: `http://www.w3.org/XML/`

[19] J. Postel, "RFC 768: User datagram protocol," Aug. 1980. [Online]. Available: `http://www.faqs.org/rfcs/rfc768.html`

[20] ——, "RFC 791: Internet Protocol," Sept. 1981. [Online]. Available: `http://www.faqs.org/rfcs/rfc791.html`

[21] "Ruby home page," [Online; accessed 16 December 2006]. [Online]. Available: `http://www.ruby-lang.org/`

[22] "User mode linux," [Online; accessed 16 December 2006]. [Online]. Available: `http://www.usermodelinux.org`

[23] "Debian," [Online; accessed 16 December 2006]. [Online]. Available: `http://www.debian.org/`

[24] "olsr.org," [Online; accessed 16 December 2006]. [Online]. Available: `http://www.olsr.org/`

[25] T. Clausen and P. Jacquet, "Optimized link state routing protocol (OLSR)," 2003. [Online]. Available: `http://www.faqs.org/rfcs/rfc3626.html`

[26] C. Perkins, E. Belding-Royer, and S. Das, "Ad hoc On-demand Distance Vector (AODV) routing," 2003. [Online]. Available: `http://www.faqs.org/rfcs/rfc3561.html`

[27] D. B. Johnson and D. A. Maltz, "Dynamic source routing in ad hoc wireless networks," in *Mobile Computing*, Imielinski and Korth, Eds. Kluwer Academic Publishers, 1996, vol. 353. [Online]. Available: `citeseer.ist.psu.edu/johnson96dynamic.html`

[28] "The xen virtual machine monitor," [Online; accessed 16 December 2006]. [Online]. Available: `http://www.cl.cam.ac.uk/Research/SRG/netos/xen/`

[29] A. Conta and S. Deering, "RFC 2463: Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification," Dec. 1998, obsoletes RFC1885. Status: DRAFT STANDARD. [Online]. Available: `ftp://ftp.internic.net/rfc/rfc1885.txt`, `ftp://ftp.internic.net/rfc/rfc2463.txt`, `ftp://ftp.math.utah.edu/pub/rfc/rfc1885.txt`, `ftp://ftp.math.utah.edu/pub/rfc/rfc2463.txt`

[30] "Tcpdump public repository," [Online; accessed 14 January 2007]. [Online]. Available: `http://www.tcpdump.org/`

[31] "Wireshark: The world's most popular network protocol analyzer," [Online; accessed 16 December 2006]. [Online]. Available: `http://www.wireshark.org/`

[32] "gnuplot homepage," [Online; accessed 14 January 2007]. [Online]. Available: `http://www.gnuplot.info/`

[33] "JXTA shell," [Online; accessed 16 December 2006]. [Online]. Available: `http://shell.jxta.org/`

[34] "Freeband communication," [Online; accessed 16 December 2006]. [Online]. Available: `http://www.freeband.nl/news.cfm?language=en&view=AWA&page=1`

[35] "Mac OS X," [Online; accessed 16 December 2006]. [Online]. Available: `http://www.apple.com/macosx/`

[36] "XML-RPC homepage," [Online; accessed 16 December 2006]. [Online]. Available: `http://www.xmlrpc.com/`

[37] Wikipedia, "Common object request broker architecture — Wikipedia, the free encyclopedia," 2006, [Online; accessed 16 December 2006]. [Online]. Available: `http://en.wikipedia.org/w/index.php?title=Common_Object_Request_Broker_Architecture&oldid=94430695`