University of Manchester
School of Computer Science
Third Year Project Report

Using Prediction to Improve PROPHET Routing

Author:
Thomas Nixon

Supervisor:
Dr Nick Filer

Abstract

This report presents the design, implementation and evaluation of an extension to PROPHET routing, based on prediction of encounters between devices. To facilitate this, a network simulator with a novel event-based architecture is designed and implemented. A scheme for predicting encounters between devices based on k-nearest-neighbour regression is implemented, and evaluated using the simulator; it is found to be effective, but too slow to use in a routing protocol.

A simpler and faster, but less general predictor is implemented, and integrated into the PROPHET routing protocol. This is evaluated in the simulator, and found to be effective in decreasing the overall latency and number of packets dropped.

University of Manchester
School of Computer Science
Third Year Project Report

Using Prediction to Improve PROPHET Routing

Author:
Thomas Nixon

Supervisor:
Dr Nick Filer

Acknowledgements

I would primarily like to thank my supervisor Nick for his support and guidance throughout this project. Thank you for encouraging me to work on interesting problems, consistently managing to motivate me when I had given up, and for defending my work despite glaring deficiencies.

Thanks to my friends, who not only put up with my incoherent ramblings, but managed to provide many a valuable insight.

Contents

1 Introduction **6**
1.1 Outline 7

2 Background **8**
2.1 Connections 8
2.2 Role 8
2.3 Classification 9
2.3.1 Infrastructure Networks 9
2.3.2 Mesh Networks 9
2.3.3 Delay Tolerant Networks 10
2.4 Issues Raised by Delay Tolerant Networking 10
2.4.1 Routing 10
2.4.2 Security 10
2.4.3 Power usage 10
2.4.4 Implementation and Applications 11
2.4.5 Testing 11
2.4.6 Delay and Unreliability 11
2.5 Routing in Delay Tolerant Networks 11
2.5.1 Epidemic 11
2.5.2 PROPHET 12

3 Tools **14**
3.1 Simulation Framework 14
3.1.1 Core Event Simulator 15
3.1.2 Event System 15
3.2 Example Simulation Set-Up 15
3.2.1 Connection Dataset 16
3.2.2 Connection Dataset 16
3.2.3 Splitter 16
3.2.4 Router 17
3.2.5 CsvWriter 17
3.2.6 AddTime 17
3.2.7 Statistics Logger 17

3.3 Datasets 18
3.3.1 Reality 18

4 Extending PROPHET with Prediction **19**
4.1 Using Regression for Prediction 19
4.1.1 Time Features 20
4.1.2 Regression Methods 21
4.1.3 Testing Methodology 23
4.1.4 Results 23
4.1.5 Evaluation 25
4.2 Week-In-Week-Out Approach 25
4.2.1 Prediction 29
4.2.2 Routing 29
4.2.3 Maintaining the Predictability Log 30
4.2.4 Testing 30

5 Future Work **35**
5.1 Prediction 35
5.1.1 Feature Selection 35
5.1.2 Weighting for Weeks 35
5.2 Routing 36
5.2.1 Devalue PROPHET 36
5.2.2 Evaluation 36

List of Figures

3.1 Example simulation set-up 16

4.1 Plot showing the distribution of predictabilities sampled during a simulation. 21
4.2 Simulation set-up for evaluating prediction 23
4.3 Plot showing the correlation between predictabilities and KNN predictions. 24
4.4 Predictability plot between two sample devices over several weeks. 27
4.5 Plot showing the correlation between predictabilities and corresponding predictabilities from one week ago. 28
4.6 Latency density estimates for each router. 32
4.7 Number of hops packets take for each router. 33

List of Tables

4.1 Features derived from an observation time. 20

Chapter 1

Introduction

Today, mobile phones have become abundant and inexpensive, causing a revolution in the way we communicate; not being tied to a land-line, post box or personal computer allows us to be where we want to be without sacrificing the ability to communicate with anyone on earth.

Mobile phones are widely used, but the way they typically communicate has a major drawback in that all communication takes place via network infrastructure such as cell towers and Wi-Fi hotspots. This may not seem like a big problem to many of the people reading this report, but to many it is.

In developing countries, there is often very little or no infrastructure available, making communication difficult. In countries with oppressive governments, messages travelling over network infrastructure can be intercepted and censored, making it difficult to (for example) organise political protests without the knowledge of the authorities. Communication has become a basic human right and should be treated as such.

Nearly all modern mobile phones implement short-range networking technologies such as BlueTooth and Wi-Fi; these technologies are arguably under-utilised and could eventually, with enough work, become much more widely useful as the foundations of a new paradigm in network technology: delay tolerant networking.

In a traditional network, packets are only delivered if there is a complete unbroken path from the source to the destination. In a delay tolerant network, packets reach their destination by being stored and opportunistically forwarded to other devices, allowing packets to reach their destination even if a complete path from the source never exists in the lifetime of the packet. This comes at the expense of latency, but allows communication to occur where it would otherwise be impossible.

When to forward and when to store a packet is decided by a routing algorithm. There are many published routing algorithms for use in delay tolerant networks; this report presents a new approach, implemented as an

extension to an existing routing algorithm.

1.1 Outline

Chapter 1: Introduction This chapter. The motivation for delay tolerant networking is presented.

Chapter 2: Background An overview of various networking technologies and architectures is given. Delay tolerant networking is defined, and its challenges identified. Two existing routing algorithms, Epidemic and PROPHET are introduced.

Chapter 3: Tools Discusses the requirements, design and implementation of a simulation framework. Introduces the Reality dataset.

Chapter 4: Extending PROPHET with Prediction A technique for predicting activity in PROPHET routed networks is developed and evaluated through a series of experiments. A routing algorithm utilising these predictions is presented and evaluated.

Chapter 5: Evaluation Evaluation and possible improvements of to the work presented in this report.

Chapter 2

Background

For the purpose of this report, a computer network can be modelled as a graph whose topology changes over time; the nodes of this graph are *devices* which have a *role*, and the edges are *connections*.

2.1 Connections

A connection allows two or more devices to communicate. Once a connection has been established, any connected device can publish a message containing some data, which all other devices can receive.

Typical types of network connection:

Ethernet An Ethernet connection connects two or more devices together over a physical wire.

BlueTooth A BlueTooth connection allows two devices to communicate via short-range (typically 10m, but up to up to 100m) using radio signals. Although radio signals are inherently a broadcast technology (that is, other devices in range cannot be prevented from receiving messages), encryption is used to create logical connections between two devices.

Wi-Fi Wi-Fi connections are similar to BlueTooth connections in that they operate over radio, and create virtual connections for two devices to communicate over, but has a significantly longer range.

GSM GSM connections allow mobile phones to communicate with base stations using radio frequency, with ranges of a few kilometres.

2.2 Role

The role of a device denotes what function it performs on the network. A device may perform one or more roles.

Endpoint An endpoint device acts to send messages to other devices, and receive messages from other devices. Typical endpoint devices are mobile phones, desktop computers and servers.

Message Router A device acting as a message router is connected to several other devices, and forwards messages between these devices. When a message is received, a router looks at the destination address of that message, and decides what to do with it. It may:

Forward the message via another network connection, either to the destination, or another router device.

Discard the message if it does not know what to do with the message.

Store the message if it cannot forward the message to the destination immediately, but may be able to do so in the future.

2.3 Classification

By restricting this general description of a computer network, we can describe three common types of computer network.

2.3.1 Infrastructure Networks

In in infrastructure network, a device is *either* an endpoint or a router, never both. The topology of infrastructure networks rarely change, and when they do this usually causes some disruption of service. Message routers never store messages, except for short periods of time to alleviate congestion (queueing). Infrastructure networks tend to have a well connected core of routers for redundancy and bandwidth reasons, surrounded by trees of routers and endpoints to minimise the amount of wiring needed, at the cost of redundancy and bandwidth.

2.3.2 Mesh Networks

In a mesh network, devices are simultaneously endpoints and message routers, but do typically not store messages except for queueing. Devices in a mesh network are typically mobile devices such as laptop computers or mobile phones, communicating over short range wireless networks such as Wi-Fi or BlueTooth. The topology of a mesh network is typically fairly constant once in operation, but has no well defined structure – devices communicate with any other in-range devices.

Mesh networks are typically set up to allow several devices to communicate where there is a lack of traditional network infrastructure.

2.3.3 Delay Tolerant Networks

Delay tolerant networks are similar in architecture to mesh networks, except that message routers are allowed to store messages for future forwarding. This kind of network is most useful in situations where there is low overall connectivity, but the topology of the network changes frequently.

The primary advantage that delay tolerant networking has over mesh networking is that a message that starts at device a can be received by device b even if there is never a complete path of connections from a to b.

Delay tolerant networking raises several issues, discussed in the next section.

2.4 Issues Raised by Delay Tolerant Networking

2.4.1 Routing

Making routing decisions in a delay tolerant network is much more difficult than in an infrastructure network or a mesh network, as the usefulness of routing decisions made now is affected by changes in the topology of the network in the future.

2.4.2 Security

In an infrastructure network, the devices which perform routing are typically owned by the network operator, and are therefore considered secure – messages from endpoint devices are routed through these devices, and users trust them not to read or alter these messages. In a mesh or delay tolerant network, the devices performing routing are also owned by users of the network, and so are not considered to be trusted. Since messages must travel via untrusted devices to reach their destination, it is imperative that some form of encryption and cryptographic signing is performed on the packets, to stop the intermediate devices from reading or altering these messages.

2.4.3 Power usage

When a routing decision is being made for a packet in a delay tolerant network, the device making the decision does often not know of a complete route to the destination, and so must use a heuristic, taking into account the previous, current, and predicted connectivity of devices around it. When one device decides to forward a message to another, it uses some amount of power to do so. Since devices typically used in delay tolerant networks are running on battery power, conserving energy is important.

When making routing decisions, the device must consider both power efficiency and prompt delivery of messages. This is a trade-off, as if a message

is forwarded more, it is more likely to be delivered, but may use excessive amounts of power in the process.

2.4.4 Implementation and Applications

Networking stacks on many common computers are not designed to be used in this way. As an example, on many common mobile phones it is not possible for a piece of software to connect to another phone using BlueTooth without it asking the user to enter a pairing code; this obviously makes delay tolerant networking, which relies on constantly creating new connections more difficult to implement. It is possible to work around these kinds of restrictions by modifying the operating system, but this would definitely slow the adoption of such a technology.

2.4.5 Testing

In order to evaluate the effectiveness of a delay tolerant networking technology, it is desirable to run computer simulations to gauge the overall efficiency. In order to do this, one needs a simulator, of which there are many freely available, and some data to run it on, which presents more of a problem.

Datasets typically take the form of a connectivity log for a number of devices (listing the start time and duration of each connection between devices), and possibly a log of messages sent between these devices. Datasets of this type are expensive to collect, and so few are available.

2.4.6 Delay and Unreliability

In order for messages in a delay tolerant network to be delivered, they will by necessity be delayed by a variable and unpredictable amount of time, or not delivered at all; this style of networking is therefore inappropriate for use in situations where this is not acceptable.

2.5 Routing in Delay Tolerant Networks

To me, message routing seems like an interesting thing to work on, since there if no obvious best approach; I decided to learn more about this.

There are several established approaches to routing in delay tolerant networking, described below.

2.5.1 Epidemic

Epidemic routing[VB+00] gets it's name from the way diseases spread in a community; instead of diseases we have messages, and instead of disease carriers, we have devices.

Epidemic is probably the simplest working routing scheme in existence; each device stores a set of messages that it has seen. When two devices meet, they exchange messages such that both devices end up with the same set of messages (the union of the two previous sets). A device sends a message by adding it to it's own set of messages. A device receives a message when a message addressed to it is found in it's set of messages.

Advantages

- Guaranteed to deliver messages as early as possible.
- Simple to implement.

Disadvantages

- *Extremely* inefficient. Epidemic routing does not try to direct packets towards their destination – it just tries to deliver messages to as many devices as possible in the hope that they will reach their destination. This results in more forwards (using power), and larger buffer sizes (using memory and power).

2.5.2 PROPHET

The Probabilistic ROuting Protocol using History of Encounters and Transitivity (PROPHET)[LDS03] routing protocol uses the fact that the probability of a connection occurring between two devices is not constant to inform routing decisions.

Each device a in a PROPHET-routed network stores, for each other device b a *predictability* value, $P(a,b)$. The value $P(a,b)$ is a measure of the likelihood of device a being able to deliver a message to device b.

Predictability values are assumed to be zero if not known, and are updated using the following rules:

- When device a encounters device b, the predictabilities are updated such that:

  $$P(a,b) = P(a,b)_{\text{old}} + (1 - P(a,b)_{\text{old}}) \times P_{\text{init}}$$

  Where P_{init} is a constant. This takes care of increasing the delivery predictabilities for nodes that often encounter each other.

- Delivery predictabilities are aged over time, such that devices that haven't had an encounter in a long time have lower predictabilities:

  $$P(a,b) = P(a,b)_{\text{old}} \times \gamma^k$$

  Where γ is a constant, and k is the time since the last encounter.

- Predictabilities have a transitive property; if a is able to deliver a messages easily to b, and b is able to deliver messages easily to c, then if follows that a should easily be able to deliver messages to c. Upon a encountering b, delivery predictabilities are updated as follows:

  $$P(a, c) = P(a, c)_{\text{old}} + (1 - P(a, c)_{\text{old}}) \times P(a, b) \times P(b, c) \times \beta$$

  Where β is a constant.

When device a has the opportunity to forward a message destined for c to device b, it only forwards if:

$$P(b, c) > P(a, c)$$

This way, messages are only forwarded to devices that have a *higher* chance of delivering the message to the destination.

Advantages

- Causes far fewer forwards compared to epidemic routing, and thus uses less power.

Disadvantages

- Maintaining the predictabilities costs both storage space and power, though this is usually offset by increased routing efficiency.
- PROPHET routing may miss opportunities to deliver messages that epidemic may be able to take advantage of – it is a heuristic.

Chapter 3

Tools

In order to do any work on message routing in delay tolerant networks, it is necessary to have some way to evaluate the performance of a routing algorithm. In the ideal world, we would do this by installing our software onto a large number of mobile devices owned and used by real people going about their every-day lives, and having them use it to communicate. Unfortunately, this is usually not much more than a pipe-dream, as there are still many problems to be solved with this technology before it becomes useful to a wide audience.

To resolve this situation, simulations are usually used. A network simulator simulates the connections between devices and the users of the devices which send messages, and logs relevant data for later analysis.

3.1 Simulation Framework

There are several network simulators freely available under permissive open source licences, however I was not generally impressed by the amount of work required to get data into and out of these simulators, and the amount of effort required to implement a new routing protocol. I therefore decided to write my own simulation framework; this of course was more work overall, but ended up being an interesting part of this project in it's own right.

Python was chosen as the implementation language, as it is very flexible and has a large and high quality library of modules; this made it quick to try things out, without writing a lot of boiler-plate code, or implementing standard algorithms. Although python is an interpreted language, the performance was good enough for our purposes, and was even better when using the experimental PyPy[PyP12] interpreter.

3.1.1 Core Event Simulator

Everything that happens in the simulator is controlled by a single global clock, which runs as fast as it can; because the amount of work to do in each step of the clock is not constant, this clock has little relation to real world time.

At the core of the simulator is the `Simulator` class. This manages the global time, and ensures that callbacks added to it are run at the correct time.

Callbacks take the form (t, c), where t is the time to run c, a function taking a single argument – the `Simulator` instance. These callbacks are ran in increasing order of time. If two callbacks have the same time, they are ran in the order that they were added to the simulator.

In addition to individual callbacks, it is possible to add callbacks that are repeatedly called at a given interval. These are called when

$$t \bmod i = 0 \wedge t > t_{\text{first}}$$

where t is the current timestep, i is the given interval, t_{first} is the time of the first non-interval event.

3.1.2 Event System

Just using the above core event simulator works well enough for small simulations with only a few components, but for more complicated set-ups, with more components, this becomes more complicated. Ideally, we should be able to design generic components in isolation with similar interfaces, then join them together in a high-level way.

In order to make this a possible, a system of event sources, sinks and filters was designed, modelled after UNIX pipes.

Each event in the system consists of a time, and a map from field names to values. There are no restrictions on the field names or types; the only standard field name is 'type', a string denoting the type of event.

Event source objects can emit events, and event sink objects can receive events. Event filter objects are both event sinks and event sources, so can both emit and receive events. Event sources can be 'chained' onto event sinks, meaning that all events emitted by the event source are received by the event sink at the time of the event, no matter where it was emitted.

3.2 Example Simulation Set-Up

Figure 3.1 shows the architecture of a simple simulation run that reads in connection and messaging datasets, and produces a log of messages sent and received in CSV format, and a file of statistics (including, for example, the number of messages forwarded by each node).

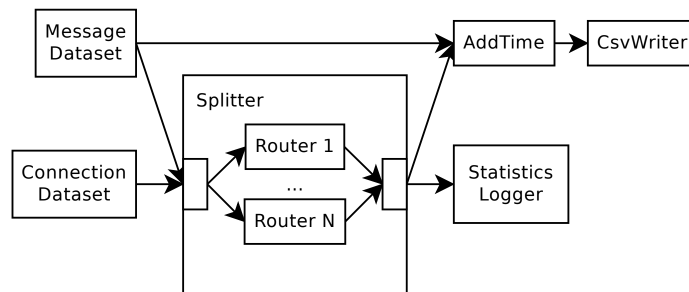

Figure 3.1: Example simulation set-up

The components of this simulation are as follows:

3.2.1 Connection Dataset

This is an event emitter, which should emit events with the following keys each time a message is to be sent:

type `"message"`.

left The ID of the device sending the message.

right The destination ID of the message.

id A unique ID for the message, to allow the devices to differentiate between otherwise identical messages.

3.2.2 Connection Dataset

This is an event emitter which emits an event every time a device goes in range of, or out of range of another device, with the following keys:

type `"connect"` or `"disconnect"`.

left The ID of the device which recorded this event.

right The ID of the device which was sighted.

3.2.3 Splitter

Splitters are event filters, and ultimately pass events to a set of inner filters, managed by the splitter. Upon initialisation, the splitter is given two parameters: the inner class (in this case, a router class), and a list of keys to split on (in this case, a single element list containing `"left"`).

When a splitter receives an event (from the outside), it tries to find an inner filter identified by the values in the event for the keys which the splitter is to split on (in this case, the value of the `"left"` key of the event). If a

matching inner filter is not found, it is created; the event is then passed unmodified to this filter.

In this case, this has the effect of creating a routing filter for each value of `"left"` in the dataset, and therefore each device.

When a new inner filter is created, it is given a reference to the splitter, so that it can communicate with the other filter instances, as well as the event that created it, so that it can determine it's own id, for example.

Whenever an inner filter emits an event, the splitter re-emits it, after adding the keys/values that identify the inner filter.

3.2.4 Router

The routers receive connection and messaging information from the datasets via the splitter, and use this to make routing decisions. Routers exchange messages via the splitter. When a router receives a message for which it is the destination, it emits an event with `type="message_received"`, so that the effectiveness of the router can be analysed.

Additionally, in the case of a PROPHET router, the first time another device is seen, an event with the following keys is emitted:

type `"p_value"`

right The ID of the device that was seen.

p A zero-argument function that returns the predictability of this device at the current timestep.

This allows other components to easily sample predictabilities without having references to the routers.

3.2.5 CsvWriter

Given a file to write to, and a list of column/key names, this simply writes the specified keys of each event as a row in a CSV file.

3.2.6 AddTime

The AddTime filter simply writes the event time of received events to a specified key, and re-emits them. This is more generally useful than it may seem, but in this case is simply there to make the CSV writer slightly simpler and more general.

3.2.7 Statistics Logger

At the end of the simulation, the routers all emit events with `type="stats"`, containing statistics gathered while they are running. The statistics logger writes these to a single CSV file for later analysis.

3.3 Datasets

Data about connections and messages is needed to drive the simulation. To be useful for our purposes, a dataset must:

- Be collected from a reasonable number of devices.
- Be collected for a large amount of time (at least a few weeks).
- Be collected in an environment where some behaviours are predictable.

These requirements are difficult and expensive to meet, so there is only one dataset currently available that meets these: the Reality Mining dataset.

3.3.1 Reality

In the 2004, 100 subjects at MIT were given mobile phones pre-installed with data collection software, which they used for 9 months. The phones collected times of BlueTooth device sightings, calls placed and received, text messages sent and received, and cell towers seen. [EP05]

Importantly, the test subjects were students and faculty at MIT; this is ideal as this kind of environment is often quite predictable.

Chapter 4

Extending PROPHET with Prediction

PROPHET works well, but there is one obvious area to work on if it is to be improved – it's use of historic data. The routing algorithm potentially has access to years of connectivity and messaging information, and yet it each device only stores a single number for each other device detailing how likely it is to be able to deliver messages to that device; this is surely a missed opportunity.

One fundamental problem with delay tolerant networking is that devices do not know what is going to happen in the future – if they did, they could have the same message delivery rate as Epidemic routing, while only transferring messages between devices that are *needed* to get get them to their destination.

In real life, these devices are often carried by people, who tend to have highly predictable behaviour in one way or another. Students taking the same course may meet at the same time every week for lectures, but may hardly meet otherwise. Workers travel to and from work at the same time every day, and tend to do predictable things once there.

We may not know exactly what is going to happen in the future, but we could *predict* what is going to happen with reasonable accuracy, this information could help a device to make better forwarding decisions, hopefully improving overall efficiency.

4.1 Using Regression for Prediction

To enable prediction of PROPHET predictability values, each device may keep a log of these values for each other device, sampled every few minutes. Each of these logs can be thought of as a function

$$f(t) = p$$

where t is the time of an observation, and p is the predictability at t. The domain of f is all times in the past.

In order to predict a predictability value in the future, we must find a function $f_p(t)$ which is defined for all values of t. This can be done using regression analysis.

Regression analysis techniques can be used to fit a function to a set of data points in n dimensions; this is often used to provide insight into a dataset (for example, by fitting a line to a set of points), and to allow extrapolation past the bounds of an experiment.

In this case, we want to relate time to predictability values. This is a complex function; far too complex for any known method of regression analysis to fit to while providing meaningful extrapolation.

4.1.1 Time Features

From a time value, we can calculate several *features*, as described in table 4.1. These features are designed to capture the way that people typically describe repeating events.

Name	Description
year	Year number (eg. 2012)
month	Month number (January = 1, February = 2 etc.)
day	Day number in the month
minute	Minutes since the start of the day
dow	Day number (Monday = 1, Tuesday = 2 etc.)
odd_wk	Week number in year modulo 2

Table 4.1: Features derived from an observation time.

As an example of these in use, these features could be used to represent an event that occurs every other Friday night between 6:00 and 7:00 as

$$\begin{aligned} &\text{dow} = 5 \\ \wedge\ &(60 \times 18) \leq \text{minute} \leq (60 \times 19) \\ \wedge\ &\text{odd_wk} = 0 \end{aligned}$$

, or an event that occurs on the second Tuesday of every month between 12:00 and 12:30 as

$$\begin{aligned} &\text{dow} = 2 \\ \wedge\ &(60 \times 12) \leq \text{minute} \leq (60 \times 12.5) \\ \wedge\ &\text{odd_wk} = 0 \\ \wedge\ &8 \leq \text{day} \leq 14 \end{aligned}$$

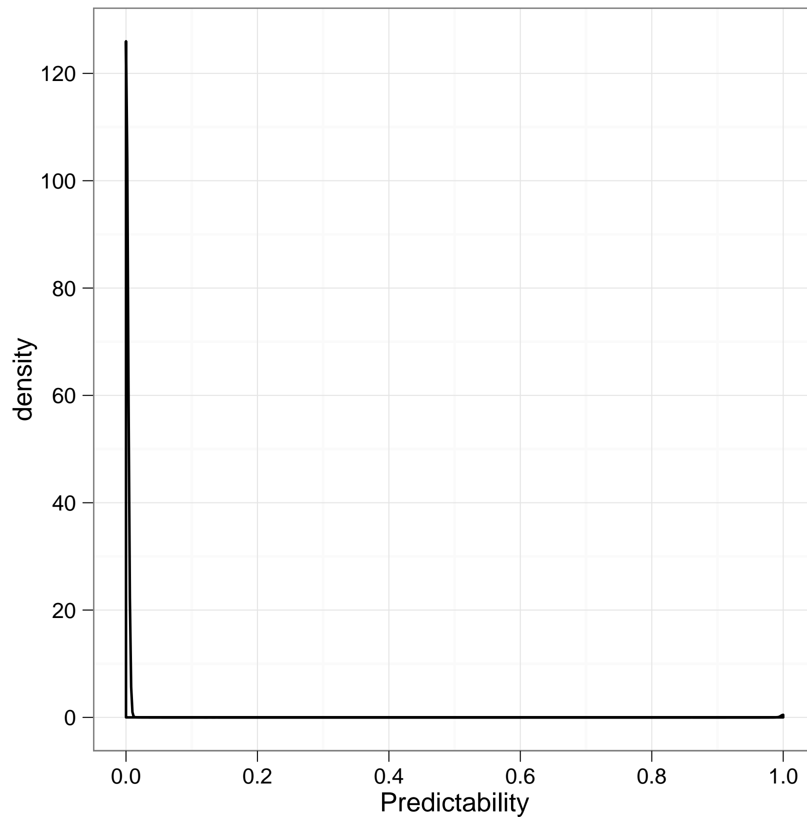

Figure 4.1: Plot showing the distribution of predictabilities sampled during a simulation. Note that predictabilities above zero barely register.

Recurring events are much more naturally expressed in terms of these features than raw time values, so it follows that it should be much easier to fit a model to a predictability log if the time values are pre-processed into these features.

4.1.2 Regression Methods

There are plenty of methods for performing regression, however most are unsuitable for use in this situation, because of both the non-linearity of the problem, and because of the high bias of predictabilities towards 0 – the average predictability recorded during a typical simulation run is just 0.0056, which is treated as noise by most regression methods. Figure 4.1 illustrates just how severe this problem is.

One regression method that seems plausible in this situation is k-nearest-neighbour regression.

K Nearest Neighbour

A k-nearest-neighbour regression[Kra11] model is simply the set of all points to perform regression on. The prediction for an example data point is simply some combination of it's k closest data points.

The k closest data points are chosen to minimise their euclidean distance from the example. This can be done in one of two ways:

Brute Force Iterate through each stored data points, calculating the euclidean distance for each, and taking the k points with the smallest distance.

k-dimensional Trees Store all the points in a k-dimensional[1] tree[Ben75], and query that to find the closest points. A k-d tree is a binary tree where each node is a point in k dimensional space, and both stores the point in the data structure, and serves to split up the space into two parts by a $k-1$ dimensional hyperplane. For a given node in the tree, all children on the left are on one side of the hyperplane, and all children on the right are on the other. The hyperplane associated with each node is always perpendicular to an axis; which axis is determined by $d \bmod k$, where d is the depth of the node in the tree. k-d trees are usually balanced, such that each node has roughly the same number of children in it's left and right subtrees.

The SciPy library for Python contains a fast implementation of k-d tree building and searching[Sci12]; this was used to implement this prediction method.

k-d tree searching is generally considered to be logarithmic in the number of points[FBF77][2], however the SciPy implementation is limited in that new points cannot be inserted without rebuilding the entire tree, incurring a large overhead.

To get around this, the implemented predictor divides all points to be searched into 'bins', each of which contain data for one day, and has it's own k-d tree representing that data. This scheme reduces the size of the tree that needs to be rebuilt after each sample is added (increasing the efficiency of adding new samples), while increasing the number of trees that need to be queried to make a prediction (decreasing the efficiency of prediction). This is a trade-off, but results in an overall efficiency improvement.

Once the k closest data points to the example have been found, they need to be combined to find a final prediction; this is usually done by taking the mean.

[1] This is a different k than in k-nearest-neighbour.

[2] The algorithm that SciPy uses is a mystery, but empirical tests show that it is faster than using brute-force search, if only because it is implemented in C.

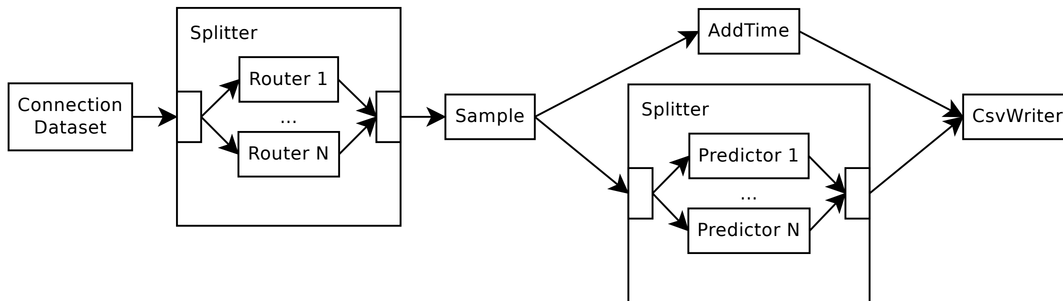

Figure 4.2: Simulation set-up for evaluating prediction

4.1.3 Testing Methodology

In order to test a prediction method, it is necessary to test it against real-world data. To do this, the PROPHET routing protocol was ran on all devices in the Reality dataset, and the predictabilities generated are sampled every 5 minutes; these samples are passed into a predictor for each pair of devices. Every 5 minutes, the predictors are queried for a prediction 1 hour into the future. Both the sampled predictabilities and the predictions are logged to a table, with each row containing a sample, and a prediction for the same time as the sample. Figure 4.2 shows the architecture of this test in the simulator.

The higher the performance of a predictor, the more correlated these two values should be; in a perfect predictor these two variables would always be equal.

4.1.4 Results

Figure 4.3 shows the aggregated predictability and prediction pairs, collected as described above for $k = 5$. In an ideal predictor, these pairs would contain perfectly correlated values, and so a straight line with positive gradient would be visible – clearly not what we have here. A few observations:

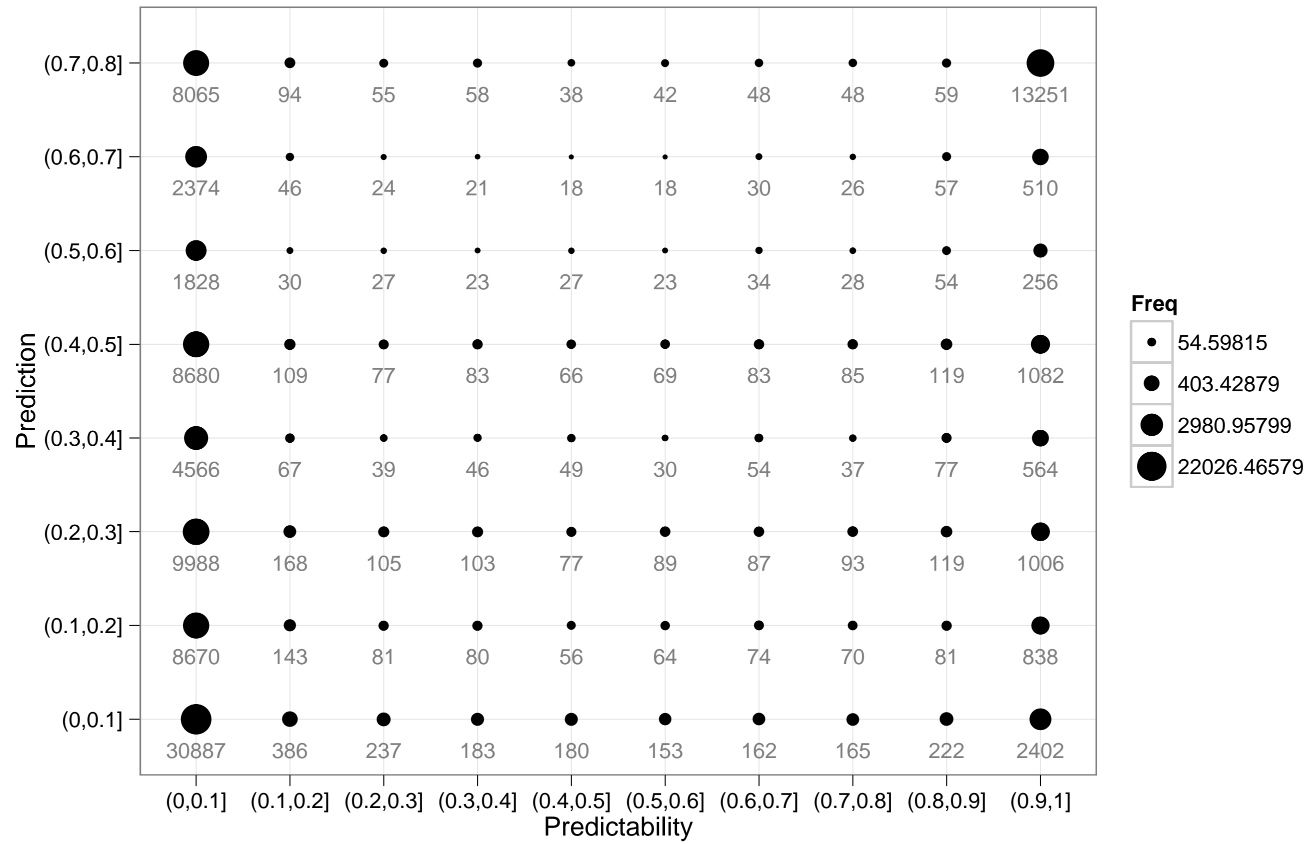

Figure 4.3: Plot showing the correlation between predictabilities and KNN predictions.

- The predicted values have a maximum of around 0.8. This is probably caused by the presence of a large number of near-zero predictabilities in the data, pulling the average down. This is not a problem for our purposes, as as the routing algorithm using these predictabilities (described in section 4.2.2 on page 29) never compares predictions against predictabilities.

- There are obviously a significant number points where the prediction is wrong (a mix of false negatives around $(1, 0)$, and false positives around $(0, 0.8)$), however there is also a large number of correct predictions (true negatives around $(0, 0)$ and true positives around $(1, 0.8)$). Although they are not perfect, these predictions would probably still be useful to a routing algorithm, as the knowledge of a possible encounter in the future is more useful than no knowledge at all.

  During routing, a false negative would result in a message not being forwarded in a situation where it may have been beneficial, and so would not cause any gain or loss in efficiency compared to PROPHET. A false positive would result in a packet being forwarded in a situation where it is not beneficial, reducing the efficiency.

It is possible to use the Pearson product-moment correlation coefficient[RN88] (PPMCC) to measure the correlation between predictabilities and corresponding predictions. The PPMCC of two variables is 1 if they are perfectly positively correlated, -1 if they are perfectly negatively correlated, and 0 if there is no correlation. The PPMCC for these two variables is 0.460, showing a weak positive correlation as expected.

4.1.5 Evaluation

The KNN approach seems to give reasonably good predictions, but is slow enough that any routing algorithm using it would be far too slow to test. A more efficient implementation may make this fast enough to be usable, but this approach has another major pitfall that makes it not worth the effort to implement.

When routing a message, it would be useful to know the maximum predictability between now and the time of expiry (based on the TTL and time of sending) of the packet. This is not something easy to do with this predictor without sampling many times in that interval; an entirely new approach is needed.

4.2 Week-In-Week-Out Approach

This new approach to prediction uses the following two observations to increase efficiency:

- Most predictable events seem to occur on a weekly basis. Figure 4.4 shows predictabilities between two devices over 7 weeks; there appear to be several cases where 'spikes' in predictability happen at the same time in several weeks. Figure 4.5 shows a plot of predictabilities against predictabilities from one week ago, which shows many of the same traits as the KNN-based predictor. These two variables actually have a slightly higher PPMCC at 0.475 (vs. 0.460 for KNN).

- Most predictability samples are zero, or very close to zero; it would be beneficial to be able to filter these out without affecting accuracy, to save storage space and power. It does not make sense to filter out these low predictabilities in a regression model, as this would alter the shape of the function; in the KNN predictor, if no predictabilities less than $P_{\min}$ are added to the model, all predictions will be greater than or equal to $P_{\min}$.

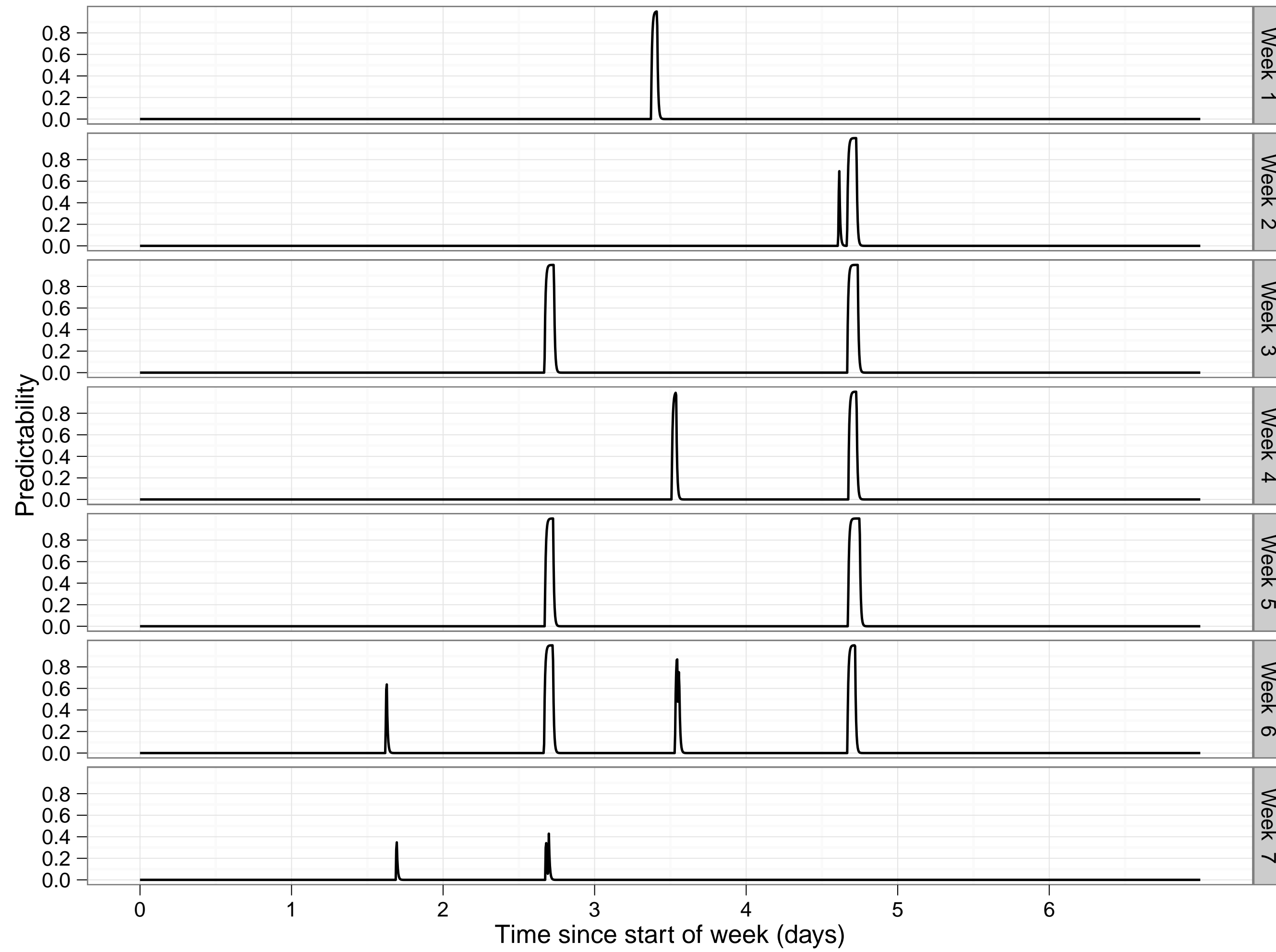

Figure 4.4: Predictability plot between two sample devices over several weeks.

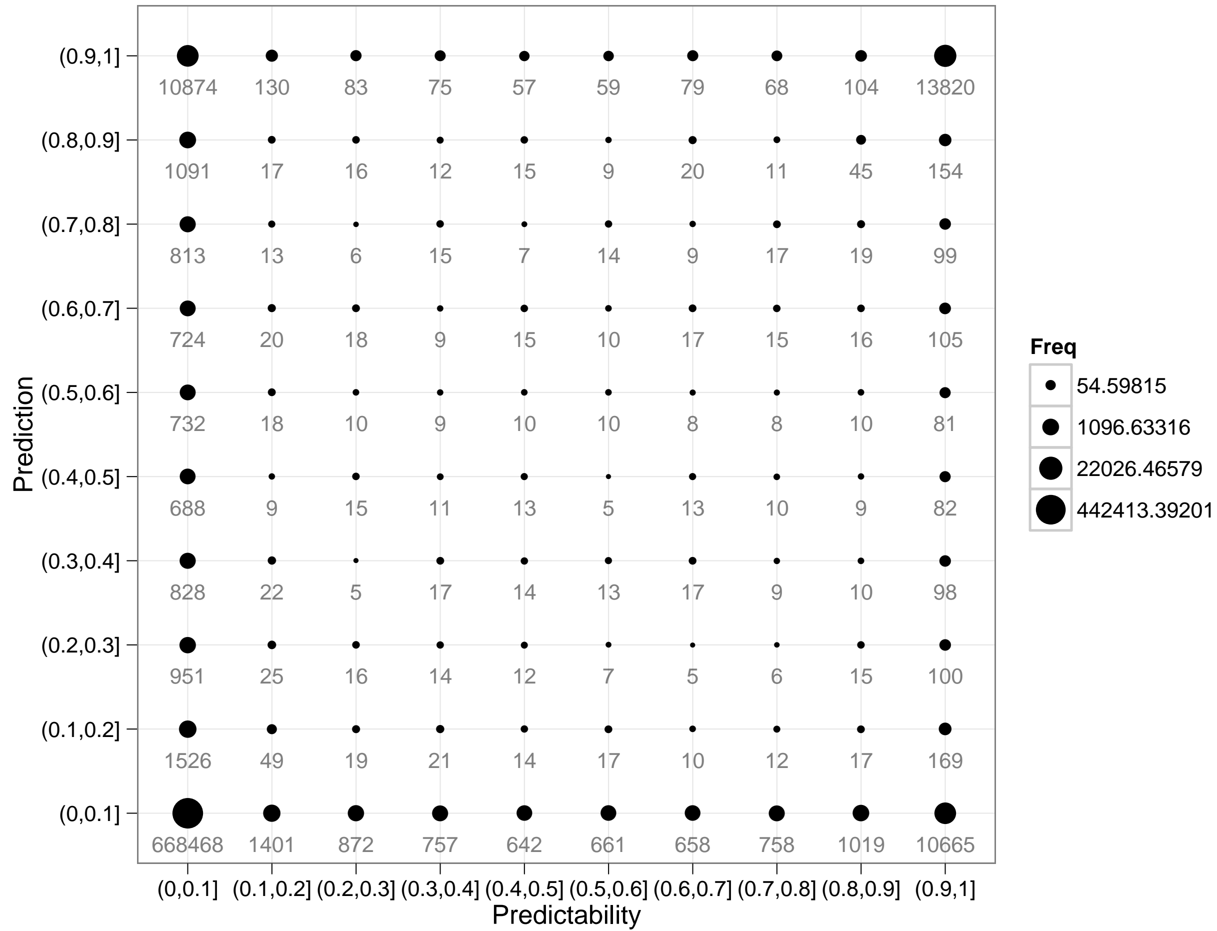

Figure 4.5: Plot showing the correlation between predictabilities and corresponding predictabilities from one week ago.

4.2.1 Prediction

In this scheme each device a stores a set $L(a, b)$ of pairs (t, p), where p is the predictability observation at time t for device b. These observations have the following restrictions:

- Samples are taken every i seconds:

$$t \bmod i = 0$$

- Only the most recent n weeks of samples are stored:

$$t > t_{\text{now}} - n \times 7 \times 24 \times 60 \times 60$$

  Where t_{now} is the current simulator time.

- Only probabilities greater than P_{min} are stored:

$$p > P_{\text{min}}$$

Rather than calculating a predictability estimate for a specific time, this predictor calculates an estimate of the maximum predictability between two times, t_a and t_b:

$$P'(a, b, t_a, t_b) = \sum_{w=1}^{n} \max \left\{ \begin{array}{l} p \mid (t, p) \in L(a, b) \\ \quad \wedge\, t_a < (t + w \times 7 \times 24 \times 60 \times 60) < t_b \end{array} \right\}$$

That is, for each week stored in the log, calculate the maximum predictability between t_a and t_b in that week (the same days and times, just shifted into that week), summing across all weeks.

The idea behind this is to represent both the predictability, and our confidence that this predictability will occur; if a device has been seen at the same time the last 3 weeks, the predictability estimate will be 3 times higher than if it was just seen at this time last week.

With some careful implementation, this method is fast enough to use to help inform routing decisions. All observations are stored in a hash table indexed by date, so to predict using n weeks of data, the predictor only needs to actually look at n days of data.

4.2.2 Routing

A packet destined for device c, that expires at t_{exp} is allowed to be forwarded from device a to device b at time t_{now} if:

$$\begin{array}{l} P(b, c) > P(a, c) \\ \vee\; P'(b, c, t_{\text{now}}, t_{\text{exp}}) > P'(a, c, t_{\text{now}}, t_{\text{exp}}) \end{array}$$

The first part of this, $P(b,c) > P(a,c)$, is the same as in PROPHET routing – if b is currently a better carrier of the message than a, then forward to it. The second part deals with predictions – if b is predicted to be better than a at delivering the message before the packet expires, then forward to it.

This is best implemented using short-circuiting logic – checking the predictability is much quicker than checking the prediction, so avoiding calculating it if possible is a good idea.

4.2.3 Maintaining the Predictability Log

The predictability log must be kept up to date for this to be effective; this means that a new predictability sample must be added every i seconds if the predictability is greater than $P_{\min}$.

It is possible to implement this by simply sampling each predictability every i seconds, and updating if the sample is greater than $P_{\min}$, however this slows down the simulation significantly; calculating the predictably requires several floating point operations, and may need to be preformed up to n^2 times every i seconds in a simulation with n nodes.

To speed up the simulation, the predictor has two states – logging and not logging; when in the logging state, the predictability log is updated every i seconds, and vice versa. The following rules are applied when updating $L(a,b)$:

- When the predictor is initialised, it does not start logging predictabilities.
- When device a contacts device b, the predictor starts logging, as the predictability for this pair is now increasing.
- When a predictability value is to be logged, if $P(a,b) < P_{\min}$, and a is not currently connected to b, this predictor stops logging, as the predictability has gone below the threshold, and will continue to decrease.
- When the predictability is updated by the transitive property, if $P(a,b) > P_{\min}$, this predictor starts logging if it was not already.

Applying these rules ensures that all the predictor is always in the logging state when the predictability is above the threshold, and allows it to stop when it drops below, resulting in a marked gain in efficiency.

4.2.4 Testing

Strategy

Using this prediction method, predicted values are not 'compatible' with the predictabilities, and the prediction is done on a time range rather than a

single time, so it is not possible to evaluate the prediction method by it's self – the improved routing method and predictor must be evaluated as a whole.

To do this, simulation was used. The simulation logs the times that each message was sent and received; this allows us to analyse the distribution of latency, and find the proportion of messages delivered successfully.

Three routing schemes are compared:

`prophet` Unmodified PROPHET.

`prophet_predict` PROPHET with prediction, described above.

`prophet_manual` PROPHET with additional random forwards. The number of forwards that PROPHET and PROPHET with prediction perform are saved; this routing scheme is simply PROPHET, but with enough additional random message forwards so that it does the same number of forwards as PROPHET with prediction.

The `prophet_manual` routing scheme is useful, as it allows us to better compare `prophet` and `prophet_predict`. `prophet_predict` will always forward more packets than `prophet` and so will almost certainly reduce the latency and increase the delivery ratio compared to `prophet`; what we really want to know is how effective those extra forwards being performed are at getting messages to their destination. Comparing `prophet_predict` to `prophet` with additional random forwards allows us to determine this.

Each routing scheme was ran on the Reality connection dataset, and the same random message dataset, with 1,000,000 messages at random times, and with random source and destination devices.

Results

Figure 4.6 shows the distribution of latency across all received messages for each router. While `prophet_predict` makes a small improvement in the number of messages received compared to `prophet` across all latencies, the comparison against `prophet_random` is more telling. `prophet_random` makes virtually no improvement compared to `prophet`, and yet makes the same number of additional forwards as `prophet_predict`. `prophet_predict` is clearly doing *something* right, but quantifying how effective it is is beyond the scope of this project.

Figure 4.7 shows the number of message received at their destination after taking a given number of hops for each router. The shape of this graph is caused by two things:

- Some packets that were delivered successfully by `prophet` get to their destination quicker by using a larger number of hops in `prophet_predict`. This is most visible at one hop, where the total number of

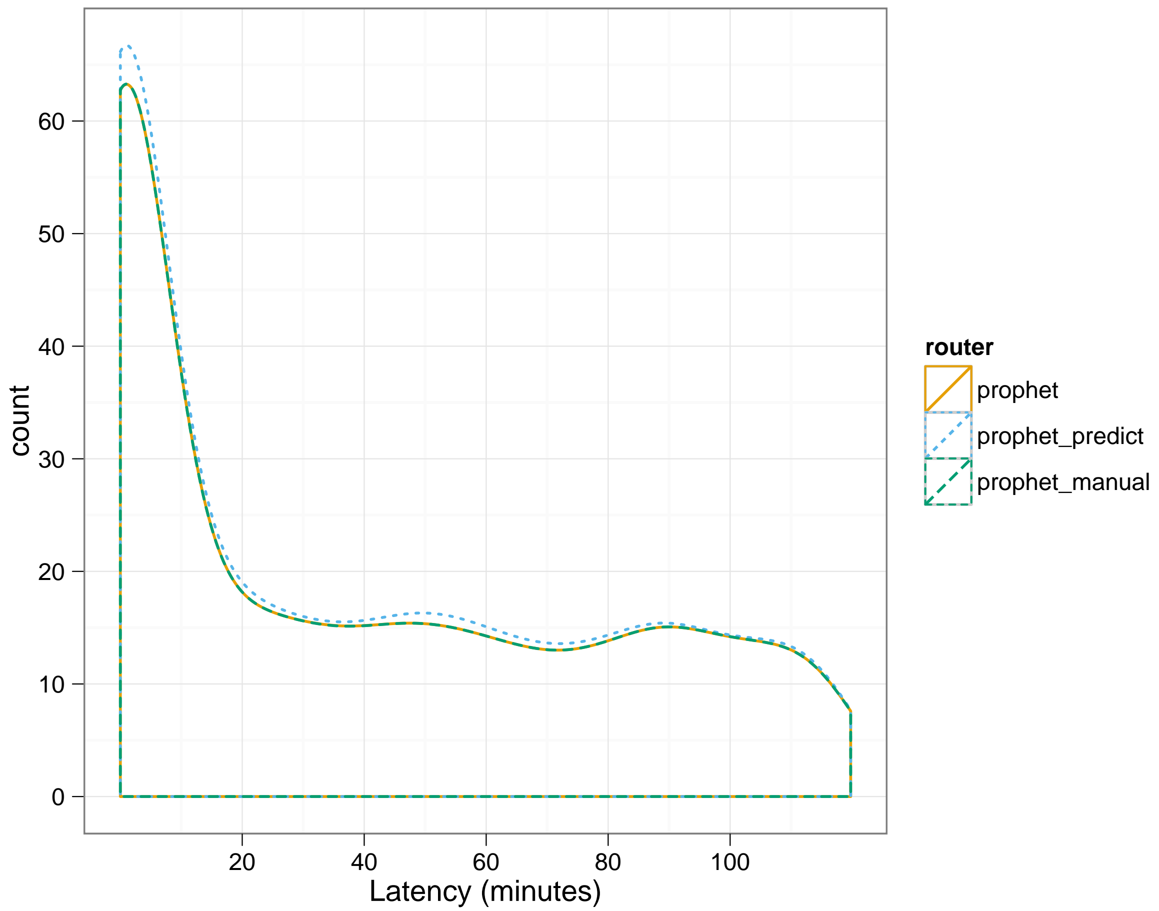

Figure 4.6: Latency density estimates for each router. The lines for `prophet_predict` and `prophet_manual` are directly on top of each other.

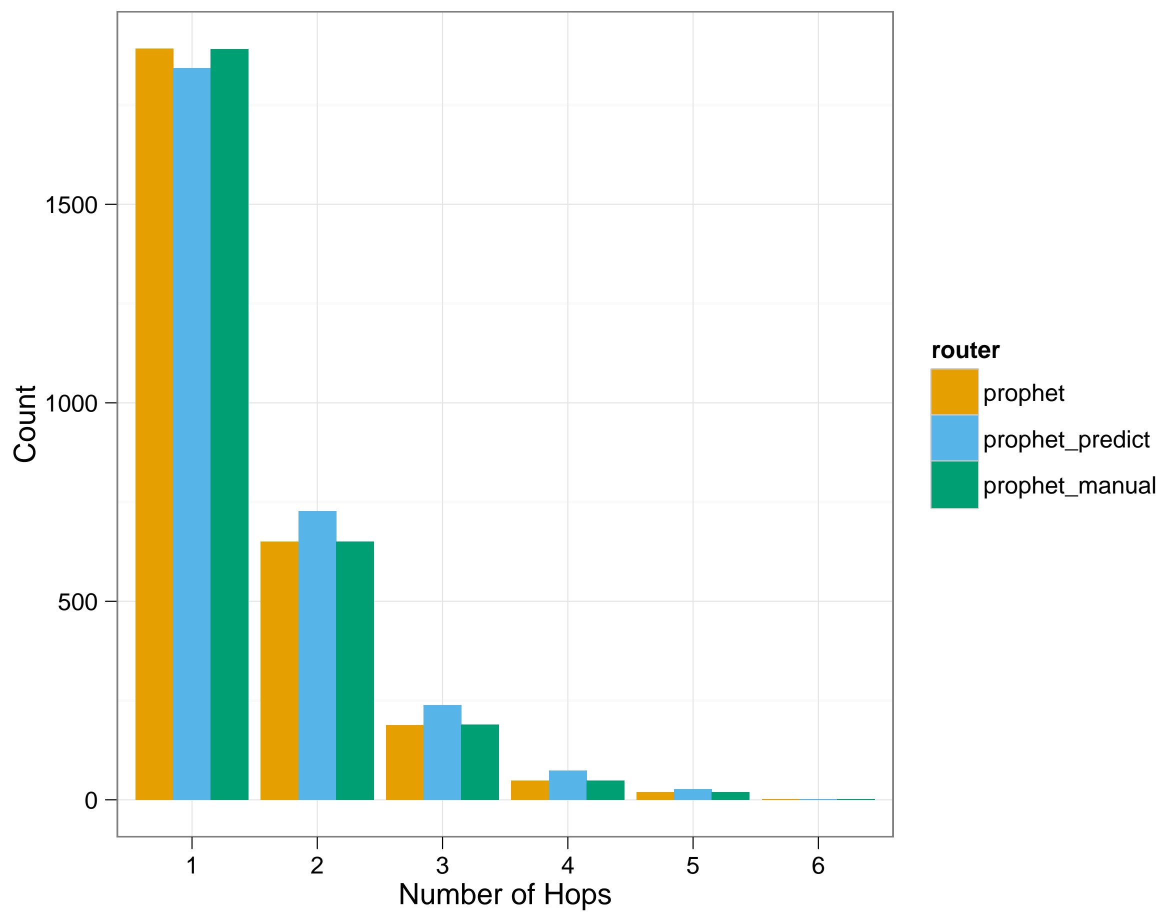

Figure 4.7: Number of hops taken by each packet to reach it's destination for each router.

packets received actually decreases. These packets were still received by the destination, but took a larger number of hops to get there.

- Some packets that were not delivered by `prophet` are delivered successfully by `prophet_predict`. This increases the number of packets received at each number of hops, apart from one hop. This is because if a packet can be delivered in one hop, it will definitely be delivered by `prophet`.

This routing algorithm is more effective than PROPHET, but not by a large amount. This is arguably held back by the quality of the prediction being used – with better prediction, a similar technique may be able to make a more significant improvement.

Chapter 5

Future Work

This section presents possible improvements to the various aspects of this project.

5.1 Prediction

A major part of this project was the design of two methods for predicting future predictabilities in a PROPHET routed network. Some possible improvements to these schemes:

5.1.1 Feature Selection

When using k-nearest-neighbour prediction, not all features are equally relevant, which can affect negatively prediction – if one of the features is not correlated with the predictability in any way, it provides no benefit but can skew the predictions.

Feature selection is a technique to alleviate this problem, and can be used to find a scaling factor for each feature, or a subset of all features. This could improve the performance of the predictor, both in terms of speed and effectiveness.

We found that using only the time since the start of the week as a single feature was similarly effective to a large number of features. This was true in the environment recorded by the Reality dataset, but is probably not true in general; some environments may show more day-to-day similarity than week-to-week, or may exhibit a mixture of both. It would be much more general if each device could perform feature selection for each other device to decide how best to perform prediction for that device.

5.1.2 Weighting for Weeks

In the current predictor, if a device was seen at the appropriate time one week ago (and not before that), the prediction will be the same as if the

device was seen at the appropriate time three weeks ago. This is perhaps counter-intuitive, but on the other hand, there is no obvious function from a vector of maximum predictabilities for the previous n weeks to a single prediction value. It may be possible to learn this function by performing regression between the predictability vectors and the observed predictabilities for this time.

5.2 Routing

Some possible improvements to prediction-aware routing:

5.2.1 Devalue PROPHET

In the extended PROPHET routing protocol, a packet destined for device c, that expires at t_{exp} is allowed to be forwarded from device a to device b at time t_{now} if:

$$\begin{aligned} & P(b,c) > P(a,c) \\ \vee\ & P'(b,c,t_{\mathrm{now}},t_{\mathrm{exp}}) > P'(a,c,t_{\mathrm{now}},t_{\mathrm{exp}}) \end{aligned}$$

This means that it always forwards if PROPHET would. This is possibly not the best idea, as PROPHET may well forward in cases where it is inefficient to do so. It may be worth removing PROPHET entirely:

$$P'(b,c,t_{\mathrm{now}},t_{\mathrm{exp}}) > P'(a,c,t_{\mathrm{now}},t_{\mathrm{exp}})$$

Or making it so that PROPHET only forwards if the predictability for b is significantly greater:

$$\begin{aligned} & P(b,c) > P(a,c) + \Delta_p \\ \vee\ & P'(b,c,t_{\mathrm{now}},t_{\mathrm{exp}}) > P'(a,c,t_{\mathrm{now}},t_{\mathrm{exp}}) \end{aligned}$$

Where Δ_p is an initialisation constant.

5.2.2 Evaluation

I feel that the evaluation of the extended PROPHET routing protocol could have been much more rigorous. It may have been useful to:

- Compare several runs on different messaging datasets – they are generated randomly, so it would be useful to check that `prophet_predict` isn't just performing better than `prophet` due to some quirk of the random number generation.

- Pre-compute the best path each message could have taken to reach it's destination using Epidemic routing, and compare the forwarding decisions made by the routers to these paths – if a router makes the best possible routing choices more often, it should have a higher overall performance.

- Test the routing protocols in other environments – the Reality dataset was recorded in an academic environment, and so we have only shown extended PROPHET routing to be effective in this environment. Datasets for other environments may have to be generated, as there are not many recorded ones available, and those that are tend to be too short for use in a routing simulation that requires historic data.

- Test the routers on a messaging dataset that more closely resembles real life. People do not send messages randomly, so we should not be testing against random messages. The Reality dataset does include a log of messages sent and received for each device, but unfortunately there are too few messages to produce any reliable results. It would be interesting to generate a larger messaging dataset based on the patterns seen in the Reality dataset.

Bibliography

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[EP05] Nathan Eagle and Alex (Sandy) Pentland. CRAWDAD data set mit/reality (v. 2005-07-01). Downloaded from `http://crawdad.cs.dartmouth.edu/mit/reality`, July 2005.

[FBF77] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, September 1977.

[Kra11] Oliver Kramer. Unsupervised K-Nearest Neighbor Regression. July 2011.

[LDS03] Anders Lindgren, Avri Doria, and Olov Schelén. Probabilistic routing in intermittently connected networks. *SIGMOBILE Mob. Comput. Commun. Rev.*, 7(3):19–20, July 2003.

[PyP12] PyPy. Pypy website. `http://pypy.org/`, 2012. Accessed 29 April 2012.

[RN88] Joseph Lee Rodgers and W. Alan Nicewander. Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42(1):pp. 59–66, 1988.

[Sci12] SciPy. `scipy.spatial.ckdtree` class. `http://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.cKDTree.html`, 2012. Accessed 29 April 2012.

[VB+00] A. Vahdat, D. Becker, et al. Epidemic routing for partially connected ad hoc networks. Technical report, Technical Report CS-200006, Duke University, 2000.