

A cognitive approach to congestion control in
Delay Tolerant Networks



Dissertation

**Submitted in partial fulfillment of the requirements for the award of
the degree of Bachelor of Technology in Electronic Engineering at
Indian School of Mines, Dhanbad**

Submitted by

Durga Prasad Pandey, Admission number: 7655

Supervisor

Dr Vishnu Priye

**Department of Electronics and Instrumentation
Indian School of Mines
Dhanbad 826004 India**

Acknowledgements

I would like to express my heartfelt thanks to my project guide, Dr Vishnu Priye. He gave me full freedom in pursuing new ideas, and provided a supportive and challenging environment to work in.

I am thankful to Sri BSR Sastry, Head of my department, and various teachers who provided moral support and encouragement throughout the duration of the project.

I am also indebted to the following people whose active support and critical input enabled the project to reach its logical conclusion: Scott Burleigh, Dr Vint Cerf, Prof Leonard Kleinrock, Adrian Hooke, Chris Ramming, Dr Keith Scott, Pat Feighery, Leigh Torgerson, Dr Kevin Fall and Stephen Farrell. I thank them all.

Lastly, I would like to thank all my friends, and family members, who were always there.

Durga Prasad Pandey
Admission Number 7655

Contents

Section	Topic	Page number
1	Introduction	3
2	Background	3
3	Congestion in DTNs	5
4	Our solution: Structural flow control	5
5	Implementation	8
6	Results and Analysis	14
7	Conclusion and future work	21
8	References	22

1. Introduction

The aim of this project is to test the hypothesis that a machine-learning program can detect congestion in a Delay Tolerant Network (DTN). To realise this objective, a C program (congestion detection daemon) was designed which would simulate the characteristic network status information at a DTN node. This would be in the form of periodic report from the DTN node on:

- Rates at which it is receiving data from other nodes,
- Rates at which it is sending data to other nodes, and
- Rate at which local free storage is increasing or decreasing.

Based upon this data and its derivatives, the congestion detection daemon would output a numeric figure, whose range would subsequently be used to obtain the predicted level/degree of congestion at the node on a scale from 1(least likely) to 5(imminent).

The data obtained from the C program was then used to train and test three machine learning algorithms: ZeroR, OneR and Decision Table. The algorithms are available as an open source implementation using Java in a software package called *Weka*. Weka is a publicly available Java-based machine learning package developed at the University of Waikato in New Zealand[14].

The results from this project have strengthened the belief that machine learning techniques can be useful in predicting congestion in a DTN, even when the input data to the program is varying randomly. It is hoped that this effort will give some encouragement to current efforts in developing approaches for ‘cognitive networking’, where networking approaches go beyond being algorithmic-based to techniques based on cognition and learning.

2. Background

Delay Tolerant Networks (DTNs) [1] are characterized by large latency, intermittent connectivity, and high and/or variable error rates. Research on DTNs originally started with an attempt by Vint Cerf, a co-author of the TCP/IP protocol suite, and a team of scientists at JPL led by Adrian Hooke to build the framework for an InterPlanetary Internet(IPN)[2] which would extend internetworking to space. The IPN team developed a message switched architecture[3][4] based on bundling[5], which was later generalized into Delay Tolerant Networking[6] to include networks whose operational characteristics make traditional approaches either unworkable or impractical. Examples[7] of such networks include terrestrial mobile networks, with a commuter bus acting as a store and forward message switch[8]; exotic media networks such as undersea and free-space optical communication networks; military ad-hoc networks; and sensor and sensor/actuator networks. Thus the InterPlanetary Internet was the precursor to DTNs in much the same way as the ARPANET was to the Internet. As an illustration, we will use the example of the InterPlanetary Internet for our discussions.

The IPN architecture proposes a new layer called the bundling layer that lies just below the application layer. At interplanetary gateways, transport protocols hand over

packets to the upper bundling layer, which ‘consumes’ them to ‘produce’ bundles. The bundles are routed through a deep space network that consists of gateways at the edge of different interplanetary regions. The IPN architecture defines a region as an Internet that shares the same networking protocols. The IPN is thus a ‘network of Internets’[6].

Application Layer
Bundling Layer
Convergence sub-layer
Data link layer
Licklider Transmission Protocol (sub-layer)
CCSDS Link Layer Protocol (sub-layer)
Physical Layer

Figure 1: Bundling architecture in the IPN backbone. The IPN uses an addressing scheme based on a tuple, which consists of a region ID(e.g. mars.sol) and a region-specific entity ID(e.g. ism.edu).

The entity ID of a tuple is not interpreted (e.g., bound to an address) outside its local region: a concept called late binding. The IPN backbone is concerned only with delivering the bundle to the destination region, without worrying about who the intended recipient is. The IPN gateway in the destination region ‘consumes’ the bundles, and hands over the packets to the Internet in the destination region, which routes the packet(s) to the recipient defined by the entity ID.

Bundling eliminates the chattiness that is characteristic of TCP/IP based protocols by including all the control information (metadata) needed at intermediate/destination nodes in the bundle itself. This enables connections to be set up without waiting for multiple round trip times(RTTs) exchanging SYN/ACK messages. The expectation that a bundle will be delivered is very high, so recipients are not expected to send acknowledgements, except in special cases. Mechanisms at the data link layer provide reliability through the use of strong Forward Error Correction (FEC) techniques along with a limited amount of retransmission. The point of retransmission is progressively moved forward towards the destination region by a mechanism called ‘custodial transfer’; when a node accepts custody of a bundle, it stores a copy of the bundle till the next custodian accepts custody and informs it, upon which it discards the bundle to free up its storage space.

When reasoning about what is desirable and possible in a DTN, it is often helpful to consider the analogy[9] of living in England in the mid-19th century, before telephones or even widely available telegraphy. The only way to communicate with friends or business associates is by letters, conveyed through the postal system. The postal system is very good - there are three or four mail deliveries per day in central London - but one never has absolutely up-to-the-minute information about *anything*. One is always reasoning from past experience and the most recent information, which is always hours or days old. This is exactly the condition in which delay-tolerant networks may often (though not always) need to function.

3. Congestion in DTNs

Congestion has been a key issue for the Internet community, right from the ARPANET days. It is often difficult to define congestion precisely, since it depends on a network's particular conditions, as well as users' expectations and perceptions.

The following symptoms [10] are generally associated with congestion in the Internet:

- The queuing delay of the data packets increases.
- There may be packet losses.
- Traffic is dominated by retransmissions, so that the effective data rate decreases.

These parameters are however inadequate to characterize congestion in a delay tolerant network. In a DTN, queuing delays are *expected*, because of the high latencies and intermittent connectivity. Further, paths are very lossy, so losses do not necessarily indicate congestion. Finally, bundling is designed to eliminate chattiness, so there is no question of traffic being dominated by retransmissions.

We believe that congestion in the InterPlanetary Internet is likely to be due to unexpected link failures, or data bursts from opportunistic contacts. Keshav has given a definition [10] of congestion based on economic theory, which appears to be suitable to characterize congestion in a DTN. According to this definition, a network is said to be congested from the perspective of user i if the utility of the network to i decreases due to an increase in network load. 'Utility' here refers to a user's preference for a resource, or a set of resources. Strictly speaking, the utility to a user is a number that represents the relative preference of that user for a resource or set of resources, so that if a user prefers A to B, the utility of A is greater than the utility of B.

In a DTN, nodes could *learn* to expect a particular level of utility(performance) from the network. The utility will have to be defined/inferred by the network. A reduction in the utility might be a possible indication of impending congestion.

4. Our solution: Structural flow control

Scott Burleigh has proposed a mechanism called structural flow control [9] to deal with congestion in DTNs. To understand this mechanism, let us consider the following analogy. Suppose all the links in some small neighborhood ad-hoc network are dial-up telephone links using cheap modems.

The owner of one node of the network, noticing that data are arriving too fast for his node to keep up with processing, might phone all his friends and ask them to switch their modems' transmission rates from 56 Kbps to 28.8 Kbps. If they all complied, then data would flow more slowly in the network even though there would be no additional TCP or IP flow control. In effect, aggregate transmission speed would have been "structurally" degraded – at the physical level, by human agency - rather than operationally or logically by protocol activity.

The notion of structural flow control is built on feedback loops similar to flow control built into TCP and IP. There are some important differences [9], however:

- The loops are potentially quite long and slow, not tight and fast as in TCP,
- The flow control information (commands and feedback) is carried at the bundling layer ("a higher level") rather than built into the transport protocol itself, and
- There may (or may not) be a human in the loop.

Structural flow control is based on the following principles:

- Autonomous cognition: Each node takes decisions independently of others, based only on its own input and output rates, and the condition of its buffers. A cognitive model helps the node take accurate decisions based on past experience.
- Decentralized operation: There is no central authority that regulates the network or influences its decisions. The network operates in a decentralized fashion like the Internet.
- Tolerance of imperfect information/mistakes: Imperfect information refers to delayed, incomplete or conflicting information. In the latter case, we expect that the program will be able to detect if a certain piece of information that has come in is wrong, based on past experience. Because of the imperfect nature of the information available about the network at any given moment, mistakes are inevitable. Nodes learn from past mistakes and try to minimize their occurrence. At the same time, they must be resilient to mistakes, by being prepared to deal with them.

Congestion and flow control are delegated to the bundling layer or the convergence sub-layer. The convergence sub-layer can force flow control by refusing to read on the input socket, and sending NACKs to the source; while the bundling layer can send quench packets to the source to ask it to reduce its sending rate. The aim is to decrease the amount of data entering the IPN backbone. In the event of congestion occurring in the DTN, nodes would have to start discarding bundles - resulting in NACKs - as an immediate step, and thereafter ask senders to reduce their sending rates after evaluating the possible impact of reduction in sending rates by individual senders and by different factors.

Nodes can ask certain senders to reduce their rates, while letting others continue sending at the previous rates. To accomplish this, they would have to run AI programs that search the set of possible solutions, and select the best, after analyzing the effect of various combinations. Simultaneously, they could use online machine learning programs that suggest which option(s) is better in the light of past experience. In this context, we wouldn't want hardware limitations to constrain our thinking unduly, because although space-qualified processors will likely always be slower than their terrestrial counterparts, they too are getting faster and more capable.

Cascaded buffer exhaustion, caused by backward propagation of hop by hop flow control[11], can possibly be avoided by using some form of Random Early Detection[12], which will be an integral part of the structural flow control mechanism.

Currently, all interplanetary communication devices are radios either mounted on spacecraft (including rovers, etc.) or installed in the tracking stations of the Deep

Space Network[13] and other tracking systems. Links are highly directional, so these devices - analogous to the hosts and routers of the Internet - have to be actively managed: the antennae of the spacecraft and tracking stations must be physically moved (pointed) and the radios powered on and off on an intricate and detailed schedule in order for data to flow. The transmission capacity of each such radio is known in advance, so each individual communication opportunity has a known duration and data rate; hence capacity. The data to be transmitted in each opportunity are carefully selected so as not to exceed that capacity. The aggregate effect of all of this planning and management is, in effect, structural flow control[9]. The future growth of capacity in the Interplanetary Internet will make some of this management infeasible: pointing and scheduling are unavoidable, but today's manual negotiation of schedule and selection of the data to be transmitted in each opportunity will have to be supplanted by automation. That automated management will similarly be structural flow control[9].

4.1 Rationale for a cognition-based approach

Traditional algorithmic approaches fail in the face of imperfect information, because the complete information that they need to take correct decisions is not available. On the other hand, machine learning programs often provide results with a high degree of accuracy once they have been trained adequately, even when the input data is incomplete.

We propose to utilize this property of machine learning programs of taking accurate decisions even with imperfect information to detect the likelihood of congestion in a DTN. We believe that this approach is especially relevant to DTNs, where information is always delayed and often incomplete. To keep the network running at an acceptable efficiency, nodes in DTNs would need to assess the likelihood of congestion, by analyzing the typical data at the node such as input rates, output rates, and buffer availability, etc. The congestion detection daemon running at the node could suggest a numerical figure that indicates the likelihood of congestion. This would enable the node to take appropriate measures, such as discarding bundles, rejecting custody, or asking a sending node to reduce its sending rate.

Taking flow control decisions is especially tricky in DTNs because the decision is to be based on old information, and the effects of the decision will often not be visible until some significant amount of time in the future. This dual disadvantage in time implies that a node must have the ability to explore plausible scenarios - similar to a chess program - and suggest the best course of action in the light of available information. Furthermore, the need to minimize human intervention requires the ability to learn from both successes and failures, and evolve mechanisms to diagnose and fix problems in the network [14]. These considerations strongly support the choice for an AI based approach to congestion and flow control.

5. Implementation

This section includes the C program used to generate the typical data available at a DTN node such as input rate, output rate and available buffer space, which, along with their derivatives are used to predict the level of congestion in a DTN. The program is preceded by a brief discussion of parameters used, and a description of terms used in the program.

5.1 Discussion of Parameters

Our approach entails that a node takes flow control decisions based on its local information - the input rate, output rate and available buffer space at the node. The possibility of congestion is indicated by the cognition based congestion detection daemon on a scale from 1 to 5 (which could be different for different situations and environments) -- where 1 and 5 indicate the least and most likelihood of impending congestion.

In this context, three parameters look distinct and important, in deciding the level of congestion warning :

1. Available buffer space. The congestion level indicated by the daemon will depend on the available buffer space. To understand why, let us assume that there are 5 senders, and the bandwidth-delay product for each of these links is 200 . This means that if there is a data burst on all these links simultaneously (worst-case scenario), then a minimum of $200*5 = 1000$ MB will be used up before the flow control command becomes effective in throttling the flow of data from the senders. A very cautious approach could be to always keep 1000 MB free, keeping the worst case scenario in mind. But a worst case scenario is not always likely; its probability is generally too low. Hence, lesser amount of buffer can be reserved free (say 500). This value will be particular to the specific conditions in a network. It of course implies taking a risk; but the gain is better utilisation of the link bandwidth.

Effectively, the ratio of the Max BW-Delay product of all sources to the buffer space at a node should be a key factor in determining the different warning levels at different levels of buffer occupancy.

2. Change in Input rate: The change in input rate, over fixed intervals of time(the slope of the input rate v/s time curve, effectively -- though scaled down) could give a warning of a burst. Suppose that the input rate suddenly goes up by 64 Kbps on a link where the maximum data rate is 128 Kbps. Now, if this is unusual, it could mean that a burst is coming in. Similarly, a sudden decrease could mean that the node can 'relax' and reduce the congestion warning level.

3. Time for available buffer space to fill up, at the present rate of consumption: This parameter is given by:

$$\text{Time_for_buffer_fill} = \text{Available_buffer_space}/(\text{Input_rate}-\text{Output_rate})$$

This is an important parameter, because it indicates *how imminent* the threat of buffer exhaustion is. For example, if a node realises that the buffer fill up time is less

that the RTT to its sender, it could mean an imminent loss of bundles. Again, the node may find from simple calculations that its buffer will be exhausted before the next contact to a node that accepts bundles from it is scheduled, in which case it would have to send a command to its sender to reduce the sending rate by a particular factor/amount.

A large available buffer space alongwith minor change in input rate will give a ‘low’ congestion indication, but it may so happen that the buffer is being used up at a significantly high rate. In such situations particularly, this parameter is expected to play a key role in detecting the likelihood of congestion in a DTN.

5.1.1 Terms used:

Input_Rate: Input rate at a particular node

Output_Rate: Output rate at the node

Buffer_MB: Buffer space available at the node(in Megabytes)

Buffer: Buffer space available at the node in bytes

Last_Input_Rate: The value of Input_Rate at most recently when congestion level was predicted

Change_in_Input_Rate: (Input_Rate – Last_Input_Rate)

Time_to_Buffer_Full: Time period in which buffer will fill up at the current rates.

Time_to_Buffer_Full_Index, **Change_in_Input_Rate_Index** and **Buffer_Index** are indexes (on a scale from 1 to 6) which which respectively indicate how imminent buffer exhaustion is, if a data burst is coming in, and amount of free buffer available.

Congestion_Index gives the level of congestion as a simple or weighted average of the above three indexes on a scale of 1 (least chance of congestion) to 5 (congestion is highly imminent). Respective weights for calculating weighted average are **t1**, **i1** and **b1**.

5.2 C Program

```
#include<iostream.h>
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<time.h>
#include<dos.h>
#include<fstream.h>
#include<math.h>
```

```

int select_range(long double x);
long double Buffer_Index, Buffer_;
long double Change_in_Input_Rate_Index;
int count1=0, count2=0, count3=0, count4=0, count5=0;
int t1=1,i1=2, b1=1;
void main()
{
    clrscr();
    int i, range;
    long double Input_Rate, Output_Rate, Time_to_Buffer_Full;
    randomize();
    long double Last_Input_Rate=0;
    int No_of_Loops=2000;
    ofstream out("cong/result");          //Send output to file
    'result' in cong folder

    //ARFF file formatting data
    out<<"@relation
Predicting Levels of Congestion"<<endl<<endl;
    out<<"@attribute Change_in_Input_Rate real"<<endl;
    out<<"@attribute Buffer real"<<endl;
    out<<"@attribute Time_for_Buffer_Fill real"<<endl;
    out<<"@attribute Level_of_congestion {Congestion_Level_1,
Congestion_Level_2, Congestion_Level_3, Congestion_Level_4,
Congestion_Level_5}"<<endl;
    out<<endl<<"@data"<<endl;

    for(i=0; i<No_of_Loops; i++)
    {
        Input_Rate=(rand()%100)*1000; //Varies from 10bps to
128Kbps
//Deep Space Link Modelling
        long double Change_in_Input_Rate=Input_Rate-
Last_Input_Rate;
        if(Change_in_Input_Rate>-32000 && Change_in_Input_Rate<=-
20000) Change_in_Input_Rate_Index=1;
        if(Change_in_Input_Rate>-20000 && Change_in_Input_Rate<=-
10000) Change_in_Input_Rate_Index=2;
        if(Change_in_Input_Rate>-10000 &&
Change_in_Input_Rate<=0) Change_in_Input_Rate_Index=3;
        if(Change_in_Input_Rate>0 && Change_in_Input_Rate<=10000)
Change_in_Input_Rate_Index=4;
        if(Change_in_Input_Rate>10000 &&
Change_in_Input_Rate<=20000) Change_in_Input_Rate_Index=5;
        if(Change_in_Input_Rate>20000 &&
Change_in_Input_Rate<=32000) Change_in_Input_Rate_Index=6;
// out<<Input_Rate<<endl;
// out<<Change_in_Input_Rate_Index<<endl;
// cout<<Change_in_Input_Rate_Index<<endl;

        Time_to_Buffer_Full=(rand()%100)*1000; //Minimum value
is 250 seconds

        if(Time_to_Buffer_Full==0)
        {
            Time_to_Buffer_Full=Time_to_Buffer_Full+250;

```

```

    }
        if(Time_to_Buffer_Full<0)
    {
Time_to_Buffer_Full=Time_to_Buffer_Full*(-1);
    }

Buffer_=(rand()%40)*1000;
if(Buffer_<0) Buffer_=Buffer_*(-1);
if(Buffer_==0) Buffer_++;
if(Buffer_<=5000) Buffer_Index=6;
if(Buffer_>5000 && Buffer_<=10000) Buffer_Index=5;
if(Buffer_>10000 && Buffer_<=15000) Buffer_Index=4;
if(Buffer_>15000 && Buffer_<=20000) Buffer_Index=3;
if(Buffer_>20000 && Buffer_<=25000) Buffer_Index=2;
if(Buffer_>25000) Buffer_Index=1;

long double Buffer=Buffer_*1000000;

Output_Rate=Input_Rate-(Buffer/Time_to_Buffer_Full);

if(Time_to_Buffer_Full>0)
{
range=select_range(Time_to_Buffer_Full);

    if(count1<=1500)
    {
        if(range<=6)
        {

cout<<Input_Rate<<","<<Change_in_Input_Rate<<","<<Output_Rate<<
<","<<Buffer<<","<<Time_to_Buffer_Full<<","Congestion_Level_1"<<
<endl;

out<<Input_Rate<<","<<Change_in_Input_Rate<<","<<Output_Rate<<
","<<Buffer<<","<<Time_to_Buffer_Full<<","Congestion_Level_1"<<
endl;

                count1++;
            }
        }

    if(count2<=1500)
    {
        if(range>6 && range<=9)
        {

cout<<Input_Rate<<","<<Change_in_Input_Rate<<","<<Output_Rate<<
<","<<Buffer<<","<<Time_to_Buffer_Full<<","Congestion_Level_2"<<
<endl;

out<<Input_Rate<<","<<Change_in_Input_Rate<<","<<Output_Rate<<
","<<Buffer<<","<<Time_to_Buffer_Full<<","Congestion_Level_2"<<
endl;

                count2++;
            }
        }
    }
}

```

```

        if(count3<=1500)
        {
            if(range>9 && range<=12)
            {

cout<<Input_Rate<<" , "<<Change_in_Input_Rate<<" , "<<Output_Rate<<
<" , "<<Buffer<<" , "<<Time_to_Buffer_Full<<" , Congestion_Level_3"<<
<endl;

out<<Input_Rate<<" , "<<Change_in_Input_Rate<<" , "<<Output_Rate<<
" , "<<Buffer<<" , "<<Time_to_Buffer_Full<<" , Congestion_Level_3"<<
endl;

                count3++;
            }
        }

        if(count4<=1500)
        {
            if(range>12 && range<=15)
            {

cout<<Input_Rate<<" , "<<Change_in_Input_Rate<<" , "<<Output_Rate<<
<" , "<<Buffer<<" , "<<Time_to_Buffer_Full<<" , Congestion_Level_4"<<
<endl;

out<<Input_Rate<<" , "<<Change_in_Input_Rate<<" , "<<Output_Rate<<
" , "<<Buffer<<" , "<<Time_to_Buffer_Full<<" , Congestion_Level_4"<<
endl;

                count4++;
            }
        }

        if(count5<=1500)
        {
            if(range>15 && range<=18)
            {

cout<<Input_Rate<<" , "<<Change_in_Input_Rate<<" , "<<Output_Rate<<
<" , "<<Buffer<<" , "<<Time_to_Buffer_Full<<" , Congestion_Level_5"<<
<endl;

out<<Input_Rate<<" , "<<Change_in_Input_Rate<<" , "<<Output_Rate<<
" , "<<Buffer<<" , "<<Time_to_Buffer_Full<<" , Congestion_Level_5"<<
endl;

                count5++;
            }
        }
        else i--;

        Last_Input_Rate=Input_Rate;

    }
    else i--;

```

```

    }

    getch();
    return;
}

int select_range(long double x)
{
    long double Time_to_Buffer_Full_Index;
    int Congestion_Index;
    //X HERE IS TIME FOR BUFFER FULL
    if(x<=5000) Time_to_Buffer_Full_Index=6;
    if(x>5000 && x<=10000) Time_to_Buffer_Full_Index=5;
    if(x>10000 && x<=15000) Time_to_Buffer_Full_Index=4;
    if(x>15000 && x<=20000) Time_to_Buffer_Full_Index=3;
    if(x>20000 && x<=25000) Time_to_Buffer_Full_Index=2;
    if(x>25000) Time_to_Buffer_Full_Index=1;
    //    ofstream out("cong/12");
    //    cout<<Time_to_Buffer_Full_Index<<endl;
    Congestion_Index=(int)
    (Time_to_Buffer_Full_Index*t1+Change_in_Input_Rate_Index*i1+Bu
    ffer_Index*b1)*3/(t1+i1+b1);
    return Congestion_Index;
}

```

5.3 Weka, an open source implementation of C4.5[15], Quinlan's powerful inductive machine learning software has been used for testing the efficiency of certain machine learning programs in detecting congestion in a DTN. The choice of a machine learning method is dictated by the amount of information available about a system[16]. The decision to select inductive machine learning is motivated by the fact that this method works even when very little information is available about the system. The algorithms selected for testing from amongst the large number of available ones are:

- **ZeroR:** The ZeroR algorithm simply predicts the majority class (“neutral”) in the training data and thus is used as a performance benchmark for the other algorithms.
- **OneR:** The OneR algorithm produces very simple rules based on a single attribute when given a set of data.
- **DecisionTable:** The Decisiontable algorithm associate conditions with actions to perform like if-then-else and switch-case statements. It can associate many independent conditions with several actions in an elegant way.

The experiment was run a number of times, by varying the parameters that alter traffic flow, to evaluate the effectiveness of the daemon. The results are discussed in the next section.

6. Results and analysis:

6.1 Sample Weka Explorer output:

6.1.1 Evaluation on ZeroR with 33% training data and 67% test data from 2000 cases

==== Run information ====

Scheme: weka.classifiers.rules.ZeroR
Relation: Predicting_Levels_of_Congestion
Instances: 2000
Attributes: 4

Change_in_Input_Rate
Buffer
Time_for_Buffer_Fill
Level_of_congestion

Test mode: split 33% train, remainder test

==== Classifier model (full training set) ====

ZeroR predicts class value: Congestion_Level_2

Time taken to build model: 1.59 seconds

==== Evaluation on test split ====

==== Summary ====

Correctly Classified Instances	438	32.6866 %
Incorrectly Classified Instances	902	67.3134 %
Kappa statistic	0	
Mean absolute error	0.2926	
Root mean squared error	0.384	
Relative absolute error	100 %	
Root relative squared error	100 %	
Total Number of Instances	1340	

==== Detailed Accuracy By Class ====

TP Rate	FP Rate	Precision	Recall	F-Measure	Class
0	0	0	0	0	Congestion_Level_1
1	1	0.327	1	0.493	Congestion_Level_2
0	0	0	0	0	Congestion_Level_3
0	0	0	0	0	Congestion_Level_4
0	0	0	0	0	Congestion_Level_5

==== Confusion Matrix ====

a b c d e <-- classified as
0 269 0 0 0 | a = Congestion_Level_1

```
0 438 0 0 0 | b = Congestion_Level_2
0 430 0 0 0 | c = Congestion_Level_3
0 169 0 0 0 | d = Congestion_Level_4
0 34 0 0 0 | e = Congestion_Level_5
```

6.1.2 Result of 10-fold cross validation:

==== Run information ====

```
Scheme: weka.classifiers.rules.ZeroR
Relation: Predicting_Levels_of_Congestion
Instances: 2000
Attributes: 4
    Change_in_Input_Rate
    Buffer
    Time_for_Buffer_Fill
    Level_of_congestion
Test mode: 10-fold cross-validation
```

==== Classifier model (full training set) ====

ZeroR predicts class value: Congestion_Level_2

Time taken to build model: 0 seconds

==== Stratified cross-validation ====

==== Summary ====

Correctly Classified Instances	681	34.05 %
Incorrectly Classified Instances	1319	65.95 %
Kappa statistic	0	
Mean absolute error	0.2926	
Root mean squared error	0.3825	
Relative absolute error	100.0029 %	
Root relative squared error	100.0003 %	
Total Number of Instances	2000	

==== Detailed Accuracy By Class ====

TP Rate	FP Rate	Precision	Recall	F-Measure	Class
0	0	0	0	0	Congestion_Level_1
1	1	0.341	1	0.508	Congestion_Level_2
0	0	0	0	0	Congestion_Level_3
0	0	0	0	0	Congestion_Level_4
0	0	0	0	0	Congestion_Level_5

==== Confusion Matrix ====

a b c d e <-- classified as

```
0 381 0 0 0 | a = Congestion_Level_1
0 681 0 0 0 | b = Congestion_Level_2
0 626 0 0 0 | c = Congestion_Level_3
0 268 0 0 0 | d = Congestion_Level_4
0 44 0 0 0 | e = Congestion_Level_5
```

6.1.3 Evaluation on OneR with 33% training data and 67% test data from 2000 cases

==== Run information ====

```
Scheme: weka.classifiers.rules.OneR -B 6
Relation: Predicting_Levels_of_Congestion
Instances: 2000
Attributes: 4
    Change_in_Input_Rate
    Buffer
    Time_for_Buffer_Fill
    Level_of_congestion
Test mode: split 33% train, remainder test
```

==== Classifier model (full training set) ====

```
Change_in_Input_Rate:
  < -57268.0 -> Congestion_Level_3
  < -53268.0 -> Congestion_Level_4
  < -50268.0 -> Congestion_Level_2
  < -47304.0 -> Congestion_Level_4
  < -44268.0 -> Congestion_Level_3
  < -42768.0 -> Congestion_Level_4
  < -41268.0 -> Congestion_Level_2
  < -39268.0 -> Congestion_Level_3
  < -32268.0 -> Congestion_Level_2
  < -23768.0 -> Congestion_Level_1
  < -23232.0 -> Congestion_Level_2
  < -19768.0 -> Congestion_Level_1
  < -18232.0 -> Congestion_Level_2
  < -16768.0 -> Congestion_Level_1
  < -14768.0 -> Congestion_Level_2
  < -13768.0 -> Congestion_Level_1
  < 232.0 -> Congestion_Level_2
  < 3232.0 -> Congestion_Level_3
  < 3768.0 -> Congestion_Level_2
  < 4768.0 -> Congestion_Level_3
  < 5232.0 -> Congestion_Level_2
  < 6232.0 -> Congestion_Level_3
  < 7232.0 -> Congestion_Level_2
  < 8232.0 -> Congestion_Level_3
  < 10232.0 -> Congestion_Level_2
  < 20232.0 -> Congestion_Level_3
```

< 20500.0 -> Congestion_Level_4
< 23232.0 -> Congestion_Level_3
< 25268.0 -> Congestion_Level_4
< 25768.0 -> Congestion_Level_3
< 31768.0 -> Congestion_Level_4
< 33268.0 -> Congestion_Level_3
< 37304.0 -> Congestion_Level_1
< 38768.0 -> Congestion_Level_3
< 49768.0 -> Congestion_Level_2
< 55500.0 -> Congestion_Level_3
>= 55500.0 -> Congestion_Level_2
(1059/2000 instances correct)

Time taken to build model: 0.77 seconds

6.1.4 Evaluation on DecisionTable with 33% training data and 67% test data from 2000 cases

==== Run information ====

Scheme: weka.classifiers.rules.DecisionTable -X 1 -S 5
Relation: Predicting_Levels_of_Congestion
Instances: 2000
Attributes: 4
Change_in_Input_Rate
Buffer
Time_for_Buffer_Fill
Level_of_congestion
Test mode: split 33% train, remainder test

==== Classifier model (full training set) ====

Decision Table:

Number of training instances: 2000
Number of Rules : 96
Non matches covered by Majority class.
Best first search for feature set,
terminated after 5 non improving subsets.
Evaluation (for feature selection): CV (leave one out)
Feature set: 1,2,3,4

Time taken to build model: 1.65 seconds

==== Evaluation on test split ====

==== Summary ====

Correctly Classified Instances	795	59.3284 %
--------------------------------	-----	-----------

Incorrectly Classified Instances 545 40.6716 %
 Kappa statistic 0.435
 Mean absolute error 0.1819
 Root mean squared error 0.3151
 Relative absolute error 62.1819 %
 Root relative squared error 82.0764 %
 Total Number of Instances 1340

=== Detailed Accuracy By Class ===

TP Rate	FP Rate	Precision	Recall	F-Measure	Class
0.472	0.077	0.605	0.472	0.53	Congestion_Level_1
0.747	0.318	0.533	0.747	0.622	Congestion_Level_2
0.551	0.133	0.662	0.551	0.602	Congestion_Level_3
0.497	0.044	0.618	0.497	0.551	Congestion_Level_4
0.588	0.002	0.909	0.588	0.714	Congestion_Level_5

=== Confusion Matrix ===

```

a b c d e <-- classified as
127 128 12 2 0 | a = Congestion_Level_1
64 327 33 14 0 | b = Congestion_Level_2
15 151 237 27 0 | c = Congestion_Level_3
4 8 71 84 2 | d = Congestion_Level_4
0 0 5 9 20 | e = Congestion_Level_5
  
```

6.2 Analysis

The performance of ZeroR, OneR and DecisionTable Classifiers for 1000, 2000 and 4000 instances and 15%, 33%, 66% and 90% training split is given in Tables 1, 2 and 3 respectively.

Table 1: Performance of ZeroR algorithm with varying number of training and test cases

Classifier	Number of instances	% of test instances classified correctly for different % of total instances used for training			
		15% training	33% training	66% training	90% training
ZeroR	1000	33.64	34.77	32.94	26
ZeroR	2000	36.58	36.94	37.79	38.5
ZeroR	4000	34	33.91	33.89	34

Table 2: Performance of OneR algorithm with varying number of training and test cases

Classifier	Number of instances	% of test instances classified correctly for different % of total instances used for training
------------	---------------------	---

		15% training	33% training	66% training	90% training
OneR	1000	43.29	43.28	43.82	48
OneR	2000	42.52	42.16	42.2	42
OneR	4000	40.17	42.27	41.76	43

Table 3: Performance of DecisionTable algorithm with varying number of training and test cases

Classifier	Number of instances	% of test instances classified correctly for different % of total instances used for training			
		15% training	33% training	66% training	90% training
DecisionTable	1000	34	62.38	62.64	77
DecisionTable	2000	59.82	67.16	72.79	77.5
DecisionTable	4000	60.52	70.59	74.7	83.75

As is evident from the table, there is a certain ideal training limit, beyond which the Classifier gets overtrained and its performance drops, particularly for the ZeroR and OneR cases. Within the limit, the performance increases with the number of training instances.

It can also be seen that overall, the Decision Table Classifier performs better than the OneR classifier, which in turn performs better than the ZeroR classifier (measured as a function of the % of instances classified correctly). However the number of training cases needed to reach the peak performance is least in the case of the ZeroR classifier and most in the case of DecisionTable.

This result is understandable, considering the different nature of the three classifiers. The ZeroR algorithm simply predicts the majority class (“neutral”) in the training data and thus is used as a performance benchmark for the other algorithms. The OneR algorithm, on the other hand produces very simple rules based on a single attribute when given a set of data. As its name suggests, this system learns one rule. In some circumstances it is almost as powerful as sophisticated systems. Finally, decision tables are a precise, yet compact way to model complicated logic. Like if-then-else and switch-case statements, they associate conditions with actions to perform. But, unlike the control structures found in traditional programming languages, decision tables can associate many independent conditions with several actions in an elegant way.

Table 4: Result of 5 Iterations using 33% training in 2000 instances for the three algorithms

Number of instances	% of test instances classified correctly		
	ZeroR	OneR	DecisionTable
2000	34.92	40.59	62.31

2000	32.31	42.08	67.91
2000	33.05	41.04	66.04
2000	35.22	42.01	63.73
2000	32.68	41.11	65.29
Mean	33.636	41.366	65.056
Standard Deviation	1.339	0.652	2.148

From repeated tests keeping the number of instances fixed at 2000, we see that the performance of any of the three classifiers is pretty much consistent, with a maximum standard deviation of 2.148 in the case of DecisionTable, which is small, considering the numbers involved.

Table 5: Testing by percentage split with training data and 2000 total instances for different attributes supplied to Classifier

Attributes supplied to Classifier	Correctly Classified Instances (in %)		
	ZeroR	OneR	DecisionTable
Input_Rate Output_Rate Buffer	30.29	32.31	35.40
Change_in_Input_Rate Buffer Time_for_Buffer_Full	32.68	45.67	59.32
Input_Rate Change_in_Input_Rate Output_Rate Buffer Time_for_Buffer_Full	31.04	49.55	68.43

We find that even when only the raw data, i.e. Input rate, output rate and available buffer is supplied to the program, the classifiers each predict 30% of their test cases correctly. Supplying the Change_in_Input_Rate and Time_for_Buffer_Full values leads to appreciable increase in the performance. Finally, when all the attributes are fed to the program, a peak performance of 68.43% is achieved by the decision table. From iterations (shown in table 6), the last values are found to be quite consistent.

Table 6: Result of 5 iterations for 5 attributes case discussed above with 33% training data and 2000 total instances

Attempts	ZeroR	OneR	Decisiontable
2000	31.04	49.55	68.43
2000	33.88	50.89	63.73
2000	33.50	50.52	68.13
2000	31.26	50.67	70.59
2000	31.11	52.46	71.41
Mean	32.158	50.818	68.45

Table 7: Comparison of percentage split: 33% test and 10-fold cross validation for 2000 total instances

Test options	Correctly Classified Instances (in %)		
	ZeroR	OneR	DecisionTable
Percentage Split(33%)	32.68	45.67	59.32
Crossvalidation	34.05	46.60	74.55

It can be seen that crossvalidation gives significant improvement in the case of DecisionTable.

7. Conclusion and future work:

The results from this project confirm our belief that a machine learning program can be useful in detecting congestion in a DTN. We actually modelled the worst case scenario, by randomly varying the different input parameters to the daemon. An actual network is expected to be much more predictable in its behaviour than what we have modelled here. Hence, a machine learning program would be expected to take decisions with significantly better accuracy when it is fed with real-time data from a DTN.

DTNs, like the InterPlanetary Internet are still in the planning and conceptual stage of formation. So as such, there isn't a network from which real data can be obtained (other than from space missions whose data is generally classified). Future experiments, such as rigorous network simulations of DTNs will provide a platform to check the validity of our assumptions and inferences.

Scott Burleigh has suggested[9] using data abandonment rate at a node as a parameter to measure the level of congestion which deserves to be investigated. Prof Leonard Kleinrock has suggested[17] that bundles could carry information about congestion and buffer status and network parameters regarding all nodes it has passed through on its way to their destination. The challenge here would be how to describe the data, and use it. Current attempts at 'cognitive networking' using a Knowledge Plane [18] will provide new ways to represent knowledge, and hence inspire novel knowledge-based approaches to congestion in a DTN.

8. References

- [1] Delay Tolerant Networking Research Group, <http://www.dtnrg.org>.
- [2] InterPlanetary Internet Special Interest Group, <http://ipnsig.org>.
- [3] Hooke, A. *Towards an Interplanetary Internet: A Proposed Strategy for Standardization*, SpaceOps 2002, 9-12 October 2002, Houston, Texas.
- [4] Burleigh, S., Cerf, V., Durst, R., Fall, K., Hooke, A., Scott, K., and Weiss, H. *The Interplanetary Internet: A Communications Infrastructure for Mars Exploration*, 53rd International Astronautical Congress, The World Space Congress - 2002, 10-19 October 2002, Houston, Texas.
- [5] Burleigh, S., Cerf, V., Durst, R., Fall, K., Hooke, A., Scott, K., Torgerson, L., and Weiss, H. *Bundle Layer Protocol Specification v0.4*, work in progress, Sept 2002.
- [6] Burleigh, S., Hooke, A., Torgerson, L., Fall, K., Cerf, V., Durst, R., Scott, K., and Weiss, H. *Delay-Tolerant Networking: An Approach to Interplanetary Internet*, IEEE Communications Magazine, June 2003, vol. 41 no. 6, pp 128-136.
- [7] Fall, K. *A Delay-Tolerant Network Architecture for Challenged Internets*, IRB-TR-03-003, Feb 2003.
- [8] Daknet project, <http://www.daknet.org>.
- [9] Burleigh, S. Personal communication.
- [10] Keshav, S. *Congestion Control in Computer Networks*, PhD thesis, University of California, Berkeley, August 1991.
- [11] Burleigh, S. *Flow control/congestion control/resource management in DTNs*, dtn-interest mailing list, 16 March 2003.
- [12] Floyd, S., and Jacobson, V. *Random Early Detection gateways for Congestion Avoidance V.1 N.4*, August 1993, p. 397-413.
- [13] Deep Space Network, <http://dsn.nasa.gov>.
- [14] Clark, D., Partridge, C., Ramming, J., and Wroclawski, J. *A Knowledge Plane for the Internet*, SIGCOMM 2003.
- [15] *Weka* homepage, <http://www.cs.waikato.ac.nz/ml/weka/index.html>.
- [16] Blurock, E. *ANALYSIS: Notes in Machine Learning -- What is Machine Learning?* From <http://www.risc.uni-linz.ac.at/people/blurock/>.
- [17] Kleinrock, L. Personal communication
- [18] Knowledge Plane homepage, <http://www.isi.edu/know-plane/>.