http://researchcommons.waikato.ac.nz/

Research Commons at the University of Waikato

Copyright Statement:

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

The thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author’s right to be identified as the author of the thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author’s permission before publishing any material from the thesis.
Techniques for Failure Recovery in a Software-Defined Network

A thesis submitted in fulfillment of the requirements for the degree of Master of Science in Computer Science at The University of Waikato by Christopher Lorier

Department of Computer Science
Hamilton, New Zealand
August 18, 2014

© 2014 Christopher Lorier
Abstract

As our lives become ever more dependent on network connectivity, it becomes increasingly more important for networks to be able to overcome the failure of individual components and continue to function. This thesis examines approaches to fault tolerance in software defined networks, and how the global viewpoint that Software Defined Networking (SDN) provides can be leveraged to create more reliable networks.

In order to continue operation after the failure of a network component, the failure must first be detected, and then the network must automatically change its behaviour to mitigate any adverse consequences. This thesis evaluates a variety of fault detection methods and potential responses. Based on these evaluations the design for a fault tolerance system for software defined networks is presented. This system builds protected paths using Ring Based Forwarding, an algorithm for creating a full mesh of paths between switches in a network where each path has a fail-over path at each hop. The system monitors the network for faults using Traffic Colouring, a technique for passively monitoring network traffic.
I would like to thank everyone that has helped me throughout this thesis—especially Brad Cowie, Shane Alcock, Brendon Jones, Richard Nelson, Josh Bailey and Perry Lorier.
Contents

1 Introduction .......................................................... 1
  1.1 Document Structure ............................................. 4

2 Background .......................................................... 7
  2.1 RouteFlow ......................................................... 7
    2.1.1 Distributed RouteFlow ..................................... 8
    2.1.2 Fault Tolerance ............................................. 8
  2.2 OpenFlow ......................................................... 9
    2.2.1 OpenFlow Versions ......................................... 9
    2.2.2 OpenFlow Ports ............................................ 10
    2.2.3 Flow Tables ............................................... 10
    2.2.4 Group Tables ............................................... 11
    2.2.5 OpenFlow Statistics ....................................... 12
    2.2.6 The OpenFlow Protocol ................................... 12
  2.3 OpenFlow Switch Implementations ............................... 13

3 Evaluation of Previous Work .................................. 15
  3.1 Fault Detection ................................................. 15
## Contents

3.1.1 Bidirectional Forwarding Detection .................................. 16
3.1.2 Automatic Test Packet Generation ................................. 17
3.1.3 Topology Aware Blackbox Monitoring ........................... 18
3.1.4 A Packet Based Method for Passive Performance Monitoring .................................................. 19
3.1.5 Realistic Passive Loss Measurement for High-Speed Networks ....................................................... 21
3.1.6 Summary of Existing Work in Fault Detection ................. 21
3.1.7 Conclusion ................................................................. 23
3.2 Fault Recovery ............................................................... 23
3.2.1 Method and Apparatus for Determining Multiple Minimally-Overlapping Paths ........................................... 24
3.2.2 FatTire: Declarative Fault Tolerance for Software-Defined Networks ................................................. 25
3.2.3 Network Architecture for Joint Failure Recovery and Traffic Engineering ........................................... 25
3.2.4 Automatic Failure Recovery for Software-Defined Networks ............................................................. 26
3.2.5 Summary of Existing Work in Fault Recovery ................. 27
3.2.6 Conclusion ................................................................. 27

4 Fault Detection ................................................................. 29
4.1 Traffic Colouring ............................................................. 29
4.1.1 Packet Labelling .......................................................... 30
4.1.2 Flow Tables ............................................................... 31
4.1.3 Locating Faults ........................................................... 34
4.1.4 Statistics Retrieval ....................................................... 35
4.1.5 Controller Module ....................................................... 35
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.6</td>
<td>Counter Accuracy</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>Test Packet Injection</td>
<td>38</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Packet Labels</td>
<td>39</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Sending Probes</td>
<td>39</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Switch Configuration</td>
<td>40</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Probe Timing and Statistics Requests</td>
<td>41</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Controller Module</td>
<td>42</td>
</tr>
<tr>
<td>4.2.6</td>
<td>Barriers to Implementation</td>
<td>42</td>
</tr>
</tbody>
</table>

### 5 Fault Response

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Forwarding Rings</td>
<td>44</td>
</tr>
<tr>
<td>5.2</td>
<td>Pairs of Minimally Overlapping Paths</td>
<td>48</td>
</tr>
<tr>
<td>5.3</td>
<td>Minimal Pairs of Minimally Overlapping Paths</td>
<td>55</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Algorithm i</td>
<td>56</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Algorithm ii</td>
<td>60</td>
</tr>
<tr>
<td>5.4</td>
<td>Ring Based Forwarding</td>
<td>62</td>
</tr>
</tbody>
</table>

### 6 Evaluation

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Fault Detection</td>
<td>67</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Speed of Detection</td>
<td>68</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Flow Table Requirements</td>
<td>70</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Controller Load Requirements</td>
<td>70</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Network Load</td>
<td>72</td>
</tr>
<tr>
<td>6.1.5</td>
<td>Locating Faults</td>
<td>72</td>
</tr>
<tr>
<td>6.1.6</td>
<td>Other Considerations</td>
<td>72</td>
</tr>
<tr>
<td>6.1.7</td>
<td>Conclusion</td>
<td>73</td>
</tr>
<tr>
<td>6.2</td>
<td>Fault Recovery</td>
<td>74</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Generation of Random Networks</td>
<td>74</td>
</tr>
</tbody>
</table>
6.2.2 Primary Path Length ........................................ 74
6.2.3 Alternate Path Length ................................. 77
6.2.4 Flow Table Entries ...................................... 79
6.2.5 Link Load .............................................. 85
6.2.6 Other Considerations .................................. 87
6.2.7 Conclusion ........................................... 88
6.3 Conclusion ............................................. 88

7 Conclusion ...................................................... 91
7.0.1 Contributions .......................................... 92
7.0.2 Future Work ......................................... 93

References .................................................... 95
List of Figures

5.1 A graph representing an example network. . . . . . . . . . . . . 48
5.2 Rings to vertex 10. . . . . . . . . . . . . . . . . . . . . . . . . . . 48
5.3 Step 1 of building paths to vertex 10 for the vertices on ring 4. . 49
5.4 Step 2 of building paths to vertex 10 for the vertices on ring 4. . 50
5.5 Step 3 of building paths to vertex 10 for the vertices on ring 4. . 50
5.6 The final paths to vertex 10 for the vertices on ring 4 . . . . . . 51
5.7 Vertex is its own subdestination . . . . . . . . . . . . . . . . . . . 54
5.8 Vertex uses parent’s subdestination . . . . . . . . . . . . . . . . 54
5.9 Vertex uses parent as its subdestination . . . . . . . . . . . . . . 54
5.10 Building paths to vertex 10 for the vertices on ring 3. . . . . . 57
5.11 The final paths to vertex 10 for the vertices on ring 3. . . . . . 57
5.12 Rings to vertex 10 rearranged to demonstrate vertex locations. . 61
5.13 Paths to vertex 10 for the vertices on ring 3. . . . . . . . . . . . . 61
6.1 Controller traffic per cycle. . . . . . . . . . . . . . . . . . . . . . 71
6.2 Mean primary path length in networks of 20 switches . . . . . . 75
6.3 Mean primary path length in networks of 100 switches . . . . . . 76
6.4 Mean primary path length in networks of 500 switches . . . . . . 76
List of Figures

6.5  Mean alternate path length in networks of 20 switches . . . . . 78
6.6  Mean alternate path length in networks of 100 switches . . . . 78
6.7  Mean alternate path length in networks of 500 switches . . . . 79
6.8  Mean flow table entries per switch in networks of 20 switches . 80
6.9  Mean flow table entries per switch in networks of 100 switches 80
6.10 Mean flow table entries per switch in networks of 500 switches 81
6.11 Max flow table entries per switch in networks of 20 switches . . 82
6.12 Max flow table entries per switch in networks of 100 switches . 82
6.13 Max flow table entries per switch in networks of 500 switches . 83
6.14 Flow table entries per switch in networks of 20 switches and 30 links . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
6.15 Flow table entries per switch in networks of 100 switches and 150 links . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
6.16 Flow table entries per switch in networks of 500 switches and 750 links . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
6.17 Paths per link in networks of 20 switches and 30 links . . . . 85
6.18 Paths per link in networks of 100 switches and 150 links . . . 86
6.19 Paths per link in networks of 500 switches and 750 links . . . 86
List of Algorithms

5.1  Forwarding rings algorithm pseudo-code  . . . . . . . . . . . . 45
5.2  Split Path Trees function pseudo-code  . . . . . . . . . . . . 47
5.3  Path Pairs algorithm pseudo-code  . . . . . . . . . . . . . . . 52
5.4  Overlapping paths algorithm pseudo-code  . . . . . . . . . . . 53
5.5  Shared Paths algorithm i pseudo-code.  . . . . . . . . . . . . 58
5.6  Shared Paths algorithm ii pseudo-code  . . . . . . . . . . . . 64
5.7  Forwarding for vertices without rings pseudo-code  . . . . . 65
Acronyms

**AFRO** Automatic Failure Recovery for OpenFlow

**ATPG** Automatic Test Packet Generation

**BFD** Bidirectional Forwarding Detection

**CDF** Cumulative Distribution Function

**CIDR** Classless Inter-Domain Routing

**DSCP** Differentiated Services Code Point

**IP** Internet Protocol

**IPv4** IP version 4

**LSP** Label Switched Path

**MPLS** Multiprotocol Label Switching

**OFP** OpenFlow Protocol

**OSPF** Open Shortest Path First

**QoS** Quality of Service
**Acronyms**

**RSTP** Rapid Spanning Tree Protocol  
**RTT** Round Trip Time  
**SDN** Software Defined Networking  
**SLA** Service Level Agreement  
**TC** Traffic Class  
**TCAM** Ternary Content-Addressable Memory  
**VC** Virtual Circuit  
**VLAN** Virtual Local Area Network  
**VM** Virtual Machine
High availability is one of the most fundamental requirements of any computer network. When there is a fault in a network component, such as a router or cable, it is important that the network is able to continue operation with as little disruption to the network’s users as possible. This requires detecting and reacting to faults as quickly as possible, and responding in a manner that minimises the reduction in performance of the network.

SDN is a new methodology for managing networks that shifts the control of a network from the devices that comprise that network to software running on a central server. This gives network operators an unprecedented level of control over the behaviour of their network and has generated a large amount of interest in the research community. However, in order to see widespread deployment SDN must be able to match or improve upon the reliability of existing networks.

The global viewpoint of the control software in a software defined network may allow new tools to be developed to improve the reliability of net-
works. This thesis explores potential methods of detecting and responding to faults that could be the basis for such tools.

The target application for this research is RouteFlow (Stringer, Fu, Lorier, Nelson, & Rothenberg, 2013), an open-source software defined routing architecture. RouteFlow controls a fabric of OpenFlow switches so that they behave as a single logical IP router. Aggregating switches into a single layer 3 device reduces configuration, simplifies the network structure and helps ensure that all the devices share a consistent policy. However, there is very little fault tolerance built into RouteFlow, making it unsuitable for deployment in a production environment. This thesis evaluates existing work in fault detection and recovery and presents a design for a fault tolerance system for RouteFlow based on the results of that evaluation. This system will allow RouteFlow to continue to function in the face of network failures with as close to normal performance as possible.

The primary aim of this system is to detect faults in the underlying network hardware that causes behaviour that is inconsistent with the network configuration. While software faults and misconfiguration are the causes of the majority of network faults it is also important to verify the integrity of the hardware. The most common faults in network hardware are complete failures of a device or link, but faults may also exhibit more esoteric symptoms such as partial packet loss, packet duplication or packet corruption and may only affect some packets but not others.

The evaluation of the system is based on the following requirements:

*Minimal Impact:* The system should not cause a significant reduction of performance to RouteFlow when running on a fault free network.

*Fast Response:* The speed at which the system discovers and responds to
network faults should be comparable to existing protocols such as OSPF (Goyal, Ramakrishnan, & Feng, 2003) or RSTP (IEEE, 2001). Faults should be detected within seconds and once the fault is discovered the operation of the network should be restored within milliseconds.

**Accurate:** The system must be able to detect as wide a variety of potential faults as possible with minimal false positives.

**Scalable:** The state requirement for each switch in the network should scale at a linear rate with the number of switches in the network and the load on the control software should be kept to a minimum.

**Minimal Configuration:** One of the advantages of an aggregated router is that multiple devices can be configured by configuring one virtual router. It is important that the fault tolerance system must not require onerous configuration that would undermine this advantage.

**Versatile:** The system must be able to function effectively regardless of the topology of the network.

An initial review of existing work was performed to identify potentially suitable methods of detecting and recovering from faults. Implementations of these methods were then created for further evaluation. Two methods of fault detection were implemented: *Traffic Colouring*, a method for passively monitoring traffic on a network by assigning packets a colour and comparing the number of packets of each colour seen at various points in the fabric; and *Test Packet Injection*, a method of detecting faults by injecting probe packets into the fabric. For fault recovery methods the initial review resulted in the decision to use protected Virtual Circuits (VCs). Three algorithms for building protected paths for the VCs were created and im-
implemented for further evaluation: *Path Pairs*, a method for building pairs of minimally overlapping paths between switches on a network; *Shared Paths*, a method for building pairs of minimally overlapping paths between switches with the added constraint that no switch will ever require more than two forwarding rules to send all traffic to a given destination; and *Ring Based Forwarding*, a method for building a path between each switch in a network to a given destination switch with redundancy at every hop.

Experiments were set up to further evaluate these methods and the results of these experiments were used to present a final design for a fault tolerance system for RouteFlow. This system uses ring based forwarding to build paths and traffic colouring to detect faults.

### 1.1 Document Structure

*Chapter 2* describes the RouteFlow architecture, the OpenFlow standard used by RouteFlow and the OpenFlow switch implementations used during the work carried out for this thesis.

*Chapter 3* examines existing work in fault detection and recovery to identify potential approaches that may be suitable for use with RouteFlow. The most promising approaches are selected for further evaluation.

*Chapter 4* gives a detail description of implementations of the approaches to fault detection chosen for further evaluation in Chapter 3.

*Chapter 5* describes implementations of the approaches to fault recovery created for this thesis. These implementations are based on approaches selected for further evaluation in Chapter 3.

*Chapter 6* evaluates the implementations of fault detection and recovery
described in Chapters 4 and 5. A final design of a fault tolerance system for RouteFlow based on the outcomes of these evaluations is presented.

Chapter 7 summarises the work undertaken for this thesis and its outcomes and potential future work is described.
In this chapter some of the prior work that this thesis builds upon is explained in detail. Firstly, the RouteFlow architecture is described, then the OpenFlow (McKeown et al., 2008) features relevant to this thesis are explained and finally the limitations of the OpenFlow switch implementations used during this thesis are described.

2.1 RouteFlow

RouteFlow is an open source software-defined IP routing architecture. It controls a fabric of OpenFlow switches to act as a single logical router. RouteFlow consists of a Linux Virtual Machine (VM) running routing software, for instance Quagga (Free Software Foundation, n.d.) or BIRD (Filip, Forst, Machek, Mares, & Zajicek, 2010). The ports of the VM are mapped to the external ports of switches in the fabric. When the switches receive control plane traffic on a port it is redirected to the corresponding port on the VM. The routing software on the VM uses this traffic to learn routes,
and these routes are installed into the VM’s routing table as though the VM had received the control plane traffic directly. RouteFlow then learns the routes from the VM’s routing table and modifies the switches so that the entire fabric emulates the forwarding behaviour of the VM.

### 2.1.1 Distributed RouteFlow

In order to control a fabric of multiple OpenFlow switches to behave as a single IP router, VCs are created between each switch to each other switch on the fabric (Lorier, 2013). When data plane traffic arrives at a switch that matches a route that leaves the fabric on another switch, the traffic is forwarded through the VC to the correct egress switch where it is output. The VCs are implemented as Label Switched Paths (LSPs) using Multiprotocol Label Switching (MPLS) labels. Virtual Local Area Network (VLAN) tags are used if the switches in the fabric do not fully support MPLS operations. The VC paths follow the shortest path between the switches, calculated using Dijkstra’s algorithm.

In order to reduce the number of flow table entries required to implement these paths, the LSPs share labels whenever possible. So that all traffic for a particular destination exiting any switch will share the same MPLS label whether it originated at that switch or arrived at that switch as part of a VC. The next switch in the path only needs to have one flow table entry to match all traffic for each destination, rather than a separate entry for every VC.
2.1.2 Fault Tolerance

The fault tolerance of RouteFlow when controlling a multiple switch fabric is very poor. The only fault in the fabric that RouteFlow is able is when the controller loses the connection to one of the switches. This represents a small proportion of the possible faults that the system should be capable of dealing with. Other faults, such as link cuts, will never be detected and will result in all traffic that is sent over that link to be lost.

RouteFlow responds to the loss of the connection to a switch by setting every port on the VM associated with that switch to be down and then recalculates the path taken by every VC that passes through that switch and installs the new paths onto the switches. This is a very slow way of correcting the issue as every switch on every affected path needs to be updated before the traffic can be changed. There may be many affected paths and each is repaired one at a time.

2.2 OpenFlow

OpenFlow (McKeown et al., 2008) is a standard for the control of switches in a software defined network. It includes a specification (Open Networking Foundation, 2013) for switches to allow them to be controlled remotely in software with use of the OpenFlow Protocol (OFP). The remote software is referred to as the controller.

In this section the features of the OpenFlow standard that are relevant to this thesis are described.
2.2.1 OpenFlow Versions

RouteFlow supports OpenFlow version 1.2 (Open Networking Foundation, 2011), but at the time of writing the latest OpenFlow version is 1.4 (Open Networking Foundation, 2013). Because most of the features used by RouteFlow are compatible between versions 1.4 and 1.2 the work carried out as part of this thesis uses OpenFlow version 1.4.

2.2.2 OpenFlow Ports

Ports on OpenFlow switches can be physical or logical ports. Each port collects statistics on the traffic passing through it and has a state. The state is changed by the switch and includes the OFPPS_LIVE flag, which is used with fail-over group tables, and the OFPPS_LINK_DOWN flag which is used to indicate that the physical link is not present. The port is live when the OFPPS_LINK_DOWN flag is not set, but can also be disabled by code running on the switch not specified by the OpenFlow standard. This allows the switch to support protocols such as Bidirectional Forwarding Detection (BFD) (Katz & Ward, 2010). Whenever the port state changes the controller is notified with a OFPT_PORT_STATUS message.

2.2.3 Flow Tables

The packet handling behaviour of OpenFlow switches is defined by adding entries to the switches’ flow tables. Flow table entries have a priority value, a cookie value, matches, and actions. When a packet arrives at the switch, the first flow table is searched for the matching entry with the highest priority. The matches consist of a list of masked values for match fields that specify which packets this entry can be applied to. Match fields include
packet header fields as well as in port and metadata, which can be applied to packets as they are processed. Once the matching entry is found, the entry statistics are updated and the actions from the entry are applied to the packet. Actions include modifying packet headers, adding metadata, sending the packet to another table and outputting the packet to a port or dropping it. The cookie value is used to identify particular flows when they are being modified or when retrieving statistics from them.

While the implementation of the flow tables is entirely left in the hands of the vendor, the requirement for masked matches means that, in hardware, the flow table is typically implemented using Ternary Content-Addressable Memory (TCAM). TCAM performs masked value look-ups quickly, but is expensive in terms of price and power. Because of this flow tables are often limited in terms of the number of flows they can contain.

### 2.2.4 Group Tables

Group tables are tables that contain a list of action buckets, each containing separate sets of actions. When a packet arrives at a group table, the action buckets are applied to the packet based on the group type. There are several group types, the most relevant for this thesis are the all type and the optional fast fail-over and select types.

The all group, as the name suggests, creates copies of the packet for each bucket in the group and performs that bucket’s actions on the copy.

In a fast-failover group buckets are ranked and are associated with a port (or another group) to indicate its liveness. The actions from the first live bucket are applied to the packets. As the name suggests this allows for a quick response to a failure of a port.
The select type group choses one bucket to apply the actions on based on weights applied to each bucket. The selection is based on a hash of the packet headers to ensure flows follow the same path, but how this is implemented is up to the vendor. Buckets in the select group can implement a liveness mechanism similar to the fast fail-over group.

### 2.2.5 OpenFlow Statistics

Tables, individual flow entries and ports all keep statistics on network traffic. Flow table entries count the number of packets and the byte count of matching flows. Tables record the number of look-ups and the number of matches. Ports have separate counters for packets received and packets transmitted, each recording the number of packets, the number of bytes, the number of drops and the number of erroneous packets.

### 2.2.6 The OpenFlow Protocol

The OpenFlow Protocol defines the messages used by controllers to communicate with OpenFlow switches.

The messages used in the work carried out for this thesis are described below.

**Modify Flow Entry Messages**

The `OFPT_FLOW_MOD` message is used to modify the flow table entries on a switch.

**Barrier Messages**

Barrier messages are used to notify the controller that messages have been successfully processed. The controller sends a `OFPT_BARRIER_REQUEST`
message to the switch, which processes any messages it received before
the barrier request before sending a \texttt{OFPT\_BARRIER\_REPLY} message back
to the controller.

\textbf{Multipart Messages}

The messages used to collect statistics from the switches can be large for a
single OFP message, which are limited to 64KB. As a consequence stats-
tics requests and replies are carried in Multipart messages. The \texttt{OFPT\_MULTIPART\_REQUEST} message is used by the controller to request statis-
tics from the switch. The switch then replies with a \texttt{OFPT\_MULTIPART\_REPLY} message containing the relevant statistics.

All OpenFlow messages contain a \texttt{xid} field to identify the transaction.\texttt{OFPT\_MULTIPART\_REPLY} messages use the same \texttt{xid} as the correspond-
ing \texttt{OFPT\_MULTIPART\_REQUEST} message to allow the controller to match
the reply to the request.

\textbf{Port Status Message}

When the state of a port is modified, or a port is added or deleted from
an OpenFlow switch, the switch will generate a \texttt{OFPT\_PORT\_STATUS} mes-
sage to inform the controller of the change.

\textbf{Packet-Out Message}

The \texttt{OFPT\_PACKET\_OUT} message is used by the controller to output pack-
ets from one of the switches it controls. The \texttt{OFPT\_PACKET\_OUT} message
contains the packet data, as well as a set of actions for the switch to apply
to the packet. This can include sending the packet to the first OpenFlow
table for processing or to a group table.
2.3 OpenFlow Switch Implementations

The OpenFlow switch implementations available for use during the work for this thesis were Open vSwitch (Pfaff et al., 2009), an open source production quality software switch and Pronto 3290 and 3780 switches made by pica8. The Prontos run PicOS (Pica8 Inc., 2014), a hardware-agnostic Linux derived operating system for open switches based on Open vSwitch. Both of these switches offered partial support for OpenFlow versions up to 1.3 at the time the work for this thesis was being carried out.

Because both of these implementations are based on the same software, they share many of the same problems. Some features were not supported while the work for this thesis was being carried out: group tables and most MPLS actions were not supported and the collection of statistics from flow table entries in both systems had bugs that caused erroneous results to be returned in certain situations. In addition to these problems PicOS also has a faulty implementation of multiple tables that causes priorities on all tables but the first to be ignored. This effectively rendered multiple table implementations impossible on the Pronto switches.
Resilience is one of the most important properties of a network and as a consequence there is a large body of work in this area, including within SDN. In this chapter existing work is evaluated for its suitability for use in the RouteFlow architecture. Firstly, existing fault detection techniques will be evaluated. This evaluation will focus on speed of detection; scalability, both in terms of switch state and controller load; comprehensiveness of detection; and practicality in terms of implementation in OpenFlow. Secondly, approaches to mitigation of these faults will be examined to determine their suitability for use with the fault detection techniques chosen and with the RouteFlow architecture.

3.1 Fault Detection

Much of the previous work in fault resilience in OpenFlow has focussed on proving the correctness of the controller software and the configuration of switch flow tables (e.g., Al-Shaer & Al-Haj, 2010; Guha, Reitblatt, & Foster,
This is a reflection of the fact that in traditional networks most network faults are caused by software bugs or misconfiguration. However, proving the setup is correct is pointless if faults in the underlying layers result in actual behaviour that is different to what is configured. In the context of a network running RouteFlow it is more important to be able to verify the actual behaviour of the network itself rather than the correctness of the configuration.

In this section existing methods for discovering faults in networks are summarised and evaluated for use with RouteFlow.

### 3.1.1 Bidirectional Forwarding Detection

There has been a lot of investigation into the performance of BFD (Katz & Ward, 2010) with fast fail-over groups and whether an OpenFlow network is able to meet the requirements of carrier-grade networks (e.g., Sharma, Staessens, Colle, Pickavet, & Demeester, 2013a, 2013b). Carrier-grade networks have Service Level Agreements (SLAs) specifying levels of availability, typically 99.999%, and failure recovery time, typically less than 50ms, that must be met. While speed of recovery is an important goal for the work undertaken in this thesis, a 50ms response time is not necessary. Slower recovery could be acceptable if it allows meeting other goals listed in Chapter 1. For instance a slower solution may be more suitable if it reduces the configuration needed.

In *Fast Failure Recover for In-Band OpenFlow Networks* Sharma et al. (2013a) found that the 50ms recovery requirement could be achieved with the use of fast fail-over groups and BFD as the liveness detection mechanism. Because network faults in an in-band controlled network could affect both the traffic passing through the network and the control traffic between the
switches and the controller, it was found that the fail-over paths need to be pre-established for both the control plane traffic and the data plane traffic. The use of BFD, however, is inappropriate for this context as it requires additional configuration to be applied to each switch. The OpenFlow specification has no mechanism for the configuration and initiation of BFD or any other liveness mechanism. While the controller could take responsibility for the generation of keep-alive packets, Kempf et al. (2012) found that this places too great a burden on the controller and the control network. BFD also is very poor at discovering anything but severe packet loss or complete connectivity failure.

3.1.2 Automatic Test Packet Generation

Zeng, Kazemian, Varghese, and McKeown (2012) present a system for automated testing of networks called Automatic Test Packet Generation (ATPG). ATPG retrieves the configuration from devices then determines the minimum set of packets to test every rule (in a SDN, every flow table entry) in the network. Alternatively it can build a set of packets to test every link in order to reduce the impact of the test traffic.

ATPG discovers connectivity problems and packet loss but can also determine congestion and available bandwidth on links. Congestion is found by looking at the round trip time of test packets. Delays are assumed to be caused by congested queues. Bandwidth can be determined with packet pairs (Lai & Baker, 2001), an algorithm for determining bandwidth by measuring the difference in RTT of two packets of the same size travelling through the network. Once a fault is detected, a series of packets are sent through subsections of the path where loss was detected to pinpoint the exact location of the fault.
To fully test every flow on a RouteFlow controlled fabric, would require one test packet per route per external port on the network. As the number of routes could exceed 500,000 this would place far too high a demand on the controller. To test each link requires at most one packet for each link on the network, but a more thorough compromise could be to send a packet between each pair of switches or, on a virtual circuit based fabric, along each path. To reduce the controller load, a single packet out message could be sent to each switch which is directed to a group table that forwards the packet to all switches on the network. The destination switches could match the test packets with an entry that just updates its counters and drops all packets. The controller could then determine whether or not the correct number of packets have been received with a stats request message. This prevents the detection of congestion or available bandwidth, but that could be tested separately on a per link basis.

One further complication with the testing of bandwidth or congestion is the use of in-band control. Because the bandwidth testing will discover only the bandwidth on the bottleneck of a path, if the controller is communicating with the switch via a link with less bandwidth than the path bottleneck then the bandwidth test will detect only the bandwidth on the controller link. This is exacerbated by the fact that the packets to be output are encapsulated between the controller and switch and therefore are larger before they are decapsulated. The congestion test is affected in a similar manner as the delay could be caused between the controller and the switches rather than on the link intended to be tested.
3.1.3 Topology Aware Blackbox Monitoring

*Localizing Packet Loss in a Large Complex Network* (Guilbaud & Cartlidge, 2013) describes a method of using packet injection to exhaustively test every forwarding path in the network, including between every pair of ingress and egress ports on every device. This is achieved by encoding the path for each test packet to take as a stack of MPLS labels indicating the egress port at each hop. The device identifies the egress port associated with the label, pops it and forwards the packet through that port. The faults are localised by identifying the minimal set of links that could be the cause of the loss.

There are two main advantages of this system over ATPG. The use of an MPLS stack to route the test packets allows this system to test links before they are put into production and treating the localisation of faults as a coverage problem rather than probing subsections of the path potentially allows for faster localisation of low levels of packet loss. The first of these is only significant for a RouteFlow deployment if a fail-over path is not pre-installed onto the devices. If they are installed, then they can be tested by ATPG without extra cost, whereas the MPLS routed test packets would require an additional flow table entry for each port.

3.1.4 A Packet Based Method for Passive Performance Monitoring

*A Packet Based Method for Passive Performance Monitoring* (Capello, Cociglio, Bonda, & Castaldelli, 2014) proposes a method to synchronise counters of traffic on multiple devices, allowing exact monitoring of packet loss and network delay in real time. The method involves “colouring” packets by modifying the packet headers in some way, for instance using the lowest
order bit of the DSCP field in the IPv4 header. The aim is to split the traffic into sequential blocks by assigning the same colour to all packets for an arbitrary time period then swapping to the alternative colour. While one colour is being applied to packets, the number of packets of the alternative colour seen at the source and at the destination can be compared, and any difference represents packet loss (or duplication) in the network.

Comparing packet counts requires knowledge of the source and destination of all traffic as it enters and leaves the network. This lends itself to a path based approach, but can be applied to any situation where you know where the traffic will exit the network at ingress.

Delay can be measured by comparing the time the first packet of a new colour was output from the source and the arrival time of the first packet of that colour at the destination.

To implement this method in OpenFlow requires four flows per switch for each other switch on the network: one for each colour for traffic entering and exiting the network. Pinpointing the location of faults can be achieved with a flow for each colour counting traffic entering and exiting each internal port, effectively four flows per internal port. These may be able to be added to switches only after a fault is detected, which could reduce the number of flows required.

The advantages of this system are the low demands on the controller, and the detail and accuracy of the monitoring. The controller only needs to send and receive a single OFP Multipart Message to and from each switch to receive all relevant individual statistics. Any packet loss will be recognised and located immediately and the controller also receives constant accurate information of the exact load on all parts of the network in real
3.1 Fault Detection

3.1.5 Realistic Passive Loss Measurement for High-Speed Networks

Friedl, Ubik, Kapravelos, Polychronakis, and Markatos (2009) suggest a similar approach to Capello et al. (2014). Instead of assigning packets colours they propose counting the total number of packets in each flow at the transport layer\(^1\) as they enter and exit the network. When the flow expires, the counts are compared, and any difference represents loss in the network. This is less suitable for RouteFlow than using colouring as it requires flow table entries to match each transport layer flow. Furthermore the counts are only generated as each flow expires, making it slower at detecting faults.

3.1.6 Summary of Existing Work in Fault Detection

The methods of fault detection investigated here fall into three main groups: passively monitoring the network as described by Capello et al. (2014) or Friedl et al. (2009); injecting test packets as described by Zeng et al. (2012) or Guilbaud and Cartlidge (2013); or setting up BFD sessions on each path in the network. Of the passive methods the colouring method described by Capello et al. (2014) is more suitable as it is faster and requires much less flow table entries. The active monitoring methods each have advantages over one another, but both would need to be modified to be usable with RouteFlow.

The suitability of each of these methods is summarised below.

\(^1\)The flow is defined by the Internet Protocol (IP) source and destination addresses, and the transport protocol, source port and destination port, known as the 5-tuple.
Switch State

Both passive and active monitoring should comfortably meet the requirement of the number of flow table entries per switch scaling linearly in the number of switches in the fabric. Each switch will require at most some constant number of entries for each other switch in the network. BFD does not require additional flow table entries at all, but requires the switches to create BFD sessions between each other. This again scales linearly in the number of switches in the fabric.

Load

As BFD is performed solely on the switches it does not add any load to the controller, but it should be noted that for a full mesh of switches the amount of traffic added to the network between switches will increase quadratically. The amount of traffic even in a very large network is unlikely to be of a significant concern however. Both passive and active monitoring involve regular communication between controller and switch. As the controller is receiving information about a number of paths that grows quadratically in the number of switches in the fabric it is likely that the controller load will also grow quadratically, so an important design goal will be to keep this traffic to a minimum.

Speed and Comprehensiveness

Because passive monitoring counts every every packet as it enters and leaves the network, it will discover any packet loss very quickly after it occurs. Active monitoring only detects loss on the test packets and may take significantly longer to detect low levels of packet loss. BFD is only suitable for detecting very high levels of loss. Both active and passive mon-
3.2 Fault Recovery

Monitoring can also be used for detecting latency across the network but it may not be possible to effectively implement this in OpenFlow or with the RouteFlow architecture.

3.1.7 Conclusion

Based on the evaluation of the existing work in fault detection two fault detection methods will be implemented: Traffic Colouring, based on the technique described by Capello et al. (2014); and Test Packet Injection, based on combining the features of ATPG and those described by Guilbaud and Cartlidge (2013).

These implementations will be evaluated to more accurately determine the relative speed and accuracy as well as their scalability in terms of controller load and flow table requirements.

3.2 Fault Recovery

Approaches to fault recovery fall into two main groups, protection and restoration. Protection is typically used with VCs and involves installing back-up paths for traffic before a fault is detected. Upon detection of a fault the device can automatically redirect traffic onto the back-up path allowing for very fast response to faults. Restoration is more often used with systems that make forwarding decisions separately at each hop and involves waiting until a fault occurs before determining how packets should be forwarded in reaction to the fault.

The advantage of restoration over protection in OpenFlow is the reduction in the number of flow table entries required on the switch. As speed is not the primary goal of this system a restoration approach could be ap-
appropriate. However the additional state required for protection may not be onerous and the use of in-band control can further delay recovery in a restoration based system. If the control traffic is travelling through the same network as the data, a fault in the network will affect the control traffic as well. In order to perform the restoration following a fault, first the control network will need to be restored, effectively doubling the time it takes to restore the network. The requirement of being able to restore function in comparable time to other network protocols such as RSTP cannot be met if the fabric needs to wait for RSTP to restore the network before restoration of the RouteFlow fabric can even begin.

Path protection will work well with RouteFlow as the existing internal forwarding mechanism already uses VCs. VCs also aid the fault detection mechanisms chosen for further evaluation in Section 3.1. Using VCs greatly improves the ability of traffic colouring to locate faults and it simplifies the routes needed to be tested by test packet injection.

The following subsections examine existing work around creating protected paths through networks for suitability in RouteFlow, focussing on the number of flow table entries required to implement them and how well these scale as the fabric grows.

3.2.1 Method and Apparatus for Determining Multiple Minimally-Overlapping Paths

The simplest approach to protection is to build two paths that do not overlap between each pair of switches on the network. If the primary path fails traffic is immediately redirected onto the alternative path. A method for building multiple minimally overlapping paths through a network is de-
scribed in *Method and Apparatus for Determining Multiple Minimally-Overlapping Paths* (Callon, 2001). The paths are not guaranteed to be the shortest possible pair of paths nor is there a guarantee of restricting the number of flow table entries.

### 3.2.2 FatTire: Declarative Fault Tolerance for Software-Defined Networks

*FatTire* (Reitblatt, Canini, Guha, & Foster, 2013) is a high-level language for specifying fault tolerant paths through a software defined network. It allows paths to be defined as regular expressions indicating any requirements for the route and the level of redundancy. The regular expressions are compiled into a set of OpenFlow flow table entries. The redundancy is built into the paths by adding a fail-over path at each hop. This allows completely redundant paths to be achieved with no more than three flows for each destination per switch.

The FatTire compiler itself is not suitable for use with RouteFlow as it compiles directly to OpenFlow table entries, which would not interact with how RouteFlow uses flow tables. Furthermore the FatTire compiler fails if the required level of redundancy cannot be achieved. It is necessary that the path generation can cope with any arbitrary network, and provide the best possible level of redundancy if complete redundancy is not possible. However, the approach of using fail-over groups at each hop on the path between nodes is potentially viable, as it is fast, it allows the use of shortest paths for forwarding when there are no faults, and it can be achieved with a small number of flows.
3.2.3 Network Architecture for Joint Failure Recovery and Traffic Engineering

Traffic engineering and fault recovery in networks are typically handled by separate mechanisms, however Network Architecture for Joint Failure Recovery and Traffic Engineering (Suchara, Xu, Doverspike, Johnson, & Rexford, 2011) proposes combining the two problems by automatically redistributing traffic in a balanced manner in response to faults. For each node diverse paths are created to each destination, then traffic distribution ratios are calculated for each combination of possible path failures. The state requirements can be reduced further by having a single set of weights for each path that provide acceptable (or as close as possible to acceptable) load-balancing for any combination of path failures. The paths are found by combining the optimal set of paths for each possible failure state. The optimal sets of paths are based on the expected traffic demands, which need to be known in advance.

There are two problems with applying this method to RouteFlow. The first is the requirement of knowing the expected traffic demands in advance. The best a truly generic solution could do is assume equal traffic demands between each pair of nodes. The second issue is that the number of flows per switch grows greater than a linear rate with the number of switches in the network. The largest network tested on had a median of 5 paths per switch per destination and some switches with over 10 paths to some destinations. That cost for little gain in terms of the quality of the load-balancing mean that this method is inappropriate for use with RouteFlow, however the concept of combining traffic engineering with fault recovery is worth investigation.
3.2.4 Automatic Failure Recovery for Software-Defined Networks

Kuźniar, Perešini, Vasić, Canini, and Kostić (2013) propose a system for providing fault recovery for failure-agnostic controller modules called Automatic Failure Recovery for OpenFlow (AFRO). In the event of a failure, AFRO recalculates the forwarding state by quickly replaying all communication sent to the controller by the switches in a clean copy of the original controller, operating on the network with all failed components removed.

This is inappropriate for use with RouteFlow firstly because it is a recovery rather than protection approach, which prevents meeting the speed requirements, but also because the internal forwarding state is learned from configuration not from the network switches. Replaying the communication from switches is pointless as it does not have any bearing on the internal forwarding.

3.2.5 Summary of Existing Work in Fault Recovery

Of the works discussed in this chapter, Kuźniar et al. (2013) and Suchara et al. (2011) are unsuitable for use with RouteFlow. FatTire itself is not appropriate, but it could be used as the basis for a method of creating redundancy in the network. Building minimally overlapping path pairs as described by Callon (2001) is the simplest approach to apply to RouteFlow but it does not guarantee linear growth in path requirements.

3.2.6 Conclusion

As none of the methods evaluated are certain to be suitable for use with RouteFlow, implementations will be created based on FatTire, generating
a path with redundancy built in at each hop; on the method described in *Method and Apparatus for Determining Multiple Minimally-Overlapping Paths*; and a third approach, building minimally overlapping paths that are guaranteed to scale linearly with the number of switches in the fabric.

These will be evaluated in terms of the number of flows required per switch, the length of paths created and how well they distribute load around the fabric.
Implementations of fault detection methods were created for further evaluation. These were based on the Traffic Colouring and Test Packet Injection methods described in the previous chapter. These implementations are proofs-of-concept intended for testing, due to limitations in Open vSwitch and PicOS. Implementations for an ideal OpenFlow version 1.4 compliant switch were also designed but not implemented.

In this chapter the proof-of-concept and ideal implementations are described in detail and the limitations of Open vSwitch and PicOS that enforced the difference are explained.

## 4.1 Traffic Colouring

The ideal implementation of Traffic Colouring closely follows the method described by Capello et al. (2014) for identifying packet loss.

This section describes the necessary modifications to RouteFlow and a description of the controller module that monitors the traffic.
4.1.1 Packet Labelling

The simplest method for implementing Traffic Colouring is as part of MPLS VCs. MPLS labels contain a Traffic Class (TC) field, intended to allow Quality of Service (QoS) guarantees with different types of traffic. As RouteFlow does not implement QoS one of these bits can be used to indicate the colour of the packet instead.

Traffic Colouring requires that statistics collected on traffic leaving the network can be matched against the statistics collected on the same traffic as it entered the network. This is trivial to determine when you have unique labels at each link for all the paths your network. However, to support this in OpenFlow requires a number of flow entries that grows quadratically with the number of nodes in your network. In order to reduce the number of flows, VC paths in RouteFlow form trees: at each switch in the network, all traffic following the same path to each egress switch is aggregated into a single path. As a consequence, traffic arriving at an egress switch on any given path may have come from multiple possible ingress switches. With an MPLS implementation the source of the traffic can be identified using a label inside the label stack. This is pushed by the ingress switch before the path label.

As not all MPLS related actions in OpenFlow were supported by PicOS or Open vSwitch during this thesis, the proof-of-concept implementation used VLAN tags with two unique, complete paths between each pair of nodes. The colour of packets was indicated by the uppermost bit of the VLAN identifier field. This limits the number of nodes to 8 per network which is insufficient for most applications, but enough for the tests that were performed.
4.1.2 Flow Tables

RouteFlow switches use a single flow table to determine the actions to apply to each packet. Traffic Colouring can be implemented in a single table, but to do so means all traffic cannot be measured. The flow entries for each colour must remain while the other colour is being applied to traffic so that the statistics can be retrieved from them.

At the source, flow entries for each colour match exactly the same traffic. When the switch changes traffic a new flow table entry must be added that matches the same traffic but takes precedence over the existing entry. The previous entry cannot be deleted as the packet counts still need to be retrieved from it. The only way to change the colour of traffic in a single flow table implementation without deleting the existing flow entry is to give the new entry a higher priority.

At some point the new entry will not be able to receive a higher entry and the old entry will need to be deleted before the statistics can be measured. To solve this problem, a multi-table RouteFlow architecture was developed. The table structure is as follows:

*Table 0* separates traffic into two categories based on whether it arrived on an external port or whether it is traffic from inside the network. The traffic arriving on external ports is checked for a correct Ethernet destination address and is forwarded to *Table 4* to determine its route. Traffic from internal ports is forwarded to *Table 1*. Optionally internal ports can have entries matching each colour and ingress port to help identify the exact location of faults.

*Table 1* is for internal traffic and matches on the MPLS label to determine
if the packet exits the network at this switch or whether it needs to be forwarded on to another switch. Packets to be forwarded have metadata specifying their output port and are sent to Table 7. Other packets have the outermost MPLS label popped and are sent to Table 2. In the proof-of-concept implementation, this table has matches for each colour of each path terminating at this switch, and the packet counts are recorded here.

Table 2 is where the statistics for outgoing traffic in the ideal implementation are collected. Entries match on the MPLS label, representing the ingress switch, and the colour. The label is popped and the packets are forwarded to Table 3.

Table 3 has entries matching the final MPLS label on the stack. The labels represent the port the traffic should use to exit the fabric as well as indicating the Ethertype that should be restored to the Ethernet frame. The label is popped and the packets are output from the fabric. An alternative system could identify the Ethertype of packets in Table 2. As it is likely the only two packet types will be IPv4 and IPv6 this will only require one bit of the MPLS label. The packets can then be sent to Table 4 to determine the egress port, reducing the number of labels required per packet.

Table 4 has an entry for every route learned by the RouteFlow VM as well as entries for identifying control plane traffic and forwarding it to the controller. The route entries match on masked IP destination address values with priorities based on the CIDR prefix length. The Ethernet source and destination address fields are updated. Even if the packet is being passed through the fabric it will be forwarded based entirely
on MPLS label, therefore the Ethernet addresses do not matter and can be updated immediately. If the port where the packet leaves the fabric is on the current switch then the packet is output immediately. Otherwise the egress port and Ethertype label are pushed onto the MPLS stack, metadata is added to the packet indicating the egress switch and the packet is sent to Table 5.

Table 5 contains a single entry matching all traffic that sets the colour bit in the TC field and passes the packets on to Table 6. There are two reasons this is in a table by itself. Firstly because the colour is changed regularly, being able to do so with a single OFPT_FLOW_MOD message greatly reduces the traffic the controller has to send. Secondly, the entries where the traffic is counted have to persist while the other colour is being applied. Separating the flow entry that sets the colour from the entries that collect the statistics allows the collection entries to coexist with equal priorities.

Table 6 has entries for each destination and colour combination, matching the packet metadata (representing the egress) and colour bit in the current outermost label. This is where the statistics for traffic originating at this node are collected. This entry also pushes the outermost label, denoting the path the traffic will take through the network; sets the colour bit, so that it is indicated on both the outermost and the second outermost label; modifies the packet metadata to represent the appropriate output port and sends the packet to Table 7.

Table 7 aggregates all traffic being output to each internal port into two flow entries by matching on metadata and colour. The packets are counted and output via the port indicated by the metadata.
Faults are identified by comparing the numbers of packets at the flow entry representing each ingress/egress pair in Table 6 of the ingress node and Table 2 of the egress node. Any difference in packet counts between the two represents some fault in the delivery of packets through the network.

### 4.1.3 Locating Faults

Once a fault is detected, its exact location can be pinpointed by examining the packet counts of flow table entries on transit nodes along the path. The flow table entries for locating faults count traffic on a link by link basis rather than per path. This reduces the number of flow table entries required, as well as reducing the number and size of messages between switches and controllers to retrieve the statistics.

For this to work, it means that all switches in the network must apply the same colour to all traffic synchronously. The changes in colour do not have to happen at the exact same point in time, but at the point when the statistics for a given colour are retrieved from the switches, all switches must be applying the alternative colour to all traffic. This can be enforced with the use of `OFPT_BARRIER_REQUEST` messages.

The flow entries to match each colour of traffic at each hop can be installed permanently in anticipation of faults, requiring an additional two entries per internal port. Alternatively they can be installed in response to a fault. It is possible, however, that this could involve installing rules to count the traffic for every internal link in the fabric. This is even more likely if faults exist and are discovered in multiple places in the network at the same time. For this reason it is recommended that the fault locating flow entries are installed permanently.
4.1.4 Statistics Retrieval

The statistics recorded by flow tables in Open vSwitch, and consequently PicOS, are updated every second. This acts as the main limitation to the speed at which faults can be detected using Traffic Colouring. After receiving the final OFPT_BARRIER_RESPONSE message indicating that all switches are applying the new colour, it is necessary to wait at least one second plus the maximum expected time for traffic to traverse your network. The duration of this wait period is known as the request interval. There is no way for the controller to know exactly when flow statistics will be updated. A node may update its statistics shortly after the controller receives the final OFPT_BARRIER_RESPONSE message but before all traffic of the previous colour has reached it. By waiting for one second plus the expected maximum time for traffic to cross the network then any traffic that has not reached its destination by the time a node last updated its statistics is itself an indication of a fault in the network.

4.1.5 Controller Module

The implementation of the controller module uses two objects types to store information about the network, Stations and Tubes. Stations represent a single table entry on a switch that the controller needs to poll for statistics. A Tube has two sets of stations, a source_tubes set and a destination_tubes set, where all of the traffic passing through the stations in the source_tubes set must subsequently pass through exactly one Station in the destination_tubes set.

Stations contain a type field and an identifier field. The type indicates what the Station represents, in the current implementation this
is always a flow table entry but it could also refer to a group table entry. The identifier for a flow table entry is its cookie value and is used to identify the correct Station when a OFPT_MULTIPART_RESPONSE message is received. Stations also contain fields for the current values of the statistics being retrieved from the switches, in the current implementations this is only the packet counts, but could also include other statistics such as byte counts.

Tubes contain the two sets of tubes as well as a test_callback function to be used to test for faults after the Stations have been updated.

The colouring module handles scheduling the OFPT_MULTIPART_REQUEST messages to retrieve the statistics, the performance of tests to detect and locate faults, and changing the colour being applied to packets.

In its Initial state, the coloring module installs all the necessary flow table entries on the switches and waits for the request interval to allow enough traffic to pass through the fabric to be able to retrieve meaningful statistics from the switches. The module then moves into the Change Colour state.

In the Change Colour state the controller sends OFPT_FLOW_MOD messages to change the traffic colour. Each OFPT_FLOW_MOD message is followed by an OFPT_BARRIER_REQUEST message, the xid value of which is added to a set. When the OFPT_BARRIER_RESPONSE message is received its xid is removed from the set. When the final xid is removed the module adds all active tubes to a set of tubes to be tested, called the test_tubes set, the module then waits for the request interval, before moving into the Request Statistics state.

In the Request Statistics state the controller iterates over the Tubes in the test_tubes set and generates OFPT_MULTIPART_REQUEST messages to
collect all the necessary statistics for the Tubes’ Stations from the switches. In order to reduce the traffic between the controller and the network, statistics from multiple Stations are requested with one OFP_MULTIPART_REQUEST message. When OFP_MULTIPART_REQUEST messages are sent the Stations are added to a dictionary using their identifier values as keys. As OFP_MULTIPART_RESPONSE messages are received the correct Stations are located and removed from the dictionary and their statistics fields are updated. When the dictionary is empty, the controller moves into the Test state.

In the Test state the test_callback functions from the Tubes in the test_tubes set are executed. These may cause tests of other Tubes to be scheduled. The test_tubes set is cleared and any newly scheduled Tubes for testing are added to it. After this if the test_tubes set is empty the controller moves back to the Colour Change state, otherwise it moves to the Request Statistics state.

4.1.6 Counter Accuracy

At the time of writing this thesis there is a bug in Open vSwitch and PicOS that causes packets counted by the data plane to be attributed to flow table entries incorrectly. When a traffic flow has been identified as matching a specific flow table entry and then the flow table changes so that the traffic flow now matches a different entry, some packets will have the actions from the new flow table entry applied to them, but will have their statistics applied to the old flow table entry.

A work around for this is to use two Open vSwitch bridges in the place of each switch, one internal and one external. The internal bridge connects to the external bridge and to the other internal bridges in the fabric, and the
external bridge connects to the internal bridge and has the external ports. When traffic arrives at the external bridge it determines the egress, pushes the necessary MPLS labels, sets the colour and passes the traffic to the internal bridge. This modifies the traffic flow at the second bridge allowing it to accurately attribute the statistics collected to flow table entries.

Since two bridges can be instantiated in one Pronto switch or in one Linux server running Open vSwitch this does not require additional hardware. However in the case of the Pronto this requires dedicating two ports on the device to act as the link between the two bridges. All traffic between the bridges must pass over that link, significantly reducing throughput. This can be mitigated somewhat by adding additional links between the two bridges, but this obviously has an additional cost in terms of the ports used. Furthermore it doubles the processing requirements for each packet as they have to be processed by the external and internal bridges.

4.2 Test Packet Injection

The implementation of Test Packet Injection differs significantly from either the method described by Zeng et al. (2012) or Guilbaud and Cartlidge (2013). These changes improve scalability when injecting packets and monitoring from a single device or were enforced by limitations of OpenFlow or by specific requirements of RouteFlow.

The controller sends a single instruction telling the switches to send a probe packet to every other switch in the fabric. When these probes arrive at the destination, a flow table entry updates its counters and drops them. The controller sends a series of these instructions then waits for all counters to update, then polls the destinations to determine how many of the test
4.2 Test Packet Injection

packets arrived. If the number of packets received does not match the number of probes sent then that indicates a fault in the network.

4.2.1 Packet Labels

As with Traffic Colouring, Test Packet Injection works best with MPLS virtual circuits. Once the topology is configured all the paths that may be used can be pre-calculated so that testing can be limited to the existing paths.

To differentiate the source of test packets arriving on aggregated paths, the test packet’s ingress switch can be identified with a label inside the MPLS stack in the same way as was used with Traffic Colouring. With Test Packet Injection this only needs to occur on the test packets, the data plane traffic needs only the label indicating its Ethertype and egress port and the label identifying its path.

In order to locate faults, traffic must be identifiable as test packets from the outermost MPLS label. This can be indicated using a bit in the MPLS TC field.

4.2.2 Sending Probes

Sending a probe along every path in the fabric can be achieved with a single `OFPT_PACKET_OUT` message from the controller to each switch. The controller directs the packet to a group that sends a copy of the packet to every other switch in the fabric. Each switch can have a flow table entry matching test packets from each other switch. These entries update their counters and drop the packets. This way the controller can simply send one `OFPT_MULTIPART_REQUEST` message and receive confirmation that multiple probes have been received.
4.2.3 Switch Configuration

The flow table structure to implement Test Packet Injection is much simpler than that required to implement Traffic Colouring. Flow entries are needed to collect and drop test probes and matching test probes separately at transit hops allows faults to be located more easily. The flow table structure is shown below:

Table 0 is the same as in the Traffic Colouring implementation, it separates packets arriving from internal links from those arriving from external ports, which are checked for correct Ethernet destination addresses.

Table 1 contains entries matching the MPLS label for paths terminating at this switch and the MPLS label and probe identifier for transit packets. Packets from paths terminating at this switch have the outermost MPLS label popped and are sent to Table 2.

Test and data packets matching labels for transit paths are output. The statistics from the transit test packet flow entries are used to locate faults.

Table 2 contains entries that match on the inner MPLS label. For data plane packets this represents the Ethertype to be restored to the packet’s Ethernet frame and the egress port. On probe packets the label represents the ingress switch. The probe packets are dropped and the controller can examine the counters on these flow entries to determine if any probes have been lost.

Table 3 contains entries matching routes learned by the RouteFlow VM and entries matching control plane protocols. The matching is the same as in the Traffic Colouring implementation. The actions set the packet’s
4.2 Test Packet Injection

Ethernet source and destination fields and if the packet exits the fabric at a different switch to the current switch the full MPLS stack is loaded onto the packet, one label representing Ethertype and one representing the path through the fabric. Once the packet headers have been modified the packet is output.

There is also a group table to apply to the test packets originating from each switch. It is simply an all type group with a bucket for each other switch in the fabric. The bucket pushes the correct labels, sets the probe identification bit and outputs via the correct port.

4.2.4 Probe Timing and Statistics Requests

When the controller sends a probe it first must travel to the switch as a OFP_PACKET_OUT message before it can be forwarded along its path through the fabric. So the expected time it will take to arrive at its destination is the expected time between the controller and switch plus the expected time between the two switches. However, the nature of the channel between the switch and controller is not detailed in the OpenFlow specification, making it difficult to create a generic method of monitoring the progress of test packets before they reach the source switch.

To be sure the probes have reached the path being tested the controller can use an OFPT_BARRIER_REQUEST message after each probe or set of probes. When the OFPT_BARRIER_REPLY message is received, the controller waits for one second plus the maximum expected transit time and then sends a OFPT_MULTIPART_REQUEST message to the destination to retrieve the statistics. Because of this requirement, it makes sense to send multiple probes over an arbitrary time period before verifying their arrival at the destination.
4.2.5 Controller Module

The controller module is fairly simple. There is a single class of object for storing network state, a `Path` object representing one path between a pair of switches in the fabric. It contains an `identifier` field, fields for the statistics received from the switches and a `test_callback` function.

First the switches are initialised, then the controller can start sending test probes. It sends a sequence of probes over an arbitrary period to each switch then sends an `OFPT_BARRIER_REQUEST` message. Once the last `OFPT_BARRIER_RESPONSE` message is received the controller waits for at least one second then sends a single `OFPT_MULTIPART_REQUEST` message to each switch. As the controller receives `OFP_MULTIPART_RESPONSE` messages it updates the statistic fields in the `Path` object and calls its `test_callback` function.

4.2.6 Barriers to Implementation

Open vSwitch currently supports everything needed to implement Test Packet Injection but PicOS does not correctly support multiple tables. A single table implementation can be created simply by using unique path labels for each pair of switches. Each switch must include duplicates of each transit flow for test packets but otherwise nothing needs to be modified.
Fault Response

Following from the review of existing work in Chapter 3, three methods of generating paths with redundancy were created. The first of these creates a minimally overlapping pair of paths between each pair of switches. The second also creates a minimally overlapping pair of paths between each pair of switches but ensures that each switch can forward all traffic to a given destination using no more than two paths. This is not enforced by the previous method, and therefore may reduce the number of flow table entries needed to implement minimally overlapping paths. The third method is loosely based on FatTire and creates a single path between each pair of nodes with redundancy at each hop.

These have been implemented in Python for further evaluation. The specific details of the algorithms used are described here.


5.1 Forwarding Rings

All of these algorithms are based on a method of finding a diverse collection of paths between vertices on a connected, undirected graph. For each destination the vertices in the graph are arranged into ranked sets of sequentially connected vertices called rings. Rings are a method of encoding the redundancy in a graph that also serves to organise vertices into a hierarchy, allowing the paths of higher ranked vertices to take precedence over those of lower ranked vertices. The rank of a ring is an indication of the distance of the vertices on the ring from the destination. The lowest ranked ring is considered the innermost and it is directly connected to the destination. The highest ranked ring is considered the outermost.

The first and last vertex in the sequence of vertices on a ring each connect either to a distinct vertex that belongs to a ring with a lower rank or to the destination. The two vertices on the lower ranked rings are referred to as the parents of the higher ranked ring. The ranking of a ring represents the closeness to the destination based on the number of rings that must be traversed to reach the destination from that ring.

Each vertex can be a member of only one ring for each destination. A full path from a vertex to the destination can be found by repeatedly moving to a more inward ring by travelling through the sequence of vertices on a ring to its parent, until you reach the destination.

Pseudo-code for finding the rings is shown in Algorithm 5.1.

The algorithm is run separately for each destination vertex. Firstly a rank counter is initialised and set to 0. Then all vertices in the graph except the destination are added to a set, called free_vertices. Vertices are


5.1 Forwarding Rings

```
1  rings = {};    
2 for each (destination in vertices){
3      count = 0;  
4      pathTrees = {};  
5      freeVertices = vertices \ destination;  
6      for each (root in freeVertices where (root, destination) is in edges){
7          newPathTree = a new PathTree with root as its root node and destination as its  
8              ringParentVertex;  
9              add newPathTree to pathTrees;  
10         }
11     }
12     extendibleVertices = a queue containing every vertex in every pathTree in pathTrees;  
13     while (extendibleVertices is not empty){
14         parent = the vertex dequeued from extendibleVertices;  
15         parentPathTree = the pathTree that contains parent;  
16         for each (child in freeVertices where (child, parent) is in edges){
17             if (child is not already in a PathTree){
18                 add child to parentPathTree as a child of parent;  
19                 enqueue child to extendibleVertices;  
20             }
21             else if (child’s PathTree is not the same as parentPathTree){  
22                 childPathTree = child’s PathTree;  
23                 newRing = a Ring object with rank equal to count;  
24                 set the ringParentVertex of childPathTree and parentPathTree as the  
25                     parentVertices of newRing;  
26                 increment count;  
27                 for each (vertex in the path between childPathTree’s root node and  
28                     parentPathTree’s root node via child){  
29                     append vertex to newRing’s vertices list;  
30                     newPathTrees = split_path_trees (vertex, vertex’s PathTree);  
31                     pathTrees = pathTrees \ newPathTrees;  
32                     remove vertex from freeVertices;  
33                     enqueue all vertices in newPathTrees to extendibleVertices;  
34                 }
35             remove childPathTree and parentPathTree from pathTrees;  
36             add newRing to rings [destination ];  
37         }
38     }
39 }
```

**Algorithm 5.1**: Forwarding rings algorithm pseudo-code

removed from this set as they are added to rings.

Each vertex in the free_vertices set that is connected by an edge to the
destination is added to a separate tree structure, called a PathTree, at the
root. Each PathTree represents paths from vertices in free_vertices
to a vertex in a ring or the destination.
Each PathTree connects at the root to a vertex that belongs to a ring to the destination or to a vertex in a ring that connects to the destination. That vertex is set as that PathTree's ring_parent_vertex.

Vertices can be in only one PathTree at a time. When they are found to connect to a vertex in another PathTree the vertices on the paths between the two connecting vertices and their respective roots are used to create a new ring.

To populate the PathTree structures all the root nodes of PathTrees are added to a queue called extendible_vertices. The population process works as a loop that starts by dequeuing the first element from extendible_vertices and finding vertices that are connected to it. These are known as the parent and child respectively.

If the child is not already in a PathTree it can be added to the parent node's PathTree as its child and appended to extendible_vertices. If the child vertex is already in the same PathTree as the current node then it cannot be used to form non-overlapping paths to an existing Ring and nothing more needs to be done. If it is in a different PathTree then it can be used to form non-overlapping paths to an existing Ring. The nodes on the paths between the child vertex and the current node and their respective PathTree root nodes are used to form new Rings.

When a new Ring is created, vertices are taken from two PathTree structures. The remaining vertices in those PathTrees are used to create new PathTrees. This is performed in the function split_path_trees which is shown in Figure 5.2. It is called for each vertex being added to the new ring and returns a set of new PathTree structures for the vertex's descendants. A new PathTree is created for each child with the parent vertex
5.1 Forwarding Rings

```java
define split_path_trees
arguments:
  pathTree: a PathTree object
  node: a node in pathTree
returns:
a set of PathTree objects
{
  newPathTrees = {};
  for each (child of node in pathTree){
    newPathTree = a new pathTree with child as its root and node as its
    ringParentVertex;
    add the descendants of child to newPathTree as descendants of the root;
    add newPathTree to newPathTrees;
  }
  return newPathTrees;
}

Algorithm 5.2: Split Path Trees function pseudo-code
```

as the ring_parent_vertex. The child is added to the PathTree as the root and its descendants are used to populate the tree.

There may be connections between vertices on the PathTrees created by split_path_trees. To check for connections all the vertices in the new PathTrees are appended to extendible_vertices. The vertices used to create the ring are removed from free_vertices.

The algorithm continues in this fashion, dequeuing the first element of extendible_vertices and searching its edges for vertices that can be used to extend the PathTree or make rings, until extendible_vertices is empty. At this point any vertex not a part of a ring does not have redundant paths to the destination.

Figure 5.2 shows the rings generated by applying this algorithm to the network shown in Figure 5.1. Vertices on the same ring are connected with a bold line and the edges between rings and their parents are indicated with a dashed arrow.

Ways to extend this algorithm to create maximally redundant paths are explained later in this chapter.
5.2 Pairs of Minimally Overlapping Paths

The pairs of minimally overlapping paths algorithm, or the Path Pairs algorithm as it will be referred to throughout this thesis, finds a pair of paths between each source and destination vertex combination in the graph with the minimal overlap between the two paths.
One method for finding pairs of minimally overlapping paths is described by Callon (2001). However, as the forwarding rings need to be calculated for the other algorithms and converting forwarding rings into pairs of minimally overlapping paths is simple, this approach was used instead.

![Diagram](image)

**Figure 5.3:** Step 1 of building paths to vertex 10 for the vertices on ring 4. Bold arrows represent the paths. Dashed arrows represent the paths between the final element of the outermost paths and its parent vertices. The outermost path (terminating at vertex 1) must be extended to vertex 2 as vertex 8 is the final element of the innermost path.

There are two stages to this algorithm. In the first stage fully redundant paths between each pair of nodes are found where possible. This is described in pseudo-code in Algorithm 5.3 and a demonstration of this algorithm is shown in Figures 5.3 to 5.6. The second stage is used to find the paths where complete redundancy is impossible. The pseudo-code for this method is shown in Figure 5.4.

For the first stage, the algorithm iterates over each vertex and each ring with that vertex as its destination. Two lists are created containing the two parent vertices of the ring to represent the paths to the destination, these are referred to as $\text{path}_1$ and $\text{path}_2$. 
Figure 5.4: Step 2 of building paths to vertex 10 for the vertices on ring 4. The outermost path (terminating at vertex 2) can be extended to either vertex 5 or vertex 9.

Figure 5.5: Step 3 of building paths to vertex 10 for the vertices on ring 4. Both paths terminate at vertices on the same ring. The paths are advanced so as to avoid collisions.

If the final element in both lists is the destination, then the paths are complete and the algorithm moves onto the next ring. Otherwise the path where the final element is on the higher ranked Ring appends the path between the final element and one of the Ring parents to itself. The choice of parent is arbitrary provided it is not the final element of the other path. If
both final elements are on the same Ring then both paths append the path between their final element and the parent that does not cause a collision. To avoid the collision the full path to the parent must be checked to see if it contains the other path’s final element. This only needs to be done for the first path to be advanced, the second only needs to test the final element.

When choosing a parent for the path to advance to, it is sufficient to only test the final element in the alternative path for a collision because of two aspects of this algorithm. Firstly, paths are only extended when the path is further outward from the destination than the other path. The one exception to this is when both paths are on the same Ring, in which case they advance in such a way that guarantees no collisions. Secondly, when a path is extended it only travels through one Ring to its parent. The nodes on that Ring are necessarily further outward than the other path so only the parent needs to be tested.

This process of extending the outermost path continues until the destination is reached by both paths. Once both paths are found, the process is
for each (destination, destinationRings pair in rings) {
    for each (ring in destinationRings from lowest ranked to highest) {
        path_1 = a list containing the first parent of ring;
        path_2 = a list containing the second parent of ring;
        while (true) {
            if (the final element in both path_1 and path_2 is destination) {
                the paths to destination for ring are found so break;
            }
            else {
                ring_1 = the Ring containing the final element of path_1;
                ring_2 = the Ring containing the final element of path_2;
                if (ring_1's rank is greater than ring_2's) {
                    newDestination = an arbitrarily chosen parent of ring_2 that is not the current final element of path_2;
                    append the path between the final element of path_1 and newDestination to path_1;
                }
                else if (ring_2's rank is greater than ring_1's) {
                    append the path between the final element of path_2 and an arbitrarily chosen parent of ring_2 to path_2;
                }
                else if (ring_1 is ring_2) {
                    append the path between the final element of path_1 and the parent of ring_2 that can be reached without passing the final element of path_2 to path_1;
                    append the path between the final element of path_2 and the other parent of ring_2 to path_2;
                }
            }
        }
        Append path_1 and path_2 to the paths between each vertex in ring and the parents of ring, these are the fully redundant paths to the destination for these vertices;
    }
}

Algorithm 5.3: Path Pairs algorithm pseudo-code

Finding pairs of paths when it is not possible to do so without overlaps is not done until after all the fully redundant paths to all destinations are found. Again this iterates through each destination vertex in turn. Pseudo-code for this algorithm is shown in Algorithm 5.4.

Firstly, a queue called potential_parents is initialised to contain only the destination. The vertices in this queue have paths to the destination vertex but may be connected to other vertices that do not. A set of vertices
Algorithm 5.4: Overlapping paths algorithm pseudo-code

that have not yet been tested for paths to the destination is created, initially containing all vertices except the destination.

The algorithm then runs in a loop, each time dequeuing the first element of potential_parents and checking the vertices connected to it for paths to the destination. This repeats until potential_parents is empty, meaning every vertex now has a pair of minimally overlapping paths.

If a vertex already has redundant paths to the destination then it is enqueued to potential_parents and the next vertex can be checked.
Figure 5.7: Vertex is its own subdestination

Figure 5.8: Vertex uses parent's subdestination

Figure 5.9: Vertex uses parent as its subdestination
If the vertex does not have paths to the destination then it will find a subdestination, the first vertex in the network where an overlap must occur. If the subdestination does not have redundant paths to its parent, then the subdestination is the vertex itself, this is shown in Figure 5.7. The vertex is appended to each of the parent’s paths to the destination. The parent’s subdestination is used when the parent’s paths overlap and the vertex has non-overlapping paths to the parent’s subdestination. The vertex’s paths to the destination are created by appending the vertex’s paths to the subdestination to the subdestination’s paths to the destination. This is shown in Figure 5.8. If the vertex has non-overlapping paths to its parent but not to the parent’s subdestination or the destination then the vertex will use the parent as it’s subdestination. The paths are created by appending the vertex’s paths to the parent to the parent’s paths to the destination. This is shown in Figure 5.9. Once the paths are created the vertex is enqueued to potential_parents and the next vertex is tested.

5.3 Minimal Pairs of Minimally Overlapping Paths

Pairs of minimally overlapping paths between each pair of switches in a network can be implemented so that each switch requires no more than two flow table entries for each other switch. To achieve this all VCs to a destination that transit through a switch need to follow the paths to that destination the switch uses for its own VCs. The VCs can share MPLS labels from that point onward, thereby reducing the amount of flow table entries required. This is not guaranteed to be possible with the previous algorithm.

New algorithms to ensure that the path taken by any VC transiting a switch
to a destination must be the same as one of that switch’s own VCs paths to that destination were created. The first was created and tested, but during the evaluation a flaw in the algorithm was discovered that caused switches to require more than the minimum number of flows. A second algorithm was designed, but there was not adequate time to implement it. Further evaluation of the second algorithm could be the basis for future work. Throughout this thesis these algorithms will be referred to as the Shared Paths algorithms.

### 5.3.1 Algorithm i

Similar to the Path Pairs algorithm described in the previous section, the Shared Paths algorithms also have two stages. The second stage in both algorithms, creating paths with overlaps, is identical to the algorithm used in the previous section (shown in Algorithm 5.4). The first stage of algorithm i is described in Algorithm 5.5. The procedure for finding paths is illustrated in Figures 5.10 and 5.11.

The premise of this algorithm is that the paths of higher ranked rings take precedence over the paths of lower ranked rings. The algorithm finds pairs of paths for rings in order of rank starting with the highest ranked and finishing with the lowest. Path pairs are generated together by advancing the outermost path until the paths reach their destination or a vertex with pre-existing paths. Once the pair of paths have finished advancing any path that has not reached the destination is completed by selecting a pre-existing path. It was initially believed that if two paths both reach vertices with pre-existing paths then they would always be able to chose paths that do not collide from the pre-existing paths. This assertion was proven to be incorrect during testing.
Figure 5.10: Building paths to vertex 10 for the vertices on ring 3. The paths for ring 4 have been generated and are shown as dashed arrows. To avoid a collision, the path terminating at vertex 2 must follow the pre-existing path via vertex 1.

Figure 5.11: The final paths to vertex 10 for the vertices on ring 3.

The algorithm iterates over each ring to each destination beginning with the highest ranked ring and working towards the lowest ranked. As in the previous section, the paths are extended by moving the path on the outermost ring one ring at a time towards the destination.

When a path is being extended by appending a ring each vertex is added
Chapter 5 Fault Response

for each (destination, destinationRings pair in rings){
  for each (ring in destinationRings from highest ranked to lowest ranked){
    path 1 = a list containing the first parent of ring;
    altpath 1 = NULL;
    path 2 = a list containing the second parent of ring;
    altpath 2 = NULL;
    while (the final elements of path 1 and path 2 are not both destination){
      if (the final element of path 1 has previously been included in paths to destination){
        set altpath 1 to be a list containing the same elements as path 1;
        append one path of the paths from the final element of path 1 and the destination to path 1 and the other to altpath 1;
      }
      else if (the final element of path 2 has previously been included in paths to destination){
        set altpath 2 to be a list containing the same elements as path 2;
        append one path of the paths from the final element of path 2 and the destination to path 2 and the other to altpath 2;
      }
      else if (neither final element has previously been included in paths to destination){
        ring 1 = the Ring containing the final element of path 1;
        ring 2 = the Ring containing the final element of path 2;
        if (ring 1’s rank is greater than ring 2’s){
          newDestination = an arbitrarily chose parent of ring 1 that is not the current final element of path 2;
          append the path between the final element of path 1 and newDestination to path 1;
        }
        else if (ring 2’s rank is greater than ring 1’s){
          append the path between the final element of path 2 and an arbitrarily chosen parent of ring 2 to path 2;
        }
        else if (ring 1 is ring 2){
          append the path between the final element of path 1 and the parent of ring 2 that can be reached without passing the final element of path 2 to path 1;
          append the path between the final element of path 2 and the other parent of ring 2 to path 2;
        }
      }
    }
    find a pair of paths out of path 1, path 2, altpath 1 and altpath 2 that do not contain collisions. These are the redundant paths; set each hop in your paths and each vertex in ring to use these paths as their redundant paths to destination;
  }
}

Algorithm 5.5: Shared Paths algorithm i pseudo-code.

one at a time. As the vertices are added they are tested for an already existing pair of paths to the destination. Once a path reaches a vertex with pre-existing paths the path stops advancing. The algorithm continues but
will always extend the other path, until it too reaches a vertex with pre-existing paths or the destination.

Once both paths have reached the destination or a vertex with pre-existing paths then a pair of paths is selected. Any path that reached the destination without crossing a vertex with pre-existing paths can be used without further processing. If one path crossed a vertex with pre-existing paths but not the other then the incomplete path can be completed by appending either pre-existing path to it.

If both paths crossed vertices with pre-existing paths then the vertices are checked for paths belonging to the same ring. The paths will follow the pre-existing paths made by the highest ranked ring that has paths containing both vertices. Otherwise any pair of non-overlapping paths can be used.

The belief that the pre-existing paths of the two final vertices of the paths would always be able to make non-overlapping paths to the destination was based on the fact that a collision can only be caused when one path is following the path of a higher ranked ring’s VC and the other path is entering that ring’s path. Because the paths of the higher ranked ring’s VCs take precedence over the current ring’s paths, if the path entering the higher ranked path had previously crossed the higher ranked ring’s other VC it would have followed it to the destination and could not reach the vertex where the collision has occurred. Therefore the path of the current ring that is following the pre-existing path could avoid the collision by following the alternative path when it first reached a vertex on the higher ranked ring’s path.

The flaw with this reasoning is that the path entering the higher ranked path may have been able to branch away from the pre-existing path at
the point of entry if it was already following another path. This other path and the original higher ranked path may have chosen differing pre-existing paths at a vertex to avoid a collision. This could then cause a situation where a pair of vertices have pre-existing paths which both collide with both of each others paths.

5.3.2 Algorithm ii

The second implementation of the Shared Paths algorithm is simpler, runs faster and produces shorter paths. It is shown in Figure 5.6. The algorithm modifies the forwarding rings algorithm, so that the paths are calculated as each ring is created. As vertices are added to rings they are assigned a location in one dimensional space. When a ring is created the parents location is compared. The vertices in the ring are then assigned locations directly to the right of the leftmost parent. The paths are found by either always moving to the leftmost parent or always following the rightmost. Because the paths only enter vertices to the left or right of their starting point, they can never cross, and each vertex in each ring only ever needs to forward traffic in two possible directions. Figure 5.12 depicts the rings shown in Figure 5.2 rearranged to demonstrate the locations of all of the vertices and Figure 5.13 illustrates the generation of paths.

The location of the vertices is stored as a list of integers. To determine the location of vertices on a new ring the additive inverse of the rank is appended to the leftmost parent’s location. Then the vertices are assigned an individual value which increments from the vertex that connects to the leftmost parent through the sequence of vertices until the largest value is assigned to the vertex that connects to the rightmost parent. These values are appended to the ring location to create the location of each vertex.
Figure 5.12: Rings to vertex 10 rearranged to demonstrate vertex locations.

Figure 5.13: Paths to vertex 10 for the vertices on ring 3.
To compare the location of two vertices the values in the location lists at the same index are compared from first to last. When one location has a lower value at a given index or has no element at that index it is considered to be to the left of the other location.

This should be significantly faster than algorithm i because once the rings are created the paths are found without requiring any collision detection, which requires comparing each element in each path to each element in the other path. The only comparisons required are used to determine which parent is the leftmost when building the rings. The method of comparing the position of vertices presented here may be able to be improved upon with the use of floats in the place of lists of integers.

The paths created should be much shorter because vertices on lower ranked rings are not forced to follow the paths of vertices on higher ranked rings. Paths from higher ranked rings have to pass through more rings to reach the destination which causes them to be longer.

5.4 Ring Based Forwarding

Ring Based Forwarding is a new algorithm loosely based on the algorithm used by FatTire to add redundancy to the forwarding behaviour of an OpenFlow network. The modifications allow the algorithm to be used in the case that a redundant path is not available at every hop and enforce a limit of three flow table entries per switch for each destination. Once forwarding rings are calculated, the method is very simple; each switch in the network forwards traffic to an arbitrarily chosen parent of its ring to the destination. If the path to the parent is down, traffic can be forwarded to the ring’s other parent. To avoid loops the parent that the traffic will be
forwarded to is chosen only once, as the traffic enters a new ring. If the parent is unreachable the traffic is dropped.

As with the previous algorithms the paths are found in two stages. First the fully redundant paths are found to all destinations. Then to determine the forwarding of behaviour of switches that are not in the ring to a given destination the algorithm described in Algorithm 5.7 is used.

This algorithm is similar to the algorithms used to generate the overlapping paths earlier this chapter. Initially each vertex is added to a set called unvisited except the destination which is added to the queue potential_parents.

Each parent is dequeued from potential_parents and each child (a vertex in unvisited connected to parent by an edge) is tested for a ring to the destination.

When a child does not belong to a ring to the destination it is checked for a ring to its parent’s subdestination. If parent has redundant paths to the destination then its subdestination will be itself.

If child does not belong to a ring to the subdestination it is checked for a ring to parent. Once a ring is found then the vertex forwards traffic for the destination to an arbitrarily chosen parent of that ring. If child does not belong to a ring to parent, parent’s subdestination, or the destination, then child instead forwards all traffic directly to parent. The subdestination of child is then set to either the destination of the ring it is using to forward traffic or to itself if no ring could be found. child is then removed from unvisited and enqueued to potential_parents.
rings = \{\};
for each (destination in vertices) {
    count = 0;
    pathTrees = \{\};
    freeVertices = vertices \ destination;
    for each (root in freeVertices where (root, destination) is in edges) {
        newPathTree = a new PathTree with root as its root node and destination as its
        ringParentVertex;
        add newPathTree to pathTrees;
    }
    extendibleVertices = a queue containing every vertex in every pathTree in pathTrees;
    while (extendibleVertices is not empty) {
        parent = the vertex dequeued from extendibleVertices;
        parentPathTree = the pathTree that contains parent;
        for each (child in freeVertices where (child, parent) is in edges) {
            if (child is not already in a PathTree) {
                add child to parentPathTree as a child of parent;
                enqueue child to extendibleVertices;
            } else if (child’s PathTree is not the same as parentPathTree) {
                newRing = a Ring object with rank equal to count;
                if (the ringParentVertex of childPathTree is to the left of the ringParentVertex
                    of parentPathTree) {
                    leftTree = child’s PathTree; leftNode = child; rightTree = parentPathTree;
                    rightNode = parent;
                } else {
                    rightTree = child’s PathTree; rightNode = child; leftTree = parentPathTree;
                    leftNode = parent;
                }
                leftParentVertex of newRing = the ringParentVertex of leftTree;
                rightParentVertex of newRing = the ringParentVertex of rightTree;
                ringLocation = the location of leftTree’s ringParentVertex;
                append count * -1 to ringLocation; increment count;
                locationCount = 0;
                for each (vertex in the path between leftTree’s root node and rightTree’s root
                    node via child) {
                    append vertex to newRing’s vertices list;
                    set vertex’s location to ringLocation;
                    append locationCount to vertex’s location;
                    increment locationCount;
                    newPathTrees = split_path_trees (vertex, vertex’s PathTree);
                    pathTrees = pathTrees \cup newPathTrees;
                    remove vertex from freeVertices;
                    enqueue all vertices in newPathTrees to extendibleVertices;
                }
            } remove leftTree and rightTree from pathTrees;
        } add newRing to rings [destination ];
    }
}

Algorithm 5.6: Shared Paths algorithm ii pseudo-code
for each (destination in vertices) {
    subdestinations = {}; 
    subdestinations [destination ] = destination; 
    unvisited = vertices destination; 
    potentialParents = a list containing destination; 
    while (potentialParents is not empty){ 
        parent = the first element dequeued from potentialParents; 
        subdestination = subdestinations [parent ]; 
        for each (child in unvisited where (child, parent) is in edges){ 
            if (child belongs to a ring to destination){  
                subdestinations [child ] = destination; 
            }
            else if (child belongs to a ring to subdestination){ 
                traffic arriving at child for destination should be forwarded to subdestination; 
                subdestinations [child ] = subdestination; 
            }
            else if (child belongs to a ring to parent){ 
                traffic for destination arriving at child should be forwarded to parent; 
                subdestinations [child ] = parent; 
            }
            else{ 
                traffic for destination arriving at child should be forwarded directly to parent without redundancy; 
                subdestinations [child ] = child; 
            }
            remove child from unvisited; enqueue child to potentialParents; 
        }
    }
}
In this chapter the fault detection and fault recovery methods described in Chapters 4 and 5 are evaluated for suitability for use with the RouteFlow architecture. The results of this evaluation are then used to create a final design for a fault tolerance system to be added to RouteFlow.

6.1 Fault Detection

In this section the methods of evaluating the implementations of fault detection described in Chapter 4 are described and the results of these evaluations are presented and discussed. The criteria for evaluation are the speed and accuracy of detection, the required switch state, the load on the controller and on the network, and the ability of each method to identify the location of faults.
6.1.1 Speed of Detection

To determine the speed at which the approaches to fault detection can discover low levels of packet loss an experiment was performed on simple virtual networks. The experiment consisted of introducing faults into the networks and timing how long each fault detection method took to recognise the fault.

The network to evaluate Test Packet Injection consisted of two virtual switches connected by a single link. A controller was set up to monitor the link using Test Packet Injection. To test Traffic Colouring a similar network structure was used, but with a pair of directly connected Open vSwitch bridges in the place of each switch, one internal bridge and one the external as described in Section 4.1.6. Servers were connected to the first and last switches of both networks to generate a constant stream of traffic between each other.

Varying levels of loss were introduced to the link between the two switches, or between the two internal bridges in the Traffic Colouring test. The loss was created using NetEm (Hemminger et al., 2005). When the controller discovered the fault, the time between the introduction of loss and its discovery was recorded and the link would be restored to normal operation. Then after a random interval the test would be repeated.

Because detection of a single lost packet could be an indicator of a fault causing high levels of loss that has only just started or a fault that causes very low levels of loss that has been occurring for a long time, for the Test Packet Injection test two measurements were recorded. The first is the time for the controller to detect that the first probe has been lost, the second is the time for the controller to detect the loss of two probes. The loss would not be reset until the second probe was received. Because of
the significantly greater number of packets being counted by the Traffic Colouring method this was not repeated for the Traffic Colouring test.

Both methods perform tests in cycles, Traffic Colouring cycles consist of changing the colour being used on the switch, waiting the request interval, and retrieving the statistics from each switch. The cycles in Test Packet Injection involve sending ten probes, waiting the request interval then retrieving the statistics. The probes are sent one one-hundredth of a second apart. For both tests the request interval used was 1.1 seconds.

<table>
<thead>
<tr>
<th>Loss Rate</th>
<th>Traffic Colouring</th>
<th>Test Packet Injection 1st lost probe</th>
<th>Test Packet Injection 2nd lost probe</th>
</tr>
</thead>
<tbody>
<tr>
<td>100%</td>
<td>1.53</td>
<td>1.77</td>
<td>1.79</td>
</tr>
<tr>
<td>75%</td>
<td>1.54</td>
<td>1.72</td>
<td>1.72</td>
</tr>
<tr>
<td>50%</td>
<td>1.64</td>
<td>1.77</td>
<td>1.78</td>
</tr>
<tr>
<td>25%</td>
<td>1.71</td>
<td>1.78</td>
<td>2.01</td>
</tr>
<tr>
<td>10%</td>
<td>1.86</td>
<td>2.60</td>
<td>3.60</td>
</tr>
<tr>
<td>1%</td>
<td>3.13</td>
<td>12.63</td>
<td>23.48</td>
</tr>
<tr>
<td>0.1%</td>
<td>15.37</td>
<td>143.10</td>
<td>263.79</td>
</tr>
</tbody>
</table>

Table 6.1: Mean detection time in seconds

In neither test were any false positives received. The average detection times are shown in Table 6.1.

The difference between Traffic Colouring and Test Packet Injection detecting one lost probe at high levels of packet loss is largely due to the extra time it takes for the Test Packet Injection controller to send all ten probes. At the higher levels the speed of Traffic Colouring is clearly shown. Even at 10% packet loss, a considerable level of loss, there is a large difference in time between Traffic Colouring identifying the fault and Test Packet Injection discovering the second lost probe. This is an imperfect test as Traffic Colouring does not need to detect a second lost packet, but it still was significantly faster than Test Packet Injection at detecting a single lost packet.
6.1.2 Flow Table Requirements

The additional flow table entries required for Test Packet Injection are two entries per port and one entry for every other switch in the network. Traffic Colouring requires four entries per port, four entries for every other switch in the network and one entry to set the colour of traffic. Test Packet Injection is the clearly superior option in terms of reducing the numbers of flow table entries, but both methods do meet the requirement of increasing at a linear rate with the number of switches in the network.

6.1.3 Controller Load Requirements

Each cycle the controller for Test Packet Injection has to send ten 92 byte OFPT_PACKET_OUT messages, one 8 byte OFPT_BARRIER_REQUEST message, and one 64 byte OFPT_MULTIPART_REQUEST message to each switch in the network. Each switch responds with one 8 byte OFPT_BARRIER_RESPONSE message and a OFPT_MULTIPART_REPLY message with the size equal to 24 bytes plus 40 bytes for every other switch in the network.

The controller for Traffic Colouring sends one 109 byte OFPT_FLOW_MOD message, one 8 byte OFPT_BARRIER_REQUEST message, and two 64 byte OFPT_MULTIPART_REQUEST messages to each switch each cycle. The switches each reply with one 8 byte OFPT_BARRIER_REPLY message and two OFPT_MULTIPART_REPLY messages with size equal to 24 bytes plus 40 bytes for every other switch in the network.

A graph of the total control traffic per cycle for the two methods is shown in Figure 6.1. It is clear that in larger networks the load on the controller becomes a significant burden. This could be mitigated by reducing the frequency of path based fault detection and focussing instead on link based
6.1 Fault Detection

detection. The total size of the OFPT_MULTIPART_REPLY messages sent by each switch to the controller would reduce to 24 bytes plus 40 bytes for each internal port on the switch for Test Packet Injection and to 48 bytes plus 80 bytes for each internal port on the switch for Traffic Colouring. This reduces control traffic so that it scales at a linear rate with the number of links in the network, rather than at a quadratic rate with the number of switches.

Full path detection is still useful as it verifies that the switches are forwarding traffic so that it reaches its correct destination, but the rate of testing needs to be greatly reduced in large networks. To reduce the tests to a point that the load scales at a linear rate requires testing each full path once in some number of cycles that increases with the number of switches in the network.

Figure 6.1: Controller traffic per cycle.
6.1.4 Network Load

In the Test Packet Injection method the total load created by the probe packets themselves passing through the network is roughly proportional to the load on the controller. This should be distributed evenly throughout the network but a single link may carry up to half of this load. In such a case, that particular link must also carry any traffic between one half of the nodes in the network and the other half, suggesting that it should have sufficient bandwidth to handle the probes. However, if each link is being tested individually it is unnecessary to send so many probes through one link. The probes instead should focus on covering each link in the network evenly. This is not an issue for Traffic Colouring as it does not create any network load aside from the controller traffic.

6.1.5 Locating Faults

The locating of faults for both approaches is identical. Once a fault is detected on a path statistics can be retrieved for each link along the path to pinpoint the exact location of the loss. This requires the same time and number of messages for either fault detection approach. With link by link testing faults are located as soon as they are detected.

6.1.6 Other Considerations

There are two other significant advantages to using Traffic Colouring over Test Packet Injection. The first is the fact that Traffic Colouring monitors all traffic on the network. This allows it to find issues that might be missed by Test Packet Injection if for whatever reason the test probes are unaffected by them. For instance faulty memory may cause certain types of packets to
be corrupted but not others. If the probes are unaffected the fault will go undetected.

The other advantage of Traffic Colouring is that as well as detecting packet loss, it also retrieves a lot of information about the utilisation of the network. Future work could use this information to improve load balancing or other network functions.

Test Packet Injection has a major advantage over Traffic Colouring that it is able to test paths that are not in use. This is only relevant for testing full paths, not for testing individual links, provided each link is utilised by the path generation algorithm.

6.1.7 Conclusion

Due to its superior speed and comprehensiveness of detection, Traffic Colouring was chosen ahead of Test Packet Injection for use with RouteFlow. While Test Packet Injection had the advantage in terms of the flow table entries required, Traffic Colouring achieved an adequate standard. The benefits of Test Packet Injection in terms of controller load were offset by the additional network load it produced and the fact that neither method was acceptable when polling each path between switches and both were acceptable when focussing only on links. The disadvantage Traffic Colouring incurs due to not being able to monitor unused paths is mitigated by the shift of focus towards monitoring links rather than full paths.

The Traffic Colouring approach to be implemented in RouteFlow will be modified to poll the statistics for each link each cycle but not all of the full paths. The controller can poll some number of full paths each cycle, reducing the total load.
6.2 Fault Recovery

In this section the algorithms for generating protected paths through a network presented in Chapter 5 are evaluated for their suitability for use with RouteFlow.

Each algorithm was applied to a series of randomly created networks. Then the paths created were evaluated based on their lengths, the number of flow table entries per switch required to install the paths onto a network of OpenFlow switches, and how evenly the paths were distributed across the links of the network.

Of the four algorithms presented in Chapter 5, only three were tested, as the second Shared Paths algorithm was not designed early enough to be implemented.

6.2.1 Generation of Random Networks

The generation of the random networks was performed by a function that took the numbers of switches and links as arguments. The number of links must exceed the number of switches to ensure the graph created is connected. The function first creates a link between two switches, then adds each switch to the network by creating a link between it and a switch that is already part of the network. Then any remaining links are added between two randomly chosen switches.

6.2.2 Primary Path Length

The first test compares the length of the primary paths: the paths used by the VCs between each pair of switches when the network is fault free. For the paths created by the ring based forwarding algorithm the primary path
is the path taken by traffic originating at the source switch through the network. For the other algorithms, which produce pairs of paths between each pair of switches, the shorter of the two paths is chosen.

Three sets of tests were performed, each using networks with a set number of switches and increasing numbers of links. The first set used networks of twenty switches, the second used networks of one hundred switches and the third used networks of five hundred switches. Each set consists of tests of eighty different networks. The first with a number of links equal to 1.05 times the number of switches, the subsequent tests use a number of links increasing by .05 times the number of switches until the final test has 5 links for each switch.

![Graph](image)

**Figure 6.2:** Mean primary path length in networks of 20 switches

Figures 6.2 to 6.3 show the mean primary path length for each of these networks. The primary paths of the Shared Paths and Path Pairs algorithms are consistently shorter than the paths created by the Ring based Forwarding algorithm. The Ring Based Forwarding could build paths identical to
those built by the Path Pairs algorithm so the longer paths here must be a consequence of the selection of parents to advance to at each ring. Since it is completely arbitrary in both cases (in both implementations a random
parent on the innermost ring is chosen), the likely reason for the shorter on average primary paths is that the Path Pairs algorithm build two paths and chooses the shortest, whereas Ring Based Forwarding only builds one path.

### 6.2.3 Alternate Path Length

The second test compares the length of the alternate paths created by each algorithm. The tests were performed on the same networks created for the primary path tests.

Because the Ring Based Forwarding algorithm does not create a single alternative path, instead a random link on the primary path was chosen as the location of a potential fault. A new path was created that does not include that link and the length of that path was used for this test. If a new path was not able to be created without passing through the faulty link another link was chosen. If all links on the path have been chosen and no path has been able to be created, no alternate path was recorded for that primary path.

Figures 6.5 to 6.7 show the mean length of the alternate paths for all pairs of switches in each set of networks. The cost of prioritising the paths of higher ranked rings in the first of the shared paths algorithms is shown with these results, with the Shared Paths algorithm consistently creating significantly longer alternate paths.

The second algorithm for shared paths is likely to have significantly shorter alternative paths, though the option to choose which parent a path is advanced to means that the path pairs algorithm will always be able to at worst match the alternate path lengths of the shared paths algorithm.
The benefit of the Ring Based Forwarding algorithm is also seen in Figures 6.5 to 6.7. The alternate paths are much closer in length to the primary paths and are consistently advantageous over the alternative paths created
6.2 Fault Recovery

### 6.2.4 Flow Table Entries

Tests were performed to determine the number of flow entries required to install the paths generated by each algorithm. Firstly tests were performed on the networks used in the previous two subsections, then more comprehensive tests were performed on networks with fixed numbers of switches and links.

The first test was to find the mean number of flow entries per switch for each algorithm using the networks used earlier in this section. The results are shown in Figures 6.8 to 6.9. These graphs illustrate the unintuitive result that the first shared paths algorithm uses more flow table entries than the path pairs algorithm, even though it has the added constraint to reduce the total number of flows. Additional flow table entries are only required to install the paths on transit hops, not at the source switch. Longer
paths have more transit hops and therefore end up requiring more flow table entries even though the number of flow table entries per switch is restricted. It is unlikely that this outcome would have occurred with the
The Ring Based Forwarding algorithm has the advantage of the shortest paths and an upper limit of three flows per destination. This advantage could have been extended as Ring Based Forwarding could have been implemented with even fewer flows per destination at the cost of a small reduction in the level of redundancy. Even so, Ring Based Forwarding would still offer greater protection than any of the other algorithms.

While the mean number of flows per switch in the path pairs algorithm is lower than in the constrained version, it is possible that the lack of any constraint causes some switches to require particularly high numbers of flow table entries. RouteFlow currently has no way of resolving the issue of a particular switch running out of memory to store flow table entries, so any one switch running out of memory may cause faulty behaviour in the fabric. Figures 6.11 to 6.12 show the maximum number of flows
per switch in each of the networks used in previous tests. In networks with fewer than two links per switch, the Path Pairs algorithm requires significantly higher flow entries in the maximum case. Furthermore the
maximum case seems to grow at a greater than linear rate with the number of switches in the network. This graph also illustrates the flaw in the first Shared Paths algorithm. Some switches must have greater than two paths to each destination, despite the intended constraint to prevent this from happening, as the maximum number of flow table entries per switch for many of these networks is greater than twice the number of switches in the network.

To further evaluate the flow entry requirements in-depth tests were performed on networks of three sizes: a 20 switch network with 30 links, a 100 switch network with 150 links and a 500 switch network with 750 links. 1,000 networks of each size were created. The results of these tests are shown in Figures 6.14 to 6.16. These graphs illustrate the long tail in the distribution of flows across switches caused by the Path Pairs algorithm. In one instance a switch required 8,525 flow table entries to install the paths created by the Path Pairs algorithm in a network with 500 switches. As a
Figure 6.14: Flow table entries per switch in networks of 20 switches and 30 links

Figure 6.15: Flow table entries per switch in networks of 100 switches and 150 links

consequence of this the Path Pairs algorithm is unacceptable for use with RouteFlow.
6.2 Fault Recovery

![Graph showing flow table entries per switch in networks of 500 switches and 750 links.](image)

**Figure 6.16:** Flow table entries per switch in networks of 500 switches and 750 links

### 6.2.5 Link Load

![Graph showing paths per link in networks of 20 switches and 30 links.](image)

**Figure 6.17:** Paths per link in networks of 20 switches and 30 links

The final tests performed were to determine how evenly the algorithms
distribute the paths they generate across links in the network. This test was performed on the networks used to produce the CDF graphs in the previous subsection. The results of these tests are shown in Figures 6.17 to
These graphs indicate that the algorithm with the worst distribution of paths across links is Ring Based Forwarding. As Ring Based Forwarding could produce identical primary paths to the path pairs algorithm, these results could be improved with a different criteria for selecting which parent each switch forwards its traffic to. A potential criteria could be based on the work presented in *On Path Selection with Bandwidth Guarantees* (Ma & Steenkiste, 1997), but evaluation of this is left for future work.

### 6.2.6 Other Considerations

Ring Based Forwarding has two other key advantages over the other two approaches. The first is the greater redundancy provided by the fact that the path can be changed at each ring rather than only at the source. This allows RouteFlow to utilise more of the redundancy in the network. This is particularly important because RouteFlow is not able to respond to a situation where two switches cannot forward to each other except by disabling the ports on one of the switches.

The second advantage is that the path is only changed on the ring affected by the fault. When pairs of paths are created, if a link has a fault then every switch that sends traffic through that link needs to be modified to use their alternate paths. With Ring Based Forwarding only the switches that are on a ring that uses the affected link need to be modified. This reduces control traffic and should speed up the recovery from faults.
6.2.7 Conclusion

For every criteria evaluated, Ring Based Forwarding either outperformed the other algorithms or could have matched their performance with an alternative method of selecting which parent of each ring should be forwarded traffic. Unfortunately the second Shared Paths algorithm could not be evaluated, but it is unlikely to surpass Ring Based Forwarding on any of the criteria tested here, as it is effectively the same algorithm but with an added constraint when choosing which parent to forward traffic to.

6.3 Conclusion

The final design for the addition of fault tolerance for the RouteFlow architecture will implement protected paths using the Ring Based Forwarding algorithm created as part of this thesis and a fault detection process using Traffic Colouring in combination with `OFPT_PORT_STATUS` messages and fast fail-over group types to allow for expedited responses to complete link failure.

The Traffic Colouring method will primarily focus on detecting packet loss on each link in the network rather than on each path. This reduces the controller load and also works well in conjunction with the Ring Based Forwarding algorithm. When a fault is detected on a link only switches on the rings that use that link have their forwarding behaviour modified. Even when monitoring full paths the faulty link needs to be identified before any action can be taken.

The flexibility offered by the fact that Ring Based Forwarding allows a choice of which parent to forward traffic to could be used in conjunction
with the live updates of link utilisation provided by Traffic Colouring to balance load across the network. This could be the basis for future work in this area.

Other future work could include evaluating the second shared paths algorithm.
Conclusion

This thesis presents the design of a fault tolerance system for use in a software defined network with RouteFlow as its model application. The system is able to discover a variety of networks faults, can quickly identify their exact location, and take action to allow the network to continue to function. The system requires minimal configuration to run on any arbitrary network topology and scales efficiently to run on large networks.

The fault tolerance system uses Ring Based Forwarding a novel approach to building protected paths through the network. Each switch has a default path to each destination and an alternative path to a switch that is closer to the destination than itself. When a fault is detected, switches that are close to the fault shift the affected traffic onto their alternative paths. The main benefits of this approach are that only a small number of switches need to have their forwarding behaviour changed in the event of a fault, and the alternative paths are smaller, resulting in a reduction in the number of flow table entries per switch required to install the paths.
To detect when faults occur in the network the fault tolerance system uses Traffic Colouring: assigning all packets in the network a colour in alternating blocks and comparing the numbers of packets of each colour entering and exiting sections of the network. The system primarily focuses on monitoring the number of packets entering and exiting each link. Monitoring of full paths is also necessary in order to verify correct forwarding behaviour, but each path is monitored only intermittently in order to reduce controller load.

The techniques used in the design were chosen based on a thorough evaluation of existing approaches to fault detection and recovery. Other approaches that were evaluated but rejected include:

*Test Packet Injection:* This method of detecting faults involved injecting probe packets into the network and verifying their arrival at their intended destinations. This was rejected as it was found to be considerably slower at detecting faults than Traffic Colouring. The advantage that Test Packet Injection is able detect faults in unused paths was undermined by the fact that the final design focussed on link based testing.

*Path Pairs and Shared Paths:* These methods of building protected paths through the network required considerably more flow table entries per switch than Ring Based Forwarding while providing less resilience.

### 7.0.1 Contributions

The contributions of this thesis are as follows:

- The design of a fault tolerance system for use with RouteFlow was presented.

- A new algorithm for creating a path through a connected undirected
graph where an alternative path is found at each hop that does not include the next edge of the original path.

- New algorithms for generating pairs of minimally overlapping paths between each source and destination vertex pair in an undirected connected graph were created. The first generates an arbitrary pair of minimally overlapping paths and the second generates a pair of minimally overlapping paths with the added constraint that any path to a given destination vertex that includes a given transit vertex will follow one of the two paths between the transit vertex and the destination vertex.

### 7.0.2 Future Work

Future work stemming from this thesis includes:

- The second Shared Paths algorithm has not been fully evaluated. The evaluation of this algorithm could be based on the experiments performed in Section 6.2.

- The statistics collected by the Traffic Colouring module include counts of the numbers of bytes of traffic seen at various points in the network. The Ring Based Forwarding system could use this information to balance the traffic load more efficiently throughout the network.

- The fault tolerance system has not been fully implemented or deployed. Evaluation of the full system could expose weaknesses not anticipated by this thesis. Further testing is required before the system could be deployed in production.
References


Communications, 36(6), 656–665.

