Redundancy Control Using Software Defined Networking

Douglas Comer*, Rajas H Karandikar†, Adib Rastegarnia †
† Department of Computer Science, Purdue University, West Lafayette, IN, 47907
e-mail: * comer@cs.purdue.edu, †{rkarandi, arastega}@purdue.edu

***Abstract*—We present a technique of providing resiliency in a computer network which some mission-critical applications require. The technique involves the use of redundancy through packet duplication and transmission over multiple disjoint paths between a pair of nodes in a network. To create a general-purpose resiliency system, we introduce a new network element, a R*edundancy Controller* (*RC*) that establishes and maintains the redundancy end hosts request. An RC also monitors path performance and replaces poorly performing paths. End users are able to specify criteria required for the performance of paths. The paper reports on work in progress. It outlines the motivation behind the project and describes our implementation plan.**

***Index Terms*—Software Defined Networking, network reliability, redundant paths, mission-critical applications, OpenFlow.**

I. Introduction

The Internet has been tremendously successful in providing connectivity between pairs of end systems. A key to success lies in the use of the Internet Protocol (IP), which handles addressing and defines a virtual packet format that is independent of the underlying hardware. Despite its advantages, IP follows a best-effort delivery model, which means datagrams can be lost, delayed, duplicated, or can arrive out of order. Most applications rely on Transmission Control Protocol (TCP), a layer 4 protocol. to guarantee reliable, in-order data delivery. To ensure all data reaches the destination correctly, TCP employs techniques of congestion avoidance and flow control as well as the key technique of timeout-and-retransmission.

Although most applications work well with the service TCP provides, mission-critical applications cannot tolerate the long delays associated with retransmission. In particular, applications associated with the Internet of Things (IoT) often need undelayed, uninterrupted service. They simply cannot wait for TCP to timeout and retransmit lost data. For example, consider the following uses of a network in which constant communication is essential.

- Remote surgery (a.k.a Telesurgery) consists of a robot surgical system controlled by a surgeon who may not be physically present at the same location as the robot. One application of telesurgery arises from medical support to space missions [1].
- Rescue robots are used in natural or man-made disasters where the disaster leads to an environment that is hazardous to humans [2]. One such example is the nuclear power reactor meltdown following tsunami in the Fukushima nuclear power plants in Japan. Robots were sent in to perform the repair work because radiation levels were dangerous for humans. The robots need constant network connectivity because they are controlled remotely and must perform delicate repair operations with precision.

If mission-critical applications use TCP to provide reliable data delivery, communication will experience delays when packet drops occur or the network path becomes congested. Retransmission delays are unacceptable for time critical applications like those mentioned above, especially in cases where delays affect human life or safety. Such applications typically impose an upper bound on the acceptable latency. It is important to understand that the problem cannot be solved by making protocol improvements to TCP because protocols cannot use retransmission to compensate for delays caused by congestion or loss along the underlying network path.

An effective solution to provide uninterrupted service in a system consists of using redundancy. Many network elements already use redundant hardware that maintains hot or cold stand-by mechanisms in high-priority situations. Applying the same principle broadly to network packets provides a solution to the problem described above. If each packet is duplicated and sent along disjoint paths to the destination, the probability of delay due to loss or congestion can be reduced. Going one step further, a feedback system can be used to monitor the performance of each path and replace any path that is performing poorly with a new path that performs better.

A key requirement is needed to make the redundant solution viable: duplicated packets must be sent along disjoint paths. Traditional IP routing protocols (e.g., RIP and OSPF) use a next-hop forwarding paradigm that forwards a packet based on the IP destination address. Thus, in a traditional IP routing system, all packets to a given destination follow the same path because they contain the same destination address. To ensure copies of a packet follow disjoint paths, the network system must support a forwarding paradigm that honors multiple paths and allows software to map each redundant copy of a packet onto a specific path. The paradigm known as Software Defined Networking (SDN) offers one way to achieve the goal.

Traditional networks utilize distributed routing algorithms to compute next-hop forwarding tables and dedicated hardware that uses the tables to forward packets, monitor data flows, and detect link failures. A vendor who creates

978-1-4673-9292-1/16/$31.00 ©2016 IEEE

a traditional network element combines routing protocols, forwarding mechanisms, and management software into each device. SDN decouples control functions from data forwarding, allowing software modules running outside a given network element to specify and configure forwarding rules rather than integrating control functions in a device that also performs the data forwarding function [3], [4]. SDN simplifies and unifies control of a network by allowing a centralized software system to apply uniform rules across a set of network switches. When using SDN, individual data flows can be configured in a set of network devices and the characteristics of the devices can be altered by a centralized controller – a piece of software running on a conventional server (possibly a VM) that accepts policies from a network administrator and implements the policies on network devices without requiring an administrator to access and and configure each individual device. The OpenFlow protocol [5] provides the most widely-used mechanism to realize the separation of control and data planes. A logically centralized controller can use OpenFlow to add or remove data plane forwarding rules in OpenFlow-enabled switches. The forwarding rules specify the use of fields in the layer 2 frame header, the IP header, the TCP header, or other information, including the switch port on which a packet arrives. We have chosen to use OpenFlow to allow our software system to install and manage redundant paths.

The rest of the paper is organized as follows: the following section presents a brief survey of related work. Section III presents proposed redundancy control framework. Programming language and experimental environment are described in Section IV. Finally, Section V concludes the paper.

II. Related Work

Networking researchers and engineers have explored a variety of ways in which redundancy can be used in computer networks. We present a few related projects here.

Multipath TCP (MPTCP): provides higher throughput by dividing a single TCP flow into subflows carried on multiple paths between the peers. Although MPTCP primarily focuses on increased performance, resilience and improved resource utilization can be achieved by mapping subflows to the same sequence space [6]. MPTCP is an extension of TCP, and as such cannot be used for other Transport Layer protocols, such as UDP. Our solution works with any Transport Layer protocol and does not require any changes to the protocols whatsoever.

Duplicating RTP Streams: The Real-Time Transport Protocol(RTP) is used to transport real-time multimedia sessions like IPTV. The work acknowledges that that some mission-critical applications cannot tolerate delays due to packet loss or network congestion. The work uses duplicate RTP streams to achieve resiliency. However, conventional IP forwarding is used in the work. All duplicate packets have the same IP header, which means that instead of sending the streams over diverse paths, all copies are sent along the same underlying path [7]. Our solution uses a new network element and Software Defined Networking to create disjoint paths and to send duplicated packets across disjoint paths.

Fast Recovery in Software Defined Networking: In [8] a fast recovery method is proposed based on combination of failover schema that relies on link failure detection and primary and backup paths that are installed by using a central OpenFlow controller. The method proposed in the paper uses Bi-directional Forwarding Detection for link failure detection.

III. Proposed Redundancy Control Framework

In this section, we present a short description of the architecture and functionality of our proposed framework.

A. Redundancy Controller

Duplicating packets and sending them along disjoint paths reduces the probability of delays due to packet loss or link failures. The duplication of packets in a flow can be achieved in the source host, provided protocol software on the host can be modified. The need to change network software in an end host makes the approach unattractive. As an alternative, duplication of packets can be performed by a network element along the path from source to destination, such as an OpenFlow-enabled switch. On the sending side, Openflow [5] allows a manager to configure rules that cause a packet to be duplicated and the copies forwarded to multiple output ports. On the receiving side, however, OpenFlow does not provide feedback on the performance of a path. The statistics that OpenFlow provides are limited to basic information, such as the number of packets that have matched a particular flow rule.

One way to measure the performance of a path, consists of counting packets to compute the number of packets dropped along the path. Although it suffices as a measure for lossy paths, packet loss does not account for delays caused by congestion along the path. Congestion can be estimated by measuring the differential delay between packets arriving on redundant paths. We define differential delay as follows: if a packet P arrives at time t, and a duplicate of P arrives at time $t + d$, the instantaneous differential delay is d. We are using the existence of disjoint paths to measure the relative congestion along the slowest path. Unfortunately, the current capabilities of OpenFlow make it impossible to use an Openflow-enabled switch to compute differential delay. Therefore, we propose a new network element called *Redundancy Controller* (RC).

We will place an RC at each attachment point on a network, which means a pair of RCs will lie between any two hosts, one near one of the hosts and the other near the other host. An RC is responsible for setting up multiple disjoint paths as needed, duplicating outgoing packets and forwarding each copy along one of the paths, monitoring path performance, and replacing paths that perform poorly relative to other paths

of the same flow. Each host will be connected to an RC, and the RC will have at least two independent connections to the network. In essence, RCs will be present on the edge of the network, and when a host requests redundant service, the RC near the host will set up redundant paths to the RC near the destination host. That is, a flow between two hosts will involve two RCs at either end of the network. In terms of data handling, the RCs operate symmetrically – each RC will duplicate packets that arrive from the local host and send the packets along redundant paths to the RC near the other host (the paths will be installed using OpenFlow before packets flow across the connection). Although multiple copies of a given packet arrive at the RC connected to the destination, the RC only forwards one copy to the destination host. The RC logs the each packet arrival along with a timestamp, and uses the log to calculate the differential delay. However, subsequent copies are only used to assess path performance; a host only receives the first copy that arrives.

Observe that our architecture places an RC inline (i.e., in the data path). Because an RC performs operations on each incoming packet (either duplicating the packet or logging the timestamp), an RC introduces additional latency. For our prototype, an RC is an $x86$ machine running Linux. Clearly such a system can become the bottleneck in a network of high-speed switches. Using techniques like Kernel Bypass Networking for the Ethernet driver [9], the bottleneck can be significantly reduced or eliminated. Kernel-bypass can handle network traffic of 10 Gbps, which means that an RC handling each packet will not introduce any significant delays. At higher network speeds, the RC functionality will have to be implemented in hardware (e.g., in the switch forwarding hardware). Our project focuses on functionality; we are satisfied that higher speed solutions are possible.

Figure 1 illustrates the role and position of RCs in the network topology.

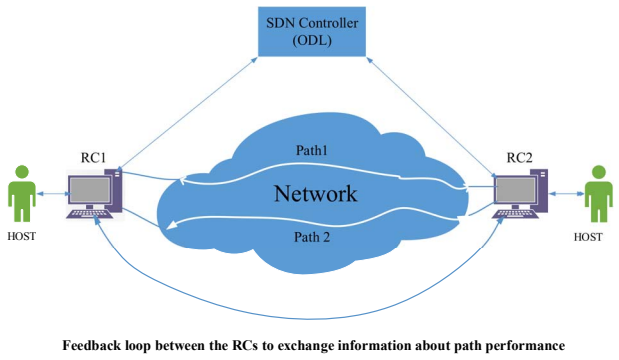

Fig. 1. The role and position of RCs in the network topology

B. SDN Controller

OpenDaylight [10], which is an OpenFlow controller is used to fetch the network view, add and remove flow entries in the network switches. OpenDaylight provides a RESTful API for network management applications to perform the above mentioned operations. The RC software uses the libcurl library [11] to perform REST operations.

C. Data Plane Network

Our prototype network consists of 7 Hewlett-Packard (HP) Switches (six HP 2920 series, one HP 3800 series). The switches support OpenFlow version 1.3.

D. Software Architecture of Redundancy Controller

For this project, an RC is implemented with an $x86$ machine running Linux. Each RC has at least three network interfaces (one on the host side and two on the network side). The RC software is divided into following modules:

- **Network View**: The network view module maintains a current view of the network and stores the information in the form of a database of nodes and links. The module uses the REST API provided by OpenDaylight to fetch the information. OpenDaylight provides the information in the JSON or XML format; the current RC implementation supports both data formats.
- **Path Module**: The path module is responsible for finding edge-disjoint paths between two RCs. The input consists of a network topology represented as a list of nodes and links, plus an adjacency list to represent current connectivity among nodes. Because links are un-weighted, a simple Breadth-First Search (BFS) from the source node gives the shortest path to the destination node. Once a path has been allocated, the links on that path are marked as *allocated*. A subsequent BFS on the graph with the same source and destination will ignore allocated links, and will therefore generate an edge-disjoint path. Thus using BFS and marking the state of each link as allocated or unallocated suffices to generate multiple edge-disjoint paths between any two end nodes.
- **OF Flow Module**: The OF flow module is responsible for installing and removing end-to-end OpenFlow flows in the switches using the OpenDaylight controller's REST API.
- **Client Handler**: Requests for redundant paths from the hosts are handled by a client handler. The module tracks all flows currently active in the RC. A client provides criteria for each flow: the degree of redundancy, and an upper bound on acceptable differential delay.
- **Duplication and Aggregation Module**: The duplication and aggregation module handles packets flowing in both directions. The module is responsible for duplicating and forwarding packets that arrive from the host and for logging the timestamp information from duplicate packets that arrive over the network. Modules listed above populate the forwarding rules and the information necessary for assessing path performance. The module uses the kernel bypass mechanism mentioned in the previous section to handle packets at high speed.

IV. Programming Language and Experimental Environment

For this project we use version 1.3 of OpenFlow. The prototype network contains two RCs, each of which has two separate connections to network switches. Having separate physical connections to network switches allows us to establish edge-disjoint paths between the redundancy controllers.
Figure 2 shows a screenshot of the network topology as displayed by the OpenDaylight controller. OpenDaylight displays each interface on an RC as a separate "host" because our RC appears to be a host attached to the network, and because the RC does not act as a switch, OpenDaylight has no way of mapping the interfaces to a physical device. To help clarify, the interfaces that belong to each RC are circled and labeled.

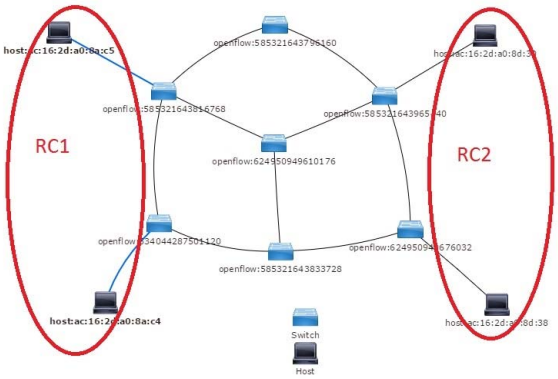

Fig. 2. Screenshot of the Network Topology as displayed by OpenDaylight

V. Conclusion and Future Work

We expect more mission-critical applications to communicate over computer networks in the future. Reliable, timely service is a key requirement for such applications. One way to achieve reliability uses redundancy. This project explores path redundancy. We have proposed an SDN-based framework that provides path redundancy in computer networks. To meet the goal of timely, reliable delivery, our system introduces a new network device known as a Redundancy Controller (RC). An RC uses SDN technology to establish and maintain redundant paths between endpoints according to specifications given by an application. To provide redundancy between end hosts, an RC establishes disjoint paths through the network, and duplicates outgoing packets so a copy can be forwarded along each of the paths. To establish paths, RC communicates with an SDN controller, using REST API. The RC retrieves the current network topology, runs an algorithm that uses the topology to compute a set of disjoint paths, and then communicates with the SDN controller to install end to end flows. The proposed solution is protocol-independent in the sense that it works with an arbitrary transport-layer protocol without any change to the protocol itself. Our project is a work in progress. Some of the RC software modules have been implemented, and most of the major pieces of the system are in place. We plan to finish the software, measure performance using the experimental facility described in the paper, and then to extend the project to a larger network that offers a more complex topology.

Acknowledgment

The authors would like to thank Hewlett-Packard (HP) for providing financial support and HP switches. In addition we would like to thank two undergraduate students at Purdue - Eric Templin and Nathaniel Cherry for volunteering to help with switch configuration and the Redundancy Controller software. Finally we thank The Information Technology at Purdue (ITaP) department for donating switches and helping set up the experimental environment.

References

[1] T. Haidegger, J. Sándor, and Z. Benyó, "Surgery in space: the future of robotic telesurgery," *Surgical endoscopy*, vol. 25, no. 3, pp. 681–690, 2011.
[2] R. R. Murphy, S. Tadokoro, D. Nardi, A. Jacoff, P. Fiorini, H. Choset, and A. M. Erkmen, "Search and rescue robotics," in *Springer Handbook of Robotics*. Springer, 2008, pp. 1151–1173.
[3] F. Hu, Q. Hao, and K. Bao, "A Survey on Software Defined Networking (SDN) and OpenFlow: From Concept to Implementation," in *Communications Surveys & Tutorials, IEEE*, vol. PP, no. 99, 2014, p. 1.
[4] Wenfeng Xia, Yonggang Wen, Chuan Heng Foh, D. Niyato, and Haiyong Xie, "A Survey on Software-Defined Networking," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 1–1, 2014.
[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.
[6] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "Tcp extensions for multipath operation with multiple addresses," Tech. Rep., 2013.
[7] A. Begen and C. Perkins, "Duplicating rtp streams," Tech. Rep., 2014.
[8] N. L. van Adrichem, B. J. van Asten, and F. A. Kuipers, "Fast Recovery in Software-Defined Networks," in *2014 Third European Workshop on Software Defined Networks*. IEEE, Sep. 2014, pp. 61–66.
[9] "Kernel bypass Networking." [Online]. Available: http://lukego.github.io/blog/2013/01/04/kernel-bypass-networking/
[10] "OpenDayLight." [Online]. Available: http://www.opendaylight.org/
[11] "Libcurl." [Online]. Available: http://curl.haxx.se/libcurl/