OPENFLOW-BASED DISTRIBUTED AND FAULT-TOLERANT SOFTWARE SWITCH ARCHITECTURE

A Thesis

Submitted to

The Faculty of the Department of Engineering Technology

University of Houston

_____

In Partial Fulfillment

Of the Requirements for the Degree

Of

Master of Science in Engineering Technology

_____

By

Gandhimathi Velusamy

April, 2014

OPENFLOW-BASED DISTRIBUTED AND FAULT-TOLERANT SOFTWARE SWITCH
ARCHITECTURE

An Abstract of a Thesis

Submitted to

The Faculty of the Department of Engineering Technology

University of Houston

In Partial Fulfillment

Of the Requirements for the Degree

Of

Master of Science in Engineering Technology

By

Gandhimathi Velusamy

Spring, 2014

Abstract

We are living in the era where each of us is connected with each other virtually across the globe. We are sharing the information electronically over the internet every second of our day. There are many networking devices involved in sending the information over the internet. They are routers, gateways, switches, PCs, laptops, handheld devices, etc. The switches are very crucial elements in delivering packets to the intended recipients. Now the networking field is moving towards Software Defined Networking and the network elements are being slowly replaced by the software applications run by OpenFlow protocols. For example the switching functionality in local area networks could be achieved with software switches like OpenvSwitch (OVS), LINC-Switch, etc. Now a days the organizations depend on the datacenters to run their services. The application servers are being run from virtual machines on the hosts to better utilize the computing resources and make the system more scalable. The application servers need to be continuously available to run the business for which they are deployed for. Software switches are used to connect virtual machines as an alternative to Top of Rack switches. If such software switch fails then the application servers will not be able to connect to its clients. This may severely impact the business serviced by the application servers, deployed on the virtual machines. For reliable data connectivity, the switching elements need to be continuously functional. There is a need for reliable and robust switches to cater the todays networking infrastructure. In this study, the software switch LINC-Switch is implemented as distributed application on multiple nodes to make

it resilient to failure. The fault-tolerance is achieved by using the distribution properties of the programming language Erlang. By implementing the switch on three redundant nodes and starting the application as a distributed application, the switch will be serving its purpose very promptly by restarting it on other node in case it fails on the current node by using failover/takeover mechanisms of Erlang. The tolerance to failure of the LINC-Switch is verified with Ping based experiment on the GENI test bed and on the Xen-cluster in our Lab.

Table of Contents

List of Figures

Chapter 1: Introduction

There is a paradigm shift about to happen in networking that is being propelled by powerful network white boxes (e.g. Intel's Open Network Platform, Pica8) that can be programmed to replace expensive ASIC-based middle boxes and switches. When networked applications run on such programmable platforms the application-specific requirements can be fulfilled with more transparency in the network operations than before. For example, application-dependent flow path setup is possible with programmability built directly into the application such as a Hadoop-acceleration using OpenFlow protocol in [1]. Furthermore, such software-based networking systems can become fault-tolerant and highly available using the vast research and techniques developed in the area of distributed systems. A distributed system consists of a collection of autonomous computers connected by a network while running distributed middleware. The system enables the computers to share their resources and coordinate their activities so that the entire distributed system appears to a user as a single fabric. In this respect, one way to achieve fault tolerance is through introduction of redundancy, namely, running multiple software switches that are part of a distributed system. If one software switch fails, another one can start running (failover). The cost to host such extra software switches on the system is much less than the hardware counterparts.

To the best of our knowledge, there are two leading open source implementations of software switches: OpenvSwitch (written in C), and LINC (open-sourced by [2] and written in Erlang). Erlang programming language has built-in support for concurrency and distribution [3].

An Erlang system can instantiate multiple nodes and provide the abstractions to specify a failover and take-over mechanism between the nodes. An Erlang node has a built-in protocol to communicate with another Erlang node.

This thesis presents a novel distributed software switch architecture that guarantees fault tolerance using the Erlang OTP (Open Telecommunications Platform) with OpenFlow-capable LINC switches.

We present the functionality of switches in transmitting the frames between two end hosts in general, then the advantage of moving into Software Defined Networking, functionality of OpenFlow switches. Then we explain Erlang' distributed system components and relevant operations. Next, we describe our experimental setup realized on the GENI (Global Environment for Network Innovation) infrastructure followed by our measurements of failover and take-over times. We conclude our work with some future work discussions.

Chapter 2: Literature Survey

Fault-tolerance is achieved in distributed systems by using redundant hardware/software for a long time in the history. To achieve fault-tolerance in Tandem computers, the hardware architecture was designed using the following principles [4]:

They system was decomposed into hierarchical modules having Mean Time Between Failures is more than a year; the modules sent Keep alive messages to other modules to let their presence; The modules exhibited fail-fast behavior that means they should work right or they should fail; Extra redundant modules were configured to pick up the loads of the failed modules with the takeover times of seconds to make the MTBF to be of millennia. By adopting above principles a MTBF of decades or centuries was achieved.

The same principles were used in designing the Erlang OTP libraries used to build fault-tolerant applications like AXD301in Ericson [5]. Fault-tolerance could be achieved by detecting failure of one machine by the other machine in the network provided if it has sufficient data to carry over the job of failed machine without noticed by its users.

In [6], a survey has been made to analyze the techniques used in replicating the objects/services in distributed systems during mid-eighteen century. It claims that linearizability as the correctness criterion for replicated services which gives an illusion of replicated objects as a single system to its clients. The failure over of one object to other will be transparent to the client. There are two techniques explained in the paper namely primary backup and active backup for replicating objects/services.

Distribution will be considered for sharing the resources like data, hardware or computing in an organization as well as to make the system fault-tolerant to failure. It is highly impossible to achieve fault-tolerance without redundancy, distribution is considered as a means for achieving fault-tolerance by replicating the system on more machines. In Delta-4 project [6], fault-tolerance is achieved by replicating software components on different machines interconnected by local area network. In this project, a sub layer protocol called Inter Replica Protocol (IRP) is used to coordinate messages exchanged between replicated endpoints on different nodes and hides the replication to the endpoints of the messages. It also emphasized that for the fault-tolerance to work smoothly, the distribution has to handle the network partitioning.

Fault-tolerance in switched networks has been studied from a distributed system perspective since the beginnings of networks. The most common solution to switch failures has been introduction of redundancy through extra ports, interfaces, and switch hardware [7].

Vicis, an ElastIC-style Network On the Chip (NoC) can tolerate loss of many network components by wear out introduced hard faults by employing N- Modular Redundancy based solutions in its network and router replications [8]. Built-in-self test is used for locating the hard faults and techniques like port swapping is used to mitigate the faults. The routers work together to run distribution algorithms to solve network-wide problems as well as protecting network from critical failures occurring in individual routers.

To avoid Layer 2 loop formation caused by the redundant connection of switches, the following protocols were used in the Clemson University Network [9]:

- Unidirectional Link Detection (UDLD) prevents one-way links to ensure no loops.

- Hot Standby Router Protocol (HSRP) makes it possible for redundant routers to act as a single virtual router.

- Common Spanning Tree (CST) implements one spanning tree for an entire physical extended LAN.

- Enhanced Interior Gateway Routing Protocol (EIGRP) for routing based on distance-vector IP routing.

- Per VLAN Spanning Tree plus (PVST) allows distinct spanning trees to be formed on multiple VLANS on the same physical network.

Continuing on this premise of redundancy, some considerations on when and how failover may happen are: "hot standby" where the secondary takes over with no data loss; "warm standby" where secondary takes over with some data loss; "cold standby" where secondary is started up when primary is detected to be off/failed/non-functional and then switched into services.

In [10], an industrial case study has been made in the project called ADVERTISE, a distributed system for advertisement transmission to on-customer–home set-top boxes (STB) over Digital TV network (iDTV) of a cable operator. The system was built using Erlang OTP. The size of the communication network was a challenge to manage as it had 100,000 customers and it was operational for 24/7. The system was designed to be resilient to node failure as well as network partitioning. It has been explained in [10] how to protect the distributed system from such failures. The CAP theorem was explained and the possibilities of having the properties together was

discussed. In a distributed system that share data, consistency should be preserved by not permitting two different operations on the same data at any time. Consistency is achieved by transaction mechanism which updated the data on all nodes beforehand. Availability is the requirement that the system should present to answer the requests sent from the clients. The answers should have meaning full values other than time out or not reachable messages. Partition-tolerance is the key property in CAP theorem. The resiliency the system possess against network partitions. If the nodes cannot send or receive messages due to the unavailability of network then the previous two properties could not be achieved.

Fault tolerance in OpenFlow-based networks has been studied recently from controller failure perspective [11]. The failover and take-over during a switch failure is possible by creating copies or duplicate threads of software switches with an underlying distributed redundancy management system such as a CPRecovery component and others listed in [11].

Chapter 3: Forwarding Elements

The role of the switch is to receive the incoming Data-link layer frames and forward them on to outgoing links as shown in figure1. The hosts and routers send and receive packets via switches in a LAN and the switches are transparent to them.
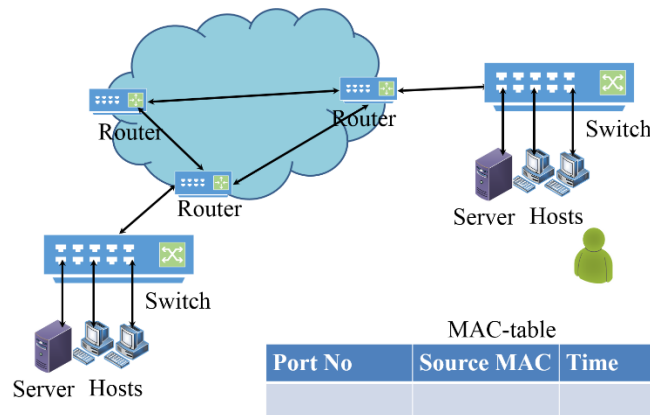
Figure 1 Role of the Switches in Networking

The two main functions of the switches are [12]:

1. **Filtering**

    The switch uses filtering function to decide whether to forward or drop an incoming frame.

2. **Forwarding**

    The switch uses the forwarding function to decide to which output port it has to send the packet to reach a particular destination based on the incoming port and destination MAC address.

Both Filtering and Forwarding are done with a switch table also known as MAC-table. The MAC-table contains entries for some but not necessarily all hosts and routers on a LAN. The MAC-table entry contains 1. MAC-address of the host connected to an interface, 2. The interface Number and 3. The time at which the entry was placed in the MAC-table. The switch forwards packet arriving at its input port to one of its output port based on the 48 bit destination MAC address in the frame.

The switch has three options to deal with a packet which arrives on an interface *x*.

I. **Broadcasting**: If there is no entry found in the MAC-table for the destination MAC address found in the frame, and then switch forwards the frame to all of its interfaces except to the interface on which it has arrived in.

II. **Filtering**: If there is entry for the MAC-address contained in the destination MAC field but it is associated to *x* itself, then the switch simply drops the frame.

III. **Forwarding**: There is an entry in the table for the MAC-address contained in destination MAC field but it is associating with an interface *y,* which is different than *x.* In this case the switch directs the frame to the interface *y.*

The interfaces of the switches have buffers attached to them to store the incoming frames if they arrive faster at the interface than the processing speed of the switch(the forwarding speed of the switch) to avoid the packets from being dropped at interfaces.

3.1 Self-Learning

The switch builds the Mac-table by self-learning that means it builds the table automatically and dynamically without the intervention of network administrator or any configuration protocol. The table will be built as follows:

I.   The switch table is initially empty.

II. The switch stores the following details when it receives a packet at its interface very first time: 1. The MAC-address in the frame's source address field, 2. the interface on which the frame came in, 3. the time at which the entry is created. If every host in the LAN sends a frame, then every host will be recorded in the table. There will be a timer with the value of 'mac-aging-time is started. Whenever the switch receives a frame for the MAC-address available in the table from the same interface, it updates the time it received and restarts the timer.

III. The switch deletes an entry for a MAC-address if it didn't receive any frame before the aging time expires from the host for which the MAC-address belongs to.

It also update the entries for any changes like if a host is moved from one port to another or a host is replaced by another host with a different MAC address. The port to MAC-address association will be updated for the changes once the MAC-aging time expires. The aging mechanism helps the switch to keep only the entries for current active hosts on the network. The aging time should be selected appropriate to the requirements. Too lengthy aging time keeps the entries for longer times in the table and thus exhausts the table space and also failed to update its

entries to accommodate the latest network changes. On the other hand too shot aging interval may result in removal of valid entries, causing unnecessary broadcasts, which may affect switch performance. The mac-aging time selected should be longer than the maximum interval in which the hosts normally transmits packets to avoid flooding in a network. For example, the traffic to the printers is less frequent, so the aging should be kept long to avoid flooding if the printer is idle. In the same way in data centers the servers connected to the switches are stable, so lengthy aging time avoids frequent flooding and thus helps to utilize the bandwidth for other useful traffic.

Chapter 4: Software Defined Networking

4.1 Need for Change in Networking

The conventional networks are built with Ethernet switches connected in hierarchical structure to meet the static Client-Server model of computing. But they are not suitable for today's dynamic nature of computing and storage needs. The outburst of mobile devices, content and server virtualization and cloud services are among the key trends driving the need for new network paradigm [13].

4.2 Limitations of Current Network

The following are the limitations of current networks to meet the challenges imposed by the evolving computing and storage requirements, datacenters, campus and carrier environments [13].

Complexity: Adding new devices or to support moving devices and implementing network-wide policies are complex, time consuming, needs manual efforts. So network changes will results in service interruption and normally discouraged to undergo.

Lack of ability to-Scale: The networks become more complex with the addition of hundreds and thousands of network devices that must be configured and managed to meet the requirements of datacenter's dynamic traffic. The time-honored approach of over subscription to provision scalability is not efficient with today's unpredictable data center traffic.

Vendor Dependence: Enterprises are willing to deploy new capabilities and services to cater their business needs or user demands very quickly but the lengthy vendor's equipment product cycles, lack of standard, open interfaces limit the ability of the network operators to tailor their network to their custom environment.

4.3 SDN

According to the definition given by Open Network Foundation [13], Software-Defined Networking is an emerging architecture that is dynamically manageable, cost effective, and adaptable, making it ideal for the high bandwidth, dynamic nature of today's applications. SDN architecture separates the network control plane from forwarding plane. By decoupling the control functionality from forwarding functionality, the network control has made directly programmable. As the switches are now concentrating only forwarding, the speed of the forwarding will be increased and the efficiency of the switches are improved. It also abstracts the underlying infrastructure to applications and network services. The OpenFlow protocol is a foundation element to build SDN solutions. The architecture offers the following features:

- **Directly Programmable**: Network control is directly programmable because it is decoupled from forwarding functions.

- **Agile**: Abstracting control from forwarding administrators are able to dynamically adjust network-wide traffic flow to meet changing needs.

- **Centrally managed**: The software based SDN controllers maintain a global view of the network by centralizing the intelligence in them.

- **Programmatically configured**: Network managers can configure, manage, secure and optimize the network resources very swiftly via dynamic programs on their own without depending on proprietary software.

- **Open-standards based and vendor-neutral**: When implementing through open standards, SDN simplifies network design and operation because instructions are provided by SDN controllers instead of multiple, vendor specific devices and protocols.

- **Reduces Capital Expenditure**: By allowing network functions to run on off-the shelf hardware SDN helps to reduce the capital expenditure spent on networking infrastructure.

- **Reduces Operational Expenditure**: SDN has made it possible to design, deploy, manage and scale networks with reduced operational cost by supporting automation and algorithm control through increased programmability

- **Enable Innovation**: It helps organizations to create new types of applications, services and business models.

4.4 OpenFlow Switches

The OpenFlow protocol allows using any type and brand of data plane devices, because the underlying network hardware is addressable through the common abstraction it provides. Importantly, it facilitates the use of bare metal switches and eliminates traditional vendor lock-in, and gives the freedom of choice in networking similar to the other areas of IT infrastructure, such as servers. There are two types of OpenFlow switches.

- OpenFlow Enabled Switches: Hardware-based commercial switches that use the TCAM and the Operating system of the switch/router to implement the Flow table and the OpenFlow protocol. This type of switches supports Layer 2, Layer3 along with OpenFlow protocol to isolate the experimental traffic from production traffic.
- Software-based (Dedicated OpenFlow) Switches: Software-based switches that use UNIX/Linux systems to implement the entire OpenFlow switch functions. E.g. OpenvSwitch, LINC-Switch, Indigo, SoftSwitch.

In general, OpenFlow Switch consists of three parts [14]:

1. A **flow Table**, with an action associated with each flow entry, which defines how to process the flow i.e., whether to forward to a particular output port or to drop or to any of reserved ports.
2. A **secure channel** that connects the switch to a remote controller, which allows commands and packets to be exchanged between the switch and the controller.
3. **OpenFlow protocol**, which provides an open and standard way for a controller to communicate with the switch.
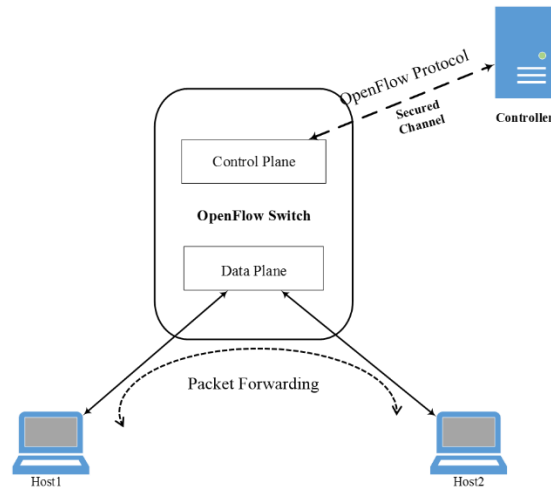
Figure 2 OpenFlow Switch, flows are inserted by the Controller from remote

4.4.1 Dedicated OpenFlow Switches

A dedicated OpenFlow switch is a dump data path element that forwards packet between ports as dictated by a remote controller as showed in the figure 2. In this environment flows are broadly defined and are limited by the capabilities of the particular implementation of the flow table. For example a flow could be a TCP connection, or all packets from a particular IP address or MAC address. A dedicated OpenFlow switch does not support normal Layer -2 and Layer-3 processing.

Each Flow entry has a simple action associated with it; the three basic actions are:

1. Forward this flow's packet to a given port or ports. Thus packets are routed through the network at line rate.

2. Encapsulate and forward this flow's packets to a controller in a secured connection. Normally first packet in a new flow will be sent to the controller and controller will decide whether the flow will be added to the table or not. In some cases all the packets could be sent to controller for processing based on the requirement imposed by the application.

3. Drop this flow's packets. This could be done for security to avoid denial of service attacks.

4.4.2 OpenFlow-enabled Switches

When the commercial switches, routers and access pointes are enhanced with OpenFlow protocol, flow tables and secured channel then they are called as OpenFlow enabled switches. Normally the flow table will re-use the existing TCAM; the secure channel and protocol will be ported to run on the switch's operating system. To isolate experimental traffic from production traffic the OpenFlow-enabled switches added the fourth action:

4. Forward this flow's packets through the switch's normal processing pipeline.

OR, the production traffic and experimental traffic can be isolated by defining separate VLANs for them. By defining either the above fourth action or adding VLAN tags, the switch will allow the regular production traffic to be processed in the usual way and at the same time allows the experimental traffic to be controlled by the way the experimenter wishes to process by the OpenFlow controller in a manner the experimenter wants to control.

Chapter 5: LINC-Switch

LINC (Link Is Not Closed) is a completely new open-source switching platform available through flow-forwarding.org, a community promoting free open source Apache 2 license implementation based on OpenFlow specifications. LINC-Switch is a software switch and ONF's OpenFlow version 1.2/1.3 compliant capable Switch with support for OF-Config 1.1 Standard [15]. OpenFlow protocol separates the control plane from data plane and allows much flexibility to the applications by implementing control logic using software programming. The OF-Config protocol allows separating the management functionalities from the OpenFlow switches to have efficient control over the networking resources like ports to better utilize them [16].

LINC architecture is designed to use generally-available commodity x86 hardware (COT) and runs on a various platforms like Linux, Solaris, Windows, MacOS, and FreeBSD with Erlang runtime. The multiple CPU cores and memory offered by x86 platform allows LINC to scale gracefully to increase and decrease compute resources. This is essential when many logical switches are instantiated on a single OpenFlow capable switch. These logical switches can have resource allocations based on the need.

LINC was implemented in the functional programming language Erlang, developed by Ericsson.

5.1 Features of Erlang

The following features offered by Erlang makes it suitable for developing LINC:

- Erlang is a functional programming language designed to develop concurrent applications. Concurrency in Erlang belongs to the language and not to the Operating System [5].

- Erlang OTP (Open Telecom Platform) is a large collection of libraries for Erlang that provides solutions for Networking and Telecommunication problems. E.g. Supervision trees.

- Bit manipulation capabilities of Erlang are suitable for protocol handling and low level communication.

- Erlang has built-in support for process creation and management to simplify concurrent programming. As Erlang processes are light weight, it needs less computational effort to create and destroy processes [17].

- It allows achieving massive concurrency, i.e thousands of processes can be created without degrading the performance. Erlang process use share nothing semantics [5] principle and they communicate among themselves only by passing messages between them. Erlang processes work on a copy of the data it needs. As there is no data sharing between concurrent processes, efficiency will be more.

- Any Erlang application is made distributed easily by running different parallel processes on different machines.

- Erlang makes best use of multi-core architecture available on the machine from which it is running which makes it best suitable for writing concurrent applications.

5.2 LINC Architecture

The main software blocks of LINC implementation are: OpenFlow Capable Switch, OpenFlow protocol module and OF_Config module. These are developed as separate applications and are designed using Erlang OTP principles.

The of_protocol librarary implements the OpenFlow protocol to define internel OpenFlow protocol strcutures, data types and enumerations. It affords encode and decode functions for OpenFlow protocol messages and validates their correctness. The figure 3 describes the software componets of LINC-Switch.
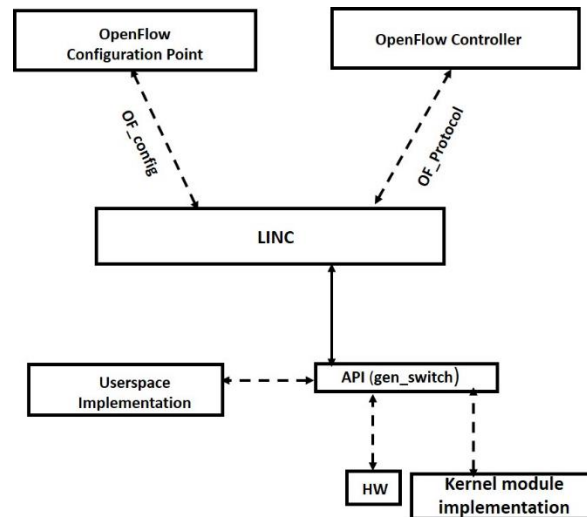

Figure 3 Software Components of LINC-Switch

The linc library implements the OpenFlow capable switch functionality. It accepts OF-Config commands and executes them in OpenFlow operational perspective. It handles one or more OpenFlow Logical Switches that consists of the channel component, replaceable back-ends and common logical switch logic. The channel componet act as the communication layer between the OpenFlow Logical Switch and the OpenFlow Controllers and by meeans of TCP/TLS connections between them. It passes parsed structures received from the OpenFlow Controller to the backend and forwards encoded messages from OpenFlow Switch to the Controllers.

The actual logic for switching the packets is implemented in replaceable back-ends. They manage flow tables, group table, ports, etc. and reply to OpenFlow protocol messages received from the Controller. LINC's logical Switch can use any of the available backends with comman API (gen_switch).

Common switch logic handles switch configuration, manages the channel component and OpenFlow reosurces like ports and dispatches messages received from the Controller.

The of_config library application implements the OF-Config protocol which handles parsing, validation and interprets the OF-Config messages received from an OpenFlow configuration point and sends as commands to the OpenFlow capable switch application 'linc' to configure OpenFlow Capable Switch.

LINC has a supervision tree for fault-tolerance purposes in accordance with the OTP principles. A supervisor is responsible for starting, stopping and monitoring its child processes. The supervisor keep its child process alive by restarting them when necessary.

5.3 Distributed LINC-Switch

According to [18], "Distributed Erlang applications can be implemented on loosely connected systems of computers with TCP/IP connectivity between them", LINC-Switch is implemented on three inter connected multiple nodes to make it behave as a distributed application to achieve fault-tolerance by utilizing the distribution properties of Erlang. Figure 4 depicts the redundant switch architecture. Fault-tolerance in Erlang applications are achieved as follows: When LINC-Switch on one node fails, then it will be running from other node (failover) and when the LINC-Switch on main node comes back again, the LINC-Switch from the back-up node will exit and stops running (takeover). Failover is the process of restarting an application on the node other than on the failed node [20].
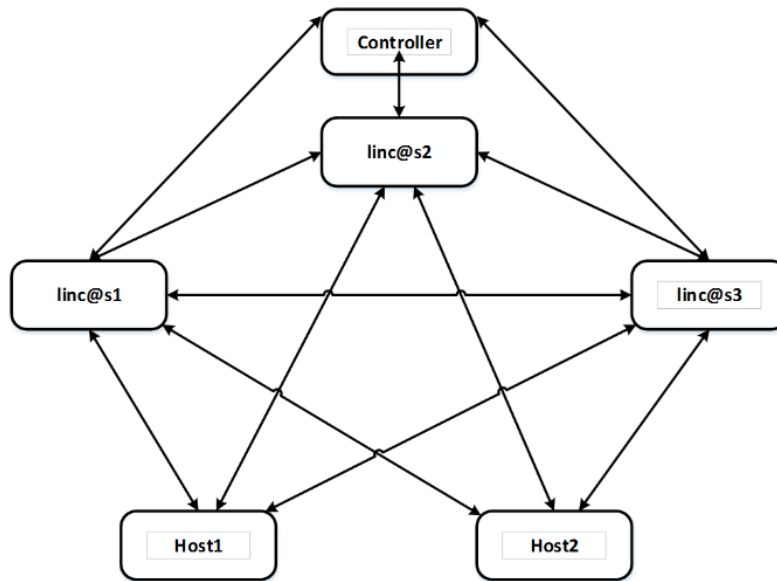
Figure 4 LINC-Switches as Distributed Switch

Takeover is the process in which the node having higher precedence (main node) takes over the control when the application is running from a lower precedence secondary node [19].

We have taken the replication factor as 3 to reduce the probability of failures as given in [5]. Erlang programming language has inherent distributed system support to enable such functions and tools to the programmer with mature messaging architecture between the instantiated nodes.

Chapter 6: How Distribution is achieved in Erlang

The Erlang language was designed specifically to develop fault-tolerant systems by keeping the following points as main criteria [21]:

- Isolation between processes

- Pure message passing between processes

- Ability to detect errors in remote processes

- Means to detect the cause for the errors.

In Erlang, one layer of the system performs the application logic while the other layer performs the error trapping functionality. It monitors the applications and restores the application to safe state if it fails. The application structure is formalized in Erlang OTP system by means of supervisor trees. The supervisor trees define very accurately what should be done in case a process fails. OTP application organize the task into tree structured group of processes and the processes in higher level (supervisor) monitor the processes in lower level (worker) of the tree structure and correct if any error occurs in the worker process[17]. The worker processes perform the computation.

A distributed Erlang system consists of many Erlang run time systems known as nodes, communicating with each other. Each Erlang node is identified by an atom in the format 'name@ hostname', where name is supplied by the user at the time of starting Erlang shell and hostname is full hostname if long name is used or part of the hostname if short name is used.

Erlang Port Mapper daemon (EPMD) is started when an Erlang system is started, which is responsible for mapping the hostnames to IP addresses. The security of the networked Erlang nodes is ensured by setting the atom known as magic cookie. The nodes are allowed to communicate with each other only if their magic cookies match.

The Open Telecom platform (OTP) framework provides takeover and fail over mechanisms to make an Erlang distributed application as a fault tolerant system. The failover is a mechanism of starting the application on a different node than on a node where it is failed. The takeover is a mechanism of running back the application on a dead main node by gracefully terminating the application on backup/secondary nodes. The modules *global*, *global group*, *net_adm*, *net_kernel*, *kernel* are involved in making an Erlang application as distributed.

6.1 Global Module

Global Module serves as a global name registration facility. It servers the following functionalities through the server called *'global_name_server'*, which resides on each node and it is started automatically when an Erlang node is started. The global denotes the set of nodes connected together. The global module takes care of the following functionalities:

- Registration of global names
- Global locks,
- Maintenance of fully connected network

The ability of registering the names globally is the central concept in distributed programming. A registered name is an alias for a process identifier (PID). The global name server

24

monitors the registered process ids. If a process terminated, the associated PID with the process will be unregistered. The registered names are stored in replicated global name tables in all the nodes. Any change in a name table will results in same change in all the tables.   The global name server continuously monitors the changes in node configuration. When a node, on which globally registered process goes down, the name will be globally unregistered. The global name server subscribes to node up and node down messages sent from the kernel.

6.2 Global group

This module is responsible for grouping nodes to Global Name registration Group. This makes it possible to group the nodes into partitions, each partition having its own global name space refer to global. These partitions are called global groups.

6.3 Net _kernel

The *net_kernel* is a system process, registered as *net_kernel*, running of which is very essential for distributed Erlang to work. The purpose of this module is to implement the distribution built-in functions *spawn/4* and *spawn_link/4*, and to monitor the network. The connection to a node is automatically established when a node is referenced from another node. Through the built-in function *monitor_node/2*, the calling process subscribes or unsubscribes to node status change messages. A node up message is delivered to all subscribing processes when a new node is connected and node down message is delivered when a node is disconnected.

6.4 Net_adm

This module defines various Net Administration Routines. One of the built in function *world/1* calls names (Host) for all hosts which are specified in the Erlang host file, '*.hosts.erlang*', collects the replies and then evaluates ping (Node) on all of those nodes and returns the list of all nodes that were successfully pinged. This function is used, when a node is started, and the names of other nodes in the network are not initially known.

6.5 Kernel

Distributed applications are controlled by both application controller and a distributed application controller process called *dist_ac*. Both are part of *kernel* application [22].

The kernel application is the first application started in any Erlang system. It is mandatory to have minimal system based on Erlang OTP should consists of kernel and STDLIB modules. The configuration parameter '*distributed*' specifies which application is distributed and on which nodes it may execute. The parameter

- *distributed = [{Application, [Timeout,] NodeDesc}]* specifies where (on which node) the application *Application*, may execute. Where:
  - *NodeDesc = [node | {node. . . Node}]* is a list of node names in priority order.
  - *Timeout = integer ()* specifies how many milliseconds to wait before restarting the application on another node.

The nodes on which, a distributed applications runs must contact each other and negotiate where to start the application. The following kernel configuration parameters specifies which

nodes are must and which nodes are optional for a particular node to start and how long it will wait for the mandatory and optional nodes to come up :

- *Sync_nodes_mandatory = [Node]*, specifies which other nodes must be started within the time *sync_nodes_timeout*.

- *Sync_node_optional = [node],* specifies which other nodes can be started within the time *sync_node_timeout.*

- *sync_node_timeout = integer () | infinity,* specifies how many milliseconds to wait for other nodes to start.

When started, the main node will wait for all the nodes specified by *Sync_nodes_mandatory* and *Sync_node_optional* to come up and when all the nodes have come up and the time specified by *sync_nodes_timeout* has elapsed, all applications will be started. If not all the *syn_mandatory_ nodes* have come up, the main node will terminate.

Net_ticktime = TickTime specifies the *net_kernel* tick time in seconds. Once every *TickTime*/4 seconds, all connected nodes are ticked and if nothing has been received from another node within the last four tick times then that node is considered to be down. This ensures that nodes which are not responding for reasons such as hardware errors are considered to be down. Thus a terminated node is detected immediately.

6.5 Protocol Behind the Distribution (Distributed Protocol)

The communication between EPMD and Erlang nodes happens based on distribution protocol [23]. The protocol has four parts:

- Low level socket connection

- Handshake, interchange node name and authenticate

- Authentication (done by *net_kernel*)

- Connected.

The EPMD starts automatically when an Erlang node starts up. It listens on the port 4369. A node fetches the port number of another node to initiate connection via EPMD. When a distributed node is started, it registers itself with EPMD using the message *ALIVE2_REQ*. The response from the EPMD is *ALIVE2_RESP*. The connection to the EPMD is kept open till the node is in distributed mode. The connection will be closed when the node unregisters from EPMD. When a node wants to connects with another node, it requests the distribution port number on which the node is listening through *PORT_PLEASE2_REQ* message to the EPMD of the destination port. The EPMD of the destination node responds with *PORT2_RESP*.

The TCP/IP distribution protocol uses connection based handshake to establish a connection. During the handshake, the cookies are sent and verified to ensure that the connection is between allowed nodes.

6.6 Starting and Stopping Distributed Applications

When all the mandatory nodes have been started, the distributed application can be started by the application master on all the nodes by calling application: start (Application). This could be automatically done by a boot script. In our LINC-Switch application, the application master has called the application callback function, *linc: start (normal, _StartArgs)* on the node *linc@s1*. The application could be stopped by calling application: stop (Application) at all involved nodes.

Failover:

If the node where the application is running goes down, the application is restarted after the specified time at the next node in the *NodeList* by the application master calling Module: start (normal, StartArgs) at that node. This is called failover. When linc@s1 goes down, the application master will call the callback function, *linc: start (normal, _StartArgs)* at S2, if linc@s2 could not be started then the application will be started from S3.

If the LINC-Switch application at linc@s2 goes down, then the application master at S3 will start it at linc@s3 by calling *linc: start (normal, _StartArgs)* at S3.

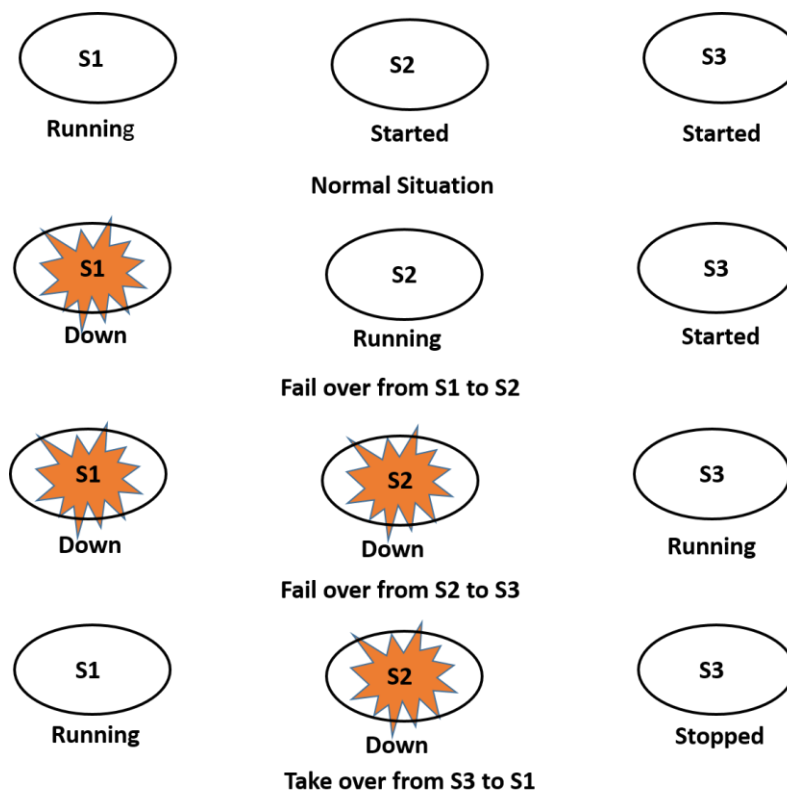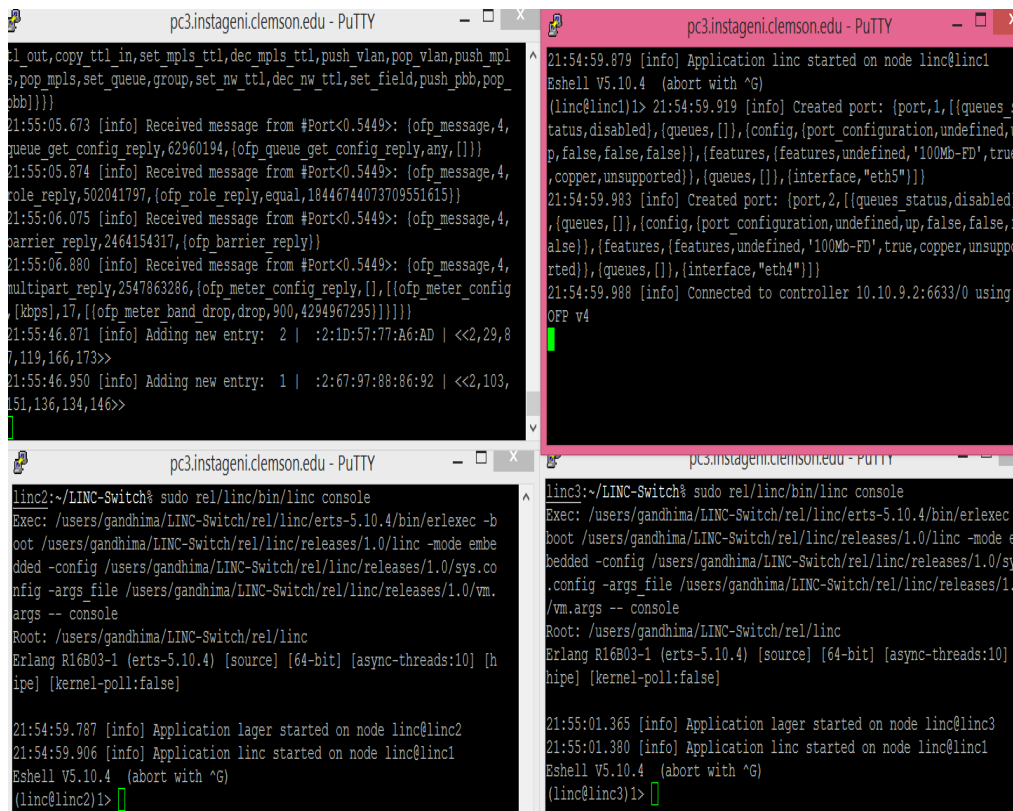*Figure 5 Failover and Takeover of application in Distributed Erlang*

Takeover:

If a node which is having higher priority according to *distributed* parameter is started, then the distributed application currently running from a lower priority node will be stopped and starts running from the higher priority new node. This is known as takeover. The application is taken over to new higher priority node by the application master at that node calling Module: start ({takeover, Node], StartArgs) where Node is the old node.

When LINC-Switch application is running from linc@s3, if we restart the application on linc@s1, the application master will call *linc: start (takeover, _OtherNode}, [])* from linc@s1 and causes LINC application to stop at linc@s3 and made it run from linc@s1. The failover and takeover events are portrayed in the following figure 5:

Chapter 7: Experimental Setup

The LINC switch is installed on three PCs, communicating nodes h1 and h2 are connected with all three switches. The kernel parameter is configured in such a way that S1 will be primary switch and S2, S3 are mandatory nodes having priority in descending order. The Switches are started at once and the primary switch will wait for other mandatory switches to start up, once all the mandatory nodes started, it will connect with the OpenFlow controller and waiting for packets to be forwarded through it as shown in the figure 6.

Figure 6 Three switches are started and only S1 will be in running state, top left terminal belongs to controller

The host1 starts sending packets to host2 through switch S1. The controller installs flow entries in the switch and the switch forwards the packets based on the flow table entries. If the switch S1 fails, then *dist_ac* of S2 waits for *Time_out* milliseconds for the node S1 to restart again, if S1 didn't start within that interval then the switch S2 will be started automatically by the *dist_ac* at *S2*. Again if S2 fails, then S3 will start after *Time_out* milliseconds. If S1 restarts again, then S3 will exit and the switch functionality will be taken over by S1 as it is having highest priority. The Erlang distribution ensures that at any instant of time only one switch will function and forward the packets between connected nodes.

7.1 Topology on GENI

The Global environment for Network Innovations (GENI) is a suit of research infrastructure available for networking and distributed systems research, funded by National Science foundation [24]. It is a geographically distributed research network/test bed which contains diverse networking resources and supports simultaneous experiments and allows end users to use and exploit the experimental protocols. The experimental topology is implemented and tested on the ProtoGeni test bed deployed on the GENI which is based on Emulab facility.

The LINC switch application is implemented on three PCs   pc529.emulab.net, pc541.emulab.net and pc557.emulab.net as a distributed application. The Open Flow controller application is running on pc534.emulab.net. The communicating hosts are pc515.emulab.net and pc560.emulab.net. The topology is illustrated in figure 7.

The kernel parameter of the *sys.config* file is configured to make an application as a distributed is generalized as follows:

[{kernel,
[{distributed, [{AppName, TimeOutBeforeRestart, NodeList}]},
{sync_nodes_mandatory, NecessaryNodes},
{sync_nodes_optional, OptionalNodes},
{sync_nodes_timeout, MaxTime}
]}].

During our experiments, we have set the parameters of fault tolerance as follows on S1:

[{kernel,
[{distributed, [{linc, 5000, ['linc@s1', 'linc@s2', 'linc@s3']}]},
{sync_nodes_mandatory, ['linc@s2', 'linc@s3']},
{sync_nodes_timeout, 5000}]}].

In the same way on S2:

[{kernel,
[{distributed, [{linc, 5000, ['linc@s1', 'linc@s2', 'linc@s3']}]},
{sync_nodes_mandatory, ['linc@s1', 'linc@s3']},
{sync_nodes_timeout, 5000}]}].

And on S3 as:

[{kernel,
[{distributed, [{linc, 5000, ['linc@s1', 'linc@s2', 'linc@s3']}]},
{sync_nodes_mandatory, ['linc@s1', 'linc@s2']},
{sync_nodes_timeout, 5000}]}].

Corresponding to

Timeout BeforeRestart	*NodeList*	*Necessary Nodes*	*MaxTime*
5000 msec	S1 (primary), S2, S3	S2 and S3	5000 msec

Table 1Kernel Parameters
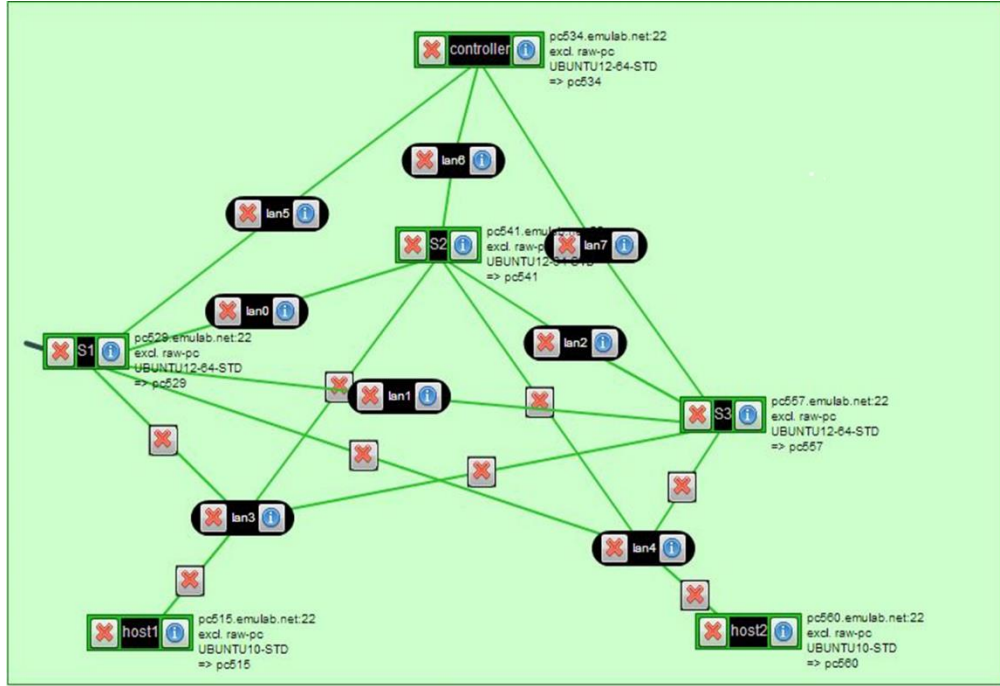
Figure 7 Distributed Fault-Tolerant system of LINC-Switches on GENI with Non- OpenFlow connect to connect host1 and host2 to the LINC-Switches

The LINC switch is started on three nodes linc@s1 (pc529.emulab.net), linc@s2 (pc541.emulab.net), linc@s3 (pc557.emulab.net) as a distributed and embedded application at the same time, but it starts running on linc@s1 and connected to the controller running on pc534.emukab.net. The host1 and host2 are now forwarding their data through linc@s1. If the LINC application is stopped by quitting using CTRL + G and q, then LINC application is start run (fail over) from linc@s2, which is a secondary node having next higher priority and connected with the controller. Now the host1 and host2 are communicating through it. If the LINC switch at linc@s2 fails, it will run from linc@s3 by the *dist_ac* at S3 (fail over again). If the LINC switch on the main node linc@s1 comes up alive and restarted, then the LINC application on the lower

priority secondary node linc@s3 will exit and stop running and start running from linc@s1 (takeover).

7.2 Measurements

The distribution feature is tested with host1 sending ping (ICMP request) packets to host2 through the LINC-Switch implemented as a distributed application on the three nodes but running from only one node at any time.

Figure 8 illustrates the tick message exchange between the Erlang nodes during normal operation. S1 is the primary switch handling the data transmissions. S2 and S3 are started but not in running state. S1 and S2 are the back-up nodes waiting for the other nodes to die. In distributed Erlang the *dist_ac* process classifies an application into running state from start state. The *global* module makes sure that the application will run on only one node at anytime.
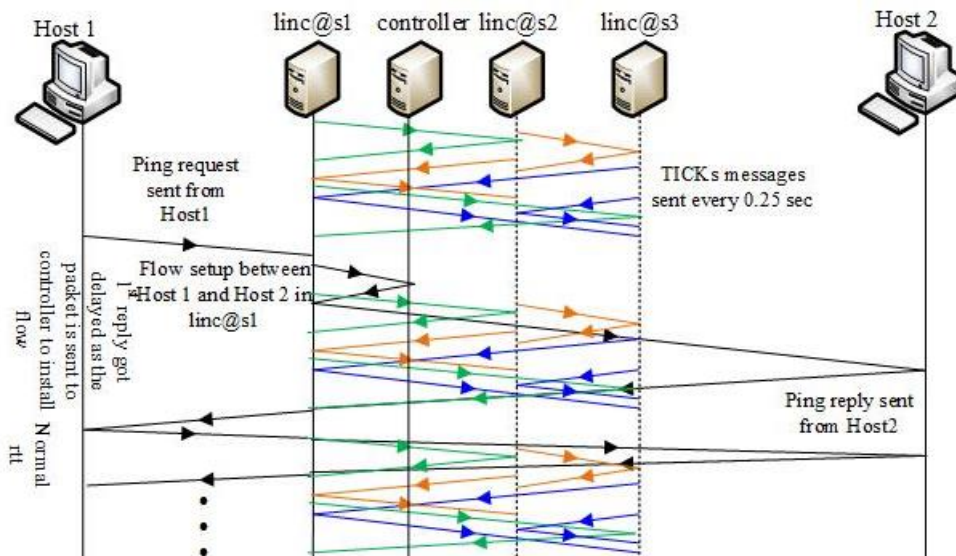
Figure 8 LINC-Switches on three nodes started and sending Tick messages to each other; The LINC-Switch at S1 is running

When S1 fails, the other nodes linc@s2 and linc@s3 will not receive tick messages from linc@s1. If linc@s2 and linc@s3 did not receive four consecutive tick messages then they considers that linc@s1 is dead and application master at S2 will start the LINC on S2 because it is having next priority to S1. This is illustrated in figure 9.

As soon as secondary node S2 starts running, it connects with the controller and exchange *Hello* messages with it. Once the controller sets the flow in S2, ping exchange starts between host1 and host2. We measured the number of ping messages lost during the failover time frame as shown in figure 12.

Figure 9 LINC-Switche S1 stopped working and no tick messages seen from it, so LINC-Switch at S2 will be started by dist_ac (failover)

Figure 10 LINC-Switche S2 stopped working and no tick messages seen from it, so LINC-Switch at S3 will be started by dist_ac (failover)
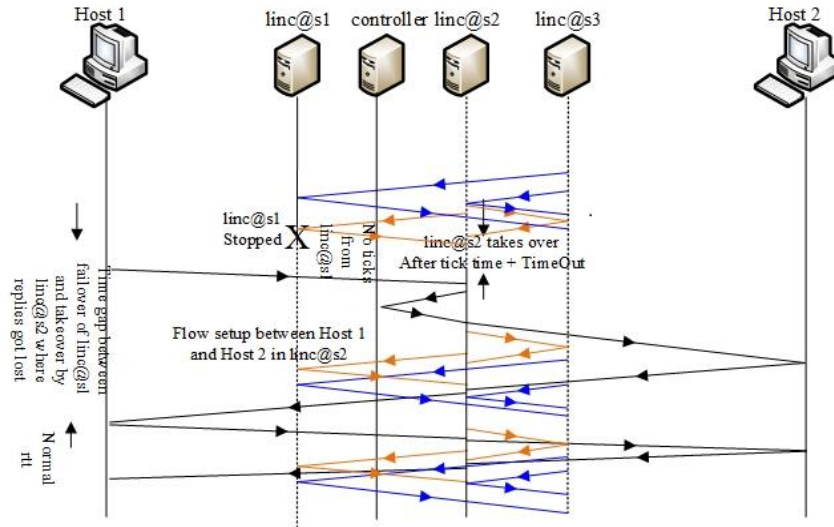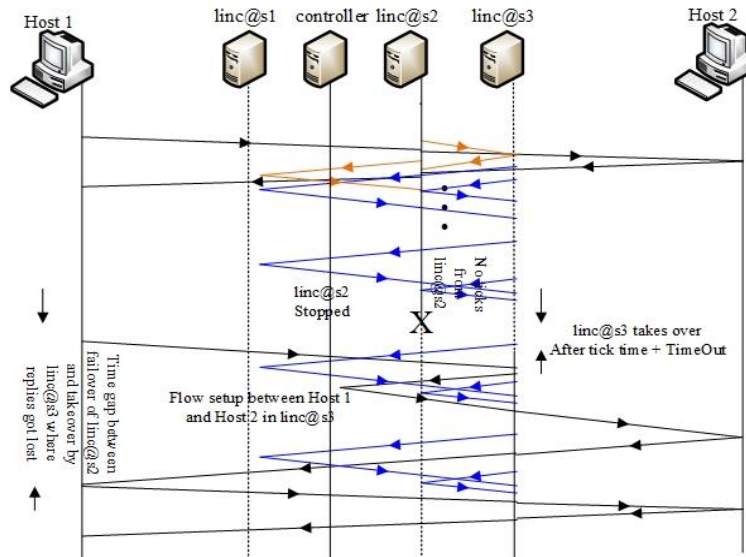
Again if S2 fails, S3 will come to running state, connect with controller and resume forwarding the packets between host1 and host2 as in figure 10.
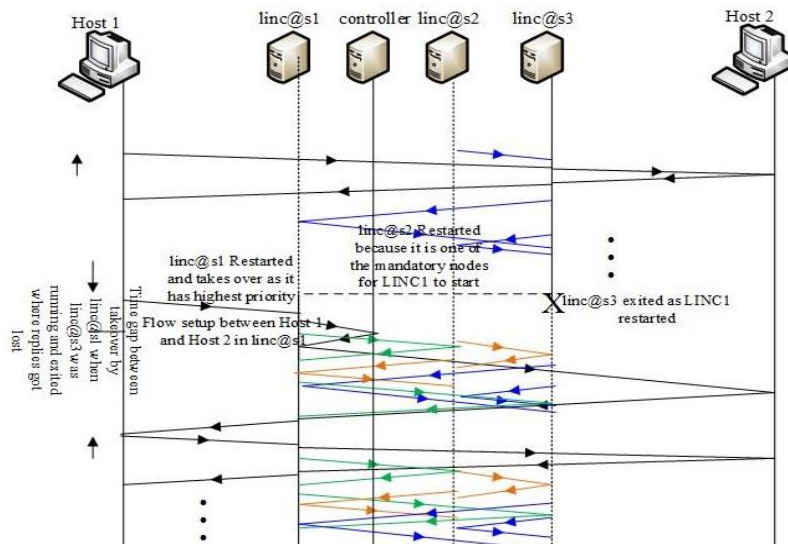

Figure 11 While LINC-Switche on S3 is running, if LINC-Switch at S1 restarts then LINC-Switch at S3 will exit and S1 takesover.

When S3 is running, if S1 is restarted S3 will stop working and exit as it is having lower priority than S1. Figure 11 shows the S1 taking over the forwarding plane from S3 when back available.

Erlang system ensures such a primary node to take over from the secondary nodes when available again from dead state.

We measured the time gap between a failure of a software switch and the time that the secondary switch taking over the forwarding plane. The main goal is to determine how many packets of supported flows may be dropped during a failover event. This time gap has the following delay components.

i.   time to detect failure of S1;
ii. time for S2 to go from "start" to "running";
iii. time for the ping requests to get directed to S2;
iv. time for the S2 to connect with the controller;
v.   time delay until flows become active in the S2.

7.2.1 TOPOLOGY ON GENI: connectivity with non-OpenFlow switches

The delay elements are bundled into one measurement in this study in order to present the potential impact of data loss during a failover and take-over situation. We have conducted a ping-based measurement between host1 and host2 similar to the study in [11]. When failover or takeover happens, we found that on an average 42 pings were missing as the hardware (non- OpenFlow) switches on GENI testbed used to connect host1 and host2 to the redundant switch architecture take time to accomplish MAC learning as the path which connects host1 to host2 changes. For example

38

when S1 is running, the path to be traversed by the packets is host1 - S1 - host2 and when failover happens to S2 now the path is host1 - S2 - host2. The hosts are moved from one physical port to another: the new port-MAC mapping will be updated only after the expiration of MAC-aging time in the hardware switches.

MAC Age Time refers to the number of seconds a MAC address the switch has learned remains in the switch's address table before being aged out (deleted). The default time interval preferred by the network administrators is 300 sec as per the HP manual [25].

Trial No.	S1-S2			S2-S3			S3-S1		
	No. of Pings lost	RTT	RTT * (No.of Pings lost)	No. of Pings lost	RTT	RTT * (No.of Pings lost)	No. of Pings lost	RTT	RTT * (No.of Pings lost)
1	32	1.424	45.568	41	1.551	63.55	49	1.5151	74.235
2	30	1.320	39.6	38	1.304	50.92	47	1.403	65.941
3	33	1.370	45.21	39	1.348	52.572	27	1.432	38.664
4	50	1.381	69.05	50	1.249	62.45	54	1.339	72.306
5	46	1.329	61.134	45	1.382	62.19	29	1.341	38.889
6	44	1.334	58.696	35	1.205	42.175	38	1.325	50.35
7	43	1.241	53.363	42	1.185	49.77	49	1.291	63.259
8	53	1.256	66.568	28	1.261	35.308	55	1.438	79.09
9	44	1.356	59.664	46	1.316	60.536	35	1.226	42.91
10	49	1.247	61.103	50	1.362	68.1	29	1.315	38.135
Avg	42.4	1.3258	55.9956	41.4	1.3163	57.000	41.2	1.3625	56.3779

Table 2 Results of sending of 200 Pings form host1 to host2 and making failover happens between S1 to S2 , S2 to S3 and takeover from S3 to S1 when Non-OpenFlow connect is used to connect hosts to fault-tolerant system. TimeOutBeforeRestart= 5000msec

The Mac-Aging should be set appropriately. Too long aging interval may cause the MAC-address table to retain outdated entries, exhaust the MAC address table resources, the following table shows the experiments results obtained on GENI test bed with the HP Procurve 5406 Switch used as a connecting device to connect host1 and host2 to the fault-tolerant architecture.
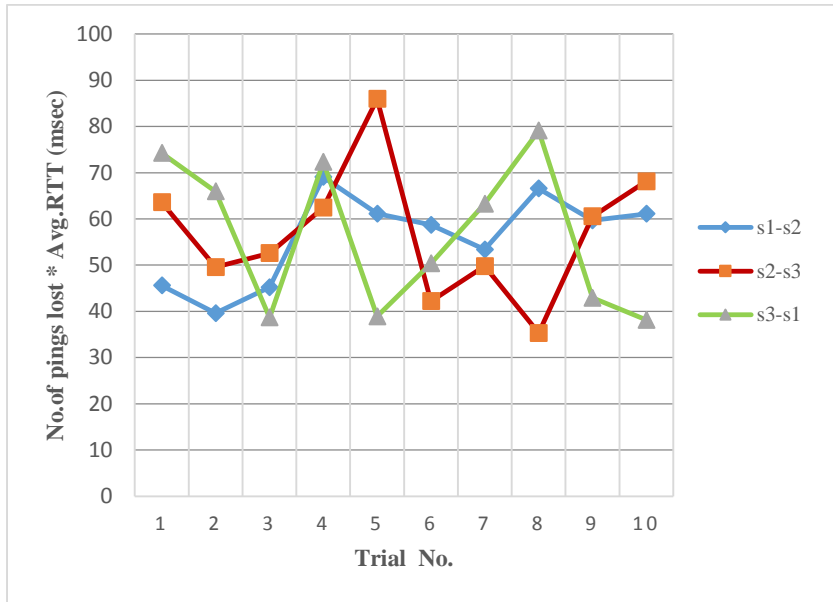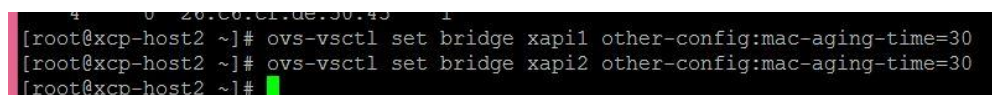

Figure 12 Measurements of failover and takeover times in terms of number of lost pings normalized with the average recorded RTT when HW Switches are used to connect hosts to fault-tolerant system with TimeOutBeforeRestart=5000 msec

7.2.2 TOPOLOGY ON GENI: connectivity with OVS

As MAC-Aging time has serious impact on the number pings getting lost in the previous experiment, we want to know how much we could reduce the loss of ping replies when failover or take over happens by varying the MAC-Aging time of the switch. AS we have no access to hardware switch's command line interface (CLI) to change the MAC-aging time on GENI, we repeated the same topology on GENI with the OpenvSwitch running from prebuilt image VMs to connect host1

and host2 to fault-tolerant switching architecture system. The ping test is conducted by setting various MAC-aging times on the OVS bridges which connect hosts to the redundant LINC-Switches. We observed that the number of pings lost was minimum (15) and remains constant for MAC-aging-time from 1 to 15 seconds after that it shows increase in number of pings lost with increase in MAC-aging time as shown in figure 15. It shows that it takes at least 15 seconds to update the mac table entries. We conducted the tests only till MAC-aging-time equal to 30 seconds as we want to know how much we could reduce the loss by setting lower mac-aging-time. The default value for the mac-aging-time is 300 seconds as described in the OpenvSwitch manual [26].

We have implemented the same topology by installing LINC-Switch on three VMs on a Xen-cluster from our Laboratory and connected host1 and host2 to the fault-tolerant LINC-Switches via the bridges xapi1 and xapi2 of the default switch OVS on the Xen hypervisor. We repeated the ping measurements by setting different MAC-aging-time on the bridges xapi1 and xapi2 as in figure13 and observed that we lost less number of pings with minimum values of MAC-aging-times.

Figure 13 Mac-aging time can bet to values between 15 to 3600 seconds using ovs-vsctl

The ovs-vswitchd daemon controls and manages the OpenvSwitch available on the local machine. The utility ovs-appctl is used to configure and run ovs daemons. The command 'ovs-

appctl fdb/show' with bridge name will lists the MAC address/VLAN pair learned by the bridge

along with port number to which the host belonging to the MAC address is attached and the age

of the entry in seconds as in figure 14.

Figure 14 MAC-table learned by the bridge xapi1of OVS1

Mac-age in seconds	No. of Pings lost (GENI)	No. of Pings lost (Xen)
1	15.46666667	15.625
5	15.46666667	15.46666667
10	15.46666667	15.4
15	15.4	15.73333333
20	20.53333333	20.66666667
25	25.86666667	25.6
30	28.33333333	30.46666667

Table 3 Results of sending Pings form host1 to host2 and making failover happens between S1 to S2, S2 to S3 and takeover from S3 to S1 when OVS is used to connect hosts to fault-tolerant system with learning switch mode on GENI and on Xen for various Mac-Aging times.
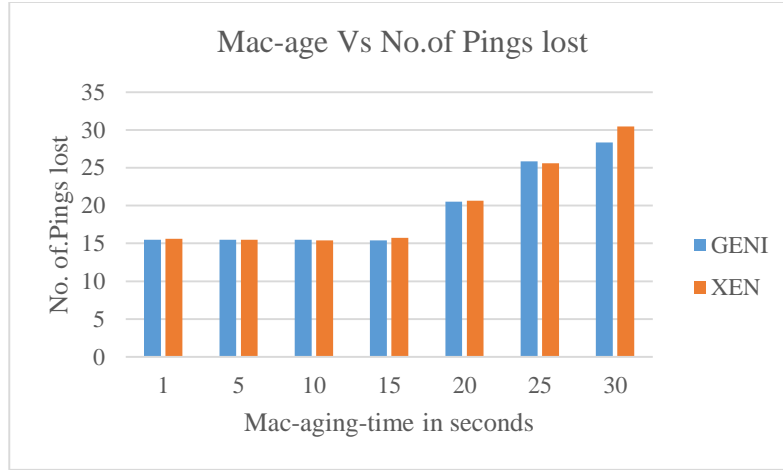
Figure 15 No. of Ping replies are constant at 15 when mac-aging time is from 1 to 15 seconds on OVS

7.2.3 TOPOLOGY ON GENI: connectivity with LINC

We also used LINC-Switches to connect host1 and host2 to the fault-tolerant architecture instead of OVS and used an OpenFlow controller which modifies the flow entries when the hosts are moved from one port another as depicted in figure 16. The topology of which is shown in figure 17. When failover happens the controller modifies the flow entries in the LINC-Switches which connects the hosts to fault-tolerant architecture and causes the ping packets to reach the new switch's input port immediately. The following figure shows the controller output when the flow entries are being modified when host2 is moved from one port to another when fail over/take over happens. The RTT values are measured by sending a count of 200 ping requests from host1 to host2 and the numbers of pings lost during the failover/takeover are also recorded.

```
18:26:44.902 [info] Adding new entry:  3 |  :2:67:97:88:86
:92 | <<2,103,151,136,134,146>>
18:26:58.868 [info] Modifying existing entry: <<2,103,151,
136,134,146>> | 3  ->  <<2,103,151,136,134,146>> | 4
18:27:20.901 [info] Modifying existing entry: <<2,103,151,
136,134,146>> | 4  ->  <<2,103,151,136,134,146>> | 2
18:27:41.965 [info] Modifying existing entry: <<2,103,151,
136,134,146>> | 2  ->  <<2,103,151,136,134,146>> | 3
18:28:36.849 [info] Modifying existing entry: <<2,103,151,
136,134,146>> | 3  ->  <<2,103,151,136,134,146>> | 4
```

Figure 16 Flow entries are modified by the controller when it sees the MAC address coming from different input port than before by using the message modify_strict

It is observed that with HW switches to connect hosts to fault-tolerant system, the average number of ping replies lost were $41.6 \approx 42$ and with LINC-Switches to connect hosts to the fault-tolerant system, it was about $5.6 \approx 6$ when *TimeOutBefore Restart =5000* msec.
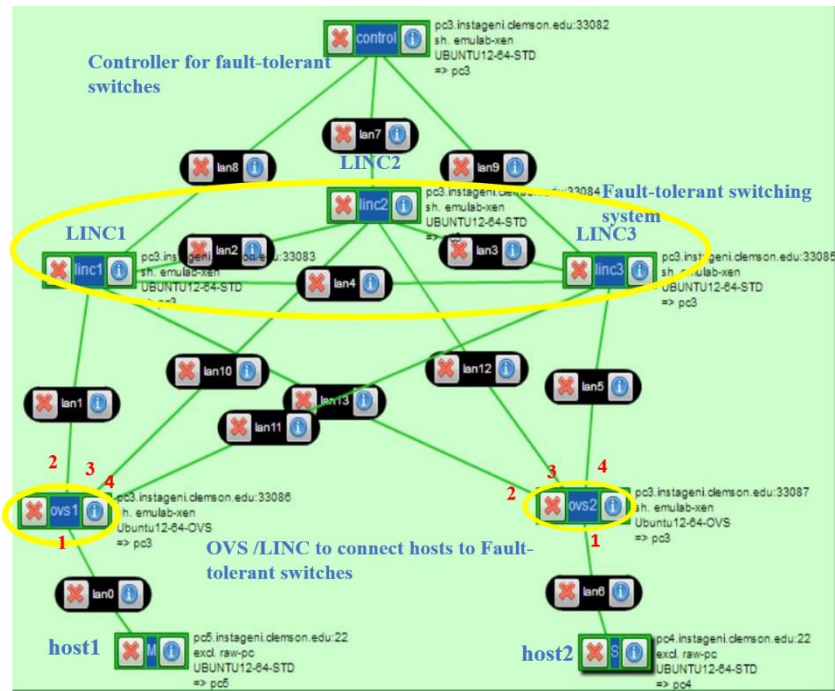

Figure 17Hardware switches used on GENI to connect communicating hosts are replaced by OpenFlow   switches OVS / LINC.

44

In Erlang OTP with distributed application, when we made the *TimeOutBeforeRestar*t value of kernel parameter equal to zero, there was no packet loss or very minimum loss and the average was 0.4 ≈ 0. The following tables 4 and 5 shows the results of the experiment with LINC-Switch to connect host1 and host2 to fault-tolerant switches with *TimeOutBeforeRestart*=5000 msec and 0 msec respectively. The figures 18 and 19 show the graph of normalized values of pings lost with Avg. RTT at different test runs.

Trial No.	S1-S2			S2-S3			S3-S1		
	No. of Pings lost	RTT	RTT * (No.of Pings lost)	No. of Pings lost	RTT	RTT * (No.of Pings lost)	No. of Pings lost	RTT	RTT * (No.of Pings lost)
1	5	6.199	30.995	6	6.232	37.392	7	6.038	42.266
2	6	6.634	39.804	5	6.264	31.32	7	6.424	44.968
3	5	6.231	31.155	5	6.098	30.49	6	6.462	38.772
4	5	6.592	32.96	5	6.139	30.695	6	6.008	36.048
5	5	6.153	30.765	5	6.128	30.64	6	6.463	38.778
6	5	6.162	30.81	5	6.222	31.11	7	6.494	45.458
7	5	6.213	31.065	6	4.651	27.906	7	6.527	45.689
8	5	6.031	30.155	6	5.64	33.84	7	6.016	42.112
9	5	6.445	32.225	5	5.921	29.605	6	6.323	37.938
10	5	6.207	31.035	5	6.017	30.085	6	6.402	38.412
Avg	5.1	6.286	32.0969	5.3	5.9312	31.3083	6.5	6.3157	41.0441

Table 4 Results of sending of 200 Pings form host1 to host2 and making failover happens between S1 to S2, S2 to S3 and takeover from S3 to S1 when LINC-Switch with OF Controller is used to connect hosts to fault-tolerant system. TimeOutBeforeRestart= 5000msec
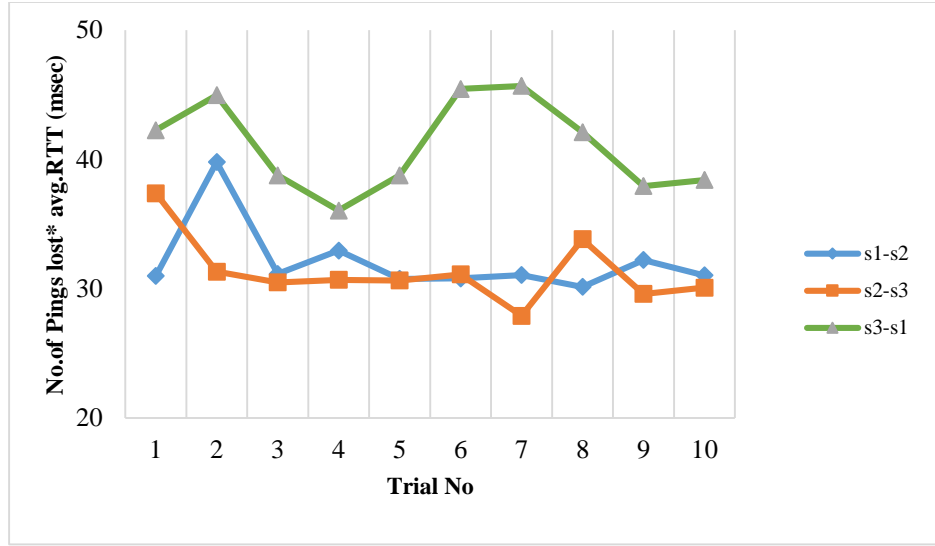
Figure 18 Measurements of failover and takeover times in terms of number of lost pings normalized with the average recorded RTT when LINC-Switches are used to connect hosts to fault-tolerant system with TimeOutBeforeRestart=5000 msec

Trial No.	S1-S2			S2-S3			S3-S1		
	No. of Pings lost	RTT	RTT * (No.of Pings lost)	No. of Pings lost	RTT	RTT * (No.of Pings lost)	No. of Pings lost	RTT	RTT * (No.of Pings lost)
1	0	5.963	0	0	6.053	0	0	6.346	0
2	0	6.016	0	0	6.314	0	1	6.383	6.383
3	0	6.317	0	0	6.081	0	1	6.049	6.049
4	1	6.443	6.443	0	6.229	0	1	5.95	5.95
5	1	6.425	6.425	0	6.166	0	1	6.223	6.223
6	1	6.606	6.606	0	6.158	0	1	6.426	6.426
7	0	6.309	0	0	6.447	0	1	6.213	6.213
8	0	6.349	0	0	6.092	0	1	6.705	6.705
9	0	6.079	0	0	6.315	0	1	6.443	6.443
10	0	6.221	0	1	6.519	6.519	1	6.451	6.451
Avg	0.3	6.2728	1.9474	0.1	6.2374	0.6519	0.9	6.3189	5.6843

Table 5 Results of sending of 200 Pings form host1 to host2 and making failover happens between S1 to S2, S2 to S3 and takeover from S3 to S1 when LINC-Switch with OF Controller is used to connect hosts to fault-tolerant system. TimeOutBeforeRestart= 0msec
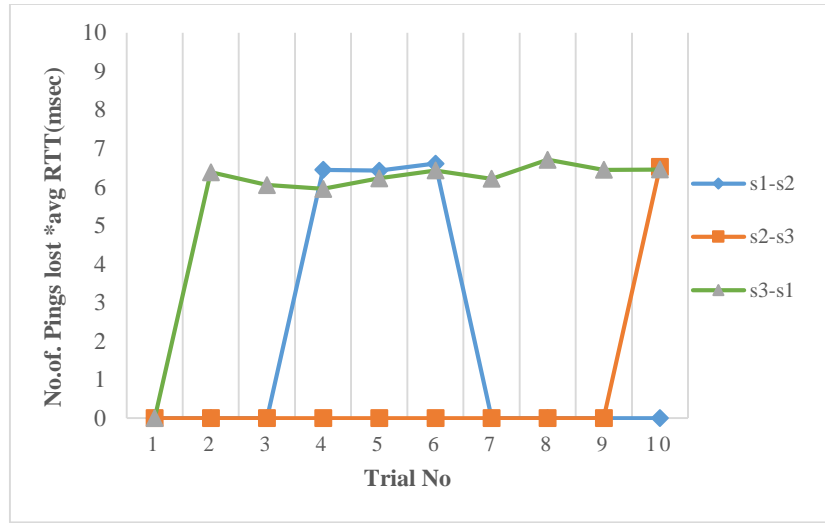
Figure 19 Measurements of failover and takeover times in terms of number of lost pings normalized with the average recorded RTT when LINC-Switches are used to connect hosts to fault-tolerant system with TimeOutBeforeRestart= 0 msec

Chapter 8 Conclusion

Fault tolerance is achieved by creating a redundancy based distributed system of LINC switches and using the built-in features of distributed Erlang. Furthermore, the built-in features of Erlang has helped to achieve fail-over and take-over functions to ensure a fault-tolerant system implementation. We have presented a sample experiment and measurements of time duration for a failover and take-over of LINC switches with different experimental set ups.

Although a fault-tolerance scheme can be realized among a redundant set of other software switches as well as hardware counterparts, Erlang shows some ease of programmability and fast deployment opportunity. Our goal is to continue the investigation of distributed system of software switches and the performance benefits realized through support of fault tolerance for example, the fault-tolerant switching architecture could be tested in Hadoop MapReduce frame work to connect nodes in the cluster to add more resiliency to the applications using Hadoop framework .

References:

[1] Narayan, Sandhya, Stuart Bailey, and Anand Daga. "Hadoop Acceleration in an OpenFlow-based cluster." In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pp. 535-538. IEEE, 2012.

[2] GitHub, Inc. "LINC-Switch" FlowForwarding.org. https://github.com/FlowForwarding/LINC-Switch (accessed June, 26, 2013).

[3] Armstrong, Joe. "The development of Erlang." *ACM SIGPLAN Notices* 32, no. 8 (1997): 196-203.

[4] Gray, Jim. "Why do computers stop and what can be done about it?." In *Symposium on reliability in distributed software and database systems*, pp. 3-12. 1986.

[5] Armstrong, Joe. "Erlang." *Communications of the ACM* 53.9 (2010): 68-75.

[6] Chérèque, Marc, David Powell, Philippe Reynier, J-L. Richier, and Jacques Voiron. "Active replication in Delta-4." In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, pp. 28-37. IEEE, 1992.

[7] LeMahieu, Paul, Vasken Bohossian, and Jehoshua Bruck. "Fault-tolerant switched local area networks." In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998*, pp. 747-751. IEEE, 1998.

[8] Fick, David, Andrew DeOrio, Jin Hu, Valeria Bertacco, David Blaauw, and Dennis Sylvester. "Vicis: a reliable network for unreliable silicon." In *Proceedings of the 46th Annual Design Automation Conference*, pp. 812-817. ACM, 2009.

[9] Ujcich, Benjamin, Kuang-Ching Wang, Brian Parker, and Daniel Schmiedt. "Thoughts on the Internet architecture from a modern enterprise network outage." In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pp. 494-497. IEEE, 2012.

[10] López, Macías, Laura M. Castro, and David Cabrero. "Failover and takeover contingency mechanisms for network partition and node failure." In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang workshop*, pp. 51-60. ACM, 2012.

[11] Fonseca, Paulo, Ricardo Bennesby, Edjard Mota, and Alexandre Passito. "A replication component for resilient OpenFlow-based networking." In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pp. 933-939. IEEE, 2012.

[12] Kurose, Jim, Keith Ross. *Computer Networking*, *A Top-Down approach*. 6th ed. New Jersey: Addison-Wesley, 2012.

[13] Open Networking Foundation. "Software Defined Networking (SDN) Definition." Opennetworking.org. https://www.opennetworking.org/sdn-resources/sdn-definition (accessed March 21, 2013).

[14] McKeown, Nick, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: enabling innovation in campus networks." *ACM SIGCOMM Computer Communication Review* 38, no. 2 (2008): 69-74.

[15] Rutka, Krzysztof, Konrad Kaplita, Sandhya Narayan, Stuart Bailey, Resources flowforwarding.org http://www.opennetsummit.org/pdf/2013/research_track/poster_papers/ons2013-final36.pdf (accessed August 23, 2013).

[16] Narisetty, RajaRevanth, Levent Dane, Anatoliy Malishevskiy, Deniz Gurkan, Stuart Bailey, Sandhya Narayan, and Shivaram Mysore. "OpenFlow Configuration Protocol: Implementation for the of Management Plane." In *Research and Educational Experiment Workshop (GREE), 2013 Second GENI*, pp. 66-67. IEEE, 2013.

[17] Armstrong, Joe. "Concurrency oriented programming in erlang." *Invited talk, FFG* (2003).

[18] Wikström, Claes. "Distributed programming in Erlang." In *PASCO'94-First International Symposium on Parallel Symbolic Computation*. 1994.

[19] Guerraoui, Rachid, and André Schiper. "Fault-tolerance by replication in distributed systems." In *Reliable Software Technologies—Ada-Europe'96*, pp. 38-57. Springer Berlin Heidelberg, 1996.

[20] Hebert Fred , " Distributed OTP Application." learnyousomeerlang.com http://learnyousomeerlang.com/distributed-otp-application (accessed August 23, 2013).

[21] Armstrong, Joe. "Making reliable distributed systems in the presence of software errors." Ph.D Thesis, Royal Institute of Technology, Stockholm 2003.

[22] Ericsson AB. "Kernel." Erlang.org http://www.erlang.org/doc/man/kernel_app.html (accessed August 23, 2013).

[23] Ericsson AB. "Distributed Erlang." Erlang.org http://www.erlang.org/doc/apps/erts/erl_dist_protocol.html (accessed August 23, 2013).

[24] Berman, Mark, Jeffrey S. Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. "GENI: A federated testbed for innovative network experiments." *Computer Networks* (2014).

[25] Hewlett-Packard Development Company, L.P. "HP Switch Software Basic Operation Guide K/KA.15.14" hp.com http://h20565.www2.hp.com/portal/site/hpsc/template.BINARYPORTLET/public/kb/docDisplay/resource.process/?spf_p.tpst=kbDocDisplay_ws_BI&spf_p.rid_kbDocDisplay=docDisplayResURL&javax.portlet.begCacheTok=com.vignette.cachetoken&spf_p.rst_kbDocDisplay=wsrp-resourceState%3DdocId%253Demr_na-c03990949-1%257CdocLocale%253Den_US&javax.portlet.endCacheTok=com.vignette.cachetoken (accessed February 3, 2014).

[26] OpenvSwitch. "OpenvSwitch Manual." openvswitch.org http://openvswitch.org/ovs-vswitchd.conf.db.5.pdf (accessed February 3, 2014).