IJESRT

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

DAC: Generic and Automatic Address Configuration for Data Center Networks

P. Gobi[*1], S.Arulselvi[2], Dr T.Kirankumar[3]

[*1,2,3] Bharath University, Chennai, India

rpgobi55@gmail.com

Abstract

DAC, a generic and automatic Data center Address Configuration system. With an automatically generated blueprint that defines the connections of servers and switches labeled by logical Ids, e.g., IP addresses, DAC first learns the physical topology labeled by device IDs, e.g., MAC addresses. Then, at the core of DAC is its device-to-logical ID mapping and malfunction detection. DAC makes an innovation in abstracting the device-to-logical ID mapping to the graph isomorphism problem and solves it with low time complexity by leveraging the attributes of data center network topologies. Its malfunction detection scheme detects errors such as device and link failures and mis-wirings, including the most difficult case where mis-wirings do not cause any node degree change. We have evaluated DAC via simulation, implementation, and experiments.

Keywords : Address Configuration, Data center networks(DCNs), graph isomorphism.

Introduction

In this paper, we have designed, evaluated, and implemented DAC, a generic and automatic Data center Address Configuration system. DAC, a generic and automatic Data center Address Configuration system. With an automatically generated blueprint that defines the connections of servers and switches labeled by logical Ids, e.g., IP addresses, DAC first learns the physical topology labeled by device IDs, e.g., MAC addresses. Then, at the core of DAC is its device-to-logical ID mapping and malfunction detection. DAC makes an innovation in abstracting the device-to-logical ID mapping to the graph isomorphism problem and solves it with low time complexity by leveraging the attributes of data center network topologies. Its malfunction detection scheme detects errors such as device and link failures and mis-wirings, including the most difficult case where mis-wirings do not cause any node degree change. We have evaluated DAC via simulation, implementation, and experiments. The requirements specification is a technical specification of requirements for the software products. It is the first step in the requirements analysis process it lists the requirements of a particular software system including functional, performance and security requirements. The requirements also provide usage scenarios from a user, an operational and an administrative perspective. The purpose of software requirements specification is to provide a detailed overview of the software project, its parameters and goals. This describes the project target audience and its user interface, hardware and software requirements. It defines how the client, team and audience see the project and its functionality. Data center networks encode locality and topology information into their server and switch addresses for performance and routing purposes. For this reason, the traditional address configuration protocols such as DHCP require a huge amount of manual input, leaving them error-prone. In this paper, we present DAC, a generic and automatic Data center Address Configuration system. With an automatically generated blueprint that defines the connections of servers and switches labeled by logical Ids, e.g., IP addresses, DAC first learns the physical topology labeled by device IDs, e.g., MAC addresses. Then, at the core of DAC is its device-to-logical ID mapping and malfunction detection. DAC makes an innovation in abstracting the device-to-logical ID mapping to the graph isomorphism problem and solves it with low time complexity by leveraging the attributes of data center network topologies. Its malfunction detection scheme detects errors such as device and link failures and mis-wirings, including the most difficult case where miswirings do not cause any node degree change. We have evaluated DAC via simulation, implementation, and experiments. Our simulation results show that DAC can accurately find all the hardest-to-detect malfunctions and can auto configure a large data center with 3.8 million devices in 46 s. In our implementation, we successfully auto configure a small 64-server BCube network within 300 ms and show that DAC is a viable solution for data center auto configuration.

CCB (communication channel building): from DAC manager broadcasts the message with level 0 to the last node in the network gets its level; timeout: there is no change in neighboring nodes for at leaf nodes; TC (physical topology collection): from the first leaf node sends out its TCM to DAC manager receives the entire network topology; mapping: device-to-logical ID mapping time including the I/O time; LD (logical IDs dissemination): from DAC manager, sends out the mapping information to all the devices to get their logical IDs.

The newly proposed data center network (DCN) structures go one step further by encoding their topology information into their logical IDs. These logical IDs can take the form of IP address, MAC address, or even newly invented IDs. These structures then leverage the topological information embedded in the logical IDs for scalable and efficient routing. For example, Portland switches choose a routing path by exploiting the location information of destination MAC. BCube servers build a source routing path by modifying one digit at one step based on source and destination BCube IDs. For all the cases above, we need to configure the logical Ids, which may be IP or MAC addresses or BCube, for all the servers and switches. Meanwhile, in the physical topology, all the devices are identified by their unique device IDs, such as MAC addresses.

When the channel is built, the next step is to collect the physical topology. For this, we introduce a Physical topology Collection Protocol (PCP). In PCP, the physical topology information, i.e., the connection information between each node is propagated bottom–up from the leaf devices to the root (i.e., DAC manager) layer by layer. After is collected by DAC manager, we go to the device-to-logical ID mapping module. Device-to-Logical ID mapping after has been collected; we come to device-to-logical ID mapping, which is a key component of DAC. As introduced in Section I, the challenge is how to have the mapping reflect the topological relationship of these devices. To this end, we devise , a fast one-to-one mapping engine, to realize this functionality.

We consider and categorize three malfunction types in data centers: node, link, and mis-wiring. The first type occurs when a given server or switch breaks down from hardware or software reasons, causing it to be completely unreachable and disconnected from the network. The second one occurs when the cable or network card is broken or not properly plugged in so that the connectivity between devices on that link is lost. The third one occurs when wired cables are different from those in the blueprint. These malfunctions may introduce severe problems and downgrade the performance.

Materials and Methods

Existing System

There are very few existing solutions, and none of them can meet all the requirements above. In this paper, we address these problems by proposing DAC—a generic and automatic Data center Address Configuration system for the existing and future data center networks. To make our solution generic, we assume that we only have a blueprint of the to-be-configured data center network, which defines how the servers and switches are connected and labels each device with a logical ID. The blueprint can be automatically generated because all the existing data center network structures are quite regular and can be described either recursively or iteratively.

Proposed System

The newly proposed data center network (DCN) structures go one step further by encoding their topology information into their logical IDs. These logical IDs can take the form of IP address, MAC address, or even newly invented IDs. These structures then leverage the topological information embedded in the logical IDs for scalable and efficient routing. For example, Portland switches choose a routing path by exploiting the location information of destination MAC. BCube servers build a source routing path by modifying one digit at one step based on source and destination BCube IDs. For all the cases above, we need to configure the logical Ids, which may be IP or MAC addresses or BCube, for all the servers and switches. Meanwhile, in the physical topology, all the devices are identified by their unique device IDs, such as MAC addresses.

Event Scheduler

This section talks about the discrete event schedulers of NS. As described in the Overview section, the main users of an event scheduler are network components that simulate packet-handling delay or that need timers. Figure (a) shows each network object using an event scheduler. Note that a network object that issues an event is the one who handles the event later at scheduled time. Also note that the data path between network objects is different from the event path. Actually, packets are handed from one network object to another using send(Packet* p) {target_->recv(p)}; method of the sender and recv(Packet*, Handler* h = 0) method of the receiver.

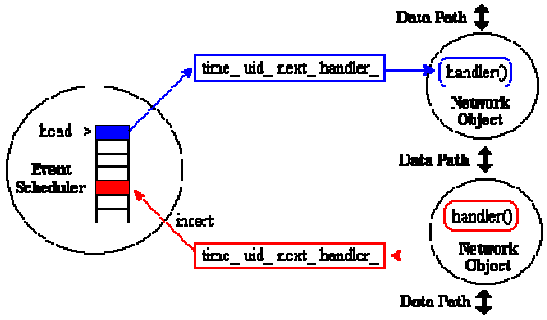


Figure (a) Discrete Event Scheduler

NS has two different types of event schedulers implemented. These are real-time and non-real-time schedulers. For a non-real-time scheduler, three implementations (List, Heap and Calendar) are available, even though they are all logically perform the same. This is because of backward compatibility: some early implementation of network components added by a user (not the original ones included in a package) may use a specific type of scheduler not through public functions but hacking around the internals. The Calendar non-real-time scheduler is set as the default. The real-time scheduler is for emulation, which allow the simulator to interact with a real network. Currently, emulation is under development although an experimental version is available. The following is an example of selecting a specific event scheduler:

```
.                .                .
set    ns    [new    Simulator]
$ns    use-scheduler    Heap
. . .
```

Another use of an event scheduler is to schedule simulation events, such as when to start an FTP application, when to finish a simulation, or for simulation scenario generation prior to a simulation run. An event scheduler object itself has simulation scheduling member functions such as at time "string" that issue a special event called AtEvent at a specified simulation time. An "AtEvent" is actually a child class of "Event", which has an additional variable to hold the given string. However, it is treated the same as a normal (packet related) event within the event scheduler. When a simulation is started, and as the scheduled time for an AtEvent in the event queue comes, the AtEvent is passed to an "AtEvent handler" that is created once and handles all AtEvents, and the OTcl command specified by the string field of the AtEvent is executed. The following is a simulation event scheduling line added version of the above example.

```
.                .                .
set    ns    [new    Simulator]
$ns    use-scheduler    Heap
$ns    at    300.5    "complete_sim"
.                .                .
proc    complete_sim    {}    {
.                .                .
}
```

ou might noticed from the above example that at time "string" is a member function of the Simulator object (set ns [new Simulator]). But remember that the Simulator object just acts as a user interface, and it actually calls the member functions of network objects or a scheduler object that does the real job. Followings are a partial list and brief description of Simulator object member functions that interface with scheduler member functions:

Simulator instproc now	# return scheduler's n
Simulator instproc at args	# schedule execution
Simulator instproc at-now args	# schedule execution
Simulator instproc after n args	# schedule execution
Simulator instproc run args	# start scheduler
Simulator instproc halt	# stop (pause) schedu

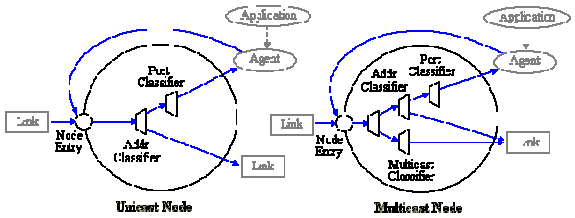
Network Components

This section talks about the NS components, mostly compound network components. The root of the hierarchy is the TclObject class that is the superclass of all OTcl library objects (scheduler, network components, timers and the other objects including NAM related ones). As an ancestor class of Tcl Object, Ns Object class is the superclass of all basic network component objects that handle packets, which may compose compound network objects such as nodes and links. The basic network components are further divided into two subclasses, Connector and Classifier, based on the number of the possible output data paths. The basic network objects that have only one output data path are under the Connector class, and switching objects that have possible multiple output data paths are under the Classifier class.

(a)Node and Routing

A node is a compound object composed of a node entry object and classifiers as shown in Figure(b) There are two types of nodes in NS. A unicast node has an address classifier that does unicast routing and a port classifier. A multicast

node, in addition, has a classifier that classify multicast packets from unicast packets and a multicast classifier that performs multicast routing.



Figure(b) Node (Unicast and Multicast)

In NS, Unicast nodes are the default nodes. To create Multicast nodes the user must explicitly notify in the input OTcl script, right after creating a scheduler object, that all the nodes that will be created are multicast nodes. After specifying the node type, the user can also select a specific routing protocol other than using a default one.

(b)Unicast

- $ns rtproto type
- type: Static, Session, DV, cost, multi-path

(c)Multicast

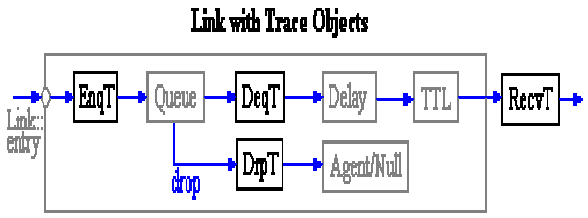
- $ns multicast (right after set $ns [new Scheduler])
- $ns mrtproto type
- type: CtrMcast, DM, ST, BST

(d)Link

A link is another major compound object in NS. When a user creates a link using a duplex-link member function of a Simulator object.One thing to note is that an output queue of a node is actually implemented as a part of simplex link object. Packets dequeued from a queue are passed to the Delay object that simulates the link delay, and packets dropped at a queue are sent to a Null Agent and are freed there. Finally, the TTL object calculates Time To Live parameters for each packet received and updates the TTL field of the packet.

(e)Tracing

In NS, network activities are traced around simplex links. If the simulator is directed to trace network activities (specified using $ns trace-all file or $ns namtrace-all file), the links created after the command will have the following trace objects inserted as shown in Figure(c) Users can also specifically create a trace object of type type between the given src and dst nodes using the create-trace {type file src dst} command.

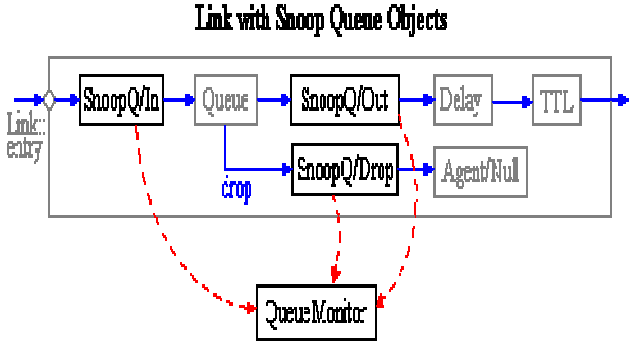


Figure(c) Inserting Trace Objects

When each inserted trace object (i.e. EnqT, DeqT, DrpT and RecvT) receives a packet, it writes to the specified trace file without consuming any simulation time, and passes the packet to the next network object. The trace format will be examined in the General Analysis Example section.

Queue Monitor

Basically, tracing objects are designed to record packet arrival time at which they are located. Although a user gets enough information from the trace, he or she might be interested in what is going on inside a specific output queue. For example, a user interested in RED queue behavior may want to measure the dynamics of average queue size and current queue size of a specific RED queue (i.e. need for queue monitoring). Queue monitoring can be achieved using queue monitor objects and snoop queue objects as shown in Figure(d).



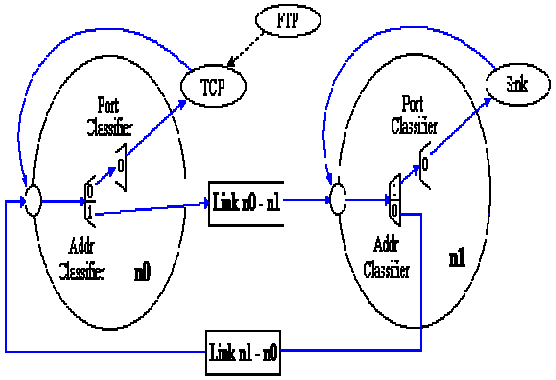
Figure(d) Monitoring Queue

When a packet arrives, a snoop queue object notifies the queue monitor object of this event. The queue monitor using this information monitors the queue. A RED queue monitoring example is shown in the RED Queue Monitor Example section. Note that snoop queue objects can be used in parallel with tracing objects even though it is not shown in the above figure.

(a)Packet Flow Example

Until now, the two most important network components (node and link) were examined. Figure(e) shows internals of an example simulation

network setup and packet flow. The network consist of two nodes (n0 and n1) of which the network addresses are 0 and 1 respectively. A TCP agent attached to n0 using port 0 communicates with a TCP sink object attached to n1 port 0. Finally, an FTP application (or traffic source) is attached to the TCP agent, asking to send some amount of data.



Figure(e) Packet Flow Example

Note that the above figure does not show the exact behavior of a FTP over TCP. It only shows the detailed internals of simulation network setup and a packet flow.

(b)Packet

A NS packet is composed of a stack of headers, and an optional data space. As briefly mentioned in the "Simple Simulation Example" section, a packet header format is initialized when a Simulator object is created, where a stack of all registered (or possibly useable) headers, such as the common header that is commonly used by any objects as needed, IP header, TCP header, RTP header (UDP uses RTP header) and trace header, is defined, and the offset of each header in the stack is recorded. What this means is that whether or not a specific header is used, a stack composed of all registered headers is created when a packet is allocated by an agent, and a network object can access any header in the stack of a packet it processes using the corresponding offset value.

Usually, a packet only has the header stack (and a data space pointer that is null). Although a packet can carry actual data (from an application) by allocating a data space, very few application and agent implementations support this. This is because it is meaningless to carry data around in a non-real-time simulation. However, if you want to implement an application that talks to another application cross the network, you might want to use this feature with a little modification in the underlying agent implementation. Another possible approach would be creating a new header for the application and modifying the underlying agent to write data received from the application to the new header. The second approach is shown as an example in a later section called "Add New Application and Agent".

Trace Analysis Example

This section shows a trace analysis example. Example(1) is the same OTcl script as the one in the "Simple Simulation Example" section with a few lines added to open a trace file and write traces to it. For the network topology it generates and the simulation scenario.To run this script download "ns-simple-trace.tcl" and type "ns ns-simple-trace.tcl" at your shell prompt.

```
# # #
#Open the NAM trace file
set nf [open out.nam w]
$ns namtrace-all $nf

#Open the Trace file
set tf [open out.tr w]
$ns trace-all $tf

#Define a 'finish' procedure
proc finish {} {
        global ns nf tf
        $ns flush-trace
        #Close the NAM trace file
        close $nf
        #Close the Trace file
        close $tf
        #Execute NAM on the trace file
        exec nam out.nam &
        exit 0
}

# # #
```

Example(1). Trace Enabled Simple NS Simulation Script

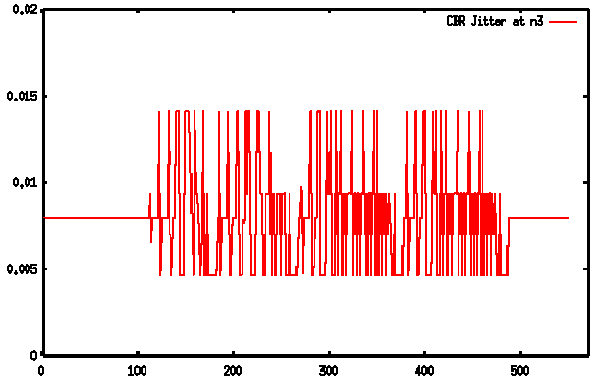
Running the above script generates a NAM trace file that is going to be used as an input to NAM and a trace file called "out.tr" that will be used for our simulation analysis. Each trace line starts with an event (+, -, d, r) descriptor followed by the simulation time (in seconds) of that event, and from and to node, which identify the link on which the event occurred. Look at Figure(c) in the "Network Components" section to see where in a link each type of event is traced. The next information in the line before flags (appeared as "------" since no flag is set) is packet type and size (in Bytes). Currently, NS implements only the Explicit Congestion Notification (ECN) bit, and the remaining bits are not used. The next field is flow id (fid) of IPv6 that a user can set for each flow at the input OTcl script. Even though fid field may not used in a simulation, users can use this field for analysis purposes. The fid field is also used when

specifying stream color for the NAM display. The next two fields are source and destination address in forms of "node.port". The next field shows the network layer protocol's packet sequence number. Note that even though UDP implementations do not use sequence number, NS keeps track of UDP packet sequence number for analysis purposes. The last field shows the unique id of the packet.

Having simulation trace data at hand, all one has to do is to transform a subset of the data of interest into a comprehensible information and analyze it. Down below is a small data transformation example. This example uses a command written in perl called "column" that selects columns of given input. To make the example work on your machine, you should download "column" and make it executable (i.e. "chmod 755 column"). Following is a tunneled shell command combined with awk, which calculates CBR traffic jitter at receiver node (n3) using data in "out.tr", and stores the resulting data in "jitter.txt".

```
cat out.tr | grep " 2 3 cbr " | grep ^r | column 1 10 | awk '{dif = $2 - old2; if (dif == 0) dif = 1; if (dif > 0)
{printf("%d\t%f\n", $2, ($1 - old1) / dif); old1 = $1; old2 = $2}}' > jitter.txt
```

This shell command selects the "CBR packet receive" event at n3, selects time (column 1) and sequence number (column 10), and calculates the difference from last packet receive time divided by difference in sequence number (for loss packets) for each sequence number. The following is the corresponding jitter graph that is generated using gnuplot. The X axis show the packet sequence number and the Y axis shows simulation time in seconds.



Figure(f) CBR Jitter at The Receiving Node (n3)
You might also check for more utilities in the Example Utilities section.

This section showed an example of how to generate traces in NS, how to interpret them, and how to get useful information out from the traces. In this example, the post simulation processes are done in a shell prompt after the simulation. However, these processes can be included in the input OTcl script, which is shown in the next section.

The following is the explanation of the script above. In general, an NS script starts with making a Simulator object instance.set ns [new Simulator]: generates an NS simulator object instance, and assigns it to variable ns (italics is used for variables and values in this section). What this line does is the following: Initialize the packet format (ignore this for now) Create a scheduler (default is calendar scheduler) Select the default address format (ignore this for now)The "Simulator" object has member functions that do the following:

1. Create compound objects such as nodes and links (described later)
2.Connect network component objects created (ex. attach-agent)
3.Set network component parameters (mostly for compound objects)
4.Create connections between agents (ex. make connection between a "tcp" and "sink")
5.Specify NAM display options Etc. Most of these functions are for simulation setup (described later as plumbing functions in the Overview section) and scheduling, however some of them are for the NAM display. The "Simulator" object member function implementations are located in the "ns-2/tcl/lib/ns-lib.tcl" file.

***1.$ns* color *fid color*:** is to set color of the packets for a flow specified by the flow id (fid). This member function of "Simulator" object is for the NAM display, and has no effect on the actual simulation.

***2.$ns* namtrace-**all *file-descriptor*: This member function tells the simulator to record simulation traces in NAM input format. It also gives the file name that the trace will be written to later by the command $ns flush-trace. Similarly, the member function trace-all is for recording the simulation trace in a general format.

3.proc *finish* {}: is called after this simulation is over by the command $ns at 5.0 "finish". In this function, post-simulation processes are specified.

4.set *n0* [*$ns* node]: The member function node creates a node. A node in NS is compound object made of address and port classifiers (described in a later section). Users can create a node by separately creating an address and a port classifier objects and connecting them together. However, this member function of Simulator object makes the job easier. To see how a node is created, look at the files: "ns-2/tcl/libs/ns-lib.tcl" and "ns-2/tcl/libs/ns-node.tcl".

5.$ns duplex-link *node1 node2 bandwidth delay queue-type*:. creates two simplex links of specified bandwidth and delay, and connects the two specified nodes. In NS, the output queue of a node is implemented as a part of a link, therefore users should specify the queue-type when creating links. In

the above simulation script, DropTail queue is used. If the reader wants to use a RED queue, simply replace the word DropTail with RED. The NS implementation of a link is shown in a later section. Like a node, a link is a compound object, and users can create its sub-objects and connect them and the nodes. Link source codes can be found in "ns-2/tcl/libs/ns-lib.tcl" and "ns-2/tcl/libs/ns-link.tcl" files. One thing to note is that you can insert error modules in a link component to simulate a lossy link (actually users can make and insert any network objects). Refer to the NS documentation to find out how to do this

***6.$ns* queue-limit *node1 node2 number*:** This line sets the queue limit of the two simplex links that connect node1 and node2 to the number specified. At this point, the authors do not know how many of these kinds of member functions of Simulator objects are available and what they are. Please take a look at "ns-2/tcl/libs/ns-lib.tcl" and "ns-2/tcl/libs/ns-link.tcl", or NS documentation for more information.

***7.$ns* duplex-link-op *node1 node2 ...*:** The next couple of lines are used for the NAM display. To see the effects of these lines, users can comment these lines out and try the simulation. Now that the basic network setup is done, the next thing to do is to setup traffic agents such as TCP and UDP, traffic sources such as FTP and CBR, and attach them to nodes and agents respectively.

8.set *tcp* [new *Agent/TCP*]: This line shows how to create a TCP agent. But in general, users can create any agent or traffic sources in this way. Agents and traffic sources are in fact basic objects (not compound objects), mostly implemented in C++ and linked to OTcl. Therefore, there are no specific Simulator object member functions that create these object instances. To create agents or traffic sources, a user should know the class names these objects (Agent/TCP, Agnet/TCPSink, Application/FTP and so on). This information can be found in the NS documentation or partly in this documentation. But one shortcut is to look at the "ns-2/tcl/libs/ns-default.tcl" file. This file contains the default configurable parameter value settings for available network objects. Therefore, it works as a good indicator of what kind of network objects are available in NS and what are the configurable parameters.

***9.$ns* attach-agent *node agent*:** The attach-agent member function attaches an agent object created to a node object. Actually, what this function does is call the attach member function of specified node, which attaches the given agent to itself. Therefore, a user can do the same thing by, for example, $n0 attach $tcp. Similarly, each agent object has a member function attach-agent that attaches a traffic source object to itself.

10.$ns* connect *agent1 agent2: After two agents that will communicate with each other are created, the next thing is to establish a logical network connection between them. This line establishes a network connection by setting the destination address to each others' network and port address pair. Assuming that all the network configuration is done, the next thing to do is write a simulation scenario (i.e. simulation scheduling). The Simulator object has many scheduling member functions. However, the one that is mostly used is the following:

***11.$ns* at *time "string"*:** This member function of a Simulator object makes the scheduler (scheduler_ is the variable that points the scheduler object created by [new Scheduler] command at the beginning of the script) to schedule the execution of the specified string at given simulation time. For example, *$ns* at *0.1 "$cbr start"* will make the scheduler call a start member function of the CBR traffic source object, which starts the CBR to transmit data. In NS, usually a traffic source does not transmit actual data, but it notifies the underlying agent that it has some amount of data to transmit, and the agent, just knowing how much of the data to transfer, creates packets and sends them.

After all network configuration, scheduling and post-simulation procedure specifications are done, the only thing left is to run the simulation. This is done by *$ns* run.

Design and Implementation Constraints

Constraints in Analysis

1.Constraints as Informal Text
2.Constraints as Operational Restrictions
3.Constraints Integrated in Existing Model Concepts
4.Constraints as a Separate Concept
5.Constraints Implied by the Model Structure

Constraints in Design

1.Determination of the Involved Classes
2.Determination of the Involved Objects
3.Determination of the Involved Actions
4.Determination of the Require Clauses
5.Global actions and Constraint Realization

External Interface Requirements

User Interfaces

1. Graphical User Interfaces not in this product.
2. Users are communicated with Buttons with network animator.

Hardware Interfaces

Linux environment of system and basic need of system feature like random access memory etc.

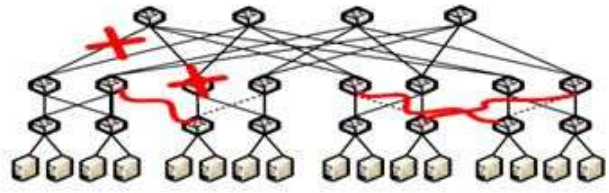
Software Interfaces

1.This software is interacted with the TCP/IP protocol.
2.This product is interacted with the and linux
3.This product is interacted with the ServerSocket
4.This product is interacted with TCL

Performance Requirements

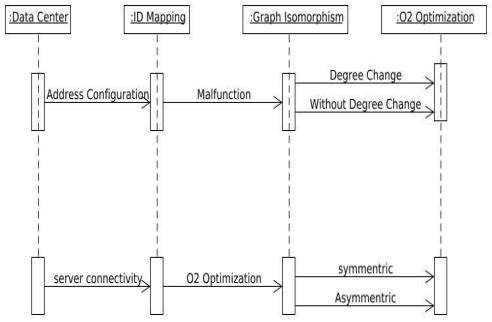
The maximum satisfactory response time to be experienced most of the time for each distinct type of user-computer interaction, along with a definition of most of the time. Response time is measured from the time that the user performs the action that says "Go" until the user receives enough feedback from the computer to continue the task. It is the user's subjective wait time. It is not from entry to a subroutine until the first write statement. If the user denies interest in response time and indicates that only the result is of interest, you can ask whether "ten times your current estimate of stand-alone execution time" would be acceptable. If the answer is "yes," you can proceed to discuss throughput. Otherwise, you can continue the discussion of response time with the user's full attention. The response time that is minimally acceptable the rest of the time. A longer response time can cause users to think the system is down. You also need to specify rest of the time; for example, the peak minute of a day, 1 percent of interactions. Response time degradations can be more costly or painful at a particular time of the day.

Architecture of Node failure:



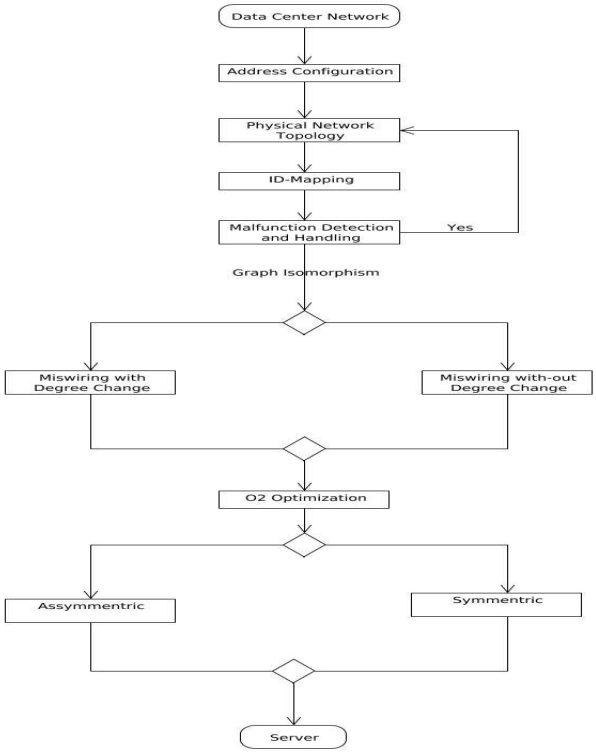
Figure(e) Architecture of node failure

Sequence Diagram:



Figure(g) Sequence diagram

Activity Diagram



Figure(h) Activity Diagram

System Design

Modules

a. Devices-to-Logical ID Mapping
b. Malfunction Detection
c. Simulation.

a. Devices-to-Logical ID Mapping

When the channel is built, the next step is to collect the physical topology. For this, we introduce a Physical topology Collection Protocol (PCP). In PCP, the physical topology information, i.e., the connection information between each node is propagated bottom–up from the leaf devices to the root (i.e., DAC manager) layer by layer. After is collected by DAC manager, we go to the device-to-logical ID mapping module. Device-to-Logical ID mapping after has been collected; we come to device-to-logical ID mapping, which is a key component of DAC. As introduced in Section I, the challenge is how to have the mapping reflect the topological relationship of these devices. To this end, we devise, a fast one-to-one mapping engine, to realize this functionality.

b. Malfunction Detection:

We consider and categorize three malfunction types in data centers: node, link, and

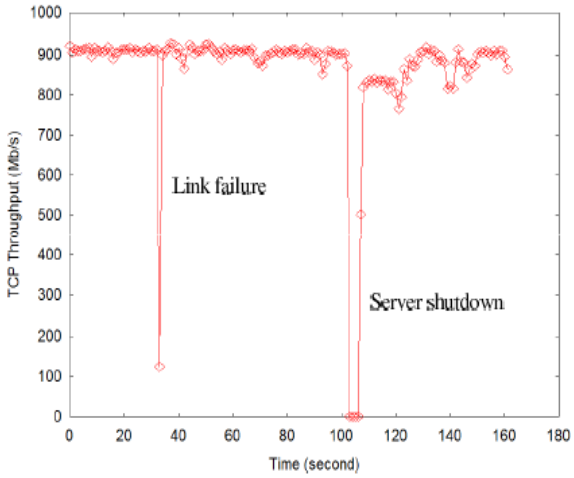
miswiring. The first type occurs when a given server or switch breaks down from hardware or software reasons, causing it to be completely unreachable and disconnected from the network. The second one occurs when the cable or network card is broken or not properly plugged in so that the connectivity between devices on that link is lost. The third one occurs when wired cables are different from those in the blueprint. These malfunctions may introduce severe problems and downgrade the performance.

c. Simulation:

Our simulation results show that DAC can accurately find all the hardest-to-detect malfunctions and can auto configure a large data center with large amount devices. In our implementation on a B Cube test bed, DAC has used. To successfully auto configure all the servers. Our implementation experience and experiments show that DAC is a viable solution for data center network auto configuration. we focus on simulations on the mis-wirings where there is no degree change. We evaluate the accuracy of our algorithm proposed in detecting such malfunction.

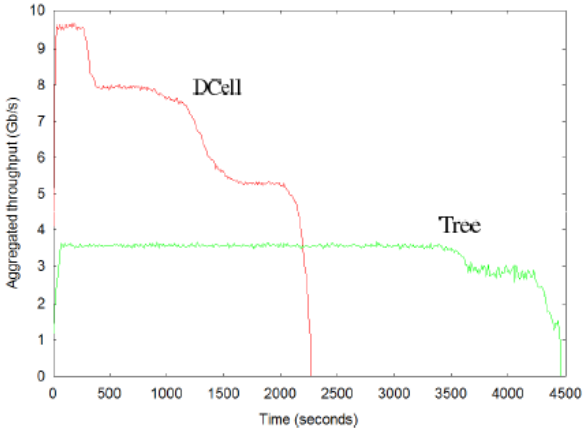
Result & Conclusion

TCP Throughput Node and Link failure



Above Diagram shows that, the link failure occurs only 1-second throughput degdation, while the node failure occurs a 5-second throughput outage that corresponds to our link-state timeout value.

Aggregate TCP Throughput



The TCP throughput is recovered to the best value after only a few seconds. second, our implementation detects link failures much faster than node failure, because of using the medium sensing cause queue to build up at each sender's buffer. All TCP connections at a server the same sending buffer on a network port. Therefore, all TCP connections are slowed down, including those not traversing the root switch. This results iin much smaller aggregate TCP throughput.

Conclusion

The results provided within this paper designed, evaluated, and implemented DAC, a generic and automatic Data center Address Configuration system. Our simulation results show that DAC can accurately find all the hardest-to-detect malfunctions and can auto configure a large data center . At the core of DAC is its device-to-logical ID mapping and malfunction detection. DAC has made an innovation in abstracting the device-to-logical ID mapping to the graph isomorphism problem and solved it in low time complexity by leveraging the sparsity and symmetry (or asymmetry) of data center structures. Our implementation experience and experiments show that DAC is a viable solution for data center network auto configuration.

Acknowledgement

Thanks to Dr. T. Kiran Kumar, who comforted me in various link failure(mis-wiring) issues simulating this paper and also Mrs. s. Arulselvi for her help in seeding me regarding the practical implementation and its pros and cons. The insightful and detailed feedback and suggestion by them which improved content and presentation of this paper.

References

[1] R. H. Katz, "Tech titans building boom," *IEEE Spectrum*, vol. 46, no. 2, pp. 40–54, Feb.2009.

[2] L. Barroso, J. Dean, and U. Hölzle, "Websearch for a planet: The Google cluster architecture," *IEEE Micro*, vol. 23, no. 2, pp. 22–28, Mar. 2003.

[3] R. Droms, "Dynamic host configuration protocol," RFC 2131, Mar. 1997

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system ," in *Proc. ACM SOSP*, 2003, pp. 29–43.

[5] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. OSDI*, 2004, pp. 137–150.

[6] C. Guo, H.Wu,K.Tan,L. Shi,Y.Zhang, and S. Lu, "DCell:Ascalable and fault tolerant network structure for data centers," in *Proc. ACM SIGCOMM*, 2008, pp. 75–86.

[7] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "BCube: A high performance, server-centric network architecture for modular data centers," in *Proc. ACM SIGCOMM*, 2009, pp. 63–74.

[8] R. N. Mysore, A. Pamboris, N. Farrington, N. Subramanya, and A. Vahdat, "PortLand: A scalable fault-tolerant layer 2 data centernetwork fabric," in *Proc. ACM SIGCOMM*, 2009, pp. 39–50.

[9] A.Greenberg,N. Jain, S.Kandula, C. Kim, P. Lahiri, D.Maltz, P. Patel, and S. Sengupta, "VL2: A scalable and flexible data centernetwork," in *Proc. ACM SIGCOMM*, 2009, pp. 51–62.

[10] "Human errors most common reason for data center outages," Oct. 2007 [Online].