Fall | 2014

Technical Report

Project - **Media Delivery Network Simulator**

Sponsor - Ericsson

Point of contact - Vladimir Katardjiev, Alvin Jude

Faculty Advisor - Jia Zhang

Team - Geng(Jeremy) Fu, Jigar Patel, Vinay Kumar Vavili, Hao Wang

Table of Contents

1. Introduction

The largest minority of traffic traversing the Internet today is video, in various shapes and sizes; combined, all media forms have a peak time majority of network traffic. The amount of media traffic is only set to increase, as more and more people choose IP as their delivery method of choice. The natural question, therefore, is whether we are transferring that traffic in a good manner, or if we can do better.

In order to answer the question of how to transfer the traffic well, we must first create the traffic. As experimenting on customer networks is very costly, and current models for traffic simulation assume traditional loads, creating a simulator will create a distinct opportunity to experiment with different algorithms and attempt to find how to better transfer media.
The goal of this project is to build a “life sized” simulation of internet-based media distribution, with a flexible framework that will allow tinkering, experimentation and evolution.

2. Motivation

The impact of the internet on our daily lives is the biggest technological innovation of the last decade. It has re-shaped our planet, our lives and global industries – including TV and Media.

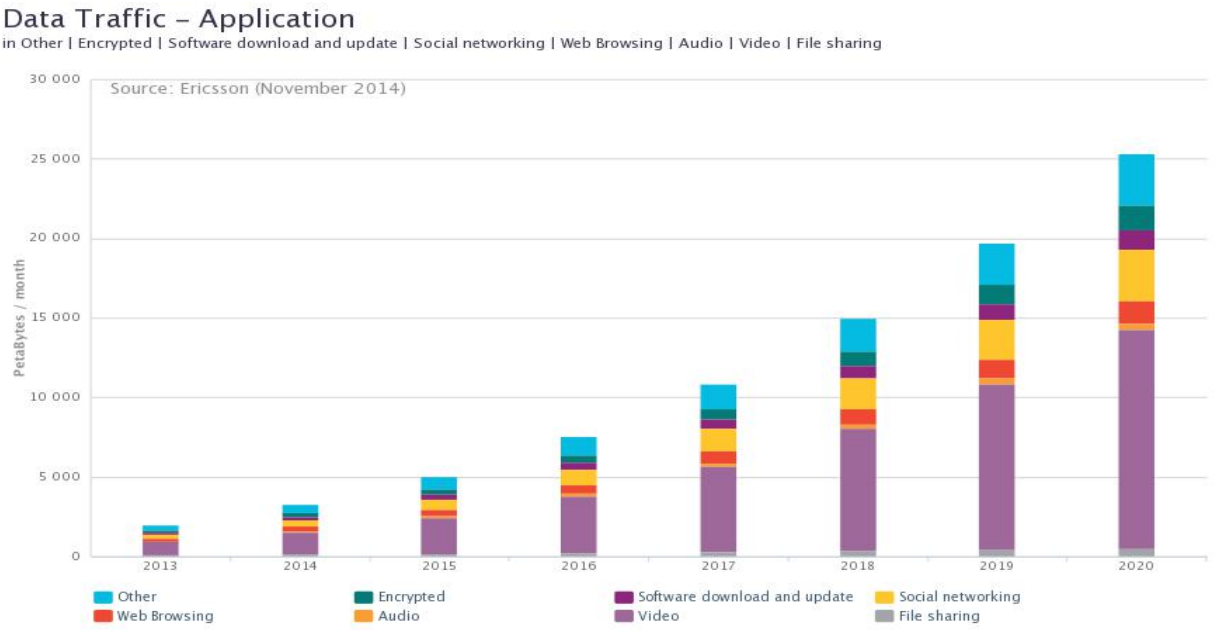
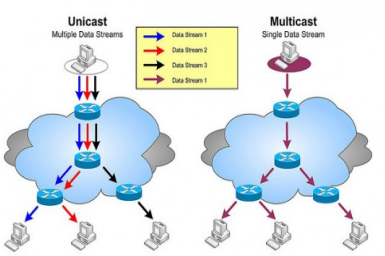
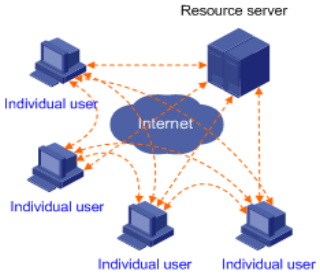


Fig 2.1 Data traffic forecast

Recent years have seen global broadband IP capacity enable the explosion in video- enabled connected devices. The number of IP connected devices that can view video has grown from 200 million (personal computers) to over 1.6 billion in the period 2000-2013 alone. In 2020, there are expected to be 15 billion video-enabled connected devices globally[1]. Global data traffic is expected to grow over 10x by 2020 with video predicted to be 50% of all this traffic.[1] Let us consider Netflix on-demand video streaming service. Netflix has a catalog which is between three and four petabytes in size. But it streams over 114,000 years of video every month. That translates into almost 1,000 petabytes of bandwidth per month. This indicates that there is lots of repeated wasteful transfer of data. Things look even worse when looking at live streams, which show the same content, at the same time. This indicates that streaming uses linear resources . Can we do better by employing other ways to deliver the content? To answer this question, we need data to compare and evaluate various models of delivery.
There are various ways to deliver media data across the internet including unicast, multicast, P2P and cache based.





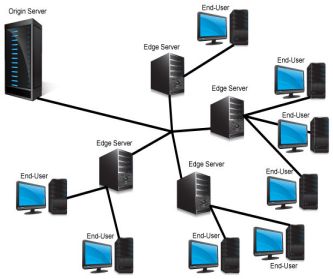


Figure 2.2 unicast/multicast *Figure 2.3 peer to peer* *Figure 2.4 cache based*

There is no “One size fits all” solution for delivery of media. For instance, we cannot use a peer to peer system or cache based system for live streaming. And in case of on-demand stored content, multicast streaming is not possible as each viewer maybe viewing different parts of the video.

To test out any model of media delivery, we will have to,

- Setup/tear-down the various nodes which are part of the media delivery change on-demand
- Generate data to flow through the media delivery change
- Perform processing on nodes that emulate various functions like encoding, transcoding, ad-insertion etc
- Collect various metrics of the system including transfer rate, packet loss and latency
- Visualize the topology and state of the system for immediate feedback
- Collect and aggregate statistics and events in the system for further study

We can in-fact do all of the above in a real-world deployment of media delivery network. But that would involve adding additional modules in the customer devices to collect and report statistics. And we do not have fine grained control on the load on the system. We will have to depend on real time load which involves testing over long periods of time. All the above factors renders testing in real-world networks a costly affair both in terms of money and time.

A Media delivery network simulator on the other hand can achieve all the mentioned requirements in a managed and distributed environment based on user input. A simulator also allows for flexibility in the sense that it allows for extensibility of existing node types to simulate functions of media delivery like ad-insertion, transcoding etc.

3. Related work

The following network simulators were studied to see if we could leverage features from them for this project.

- Ns2/Ns3[2]
- Omnet++[3]

Ns2/Ns3

Ns is a discrete event simulator targeted at networking research. Ns provides substantial support for simulation of TCP, routing, and multicast protocols over wired and wireless (local and satellite) networks.

Omnet++

OMNeT++ is an extensible, modular, component-based C++ simulation library and framework, primarily for building network simulators. "Network" is meant in a broader sense that includes wired and wireless communication networks, on-chip networks, queueing networks, and so on. Domain-specific functionality such as support for sensor networks, wireless ad-hoc networks, Internet protocols, performance modeling, photonic networks, etc., is provided by model frameworks, developed as independent projects

A simulator model of a real-world system is necessarily a simplification of the real-world system itself. Both of the above systems use mathematical models to simulate network functions. They use an event based simulation based model that models the operation of a system as a discrete sequence of events in time. This implies that both of these simulators do not actually send/receive/process IP packets, but infact model these as events based on data from real-world experimentation.

Further, these systems are not distributed. They run on a single machine and hence not scalable to our requirements. Parallel discrete event simulation (PDES), sometimes called distributed simulation, refers to the execution of a single discrete event simulation program on a parallel computer. Both Ns2/Ns3 and Omnet++ have support for PDES. But again, this involves event simulation and not actual sending and receiving of data.

It is important to note that the main aim of this project is to develop a “life sized” simulator of media delivery network involving transfer of real IP data. So the main aim of the system is to manage the flow of data based on user input across a large number of nodes that are distributed and collecting different metrics. The aim is not to simulate the network transfer of data itself which is done by the real network in our system.

Considering the existing systems and the requirement of the project, we decided that it is best to build a system from scratch. The systems focus will be to manage and monitor the system rather than simulate network functions. The “simulator” aspect of this system comes into picture in a sense that it simulates media delivery chain functions like encoding/transcoding etc.

4. System Design

4.1 Overview

Media delivery network(MDN) simulator is a distributed systems. The system follows master-slave architecture. Components running on different machines simulate an overlay network over the Internet. Therefore, the simulated network is an IP-independent network.

The figure 4.1 illustrates how the MDN simulator can be divided by the functionality of different components. The data layer contains components nodes (e.g. source node, sink node, processing node and relay node) and data path. The management layer consists of message bus, node containers and master, which is used to control the system. The user interface is denoted as web client, which provides interactions with users.

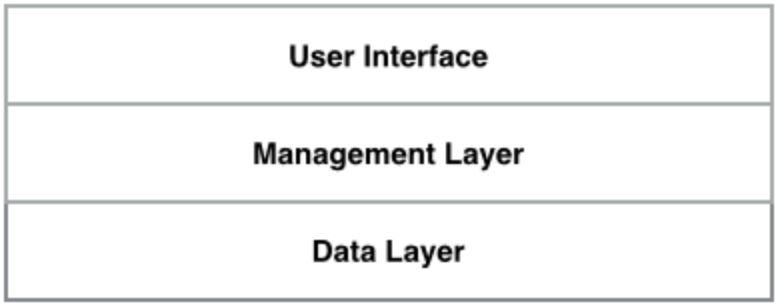


Figure 4.1 Layers of MDN Simulator

4.2 Data Layer

The data layer forms the actual media delivery network. The data layer generates the life-sized media traffic, performs some processing operations on media data and delivers the data from the source of the media chain to the clients. 4 basic types of nodes are identified and implemented: Source Node, Processing Node, Relay Node and Sink Node.

Source Node: Source node generates the life-sized media data. The size of data and the transmit rate can be configured by users input. Source node also inserts markers such as packet ID and sending timestamp for sake of computing packet loss rate and end-to-end latency at downstream nodes. Figure 4.2 is generated by MDN simulator at user interface. The components circled by red is the source node. The source node doesn't take any input and is the start of a media chain.

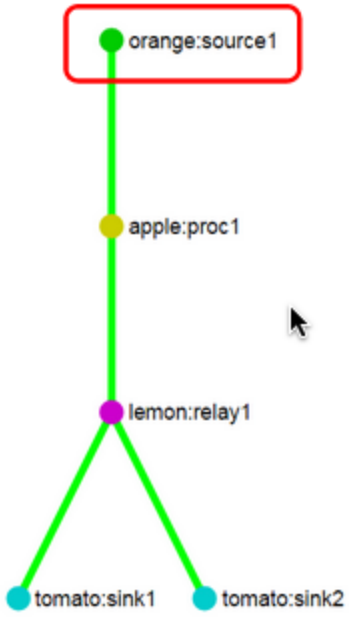


Figure 4.2 Source Node

Processing Node: Processing node simulates actual processing operations on media data such as encoding of stream data, adds insertion and subtitle addition. As a simulator, MDN simulator doesn't perform real processing operations on data. Instead, it just consumes CPU by for loops and memory by allocating dummy objects. The processing load is configurable as well. Users can add corresponding load to the media data based on actual processings performed on it.

At the input interface, where processing acts as a receiver to its upstream, processing node also monitors the packet loss rate.

The circled component in figure 4.3 is the processing node. Processing node simply takes one input and has one output.

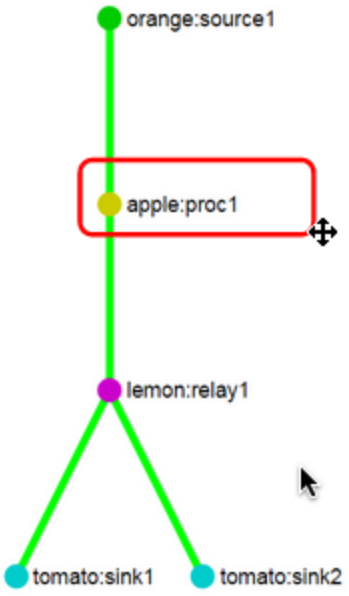


Figure 4.3 Processing Node

Relay Node: Relay node takes one input, duplicates data and delivers to multiple outputs. Relay node does broadcast to downstreams. However, it is worthwhile to mention that this broadcast doesn't mean broadcast in IP layer. Since the data is sent via UDP socket, it should be regarded as unicast in IP layer and broadcast at application layer.

The relay node can be dynamically attached with a new downstream. Upon an addition of downstream, it automatically duplicates a new copy of data and sends to it since the time of addition. On the other hand, a downstream can be dynamically removed from the relay node. Upon the removal, the relay node needs to decide whether continues working. If a relay node doesn't have any active downstream that consumes the media data, relay node notifies its upstream to stop sending data and stops its functionality. If there is at least one downstream consuming data, relay node continues working.

As like the processing node, relay node acts as receiver to its upstream. Therefore it also monitors the packet loss.

The circled component in figure 4.4 is a typical relay node. In this scenario, the relay nodes has two downstreams. The relay node can take and only can take one input and forward data to at least one downstream at output.

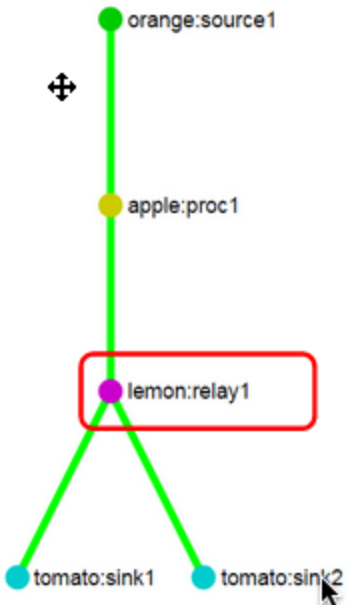


Figure 4.4 Relay Node

Sink Node: Sink node acts as the consumers of media data. The consumers can be audience in real scenario. In P2P network, it can also act as a peer in the phase of caching data. It can later sends the data to its peer. The role of this node is changed later, which calls for the requirement of interchangeable functionality during runtime from the system.

In terms of monitoring the performance of the system, sink node collects packet loss rate and computes the end-to-end delay of each packet.

Figure 4.5 demonstrates sink nodes. Sink nodes are always the end of a media chain and doesn't generate any output as itself consumes data.

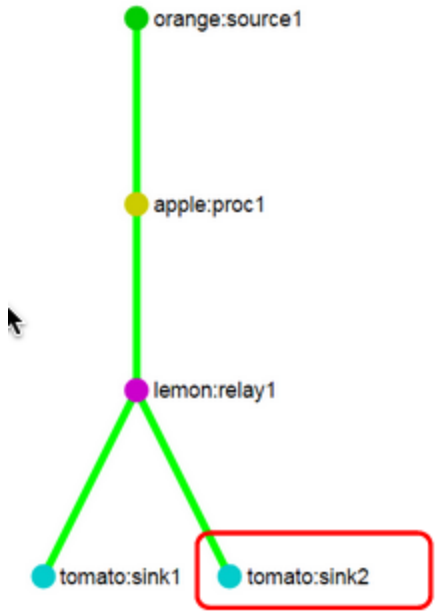


Figure 4.5 Sink Node

Media data flows in and out at different nodes via data path. In the data path, media data is sent via UDP socket. The reasons to use UDP sockets are as follows:

1. UDP protocol just wrap up IP protocol providing much less service than TCP protocol. Since UDP doesn't have any flow control and congestion control, it is possible to generate enough traffic loads to the reach the limitation of the network. And it is possible to observe packets loss by UDP sockets compared to TCP sockets because TCP protocol ensures an in-order and reliable end-to-end communication.
2. In spite of the fact most current media delivery technologies using TCP as underlying transport protocol, emerging technologies such as QUIC are trying to move to UDP sockets. For purpose of research, it is more interesting to use UDP sockets.

Nevertheless the system used UDP sockets in data path, it is very easy to change to TCP sockets.

4.3 Management Layer

Management layer forms the backbone of the distributed systems and orchestrates components to function. And it is also in charge of collecting metrics from different components. Management layer consists of three components: master, node containers and message bus.

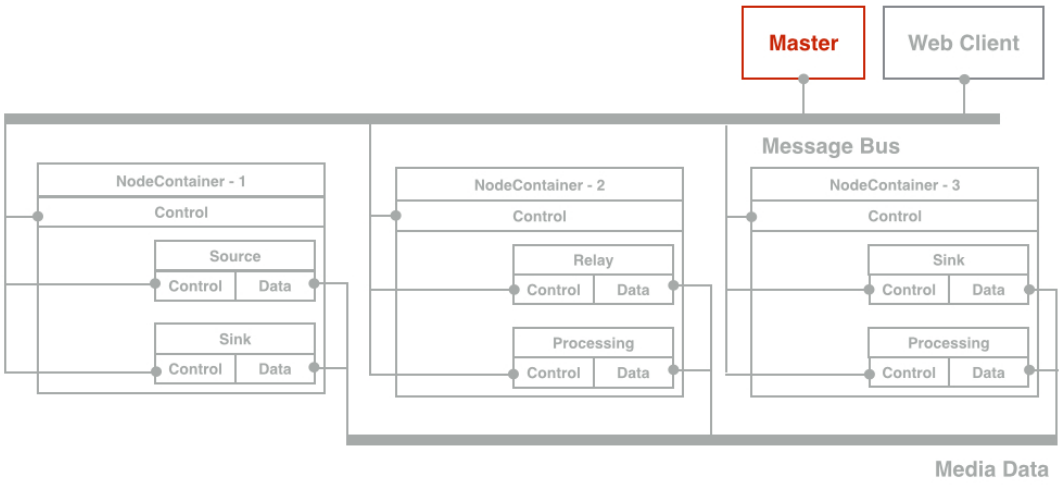


Figure 4.6 Mater

Master

The master is the core of management layer and is the core of the systems as well as is shown in figure 4.6. Master communicates with every component in the system via message bus as explained below.

Web client: In the communication between master and web client, web client sends the control message generated by user, such as start or stop a simulation, attach a new flow to a media chain and reset the whole systems. Master sends the view that reflects the status of the system.

Node container: Master configures the functionalities of each node container by instantiating functional nodes at data layer. Node containers report the states of each node they host to the master. Therefore master can take appropriate action based on the the states of nodes.

Node: Master orchestrates nodes to form the median delivery network. Master contains the whole picture of the topology and translate this knowledge to each node. Nodes only contains the knowledge of their upstream node and downstream nodes if apply. Master also controls the behavior of each node directly, such as start or stop sending data to the data path. Nodes report the performance metrics to master in return.

Node Container

Node containers are different processes running on different machines. Node containers are denoted by the red part in figure 4.6. Each node container can host multiple nodes. Therefore node containers act as composite nodes. Node containers provide flexibility to the system as each machine can be configured with different functionalities. The functionalities are configured by the master during the runtime and thus are interchangeable during run time.

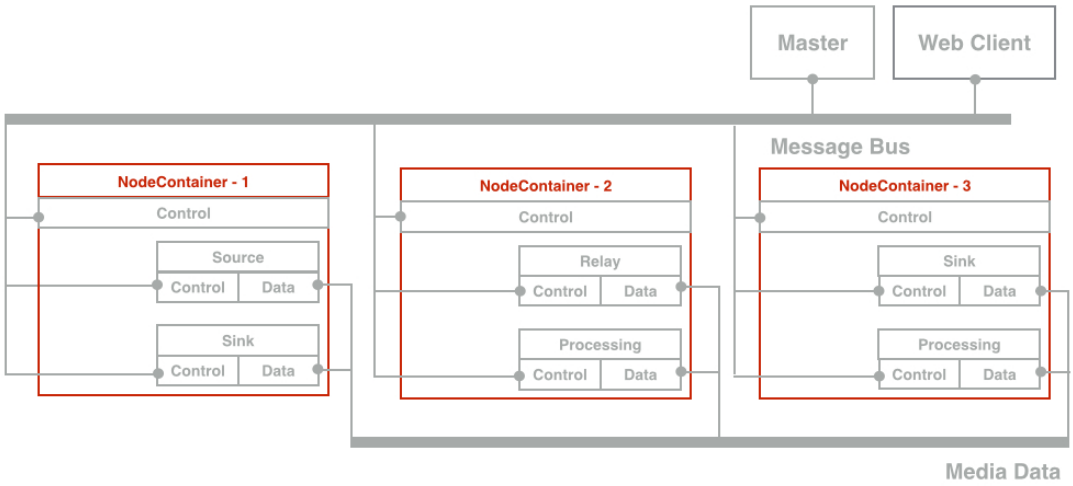


Figure 4.7 Node Container

Message Bus

Message bus is used as a channel to pass all control messages. Message bus connects every component in the system. Message bus is backed by reliable communication to ensure the control messages will be delivered.

As is shown in figure 4.7, each component contains the control logic (master and web client are complete control logic). Control logic represents message listeners. All control messages are asynchronous. Control messages comprised of messages used by master to control the behavior of node containers and nodes, messages between master and web client, performance metric reports sent from nodes to master.

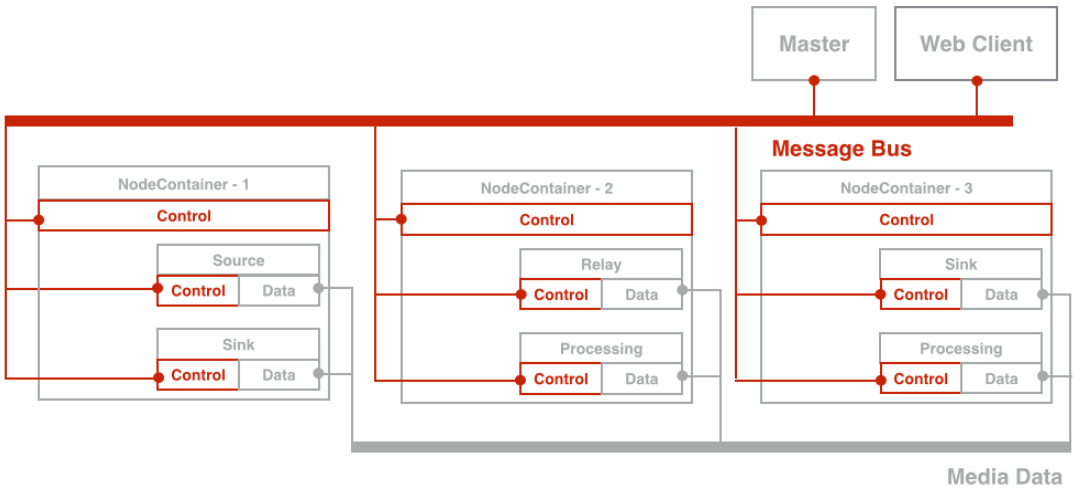


Figure 4.8 Message bus

4.4 User Interface Layer

User interface is the layer that handles the interaction with users. User interface layer is basically a web application. It is divided into two parts: input and output. The input takes the

users input in terms of the action users can take and the description of the scope to perform on the simulation. The output visualizes the topology of the media delivery network and provides the instant performance metrics to users.

4.4.1 Input Part

First, we define basic terms which are used in providing input and controlling simulation.

Work Specification

The input file given by the user that has a list of streams and stream parameters.

Stream

A unique media entity like a movie or audio clip that has a defined size and bitrate. A stream consists of a list of flows.

Flow

It is a unique path which a stream follows from source to sink node.

User can control simulation in three major ways:

1. Start a Flow
2. Stop a Flow
3. Reset entire simulation

Start a Flow: User will upload a work specification file which will contain a list of streams (each stream containing a list of flows) which he wants to start.

Stop a Flow: User will upload a work specification file of same format as above but as user has selected Stop a Flow action, the system will actually stop the flows mentioned in the file.

Reset: This will bring entire system to initial state and all the resources will be released.

Work Specification Format

The file is of JSON format. Root element is Simulation Id. It contains a list of Streams as child elements. Each Stream has following properties

Property	Sample Value	Purpose
StreamId	"stream123456"	Unique identifier of the stream. This is used to both start and stop flows.
DataSize	20000	Size of the stream in bytes. This much amount of data would be transferred from the Source Node.
KiloBitRate	25	Speed at which source node will transfer data for this stream
FlowList		List of flows under this stream

Each Flow is a collection of Nodes which forms a unique path of data flow starting from Sink Node to Source Node. Sample Flow is given below.

```
"NodeList":
            [
              {
                 "NodeType":"SinkNode",
                 "NodeId":"tomato:sink1",
                 "UpstreamId":"lemon:relay1"
              },
              {
                 "NodeType":"RelayNode",
                 "NodeId":"lemon:relay1",
                 "UpstreamId":"apple:proc1"
              },
             {
                "NodeType":"ProcessingNode",
                 "NodeId":"apple:proc1",
                 "UpstreamId":"orange:source1",
                 "ProcessingLoop":"100000",
                 "ProcessingMemory":"1000"
             },
             {
                 "NodeType":"SourceNode",
                 "NodeId":"orange:source1",
                 "UpstreamId":"NULL"
             }
        ]
```

Each Node currently has following properties.

Property	**Sample Value**	**Purpose**
NodeType	“SourceNode”	Indicates type of node - what kind of function will the node perform
NodeId	orange:source1	Node Id = NodeContainerId:NodeName This tells the system to create a particular node in XYZ node container.
UpStreamId	orange:source1	Indicates which node sends data to this node for a particular flow (and stream)
ProcessingLoop	10000	Indicates how much CPU should be utilized in Processing Node
ProcessingMemory	1000	Indicates how much memory should be occupied in Processing Node

To see more examples of the input file and how the resulting topology will look like, please refer tutorial document.

Also, note that user can send a REST requests to the start/stop flow API instead of uploading a file.

API Details

1. Start Flow
    - Purpose: To start a new flow or add new flow to existing simulation.
    - Method: POST
    - URL: WebClientURI/work_config
2. Stop Flow
    - Purpose: To stop existing flow.
    - Method: DELETE
    - URL: WebClientURI/work_config
3. Reset Simulation
    - Purpose: To reset the entire simulation.
    - Method: DELETE
    - URL: WebClientURI/simulations

4.4.2 Output Part

The output part visualizes the topology of overlay network based on current state of each nodes from data layer. Besides, monitored performance metrics are also presented to users.

Performance metrics are: transfer data rate (kbps), packet loss rate (%) and latency (ms).

The states of each node and each edge can also be seen. Performance metrics and states transitions are visualized both in graph and in log information. To see the metrics in graph, user can hover over the edge and the metrics will keep updating themselves.

The metrics are reported for each Edge in the network.

And different colors indicates different state of that edge.

Green - indicates that data flow is in progress

Black - indicates that flow has stopped

Grey - indicates that flow has not yet started

Red - indicates that there is packet loss levels in this edge is more than configurable threshold level

5. System implementation

5.1 Technologies Used

Programming languages: Java for management and data layer, Javascript for UI

Data exchange format: JSON

Communication technology: Ericsson Warp Message Bus for sending control messages between all the components of the system

Graph Visualization: Sigma.js library for generating graphs and combination of HTML5 SVG and jquery for rendering tooltips on nodes and edges

Tools: Eclipse as IDE, Maven as build and dependency management tool, FindBugs to find out some race conditions, git and github for source control

5.2 Process and Threads View

Mater runs in a separate process and Web Client runs in the browser.

Each Node Container runs as a separate process. For each stream passing through a node, a separate thread is created to handle the data for that stream. And another thread is created to handle reporting for that node and stream combination.

5.3 Reporting Module

Report thread sends data to the master node every second and master node aggregates all the reports using a WebClientGraph Singleton and updates the web client periodically. Brief explanation of report types is given below.

Packet Loss Report

Packet loss is calculated based on a sliding window. The window size is calculated by multiplying the timeout and expected packet number per second. When a packet comes, the program will fetch its packet id and compare the id with the current window. If the packet id is behind the current window, that packet is treated as a lost packet; If the packet id is in the current window, that packet is counted as a normally received packet; If the packet id is beyond the window, all the packets that with lower id than the current packet and have not come yet will be counted as lost packets, and the window will move forwards based on the current packet id. That means the current packet id will become the lower boundary of the window.

Two metrics will be reported, average packet lost ratio and instant packet lost ratio. The average packet lost ratio is the ratio of total lost number over total expected number. The instant packet lost ratio is the ratio of packet lost number in previous window over the total expected packet number in previous window.

Transfer Rate Report

Transfer rate is implemented by tracking the size of received data and the time. Two values is reported, average transfer rate and instant transfer rate. The average transfer rate is the ratio of total received data size over total time from the first packet arriving to the current time. The instant transfer rate is the ratio of total size in the recent period of time over the length of the period. The default value of the frequency is one second.

Latency Report

Latency is calculated only at Sink Nodes. It is average latency for all the packets received for that flow. Each node packet has a start timestamp within it (inserted by Source Node). Whenever a sink node receives a packet, it will take the difference of current time and start timestamp to calculate latency for that packet and update the average value. As other reports, this value will be sent only once every second.

CPU and Memory Usage Reports

These values will be reported by each machine in the deployment. It would be sent once every second to the master node which will show these values in web client in tabular format.

5.4 UML

The UML diagram showing all classes and packages used in project is available in design documents for reference.

5.5 Messaging Part

We are using REST APIs to send control messages between all the different components of the system. Each component is identified by a unique Message Bus URI. And each of them expose a resource tree which is used to handle different control messages. For each resource there is a method listener in that component. The messages are sent in JSON format.

5.6 Node Type Extensibility

Each Node Type derives from an Abstract Node which makes it easy to add a new node type in the system. Also Node Container instantiates new nodes using Java Reflection which allows creating nodes of new node types dynamic.

5.7 Node Packet Format

The packet format is shown in the following diagram. There are two parts of a packet, header and payload. The header is 12 bytes long. The first 4 bytes represent a flag; The next 4 bytes represent the message id; The last 4 bytes represent the length of the payload. The length of the header is fixed and the maximum length of the payload equals MAX_PACKET_LENGTH - HEADER_LENGTH;

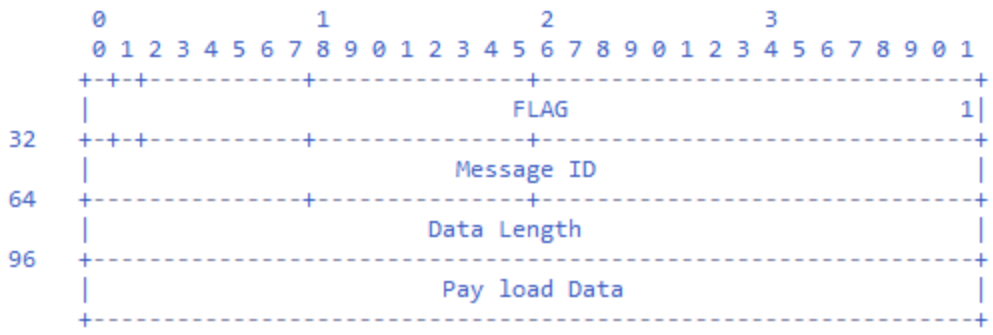


Figure 5.1 NodePacket

5.8 Start Flow

Whenever the user uploads a new work specification, Master node extracts all the flows from it. And for each flow, it sends the flow information to sink node. Sink node sets up resources required to start the flow like allocating new port. It updates the flow specification with its IP and port information. And then its sends the updated flow specification to its upstream node. This process continues until it reaches the Source Node. Once Source Node receives the specification, it will start sending data to its downstream and data flow has begun.

5.9 Stop Flow

To stop a flow, user uploads work specification with list of flows to stop. Master will send that information to Sink Node. To stop a flow, the task is divided into two phases: (1) stop receiving packets and (2) release resources. All the nodes in the flow need to finish phase 1 before the start of phase 2.

Upon arrival of stop control message, Sink Node does following steps:

1. Stop receiving further packet in the corresponding stream.
2. Stop the reporting thread of the corresponding stream.
3. Inform its upstream node to stop sending data.

The upstream node does the same thing if the node is a processing node.

If the upstream node is a Relay Node, the Relay Node needs to decide whether to receiving further packet. The criterion is that: after stop the flow, if there is at least one flow working, the Relay Node will not sending any packet to this flow, and send control message to its downstream node in the flow to release the socket resource. Otherwise, Relay Node does the same steps as Sink Node.

If the upstream node is a Source Node, the Source Node does step 1 - 2 of Sink Node. Besides, Source Node sends release resources control message to downstream node.

For the phase 2 (release resources), it starts either a relay node or source node. Each node closes sockets and forwards the releasing resources control message to its downstream node.

The reason to split stopping flow procedure into two phases is to avoid ICMP packets such as destination unreachable being transferred in the network if the downstream node closes the socket without informing its upstream node.

5.10 Reset logic

To reset the whole system, the master sends reset control message to all node containers. Node containers stop all running streams. Node container deregister all method listeners for each nodes and clear all nodes.

As for master, it clears all information regarding the nodes, streams, flows and graph to the webclient. It sets its state to the initial state except that it memorize all registered node containers and can communicate with each node container.

6. Future work

We will improve the system to increase the usability, maintainability and performance in the following aspects.

6.1 Node containers Auto-deployment

A script will be used to start node containers before launching the entire system. The user won't have to start the node containers manually. As a result, the deployment will become way easier in a automatical way even when the number of node containers is very large.

6.2 Maximum node number enhancement

Currently, one thread is assigned with one stream in one node. If there are multiple streams coming into a node, there will be multiple threads assigned to that node. So the maximum number of nodes is limited by the number of threads that a JVM can handle. In the future, one node will use minimum threads for processing, So the number of nodes the system can support will increase.

6.3 Dynamic marker insertion in data traffic

A marker in traffic packets will be applied to induce processing and encoding workloads on processing nodes. So processing node will generate workload based on what is gotten in packet instead of fixed parameters in work specification. Processing parameters such as the number of loop and the size of memory needed to process the particular packet. Different packets can have different resource requirement to be processed in this way.

6.4 Improvements in visualization part

A better way to visualize a large number of nodes will be used by providing some functions such as filtering the nodes. So users can choose to see nodes within a specific stream and other nodes won't be shown. Also, all the nodes within one node container can be put together as a cluster in UI. The users can only focus on their interested portion in the entire network topology. A zoom in and zoom out function will also be provide in the UI.

6.5 UI based Work Specification generator

We will provide an UI to fill out html a form to generate a work specification to avoid misspelling to do it manually. Users only need to input the number of nodes, define the topology of the network by specifying the upstream node id. There will be some default value in the form such as the required cpu usage and memory usage.

7. Things we tried and discarded

We made the initial design and implementation as the following diagram. Our initial design and implementation uses HTML5, JQuery and Sigma.js as front end technologies. We used Tomcat and Java Servlet for backend. There were three technologies that were used for communication within the system. Browser uses Ajax to talk with Tomcat Server; Tomcat server uses Java RMI to talk with the master; Master uses Rabbit MQ to talk with data layer.

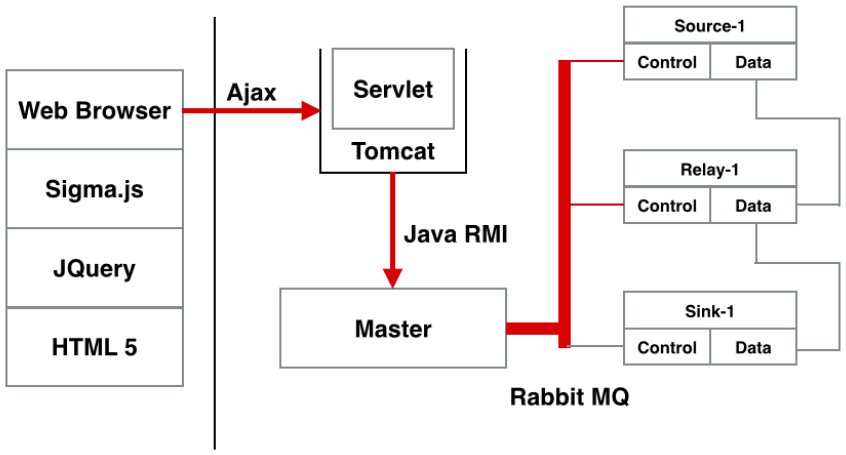


Figure 7.1 Initial system topology that was later discarded

We discarded it based on the following analysis. The major drawback of this design is multiple communication mechanisms are involved in the system and lead to a big complexity to debug and maintain it. Also, in the practical level, there were a lot of dependencies we need to rely on for running the system which is hard to maintain. It is also hard to define a unified interface for control messages that all components of the system can use for communication. Based on the above analysis, we decided to use a simpler approach for the communication in the system.

We searched for some open source projects for network simulation and found that they are not suitable to satisfy the clients requirements. The purpose of this simulator is to do some experiment on application layer. It has nothing to do with the layers below transportation layer such as IP layer. We found that existing network simulators emphasis a lot on simulating the lower layers.

Also, one of the requirement was that node functionalities should be easily extended by the users. And the fact that we wanted full control over entire system made building a new system more attractive option. Additionally, there was a steep learning curve associated with using and extending existing simulators esp. compared to the limited time available to complete the project.

8. Acknowledgements

We would like to thank Ericsson Research for sponsoring this project and providing us with this opportunity. We would like to specially thank Vlad, Jia and Alvin for guiding us through this project providing us with valuable feedback on design of the system.

9. Appendix

Github link to project - https://github.com/cmusv-sc/Practicum2014-Ericsson-Media

All the other documents are under docs folder in the root directory of project.

10. References

[1] http://www.ericsson.com/res/docs/2014/game-changers2-the-ip-imperative.pdf
[2] http://www.isi.edu/nsnam/ns/
[3] http://www.omnetpp.org/