Fall **2014**

Technical Report

Media Delivery Network Simulator

Sponsor	**Ericsson**
Point of Contact	Vladimir Katardjiev Alvin Jude
Faculty Advisor	Jia Zhang
Team	Jeremy Fu Jigar Patel Vinay Kumar Vaili Hao Wang

Version	Release Date	Revised By
0.1	Dec 6, 2014	Jeremy Fu, Jigar Patel, Vinay Kumar Vavili, Hao Wang
1.0	Dec 10, 2014	Jeremy Fu, Jigar Patel, Vinay Kumar Vavili

Table of Content

1. Introduction 3
2. Motivation 4
3. Related Work 6
4. System Design 8
5. System Implementation 15
6. Future Work 19
7. Things We Tried And Discarded 20
8. Acknowledgements 21
9. Appendix 21
10. References 21

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.

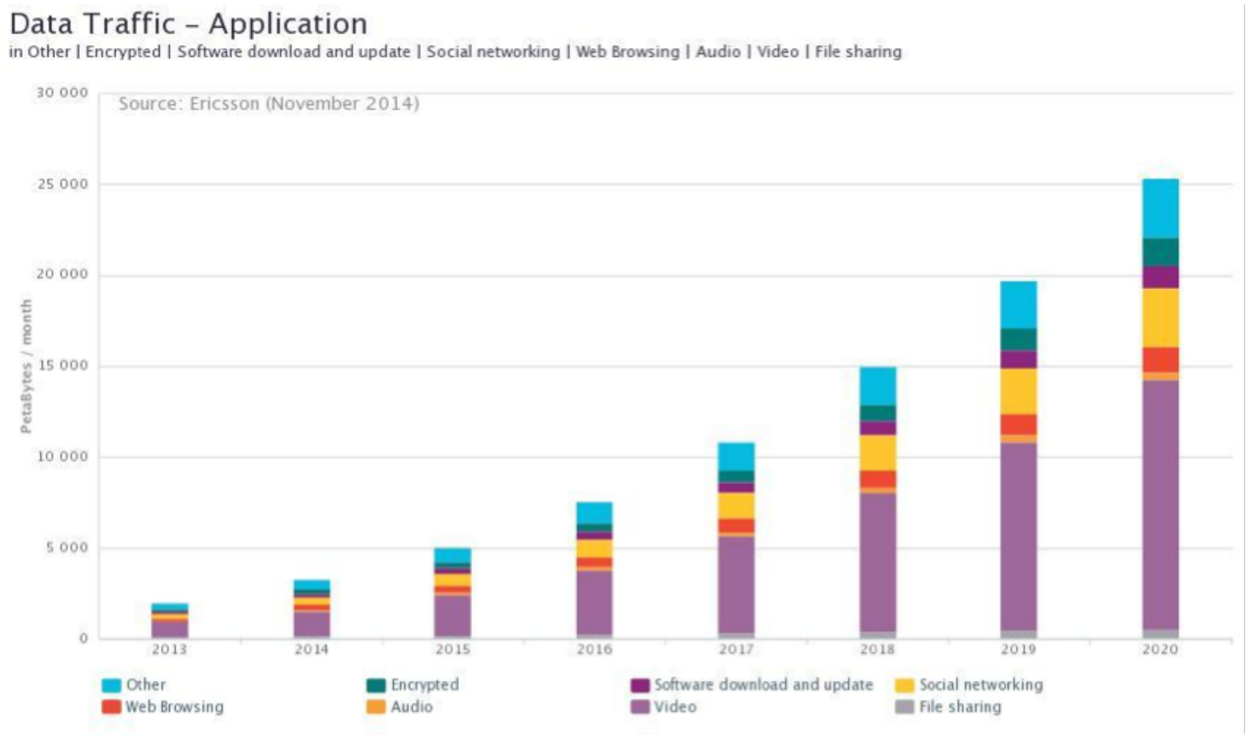

Figure 2.1 Data traffic prediction

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.

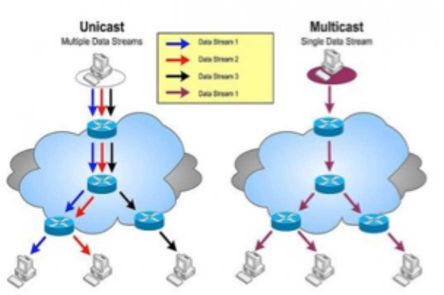

Figure 2.2 Unicast/multicast

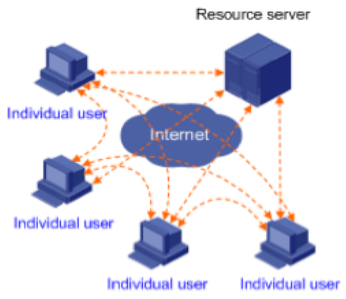

Figure 2.3 Peer to peer

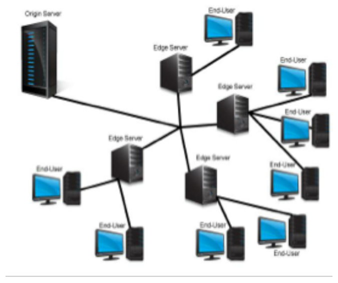

Figure 2.4 Cache based

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 in fact 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 system. The system follows master-slave architecture. Components running on different machines simulate an overlay network over the Internet.

The figure 4.1 illustrates how the MDN simulator can be divided by the functionality of different components. The data layer consists of component nodes i.e. source node, sink node, processing node and relay node. The Management Layer consists of message bus, node containers and master node which are used to control the system. The User Interface or the Web Client provides a user facing web interface that can be used to provide input to the simulator and visualize the state of the simulation.

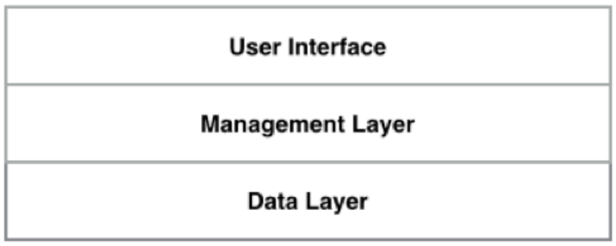

Figure 4.1 Layers of MDN Simulator

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. Four 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 user 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. As an example, Figure 4.2 shows a simple media delivery chain with one source node and two sink nodes. The component circled by red is the source node. The source node doesn't take any input and is the start of a media chain.

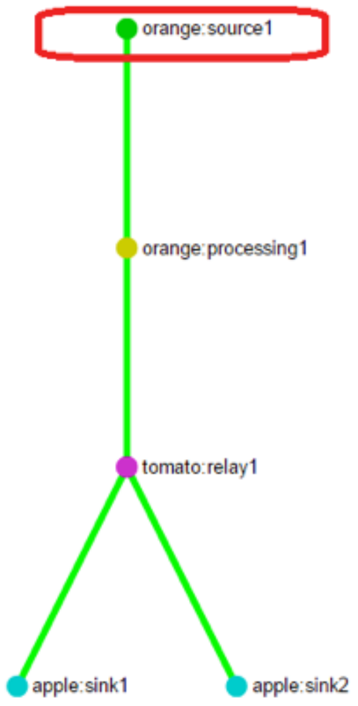

Figure 4.2 Source Node

Processing Node: Processing node simulates actual processing operations on media data such as encoding, ad-insertion and subtitle addition. As a simulator, MDN simulator doesn't perform real processing operations on data. Instead, it just simulates processing by consuming CPU (using a tight loop) and memory. The processing load is configurable as well. Users can add corresponding load on the processing node based on data collected from real-world processing work load.

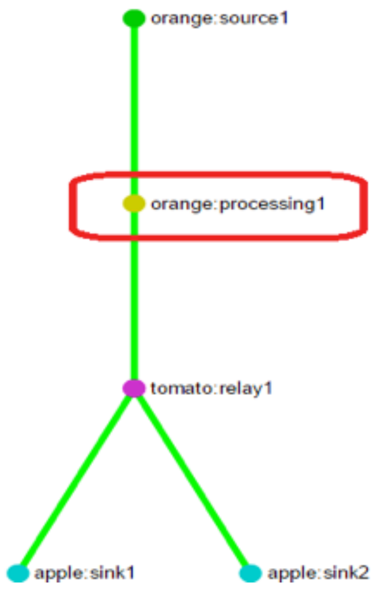

Figure 4.3 Processing Node

Processing node also monitors the packet loss rate. The circled component in figure 4.3 is the processing node. Processing node takes one input and has one output.

Relay Node: Relay node takes one input, duplicates data and multicasts it to multiple down-stream nodes. However, it is worthwhile to mention that this multicast doesn't mean IP layer multicast. Since the data is sent via UDP socket, it should be regarded as unicast in IP layer and multicast at application layer.

A down-stream node can be dynamically added to a relay node. The relay node automatically duplicates an additional copy of the data for that stream and starts sending it to the new downstream node from the point it was added. A downstream node can also be removed dynamically for that stream. Upon removal, the relay node continues to multicast data for a stream if there are active downstream nodes for that stream. If there are no downstream nodes receiving data for a particular stream, the relay node informs the upstream node to stop sending data for that stream.

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 node has two downstream nodes to which it multicasts data received from its upstream (apple:proc1).

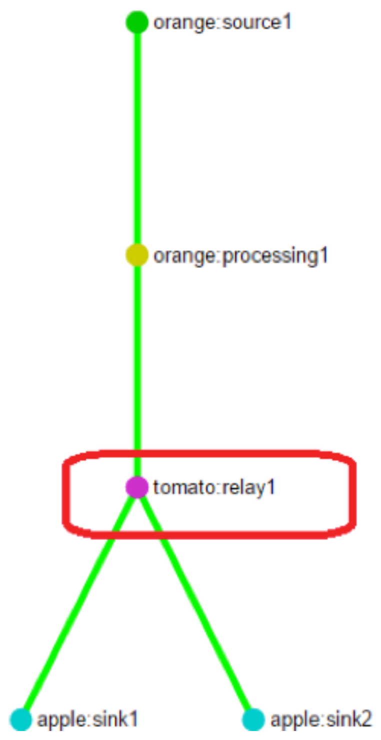

Figure 4.4 Relay Node

Sink Node: Sink node acts as the consumers of media data. The consumers can be audience/ subscribers in a real world scenario. 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 do not forward data to any other downstream nodes.

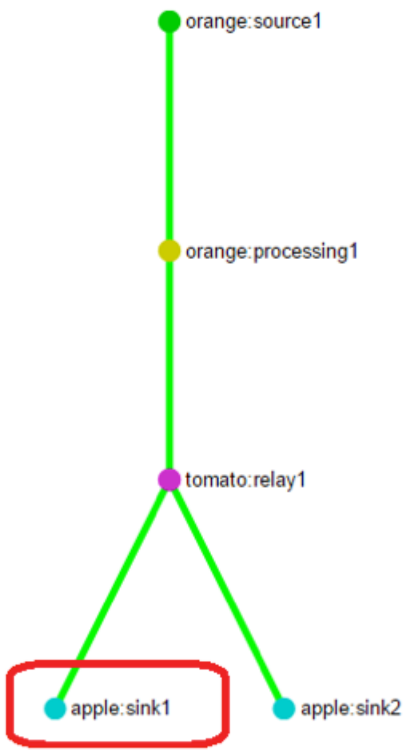

Figure 4.5 Sink Node

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 wraps up IP datagram 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 at the application layer 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.

Although the system uses 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. It is also in charge of collecting metrics from different components. Management layer consists of three components: master, node containers and message bus.

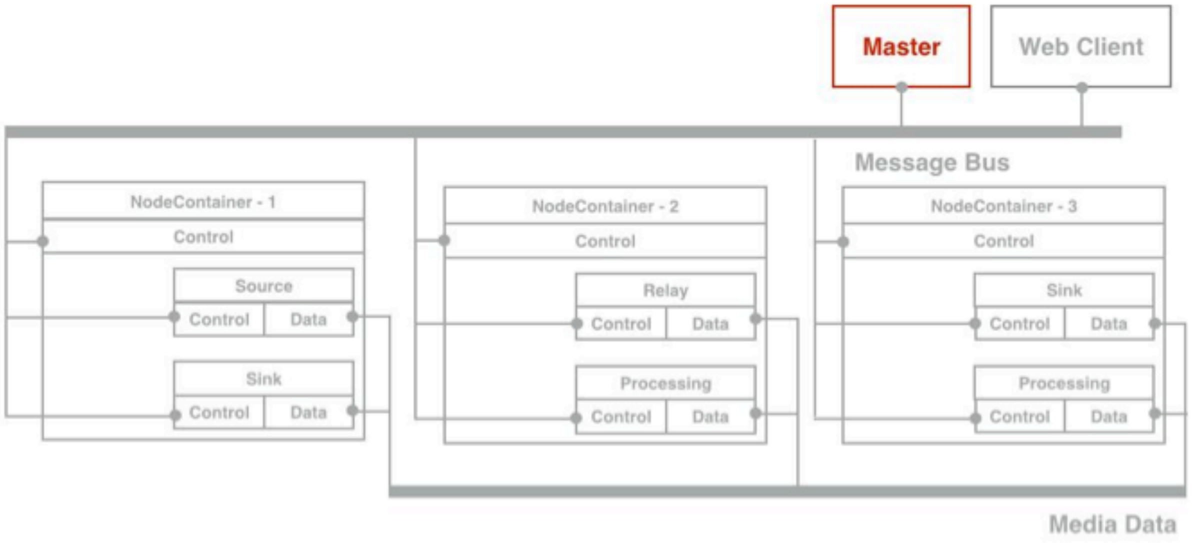

Figure 4.6 Master

Master

The master is the core of management layer. The Master node 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 input by user, such as start or stop a simulation, attach a new flow to a media chain and reset the whole system. Master periodically updates the web client with a view that reflects the status of the system.

Node container: Master configures the functionality of each node container by instantiating functional nodes at data layer. The master can get the state of the simulation using the information from the node containers and update the web client with the current state.

Node: Master deploys nodes on the node containers to form the media delivery network. Master contains the whole picture of the topology. A node only contains the knowledge of their upstream node and downstream nodes if applicable. Master also controls the behavior of each node directly, such as start or stop sending data to the data path.

Node Container

Node containers are different processes running on different machines. Node containers are denoted by the red part in figure 4.7. 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.

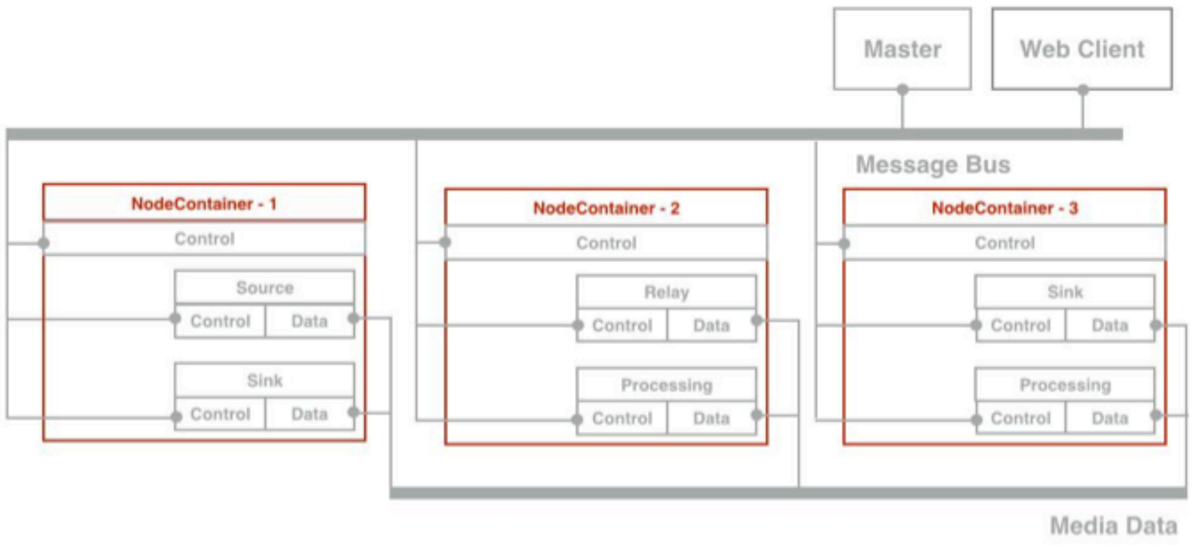

Figure 4.7 Node Container

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 guaranteed delivery of control messages.

As is shown in figure 4.8, each component contains the control logic. Control logic comprises of message listeners and senders. All control messages are asynchronous. Some of the Control messages in the system include,

- Start/Stop/Reset work configuration
- Create/Terminate nodes
- Register web client, node containers and node
- Report

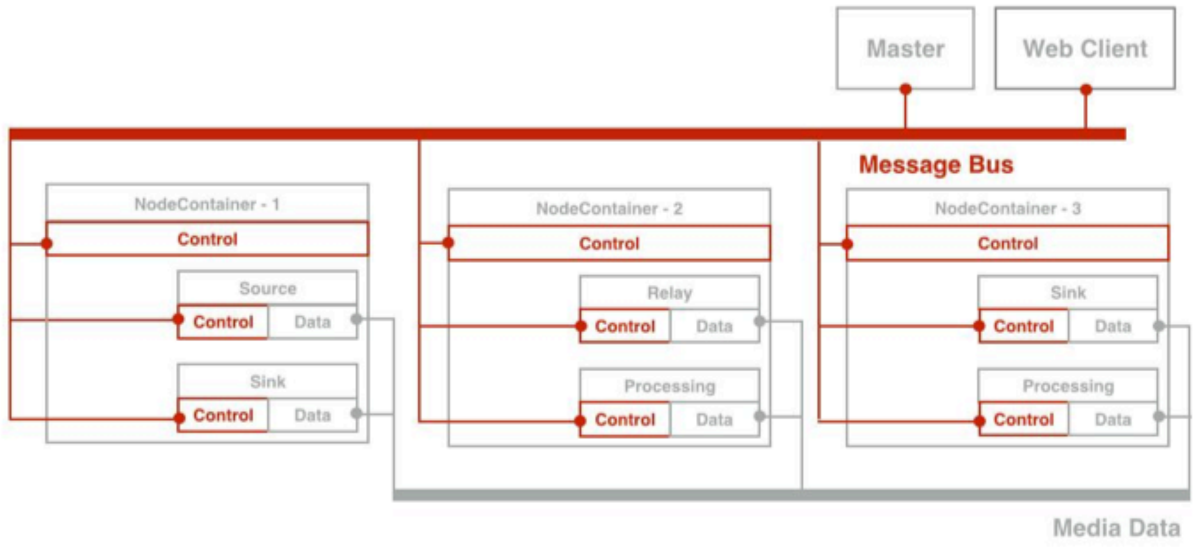

Figure 4.8 Message bus

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. User can control the simulation using the input section like starting a flow or stopping a flow. The output section helps in visualization of the topology of the media delivery network and provides instant performance metrics to users.

Please refer to the User Manual [4] for details on the Work Specification format and output visualization.

5. System Implementation

5.1 Technologies Used

Programming languages

- Management and Data layer: Java
- UI: Javascript

Control Message Format

- JSON

Message Bus

- Ericsson Warp Message Bus for sending control messages between all the components of the system

Graph Visualization

- Graph Rendering: Sigma.js, HTML5 and SVG renderer
- Tooltip rendering(for nodes and edges): jquery

Tools

- IDE: Eclipse
- Build and dependency management: Maven
- Static Analysis: FindBugs
- Source Control: Git

5.2 Process and Threads View

Master 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 object 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 with lower id than the current packet and which have not come yet will be counted as lost packets and the window will move forward based on the current packet id. That means the current packet id will become the lower boundary of the new 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 number of lost packets over total expected number of packets. The instant packet lost ratio is the ratio of packets lost in previous window over the total expected packets in previous window.

Transfer Rate Report

Transfer rate is implemented by tracking the size of received data and the time. Two values are reported, average transfer rate and instant transfer rate. The average transfer rate is the ratio of total received data size over total time (where time is calculated as difference between time of arrival of first packet and current time). The instant transfer rate is the ratio of total data size in the recent period of time over the length of the period. The default value of the period 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 [5] 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 exposes 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 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.

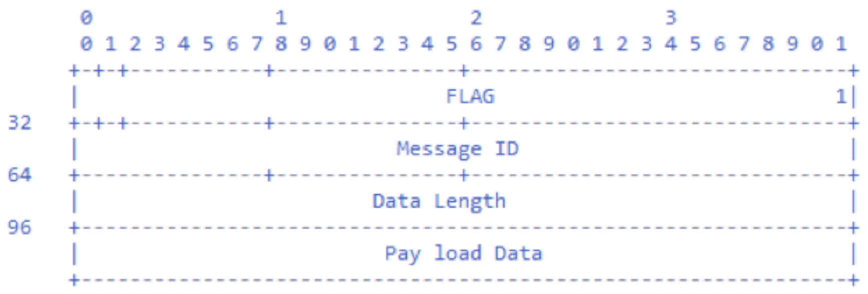

Figure 5.1 NodePacket

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 node which is already listening for data for the stream in the work specification. The forwarding ports are also setup in all downstream nodes. Hence, the data will reach the sink node once it starts from the source.

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 proceeding to 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.

In case the upstream node is a processing node, the same steps are followed. If the upstream node is a Relay Node, the Relay Node needs to decide whether to receive further packets for the stream being stopped from it's upstream. If there are other downstream nodes that are still receiving packets for the stream, the relay node continues to receive packets for the stream. If there are no further receivers, the relay node sends a message to its upstream node to stop sending data.

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

For the phase 2 (release resources, each node that receives the message, closes sockets associated with the stream and forwards the release resource control message to its downstream node.

The reason to split stopping flow procedure into two phases is to avoid ICMP unreachable errors in upstream nodes. This is done by closing the downstream sockets only after upstream stops sending data.

5.10 Reset Logic

To reset the entire system, the master sends reset control message to all node containers. Node containers stop all running streams and kill all node threads.

As for master, it clears all information regarding the nodes, streams, flows and web client graph. The user interface is also cleared. It sets its state to the initial state except for information on the node containers.

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 starting the simulation. The user won't have to start the node containers manually. As a result, the deployment will become way easier especially when the number of node containers is very large.

6.2 Maximum node number enhancement

Currently, one thread is created for every stream on the node. So the maximum number of nodes is limited by the number of threads that a JVM can handle. In the future, we can minimize number of threads per node and thereby system can support more nodes.

6.3 Dynamic marker insertion in data traffic

Currently packets passed by the simulator are quite dumb as they are not making much use of the header portion. But in future a marker in traffic packets can be applied to dynamically induce processing and encoding workloads on downstream nodes. So processing node will generate workload based on parameters in packet instead of fixed parameters in work specification. Different packets can have different processing requirement and hence overall simulation will be more dynamic.

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 of nodes. So users can choose to see nodes within a specific stream. Also, all the nodes within one node container can be put together as a cluster in UI. A zoom in and zoom out function will also be provided in the UI.

6.5 UI based Work Specification generator

We will provide an UI for automatically generating work specification input files. This will greatly reduce the pain of creating work specification files manually as they are not only time consuming but also error-prone.

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.

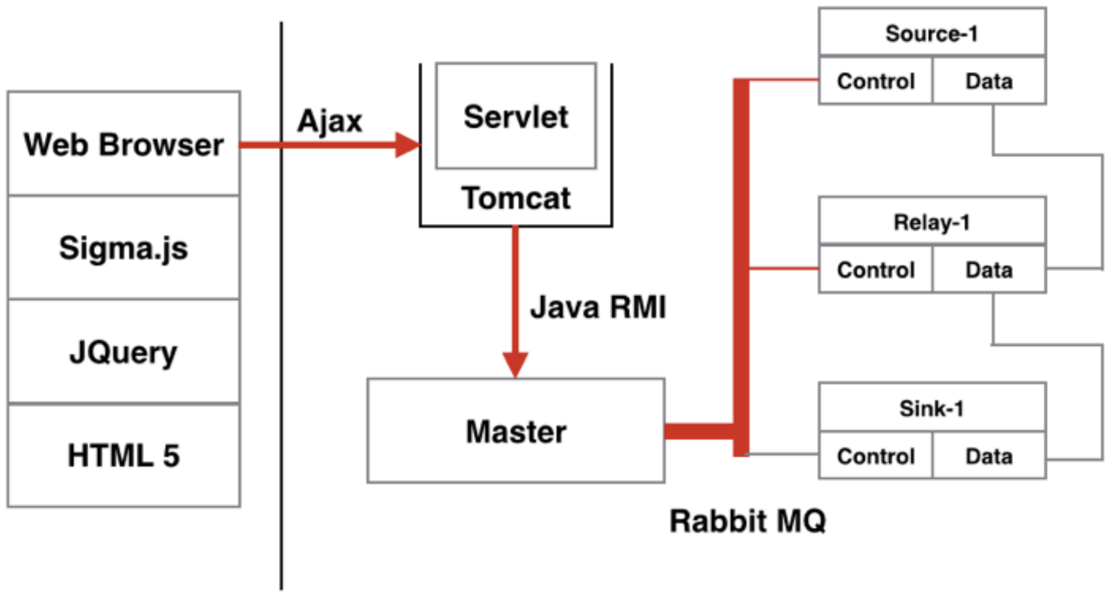

Figure 7.1 Initial system topology that was later

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 there were a lot of dependencies we needed 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. And hence used the Warp library for all types of communication as explained on the Design section earlier.

On another note, 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 of the OSI model.

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 Vladimir Katardjiev, Jia Zhang and Alvin Jude 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 available at following link:
https://github.com/cmusv-sc/Practicum2014-Ericsson-Media/tree/master/docs

10. References

1. Game Chamgers2 – The IP imperative: www.ericsson.com/res/docs/2014/game-changers2-the-ip-imperative.pdf
2. NS network simulatorhttp://www.isi.edu/nsnam/ns/
3. Omnet++ http://www.omnetpp.org/
4. User Manual, revision 1.0 https://github.com/cmusv-sc/Practicum2014-Ericsson-Media/tree/master/docs/UserManual.pdf
5. UML Design, revision 1.0 https://github.com/cmusv-sc/Practicum2014-Ericsson-Media/tree/master/docs/design/revision1.0/