Implementation of CacheCast in the ns-3 network simulator

Bekzhan Kassymbekov, Dag Henning Liodden Sørbø, Kanat Sarsekeyev, and Rizwan Ali Ahmed

***Abstract*—This paper will introduce and explain the implementation of the CacheCast system in the ns-3 network simulator. For script authors we present the supported API and explain how the module should be used in ns-3 simulations. For the developer we present the details of the implementation of the different parts of the module. Tests and evaluations has been performed to prove that the implementation follows the design of the CacheCast mechanism. The outcome of this work is an independent ns-3 module called *cachecast* which contains all functionality to create ns-3 simulations with the CacheCast technique.**

I. Introduction

THE CacheCast system is a new technique which removes redundant payload on Internet links. The system is currently implemented in the Linux operating system, the Click modular router and partially in the ns-2 network simulator. In order to test new techniques and protocols destined for the Internet, network simulators are often used. A new network simulator called ns-3 has been developed which focuses on more accurate modeling of the functionality of a modern network. The goal of this work is to implement the CacheCast system in the ns-3 network simulator.

The implementation will consist of a ns-3 module which contains the data structures and the algorithms for the CacheCast packet handling. The module will also contain example scripts and demos to make it easy the see how the module is used in ns-3 network simulations.

This document is structured in the following way. In section II we give a brief description of the CacheCast system and we follow up which an introduction to the ns-3 network simulator in section III. Then we present the API of the implementation in section IV. In section V we explain some general implementation datails, while in section VI, VII and VIII we dive into the specific details of the implementation. We perform an implementation evaluation in section IX. In section X we conclude the paper. We start now by getting to know the CacheCast system design.

II. The CacheCast system

CacheCast [1] is a new technique of removing redundant data packets on a link. The purpose of CacheCast is to remove as much as possible of the overhead when using many unicast connections for the same data over the same link. In short, CacheCast only sends the packet payload once over a link together with the destination addresses, and then the responsibility lies on the router on the link exit to forward the payload to each client. With this redundancy removal, CacheCast achieves close to multicast performance when the numbers of receivers grow large, without introducing any new protocols into the Internet. An overview of the CacheCast packet modification can be seen in figure 1.

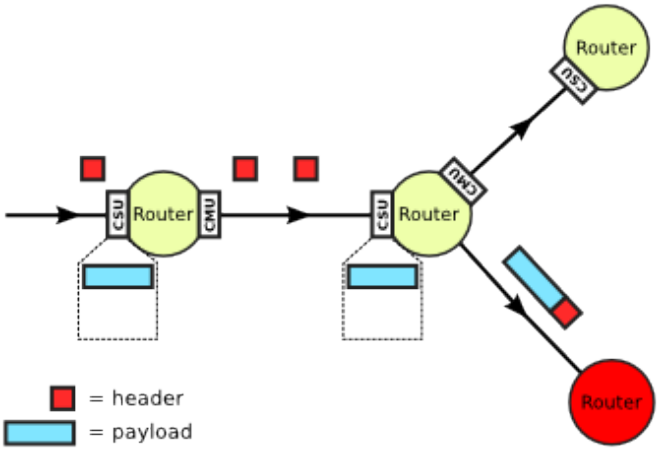

Figure 1. The packet modification in the CacheCast system

As the name implies, CacheCast uses caching to remove the redundant data packets from a link. Three main components are needed to make CacheCast work; support for CacheCast on the server, a component on the link entry called *Cache Management Unit (CMU)* and a component on the link exit called *Cache Store Unit (CSU)*. It is important to notice that these components can be divided into two groups; server support and network support. These two groups work independently of each other. Hence, there is no communication between the server and the components in the network. Each CacheCast packet contains a CacheCast header. It contains three fields; payload size (*P_SIZE*), payload id (*P_ID*) and cache index (*INDEX*). The details of these fields will be explained in the following sections.

A. Server support

The use of CacheCast demands that the server is aware of CacheCast support in the network, thus a server component is needed. Clients connected to the server requesting the same content is served via the CacheCast component. To be able to take advantage of CacheCast, data should be sent to many clients at the same time. This batching of clients is done by applications using the CacheCast support.

The responsibility of the server support is therefore to handle this client batching. Currently this handling is done through a system call (msend()) implemented in the Linux operating system. The input to this system call is a set of

connections to clients and the data to be sent. The system call will then add the CacheCast header to each packet and mark the packets as CacheCast packets. This is done to let the other CacheCast components separate CacheCast packets from regular packets.

At last the server sends the packets onto the link in a tight sequential order. However, only the first packet contains the payload whereas the other packets are truncated. Because of this removal of redundant payload the CacheCast packets follow a certain pattern. The first packet contains a header and the data, whereas the rest of the packets only contain the headers. This structure is called a packet train.

B. Network support

The network support is implemented on a per link basis and consist of two components; the CMU located at the entry of a link, and the CSU located at the exit of that same link. This way CacheCast support need not be deployed in the whole network. It can be deployed incrementally from the server.

Let us now see how this network support is realised through the CMU and the CSU.

1) CMU: The CMU's responsibility is to remove the redundant payload and manage the cache in the CSU. For each payload transfered over the link a unique payload ID (P_ID) is given the payload which identifies, together with the source address, the payload uniquely in the Internet. If the CMU receives a new CacheCast packet with a P_ID currently in the cache it removes the payload from the packet, adds a CacheCast header and sends the truncated packet onto to link. The index field in the CacheCast header identifies the payload in the cache in the CSU. If the CMU receives a CacheCast packet not currently in the cache it assigns a new P_ID to the payload and inserts this ID into a table. The index of this table entry corresponds to the slot in the cache in the CSU where the payload will be stored. The CMU then adds the CacheCast header to the packet and sends the whole packet onto the link.

2) CSU: The CSU is the component containing the actual cache. The data in the first packet of the packet train is stored in a slot in this cache specified by the CMU. The job of the CSU is to attach the correct payload to the packets containing only headers. The router will process each packet as normal IP packets.

III. The ns-3 network simulator

The ns-3 network simulator [2] is a discrete-event network simulator, in which the simulation core and models are implemented in C++. ns-3 is built as a library which may be statically or dynamically linked to a C++ main program that defines the simulation topology and starts the simulator. ns-3 also exports nearly all of its API to Python, allowing Python programs to import an ns-3 module in much the same way as the ns-3 library is linked by executables in C++.

ns-3 simulator is an open-source project, intended for educational and scientific use. Main purpose for usage of this tool is to simulate different routing protocols, scenarios that occur in different network. These scenarios and designs of network are implemented by user. ns-2 is a version prior to ns-3, which is a new simulator that does not support the ns-2 APIs. ns-3 supports popular network protocols used nowadays. Some models from ns-2 have already been ported from ns-2 to ns-3, and some has not yet been implemented in ns-3. Version of ns-3 we are using during our implementation and simulation is ns-3.13.

Abstractions

Implementation in ns-3 is based on abstraction, so it can be implemented in a simple way. These abstraction have differences from network as we know in reality. Before we start explaining our implementation of CacheCast in ns-3, it is important to understand abstractions and terms that are used in ns-3.

3) Node: In ns-3 devices like host and end-user are represented as abstractions of nodes by the class `Node`. This class provides management of how these devices will be represented in simulations.

4) Application: Application is abstractions of user programs that are suppose to produce activity on nodes to be simulated. This is represented by the class `Application`, which provides methods for managing the representations of applications in simulations.

5) Channel: In ns-3, each Node are connected by a connection that are represented by a communication channel. This abstraction is represented by class `Channel`, which provides methods for managing communication subnetwork objects and connecting nodes to them.

6) NetDevice: Net device abstraction covers both the software driver and the simulated hardware. A net device is more or less "installed" in a Node in order to enable the Node to communicate with other Nodes in the simulation via Channels. Node may be connected to more than one Channel via multiple NetDevices. This abstraction is represented by class `NetDevice` which provides management of connections to Node and Channel.

7) Topology Helpers: In a simulated network, connections between Nodes, NetDevices and Channels needs to be arranged. NetDevices needs to be attached on Nodes, Node protocol-stack needs to be configured and a communication channels need to be defined between NetDevices and so on. Topology helpers are used to arrange these attachments as easy as possible on a large scale.

IV. Application programming interface (API)

We have now talked about the general design of CacheCast and about the details of the ns-3 network simulator. In this section we will explain how the CacheCast system is used by script authors to build ns-3 simulations.

A. Services provided to applications

The CacheCast system are built to deliver the same content to many receivers. In modern networks end-to-end connections are handled by different sockets, so CacheCast requires a sequence of send-requests (corresponding to each socket) to transmit packets to all clients. A multiple-send-request

from application layer requires a container of sockets, so the application can send packets to all sockets in the container.

Since such a socket container is not supported by transport layer and is not a part of a application-layer, we chose to implement a CacheCast Application Programming Interface in the class `CacheCast`. This interface will not only support the socket container, but also the function `Msend()` would be provided to the application.

The socket container itself is implemented as a vector of sockets (`m_sockets`), which contains sockets added by application. In addition to vector `m_sockets` we chose to have another vector `m_failed` to provide the application the knowledge of sockets that failed during the Msend() procedure.

Listing 1. Services provided to application

```
void AddSocket(Ptr<Socket> socket);
void RemoveSocket(Ptr<Socket> socket);

void Merge(CacheCast cc);

Iterator Begin (void) const;
Iterator End (void) const;

Iterator BeginFailedSockets (void) const;
Iterator EndFailedSockets (void) const;

bool Msend(Ptr<Packet> packet);
```

In order to maintain the container of sockets, some facilities are provided to applications. It is necessary for the application to add or remove sockets in the container. And the possibility to merge two or more socket containers might also become handy. It is described in listing 1.

B. Usage of CacheCast API

In ns-3, applications creates and binds sockets and CacheCast API is responsible for maintaining these sockets. When a socket is bound, it is added in the socket container simply by using function AddSocket().

A packet is sent to all sockets in the socket container by using the function Msend(). There is no possibility for applications to have a complete overview of which one of socket did failed until the Msend() function is completed. We have implemented this function to return a bool depending on if packet was delivered successfully to all sockets in the container. If a socket fails, Msend() places that socket in a vector for failed sockets and returns false. The application would then know there is a socket that failed during sending procedure. An iterator is provided to the application, so it can iterate through the vector containing failed sockets. And it is entirely up to the application how failed socket should be treated. Implementation of Msend() is listed in listing 4 later in this document..

Listing 2. Example of usage of CacheCast API

```
// assuming socket1 and socket2 is set up and connected
CacheCast cc;
cc.AddSocket (socket1);
cc.AddSocket (socket2);

Ptr<Packet> packet = Create<Packet> (1400);
if(!cc.Msend(packet)){
  CacheCast::Iterator vItr = cc.BeginFailedSockets();
  while ( vItr != cc.EndFailedSockets() )
  {
    cc.RemoveSocket( (*vItr) );
    vItr++;
  }
}
```

C. Helpers

The CacheCast helper classes are used to setup the Server Unit, CMU and CSU on nodes and a channel between the nodes. In both helpers, the Install() function takes pointer for two nodes as arguments. Two helpers are implemented namely CacheCastServerHelper ad CacheCastHelper. The only difference between these two helpers is that CacheCastServer-Helper installs server support in the first node argument. CacheCastHelper installs a CMU on the first node and a CSU on the second node. While defining the server node, a CacheCastPid object is aggregated to the object, so it has ability to synchronize payloads between many instances of the CacheCast container on the server. CacheCastNetDevices are created for each of these nodes. These devices are then attached to a CacheCastChannel before they are added into a container of NetDevices.

V. General implementation details

In the previous section we looked at the implementation of CacheCast in ns-3 from the user perspective. In the following sections we will dig deeper into the implementation details and look at the implementation from a developer perspective. In this section we explain how the CacheCast packets traverse the nodes and the channels in order to get a general overview of how the implementation works, and we look at common data structures used throughout the implementation. In the next two sections we elaborate on the specific implementation details for the server support and the network support.

A. General overview

The implementation of CacheCast in ns-3 (as in the general design of CacheCast) consists of three main parts; server support, CMU and CSU. As explained in the previous section, to support CacheCast, packets are sent with the CacheCast::Msend() function in the ns-3 applications on the server node. Then the packets traverse the network layers and is intercepted by the CacheCastNetDevice. In this CacheCast-NetDevice a CacheCastServerUnit is installed which adds a CacheCastHeader to the packets and truncates packets with redundant payload. The packets then traverse the channel and is received by a CacheCastNetDevice on the other end. In this CacheCastNetDevice a CacheStoreUnit is installed which adds payload to the truncated packets and remove the CacheCastHeader. The packets are then handled as normal IP packets in the node. If the packet is destined for another node, it is again intercepted by a CacheCastNetDevice in which a CacheManagementUnit is installed. This unit adds the CacheCastHeader to the packet and truncates packets

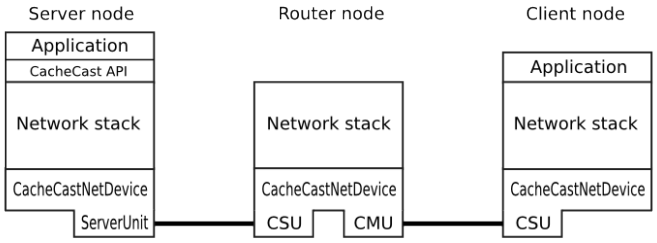

Figure 2. The structure of the CacheCast system in ns-3

with redundant payload. The packets then traverse another channel and is received by another CacheCastNetDevice with a CacheStoreUnit installed. This process is continued for each node supporting CacheCast on the packet's path to its destination. An graphical overview of this structure can be seen in figure 2.

B. Common data structures and classes

In this section we explain the common data structures used by the different parts of the ns-3 implementation of CacheCast.

1) CacheCast header: The CacheCast header is represented in ns-3 as a class named CacheCastHeader. This class is derived from the general ns3::Header class. The class contains the same members as the header described in the design of CacheCast, namely payload ID, payload size and index. An overview of the CacheCastHeader is given in listing 3 below.

Listing 3. The CacheCastHeader class

```
#include "ns3/header.h"

namespace ns3 {
class CacheCastHeader : public Header
{
public:
  CacheCastHeader ();
  CacheCastHeader (uint32_t payloadId,
    uint16_t payloadSize, uint32_t index);
  uint32_t GetPayloadId (void) const;
  uint16_t GetPayloadSize (void) const;
  uint32_t GetIndex (void) const;
  void SetPayloadId (uint32_t payloadId);
  void SetPayloadSize (uint16_t payloadSize);
  void SetIndex (uint32_t index);

  static TypeId GetTypeId (void);
  TypeId GetInstanceTypeId (void) const;
  void Print (std::ostream &os) const;
  uint32_t GetSerializedSize (void) const;
  void Serialize (Buffer::Iterator start) const;
  uint32_t Deserialize (Buffer::Iterator start);
private:
  uint32_t m_payloadId;
  uint16_t m_payloadSize;
  uint32_t m_index;
};
}
```

2) CacheCast packet tag: In ns-3 one have the possibility to add packet tags to packets in order to store information on a packet level. In the implementation of CacheCast we use packet tags to store CacheCast related values when the packet is processed in a node. Since the packets should be processed on the nodes as normal IP packets, the CacheCast header has to be removed, so this is the main reason for introducing a CacheCastTag. The contents of the CacheCast tag is the payload ID and the payload size. These are necessary for the CMU and server unit in order to uniquely identify CacheCast packets and handle them correctly. In an implementation in a normal router, metadata attributes is added to the CacheCast packets while they are being processed in the router. The CacheCastTag simulates this metadata but the CacheCastTag is not removed from the packet when the packets has been processed by a router node (as the metadata would in a normal router). There is no problem with this design choice since packet tags do not affect how the packets it being transmitted on a link, and because the payload ID and payload size is the same in the whole lifetime of a CacheCast packet. The CacheCastTag is added to the packets by the CacheCast::Msend() function (see section VI-A).

3) CacheCastUnit: The CacheCastUnit class is an abstract base class which should be derived from in order to support different packet handling schemes in CacheCast. This class contains only one CacheCast related function named virtual bool HandlePacket (Ptr<Packet> p). This function takes a pointer to a packet which it can modify. The derived classes is supposed to override this function. There are currently three derived classes from the CacheCastUnit base class; CacheCastServerUnit, CacheManagementUnit and CacheStoreUnit. These classed will be explained later in this document.

4) CacheCastNetDevice: In the ns-3 network simulator the abstraction of the physical and the data link layer is modeled by a NetDevice. This NetDevice receives a packet from the network layer, adds link layer headers, and puts the packet onto the channel. In order to handle CacheCast packets on the link level we have chosen to create a new NetDevice called CacheCastNetDevice. This NetDevice is based on the PointToPointNetDevice. In the first attempt of creating this NetDevice we tried to make CacheCastNetDevice a derived class of the PointToPointNetDevice. This proved difficult due to the design of the PointToPointNetDevice. There was no way to get to the packet after the transmission queue and before it was transmitted onto the link, which is crucial for the CacheCast technique. Because of this we chose to copy much of the code from PointToPointNetDevice and adapt it to the CacheCast scenario. Since CacheCast is only supported on point-to-point links the CacheCastNetDevice need not be generalized into arbitrary link level technologies.

The CacheCastNetDevice is used both on the server and on nodes and it is used together with the ServerUnit, the CMU and the CSU. To support these different scenarios we have added a senderUnit and a receiverUnit object to the class. These objects are derived from the CacheCastUnit class so their purpose is to modify packets. The main idea of CacheCast is to intercept the packets before they are transmitted onto the link to remove redundant payload. In our implementation this interception is done in the CacheCastNetDevice::TransmitStart() function. The packets need to be intercepted also on the receiver side, and this is done in CacheCastNetDevice::Receive(). In these functions the HandlePacket() function of the senderUnit and receiverUnit is called, respectively. So this is where the actual CacheCast

packet modification happens. More will be said in the following sections about how this packet modification is done.

In order to have a channel to connect to the new CacheCastNetDevice we had to create an adapted version of the PointToPointChannel named CacheCastChannel. The reason for this class is to support the CacheCastNetDevice. In ns-3 the design of the NetDevice and the Channel is closely related. So certain types of Channels can only be used with certain types of NetDevices. This was the only reason why we had to create a separate CacheCastChannel. No CacheCast related packet handling is done in the CacheCastChannel.

Now we have looked into the common data structures used in our implementation. Let us in the following sections go more into detail of the actual packet handling mechanisms of the CacheCast system.

VI. Server support implementation

The CacheCast system relies on support from the server in order to remove redundant payload in the network. This server support should send all packets sequentially onto the link and truncate all packets beside the first one.

The general design of the CacheCast server support consists mainly of two parts; the programming interface to the applications, explained in section IV, and the underlying packet handling mechanism. The implementation in ns-3 closely resembles this division. The implementation details of the API exposed to the applications is discussed in the next section while the underlying handling of packets is discussed in section VI-B.

A. Details of the API implementation

CacheCast::Msend() function takes pointer to a packet as an argument which is generated by the application. This functions returns a boolean value depends on if packet delivery to all sockets in socket container were successful or not.

Before sending packet to socket, payload-id must be generated and added to the packet as a CacheCastTag, which is done along with the size of the packet. CacheCastPid::CalculateNewPayloadId() is called to generate the payload-id. Usually this id is increased if packets are sent continuously. If more than 1 second has passed since the last packet was sent, the payload-id is wrapped around to 0.

Each socket in socket container is tested if it is a socket that CacheCast supports, since in CacheCast UDP and DCCP are only supported protocols in the transport layer. (DCCP is not implemented in our solution, since it is not yet supported in ns-3.)

Listing 4. Function Msend()

```
bool CacheCast::Msend (Ptr<Packet> packet) {
  bool successful = true;
  std::vector<Ptr <Socket> >::iterator socket;

  if (m_sockets.size() == 0)
    return true;

  Ptr<CacheCastPid> pid = m_sockets[0]->GetNode ()
  ->GetObject<CacheCastPid> ();
  NS_ASSERT_MSG (pid,
    "CacheCast server requires CacheCastPid");
  uint32_t payloadId = pid->CalculateNewPayloadId ();

  for(socket = m_sockets.begin();
      socket != m_sockets.end(); ++socket) {
    NS_ASSERT_MSG ((*socket)->GetSocketType ()
      == Socket::NS3_SOCK_DGRAM,
      "CacheCast supports only UDP sockets");

    Ptr<Packet> p = packet->Copy ();
    CacheCastTag tag (payloadId, p->GetSize ());
    p->AddPacketTag (tag);

    if((*socket)->Send(p) < 0) {
      successful = false;
      SetFailedSocket (socket_index);
    }
  }
  return successful;
}
```

B. Underlying packet handling mechanism

The tasks of the underlying packet handling mechanism in a CacheCast supported server is to ensure that the packets are put onto the link in a tight chain, to truncate packets with redundant payload and to add the CacheCast header to each packet. In the Linux implementation of CacheCast this mechanism is handled by a kernel module located between the network layer and the link layer. As previously explained we have chosen to create a new CacheCastNetDevice in which we can install different packet handling mechanisms. This is where the CacheCast packet modification will happen.

On the server an object of a class CacheCastServerUnit is added as a senderUnit to each CacheCastNetDevice. The CacheCastServerUnit::HandlePacket() function does the actual packet modification and its content is listed in listing 5 below.

Listing 5. HandlePacket() function in CacheCastServerUnit

```
bool CacheCastServerUnit::HandlePacket(Ptr<Packet> p)
{
  CacheCastTag tag;
  bool hasTag = p->RemovePacketTag (tag);
  NS_ASSERT_MSG (hasTag, "No CacheCast packet tag");

  CacheCastHeader cch (tag.GetPayloadId (),
    tag.GetPayloadSize (), 0);

  /* Invalidate the current payload ID
     after one second */
  if (Simulator::Now ().GetSeconds () - m_timeStamp
    > 1.0) {
    NS_LOG_DEBUG ("CacheCast server table
        invalidated");
    m_invalid = true;
    m_timeStamp = Simulator::Now ().GetSeconds ();
  }

  if (m_payloadId == tag.GetPayloadId ()
      && !m_invalid) {
    // remove payload
    p->RemoveAtEnd (tag.GetPayloadSize ());
    cch.SetPayloadSize (0);
  } else {
    // new payload ID
    m_payloadId = tag.GetPayloadId ();
    m_invalid = false;
  }

  p->AddHeader (cch);
  return true;
}
```

In the code of CacheCastServerUnit::HandlePacket() we first check if the payload has been in the table for more than 1 second, and if it has we invalidate this payload ID from the table. The reason for doing this is to support wrapping of payload IDs. The design of CacheCast specifies that a specific payload ID should be invalidated if it has been present in a table for 1 second or more. The rest of the code does the necessary modifications to the packet. If the packet's payload ID is not present in the table it is added and no further changes are done to the packet payload. If a payload ID is present in the table the payload is removed from the packet and the payload size field in the CacheCastHeader is set to 0. At last the CacheCastHeader is added to the packet. The HandlePacket() function returns control back to the CacheCastNetDevice::TransmitStart() function which continues to transmit the packet onto the channel. The transmission time on the channel is calculated based on the size of the modified packet.

The design of CacheCast demands that the packets forming a packet train is put in a tight sequential order on the link. In order to obtain a continuous packet train all packets with the same payload should be transmitted in one batch onto the link. In the implementation of CacheCast in Linux this is handled by a separate packet queue in the CacheCast kernel module to overcome the issue of the multiprogramming nature of modern operating systems. In ns-3 all code is executed in sequential order with no interruption. Also computing time on the nodes is not modeled in ns-3. Thus in our implementation we do not need to batch the packets at link layer level. The batching of sockets done in the application domain is sufficient enough to form continuous packet trains.

The fact that packets with the same payload ID are sent as a batch onto the channel also imply that the table of payload IDs on the server need only contain one element. In the code we don't even have a table, the payload ID is stored as a single variable. This fact also implies that the CSU on the other side of the channel from the server, need only have one slot in its cache. Therefore we always use an index of 0 in the CacheCastHeader appended to the packets.

We have now described the server support implementation of CacheCast in ns-3. In the next section we take a look at how the network support (CMU and CSU) is implemented. Due to time issues and problems regarding testing there are two implementations of the network support presented in this section.

VII. Network implementation (Lancaster version)

A. Cache management Unit (CMU) design

The Cache Management Unit is the part of the CacheCast system which is installed at the entry of the link. The design consists of various components which are as follows; Cache Management Unit table, CMU table Configuration, CMU Hash key Generator Function, Cache Hit Event, Cache Miss Event and Searching component.

These components are described in this document the way they have been designed and why they are used. The way the CMU works is explained in section II-B1.

1) Cache Management Unit table: The Cache Management Unit table is storage where the payload ids and the IPs associated with the cacheable packets are stored. These entries are used for determining that the packet entered in the CMU is to be handled as a Cache Miss event or as the Cache Hit event and the packet is handled accordingly. If the Cache Miss event occurs then the CMU table is updated and a new entry for this new packet is determined by the CMU Hash key Generator Function Component, and finally the associated payload id and IP of its source are stored in the CMU table.

2) CMU Table Configuration: The CMU table is configured only once when the CMU is installed. The size of CMU table is defined and it is maintained throughout the life of the CMU.

3) CMU Hash key Generator Function: The CMU Hash key Generator Function is used to get an appropriate index where the payload id plus the source IP of the associated packet is to be stored. It uses both the payload id and source IP as keys to be hashed. When the table gets fully loaded it starts expiring the oldest entries that were stored before and replaces them with the new entries, with this component the invalidation of the table entries is done automatically. This component is designed in a way such that it handles invalidation automatically.

4) CMU Cache Hit Event: The Cache Hit event occurs when the cacheable packet arrives at CMU and the associated payload id of this packet is already present in the CMU table. If we go into the semantics of the whole implementation, from source and payload id CMU analyses through its searching component that this packet was already stored in the CMU table or not. The occurrence of CMU hit event is verified by the CMU searching component so this means this is used by the searching component of CMU. Having this event occurred, the CMU places the appropriate index where it is stored in CMU table in the content header and this is then placed on the link to travel to the CSU.

5) Cache Miss Event: The Cache Miss Event occurs when the cacheable packet that arrives at CMU and the associated payload id of this packet is not previously stored in CMU Table. The CMU, after analyzing that this was a cache miss, first uses Hash key generator Component to generate an appropriate index where it places the associated payload id and source IP in its table. And then the CMU does not cut off the payload part of the packet and places the index generated in its content header and sends the packet without cutting the payload part to the CSU which then places the payload of this packet first in its table and then sends this packet to the router for further processing.

6) CMU Searching component: The CMU Searching component is the core component of the whole CMU which uses the cacheable packet's payload id and source IP to search through the table and this component verifies whether a Cache Hit event occurred or a Cache Miss event occurred. These are the only core events that convey CMU the current state and on the information they provide to CMU, it handles the packet accordingly as explained before.

B. Cache Store Unit (CSU) Design

The Cache Store Unit is the part of the CacheCast system where the payload of the cacheable packet is stored. It comprises of the following components; Cache Store Unit Table, CSU Table Configuration and Payload placement component. These components are described in this document the way they have been designed and why they are used. The way the CSU works is described in section II-B2.

1) Cache Store Unit Table: The Cache Store Unit table is storage where the payloads associated with the cacheable packets are stored. The entries where the payloads are stored are selected from the values which were placed previously by CMU in the content header's index field, the payloads are stored there and used accordingly when there is a cache miss this means that the packets payload has arrived at CSU as well and it has to be stored in the appropriate index. If there is a cache hit at CMU then the payload already placed at the index pointed by the INDEX field of the content header has to be attached to the packet to send it to the router for further processing.

2) CSU table Configuration: The CSU table is configured only once when the CSU is installed, The size of CSU table is defined and it is maintained throughout the life of CSU. And its size is equal to the size of the CMU table.

3) CSU Payload Placement Component: The CSU Payload Placement Component is the core part of the CSU which analyzes the packets that arrive at the CSU and based on its determination CSU further decides that what to do with this packet weather to just copy the payload from the table entry and place it with the packet or copy the payload from the original packet and place it inside the table. All this are handled by this component of the CSU.

Next we would like to explain the implementation of these components by naming the functions that we have in the actual components and what they do.

C. Cache Management unit element

The CMU element works on the base of the CMU table. The table is made up of a structure which is as in listing 6.

Listing 6. struct bucket

```
struct bucket {
  uint32_t payloadID;
  uint32_t  IpAddr;
  bool valid;
};
```

There is an array of this structure type which we call the CMU table. The usage of the above member variables will be explained in the functions. Beside this table array there are the following member variable in the class; CacheCastTag tag_obj and uint32_t m_size.

These following are the functions of the component.

1) void configureTable (): This function is used to configure the table, make all valid member variables false and assigning the size to the table.

2) uint32_t GenerateHashKey(uint32_t payloadID, uint32_t Ipaddr): This is a hashing function which uses payload id and Ipaddr both as key values to Hash the index where these keys are stored.

3) bool HandlePacket (Ptr <Packet> p): The core function of CMU which almost handles all of the functionality that is required by it. It removes the header of the packet and adds this header to its local CacheCastHeader object ccHrd. Then it accesses the source IP. Then it uses this IP and payload id in the searching function to know that whether its Cache hit or Cache Miss.

Cache Hit: If this event appears then this means that the payload is already present at the CSU, it simply places the Index in the ccHrd field.

Cache Miss: If this event occurs this means that this packet is a new one and payload id has to be stored in the appropriate tables. Then simply the CacheCastHeader is added and this way the whole packet is handled.

4) void Setsize (uint32_t size): This function is called the configureTable() when it has to set the table size.

D. Cache Store Unit Element

The Cache Store Unit works on the bases of a CSU table. The CSU table is made of the structure in listing 7 below.

Listing 7. struct bucket

```
struct bucket {
  uint32_t payloadSize;
  bool valid;
  bucket *next;
};
```

The member variables used in this structure can be seen as above. There are a few other member variables of these modules which are as follows; uint32_t m_size, bucket *table and CacheCastTag *tag_obj.

The core function of this element is bool HandlePacket (Ptr <Packet> p). This works as follows; This first removes the content header from the packet it receives, then extracts the index placed in it.

E. Network Wide Redundancy elimination

The network wide redundancy is eliminated obviously because the packets with same payload ids and which are from the same source and have travelled from some path before, only next time their content headers travel and arriving at the CMU and CSU which gives the full implementation of CacheCast system they simply make thing much more efficient and through small caching at the CMU which only stores the payload ids and the IPs of the corresponding cacheable packets, further transmits only the content header if there is a Cache Hit event that took place.

The CMU and the CSU components that have been implemented, provide the complete functionality of the CacheCast network support. The CMU and CSU both are able to handle invalidation of the cache automatically. For inconsistency there is no retransmission mechanism, the tables are used as storage places in both of these modules, these tables are arrays of structures. In CMU the payload id and the source IP are stored and in the CSU table the payload size is stored and to store a particular payload size whose value is greater than the slot size they are stored in multiple slots using chaining mechanism.

VIII. Network implementation (Oslo version)

The network support in CacheCast consists of two units; the Cache Management Unit (CMU) and the Cache Store Unit (CSU). This division is also present in this explanation of the network support implementation in ns-3. The units are explained in the following two sections.

1) Cache Management Unit (CMU): The general design of the CMU is discussed in section II-B1. The implementation in ns-3 follows this design closely. The main CMU implementation is done in the HandlePacket() function in the CacheManagementUnit class. This code is shown in listing 8. The HandlePacket() function is called by the CacheCastNetDevice::TransmitStart() on the router node, just like the CacheCastServerUnit::HandlePacket () function, as explained in section VI-B.

Listing 8. HandlePacket() function in CacheManagementUnit

```
bool CacheManagementUnit::HandlePacket(Ptr<Packet> p)
{
  CacheCastTag tag;
  p->PeekPacketTag (tag);

  /* Check if there are enough slots */
  uint32_t slotsCount = (tag.GetPayloadSize() != 0) ?
    (tag.GetPayloadSize() - 1) / m_slotSize + 1 : 1;
  NS_ASSERT_MSG (slotsCount <= m_size,
    "CacheCast packet is too large for the CSU");

  /* Get IPv4 address of packet */
  Ipv4Header ipHdr;
  p->PeekHeader (ipHdr);
  uint32_t addr = ipHdr.GetSource ().Get ();

  /* Construct universally unique id */
  uint64_t id = ((uint64_t) addr << 32) |
    tag.GetPayloadId ();

  /* Search for id in table */
  std::map<uint64_t, uint32_t>::iterator it;
  it = m_tableIdToIndex.find (id);

  CacheCastHeader cch (tag.GetPayloadId (),
    tag.GetPayloadSize (), 0);
  bool timeout = false;

  /* Cache hit */
  if (it != m_tableIdToIndex.end ()) {
    /* Check if element it too old */
    TableItem &item =
      m_tableIndexToItem[(*it).second];
    if (Simulator::Now ().GetSeconds () -
      item.timeStamp > 1.0) {
      m_tableIdToIndex.erase (item.id);
      timeout = true;
    } else {
      p->RemoveAtEnd (tag.GetPayloadSize ());
      cch.SetPayloadSize (0);
      cch.SetIndex ((*it).second);
    }
  }

  /* Cache miss */
  if (it == m_tableIdToIndex.end () || timeout) {
    uint32_t index = m_currIndex;
    cch.SetIndex (index);

    for (int i = 0; i < slotsCount; i++) {
      TableItem &item =
        m_tableIndexToItem [m_currIndex];

      if (item.idInSlot) {
        m_tableIndexToItem[m_currIndex].idInSlot =
          false;
        m_tableIdToIndex.erase (item.id);
      }

      m_currIndex = (m_currIndex + 1) % m_size;
    }

    m_tableIndexToItem[index].id = id;
    m_tableIndexToItem[index].idInSlot = true;
    m_tableIndexToItem[index].timeStamp =
      Simulator::Now ().GetSeconds ();
    m_tableIdToIndex[id] = index;
  }

  p->AddHeader (cch);
  return true;
}
```

Now we take a look into the important parts of the above code. First in CacheManagementUnit::HandlePacket() we check that the packet will fit in the cache. The reason for doing an assert on the invariant `slotsCount <= m_size` is that most probably this is due to an error in the simulation setup. After constructing a globally unique id for the packet we search for this id in the table. If it is present, we check the time stamp for this id in order to invalidate this entry if more than 1 second has passed (see evaluation in section IX-C). If less than 1 second had passed we remove the payload from the packet and modify the CacheCast header. Next if the id was not found in the table we create enough room for the payload in the table and add the new packet info. The CacheCast header is modified with the correct table index. At last we add the CacheCast header to the packet and return from the function.

The code for the CMU follows closely the code from Srebrny [1] written for the ns-2 network simulator. The design of the code is generally the same but the syntax and some logic has been adapted to the ns-3 framework.

Since we are working in a simulator our implementation ignores the payload content of the packets. We only need to care about the payload size. This means that no copying of payload will be done in the CSU, only the size of the packet is stored, together with the payload ID. But to model reality as precisely as possible, our implementation does take into account the size and slot size of the cache, by reserving several slots if necessary. Thus, the total number of packets which fit in the cache depends on both the number of elements in the cache and the slot size.

A. Cache Store Unit (CSU)

According to the CacheCast design, CacheStoreUnit is responsible for caching the payload from the packet that is received. It is attached on the link exit. As mentioned earlier in section II in this document, packets arrives at CSU in a structure of a packet train.

In our implementation CacheCastNetDevice receives the packet in CacheCastNetDevice::Receive() function. After determining that it is a CacheCastPacket that has arrived, the packet is handled by the CacheStoreUnit::HandlePacket().

As in the CacheManagementUnit we make an assertion on the slots count and the size of the cache.

When the packet train arrives at CSU, it knows that only the first header of a packet train is followed by payload data. For every CacheCast packet the CSU checks the size of the payload to determine where the packet is complete or is it just a header. If payload size of the packet is 0, we know that only header has arrived and adds payload size number of bytes to the packet. On arrival of a complete packet, we know that this packet is not stored in the cache and stores the payload size from the incoming packet.

Cache storage is represented as a std::map in our implementation. It used index value from the CacheCastHeader as key value. Mapped value in this map is a struct containing a unique id and size of payload (which is representing the payload is our simulation). The unique payload id in mapped struct is a combination of the packet IP address and the payload ID. The implementation of Handlepacket() in CacheStoreUnit is show in listing 9

Listing 9. Implementation of Handlepacket() in CacheStoreUnit

```
bool CacheStoreUnit::HandlePacket (Ptr<Packet> p) {
  CacheCastHeader cch;
  p->RemoveHeader(cch);

  /* Check if there are enough slots */
  uint32_t slotsCount = (cch.GetPayloadSize () != 0) ?
    (cch.GetPayloadSize () - 1) / m_slotSize + 1 : 1;
  NS_ASSERT_MSG (slotsCount <= m_size,
    "CacheCast packet is too large for the CSU");
  NS_ASSERT_MSG (cch.GetIndex() < m_size,
    "CacheCast index is too large");

  /* Get IP address of packet */
  Ipv4Header ipHdr;
  uint32_t ipRead = p->PeekHeader (ipHdr);
  NS_ASSERT (ipRead);
  uint32_t addr = ipHdr.GetSource ().Get ();

  /* Construct universally unique id */
  uint64_t id = ((uint64_t) addr << 32)
    | cch.GetPayloadId();

  /* Only header arrived */
  if(cch.GetPayloadSize() == 0) {
    TableItem &item = m_cache[cch.GetIndex()];
    if(item.id != id){
      return false;
    }
    p->AddPaddingAtEnd(item.payloadSize);
  }
  /* Full CacheCast packet arrived */
  else {
    m_cache[cch.GetIndex()].id = id;
    m_cache[cch.GetIndex()].payloadSize
     = cch.GetPayloadSize();
  }
  return true;
}
```

IX. Evaluation

We have in the previous sections looked into both the general design of CacheCast and how the system is implemented in the ns-3 network simulator. In this section will will evaluate our implementation to make sure that it is functioning correctly. First we test the server part and the network part of the implementation. Then we construct integration tests to evaluate the network implementation together with the server support to make sure that the two parts of the CacheCast module integrates well with each other.

Before we start with our test cases we introduce a useful feature when doing evaluations. In ns-3 there is a concept called trace sources. A trace source is basically a list of function pointers. At some point in the code the trace source fires, which means that all connected functions are called. A script writer is able to connect to different trace sources to get status information during the simulation. In the CacheCast module there are currently four trace sources called *CcPreSend*, *CcPostSend*, *CcPreRecv* and *CcPostRecv*, which fires before the CMU, after the CMU, before the CSU and after the CSU, respectively.

A. Server support evaluation

The CacheCast design requires server support to handle client batching and to send packets onto the link in a tight sequential order in a structure of a packet train. This is tested by setting up a topology with two nodes. The node n0 and n2 in figure 3 is used for this evaluation. In this topology, dataRate value for the server node is set to 2Mbps and delay is set to be 2ms in the channel between server node (n0) and node (n2). Payload size of packet set to 1000, in addition to header size which is set to 40.

By simulating our simulation we see following result at receiver unit of node.

Listing 10. Packets received on node

```
Time    Size
2.00616 1040
2.00632 40
2.00648 40
2.00664 40
```

As we can see, the first packet arrives with expected delay and is complete. And following packets are arriving as a packet train with with a short timeline and are in size of the header size. Thus, we see that the server puts the CacheCast packets onto the link according to the CacheCast design. It follows the packet train structure.

B. Network support evaluation

This is an isolated evaluation of the CMU and the CSU.

1) Storage Space: The storage used in both of these modules are the tables created by us both of them are arrays of structures which are discussed above in the implementation parts.

2) Computational Complexity: The computational complexity everywhere in both the CMU and CSU is of O(1). Only there is searching function in the CMU whose computational complexity is of order O(n).

3) Description of test case: We have a test case name as cachecast-example3. This example has two servers which access the same router and both apply the same payload ids to their associated cacheable packets and they both are stored in two different locations (indexes) at CMU and CSU and this solves our problem of having multiple servers assigning the same payload ids to their respective payloads and this works exactly the way it should. We have another test case named as

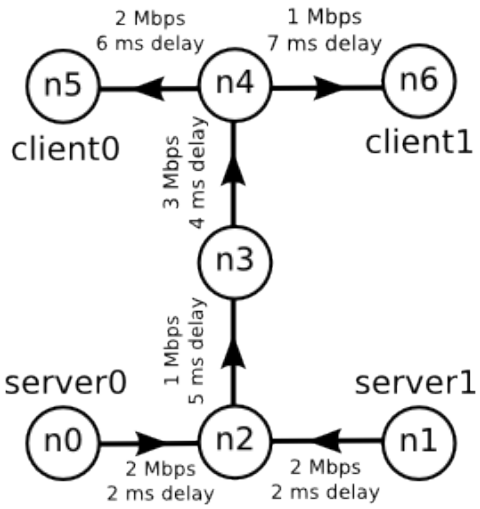

Figure 3. The network testbed topology

cachecast-example2. In this test case there is a single server and single router and the CSU part and CMU part both work exactly the way they should in this example as well.

C. CacheCast module evaluation

In the previous two sections we evaluated the server support and the network support. In this section we evaluate the integration of these two parts into the CacheCast ns-3 module. The Oslo version of the network support is used in this evaluation.

We have built a testbed in which we do our tests. The topology of this simulated network can be seen in figure 3. All the nodes in the topology supports CacheCast. The nodes n0 and n1 have server support installed. The channels have a CMU installed on the link entry and a CSU installed at the link exit, as explained in the CacheCast design. The nodes n5 and n6 have clients installed. All other nodes serve as routers.

In the server evaluation we have seen that the node adjacent to the server receives the CacheCast packets in the correct format. Now we need to test that the CMU and the CSU handles the packets from the server correctly. Our most simple approach to this is to only enable server0 and client0. Then we only have a single list of nodes in which the start node is the server and the end node is the client. To prove that the CacheCast packet handling is correct in this simple scenario we connect, for each node, to the four trace sources explained above, and prints the time and the packet size. An example of this output for node 3 is presented in listing 11. Each line in the following listings correspond to one packet being processed. Three packets are sent to client0 and we use a packet of 1000 bytes. The UDP header is 8 bytes, the IP header is 20 bytes, the CacheCast header is 10 bytes and the point-to-point header is 2 bytes, which adds up to 40 bytes of total header size.

Listing 11. Intercepted packets

```
Time (sec)  Packet size
1.01948     1040    \
1.0198      40       }  CcPreRecv (before CSU)
1.02012     40      /

1.01948     1030    \
1.0198      1030     }  CcPostRecv (after CSU)
1.02012     1030    /

1.01948     1030    \
1.02225     1030     }  CcPreSend (before CMU)
1.02236     1030    /

1.01948     1040    \
1.02225     40       }  CcPostSend (after CMU)
1.02236     40      /
```

We see in listing 11 that just before the CSU handles the packets we have the packet train structure. When the packets are processed on the node the CacheCast header (which is 10 bytes long) is removed and the packets are handled as normal IP packets. Then we see that the familiar packet train structure is again formed by the CMU. We observed that the packets, at the end, were correctly received on the client side. This test proves that the CacheCast technique does work in simple scenarios like this. But the reality is not as simple as this, so lets add a bit more complexity to the picture.

Now we enable the second server and client. This introduces (at least) two more potential points where implementation errors might be revealed; node 2, where two server channels join into one channel, and node 4 where the joint channel is split. Since the CMU identifies the CacheCast packet based on both the IP address and the payload ID the introduction of another server should not lead to other errors. The two servers send packets with a different size in order to differentiate them. Server0 sends three packets to client0 and two packets to client1, while server1 sends one packet to client0 and three packets to client1. A similar output as in the previous test can be seen in listing 12. The output is from node 3.

Listing 12. Intercepted packets with two servers

```
Time (sec)  Size    Source     Dest
1.00576     930     10.1.2.1   10.1.5.2
1.01328     930     10.1.2.1   10.1.6.2
1.0136      930     10.1.2.1   10.1.6.2
1.01392     1030    10.1.1.1   10.1.5.2
1.02224     930     10.1.2.1   10.1.6.2
1.02256     1030    10.1.1.1   10.1.5.2
1.02288     1030    10.1.1.1   10.1.5.2
1.0232      1030    10.1.1.1   10.1.6.2
1.02352     1030    10.1.1.1   10.1.6.2

After CMU
1.00576     940     10.1.2.1   10.1.5.2
1.01328     40      10.1.2.1   10.1.6.2
1.0136      40      10.1.2.1   10.1.6.2
1.01392     1040    10.1.1.1   10.1.5.2
1.02224     40      10.1.2.1   10.1.6.2
1.02256     40      10.1.1.1   10.1.5.2
1.02288     40      10.1.1.1   10.1.5.2
1.0232      40      10.1.1.1   10.1.6.2
1.02352     40      10.1.1.1   10.1.6.2
```

From the output we see that the packets originating from the two servers are mixed together when sent onto the channel after node 3. This is what one might expect. We can see that at this node the correct packets are truncated and the test showed that all the nodes handled the packets in the correct way. The clients received the correct number of packets with the correct payload attached.

Next we change the transmission rate on the channel between node2 and node3 to 10Kbps. We do this in order to test the invalidation of payloads IDs which have been present in the cache for more than one second. Then we get the output shown in listing 13, which is the packets traversing the link between n3 and n4.

Listing 13. Intercepted packets with slow link

```
Time (sec)  Size   Source    Dest
1.76276     940    10.1.2.1  10.1.5.2
1.79476     40     10.1.2.1  10.1.6.2
1.82676     40     10.1.2.1  10.1.6.2
2.65876     1040   10.1.1.1  10.1.5.2
3.41076     940    10.1.2.1  10.1.6.2
4.24276     1040   10.1.1.1  10.1.5.2
4.27476     40     10.1.1.1  10.1.5.2
4.30676     40     10.1.1.1  10.1.6.2
4.33876     40     10.1.1.1  10.1.6.2
```

In this output we can see that the fifth and sixth packet from the top is not truncated. This is because more than one second has passed since the payload ID was cached.

In all the preceding tests the table/cache size and slot size have been large enough to hold all the packets. Now we change the size of the table and cache to 10 with a slot size of 100 bytes, and resets the previous changes to the transmission rate. Introducing these changes tests two things; the handling of multiple slots per payload and the cache replacement mechanism. For the issue of multiple slots we compare the indices in which the payload should be stored, with the actual payload sizes of the packets and observe that the correct slots are in fact used. For the second issue we create a new output like in the previous tests. What we observe is that, besides that time stamps, the output is equal to the one in listing 13. This seems very reasonable since the cache and table with the current size has only room for one packet (by using the aforementioned packet sizes). So each time a packet with a different global id than the previous one enters the CMU it has to be cached.

We finish this evaluation with a test concerning packet loss. Packet loss within a router does not affect CacheCast since CacheCast packets are handled like normal IP packets within routers. Thus we do not need to test for this explicitly. The only place where packet loss affects the CacheCast technique on the link. And there we have two types of packets which might get lost; truncated packet and non-truncated packets. We deal with the truncated packets first. A truncated packet does not carry any payload, thus no CSU state should be changed when a truncated packet is received. Hence, if a truncated packet is lost on the channel, it should not affect the CSU. So we construct a simple logic to remove a truncated packet from a link and observe the consequences. Our observations shows what we expected. No harm is done to the CSU because of the packet loss. Next we change our logic to remove a non-truncated packet. What we observe now is that when a non-truncated packet is lost on the channel, the whole packet train following it is also lost. As discussed by Srebrny [1], this is a consequence of the CacheCast technique. When the packet carrying payload is lost, an inconsistency between the CMU's table and the CSU's cache is introduced. Therefore we must discard the other truncated packets with the same global id. Our tests show that our implementation handles, in the correct way, all packet loss which might affect the CacheCast technique.

X. Conclusion and future work

In this paper we have presented the implementation of the CacheCast system in the ns-3 network simulator. Through our work we have studied both the CacheCast system design and the ns-3 framework in order to introduce the CacheCast technique into ns-3. The main result of this work is a ns-3 module called `cachecast`. The module consists of the components explained throughout this paper. We have constructed several tests and evaluations to check that our implementation works correctly. The network support part have two different versions in this paper and to the respective authors' understanding, this part follows the design of CacheCast and handles the packets in the correct way. The server support has been thoroughly tested both through unit tests and through the integration tests and no severe bugs has been found. The integration tests also showed that the different components of the implementation work together in the way specified in the CacheCast design.

Several example scripts, applications, tests and demos have been made which is found in the examples directory of the module. Some scripts test only certain parts of the implementation. Information regarding this can be found in the source of each script.

The current implementation of CacheCast in ns-3 contains the features explained throughout this paper. At last we explain some of the work that might be done to the CacheCast module in the future.

1) The implementation of CacheCast in Linux supports both the UDP and DCCP transport protocols. In ns-3 there is currently no implementation of the DCCP protocol. If DCCP support is added to ns-3 in the future, the CacheCast module could be adapted to support this transport protocol.

2) Our implementation uses the PointToPointNetDevice as a basis for the CacheCastNetDevice. The PointToPointNetDevice does not model an ethernet link, just a generic point-to-point link. To further enhance the realism in the CacheCast module, the CacheCastNetDevice should support ethernet links. Version 3.13 of ns-3 (which we rely on) does not have support for switched ethernet, but this support is currently being implemented in ns-3. Hence, support for ethernet links may be easily added to the CacheCast modules in the future.

3) Lastly, there is one issue which is not handled by current implementations of CacheCast. This issue arises, most probably, when only one packet fits in the cache. First a CacheCast packet is sent and cached on the CSU. Then no packets are sent for 1 second. Then a new packet train is sent with the same payload ID and index as the previous packet. Then the first packet in the packet train is lost on the link. Thus, when the rest of the packets are processed by the CSU, the wrong payload is added to the packets. This issue might be easily solved by storing a time stamp together with the payload ID in the CSU.

ACKNOWLEDGMENT

The authors would like to thank their always positive and helpful supervisor Piotr Srebrny for useful guidelines and input in time of need.

REFERENCES

[1] P. Srebrny, "Cachecast: a system for efficient single source multiple destination data transfer," Ph.D. dissertation, University of Oslo, 2011.
[2] "The ns-3 network simulator. [ONLINE]," http://www.nsnam.org/.

CONTRIBUTIONS

Bekzhan Kassymbekov	**Report:** - Network implementation (Lancaster version) - Network support evaluation **Implementation:** - CacheManagementUnit (Lancaster version) - CacheStoreUnit (Lancaster version)
Dag Henning Liodden Sørbø	**Report:** - The CacheCast system - General implementation details *Server support implementation:* - Underlying packet handling mechanism *Network support (Oslo version):* - CMU - CacheCast module evaluation - Conclusions and future work **Implementation:** - CacheCastHeader - CacheCastNetDevice - CacheCastTag - CacheManagementUnit (Oslo version) - CacheCastServerUnit - CacheCast helpers - CacheCast examples
Kanat Sarsekeyev	**Report:** - Network implementation (Lancaster version) - Network support evaluation **Implementation:** - CacheManagementUnit (Lancaster version) - CacheStoreUnit (Lancaster version)
Rizwan Ali Ahmed	**Report:** - The ns-3 network simulator - Application programming interface (API) *Server support implementation:* - Details of the API implementation *Network support (Oslo version):* - CSU - Server support evaluation **Implementation:** - CacheCast container - Msend() function - CacheStoreUnit (Oslo version)